Tesis Doctoral

Técnicas de caching de intersecciones en motores de búsqueda

Tolosa, Gabriel Hernán

2016-09-19

Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Técnicas de caching de intersecciones en motores de búsqueda

Tesis presentada para optar al título de
Doctor de la Universidad de Buenos Aires
en el área Ciencias de la Computación

Gabriel Hernán Tolosa

Director de tesis: Dr. Esteban Feuerstein
Consejero de estudios: Dr. Esteban Feuerstein

Fecha de defensa: 19 de septiembre de 2016

Buenos Aires, 2016

Técnicas de caching de intersecciones en motores de búsqueda

Resumen

Los motores de búsqueda procesan enormes cantidades de datos (páginas web) para construir estructuras sofisticadas que soportan la búsqueda. La cantidad cada vez mayor de usuarios e información disponible en la web impone retos de rendimiento y el procesamiento de las consultas (*queries*) consume una cantidad significativa de recursos computacionales. En este contexto, se requieren muchas técnicas de optimización para manejar una alta carga de consultas. Uno de los mecanismos más importantes para abordar este tipo de problemas es el uso de cachés (*caching*), que básicamente consiste en mantener en memoria items utilizados previamente, en base a sus patrones de frecuencia, tiempo de aparición o costo.

La arquitectura típica de un motor de búsqueda está compuesta por un nodo *front-end* (*broker*) que proporciona la interfaz de usuario y un conjunto (grande) de nodos de búsqueda que almacenan los datos y soportan las búsquedas. En este contexto, el almacenamiento en caché se puede implementar en varios niveles. A nivel del *broker* se mantiene generalmente un caché de resultados (*results caché*). Éste almacena la lista final de los resultados correspondientes a las consultas más frecuentes o recientes. Además, un caché de listas de posteo (*posting list caché*) se implementa en cada nodo de búsqueda. Este caché almacena en memoria las listas de posteo de términos populares o valiosos, para evitar el acceso al disco. Complementariamente, se puede implementar un caché de interscciones (*intersection caché*) para obtener mejoras de rendimiento adicionales. En este caso, se intenta explotar aquellos pares de términos que ocurren con mayor frecuencia guardando en la memoria del nodo de búsqueda el resultado de la intersección de las correspondientes listas invertidas de los términos (ahorrando no sólo tiempo de acceso a disco, sino también tiempo de CPU).

Todos los tipos de caché se gestionan mediante una "política de reemplazo", la cual decide si va a desalojar algunas entradas del caché en el caso de que un elemento requiera ser insertado o no. Esta política desaloja idealmente entradas que son poco probables de que vayan a ser utilizadas nuevamente o aquellas que se espera que proporcionen un beneficio menor.

Esta tesis se enfoca en el nivel del *Intersection caché* utilizando políticas de reemplazo que consideran el costo de los items para desalojar un elemento (*cost-aware policies).* Se diseñan, analizan y evalúan políticas de reemplazo estáticas, dinámicas e híbridas, adaptando algunos algoritmos utilizados en otros dominios a éste. Estas políticas se

combinan con diferentes estrategias de resolución de las consultas y se diseñan algunas que aprovechan la existencia del caché de intersecciones, reordenando los términos de la consulta de una manera particular.

También se explora la posibilidad de reservar todo el espacio de memoria asignada al almacenamiento en caché en los nodos de búsqueda para un nuevo caché integrado (*Integrated caché*) y se propone una versión estática que sustituye tanto al caché de listas de posting como al de intersecciones. Se diseña una estrategia de gestión específica para este caché que evita la duplicación de términos almacenados y, además, se tienen en cuenta diferentes estrategias para inicializar este nuevo *Integrated caché*.

Por último, se propone, diseña y evalúa una política de admisión para el caché de intersecciones basada en principios del Aprendizaje Automático (*Machine Learning*) que reduce al mínimo el almacenamiento de pares de términos que no proporcionan suficiente beneficio. Se lo modela como un problema de clasificación con el objetivo de identificar los pares de términos que aparecen con poca frecuencia en el flujo de consultas.

Los resultados experimentales de la aplicación de todos los métodos propuestos utilizando conjuntos de datos reales y un entorno de simulación muestran que se obtienen mejoras interesantes, incluso en este hiper-optimizado campo.

Intersection caching techniques in search engines

Abstract

Search Engines process huge amounts of data (i.e. web pages) to build sophisticated structures that support search. The ever growing amount of users and information available on the web impose performance challenges and query processing consumes a significant amount of computational resources. In this context, many optimization techniques are required to cope with heavy query traffic. One of the most important mechanisms to address this kind of problems is caching, that basically consists of keeping in memory previously used items, based on its frequency, recency or cost patterns.

A typical architecture of a search engine is composed of a front-end node (broker) that provides the user interface and a cluster of a large number of search nodes that store the data and support search. In this context, caching can be implemented at several levels. At broker level, a *results cache* is generally maintained. This stores the final list of results corresponding to the most frequent or recent queries. A *posting list cache* is implemented at each node that simply stores in a memory buffer the posting lists of popular or valuable terms, to avoid accessing disk. Complementarily, an *intersection cache* may be implemented to obtain additional performance gains. This cache attempts to exploit frequently occurring pairs of terms by keeping the results of intersecting the corresponding inverted lists in the memory of the search node (saving not only disk access time but CPU time as well).

All cache types are managed using a so-called "replacement policy" that decides whether to evict some entry from the cache in the case of a cache miss or not. This policy ideally evicts entries that are unlikely to be a hit or those that are expected to provide less benefit.

In this thesis we focus on the *Intersection Cache* level using cost-aware caching policies, that is, those policies that take into account the cost of the queries to evict an item from cache. We carefully design, analyze and evaluate static, dynamic and hybrid cost-aware eviction policies adapting some cost-aware policies from other domains to ours. These policies are combined with different query-evaluation strategies, some of them originally designed to take benefit of the existence of an intersection cache by reordering the query terms in a particular way.

We also explore the possibility of reserving the whole memory space allocated to caching at search nodes to a new integrated cache and propose a static cache (namely Integrated Cache) that replaces both list and intersection caches. We design a specific

cache management strategy that avoids the duplication of cached terms and we also consider different strategies to populate the integrated cache.

Finally, we design and evaluate an admission policy for the Intersection Cache based in Machine Learning principles that minimize caching pair of terms that do not provide enough advantage. We model this as a classification problem with the aim of identifying those pairs of terms that appear infrequently in the query stream.

The experimental results from applying all the proposed approaches using real datasets on a simulation framework show that we obtain useful improvements over the baseline even in this already hyper-optimized field.

Agradecimientos

Es un poco difícil nombrar a todas las personas que me ayudaron a alcanzar este momento desde mis primeros pasos en la vida académica. Hay una multitud de gente a la que agradezco, algunos fueron profesores o tutores en algún punto en mi vida. La oportunidad de alcanzar un doctorado llegó relativamente tarde en mi vida, y tiene sentido entonces centrarme aquí en la gente que me ayudó a superar esta experiencia durante este último período.

Primero que todo, me gustaría expresar mi mas profundo agradecimiento y gratitud a mi director, Prof. Dr. Esteban Feuerstein por su invaluable apoyo, sugerencias, guía y paciencia durante este camino. Esteban me dio la oportunidad de comenzar el doctorado bajo su guía casi sin conocerme, después de solamente una reunión. Compartimos discusiones muy fructíferas que dispararon un montón de ideas útiles, muchas de las cuales están contenidas en este trabajo. Gracias Esteban por todo esto!

Durante estos años tuve la oportunidad de pasar algún tiempo en dos instituciones de alta calidad. Primero, realicé una estancia corta en los laboratorios de investigación de Yahoo! en Barcelona, bajo la supervisión de Ricardo Baeza-Yates. Tuve la oportunidad de conocer un lugar asombroso que lidera la investigación en temas relacionados a mi trabajo. En segundo lugar, estuve por algún tiempo en el Dipartimento di Ingegneria Informatica, Automatica e Gestionale de la Universidad de Roma, La Sapienza bajo la supervisión de Alberto Marchetti-Spaccamela y Luca Bechetti. El trabajo desarrollado allí fue la base de un capítulo en mi tesis. Pasé gratos momentos tanto en Barcelona como en Roma. Gracias Ricardo, Alberto y Luca, aprecio mucho vuestra ayuda allí.

Por otra parte, agradezco a los miembros del comité de evaluación, Gonzalo Navarro, Min Chih Lin y Fabrizio Silvestri, por dedicar tiempo y esfuerzo a leer y comentar mi tesis. Este trabajo ha sido mejorado por sus comentarios y sugerencias útiles. Estoy muy honrado de tener su respaldo.

También agradezco a las Universidades de Luján (UNLu) y Buenos Aires (UBA). Ambas instituciones pertenecen al sistema de educación pública de la Argentina y me

dieron la inestimable oportunidad de alcanzar grados universitarios. También quiero agradecer a todos mis colegas cercanos y estudiantes, por muchas discusiones y sugerencias de investigación.

Por último, pero no menos importante, agradezco a toda mi familia por su apoyo constante, y especialmente agradezco a mi novia Vanina, por toda su ayuda y apoyo durante los últimos años. Ella esperó pacientemente por mí cuando yo estaba demasiado absorto en mi trabajo, buceando en las profundidades del código en el que estaba trabajando o realizando experimentos, incluso durante muchos fines de semana.

Contents

Abstract i

Contents vii

List of Figures x

List of Tables xiv

1 Introduction 1
- 1.1 Caching in Search Engines . . . 4
- 1.2 Goals of this Work . . . 7
- 1.3 Contributions . . . 8
  - 1.3.1 Limitations . . . 9
- 1.4 Publications . . . 10
- 1.5 Organization . . . 11

2 Background and Related Work 15
- 2.1 Search Engines Architecture . . . 15
- 2.2 Data Structures . . . 15
  - 2.2.1 Index Partitioning . . . 18
    - 2.2.1.1 Document-based Partitioning . . . 18
    - 2.2.1.2 Term-based Partitioning . . . 18
    - 2.2.1.3 Hybrid Partitioning . . . 19
  - 2.2.2 Multi-tier Indexes . . . 19
  - 2.2.3 Posting List Compression . . . 20
- 2.3 Query Processing . . . 21
  - 2.3.1 Query Evaluation Strategies . . . 22
    - 2.3.1.1 Document-at-a-Time . . . 22
    - 2.3.1.2 Term-at-a-Time . . . 23
    - 2.3.1.3 Score-at-a-Time . . . 23
- 2.4 Caching in Search Engines . . . 24
  - 2.4.1 Result Caching . . . 24
  - 2.4.2 List Caching . . . 25
  - 2.4.3 Multi-level Caching . . . 26

3 Caching Policies and Processing Strategies for Intersection Caching 30
- 3.1 Intersection Caching Policies . . . 31
  - 3.1.1 Static Policies . . . 32

3.1.2 Dynamic Policies . . . 32
3.1.3 Hybrid Policies . . . 34
3.2 Query Processing using an Intersection Cache . . . 34
3.2.1 Another strategy using all possible cached information . . . 36
3.3 Experimentation Framework . . . 37
3.3.1 Document Collections . . . 39
3.3.2 Query Logs . . . 40

4 Cost-Aware Intersection Caching **44**
4.1 Description of Intersection Costs . . . 45
4.1.1 Data and Setup . . . 47
4.2 Intersection Caching with the Index in Disk . . . 47
4.2.1 Static Policies . . . 47
4.2.2 Dynamic Policies . . . 51
4.2.3 Hybrid Policies . . . 55
4.2.4 Comparing Variances . . . 59
4.3 Intersection Caching with the Index in Main Memory . . . 62
4.3.1 Static Policies . . . 62
4.3.2 Dynamic Policies . . . 66
4.3.3 Hybrid Policies . . . 70
4.3.4 Comparing Variances . . . 74
4.4 Summary . . . 76

5 Integrated Cache **79**
5.1 Proposal . . . 81
5.1.1 Compression of the Integrated Cache . . . 84
5.2 Selecting Term Pairs . . . 88
5.2.1 Greedy Methods . . . 89
5.2.2 Term Pairing as a Matching Problem . . . 89
5.3 Experiments and Results . . . 90
5.3.1 Data and Setup . . . 90
5.3.2 Integrated Cache with Raw Data . . . 91
5.3.3 Integrated Cache with Compressed Data . . . 94
5.3.4 Integrated Cache and Result Cache . . . 95
5.4 Summary . . . 98

6 Machine Learning Based Access Policies for Intersection Caches **101**
6.1 Brief Introduction to Query Log Mining . . . 103
6.2 Dynamics of Pairs of Terms in a Query Stream . . . 104
6.3 Machine Learning based Admission Policy . . . 107
6.3.1 Cumulative-Frequency Control Algorithm . . . 110
6.4 Experiments and Results . . . 112
6.4.1 Data and Setup . . . 112
6.4.2 Baseline . . . 114
6.4.3 Admission Policy using DT and RF classifiers . . . 115
6.4.4 Admission Policy with CFC algorithm . . . 115
6.5 Summary . . . 124

7 Conclusions and Future Work **127**

Bibliography **133**

List of Figures

1.1 Simplified architecture of a Search Engine 5
1.2 Dependency chart between the established knowledge in the literature (dotted circles) and the contributions of this thesis in each chapter. 9

2.1 Example of the inverted index generated after indexing the documents in the example. 16

3.1 Comparison between the simulation methodology and a real system . . . 39
3.2 Frequency distributions of queries (left) and pairs of terms (right). 41
3.3 Growth of unique queries and pairs according to new instances appearing in the query stream. 41

4.1 Frequency and cost of sampled intersections (left, both axis in log scale). Cumulative Distribution Function for the frequency distribution (right, x-axis in log scale). 46
4.2 Frequency and intersection-size of sampled intersections (left, both axis in log scale). Cumulative Distribution Function for the frequency distribution (right, x-axis in log scale). 46
4.3 Total cost incurred by the S1 processing strategy for the seven static cache policies. 48
4.4 Total cost incurred by the S2 processing strategy for the seven static cache policies. 49
4.5 Total cost incurred by the S3 processing strategy for the seven static cache policies. 49
4.6 Total cost incurred by the S4 processing strategy for the seven static cache policies. 50
4.7 Comparison of total costs incurred by the two best static policies and the four processing strategy. 50
4.8 Total cost incurred by the S1 processing strategy for the seven dynamic cache policies. 51
4.9 Total cost incurred by the S2 processing strategy for the seven dynamic cache policies. 52
4.10 Total cost incurred by the S3 processing strategy for the seven dynamic cache policies. 53
4.11 Total cost incurred by the S4 processing strategy for the seven dynamic cache policies. 53
4.12 Comparison of total costs incurred by the two best dynamic policies and the four processing strategy. 54
4.13 Total cost incurred by the S4 strategy enhanced with three-term intersections (I3) for dynamic policies. 54

4.14 Total cost incurred by the S1 processing strategy for the four hybrid cache policies. . . . 55
4.15 Total cost incurred by the S2 processing strategy for the four hybrid cache policies. . . . 56
4.16 Total cost incurred by the S3 processing strategy for the four hybrid cache policies. . . . 57
4.17 Total cost incurred by the S4 processing strategy for the four hybrid cache policies. . . . 57
4.18 Comparison of total costs incurred by the two best hybrid policies and the four processing strategy. . . . 58
4.19 Total cost incurred by the S4 processing strategy enhanced with three-term intersections (I3) for hybrid policies. . . . 58
4.20 Comparison of variance for the baselines and best static policies and strategies. . . . 59
4.21 Comparison of variance for the baselines and best dynamic policies and strategies. . . . 60
4.22 Comparison of variance for the baselines and best hybrid policies and strategies. . . . 61
4.23 Total cost incurred by the S1 processing strategy for the seven static cache policies. . . . 62
4.24 Total cost incurred by the S2 processing strategy for the seven static cache policies. . . . 63
4.25 Total cost incurred by the S3 processing strategy for the seven static cache policies. . . . 64
4.26 Total cost incurred by the S4 processing strategy for the seven static cache policies. . . . 64
4.27 Comparison of total costs incurred by the two best static policies and the four processing strategy. . . . 65
4.28 Total cost incurred by the S1 processing strategy for the seven dynamic cache policies. . . . 66
4.29 Total cost incurred by the S2 processing strategy for the seven dynamic cache policies. . . . 67
4.30 Total cost incurred by the S3 processing strategy for the seven dynamic cache policies. . . . 67
4.31 Total cost incurred by the S4 processing strategy for the seven dynamic cache policies. . . . 68
4.32 Comparison of total costs incurred by the two best dynamic policies and the four processing strategy. . . . 68
4.33 Total cost incurred by the S4 processing strategy enhanced with three-term intersections (I3) for dynamic policies. . . . 69
4.34 Total cost incurred by the S1 processing strategy for the seven dynamic cache policies. . . . 70
4.35 Total cost incurred by the S2 processing strategy for the seven dynamic cache policies. . . . 71
4.36 Total cost incurred by the S3 processing strategy for the seven dynamic cache policies. . . . 71
4.37 Total cost incurred by the S4 processing strategy for the seven dynamic cache policies. . . . 72

4.38 Comparison of total costs incurred by the two best hybrid policies and the four processing strategy. . . . 72
4.39 Total cost incurred by the S4 processing strategy enhanced with three-term intersections (I3) for dynamic policies. . . . 73
4.40 Comparison of variance for the baselines and best static policies and strategies. . . . 74
4.41 Comparison of variance for the baselines and best dynamic policies and strategies. . . . 75
4.42 Comparison of variance for the baselines and best hybrid policies and strategies. . . . 75

5.1 Simplified architecture of a Search Engine with the proposed *Integrated Cache* at search node level. . . . 80
5.2 Separated Union data structure proposed by Lam et al. [64]. . . . 80
5.3 Data Structure used for the *Integrated Cache*. . . . 82
5.4 Example of a redirection from entry 3 (Θ) to entry 1 that corresponds to term t_1 (we omit entry 2 for clarity). . . . 83
5.5 DGap value distributions for the Integrated and standard Posting Lists. . 86
5.6 Integrated cache compression performance: δ_x ratio scatter plot between lists vs integrated representations. x-axis in log scale to get a clearer view. 87
5.7 Integrated cache compression performance: Efficiency ratio (the lower, the better) between lists vs integrated representations. x-axis in log scale to get a clearer view. . . . 88
5.8 Performance of the different approaches using the AOL-1 query set and the UK collection (y-axis in log scale to get a clearer view). . . . 91
5.9 Performance of the different approaches using the AOL-1 query set and the WB collection. . . . 92
5.10 Improvements obtained by the Integrated Cache vs the baseline (List Cache). 92
5.11 Performance of the PfBT-mwm approach using the AOL-2 query set and the UK collection. . . . 93
5.12 Performance of the PfBT-mwm approach using the AOL-2 query set and the WB collection. . . . 93
5.13 Performance of the different approaches using the AOL-1 query set. . . . 94
5.14 Performance of the different approaches using the AOL-2 query set. . . . 95
5.15 Performance of the different approaches using the the AOL-1 query set and RC 250k entries. . . . 96
5.16 Performance of the different approaches using the AOL-1 query set and RC 500k entries. . . . 97

6.1 Number of singleton and non-singleton intersections in the query stream. In both cases, cumulative frequencies show a linear growth proportional to $f(x) = 0.49x + 674,826$ and $f(x) = 0.30x + 248,083$ respectively. . . . 105
6.2 Evolution of the proportion of singleton pairs in the query stream. . . . 105
6.3 Sum of the lengths of the posting lists of both terms for WB and WS document collections (top). Impact (proportion) of singletons on the total amount of data (bottom). . . . 106
6.4 Conversion rate from singleton to non-singletons through bins. . . . 107
6.5 Example of a decision tree generated using the four aforementioned features (WB Collection, $F_T = 1$) . . . 113

6.6 Performance comparison between the baseline (*noap*) and the *clairvoyant* algorithm with different thresholds ($clair - F_T \leq x$) for the two document collections. (x-axis in log scale to get a clearer view) 116
6.7 Performance of the algorithms using the Random Forest classifier for both collections with respect to the lower (*noap*) and upper ($clair - F_T \leq 14$) bounds (x-axis in log scale) 117
6.8 Performance of the algorithms using the Decision Tree classifier for both collections with respect to the lower (*noap*) and upper ($clair - F_T \leq 14$) bounds (x-axis in log scale) 118
6.9 Performance of the algorithms using the Decision Tree classifier with CFC (checking presence/absence). x-axis in log scale. 120
6.10 Performance of the algorithms using the Decision Tree classifier with CFC (checking frequency condition). x-axis in log scale. 121
6.11 Performance comparison of the best configuration of each algorithm. x-axis in log scale. 122

List of Tables

1.1 Main differences among the three considered cache levels in a typical SE architecture. 6

3.1 Notation used to describe the caching policies. 31
3.2 Collection Statistics. "Raw Size" and "Index Size" correspond to the uncompressed documents in HTML format and the resulting (compressed) index respectively. 39
3.3 Number of total and unique queries and pairs in the query log. 41

5.1 Sum of posting lists lengths for the uncompressed representation. 87
5.2 Sum of posting lists lengths for the compressed representation. 88

6.1 Confusion Matrix of the Decision Tree classifier for $F_T = 1$. Each cell corresponds to the proportion of classified instances. 110
6.2 Results summary for the WB Collection. 123
6.3 Results summary for the WS Collection. 123

Chapter 1

Introduction

Information Retrieval (IR) is the field of research in Computer Science involved in storing, organizing, and searching of relevant information in document collections [8]. In the early days, these collections (also known as 'corpus') were composed of a rather limited number of documents, in the order of thousands, with textual information only. This scenery changed dramatically with the massification of the Internet and the World Wide Web (shortened, the web). In the last years, we have witnessed the exponential growth of the number and complexity of the documents in the web, which has become the largest repository of information all around the world. Today, the size, dynamism and diversity of the web bring IR applications out of the boundaries of a single computer making the problem of finding information a challenge in both accuracy and efficiency.

Search has become a crucial task in many systems at different scales, for example, enterprise IR applications, vertical search engines or, the top case, web scale search engines (WSEs). In these cases, the use of sophisticated techniques for efficiency and scalability purposes is extremely necessary.

The determination of the precise size of the web is an unsolvable problem because there exist portions of it that cannot be accessed because of its dynamism or access restrictions. In 2005, the estimated size of the indexable Web was at least 11.5 billion pages [50]. Nowadays, some approximations show almost 48 billion pages[1] on the surface web (that is, the portion of the web indexed by a search engine) and roughly doubling every eight months [8], faster than Moore's law. Furthermore, the number of users that send millions of query requests each day is also increasing[2] and they not only "search" to satisfy their information needs but also to accomplish daily tasks (e.g., "organize a trip", "buy things", etc.) as well [9][53]. Under these considerations, the Web has become

[1] http://www.worldwidewebsize.com/
[2] http://www.internetlivestats.com/google-search-statistics/

complex and challenging for many practical purposes, as it is possible to generate an infinite number of pages.

It is quite clear that searching in a huge and distributed space of data poses serious challenges, mainly in terms of quality and performance. General-purpose search engines answer queries under strict performance constraints, namely, in no more than a few hundred milliseconds [104]. In order to achieve this goal, WSEs rely upon large clustered systems (that scale horizontally) built on high speed networks, located in multiple data centers. At such a scale, all the components of the architecture of a WSE must be optimized using sophisticated techniques that improve the utilization of the computing resources. In the particular case of industry-scale search engines (such as Google and Yahoo!) optimizing the efficiency is also important because this translates into financial savings for the company [57] that earn most of their revenue by embedding advertisements in the search results. For Also energy consumption is an important factor to consider. A recent study by Catena and Tonellotto [27] tries to quantify the energy consumption of a search engine to solve a single query. This work reinforces the idea that a search engine should reduce processing time because of both economical impact and environmental issues, posed by their large datacenters.

To handle this volume of data and to achieve such a high performance, the typical architecture of modern SE is composed by a *broker* that receives the queries and a cluster of *search nodes* that manages a huge number of documents [14] in parallel to obtain the most relevant results for the query. In some cases, search engines consider AND semantics for the queries, so the result is given by the intersection of lists of document identifiers where the involved terms [104] appear (a.k.a. posting lists). One of the main reasons of computing the intersection (instead of the union) is that "AND queries" produce higher precision results, so most search engines try to consider first documents containing all query terms. Besides, intersection produces potentially shorter lists which leads to smaller query latencies, and it has been empirically shown that high query latencies degrade user satisfaction [4] and may also lead to losses in revenues of a commercial search engine [98]. Current high-scale search engines uses sophisticated algorithms and techniques to also support disjunctive (OR) semantics in an efficient way. In these cases, the result page is finally reordered using advanced Machine Learning techniques (a.k.a. learning to rank [69]), that use different features to build a global relevance model for each document. Given huge collection sizes and high user expectations regarding both efficacy and efficiency, disjunctive query processing requires advanced computer architectures for performance reasons, so conjunctive semantics are often used in practical implementations.

Typically, each *search node* holds only a fraction of the document collection and maintains a local inverted index [8] that is used to obtain a high query throughput. For example, given a cluster of P search nodes and a collection of C documents and assuming that these are evenly distributed among the nodes, each one maintains an index with information related to only $\frac{C}{P}$ documents. In the case of a commercial search engine, the number of search nodes (including replicas) may reach thousands of machines [32].

Besides, to accomplish the strong performance requirements, WSEs usually implement different optimization techniques such as posting list compression [115] and pruning [73], results prefetching [57] and caching [10].

One of the most important mechanisms to address this kind of problems is caching, that basically consists of keeping previously used items in memory, based on its frequency, recency or cost patterns. Caching techniques exploit memory hierarchies with increasing response latencies or processing requirements (for example, processor registers, main memories, hard disks, or resources in a local or a wide area network).

In the early days of computing, this technique (a.k.a. the paging problem in virtual memory systems [15]) was extensively investigated, delivering many algorithms and heuristics [13][79][84]. Basically, the paging problem consists on the management of a two-level memory hierarchy. One level, the cache, has a limited space but offers fast access (for example, the main memory) while the second one is virtually infinite but offers a (much) slower access time than the first one (for example, the hard disk). Cache memory is divided in pages of a fixed size and it may store at most k of them. When a page is required, it is first requested to the cache. If that page is present, there is no cost to satisfy the requirement. Otherwise, a *page fault* occurs and the request must be served from the slow memory and loaded in the cache. The main goal is to minimize the number of page faults (cache misses) serving one [72] or many [39] pages from the cache to smooth the effect of slow access memories in the performance of the system.

A replacement algorithm is required to manage the cache contents efficiently, deciding which objects are worth to be kept in the cache. This decision is either off-line (static) or online (dynamic). A static cache is based on historical information and is periodically updated. A dynamic cache replaces entries according to the sequence of requests. As cache memories are limited in size, it is necessary to evict some items when the cache is full and there is a new entry to add. When a new request arrives, a dynamic cache system decides which entry is evicted from the cache in the case of a cache miss. Such online decisions are based on a cache policy (namely, an *eviction policy*), and several different ones have been studied in the past [92].

The most useful strategies are based on the notion of "locality of reference" [33], that is, programs tend to request items (pages) from a subset of all possible items. However, this subset is not static and may vary in time so the replacement heuristic may adapt the contents of the cache to minimize the number of cache faults. This type of problems are called *online problems* and are present in many fields of computer science.

Algorithms for online problems operate under partial information. The input is provided as a "stream" and, at each point in time, the algorithms take certain decisions (for example, which item is evicted from cache), based on the knowledge obtained from the input that they have seen so far, but without knowing the future input. Online algorithms are commonly evaluated using *competitive analysis* [101], a method to bound the performance of a given replacement policy that allows a comparison against another policy.

This method takes into account the online nature of the problem and has been extensively studied in the literature of caching algorithms. Basically, an online algorithm A is said to be c-competitive if the cost incurred by it to satisfy any sequence of requests is bounded (asymptotically) by c times the cost incurred by the optimal (OPT) algorithm for the same sequence (plus a constant b). Given an input sequence ρ, it becomes: $cost_A(\rho) \leq c \times cost_{OPT}(\rho) + b$. This kind of analysis has become a standard way to measure online algorithms and has been used by a vast number of works that include paging [113], weighted paging [114], multiprocessors caches [58] and some extensions such as a generalization of paging to the case where there are many threads of requests [40].

Since time and resources are always valuable in computer systems, all the theoretical and practical studies about paging show that caching exists everywhere and is used in all high performance systems. Certainly, the use of several cache technologies from hardware to applications, illustrates the importance of caching.

1.1 Caching in Search Engines

In the context of a search system, caching can be implemented at several levels. In a typical architecture, a WSE is organized in clusters of commodity hardware that offer the possibility to easily add/replace machines. Figure 1.1 shows a diagram of the WSE architecture we consider in this thesis, including the basic caching levels.

Usually, a *results cache* [77] is maintained at a broker level, which stores the final answer for a number of frequent queries. This type of cache has been extensively studied in the last years [88], as it produces the best performance gains, although it achieves lower

hit ratios compared with the other caching levels. This is due to the fact that each hit at this level implies very high savings in computation and communication time.

At the search nodes, the lowest level implements a *posting lists cache*, which stores the posting lists of some popular terms. This cache achieves the highest hit ratio because query terms appear individually in different queries more often than their combinations.

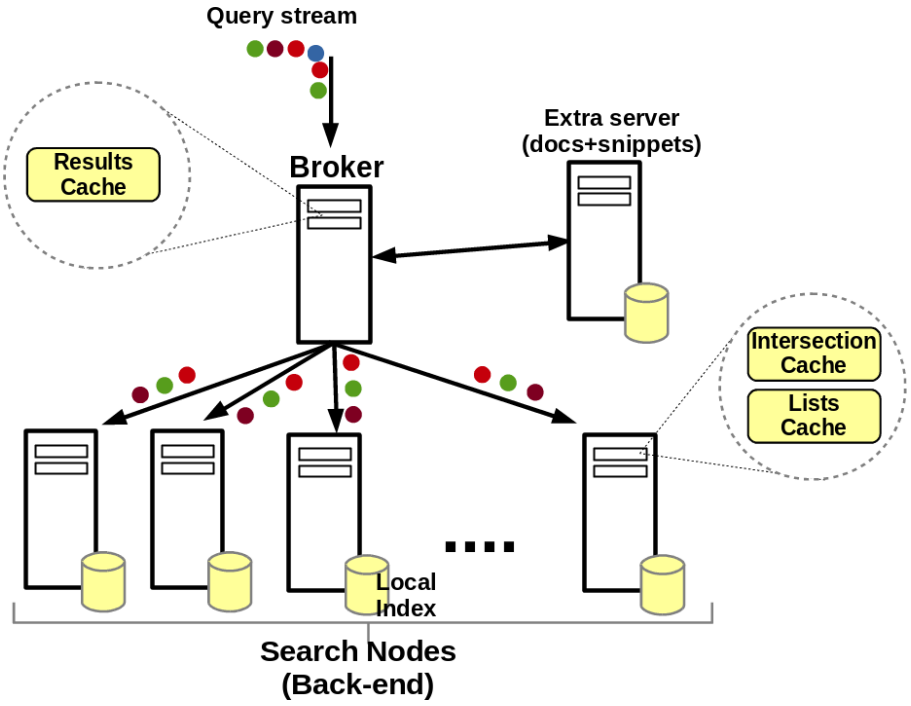

FIGURE 1.1: Simplified architecture of a Search Engine

Complementarily, an *intersections cache* [70] may be implemented to obtain additional performance gains. This cache attempts to exploit frequently occurring pairs of terms by maintaining the results of intersecting the corresponding inverted lists in memory. The idea of caching posting list intersections arises rather naturally in this architecture. However, the problem of combining different strategies to solve the queries considering the cost of computing the intersections is a rather challenging issue.

Furthermore, we know that the number of terms in the collection is finite (although big) but the number of potential intersections is virtually infinite and depends on the submitted queries. As we previously mentioned, web scale search engines only return documents that contain all of the search terms, so the cost of computing scores is dominated by the lists traversals required to compute the intersections. In these cases, caching pairwise intersections becomes a useful technique to significantly reduce this cost in subsequent queries [70].

Another important issue to note is that some high-scale search engines maintain the whole inverted index in the main memory of their search nodes. In this case, the caching of lists becomes useless, but an intersection cache may save processing time.

An interesting issue is to evaluate the interaction among different cache levels in modern hardware architectures. General caching techniques (e.g. CPU cache) are designed for all kind of application while specific ones (e.g. application-level caches) are intended for certain cases that depend on both, the problem and the nature of the data. Search systems may be deployed on different architectures with specific configurations (e.g. homogeneous/heterogeneous), so the real usefulness of the cache hierarchy (including the intersection cache) depends on that hardware architecture and the available resources. The complete understanding of this kind of interaction under different conditions is a challenging issue to be considered for future work.

One of the most common metrics used to evaluate cache policies is the *hit ratio* (i.e. the percentage of accesses that get cache hits). The highest value of this metric is bounded by the proportion of singleton items[3] in the whole stream. An important difference among the caching levels is that in some of them the size of the items is uniform (as for example in the results cache) while in the others item sizes may vary (lists or intersections caching). This implies that the value or benefit from having an item in cache depends not only on its utility (i.e. the probability or frequency of a hit) but also on the space it occupies. Moreover, there is another parameter that may be taken into account, that is the saving obtained by a hit on an item (i.e. how much time the computation of the item would require). Table 1.1 summarizes the differences among the three main cache levels.

	Result Cache	**Intersection Cache**	**List Cache**
Location	Broker	Search Nodes	Search Nodes
Potential Size	Virtually Infinite	Virtually Infinite	Up to storing the full inverted index
Items Size	Fixed (to hold the top-k results of each query)	Variable (depends on the size of each intersection)	Variable (depends on the size of each posting list)
Hit Rate	Bounded by the % of singleton queries in the stream (∼50%)	Moderate	High (Fraction of singleton terms in the total volume ∼4-10%)

TABLE 1.1: Main differences among the three considered cache levels in a typical SE architecture.

[3] A *singleton* is an item that occurs only once in a given stream. For example, a *singleton query* appears only once in the considered query set while a *singleton term* appears only once in the resulting set after decomposing each query in their individual terms.

For each type of cache, a number of item replacement policies have been developed and tested to achieve the best overall system performance. Some of the replacement policies found in the literature rely on parameters such as recency or frequency of the items (as the well-known LRU and LFU), while others take into account the gain of maintaining a particular item in the cache such as Landlord [114] or GreedyDual-Size [23].

As we previously mentioned, caching policies are commonly evaluated using the *hit ratio*. However, the processing cost of an item (an intersection between terms in our case) is not necessarily related to its popularity or recency, so it may be reasonable to use that measure to compute the *gain* of maintaining that item in the cache. Considering that cache misses have different costs and the size of the resulting lists of two-terms intersections are non-uniform we introduce these parameters in different caching policies to determine which are more useful for cost saving. To benefit from these features we explicitly incorporate the costs of the intersections into the caching policies, leading us to the main focus of this thesis.

Particularly, we are concerned with the problem of cost-aware intersection caching in search engines that complements the other caching levels (as shown in Figure 1.1). To this extent, we design and evaluate improved approaches of the Intersection Cache with the aim of reducing the query execution time. Following previous work on Result Caching [88], we propose and evaluate cost-aware cache eviction policies and design query resolution strategies that specifically take advantage of the Intersection Cache.

We also propose another cache level that may be used at search node side that *joins* both Posting Lists and Intersection caches in the same memory area using a space-efficient representation (we call this *Integrated Cache*). This cache is particularly useful when a portion of the inverted index still resides in secondary memory. Finally, we extend our cost-aware caching framework with the design of a cache admission policy using a Machine Learning approach. Few pieces of work focus on admission policies in search engine caches. The research of Baeza-Yates et al. [11] introduces a policy to prevent the storage of infrequent queries in a Result Cache, but in the context of intersection caching, this problem has not been tackled yet.

1.2 Goals of this Work

The main goal of this thesis is to advance in the field of query resolution strategies and cache data structures by designing and evaluating new approaches that exploit the existence of an Intersection Cache in a search engine. We hope that this work will enable practical implementations of these approaches that lead to a better use of the computational

resources, especially in environments where available computing power is constrained. Considering the architecture of a search engine depicted in Figure 1.1 and taking into account that a search engine is built by a huge number of search nodes [32], our aim is to model a single search node that implements the proposals in order to evaluate the performance gains.

The particular research goals in this thesis are:

- The understanding of the impact of cost-aware eviction policies in intersection caches compared to traditional hit-based policies.
- The design and evaluation of different query resolution strategies that combine the terms in different ways and obtain more cache hits using both static, dynamic and hybrid caching policies.
- The design and evaluation of a static cache that combines both posting lists and intersections using an integrated data structure.
- The study of frequency patterns of pairs of terms (namely, intersections) to define a cache access policy using a Machine Learning approach.

1.3 Contributions

In this thesis, we carefully design, analyze and evaluate caching policies that take into account the cost of the queries to implement an Intersection Cache. We can divide our contributions into three main research sub-directions. Figure 1.2 shows a dependency chart between the established knowledge in the literature and the contributions of this thesis.

In the first part, we design and implement static, dynamic and hybrid cost-aware eviction policies adapting some cost-aware policies from other domains to ours. These policies are combined with different query-evaluation strategies, some of them originally designed to take benefit from the existence of an intersection cache by reordering the query terms in a particular way. We evaluate these approaches in two scenarios: with the inverted index residing in disk or main memory.

In the second part, we explore the possibility of reserving the whole memory space allocated to caching at search nodes to a new integrated cache. More precisely, we propose a static cache (namely Integrated Cache) that replaces both list and intersection caches. We design a specific cache management strategy that avoids the duplication of cached terms and we also consider different strategies to populate the integrated cache.

Finally, we design and evaluate an admission policy for the Intersection Cache based in Machine Learning principles that minimize caching pair of terms that do not provide enough advantage. We model this as a classification problem, considering a minimal number of features, with the aim of identifying those pairs of terms that appear infrequently in the query stream. Once detected, we add this information to the query resolution strategy in order to avoid the computation and caching of intersections with very low probabilities to be requested again.

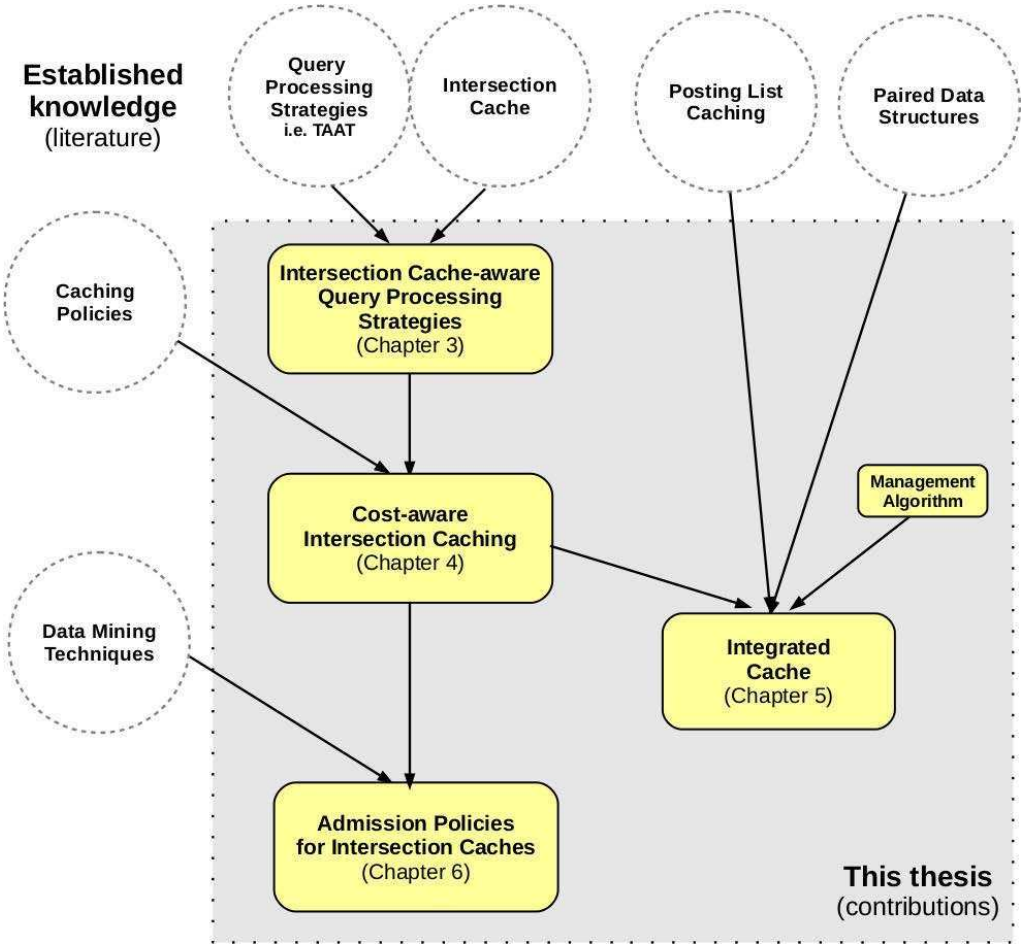

FIGURE 1.2: Dependency chart between the established knowledge in the literature (dotted circles) and the contributions of this thesis in each chapter.

1.3.1 Limitations

The results presented in this work depend on a series of assumptions:

1. We use a simulation framework that models a single node in a search cluster. We assume a uniform workload in the nodes (i.e. the execution time on a node is

assumed to be equal in every node) when we extrapolate this behaviour to the whole system.

2. We model the cost of a query in terms of disk fetch and CPU times, which involves obtaining each posting list and computing the corresponding intersections. To calculate CPU cost we run list intersection benchmarks while the disk access time is calculated fetching all the terms from disk using a real search engine (this also considers decompressing the postings). We return on this issue in Chapter 3 expanding the explanation with the details of the cost model.

3. All simulation results reported in this work assume query processing in conjunctive mode (i.e., *AND* queries) and Term-at-a-Time evaluation strategy (which enables the caching of intersections of terms).

4. The size of the datasets and the novelty of the query log are limited too[4]. However, we believe that this data corresponds to the type and size of data that would be indexed by a single search node in a distributed search engine.

1.4 Publications

Throughout the development of this thesis partial versions of the results included here have been originally published in different scientific meetings.

- The contents of Chapters 3 and 4 devoted to cost-aware intersection caching resulted in two publications:
  - Feuerstein, E. and Tolosa G. *Analysis of Cost-Aware Policies for Intersection Caching in Search Nodes.* In SCCC Conference, Temuco (Chile), November, 2013.
  - Feuerstein, E. and Tolosa G. *Cost-aware Intersection Caching and Processing Strategies for In-memory Inverted Indexes.* In 11th International Workshop on Large-Scale and Distributed Systems for Information Retrieval at WSDM'2014. New York (USA), February, 2014.
- The ideas proposed and evaluated in Chapter 5 were originally published in:
  - Tolosa, G.; Feuerstein, E.; Becchetti, L. and Marchetti-Spaccamela, A. *Performance Improvements for Search Systems using an Integrated Cache of Lists+Intersections.* In String Processing and Information Retrieval (SPIRE), Ouro-Preto (Brazil), October, 2014.

[4]In both cases, we use publicly available datasets.

- Finally, the proposal of admission policies for intersection caches introduced in Chapter 6 originated two more publications:
  - Tolosa, G. and Feuerstein, E. *Using Big Data Analysis to Improve Cache Performance in Search Engines.* In AGRANDA, Simposio Argentino de GRANdes DAtos (1st edition) at 44JAIIO - 44th Argentine Conference on Informatics. Rosario (Argentina), September, 2015.
  - Tolosa, G. and Feuerstein, E. *Optimized Admission Policies for Intersection Caches using a Data Mining Approach.* In iSWAG, International Symposium on Web Algorithms. Deauville, Normandy (France), June, 2016.

1.5 Organization

In Chapter 2, we provide the background information about web scale search engines and the main concepts regarding caching techniques in this context. We introduce the typical architecture of a search engine with the main data structures and query processing strategies involved. The literature review on search engine caching is also introduced in this chapter.

In Chapter 3 we propose cost-aware replacement policies for Intersection Caches using both static, dynamic and hybrid approaches. We also propose different query resolution strategies that consider the existence of this type of cache and finally we introduce our experimentation framework and the data used for evaluation purposes.

In Chapter 4 we introduce a study of cost-aware intersection caching using the query resolution strategies proposed in the previous chapter and different combination of caching policies. Our experiments cover two scenarios, that is, with the inverted index residing in disk and main memory.

In Chapter 5, we propose and evaluate an approach called *Integrated Cache* that replaces both lists and intersection caches at search node level. This cache tries to capture the benefits from both approaches in just one (integrated) memory area. We also analyze the impact of compression techniques and provide an evaluation using two real document collections.

In Chapter 6 we introduce an admission policy based on the usage of Machine Learning techniques to decide which items (intersections) should be cached and which should not. We build the proposed policy on the top of a classification process that predicts whether an intersection is frequent enough to be inserted in the cache. We also incorporate the information of the admission policy into the query resolution strategy by trying to compute

only intersections that are likely to appear again in future thus increasing the effectiveness of the intersection cache as well.

Finally, we establish the conclusions of our work in Chapter 7 and mention some future research directions that have emerged throughout the development of this thesis.

Introducción - Resumen

La Recuperación de Información (RI) es una campo de las Ciencias de la Computación relacionado con el almacenamiento, organización y búsqueda de información relevante en colecciones de documentos. En los últimos años, este problema se ha incrementado con el crecimiento exponencial de la web, llevando a las aplicaciones de RI fuera de los límites de una sola computadora y requiriendo soluciones tanto en eficiencia como en eficacia. Las aplicaciones en este dominio son los motores o máquinas de búsqueda, los cuales usan técnicas sofisticadas para manejar el problema.

La arquitectura típica de un motor de búsqueda está compuesta por un nodo *broker* que recibe las consultas y un conjunto de nodos de búsqueda organizados en un cluster que manejan un gran número de documentos en paralelo, usando estructuras de datos distribuidas. Además, para cumplir con los requerimientos de eficiencia, implementan técnicas sofisticadas de optimización, donde una de las más importante es *caching*, que básicamente consiste en mantener en una memoria de rápido acceso ítems previamente usados, basándose en patrones de frecuencia, acceso y costo.

En el contexto de los motores de búsqueda, la técnica de caching se implementa en varios niveles. Generalmente, un caché de Resultados se implementa en el *broker*, el cual almacena la respuesta final de un conjunto de consultas previas de usuarios. Este nivel de caché ha sido ampliamente estudiado en el pasado. En los nodos de búsqueda, se implementa, al más bajo nivel, un caché de Listas, el cual alcanza la mayor tasa de aciertos ya que corresponde al contenido asociados los términos individuales de una consulta.

De forma complementaria, se puede implementar un caché de Intersecciones, el cual intenta explotar combinaciones de términos que ocurren frecuentemente en las consultas, manteniendo el resultado final de intersecar las listas de n términos. Este caché cobra interés particular cuando el índice invertido reside completamente en memoria ya que, en este caso, pierde sentido el caché de Listas. Además, El número potencial de intersecciones es virtualmente infinito y depende de las consultas realizadas por los usuarios.

Particularmente, este trabajo se focaliza en el caching de Intersecciones mediante políticas que consideran el costo de los ítems (*cost-aware*), con el objetivo de reducir el tiempo total de procesamiento. Para cada tipo de caché se han desarrollado numerosas políticas de reemplazo, evaluadas, en general, usando la tasa de aciertos. Algunas de estas políticas clásicas en la literatura se basan en parámetros como frecuencia o frescura de los ítems (como las muy conocidas LFU y LRU), mientras que otras consideran alguna clase de *ganancia* o *beneficio* de mantener un conjunto particular de ítems en caché como Landlord o GreedyDual-Size.

Teniendo en cuenta que el caché de Intersecciones no ha sido muy estudiado en el pasado, en esta tesis se diseñan, analizan y evalúan políticas de caching que tienen en cuenta el costo de los ítems para implementar un caché de Intersecciones. Las contribuciones se puede dividir en tres direcciones principales:

En la primera parte, se diseñan e implementan políticas estáticas, dinámicas e híbridas que consideran el costo adaptando algunas políticas de otros dominios a éste. Estas políticas se combinan con diferentes estrategias de resolución de las consultas, algunas de las cuales son originalmente diseñadas para tomar provecho de la existencia del caché de Intersecciones, reordenando los términos de la consulta de una forma particular. Las propuestas se evalúan considerando tanto un índice invertido en disco como en memoria principal.

En la segunda parte, se explora la posibilidad de reservar el espacio de memoria para un caché Integrado (estático) de listas e intersecciones en forma simultánea. Para ello, se diseña una estrategia de gestión específica junto con el uso de diferentes aproximaciones para inicializar su contenido.

Finalmente, se diseña y evalúa una política de admisión para el caché de Intersecciones basada en principios de *Machine Learning* que intenta que accedan al caché los ítems que podrían ofrecer mayores beneficios. Esta situación se modela como un problema de clasificación y se incorpora la decisión dentro de la estrategia de resolución de la consulta. De esta manera, se trata de minimizar el cómputo y caching de intersecciones con bajas chances de ser solicitadas nuevamente.

Chapter 2

Background and Related Work

2.1 Search Engines Architecture

As we succinctly mentioned in the previous chapter, a search engine is organized in a cluster of commodity computers. The components of its architecture are basically nodes for web servers (brokers), index servers (search nodes), document servers and snippet servers [14]. In some cases, some ad servers are also deployed. This cluster contains a replica of a web crawl[1] (the document collection).

In this chapter we expand the details about the data structures used to support the efficient retrieval of the results and the strategies used to split the document collection among the search nodes. The basic processing of a query is also covered along with the basic caching levels in the architecture.

2.2 Data Structures

Given a collection of documents the first step is to extract the set of useful terms that represents its contents. This process requires parsing, tokenization techniques and additional word handling such as stopword removal and stemming (for details, please see [8, 110]). Then, the result may be represented with a matrix $M = (R \times C)$, where rows correspond to the documents and columns represent the terms. Each element (i, j) typically stores the within-document frequency $f_{d,t}$ of term t. However, for real collections this matrix becomes too sparse and is often unpractical.

[1] *Crawling* is the process of automatically navigating the web extracting the contents of web pages and other documents. For a detailed explanation, see [8, 25]

In order to efficiently answer queries all IR systems build an index over the collection of documents (namely, an Inverted Index). In its simplest implementation, two data structures are built: the lexicon and the posting lists. The former stores the vocabulary of the collection (the set of all unique terms considered) and the latter are lists that store the occurrence of each term within the document collection. Actually, the posting list contains the document identifiers (*docID*) and additional information used for ranking (e.g. the within-document frequency $f_{t,d}$). The inverted index is, in fact, a more compact and practical representation of the term-document matrix.

Some inverted index implementations often keep the position of each occurrence of the terms, the context in which it appears (e.g. the title or a meta-tag) or even these implementations may also contain information to traverse the posting list in an efficient way (e.g. skip-lists) [80]. For example, given the following four documents:

$d_1 = \{$if you prick us we do not bleed. we do not want to suffer.$\}$
$d_2 = \{$if you tickle us we do not laugh.$\}$
$d_3 = \{$if you poison us we do not die. do you die? does she die?$\}$
$d_4 = \{$if you wrong us we shall not revenge. we never revenge.$\}$

Figure 2.1 shows an example of the resulting inverted index composed by the vocabulary, formed by all the terms and the corresponding document frequency, and the posting lists that contain the docIDs with their associated term frequencies.

Term	**DF**		**Posting Lists**
bleed	1		(1, 1)
die	1		(3, 3)
do	4		(1, 2) (2, 1) (3, 1) (4, 1)
does	1		(3, 1)
if	4		(1, 1) (2, 1) (3, 1) (4, 1)
laugh	1		(2, 1)
never	1		(4, 1)
not	4		(1, 2) (2, 1) (3, 1) (4, 1)
poison	2	$\longrightarrow$	(3, 1) (4, 1)
prick	1		(1, 1)
revenge	1		(4, 2)
shall	1		(4, 1)
she	1		(3, 1)
suffer	1		(1, 1)
tickle	1		(2, 1)
to	1		(1, 1)
us	4		(1, 1) (2, 1) (3, 1) (4, 1)
want	1		(1, 1)
we	4		(1, 1) (2, 1) (3, 1) (4, 1)
you	4		(1, 1) (2, 1) (3, 2) (4, 1)

FIGURE 2.1: Example of the inverted index generated after indexing the documents in the example.

A common approach is to sort each posting list in increasing order of docID. Although other orderings are possible (according to different query resolution strategies), the postings are usually ordered by increasing docID since this ordering is required by most of the state-of-the-art retrieval algorithms [20] and this ordering is also necessary both to compute intersections and to compress the index efficiently [110, 116].

When sorting the lists by increasing docID, gap encoding (DGap) may be used. This basically works as follows: the first document identifier is represented as it is, whereas the remaining identifiers are represented as the difference with the previous one.

For example: given a term t_i and its posting list:

$\ell_i = \{22, 28, 48, 49, 50, 51, 52, 67, 68, 69, 70, 79, 80\}$

the DGap encoding of ℓ_i becomes:

$\ell'_i = \{22, 6, 20, 1, 1, 1, 1, 5, 1, 1, 1, 9, 1\}$.

Although this approach reduces the size of the resulting data structure, compressing the inverted index is also a crucial approach used to improve query throughput and fast response times. Data is usually kept in compressed form in memory (or disk) leading to a reduction in its size that would typically be about 3 to 8 times [110], depending on index structure, stored information and compression method. The DGap'ed representation is particularly useful because the compression methods benefit from small sequences of integers, particularly of long sequences of consecutive 1s [5].

Usually, docIDs, frequency information and positions are stored separately and can be compressed independently, even with different methods. Among the compression method for posting lists we have the classical Elias [37] and Golomb [49] encoding and the more recent ones Simple9 [2], PForDelta [118], and Elias-Fano [85] encodings. These techniques have been studied in depth in the literature [8, 74, 110].

In many cases, inverted lists are logically divided into blocks (i.e. 128 DGaps each) to speed up their traversal when searching for a particular docID. This enables the decompression of only those blocks that are relevant to a particular search. In the case of extremely large document collections, more sophisticated inverted index organizations may be used. A well known case is the Block-Max Index [34], which enables a more aggressive skipping in the index to quickly access to the relevant docIDs. Another important reference is the work of Konow et al. [61] that introduces a new representation of the inverted index based on the treap data structure. The treap-based index allows the representation of both docIDs and frequencies separately, and enables competitive

compression rates of the lists by the use of compact data structures and lower query resolution times. There is a considerable body of literature on index construction (please refer to [110, 117]).

2.2.1 Index Partitioning

In a search engine organized in a cluster, a distributed version of the inverted index is used. This increases the concurrency and the scalability of the system as multiple servers run in parallel, each one accessing a portion of this index. The index is splitted in shards that are assigned to different search nodes. According to the strategy used to partition the data, different types of distributed indexes appear.

2.2.1.1 Document-based Partitioning

In this approach (a.k.a. local index), documents are distributed onto P processors where an independent inverted index is constructed for each of the P sets of documents. Therefore, answering a conjunctive query requires computing the intersection of the terms that compose the query at each processor, obtaining partial results by performing the corresponding ranking to select the top-k local documents.

This is the most common approach used by a web distributed search engine [7] because it has several advantages. As the search nodes operate independently, the deployment in a loosely coupled environment is rather easy and the index is also simple to maintain (insertion and deletion of docIDs is done locally). In the case that the search engine requires more computing power, new search nodes may be added in a simple way (this is also useful for fault tolerance and availability purposes). The strategies to distribute the documents onto the shards to maintain an even load are a research question, but this scheme seems to be still the best choice for parallelization [21].

2.2.1.2 Term-based Partitioning

In the term-partitioned index (a.k.a. global index), a single inverted file is constructed from the whole document collection. The resulting terms with their respective posting lists are then evenly distributed onto the processors. In this way, to answer a conjunctive query the broker needs to determine which processor(s) hold the posting lists of the involved terms, then gather those lists in one processor and compute their intersection. Afterwards, the ranking is performed over the resulting intersection set.

The most important advantage of this approach is that only a limited number of search nodes are used to compute the results but this generates an uneven distribution of the load across the nodes (several optimizations are available).

The construction of the index and its partitioning is another important issue. The simple approach of building the entire index before the partitioning does not scale well, and may be considered a limiting factor of performance. However, more sophisticated approaches [62] provide efficient parallel strategies to build the index that eliminate the need for the generation of a global vocabulary. The update of the index is also a complex operation because many nodes must be contacted to add/delete/update document information.

Several studies have compared these two organizations, summarizing their advantages and disadvantages [6, 75, 82]. Most of them point out the potential of term-based partitioning, which offers superior performance under particular circumstances [82]. However, as we mentioned before, the high network load and CPU imbalance pose limitations to current approaches.

2.2.1.3 Hybrid Partitioning

In addition to the two classical distributed index partitioning schemes, a 2-dimensional (2D) index was introduced in [42]. The 2D index combines document- and term-partitioning to get the "best of two worlds", i.e. to exploit the trade-off between the overhead due to the involvement of all the processors in each query resolution as in the former, and the high communication costs required by the latter. The aforementioned study shows that, given a number of processors (P), it is possible to achieve a reduction of the total processing cost by selecting an adequate number of rows (R) and columns (C).

In a later work [44], a 3D indexing strategy is proposed that properly considers the fact that data is partitioned and also replicated (D) to increase query throughput and to support failures. The authors evaluate different combinations of $C \times R \times D$ for a given number of processors. This architecture also considers a distributed intersection cache that reduces communication of posting lists among nodes.

2.2.2 Multi-tier Indexes

Another proposed architecture is the splitting of the inverted index into more than one tier [94]. This approach is intended for top-k retrieval. The processing of a query starts using the small tier and only if the result set is not satisfactory (that is, the number of results does not reach k documents), a bigger tier is used. Performance improvements

are achieved when large tiers are not accessed, sometimes at the expenses of returning an approximate answer.

A different work [83] introduces a two-tiered method that avoids the degradation of the results. Basically, the most important documents are maintained in the first tier that is located in the main memory, selecting the candidates by the use of index pruning techniques. This approach works as a cache level to speed up the query processing.

A similar approach is used in [95] but the lists with higher impacts are maintained in the small index. The authors evaluate a case where the second tier is disjointed (with regard to the first tier) and another one which contains the full index.

2.2.3 Posting List Compression

Efficient access to the lists data structure is a key aspect for a search system, mainly when the index resides in hard disk. Many compression techniques have been developed and evaluated to deal with long lists of integers that represent the document identifiers and complementary information associated with each of them. For example, classic techniques such us Variable-Byte Encoding [109] and Simple 9 [2] or the state-of-the-art PForDelta [118] and its optimized versions are commonly used. The work of Yan [112] and Zhang [115] demonstrates superior efficiency of PForDelta and its variants compared to other compression methods. Besides, for 32-bit words (for example, docIDs are commonly represented as 32-bits integers) PForDelta performs fast decompression of data which is a desirable property of a compression technique. In the same direction, a recent work of Ottaviano et al. [85] proposes two new index structures based on the Elias-Fano representation of monotone sequences. According to the evaluation, the new representations, Partitioned Elias-Fano Indexes, offer the best compression ratio/query time trade-off.

In [115] the authors explore the combination of inverted index compression and list caching to improve search efficiency. They compare several inverted list compression algorithms and list caching policies separately and finally study the benefits of combining both, exploring how this benefit depends on hardware parameters (i.e. disk transfer rate and CPU speed).

In a recent work, Catena [28] analyses the performance of modern integer compression schemes across different types of posting information (document identifiers, frequencies and positions). They analyze the space and time efficiency of the search engine when compressing different types of posting information. They show that the simple Frame of Reference [48] codec achieves the best query response times in all the cases slightly outperforming PForDelta.

Finally, a brand-new publication of Ottaviano et al. deserves special attention [86]. In this work, authors combine fast encoders for frequently accessed lists and more space-efficient ones for rarely accessed lists. They introduce a linear time algorithm that selects the best encoder for each block of data (according to its popularity) to achieve the lowest query processing time. According to their experiments, this optimization outperforms single-encoder solutions.

2.3 Query Processing

A query $q = \{t_1, t_2, ..., t_n\}$ is a set of terms that represents the user's information need. The processing of queries in a distributed (multi-node) search system is usually handled as follows [22]: the broker machine receives the query and looks for it in its result cache. If the result is found the answer is immediately returned to the user with no extra computational cost. Otherwise, the query is sent to the search nodes in the cluster where a distributed version of the inverted index resides.

Each search node searches for the posting lists of the query terms. In the case that the inverted index resides completely in the main memory the cost of fetching data from disk is saved. However, this is not always the case and some systems store a portion of the index in secondary memories and another portion is kept in a posting list cache. In this scenery, for each query term, the search node tests first the existence of its posting list in cache to avoid disk access. Finally, it executes the intersection of the sets of document identifiers and ranks the resulting set. For conjunctive queries it makes sense to cache the intersection of *some* terms. This is a time-consuming task that is critically important for the scalability of the system because of disk accesses (partially affected by the size of the document collection). To mitigate this situation different cache levels are used to reduce disk costs.

In the final stage of this process, a scoring function computes the scores for documents containing the query terms using the information in the posting lists (i.e. the within-document frequency $f_{d,t}$ or impact scores). There exists a wide range of scoring functions based on different theoretical models such as the vector space model, language models or the commonly used BM25 method. Different approaches are described in the literature (please refer to [8, 74]). Finally, documents are sorted in decreasing order according to their scores and the top-k resulting document identifiers are sent back to the broker (along with their corresponding scores).

The last part of the computational process consists typically of taking, at the broker level, the top-k (in general, between 10 and 1000) ranked answers to produce the final

result page. This is composed of titles, snippets and URL information for each resulting item. As the result page is typically made of 10 documents, the cost of this process may be considered constant as it is only slightly affected by the size of the document collection. In our work, we focus our attention on the first phase of the query processing task by using the intersection cache and different strategies to solve the query.

2.3.1 Query Evaluation Strategies

At a high abstraction level, the query processing task requires to retrieve the posting lists (from memory or disk) of the corresponding query terms, iterate through them and accumulate the scores of the candidate documents. The traversal of the lists can be made by means of two main strategies [107], namely Document-at-a-Time (DAAT) and Term-at-a-Time (TAAT) or a variant of the latter known as Score-at-a-Time (SAAT).

2.3.1.1 Document-at-a-Time

In the DAAT approach the posting lists for all query terms are read concurrently (instead of processing them consecutively). To enable this possibility, the postings of a term must be stored in increasing order of document identifiers.

Each iteration of the query processor picks a candidate docID, accumulates its score and moves forward to the next posting. In this way, it is possible to compute the final score for a document while traversing the lists and before all postings in the lists are completely processed. As a consequence, the algorithm may store only the k-th best candidates thus requiring the use of a small amount of memory (i.e. the space required to store a heap of size k).

The Max Successor algorithm [30] is an efficient strategy for DAAT processing while the WAND strategy [20] is a dynamic pruning approach that allows fast query processing for both conjunctive and disjunctive queries. Another interesting DAAT approach used to solve ranked union and intersection queries is the work of Konow et al. [61]. As we mentioned earlier, this work is based on the treap data structure, which allows to intersect/merge the document identifiers and supports the thresholding by frequency of results simultaneously list traversal. The final top-k results can be obtained without having to first produce the full Boolean intersection/union.

2.3.1.2 Term-at-a-Time

In the TAAT approach, the posting lists of the query terms are sequentially evaluated, starting from the shortest to the longest one. The postings of a term are completely processed before considering the postings of the next query term. For this reason, the query processor has to keep accumulators with the scores of partially evaluated documents. After processing the last term, the query processor has to extract the k best candidates from the scored accumulators.

TAAT can be considered as more efficient with respect to index access (especially for disk-based indexes), buffering, CPU cache and compiler optimizations. However, it has to maintain a complete accumulator set, which at the end is equivalent to a union of the posting lists (in the case of "OR" queries). On the other hand, DAAT requires parallel access to posting lists, which affects the performance of the traditional hard-disks and internal CPU cache, but it has to keep only the k best candidates seen so far.

In the case of conjunctive queries, TAAT is the common approach. Given the query $q = \{t_1, t_2, ..., t_n\}$ and its corresponding posting lists ℓ_i, this basically consists of solving the intersection $\bigcap_{i=1}^{n} \ell_i = (((\ell_1 \cap \ell_2) \cap \ell_3) \ldots \cap \ell_n)$. Experimental results suggest that the most efficient way to solve a conjunctive query is to intersect the two shortest lists first, then the result with the third, and so on [61].

2.3.1.3 Score-at-a-Time

This approach [3] is an alternative strategy to DAAT and TAAT that relies on a particular index organization (i.e. impact-ordered indexes) that sorts the posting lists by document importance. The index is organized in segments (or blocks) of data according to their impact scores. Segments for each term are ordered in decreasing impact score and documents are ordered by increasing document identifiers within each segment.

The query evaluation strategy processes posting segments in decreasing order of impact scores and stops the evaluation when a specified number of postings has been processed. During this evaluation, an accumulator structure that holds the k most relevant elements is maintained. This strategy enables a dynamic early-termination check that avoids the traversal of useless blocks of data [68].

2.4 Caching in Search Engines

Caching is a broadly used technique in computer systems. The basic idea is to store *some* items in a fast memory that might be requested in a near future. This mainly benefits from the temporal locality between two successive requests of the same item.

One of the oldest applications of caching relates to the paging problem in operating systems [102]. This basically attempts to maintain frequently or recently used disk pages in cache. In the database world [36], caching is a commonly used technique for scaling and achieving high throughput [105]. The web content delivery (i.e. proxy servers and Content Delivery Networks) [52] is also another field in which this technique becomes essential.

A cache may be managed using static, dynamic or hybrid policies. The first consists of filling a static cache of fixed capacity with a set of predefined items that will not be changed at run-time. The idea is to store the most valuable items to be used later during execution. This approach basically exploits long-term popularity of items but it cannot deal with bursts produced in short intervals of time. Dynamic policies handle this case in a better way. A dynamic cache requires a replacement strategy to decide which item to evict when the cache is full. In the last case, hybrid caching [38] combines both approaches by reserving two portions of memory space: The first one for a static set of entries (filled on the basis of usage data) and the another one for a dynamic set managed by a given replacement policy.

In the context of web search engines caching has become an important and popular research topic in recent years. Much work has been developed in issues such as *Document Caching* [16, 59, 71], *Lists Caching* [10, 12, 70, 106] and *Results Caching* [65, 77, 87, 88]. However, *Intersection Caching* [29, 70] has not received enough attention.

2.4.1 Result Caching

The first cache level (Result Cache) is located at the broker side and enables the possibility of storing the final result of the queries submitted by the users. This cache may hold the list of docIDs that forms the result set or the complete result page (in HTML format). It is important to highlight that the SERP (Search Engine Result Page) contains the title, URL and snippet of each resulting document (typically ten results per page). A snippet is a small portion of text extracted from the original result document that is related to the query itself. If the cache only stores the list of docIDs, an extra step of title/URL/snippet generation is needed (this running time is considered constant).

One of the first reference in Result Caching is the work of Markatos [77]. He shows that there exists a temporal property in query requests analyzing the query log of the Excite web search engine. His work compares static vs dynamic policies showing that the former ones perform better for caches of small sizes while the latter are advantageous for medium-size caches. Lempel and Moran [65] propose a probabilistic approach called PDC (Probabilistic Driven Caching) that attempts to estimate the probability distribution of all possible queries submitted to a search engine using a model of user behaviour that analyzes search sessions.

The work in [38] considers hybrid policies (named Static-Dynamic Caching or SDC) for managing the cache, that is, a combination of a static part for popular queries over the time and a dynamic part to deal with bursty queries. Their experiments show that SDC achieves better performance than either purely static caching or purely dynamic caching. Gan and Suel [47] consider the weighted result caching problem instead of focusing on hit ratio, while [88] propose new cost-aware caching strategies and evaluate these for static, dynamic and hybrid cases using simulation. They adapt commonly used caching policies to the cost-aware case and introduce two new policies for the dynamic case. This extensive work shows that hybrid policies perform properly, obtaining the highest performance improvements. In another interesting work, Skobeltsyn et al. [100] combine index pruning and result caching. They show that results caching is an inexpensive and efficient way to reduce the processing load at the back-end servers and easier to implement compared to index pruning techniques.

Finally, a recent work by Sazoglu et al. [97] introduces other dimension of cost analysis to be considered in a Result Cache replacement policy. They take into account a financial cost metric on the basis of the hourly electricity price rate. This idea is also supported by the observation that query energy consumption seems to be linear with the query processing time [27]. In their proposal, when a cache miss occurs, they compute the processing time of a query weighted by the electricity price at the moment of execution. This cost metric is then included in different cost-aware replacement strategies. They also propose Two-Part LRU Cache (2P-LRU) that handles cheap and expensive queries in separate memory spaces.

2.4.2 List Caching

In the case that the query cannot be found in the Result Cache, the broker sends it to the search nodes. According to the index partitioning strategy, all nodes or only a subset of them must be contacted. Each search node solves the query (or a portion of it) using its own inverted index, which may be stored in the local disk or in the main memory. In

the former case, this requires disk accesses to fetch the posting lists for each term. In order to decrease the cost of accessing the disk, a posting list cache is implemented. This basically stores the list of docIDs of a subset of terms in the index selected according to different criteria.

The caching of posting lists has been extensively studied. Baeza-Yates et al. [10] analyze the problem of posting list caching (combined with results caching) that achieves higher hit rates than caching query results. They also propose an algorithm called $Q_{tf}D_f$ that selects the terms to put in cache according to its $\frac{frequency(t)}{size(t)}$ ratio. The most important observation is that the static $Q_{tf}D_f$ algorithm has a better hit rate than all dynamic versions. They also show that posting list caching achieves a better hit rate because the repetitions of terms are more frequent than repetition of queries.

In [115], the performance of compressed inverted lists for caching is studied. The authors compare several compression algorithms and combine them with caching techniques, showing how to select the best setting depending on two parameters (disk speed and cache size). They include some eviction policies that balance recency (LRU) and frequency (LFU) but in practice the two of them perform similarly. In a more recent work, Tong et al. [106] introduce a strategy for static list caching in SSD-based search nodes. They take into consideration the block-level access latency in flash-based solid state drives vs common mechanical hard disk drives, reducing the average disk access latency per query.

2.4.3 Multi-level Caching

The idea of using a two-level caching architecture that combines caching of search results with the caching of posting list was first introduced in [96]. They use index pruning techniques for list caching and an LRU-based eviction policy in both levels and show that this approach can effectively increase the overall throughput.

The intersections cache is first used in [70] where a three-level approach is proposed for efficient query processing. In this architecture a certain amount of extra disk space (20-40%) is reserved for caching common postings of frequently occurring intersections. They also state that the use of a Landlord policy improves the performance. Their experimental results show significant performance gains using this three-level caching strategy. This idea is also taken in [29] where results of frequent subqueries are cached and a similar approach is exploited in [35] for speeding up query processing in batch mode.

Kumar [63] considers top-k aggregation algorithms for the case when pre-aggregated intersection lists are available, but neither different caching policies nor intersecting strategies are evaluated. The work in [76] and [43] consider intersection caching as a part of their architecture. In the former work, the authors propose a five-level cache hierarchy combining caching at broker and search node level while the latter presents a methodology for predicting the costs of different architectures for distributed indexes. In [44], a three-dimensional architecture that includes an intersection cache is also introduced.

Finally, Ozcan et al. [89] introduce a multi-level static cache architecture that stores five different item types that are usually independently managed in search engines. In this architecture, they reserve a global space to cache query results, precomputed scores, posting lists, intersections of posting lists, and documents. The space capacity is globally constrained so the number of items of each class depends on precomputed gains. They also propose a greedy heuristic to populate the static cache that takes into account both past access frequencies, estimated costs and storage overheads and the inter-dependency between individual items as well.

Antecedentes y Trabajos Relacionados - Resumen

La arquitectura de una máquina de búsqueda está formada principalmente por servidores web (brokers), de índice (nodos de búsqueda), de documentos y de *snippets*. En los nodos de búsqueda, la estructura de datos utilizada es el índice invertido, el cual está compuesto por un vocabulario que contiene todos los términos de la colección y un conjunto de listas invertidas que contienen, mínimamente, los documentos donde aparece cada término y su frecuencia asociada.

Usualmente, ambos tipos de información son comprimidos de forma separada utilizando diversos métodos, por ejemplo, códigos Elias y Golomb entre los clásicos o más recientes como Simlple9, PForDelta y Elias-Fano. Además, este índice se divide entre todos los nodos de búsqueda, ya sea, por documentos o por términos a los efectos de distribuir el problema entre los nodos o también, es posible organizarlo en diferentes niveles que ofrecen mejoras de performance.

En cuanto a las estrategias para la resolución de las consultas el, motor de búsqueda procede de la siguiente manera: el broker recibe al consulta y verifica si existe una entrada en su caché de resultados. De ser así, responde directamente al usuario sin necesidad de involucrar a los nodos de búsqueda (con un costo despreciable). De otra manera, debe enviar la consulta a éstos, los cuales la resuelven usando uno de dos enfoques principales, a saber: *Document-at-a-Time* y *Term-at-a-Time*. En el primero de los casos, los documentos candidatos son chequeados para evaluar si satisfacen (o no) a la consulta. Las listas de todos los términos son evaluadas en paralelo para determinar los *top-k* mejores. En el segundo caso, se evalúa cada término por completo, es decit, básicamente se resuelve la intersección $\bigcap_{i=1}^{n} \ell_i = (((\ell_1 \cap \ell_2) \cap \ell_3) \ldots \cap \ell_n)$.

Una vez que los nodos de búsqueda resuelven su parte del problema, envían los resultados parciales al broker, el cual arma la página final de resultados (que contiene los *links*, *snippets*, avisos publicitarios, etc.) y la devuelve al usuario, además de insertar los resultados en el caché correspondiente.

En cuanto a las políticas de caching en motores de búsqueda, se pueden utilizar enfoques estáticos, dinámicos o híbridos. Los primeros consisten en llenar un caché de capacidad finita al inicio (el cual no se modifica durante la ejecución). Los segundos, utilizan una política de reemplazo que decide cuál ítem expulsar del caché cuando este está lleno y se requiere espacio para insertar uno nuevo. Finalmente, los cachés híbridos combinan los dos anteriores, reservando una porción estática y otra dinámica. En el contexto de motores de búsqueda, caching ha sido un tema muy importante y popular en los últimos años, por ejemplo, caching de Documentos, de Listas y de Resultados. Sin embargo, el caching de Intersecciones no ha recibido suficiente atención.

Además, se han realizado estudios en arquitecturas multinivel que involucran los cachés antes mencionados. Saraiva y otros combinan por primera vez un caché de Resultados con uno de Listas, usando LRU como política de reemplazo y mostrando como efectivamente se incrementa el *throughput* de la aplicación. El caché de Intersecciones es introducido por Long y otros en una arquitectura de tres niveles. En este caso, este caché se almacena en disco conteniendo el resultado de intersecar los pares de términos más frecuentes.

Finalmente, Ozcan y otros introducen una arquitectura en la cual cinco diferentes tipos de ítems son mantenidos en un caché estático. Básicamente, se almacenan en un espacio global resultados, scores precomputados, listas, intersecciones y documentos, considerando la ganancia que cada tipo de ítem aporta. Su heurística para llenar el caché considera frecuencia, costos, overheads y la interdependencia de los ítems individuales también.

Chapter 3

Caching Policies and Processing Strategies for Intersection Caching

Search engines need to use different levels of cache for efficiency and scalability purposes. A number of strategies to fill and manage the cache have been proposed and evaluated. To introduce different proposals of caching techniques we may classify them into static or dynamic approaches. In the first case, a cache of fixed capacity is filled with precomputed items according to different criteria and its state is not modified at running time. In the second case, the cache is populated online. When the cache becomes full, an eviction decision is taken to free space for the new item. This decision is based on a given replacement policy that chooses the "victim" to move out according to different criteria as well. Different combinations of static and dynamic policies are also possible (known as hybrid policies) such as the well-known SDC strategy (Section).

According to Podlipnig and Böszörmenyi [92], the main factors (or features) that determine which items to select and influence the replacement process in both static and dynamic strategies are the following:

- Recency: time of last reference to a cached item.
- Frequency: number of requests to an item.
- Size: size of the item (i.e. the size of the posting list).
- Cost: cost to get an object (i.e. the cost of fetching a list from disk).
- Modification time: time of last modification.
- Expiration time: time when an object gets stale and has to be replaced immediately (i.e. an expired Time-to-Live).

Different caching policies make use of only one of the listed factors or a combination of them according to different scenarios which they are designed for. For example, cases where items have different sizes, specific patterns arrival and/or different costs, etc. are handled using specific policies. In the case of Results Caching it is stated that query processing costs may significantly vary among different queries and their processing cost is not necessarily proportional to their popularity. In the case of Intersection Caching a similar situation occurs. However, unlike query result items, the intersections have variable sizes so this is a factor to be considered.

In the first part of this chapter we introduce different static, dynamic and hybrid policies to evaluate the intersection cache. Our basic contribution is to consider different cost-aware policies adapted to our problem and compare them against the state-of-the art cost-oblivious caching policies used in other cache levels in search engines (i.e. result or list caches). In the second part, we introduce different strategies to compute list intersections. We also contribute by proposing three new strategies that take advantage of the existence of an intersection cache and compare them against a basic TAAT intersection strategy.

3.1 Intersection Caching Policies

The task of filling the cache with a set of items that maximize the benefit in the minimum space can be seen as a version of the well-known knapsack problem, as discussed in previous work related to list caching [10]. Here, we have a limited predefined space of memory for caching and a set of items with associated sizes and costs. The number of items that fit in the cache is related to their sizes so the challenge is to maintain the most valuable ones in the cache.

Initially, we only focus on two-term intersections so we introduce the notation used to describe the considered policies in Table 3.1

Identifier	**Description**
t_i	Term i
ℓ_i	The posting list of t_i
I_{ij}	The intersection between two terms: $I = (t_i \cap t_j)$
$F(I_{ij})$	Frequency of the intersection I_{ij}
$C(I_{ij})$	Cost of the intersection I_{ij}
$S(I_{ij})$	Size of the the resulting list I_{ij}

TABLE 3.1: Notation used to describe the caching policies.

3.1.1 Static Policies

Static caching is a well-known approach used both in *list caching* and in *result caching* and we determine that is also useful in *intersection caching*. We evaluate different metrics to populate the cache trying to store the most valuable items to be used later during execution.

- **FB (Freq-Based):** This method fills the cache with the most frequent intersections found in a training set.
- **CB (Cost-Based)** In this case, the cache is populated with the most costly intersections. The cost of a two-term intersection is usually computed as the sum of the cost of fetching the posting lists of the terms plus the time incurred to intersect both lists.
- **FC (Freq*Cost):** This approach uses the items that maximize the product $F(I_{ij}) \times C(I_{ij})$.
- **FS (Freq/Size):** It computes the ratio $\frac{F(I_{ij})}{S(I_{ij})}$ and chooses the items that maximize this value. It corresponds to $Q_{TF}D_F$ approach [10] used to fill a static list cache. In that work, "value" corresponds to $f_q(t)$ and "size" corresponds to $f_d(t)$.
- **FkC (Freq^k*Cost):** The same as FC but it prioritizes higher frequencies according to the skew of the distribution. This idea is introduced in [88] to emphasize higher frequency values and depreciate lower ones under the observation that queries that occur with some high frequency still tend to appear with a high frequency, whereas the queries with a relatively lower frequency may appear even more sparsely in a near future.
- **FCS (Freq*Cost/Size):** This is a combination of the three features that define an intersection trying to best capture the value of maintaining an item in the cache. This computes the product $F(I_{ij}) \times \frac{C(I_{ij})}{S(I_{ij})}$ and sorts the items in descending order according to this value.
- **FkCS (Freq^k*Cost/Size):** It follows the previous idea, but it emphasizes $F(I_{ij})$ as in FkC.

3.1.2 Dynamic Policies

Although static caching is an effective approach to exploit long-term popularity of items, it cannot deal with bursts produced in short intervals of time. A dynamic approach

handles this case in a better way according to the repetition patterns in the input stream. Different replacement policies that decide which item evicts when the cache is full are considered in this work:

- **LFU:** This strategy maintains a frequency counter for each item in the cache, which is updated when a hit occurs. The item with the lowest frequency value is evicted when space is needed.
- **LFUw:** This is the online (dynamic) version of the FC static policy. The score is computed as $F(I_{ij}) \times C(I_{ij})$. It is called LFUw in [47].
- **LRU:** The well-known strategy that chooses the least recently used item for eviction, independently of its size or cost.
- **LCU:** This policy is introduced in [88] for result caching. Each item in the cache has an associated cost (estimated according to an appropriate model). When a new item appears, the least costly cached element is evicted.
- **FCSol:** Online version of the static FCS policy. The value $F(I_{ij}) \times \frac{C(I_{ij})}{S(I_{ij})}$ is computed at run-time for each item.
- **Landlord:** In this policy [114], whenever an item i is inserted into the cache (or get a hit) a credit value is assigned proportional to its cost ($credit(i) = cost(i)$). When the cache is full, the algorithm decreases the credit of all items in cache by $\Delta \times size(i)$, where $\Delta = min_{j \in cache}(credit[j]/size[j])$. Then, the item i with $credit(i) = 0$ is selected for eviction.
- **GDS:** This is the *GreedyDual-Size* policy [23]. For each item p in the cache, it maintains a value $H\_value_p = \frac{C(I_{ij})}{S(I_{ij})} + L$. The parameter L is initialized to zero at the beginning. When the cache is full and a replacement needs to be made, L is recalculated as $L = min_{x \in cache}(H_{value}(x))$. Then, the cached item $p\prime$ with the lowest H_value is selected for eviction. The remaining items reduce their H_{value} by L. During an update, the H_value of the requested item is recalculated since L might have changed. Accordingly, the value of a recently-accessed item x retains a larger fraction of its original cost compared to items that have not been accessed for a long time.

Landlord is a generalization of the greedy-dual algorithm [113] for weighted caching that reduces all cached file's credits proportionally to its size and evicts files that run out of credit. Although GDS is essentially similar, the subtle difference of computing the score only with the item features (cost and size) and selecting the item with the smallest H_value as the victim versus Landlord that takes into account $min_{i \in cache}(credit[i]/size[i])$ as the discount factor lets GDS to perform well (better) in practice.

3.1.3 Hybrid Policies

Hybrid caching policies make use of two different sets of cache entries arranged in two levels. The first level contains a *static set* of entries which is filled on the basis of usage data. The second level contains a *dynamic set* managed by a given replacement policy. When an item is requested, it looks first into the static set and then into the dynamic set. If a cache miss occurs the new item is inserted into the dynamic part (and may trigger the eviction of some elements). The idea behind this approach is to manage frequent items with the static part and recent items with the dynamic part.

- **SDC:** As we mentioned in Section 3, this stands for "Static and Dynamic Cache" [38]. The static part of SDC is filled with the most frequent items while the dynamic part is managed with LRU.
- **FCS-LRU:** A variant of SDC where the static part is filled with the items with the highest $F(I_{ij}) \times \frac{C(I_{ij})}{S(I_{ij})}$ value. Our findings show that FCS performs well in practice so we include this strategy in the considered hybrid policies.
- **FCS-Landlord:** Similar to the previous one but the dynamic part is Landlord.
- **FCS-GDS:** In this case, the dynamic part is managed by the GDS policy.

3.2 Query Processing using an Intersection Cache

Query processing in web environments becomes a great challenge for search engines due to the continuous volume growth of indexable documents. A common technique to reduce the query processing cost is to assume that the result is only composed of those documents that match all of the query terms. Such queries are called conjunctive queries (or AND queries) and are often used in practical search engines for performance reasons [31][56].

In our case, AND queries become the target of this research because the concept and relevance of an Intersection Cache only apply to this kind of queries. It is clear that the result sets returned by AND queries are potentially smaller compared to OR (disjunctive) queries. In the case that one of the query terms is mistyped or missing in the underlying collection this leads to an empty result set. To overcome this situation, a naive solution is to evaluate a query in AND mode first, and only if the results are not sufficient, evaluate it in OR mode.

In the case of conjunctive queries, TAAT strategy is the common approach and it enables the following strategies. This basically consists of solving the intersection $\bigcap_{i=1}^{n} \ell_i$,

processing the two shortest posting lists first and then matching the result against the lists of the remaining terms [61]. Given a query $q = \{t_1, t_2, t_3, ..., t_n\}$ with the terms t_i sorted in ascending order according to the lengths of their posting lists, we consider different resolution strategies (throughout this work we name them S1, S2, S3 and S4).

In the basic conjunctive TAAT approach, (S1) we compute the result R in this way:

$$R = \cap_{i=1}^{n} \ell_i = (((\ell_1 \cap \ell_2) \cap \ell_3) \ldots \cap \ell_n) \tag{3.1}$$

This basically aims to compute the pairwise intersection of the smallest possible lists. For example, assuming a four-term query $q = \{t_1, t_2, t_3, t_4\}$, the final result would be computed as $(((\ell_1 \cap \ell_2) \cap \ell_3) \cap \ell_4)$.

However, in presence of an intersection cache, other strategies can be implemented. We propose three more methods. The first one (S2) splits the query in pairs of terms, as follows:

$$R = \cap_{k=1}^{n/2} (\ell_{2k-1} \cap \ell_{2k}) = ((\ell_1 \cap \ell_2) \cap (\ell_3 \cap \ell_4) \ldots \cap (\ell_{n-1} \cap \ell_n)) \tag{3.2}$$

Following the previous example, under this strategy the query q would be solved as $((\ell_1 \cap \ell_2) \cap (\ell_3 \cap \ell_4))$.

Another strategy we introduce (S3) also uses two-term intersections but it overlaps the first term of the i-th intersection with the second term of the $(i-1)$-st intersection as:

$$R = \cap_{i=1}^{n} (\ell_i \cap \ell_{i+1}) = ((\ell_1 \cap \ell_2) \cap (\ell_2 \cap \ell_3) \ldots \cap (\ell_{n-1} \cap \ell_n)) \tag{3.3}$$

The idea behind these strategies is to increase the probability to obtain cache hits. According to the S3 strategy, the query q would be solved as $((\ell_1 \cap \ell_2) \cap (\ell_2 \cap \ell_3) \cap (\ell_3 \cap \ell_4))$.

Using S1 the only chance of obtaining a hit in the intersection cache is to find $(\ell_1 \cap \ell_2)$ while S2 and S3 combine more terms in pairs that are tested against the cache. For example, using S2 in a four-term query we will look for (and may find) $(\ell_1 \cap \ell_2)$ and $(\ell_3 \cap \ell_4)$ in the cache. The obvious potential drawback is that S2 and S3 may require higher cost to compute more or more complex intersections (as in S2 and S3 strategies). However, this additional cost may be amortized by the gain obtained through more cache hits.

3.2.1 Another strategy using all possible cached information

The above policies check for cached intersections according to the execution order that is given by the length of the posting lists of the query terms. As we show in the previous example, strategy S1 only tests for one candidate intersection while S2 and S3 test for $\frac{n}{2}$ and $n-1$ intersections respectively. As we have just mentioned that redundancy may be positive in presence of intersection cache we consider this fact and give an additional step by developing another strategy that first tests all the possible two-term combinations in the cache in order to maximize the chance to get a cache hit. We call this strategy S4 and works as follows:

1. The query is first decomposed in all possible two-term combinations (or intersections, I_{ij}).
2. All intersections I_{ij} that are present in the cache are assigned to a candidates-set, C.
3. C is sorted according to the size of the intersections, from shortest to longest ($C\prime$).
4. The *new* query is built by picking first the intersections from $C\prime$.
5. The remaining terms (those that are not found in any cached intersection) are added to the query as in the S1 strategy.

Let us consider the following example to illustrate how S4 works. Given a query $q = \{t_1, t_2, t_3, t_4, t_5\}$, it is decomposed into the following intersections of the corresponding posting lists of each t_i:

$$(\ell_1 \cap \ell_2),\ (\ell_1 \cap \ell_3),\ \ldots,\ (\ell_3 \cap \ell_5),\ (\ell_4 \cap \ell_5)$$

Let us also assume that the cache contains the following intersections:

$$(\ell_1 \cap \ell_3),\ (\ell_1 \cap \ell_5),\ (\ell_3 \cap \ell_4) \text{ and } (\ell_4 \cap \ell_5)$$

and the relationship among their sizes (that defines $C\prime$) is given by:

$$|(\ell_1 \cap \ell_3)| < |(\ell_4 \cap \ell_5)| < |(\ell_1 \cap \ell_5)| < |(\ell_3 \cap \ell_4)|.$$

Under this configuration, the S4 strategy sets:

$$A \leftarrow (\ell_1 \cap \ell_3)$$
$$B \leftarrow (\ell_4 \cap \ell_5)$$

and rewrites the query as:

$$R = (A \cap B) \cap \ell_2$$

In this case, the search node only needs to manage (or fetch) the posting list of t_2 and compute the intersections. Notice that each time a query is evaluated, we need to check $\binom{n}{2}$ candidate pairs. This number is obviously a function of the maximum length of a query. However, the distribution of the number of terms in a query shows that 95% of queries have up to 5 terms. Despite this observation, we measured the computational cost overhead incurred doing this (across all the queries in a real log file) and we found that the increase in time is about 0.13% on average.

3.3 Experimentation Framework

In order to evaluate the techniques and algorithms that we propose in this work, we build a simulation framework and use real data for testing purposes. We consider the architecture introduced in Figure 1.1 and our aim is to emulate the resolution process when a query arrives at a search node. Given the massively parallel nature of query processing in search clusters [14], we restrict our work to a single search node.

We also assume that the collection is document-partitioned so each node holds a fraction of the documents and no interaction with other search nodes is required. We do not consider the query load on the system, assuming enough resources to compute each one. We simulate one of the nodes of a search engine running in a cluster using an Intel(R) Core(TM)2 Quad CPU processor running at 2.83 GHz and 8 GB of main memory.

The processing of a query in a search node would involve several aspects. First, the node should decompose the query into its single terms and check for the existence of the corresponding posting list in cache. Otherwise, it should fetch the posting lists from disk to the main memory. This operation is expensive and its impact should be minimized by the use of the cache. We name this cost C_{DISK}. In the case of conjunctive queries, partial results are obtained by the intersection of the lists with a cost of C_{CPU}. More specifically, the C_{CPU} cost involves decompressing the posting lists (as they are usually stored in a compressed form) and computing the set intersection to obtain a partial result. It is stated [10] that the sum of C_{DISK} and C_{CPU} are the major representatives of the

overall cost of a query. In the case that the posting list is present in cache C_{DISK} is omitted, thus reducing the total cost.

We use Zettair [17], a search engine written in C by the Search Engine Group at RMIT University[1], to index different document collections and to obtain real fetching times of the posting lists. According to [81], Zettair is one of the most efficient open source search engines due to its ability to process large amounts of information in considerable low times.

Zettair compresses posting lists using a variable-byte scheme with a B+Tree structure to handle the vocabulary. The cost of processing a query in a node is modeled in terms of disk fetch and CPU times: $C_q = C_{DISK} + C_{CPU}$. The first parameter, C_{DISK}, is calculated by fetching all the terms from disk using Zettair (we retrieve the whole posting list and measure the corresponding fetching time). In order to avoid the effect of hard disk caches and not to depend on the term stream we randomize the execution, run three trials and average the results. To calculate C_{CPU} we run a list intersection benchmark (all the operations were executed in the same machine). Our aim is to emulate a search node in a search engine cluster. According to Barroso et al. [14] this kind of cluster, used by a web-scale search engine like Google, is made up of more than 15,000 commodity class PCs. In our experiments we simulate one of the nodes of a search engine running in a cluster of that size using the hardware described above.

To validate our methodology we compare it against a real system implemented on the top of the Zettair search engine using a posting list cache and a small sample of 100,000 queries. The results lead to differences around 2.5% on average between the simulation and the real implementation and the correlation $R^2 = 0.9991$ which we consider acceptable. The results are drawn in Figure 3.1.

Given a document collection, we need to obtain the real resulting size of each intersection (t_i, t_j). To this aim, we execute the corresponding query using Zettair in conjunctive mode including the "AND" operator between each term (i.e. t_i AND t_j).

Besides, search engines may be built under two main assumptions regarding the physical location of the inverted index. Obviously, this also depends on the available resources and the scale of the search problem. In the case of web scale search engines, it is known that the major ones (i.e. Google, Yahoo!, Bing) store all posting lists in the main memory of their search nodes [32]. This approach totally eliminates the cost of disk access. However, smaller search applications still store a portion of the index in secondary memory (hard disk or SSDs [67]) and use different levels of cache. Throughout this work, we assume both scenarios but the main evaluated component becomes the Intersection

[1] http://www.seg.rmit.edu.au/zettair/

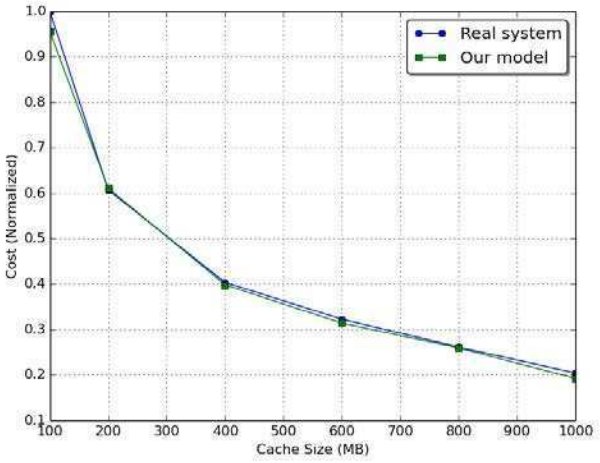

FIGURE 3.1: Comparison between the simulation methodology and a real system

Cache. In the case of memory-resident inverted indexes, the Posting List cache becomes useless (another view of the situation is to consider that all posting lists are cached) but an Intersection Cache may still be useful.

3.3.1 Document Collections

Throughout this work, we use three completely different document collections to assess the performance of the proposed techniques. The first one consists of a subset of a large web crawl of the UK domain obtained by Yahoo! in 2005. For the second dataset we crawl a subset of the terabyte-order collection provided by the Stanford University's WebBase Project[2] [51]. We select a sample crawled in March 2013, that contains about 7M documents. Finally, the third dataset corresponds to a sample of the Web Spam collection [26], a large set of web pages publicly distributed for research purposes (released in 2006). We refer to these collections as UK, WB and WS respectively. Table 3.2 shows basic statistics about the three datasets.

Dataset	Documents	Total Terms	Unique Terms	Raw Size GB	Index Size GB
UK	1,479,139	834,510,076	6,493,453	29.1	2.1
WB	7,774,632	9,143,511,516	110,838,794	241.0	23.0
WS	12,696,607	6,239,895,874	24,917,560	168.0	14.3

TABLE 3.2: Collection Statistics. "Raw Size" and "Index Size" correspond to the uncompressed documents in HTML format and the resulting (compressed) index respectively.

As we previously mentioned, the datasets are indexed using the Zettair search engine without stemming, stopword removal or positional information. Each entry in the posting

[2]http://dbpubs.stanford.edu:8091/testbed/doc2/WebBase/

lists includes both a docID and the term's frequency in that document (used for ranking purposes).

3.3.2 Query Logs

To evaluate the different proposals we use the well-known and widely used AOL Query Log [91] that contains around 20 million queries. This dataset has the benefits of being publicly available and corresponds to *real* user queries. However, the indexed collections do not belong to the same data source. Although Webber and Moffat [108] state that *"From an efficiency point of view, the semantic relevance of the queries to the indexed collection is not terribly important"* we try to minimize the mismatch between both documents and queries. First, we include two collections that are contemporary with the query log: the UK and WS collections were crawled in 2005 and 2006 respectively while the AOL Query log was released in 2006. Besides, we eliminate all queries that do not match the vocabulary of each document collection. We also include a newer collection (WB) because it contains longer documents that correspond to a recent snapshot of the web.

All queries are processed using standard approaches: terms are converted to lower case, stopwords are not eliminated and no stemming algorithm is applied. We select a sample of about 12 million queries and use different portions of this subset according to particular needs, mainly: computing statistics, warming-up of dynamic caches and testing the strategies.

We compute the query frequency distribution of the resulting dataset. We also generate all the possible pairs of terms (i.e. intersections) from the query log and compute their frequency distribution. As expected, we observe a Zipf-like distribution [8] (i.e. a power-law) in both cases. A power law distribution is defined as $f(i) = \frac{C}{r^{\gamma}}$, where $f(i)$ is the frequency of the i-th most frequent item and γ is the parameter of the distribution. This kind of distribution models phenomena where a small portion of the observed items are most often used, while the remaining items are used less often individually, thus forming a long-tail shaped curve. Figure 3.2 shows the data distributions and the corresponding fit curves.

Finally, we compute the total number of pairs compared to the number of full queries (Table 3.3) and the number of unique queries and pairs according to new instances appearing in the query stream.

Figure 3.3 shows the results. As expected, the number of unique pairs grows faster than the number of unique queries (we include one-term queries). We find that the

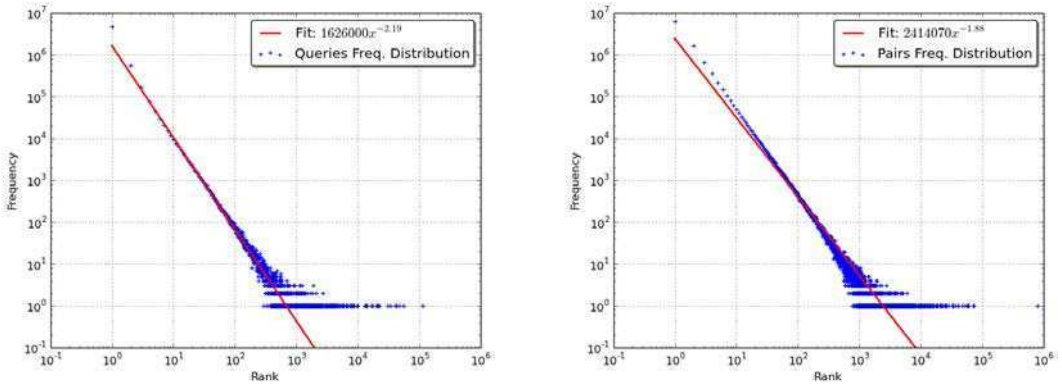

FIGURE 3.2: Frequency distributions of queries (left) and pairs of terms (right).

Queries		**Pairs**	
Total	**Unique**	**Total**	**Unique**
11,972,277	5,722,691	42,301,175	10,011,656

TABLE 3.3: Number of total and unique queries and pairs in the query log.

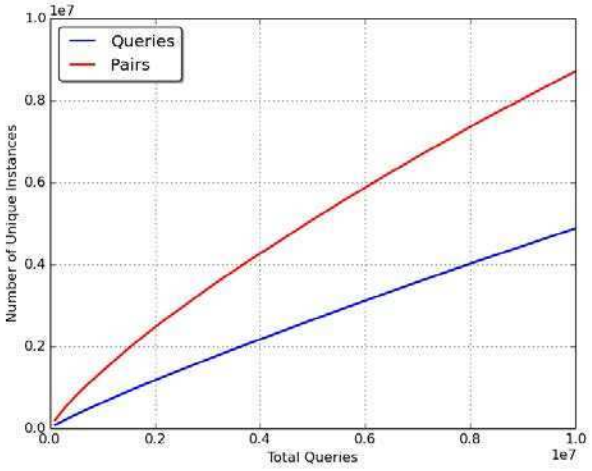

FIGURE 3.3: Growth of unique queries and pairs according to new instances appearing in the query stream.

proportion of singleton queries (i.e. those queries that appear only once) is about 40% while the proportion of singleton pairs is about 14.8% which enables more opportunities to benefit from intersection caching.

Políticas y Estrategias para Caching de Intersecciones - Resumen

Como se mencionó anteriormente, los motores de búsqueda requieren de diferentes niveles de caché por cuestiones de eficiencia y escalabilidad. Las políticas usadas para manejar los diferentes cachés habitualmente se basan en factores como frescura, frecuencia, tamaño, costo, tiempo de modificación y expiración, principalmente.

En este trabajo, como una contribución básica, se propone el uso específico de políticas que consideren el costo (*cost-aware*) en el caché de Intersecciones, tanto para casos estáticos, dinámicos e híbridos. En una segunda parte, se diseñan estrategias específicas de resolución de la consulta que consideran la existencia del caché de intersecciones.

La tarea de completar un caché con un conjunto estático de ítems se puede ver como una variante del problema de la mochila (*knapsack*), modelo utilizado previamente para caching de Listas. Aquí, se cuenta con un espacio predefinido para caching y un conjunto de ítems con tamaños y costos asociados. El desafío es mantener en memoria los ítems más valiosos con el fin de maximizar el beneficio que éste ofrece. Este nivel de caché maneja eficientemente los ítems populares a largo plazo pero no las ráfagas (*bursts*) de ítems inexistentes. Para ello, un caché dinámico ofrece una mejor aproximación, de acuerdo a los patrones de repetición que van ocurriendo en el flujo de consultas.

De esta manera, las políticas híbridas combinan lo mejor de ambos mundos. Por un lado, mantienen una porción estática completada de acuerdo a patrones de uso y, por otro lado, una porción dinámica maneada por una política de reemplazo. La política híbrida clásica se llama SDC (Static and Dynamic Cache) que utiliza LRU en la parte dinámica, mientras que la porción estática contiene los ítems más frecuentes.

En cuanto a las estrategias de resolución de consultas, la estrategia clásica para casos conjuntivos (AND) se basa en un enfoque Term-at-a-Time. Cabe mencionar que el caché

de intersecciones solo tiene sentido en el caso de este tipo de consultas. Dada una consulta $q = \{t_1, t_2, t_3, ..., t_n\}$, básicamente, consiste en resolver la intersección $\bigcap_{i=1}^{n} \ell_i$ (donde ℓ_i corresponde a la lista invertida de t_i) procesando las dos listas más cortas primero y así sucesivamente (denominada aquí como estrategia S1).

Sin embargo, en este trabajo se proponen otras estrategias que se basan en la existencia de un caché de intersecciones. La siguiente (S2) consiste en dividir la consulta en pares de términos como: $R = \cap_{k=1}^{n/2}(\ell_{2k-1} \cap \ell_{2k})$. Una estrategia posterior (S3) se basa en superponer el primer término de la i-ésima intersección con el segundo término de la *(i-1)*-ésima intersección como: $R = \cap_{i=1}^{n}(\ell_i \cap \ell_{i+1})$.

La idea detrás de estas estrategias es aumentar las chances de obtener un acierto en el caché de intersecciones, al costo de cierto overhead que se introduce durante la construcción del plan de resolución. Finalmente, una última estrategia (S4) descompone la consulta en todos los pares posibles y los chequea contra en caché. Luego, a partir de las intersecciones que se encuentran en caché rearma la consulta, considerando éstas primero y luego los términos restantes.

A lo largo de este trabajo, todas las políticas estáticas, dinámicas e híbridas son evaluadas para los cuatro casos de estrategias de resolución (S1, S2, S3 y S4), donde S1 es la estrategia de referencia (baseline) contra la cual se comparan los demás enfoques. La evaluación se realiza en un entorno de simulación utilizando datos reales. Concretamente, se utilizan tres colecciones de documentos públicos obtenidos de la web y un conjunto de consultas (*query log*) ampliamente utilizado por la comunidad de investigación de la temática.

Chapter 4

Cost-Aware Intersection Caching

In the previous section we have explained how the list intersections are computed and how different caching policies decide which intersections are kept in the cache. The task of filling the cache with a set of items that maximize the benefit in the minimum space can be seen as a version of the well-known knapsack problem, as discussed in previous work related to list caching [10]. Here, we have a limited predefined space of memory for caching and a set of items with associated sizes and costs. The number of items that fit on the cache is related to their sizes, so the challenge is to maintain the most valuable ones in the cache.

In this section we introduce a study of cost-aware intersection caching using the four query resolution strategies described in Section 3.2 and different combination of caching policies. Basically, we consider different cost-aware cache replacement policies adapted to our problem and compare these against common cost-oblivious policies used as a baseline.

As we mention in Section 2.4.3 the idea of caching intersections is first introduced in [70]. This is the closest work related to ours. In that paper, the authors propose the use of an intersection cache that adds a third level to the cache hierarchy (cache of Results at the broker side and caches of Intersections and Posting Lists at the search node side). This *new* cache contains common postings of frequently occurring pairs of terms implemented as *projections* of each term. Given a pair of terms t_i and t_j, the projection of the inverted list of t_i in t_j, denoted as $t_{i \to j}$, contains all postings in the list of t_i whose document identifiers also appear in the list of t_j. For example, given t_i and t_j and their respective posting lists:

$$\ell_i = \{(1,2);(3,5);(9,13);(10,1)\}$$
$$\ell_j = \{(3,1);(4,3);(10,2);(11,5);(15,3)\}$$

where each pair (x, y) denotes the document identifier and the corresponding frequency of term t_i in the document, the obtained projections become:

$$t_{i \to j} = \{(3, 5); (10, 1)\}$$
$$t_{j \to i} = \{(3, 1); (10, 2)\}$$

This representation is equivalent to handling the final intersection as:

$$(t_i \bigcap t_j) = \{(3, 6); (10, 3)\}$$

The disadvantage of using projections is that the two projections are larger than a single intersection as the document identifiers are stored twice. An important difference with respect to our work is that their intersection cache resides in secondary memory. In that case, a certain amount of extra disk space (20-40% of the index size) is reserved for caching common postings of frequently occurring intersections. This decision also softens the space penalty due to store projections. In our case, the intersection cache resides in the main memory so we decide to store the final intersection (instead of the projection) to use the memory space efficiently.

Another interesting related study to mention is the work of Ding et al. [35] where the authors propose some efficiency optimizations for batch query processing in web search engines. They identify possible applications of batched search engine queries and propose different techniques to process queries more efficiently. One of these techniques consists of the reuse of partial results (i.e. intersections). Basically, they rewrite the query stream following a grammar and automatically cache the sub-queries in order to reuse them. In this way, they take advantage of locality and can make use of a clairvoyant cache eviction policy (since all future queries are known) thus improving caching performance. The cost of query processing is modeled as the sum of the inverted list length for all the terms in the query.

4.1 Description of Intersection Costs

Our main goal is to minimize the total cost of solving a query taking into account the existence of the intersection cache. If we consider that cache misses have different costs and the size of the resulting lists of two-term intersections are non-uniform we introduce these parameters in different caching policies to determine which are more useful for cost saving. The first observation is described in detail in [88] for full queries while the second one is completely different because the caching of results realistically assumes

uniform sizes for the lists (i.e. the top-10 or top-30 results). In our case, we can see each intersection as a two-term query so we perform a cost analysis to understand the variations with respect to its frequency. We sample 1 million intersections and compute their cost and frequencies. Figure 4.1 (left) shows that intersection costs and frequency are related following a certain pattern such as a power-law distribution. The Cumulative Distribution Function (Figure 4.1, right) shows that around 75% of the intersections are limited to a two-order magnitude interval.

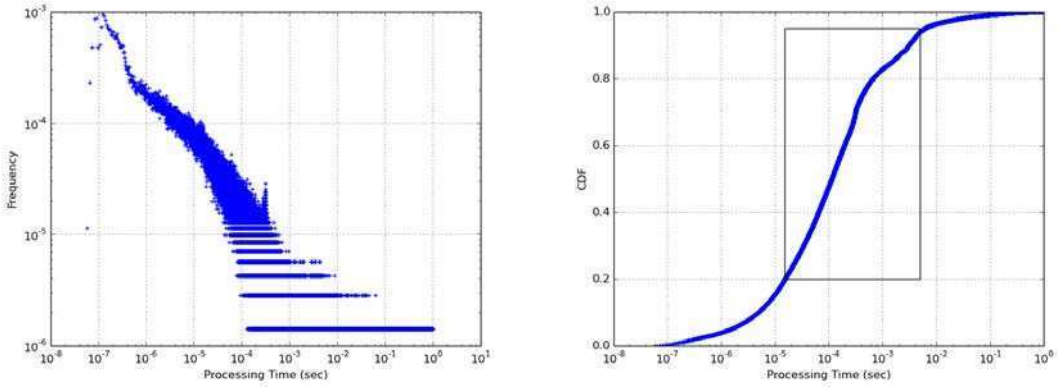

FIGURE 4.1: Frequency and cost of sampled intersections (left, both axis in log scale). Cumulative Distribution Function for the frequency distribution (right, x-axis in log scale).

From another point of view, we analyze the frequency distribution of the resulting intersection size (i.e. the length of the resulting set after intersecting two-term lists). Figure 4.2 shows the distributions. The frequency distribution of the intersection sizes follows a skewed distribution and the CDF shows that around 65% of the resulting lists are limited to a three-order magnitude interval.

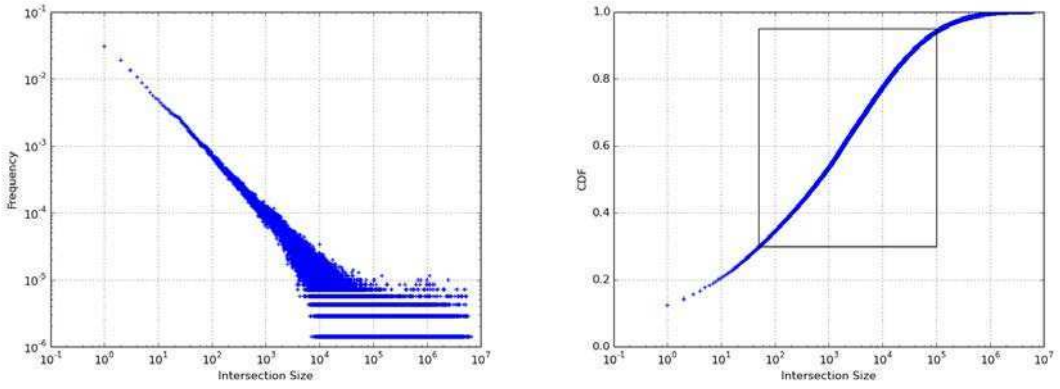

FIGURE 4.2: Frequency and intersection-size of sampled intersections (left, both axis in log scale). Cumulative Distribution Function for the frequency distribution (right, x-axis in log scale).

To benefit from these distributions we explore the possibility of explicitly incorporating the costs of the intersections into the caching policies. Our complete research covers two scenarios:

(a) The inverted index residing on disk: in this case, our aim is to determine the usefulness of the intersection cache comparing the different query resolution strategies and the proposed cost-aware caching policies (against cost-oblivious ones).

(b) The whole index residing in the main memory: here, we analyze another scenario such as the case of search engines that store the full inverted index in the main memory of the search nodes.

4.1.1 Data and Setup

We provide a simulation-based evaluation of the proposed strategies and caching policies. The total amount of memory reserved for the intersection cache ranges from 1 to 16 GB (that can hold about 40% of all possible intersections of our dataset). For each query (and each intersection strategy), we log the total cost incurred using the cache in each case rather than the hit rate (commonly used to evaluate cost-oblivious policies).

To evaluate the intersection cache we use the UK document collection (Section 3.3.1) and a subset of the AOL query log (Section 3.3.2). We use 2 million queries to compute statistics for filling static caches. In the case of dynamic replacement policies we warm up the cache with roughly 4 million queries and reserve the remaining queries as the test set (3.9 million queries).

4.2 Intersection Caching with the Index in Disk

We experimentally evaluate the cache replacement policies described in Section 3.1 and the four processing strategies (S1, S2, S3 and S4) introduced in Section 3.2. We show that some of the combinations may lead to significant improvements in the performance at search-node level. We first report the results of intersection caching when the inverted index resides in disk for the aforementioned five cache sizes (1 to 16 GB).

4.2.1 Static Policies

We consider two cost-oblivious strategies as a baseline. The first one is FB, which sorts the intersections according to frequency (in descending order) and FS, which computes

the $\frac{Freq}{Size}$ ratio to select the items to keep in cache. This is the most competitive policy for list caching as reported in [10] to improve the cache hit rate. Figures 4.3, 4.4, 4.5 and 4.6 show these results. To allow a fair comparison among all policies and processing strategies we normalize all the values with respect to the performance obtained by the S1 policy running without an intersection cache.

The first interesting observation is that S2 and S3 outperform S1 which means that some redundancy on the query processing strategy is useful to reduce computing time. As we expected, S4 is the best strategy. Abusing notation, we find that the ordering of the remaining policies is S1 > S2 > S3 on average. We also observe that CB (cost-based) policy performs worse than the others. This is explained by the fact that the most costly intersections are not necessarily frequent enough to be repeatedly used. This policy also uses a huge amount of cache space because its items are, in general, the result of fetching and intersecting long posting lists. The big picture shows that the best policies offer a tradeoff among frequency, cost and size.

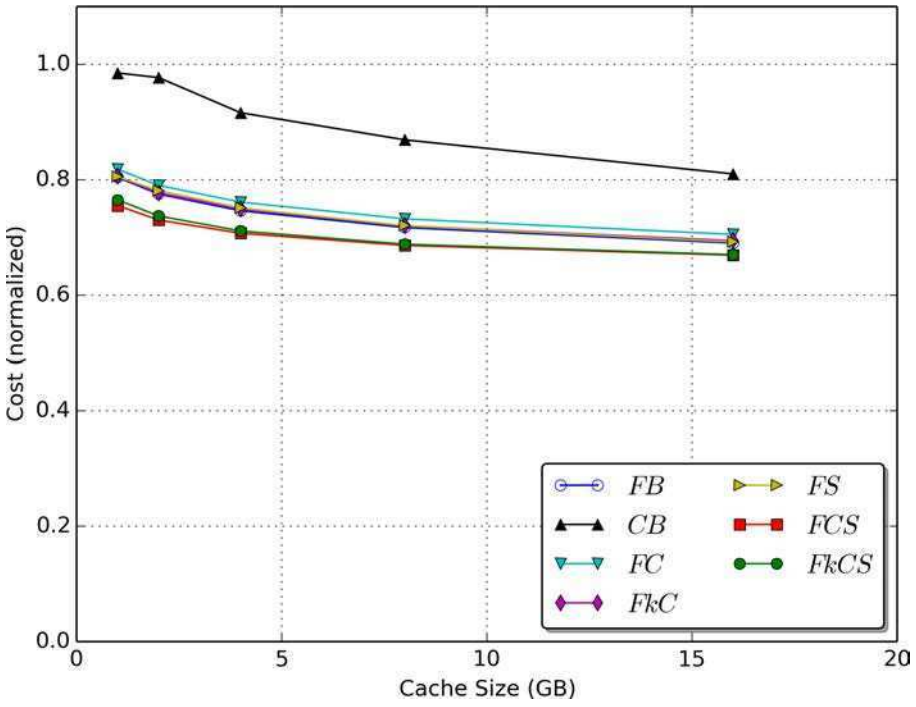

FIGURE 4.3: Total cost incurred by the S1 processing strategy for the seven static cache policies.

In all cases, FCS and FkCS (with parameter $k = 1.5$, experimentally tuned) become the best policies to populate the static cache. Considering these two policies, the average cost reduction with respect to the baseline is 5.0%, 14.7%, 36.8% and 39.1% for S1, S2, S3 and S4 respectively. The comparison among strategies shows that S2 is 20.2% better than S1, S3 is 37.2% better than S2, and finally S4 is 42.4% better than S3. Figure 4.7 illustrates the best static policies (FCS and FkCS) for S1, S2, S3 and S4 strategies.

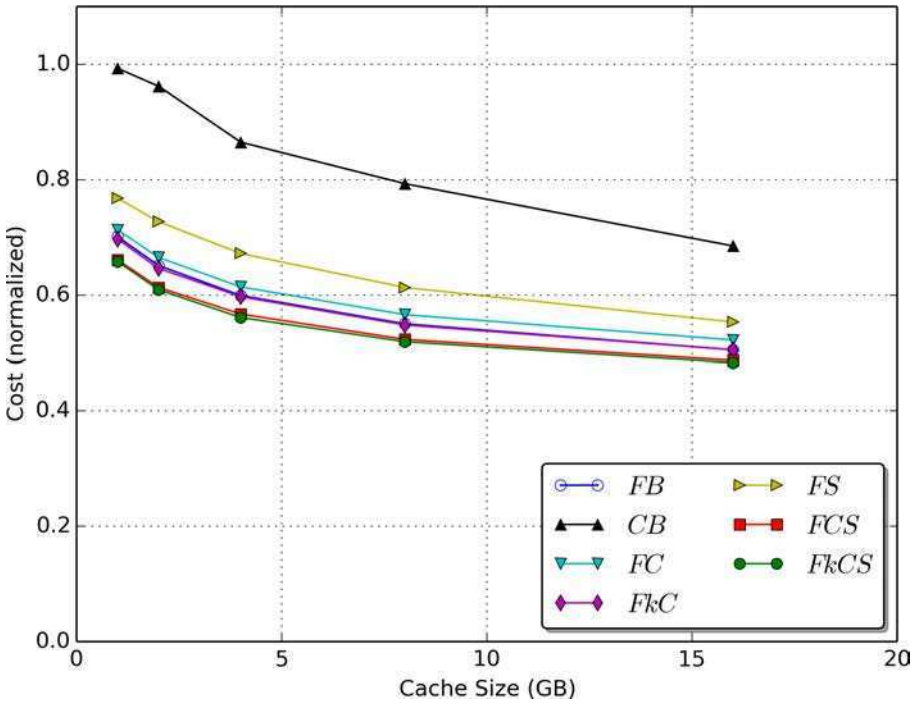

FIGURE 4.4: Total cost incurred by the S2 processing strategy for the seven static cache policies.

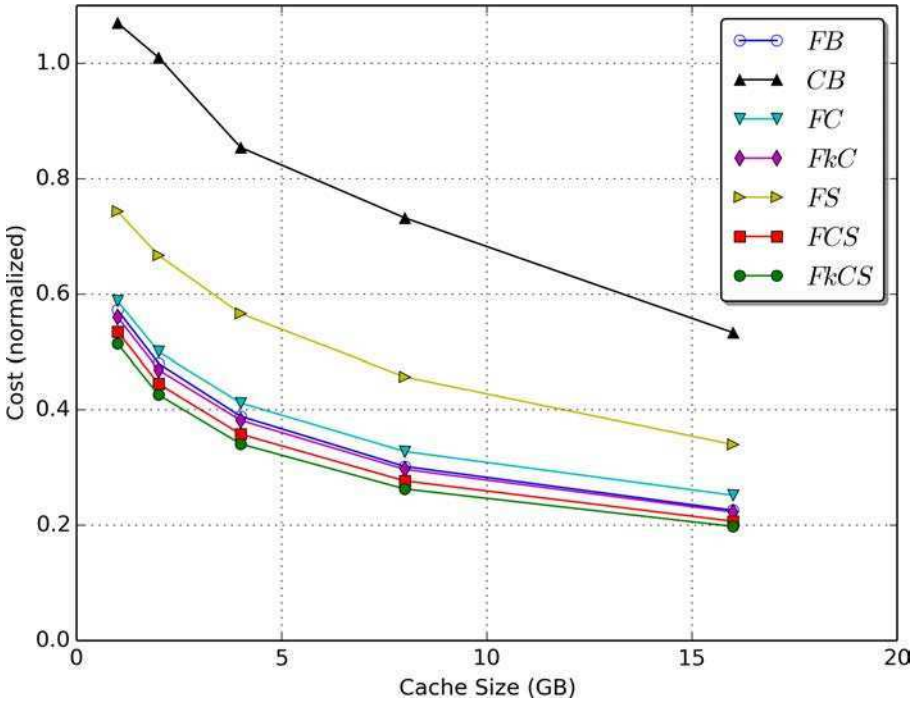

FIGURE 4.5: Total cost incurred by the S3 processing strategy for the seven static cache policies.

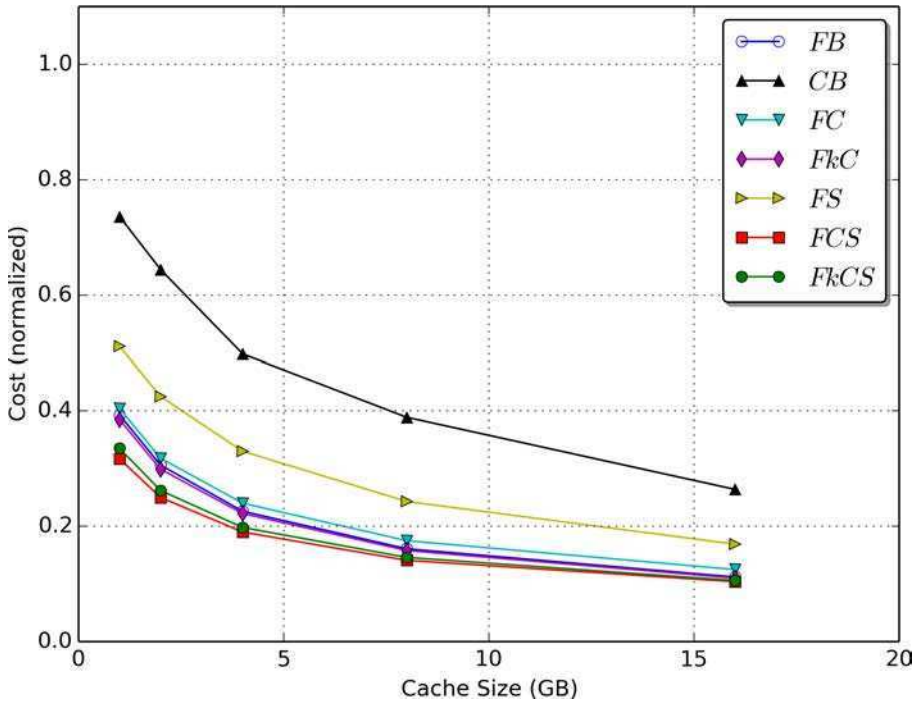

FIGURE 4.6: Total cost incurred by the S4 processing strategy for the seven static cache policies.

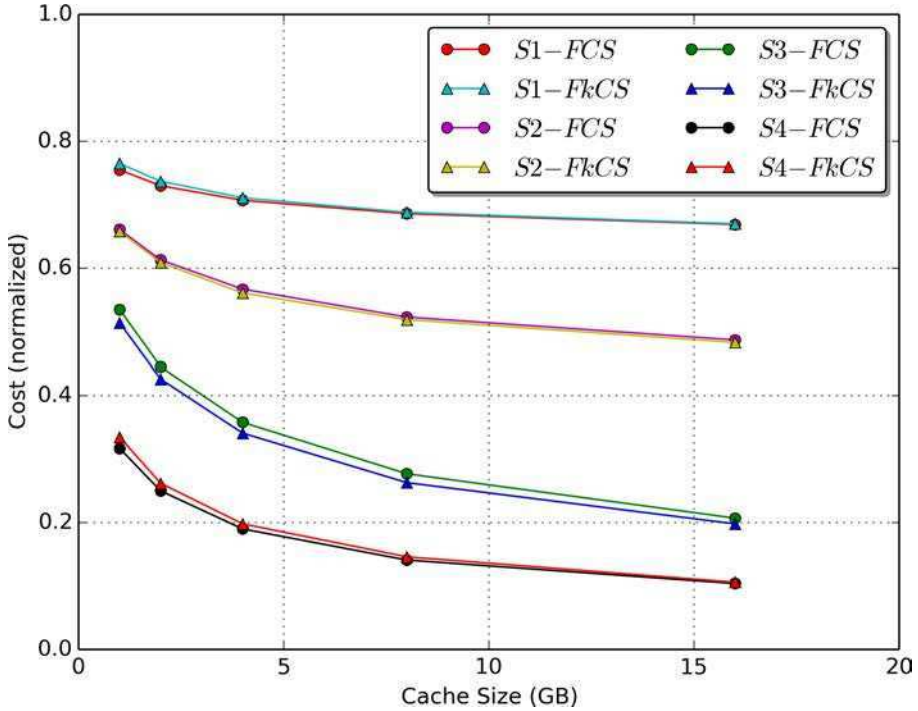

FIGURE 4.7: Comparison of total costs incurred by the two best static policies and the four processing strategy.

4.2.2 Dynamic Policies

We carry out a similar study for dynamic policies. In this case, we consider LRU and LFU as the baseline strategies. These two caching policies are popular examples of cost-oblivious strategies that only consider recency and frequency of the items in the cache respectively. LRU is a widely used strategy and often considered as a baseline to compare against [115]. These results are reported in Figures 4.8, 4.9, 4.10 and 4.11. We use the same normalization as in the previous case.

The first observation shows that the global ordering of the policies is similar to the static cases, as expected (S1 > S2 > S3 > S4). In the case of S1, the Landlord policy performs similarly to LRU (points are overlapped in the picture) but the GDS policy outperforms them by around 15.0% on average. The other cost-aware policies perform poorly and do not offer cost savings. The most important observation is that the best static policy (FCS) combined with the S4 strategy outperforms the best dynamic policy (GDS with S4) in about 8.0%. This result is similar to that reported in [10] for list caching in which the static QTF/DF algorithm performs around 10.0% better than dynamic ones.

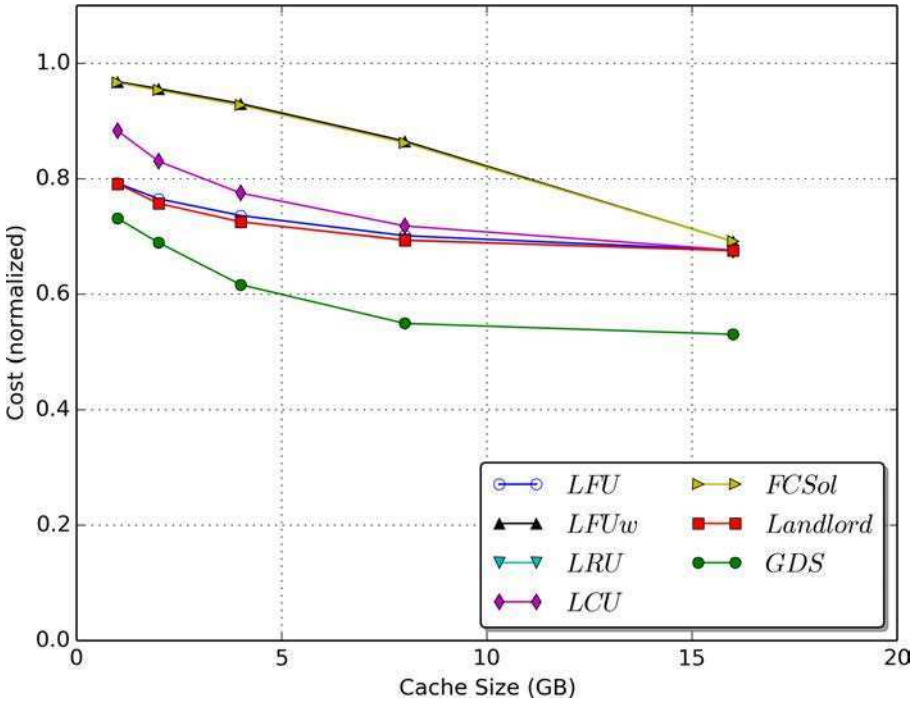

FIGURE 4.8: Total cost incurred by the S1 processing strategy for the seven dynamic cache policies.

Considering the best replacement policy in each strategy (GDS), S2 is 27.3% better than S1, S3 is 28.6% better than S2, and finally S4 is 32.9% better than S3. Figure 4.12 shows this comparison for the two best dynamic intersection caching policies.

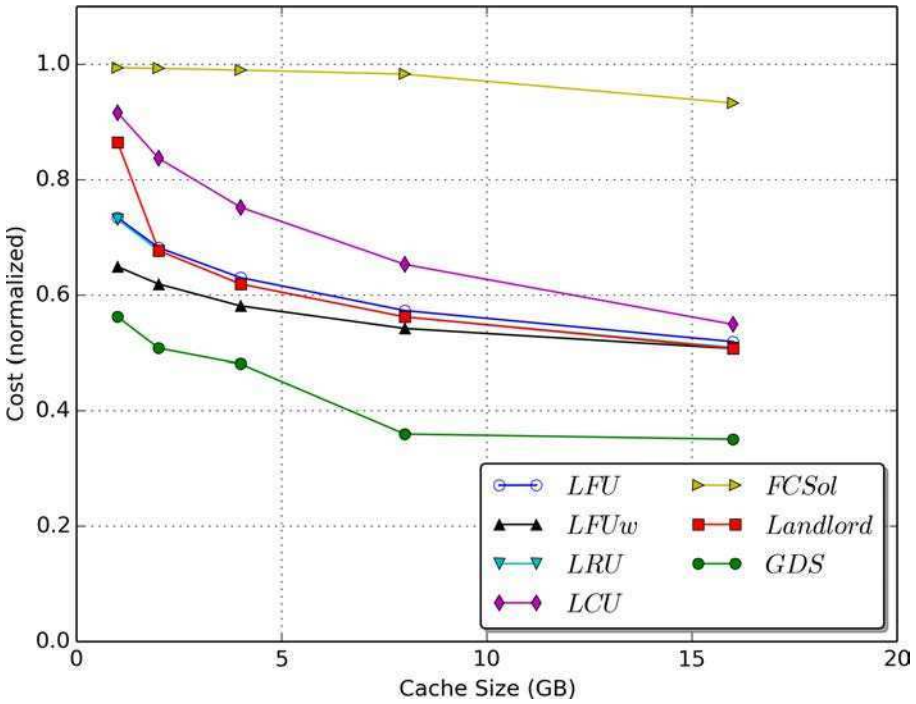

FIGURE 4.9: Total cost incurred by the S2 processing strategy for the seven dynamic cache policies.

We also extend this experiment to three-term intersections (Figure 4.13). The results show a slight improvement in the performance of about 3.2%, 1.5% and 5.2% for LRU, Landlord and GDS policies respectively.

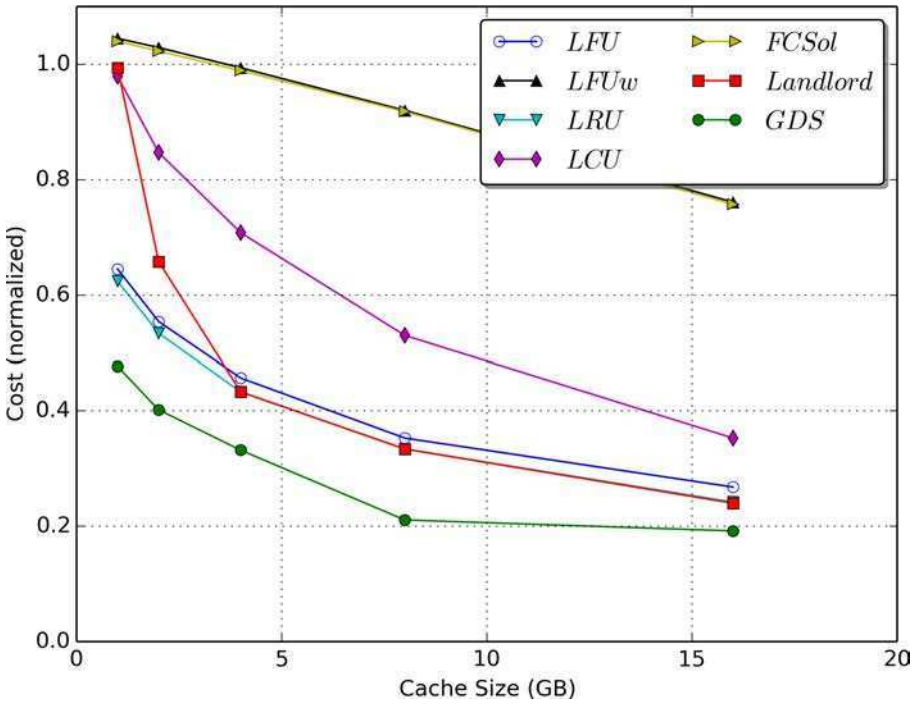

FIGURE 4.10: Total cost incurred by the S3 processing strategy for the seven dynamic cache policies.

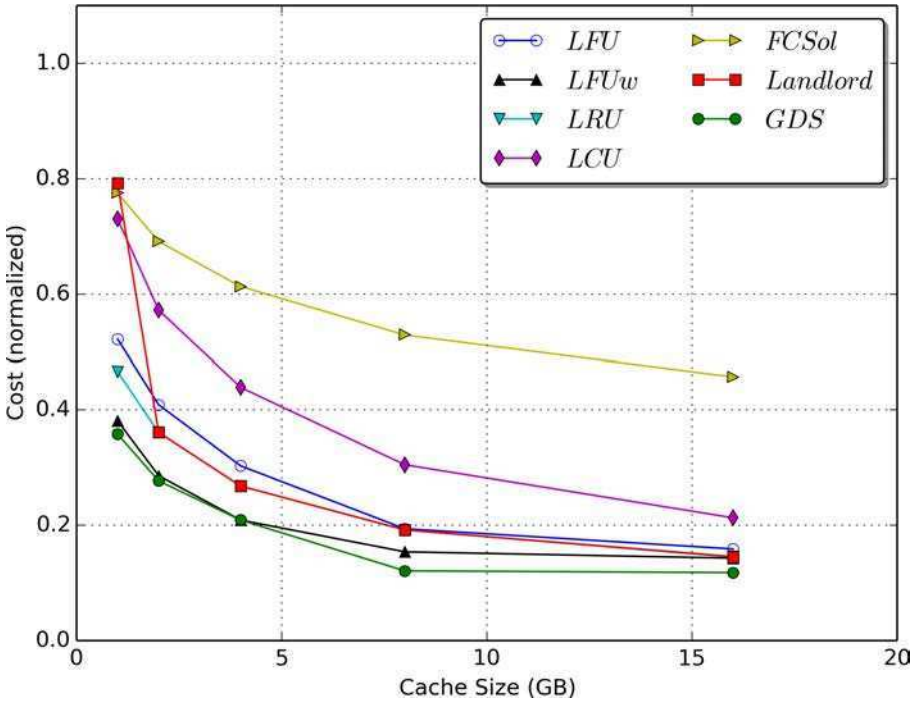

FIGURE 4.11: Total cost incurred by the S4 processing strategy for the seven dynamic cache policies.

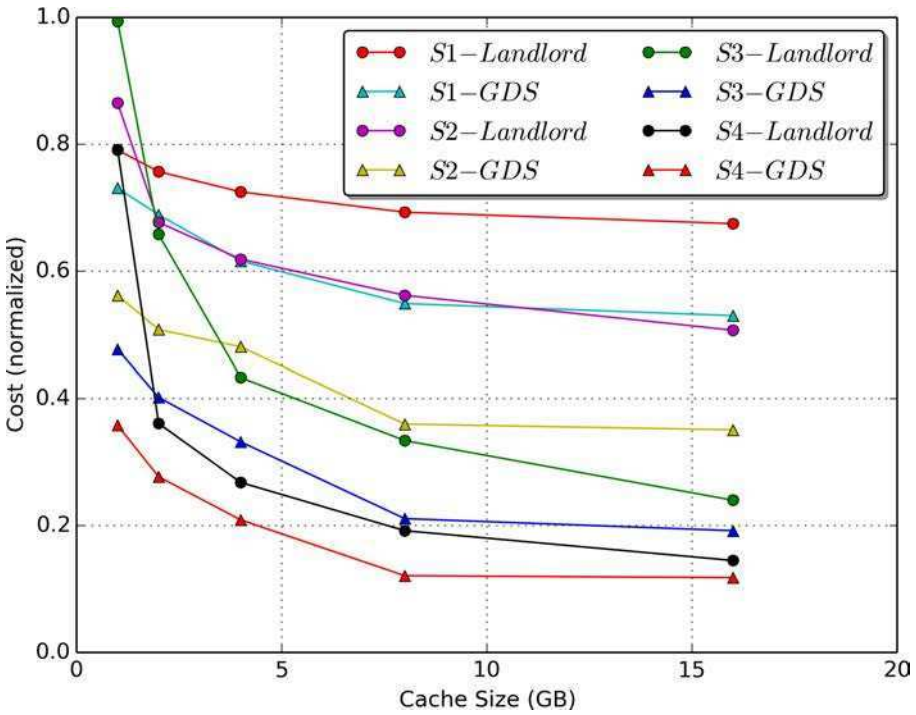

FIGURE 4.12: Comparison of total costs incurred by the two best dynamic policies and the four processing strategy.

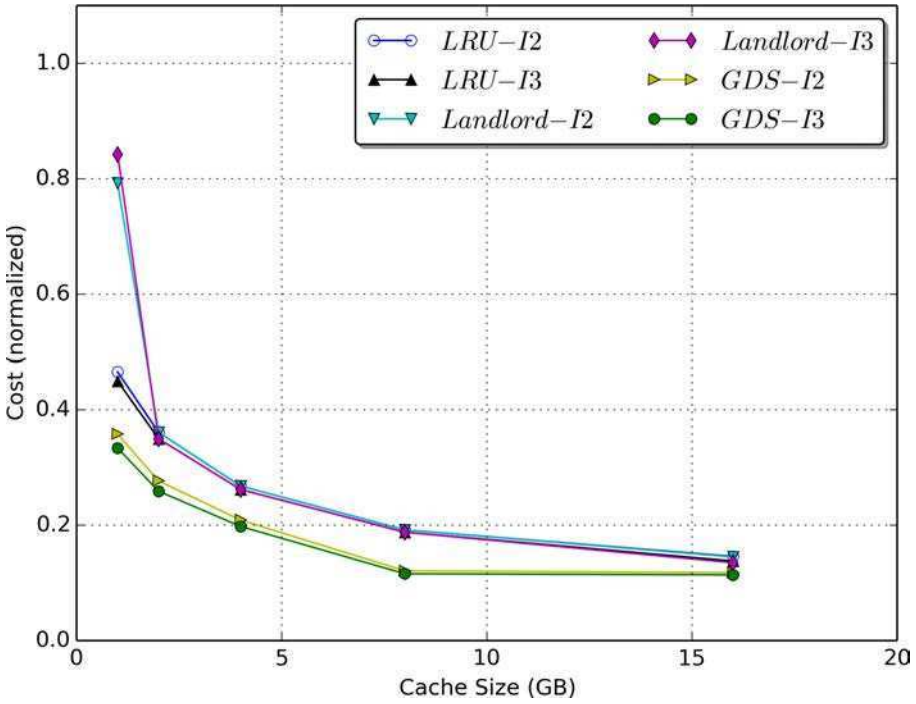

FIGURE 4.13: Total cost incurred by the S4 strategy enhanced with three-term intersections (I3) for dynamic policies.

4.2.3 Hybrid Policies

Finally, we evaluate different combinations of hybrid policies. We consider SDC as a baseline and a variation of this with the static part filled with items selected according to the FCS strategy (this is the most competitive policy for static caching when using the S4 strategy). We split the cache in a 80/20 allocation proportion, that is, 80% of the space is allocated for the static part and the remaining 20% for the dynamic part. This is a suggested splitting point in the literature [38] that performs well in our case, indeed. However, the determination of the best splitting point is still an open question that deserves a deeper research. We use the same normalization as in the previous cases. Figures 4.14, 4.15, 4.16 and 4.17 show the results.

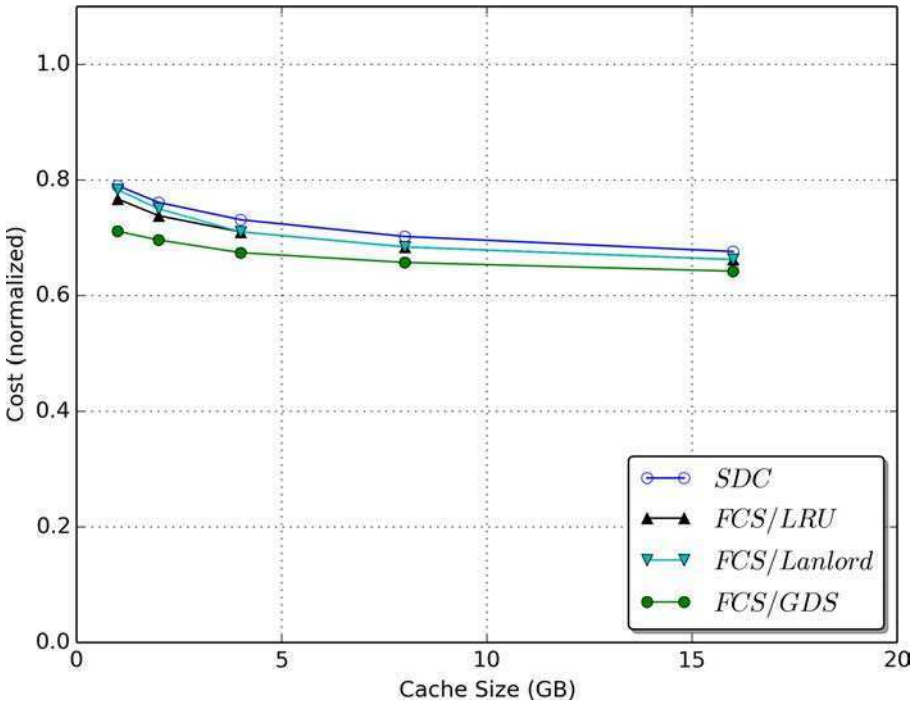

FIGURE 4.14: Total cost incurred by the S1 processing strategy for the four hybrid cache policies.

As the first observation, we find that the best dynamic policy (GDS) outperforms hybrid ones for S1, S2 and S3 strategies. However, the S4 strategy becomes the most efficient one, achieving the highest costs savings. The hybrid FCS/GDS cache replacement policy combined to the S4 strategy outperforms GDS in about 8.0% on average (up to 17.5%) and FCS in about 1.0% (up to 2.5% in the best case). This policy is precisely a combination of the best static policy (FCS) with the best dynamic policy (GDS). In practice, FCS and FCS/GDS perform similarly but the hybrid version may adapt to some slight changes in the query stream patterns.

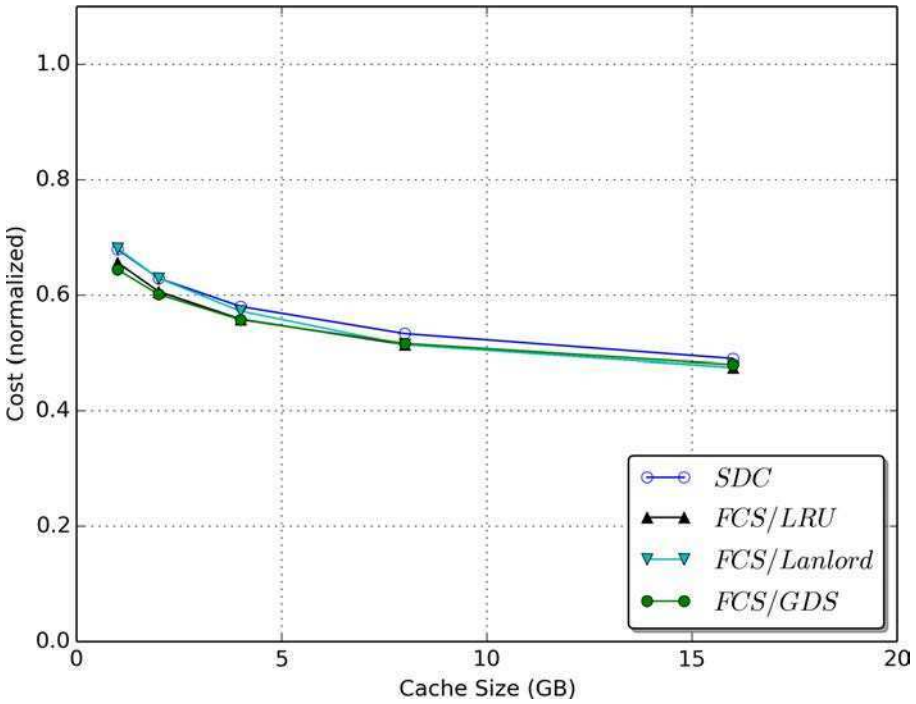

FIGURE 4.15: Total cost incurred by the S2 processing strategy for the four hybrid cache policies.

We also compare the best hybrid caching policies combined with each resolution strategy. Figure 4.18 shows the resulting performance. Again, S4 becomes the best strategy, achieving a performance improvement (on average) of about 44%, 67% and 78% with respect to S3, S2 and S1 strategies, respectively.

Finally, we also extend this experiment considering three-term intersections too (Figure 4.19). The results show an interesting improvement in the performance around 5.0% on average in the three cases.

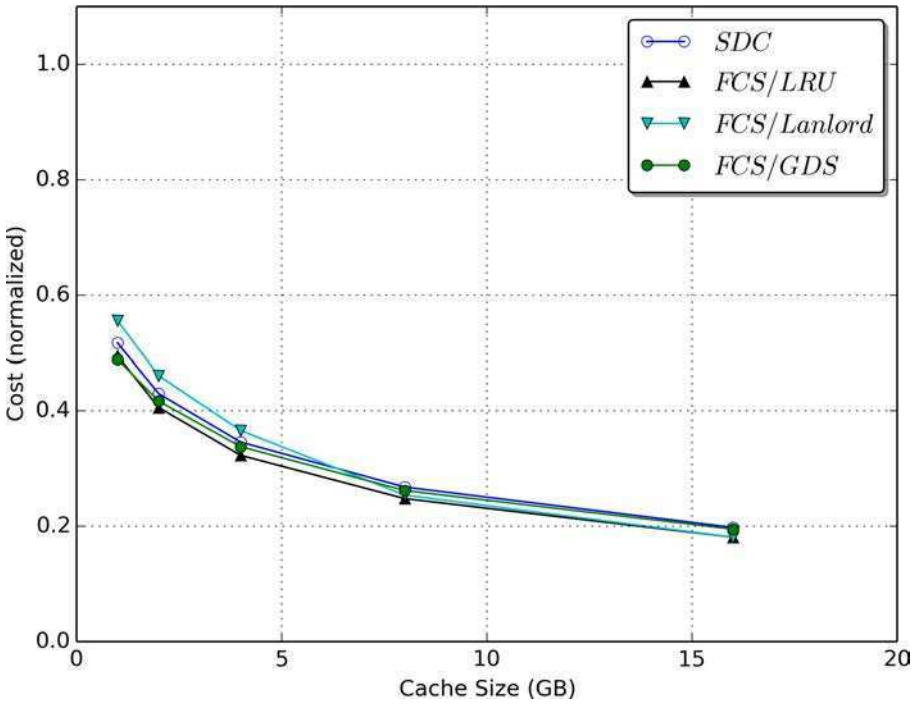

FIGURE 4.16: Total cost incurred by the S3 processing strategy for the four hybrid cache policies.

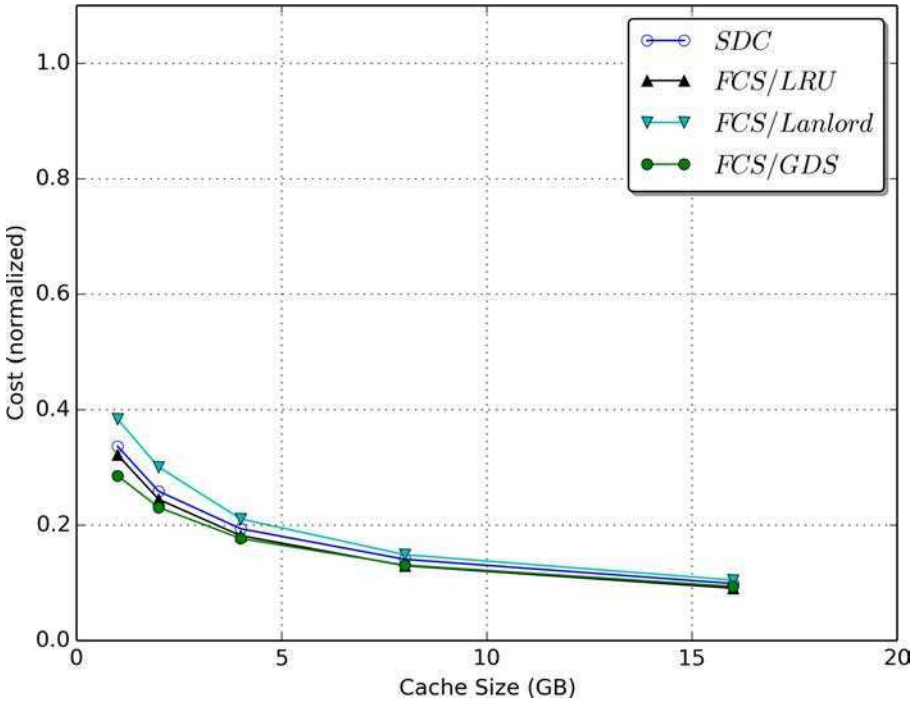

FIGURE 4.17: Total cost incurred by the S4 processing strategy for the four hybrid cache policies.

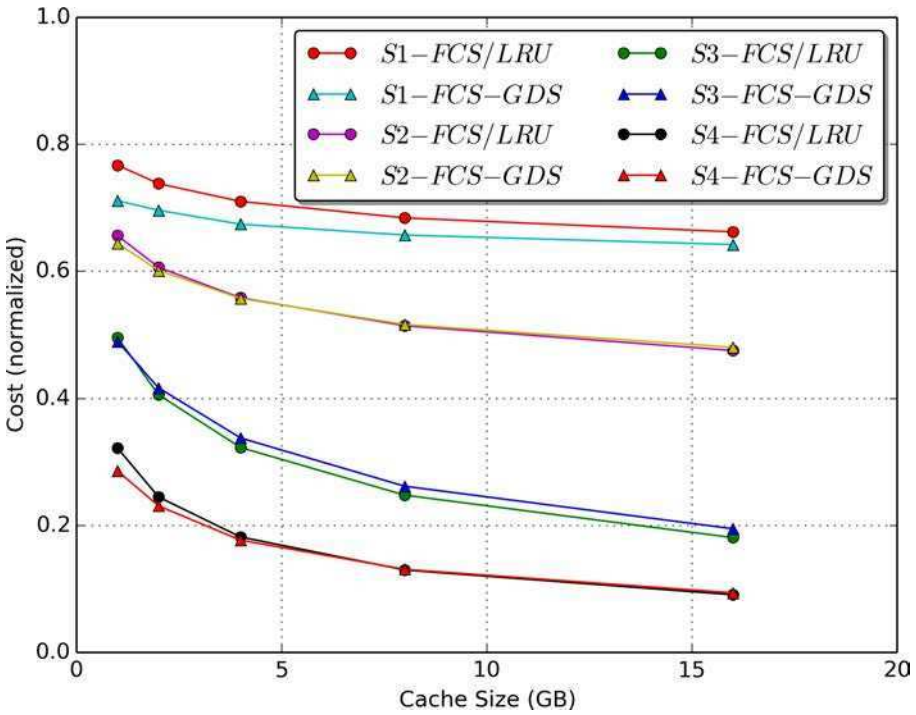

FIGURE 4.18: Comparison of total costs incurred by the two best hybrid policies and the four processing strategy.

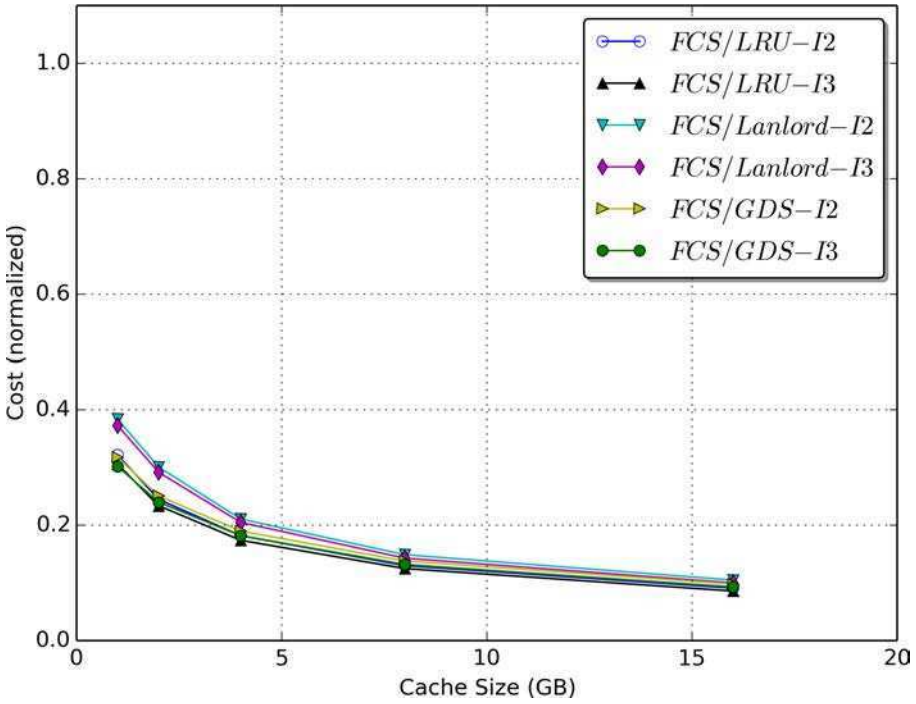

FIGURE 4.19: Total cost incurred by the S4 processing strategy enhanced with three-term intersections (I3) for hybrid policies.

4.2.4 Comparing Variances

In previous sections we show the reduction of the average processing time by the use of cost-aware intersection caching policies. In this section we compare the variance of the cost distributions for some selected configuration. This is particularly interesting in parallel systems (such as the case of a search engine) because the slowest machine becomes the bottleneck in the query resolution process. This comparison allows us to evaluate whether the new strategies and policies incur in higher variability in the corresponding query cost distributions, or not.

To this aim, we compute the variance values for different configurations according to the cache size increases. We select the basic S1 strategy and the baseline caching policy to compare them against the best strategy (S4) and the cost-aware caching policy that achieves the best performance (for static, dynamic and hybrid cases, respectively). Figures 4.20, 4.21 and 4.22 show the results.

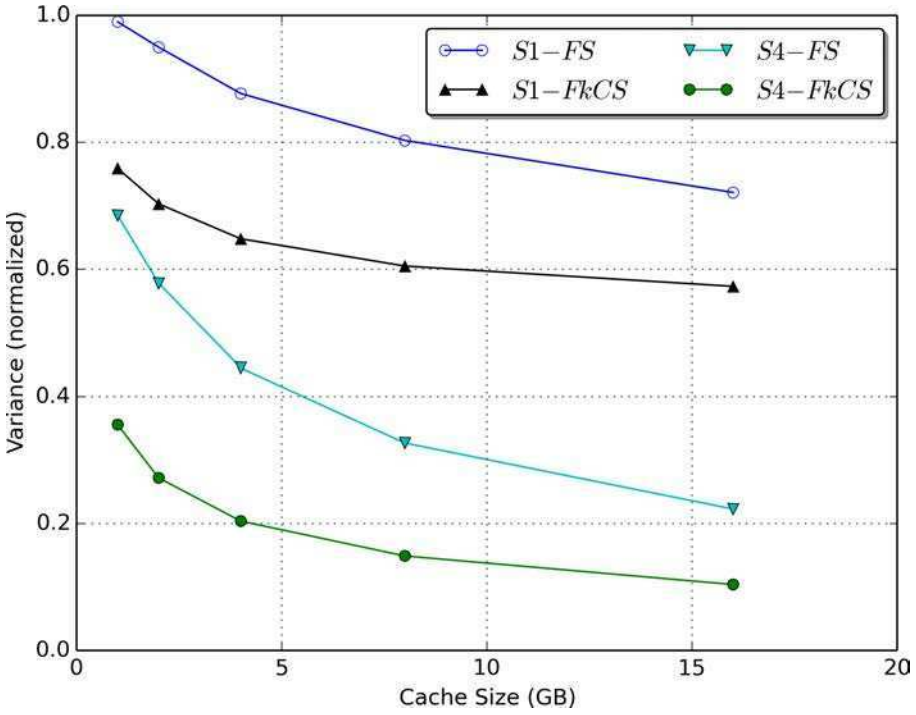

FIGURE 4.20: Comparison of variance for the baselines and best static policies and strategies.

In all cases, the results show a reduction of the variance in the cost distributions in both the caching policy (baseline vs cost-aware) as in the query resolution strategy (S1 vs S4). In the case of static policies (FS vs FkCS), the variance reduction is about 38% (on average). When comparing strategies (S1 vs S4) the reduction is about 58%. A similar behaviour occurs for dynamic policies: GDS reduces the variance in about 29% (vs LRU) while a reduction close to 70% is observed when comparing S4 and S1. Finally, hybrid

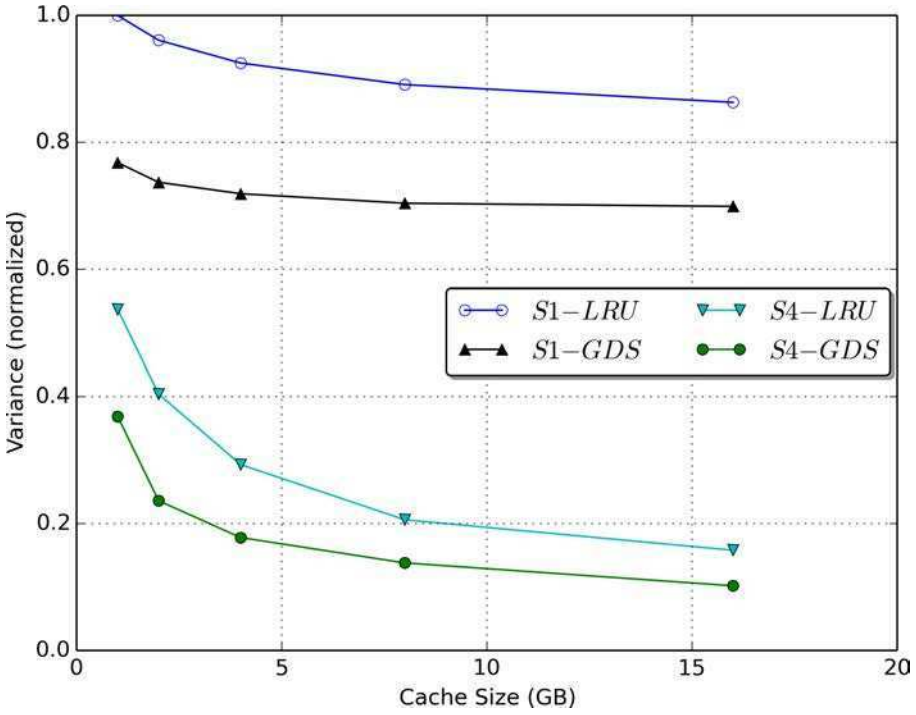

FIGURE 4.21: Comparison of variance for the baselines and best dynamic policies and strategies.

policies perform similarly. A reduction of about 26% and 71% is observed for caching policies (SDC vs FCS/GDS in this case) and query resolution strategies, respectively.

We also run the Levene's test [66] for equality of variances. This is a non-parametric statistical tool used to assess the equality of variances. The main idea is to test the null hypothesis that the samples variances are equal. If the resulting *p-value* of the test is less than some significance level (typically 0.05), the obtained differences in the variances are unlikely to have occurred based on random sampling and there is a difference between the variances in both distributions. We select the two extreme configurations of cache size (1GB and 16GB) for each type of policy (static, dynamic and hybrid). In all cases we obtain a $p\text{-}value < 0.0001$ which means that the compared distributions do not correspond to a random sampling of the same distribution.

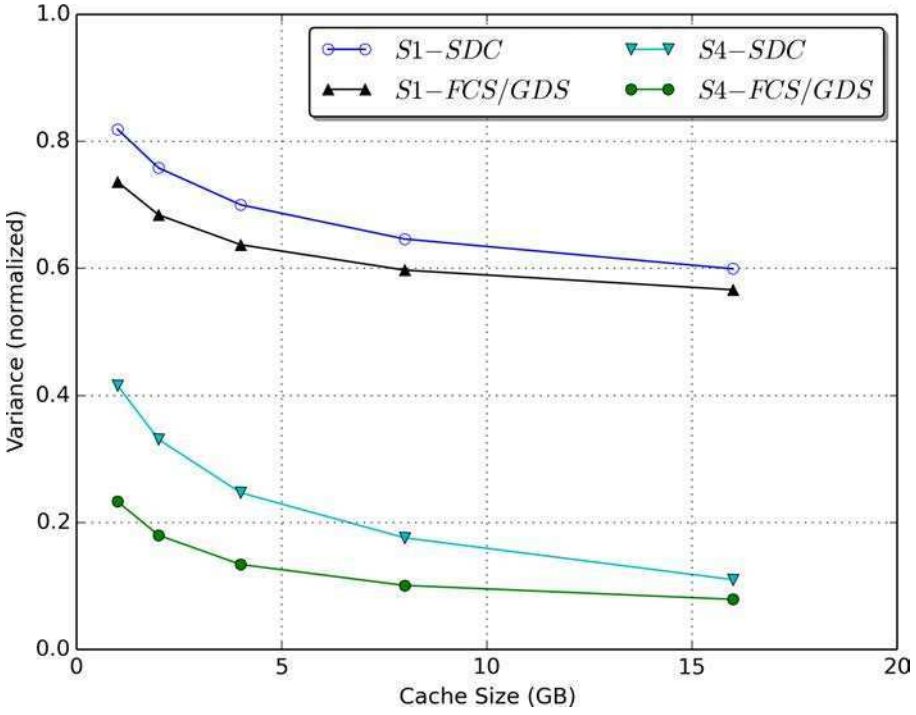

FIGURE 4.22: Comparison of variance for the baselines and best hybrid policies and strategies.

4.3 Intersection Caching with the Index in Main Memory

It is known that some major search engines store the whole inverted index in the main memory, which leads to eliminate the cost of disk access [32]. Moreover, this is a growing tendency due to the increasing availability of bigger low-cost memories. In this case, it makes no sense to have a posting list cache, so the use of an intersection cache becomes an interesting approach to decrease the cost of executing a query by caching previously computed list intersections.

4.3.1 Static Policies

We use the seven static caching policies and the four proposed strategies (S1, S2, S3 and S4). As we expected, the best strategy is S4. Abusing notation, we find that the ordering of the remaining policies is S3>S1>S2 for big caches and S4 is 16.9% better (on average) than S2. As we mentioned earlier, S3 requires the computation of more intersections (and this fact may be seen as an unnecessary overhead) that do not report cost savings in this static setup. The most competitive policy is FS (Figures 4.23, 4.24, 4.25 and 4.26). As we mentioned earlier, it is known that FS achieves the best hit rate for list caching [10], which leads to the best performance on intersection caching when the costs of the items are small.

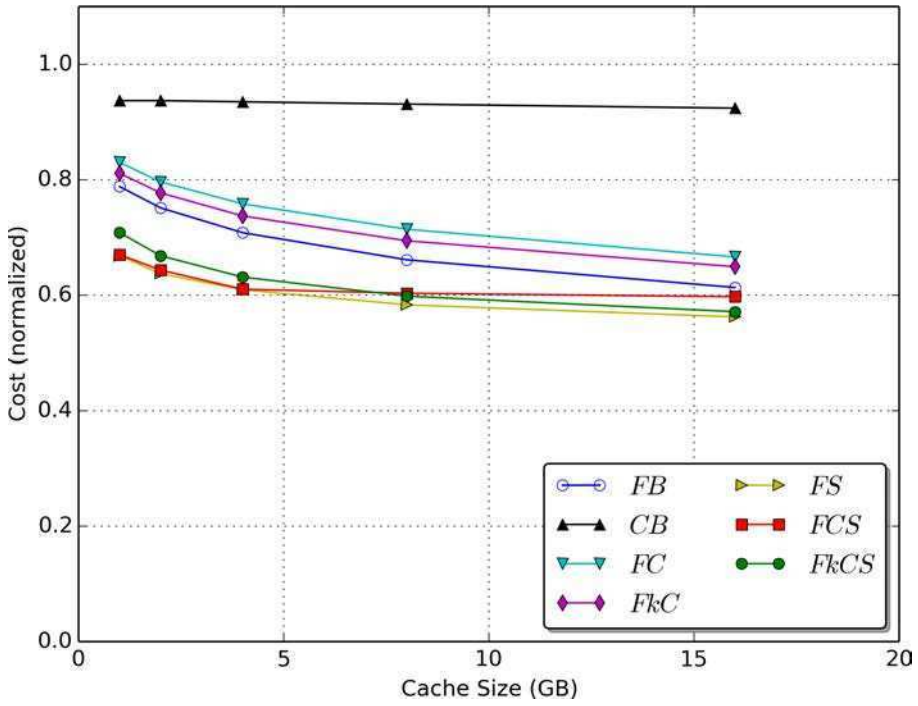

FIGURE 4.23: Total cost incurred by the S1 processing strategy for the seven static cache policies.

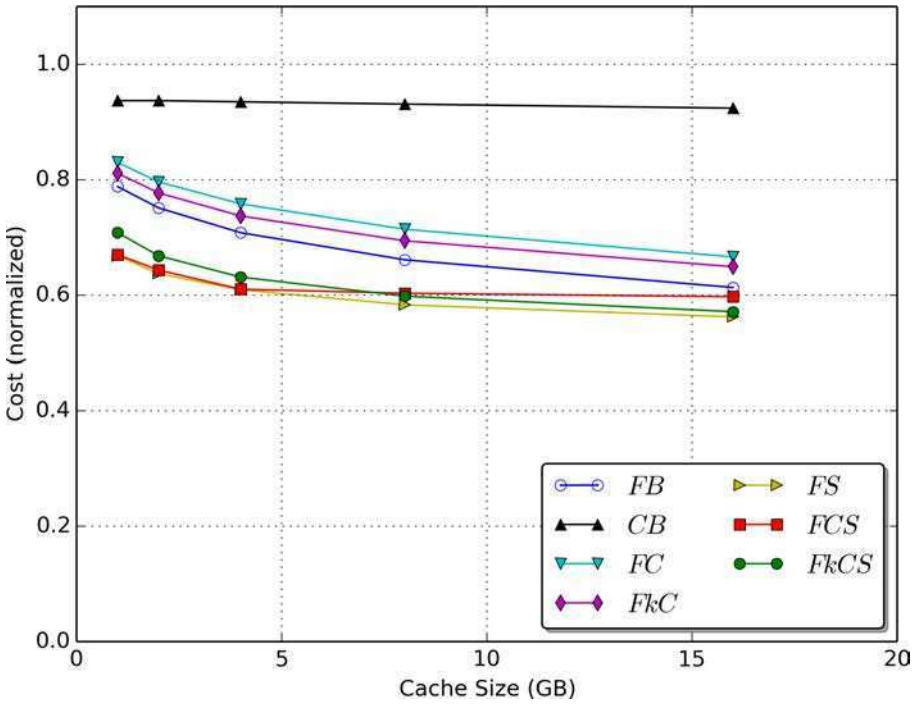

FIGURE 4.24: Total cost incurred by the S2 processing strategy for the seven static cache policies.

Considering FS as a baseline, the best cost-aware policy is FkCS whose performance is 1.0% worse (on average) than FS using the S4 strategy but it performs similarly for big cache sizes. Again, the cost-based policy (CB) performs the worst. Comparing the two most competitive cost-aware replacement policies we find that FCS performs slightly better than FkCS for smaller cache sizes (between 2.2% and 4.0%) but this situation reverses for bigger cache sizes (up to 9.6%), thus showing the usefulness of the k parameter to exacerbate the importance of the items' frequency.

We also observe that the gains of computing extra intersections in S3 (to improve the hit rate) are not compensated with the savings due to cache hits because the involved costs are small (only CPU time). This fact is clearly seen in Figure 4.27 that illustrates the best static policies (FCS and FkCS) for S1, S2, S3 and S4 strategies. Another observation is the poor performance of CB and the little improvement when increasing the cache size. The most costly items are not frequent enough so this policy performs badly because the gains of a cache hit are small.

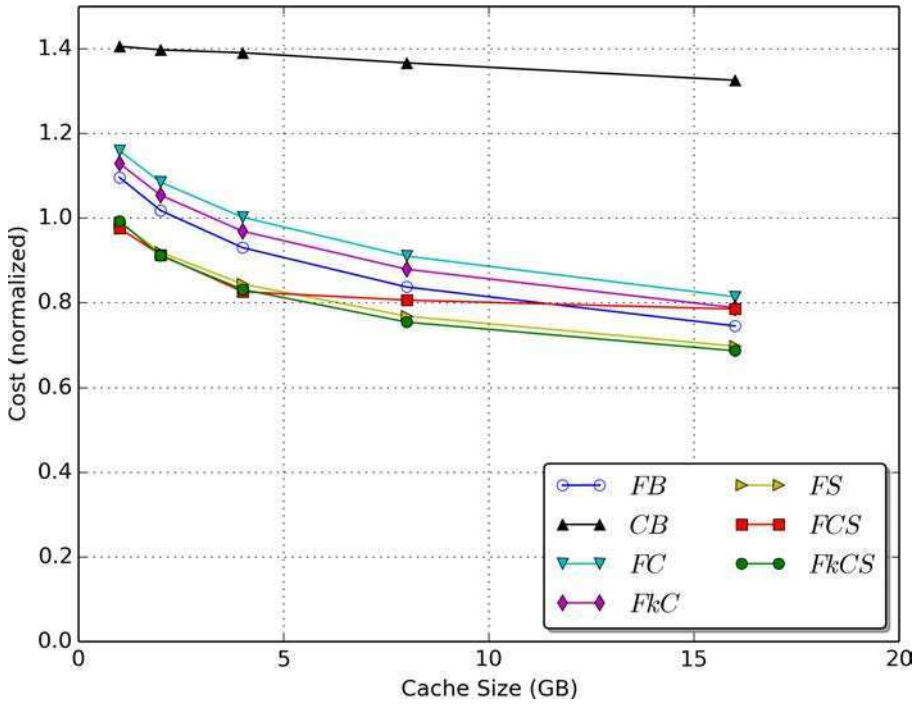

FIGURE 4.25: Total cost incurred by the S3 processing strategy for the seven static cache policies.

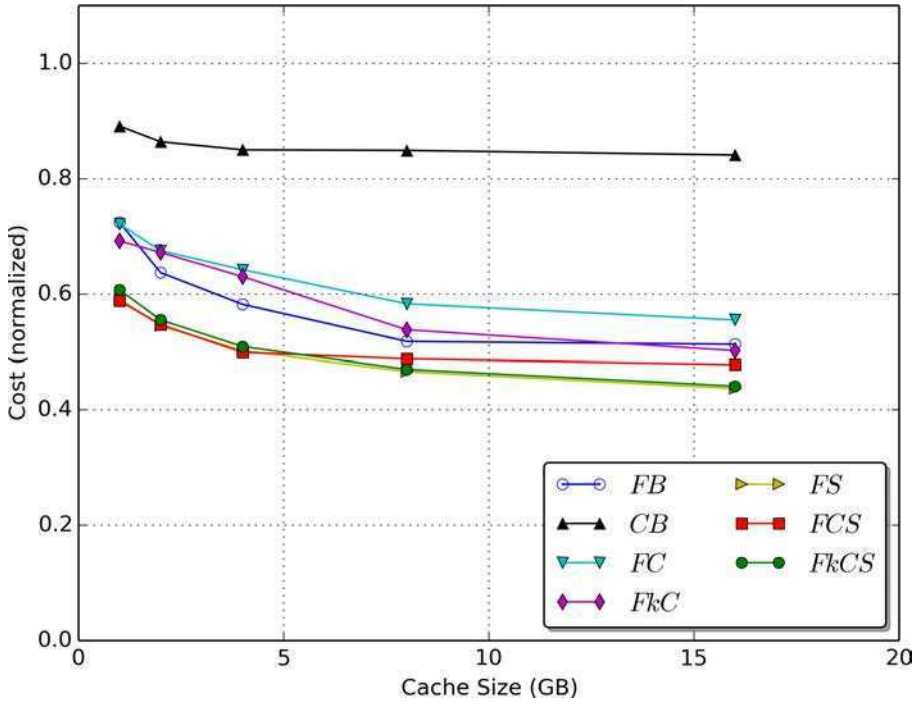

FIGURE 4.26: Total cost incurred by the S4 processing strategy for the seven static cache policies.

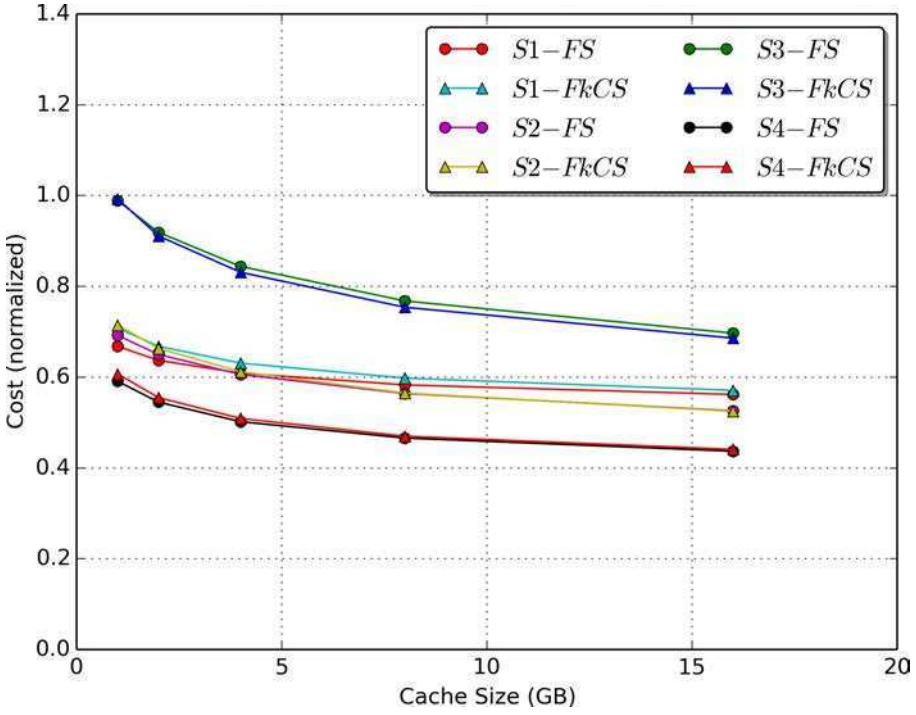

FIGURE 4.27: Comparison of total costs incurred by the two best static policies and the four processing strategy.

4.3.2 Dynamic Policies

In the case of pure dynamic policies, they outperform the static ones for all the strategies. The S4 strategy is the best and the global ordering of the remaining ones is S3>S2>S1. We find that GDS is the best policy for S1, S2 and S3 strategies (Figures 4.28, 4.29, 4.30 and 4.31) while FCS is the best for S4. This is a rather surprising result because FCS performs poorly with the S1 strategy. We observe that the gains of computing extra intersections in S3 (to improve the hit rate) is not compensated with the saving due to cache hits because the involved costs are small (only CPU time). Another observation is the poor performance of CB and the little improvement when increasing the cache size. The most costly items are not frequent enough so this policy performs badly because the gains of a cache hit are small.

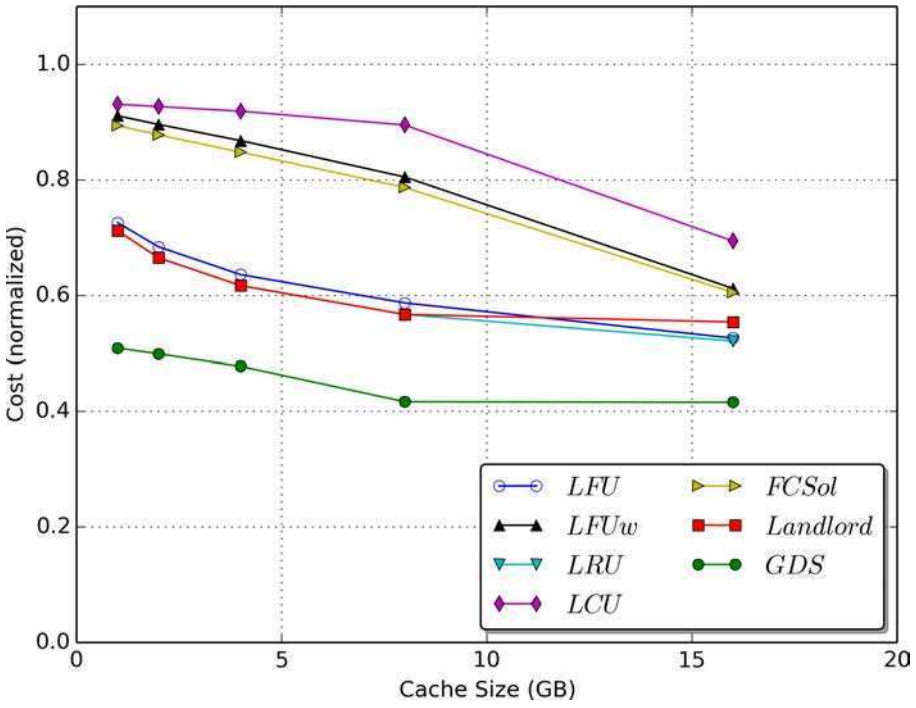

FIGURE 4.28: Total cost incurred by the S1 processing strategy for the seven dynamic cache policies.

When we consider the best policy for each strategy, S2 is 26.0% better than S3, S1 is 2.3% better than S2, and finally S4 is 19.6% better than S1. Looking at only S4, we see that GDS (cost-aware) outperforms LRU (our cost-oblivious baseline) in about 21.1%. Figure 4.38 shows the comparison of the best dynamic policies and the four processing strategies.

Finally, we introduce the results when we consider three-term intersections too (Figure 4.33). An interesting improvement is shown in the performance around 12.3% (on average) for LRU, 10.5% for Landlord, and 2.4% for GDS.

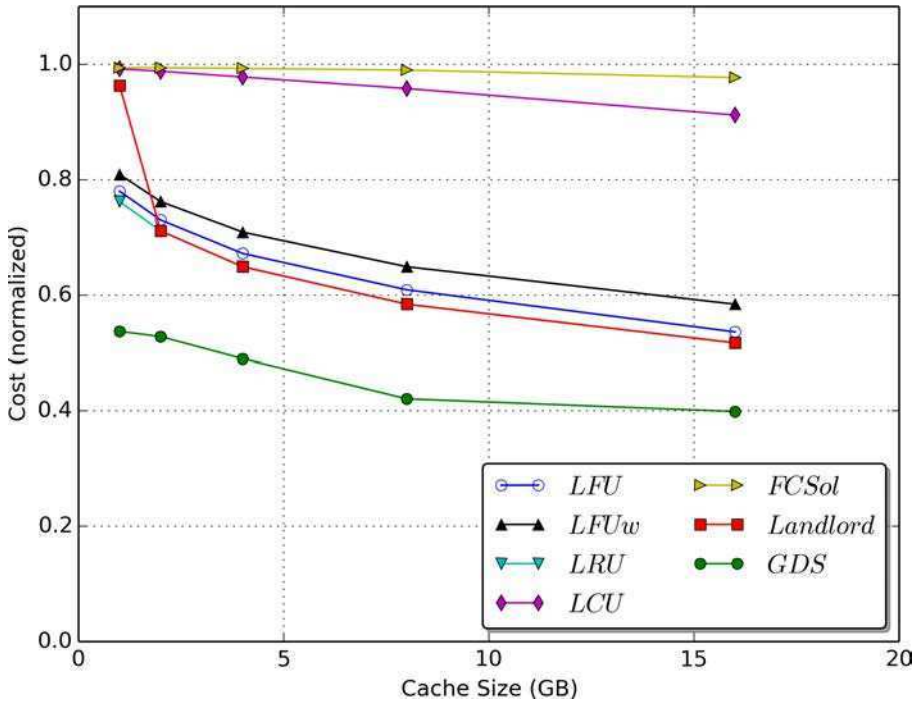

FIGURE 4.29: Total cost incurred by the S2 processing strategy for the seven dynamic cache policies.

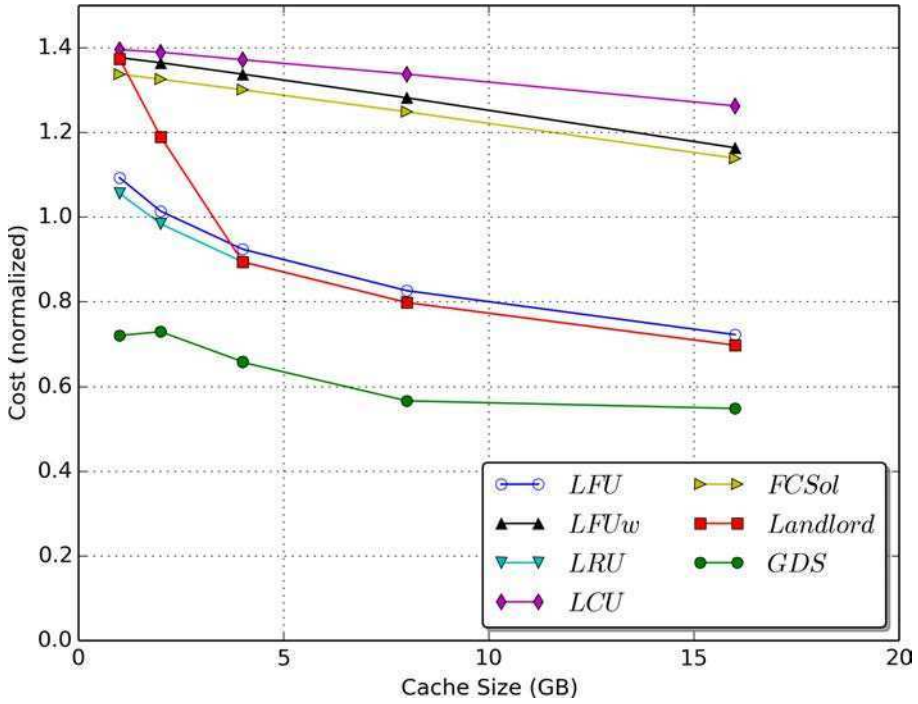

FIGURE 4.30: Total cost incurred by the S3 processing strategy for the seven dynamic cache policies.

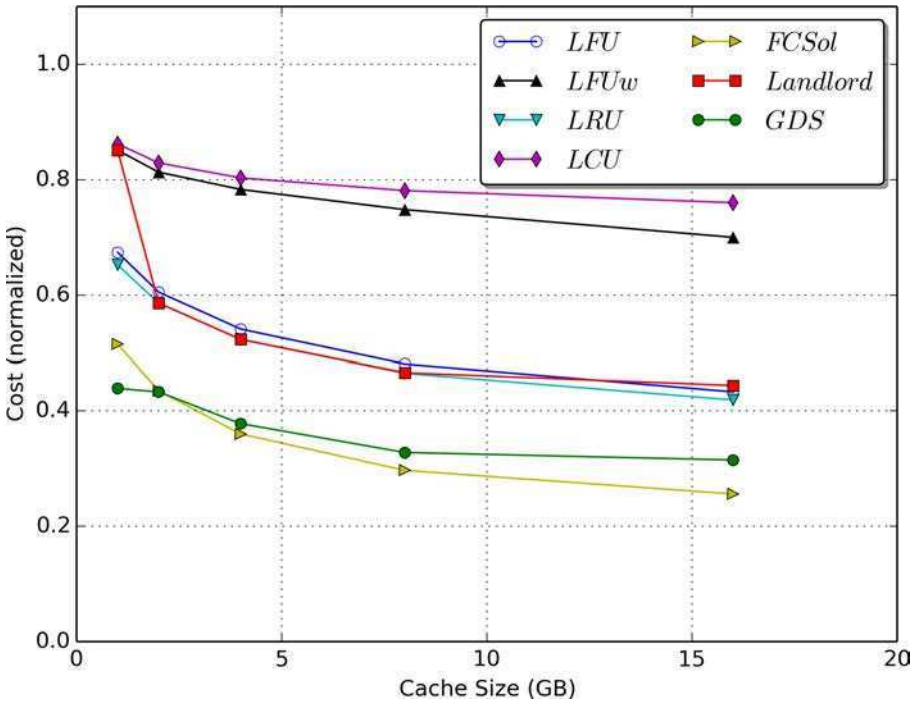

FIGURE 4.31: Total cost incurred by the S4 processing strategy for the seven dynamic cache policies.

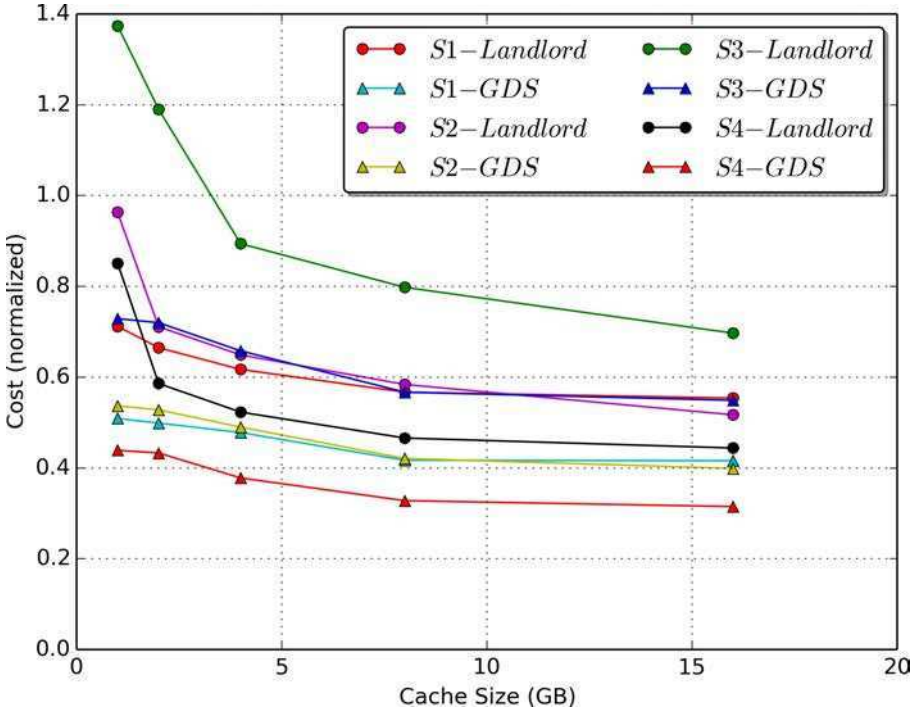

FIGURE 4.32: Comparison of total costs incurred by the two best dynamic policies and the four processing strategy.

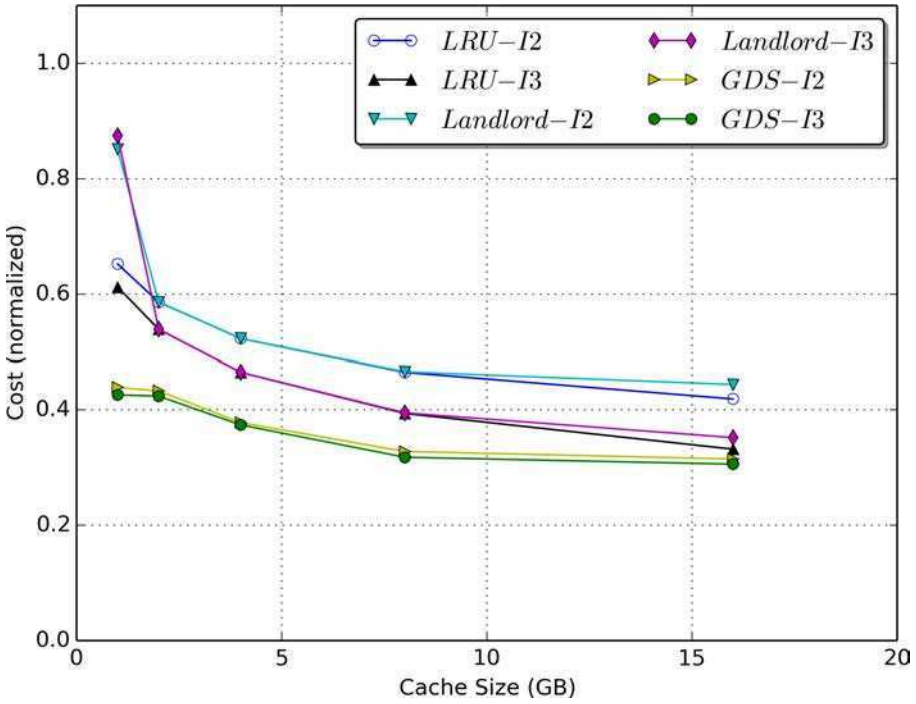

FIGURE 4.33: Total cost incurred by the S4 processing strategy enhanced with three-term intersections (I3) for dynamic policies.

4.3.3 Hybrid Policies

At last, we evaluate hybrid caching policies with the index loaded in the main memory using the same split of the cache space (80/20% for the static and dynamic parts respectively). We use the same normalization as in the previous cases to allow the comparison among strategies. Figures 4.34, 4.35, 4.36 and 4.37 show the results.

We observe that it is possible to obtain improvements when the dynamic portion of the cache is cost-aware. We find a similar relation among the strategies as in previous cases ($S3 > S2 > S1 > S4$). In S1 and S4 (the winning strategies), FCS/GDS is the best replacement policy while SDC becomes the best choice in the remaining strategies (S2 and S3). Under these considerations, S2 is 14.5% better than S3, S1 is 14.5% better than S2, and finally S4 is 14.5% better than S1. Figure 4.38 shows the resulting performance for the best hybrid caching policies combined with each resolution strategy.

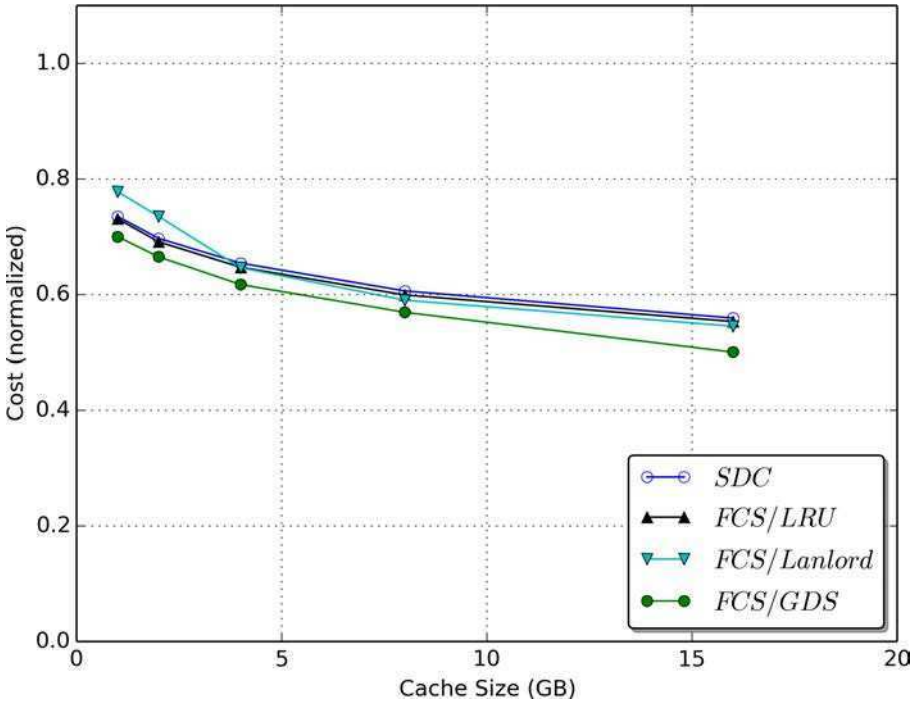

FIGURE 4.34: Total cost incurred by the S1 processing strategy for the seven dynamic cache policies.

We also test two-term and three-term intersections (Figure 4.39). In this case, incorporating three-term intersections leads to an improvement close to 3.0% (on average).

The big picture shows that hybrid policies perform similar to static ones while fully dynamic replacement policies become 20.0% better (on average).

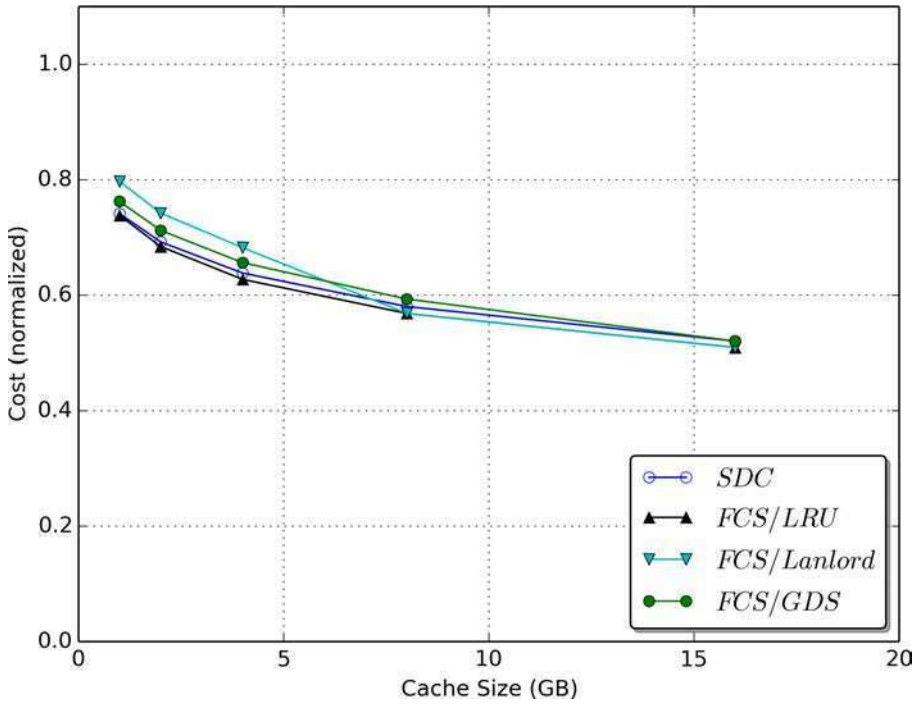

FIGURE 4.35: Total cost incurred by the S2 processing strategy for the seven dynamic cache policies.

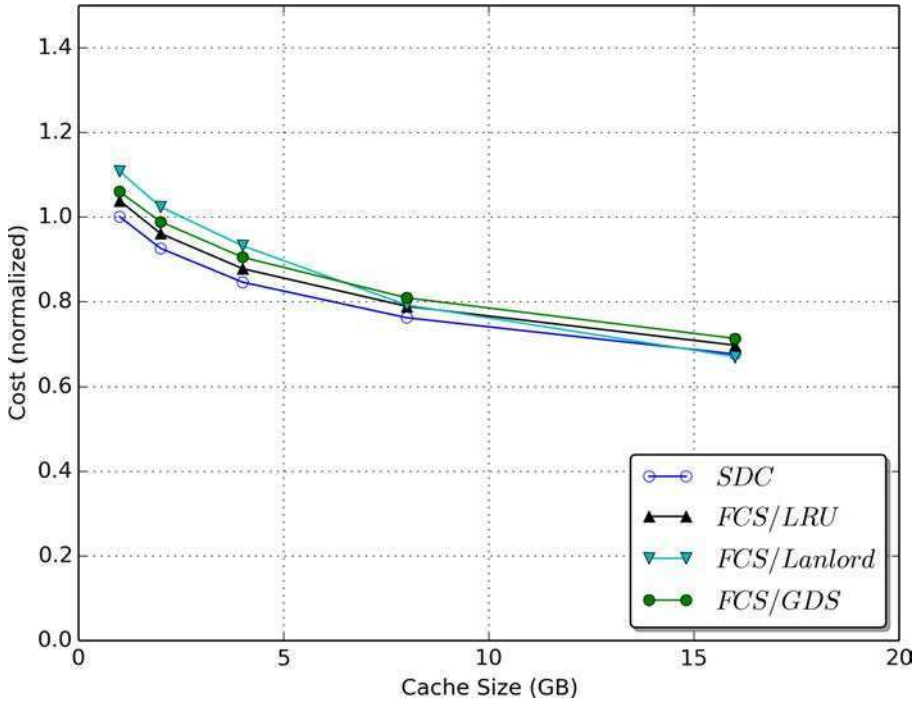

FIGURE 4.36: Total cost incurred by the S3 processing strategy for the seven dynamic cache policies.

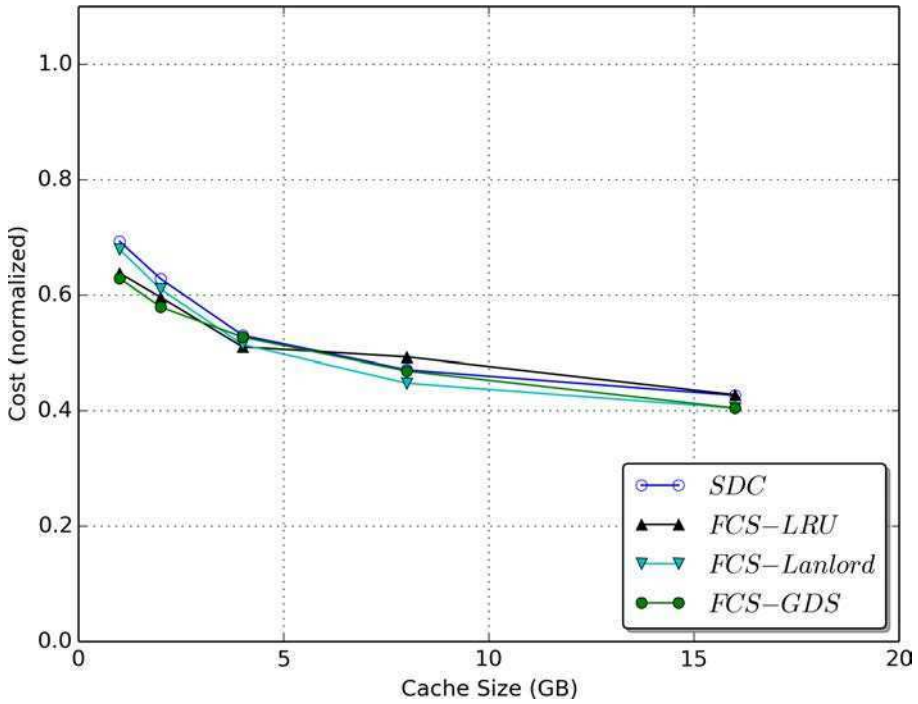

FIGURE 4.37: Total cost incurred by the S4 processing strategy for the seven dynamic cache policies.

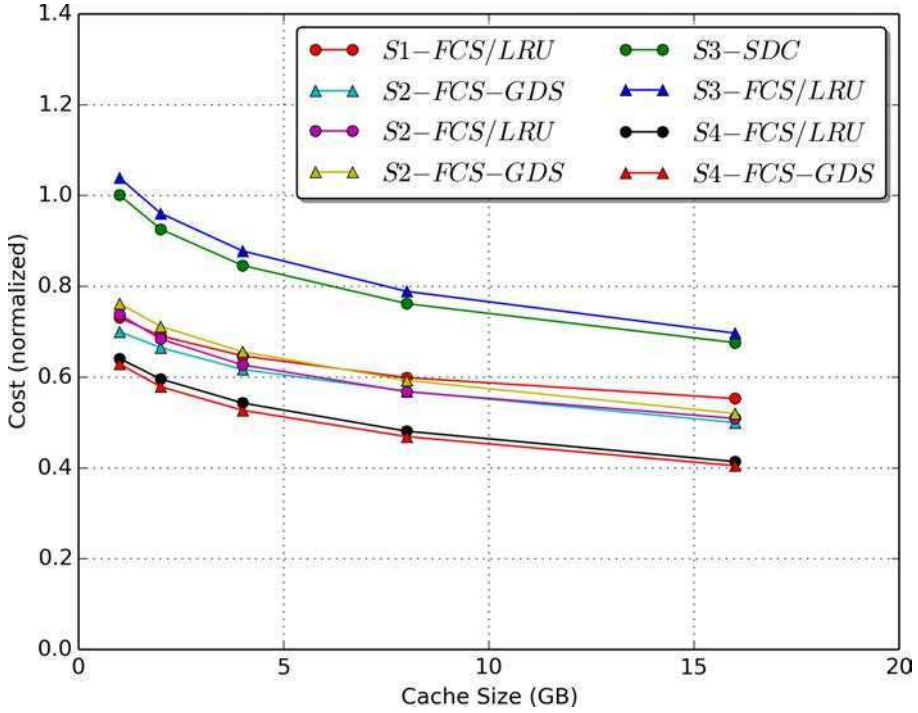

FIGURE 4.38: Comparison of total costs incurred by the two best hybrid policies and the four processing strategy.

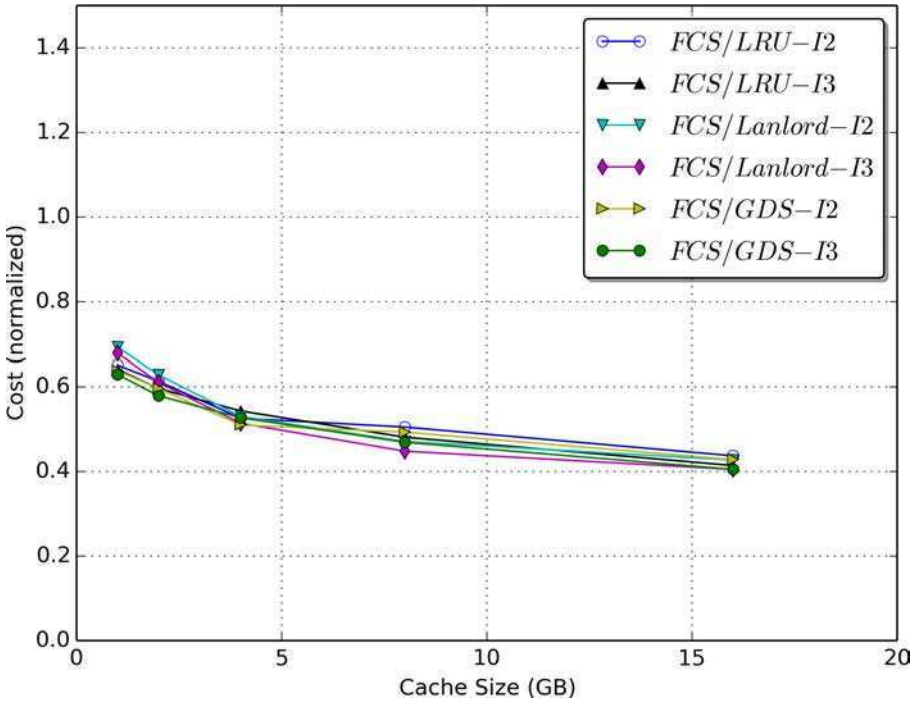

FIGURE 4.39: Total cost incurred by the S4 processing strategy enhanced with three-term intersections (I3) for dynamic policies.

4.3.4 Comparing Variances

In this section we conduct a similar study as in Section 4.2.4 by computing and comparing the variance values for different configurations according to the cache size increases. Again, we select the basic S1 strategy and the baseline caching policy to compare them against the best strategy (S4) and the cost-aware caching policy that achieves the best performance (for static, dynamic and hybrid cases, respectively). In all cases, the results show a reduction of the variance in the cost distributions given by the query resolution strategy (S1 vs S4). Figures 4.40, 4.41 and 4.42 show the results.

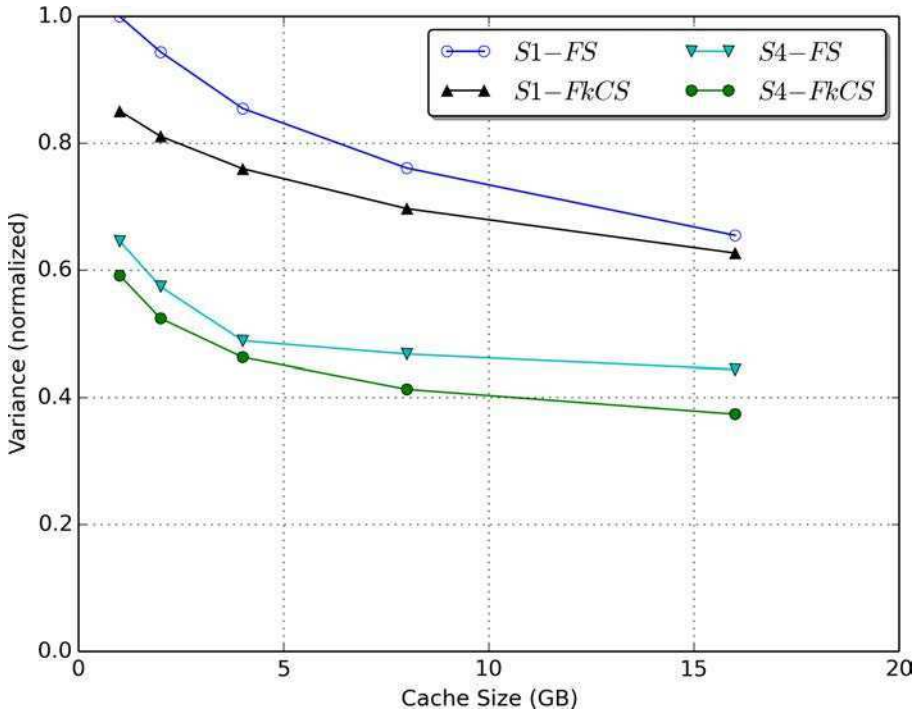

FIGURE 4.40: Comparison of variance for the baselines and best static policies and strategies.

In the case of static policies (FS vs FkCS), the variance reduction is about 10% (on average). When comparing strategies (S1 vs S4) the reduction is about 37%. Dynamic policies perform similarly than the previous cases but narrow differences arise between LRU and GDS (reduction of about 2.5%). However, a reduction close to 37% is observed when comparing S4 and S1. Finally, hybrid policies perform similarly to dynamic ones: the variance reduces around 2.0% (SDC vs FCS/GDS) and the difference between query resolution strategies decreases in around 35%. The Levene's test for equality of variances also shows a *p-value* < 0.0001 for the considered configurations (the same as in the case of a disk-resident inverted index, 1GB and 16GB).

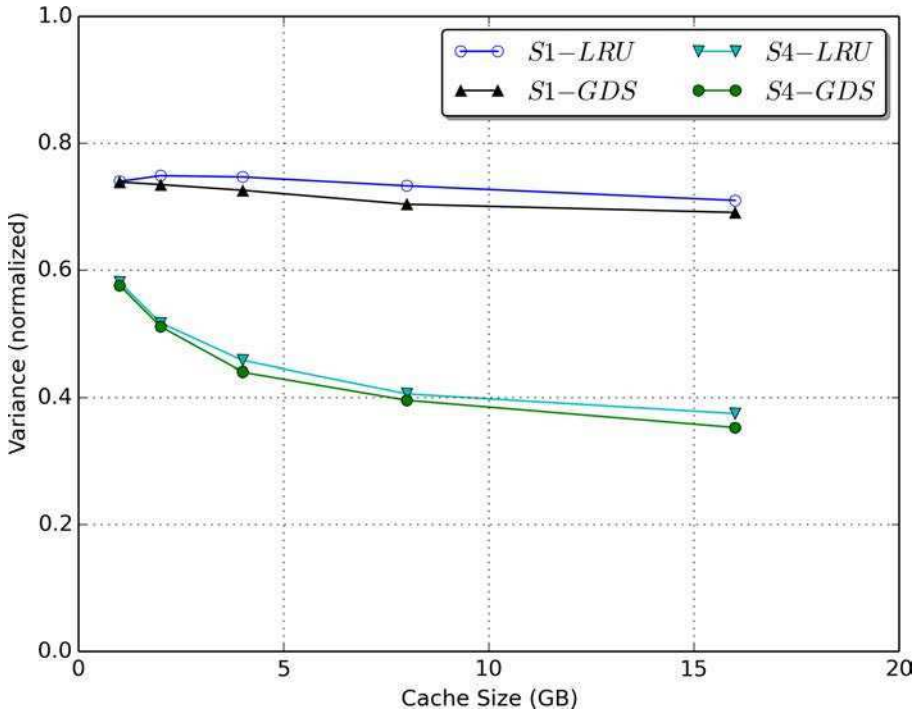

FIGURE 4.41: Comparison of variance for the baselines and best dynamic policies and strategies.

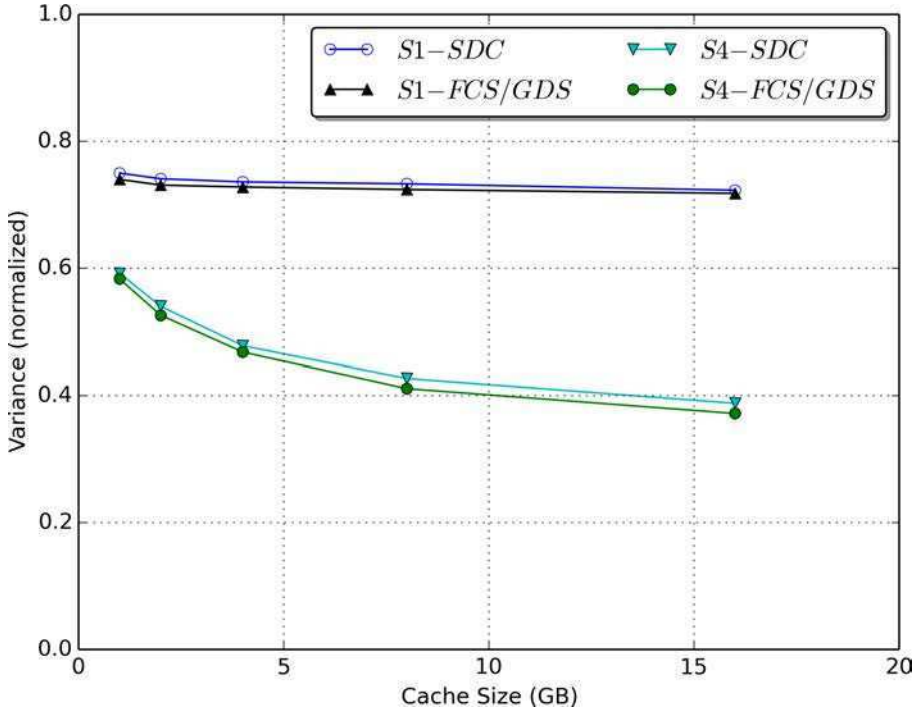

FIGURE 4.42: Comparison of variance for the baselines and best hybrid policies and strategies.

4.4 Summary

In this chapter, we analyze the usefulness of an intersection cache to reduce query processing time. We show that the cost and size distributions of term pairs follow a highly skewed behaviour (i.e. a power-law distribution). We propose and evaluate cost-aware intersection caching policies and different query resolution strategies that consider the existence of this kind of cache. Our study covers a wide range of cache replacement policies (also including cost-oblivious ones used as baselines) for static, dynamic and hybrid caching cases.

We consider two different configurations: with the index residing in disk and with the index residing in the main memory, as in the case of web search engines. We observe reductions in the total query processing cost in both cases. The overall results show that S4 is the best strategy, and this is independent of the replacement policy used. In the case of a disk-resident index and S4 strategy, hybrid policies perform better than static and dynamic (in this order). When considering a memory-resident index, dynamic policies perform better than hybrid and static, respectively. In the same way, we can observe that cost-aware policies are better than cost-oblivious ones, thus improving the state-of-the-art baselines.

Caching de Intersecciones Considerando el Costo - Resumen

En este capítulo se analiza la utilidad del caché de intersecciones para reducir el tiempo de procesamiento de un conjunto de consultas. Básicamente, se consideran diferentes políticas de reemplazo que consideran el costo de los ítems para decidir cuál desalojar, adaptadas a nuestro problema y se las compara contra políticas estándard usadas como referencia. En todos los casos, se consideran las cuatro estrategias de resolución de las consultas introducidas en el capítulo anterior, cubriendo casos estáticos, dinámicos e híbridos para las políticas de caching.

Complementariamente, se muestra que las distribuciones de costos y tamaños de las intersecciones siguen patrones muy sesgados (concretamente, ajustan a leyes de potencia), lo cual resulta adecuado para aplicar las políticas propuestas. Este estudio cubre, además, dos escenarios reales. En el primero, se considera que el índice invertido reside en almacenamiento secundario (disco), mientras que el segundo, considera que el índice reside completamente en la memoria principal. Este último caso corresponde a los motores de búsqueda de escala web, los cuales cuentan con recursos suficientes para soportar esta configuración y, además, deben responder en pequeñas fracciones de tiempo (típicamente, milisegundos).

La evaluación muestra una reducción del tiempo total de procesamiento en todos los casos. La estrategia S4 resulta la que aporta mayor beneficio independientemente del tipo de política utilizada. Además, también se verifica que las políticas propuestas que consideran el costo de las intersecciones (*cost-aware*) resultan más eficientes que aquellas que no lo tienen en cuenta (*cost-oblivious*).

Cuando se considera el índice en disco, aparece una observación interesante: las estrategias S2 y S3 resultan mejores que S1, lo que indica que cierto nivel de redundancia es útil. En el caso de políticas estáticas, la reducción de costo en el mejor caso alcanza el 39% y el ordenamiento final de las estrategias resulta S1 > S2 > S3 > S4. El panorama general muestra que las mejores políticas balancean frecuencia, costo y tamaño.

En el caso se políticas dinámicas, ocurre una situación similar, pero alcanzando un 8% de mejora entre la que considera el costo (GDS) y la de referencia (LRU). Sin embargo, no mejoran a las políticas estáticas. Finalmente, las políticas híbridas ofrecen la mejor performance. En particular, la combinación de la mejor estática (FCS) con la mejor dinámica (GDS) ofrece la más alta performance utilizando la estrategia S4.

Cuando se considera un índice residente en memoria, la situación es un poco diferente. El ordenamiento global de las estrategias resulta S3>S2>S1>S4 (en general) mostrando que el *overhead* de calcular intersecciones extra (en S2 y S3) no se compensa con la ganancia. Esto se debe que los costos en este caso son muy pequeños y la ganancia es marginal (cabe recordar que al estar el índice en memoria se evita el costo de la transferencia desde disco). Sin embargo, se verifica nuevamente que las políticas que consideran el costo de las intersecciones resultan más eficientes que las que no lo hacen y el panorama general muestra que, en este caso, las políticas dinámicas son las más eficientes (hasta un 20.0% mejores, en promedio), para el caso de la estrategia más competitiva (S4).

Los estudios realizados aquí se complementan con los correspondientes análisis de las varianzas de las distribuciones de costos producidos por las políticas y estrategias tomadas como referencia (baselines) y las mejores para caso. Esto es particularmente importante ya que, en un sistema distribuido como un motor de búsqueda, el tiempo de resolución está condicionado por el nodo más lento, y se espera que las estrategias propuestas mejoren además esta situación. Los resultados muestran que se aprecia una disminución de las varianzas ya sea por la estrategia de resolución (S1 vs S4) como de la política de reemplazo (sin considerar el costo vs considerando el costo), lo que refuerza la utilidad de los enfoques propuestos.

Chapter 5

Integrated Cache

In previous sections we have explained that industry-scale web search engines run in clusters that hold a large number of machines [14] and process queries in parallel, thus achieving fast response times and increasing query throughput. One of the key features of their architecture is that they maintain the entire (distributed) inverted index in the memory for efficiency and scalability purposes.

Under this consideration we argue that the traditional *List Cache* becomes useless; however the *Intersection Cache* is still useful [41] because it allows to save CPU time (i.e. the cost of intersecting two posting lists). For more general cases, such as medium-scale systems, only a fraction of the index is maintained in the memory. Here, list and intersection caches are both helpful to reduce disk access and processing time.

As we mentioned earlier, list and intersection caches are implemented at search node level. Usually, these are independent and try to offer benefits from two different perspectives. The List Cache achieves a greater hit rate because the frequency of individual terms is higher than that of pairs of terms, but each hit in the latter one entails a higher benefit because the posting lists of two or more terms are involved and some computation is also avoided.

In this chapter, we consider the scenario of medium-scale systems where the inverted index of the search nodes resides in disk and both list and intersection caches help to reduce query processing costs. Based on the observation that many terms co-occur frequently in different queries, our motivation is to build a cache that may capture the benefits of both approaches in just one memory area (instead of two). We call this approach *Integrated Cache* and it replaces both list and intersection caches at search node level, as shown in Figure 5.1.

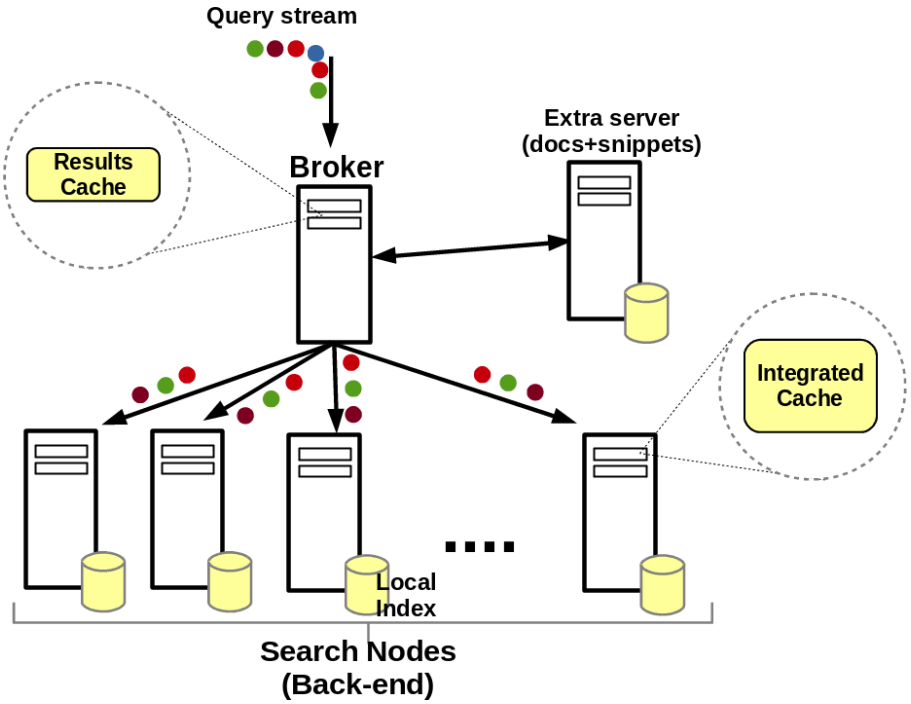

FIGURE 5.1: Simplified architecture of a Search Engine with the proposed *Integrated Cache* at search node level.

To reach this aim, we adapt a data structure previously proposed by Lam et al. [64]. The original idea is to merge the entries of two frequently co-occurring terms to form a single one and to store the inverted lists more compactly as shown in Figure 5.2.

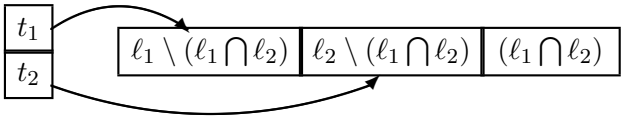

FIGURE 5.2: Separated Union data structure proposed by Lam et al. [64].

In this type of representation the postings of the *paired* terms are split into 3 partitions. Given two terms, t_1 and t_2, and their corresponding posting lists, namely ℓ_1 and ℓ_2, the storage works as follows: the first partition contains postings for t_1 only, the second partition contains postings for t_2 and finally, the last partition contains postings that belong to both terms (i.e. the intersection of t_1 and t_2). This representation is called Separated Union (SU)[1]. The authors also propose the use of an index compression technique obtaining a more compact structure that may reduce query processing time even more. Specifically, they combine the paired representation with Gamma Coding and Variable Byte Coding [8] schemes.

[1]In the original work, the authors propose another approach named "Mixed Union" that does not split the posting list into 3 portions but it uses two extra bits per posting that indicate the source of each one (i.e. '10' for the first term, '01' for the second one, or '11' for both). However, the "Separated Union" approach is more flexible and uses no extra space to encode the postings.

The Separated Union data structure is used to encode the postings of two frequently co-occurring terms which results in a reduction of space used by a disk-resident inverted index. Paired entries speed up query processing in both conjunctive and disjunctive mode. In the first case, the intersection portion contains the precomputed final result, allowing to save CPU cost. In the second one, to solve a disjunctive query that contains both terms, the computation of the disjoint set-union of the three parts is required.

5.1 Proposal

We adapt the structure depicted in Figure 5.2 to build a static cache in which the selected pair of terms offer a good balance between hit rate and benefit, leading to an improvement in the total cost of solving a query. We also investigate different ways of choosing and combining the terms.

The main idea is to design an *Integrated Cache* that behaves as list and intersection caches at the same time. We also extend the aforementioned approach by using the "Separated Union" (SU) [64] representation to maintain an in-memory data structure used for caching purposes. We add some features to the structure and a management algorithm to avoid the repetition of single term lists when these can be reconstructed using information held in previous entries. This leads to extra space savings and a more efficient use of memory, at the expense of some extra computational cost. The main idea is to keep in cache those pairs of terms that maximize the high hit ratio of the Posting Lists Cache and the savings of the most valuable precomputed intersections. The three main components of the Integrated Cache are:

1. An adapted SU data structure.
2. A redirection table.
3. A management algorithm.

The adapted SU data structure is shown in Figure 5.3. Entries 1 and 2 correspond to standard paired entries while lines 3 and 4 show the proposed improvements. We illustrate the idea with the following example: let us assume ℓ_i represents the inverted list of term t_i. Line 1 shows the entry for terms t_1 and t_2; the line contains the document identifiers (DocIDs) for the first term only $(\ell_1 \setminus (\ell_1 \bigcap \ell_2))$, then the postings of the second term only $(\ell_2 \setminus (\ell_1 \bigcap \ell_2))$, and finally, the last area holds the postings common to both terms (i.e. the intersection $(\ell_1 \bigcap \ell_2)$). Line 2 is analogous for terms t_3 and t_4. Note that obtaining

the posting list of any single term requires an extra computational cost for merging lists of the entry; however this is cheaper than loading it from disk.

Line 3 shows an entry that contains a previously cached term, t_1, that already appears in the first intersection. To reduce the memory requirement of this cache entry we avoid the repetition of part of the postings; namely, we propose to reconstruct the full posting list of term t_1 from the first entry and include a redirection (Θ) towards it. Line 4 corresponds to a similar case than the previous one, but it contains both terms already cached. In this case, the first and second entries include the redirections to the corresponding terms. The space allocated is required to store the result of $(\ell_3 \cap \ell_5)$ only. Figure 5.4 illustrates an example of a redirected list.

All redirections are kept in a simple lookup table whose structure consists of $(t_i \rightarrow (pair))$ and can be accessed in $O(1)$ average time using a hash function. The management algorithm handles the insertion of items during the initialization step, populating the static cache (by the use of Algorithm 1) and creating the redirection table.

At running time, it is possible to test the cache looking for a pair (t_i, t_j) or a single term (t_i) using Algorithm 2.

This structure and its management strategy offers some flexibility. For example, if we want to cache a single term, let's say t_1, the list is completely stored in the first area and the remaining two are kept empty $(t_1 \rightarrow |\ell_1|\phi|\phi|)$.

The proposed data structure also enables the possibility of solving disjunctive queries (OR) by simply joining the corresponding posting lists. For instance, to solve the query $q = \{t_1, t_2, t_3\}$ in a disjunctive way (using the same data shown in Figure 5.3) it is sufficient to join the lists of t_1 and t_2 (line 1) with the list of t_3 (line 2). Clearly, if one of the lists is not present in the cache, it must be retrieved from disk.

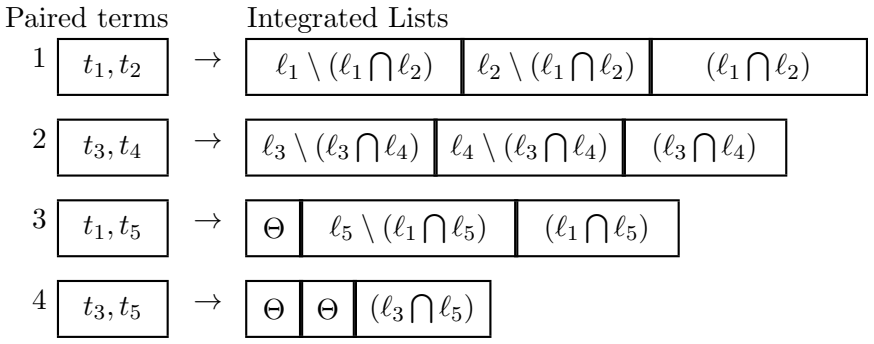

FIGURE 5.3: Data Structure used for the *Integrated Cache*.

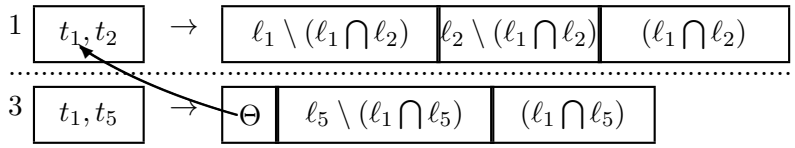

FIGURE 5.4: Example of a redirection from entry 3 (Θ) to entry 1 that corresponds to term t_1 (we omit entry 2 for clarity).

Algorithm 1: Insert an item in cache

Input: IC: Integrated Cache, (t_1, t_2): Key (pair of terms), L: (posting lists), RT: Redirection table
Output: IC: Integrated Cache, RT: Redirection table
$pair \leftarrow (t_1, t_2)$
$IC\{pair\} \leftarrow L$
if *(!exists(RT, t_1))* **then**
  $RT\{t_1\} \leftarrow (pair)$
end
if *(!exists(RT, t_2))* **then**
  $RT\{t_2\} \leftarrow (pair)$
end
return IC, RT;

In this way, the Integrated Cache replaces both the intersection and posting list caches at the search node level. The query resolution strategy first decomposes the query into pairs of terms. Each pair is checked in the Integrated Cache and the final resolution order is given by first considering the pairs that are present in the cache and afterwards intersecting them with the remaining ones. "Separate" terms (i.e. terms that are not present in any pair in the cache) are also checked in the Integrated Cache as a single term using the redirection table.

For example, given a query $q = \{t_1, t_3, t_4, t_6\}$ and the configuration of the Intersection Cache as shown in Figure 5.3, the query is solved in the following way:

(a) The pairs are checked in cache and the final resolution order becomes: $(((t_3 \bigcap t_4) \bigcap t_1) \bigcap t_6)$.

(b) The intersection $(t_3 \bigcap t_4)$ is retrieved from cache.

(c) The posting list of t_1 is obtained from the first entry of the cache using the redirection table (working as a Posting List Cache).

(d) The posting list of t_6 is retrieved from disk.

Starting from (c), the corresponding list is intersected with the resulting one of the previous step. In this example, t_1 incurs in the cost of joining the contents of the first and

Algorithm 2: Test item in cache

Input: IC: Integrated Cache, RT: Redirection table, p: search pattern ($p = (t_i, t_j)$ when looking for an intersection or $p = t_i$ in the case of a single term)

Output: List r: The results list (if p is found in cache) or [] (otherwise)

$r = []$;
if *(isPair(p))* **then**
  if *(found(IC, p))* **then**
    $r \leftarrow IC\{p\}[(\ell_i \bigcap \ell_j)]$
  end
else
  if *(found(RT, p))* **then**
    $(t_x, t_y) \leftarrow (RT\{p\})$
    if $(p == t_x)$ **then**
      $r \leftarrow IC\{(t_x, t_y)\}[(\ell_x \setminus (\ell_x \bigcap \ell_y))] \bigcup IC\{(t_x, t_y)\}[(\ell_x \bigcap \ell_y)]$
    else
      $r \leftarrow IC\{(t_x, t_y)\}[(\ell_y \setminus (\ell_x \bigcap \ell_y))] \bigcup IC\{(t_x, t_y)\}[(\ell_x \bigcap \ell_y)]$
    end
  end
end
return r;

third areas ($(\ell_1 \setminus (\ell_1 \bigcap \ell_2))$ and $(\ell_1 \bigcap \ell_2)$, respectively) of the first entry in the integrated cache to reconstruct its full posting list. Finally, only term t_6 incurs in disk access cost.

5.1.1 Compression of the Integrated Cache

Although the Integrated Cache approach already reduces the size of the resulting data structure, compressing the inverted index (namely, each of its posting lists) is a crucial tool used to improve query throughput and fast response times in WSEs. Data is usually kept in compressed form in memory (or disk) leading to a reduction in its size that would typically be about 3 to 8 times, depending on index structure, stored information and compression method. Usually, document identifiers, frequency information and positions are stored separately and can be compressed independently, even with different methods. These techniques have been studied in depth in the literature [110], [74], [8]. Among the compression methods for posting lists we can mention the classical Elias [37] and Golomb [49] encodings and the more recent Simple9 [2] and PForDelta [118] encodings. In this work, we also evaluate the compressed version of our proposal using the state-of-the art PForDelta method which improves the tradeoff between compression ratio and decompression speed.

In the standard inverted index, the postings lists are represented separately while in the Integrated Cache representation, the postings lists are paired[2] in each entry. This leads to some differences which affect the compression ratio when compressing the data structure. As we observed, DGap encoding is usually used to represent posting lists because this compression technique is effective in compressing lists of relatively small integers.

In the case of the Integrated Cache the original sequence is modified as shown in the following example. Let $\ell_i = \{10, 11, 12, 13, 15, 18\}$ and $\ell_j = \{11, 15, 21, 23\}$ be the posting lists of terms t_i and t_j respectively.

In the case of an inverted index, the DGap'ed representation becomes:

$$\ell'_i = \{10, 1, 1, 1, 2, 3\} \text{ and } \ell'_j = \{11, 4, 6, 2\}$$

which are then compressed. However, the *integrated* representation of a pair of terms (t_i, t_j) is as follows:

$(t_i \setminus t_j) = \{10, 12, 13, 18\}$
$(t_j \setminus t_i) = \{21, 23\}$
$(t_i \bigcap t_j) = \{11, 15\}$

and its DGap'ed version $(t_i, t_j)'$ becomes:

$(t_i \setminus t_j)' = \{10, 2, 1, 5\}$
$(t_j \setminus t_i)' = \{21, 2\}$
$(t_i \bigcap t_j)' = \{11, 4\}$

The comparison of both representations shows that ℓ'_i and ℓ'_j can be better compressed using DGap because these lists contain more runs of 1's than $(t_i, t_j)'$ that contains larger integers. Both series of data follow a power-law distribution $f(x) = Cx^{-\beta}$ with parameter $\beta = 1.34$ and $\beta = 2.17$ for lists and integrated representation respectively. In Figure 5.5, we can observe that DGap'ed lists have more runs of 1's (and lower values) than the integrated one.

However, the performance comparison of the two proposals requires to analyze the compression ratios in both cases. Let ℓ_i denote the posting list of term t_i, and I_{ij} an entry

[2]The framework also supports the representation of three-term intersections at the expense of the integrated list structure becoming more complex.

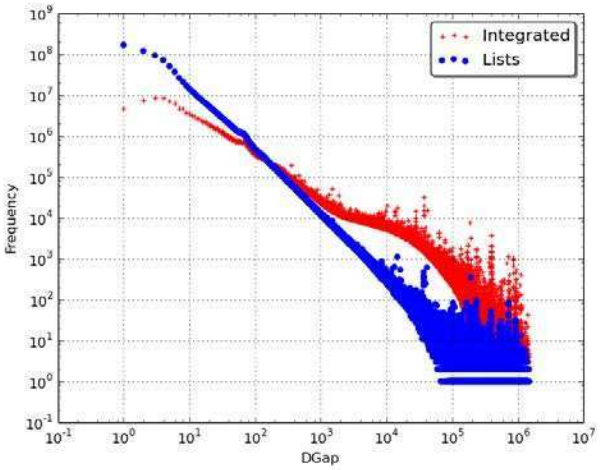

FIGURE 5.5: DGap value distributions for the Integrated and standard Posting Lists.

in the Integrated Cache that represents the pair $x = (t_i, t_j)$. Note that this representation becomes:

$$I_{ij} = (\ell_i \setminus (\ell_i \bigcap \ell_j)) \cup (\ell_j \setminus (\ell_i \bigcap \ell_j)) \cup ((\ell_i \bigcap \ell_j))$$

Then, we denote their compressed forms as $C(\ell_i)$ and $C(I_{ij})$ respectively. To compare the space used for each representation we use the ratio:

$$\delta_x = \frac{|C(I_{ij})|}{|C(\ell_i)| + |C(\ell_j)|} \tag{5.1}$$

If $\delta_x < 1$, compressing the Integrated entry is more efficient than compressing the posting lists separately; otherwise, it requires more space than the sum of both lists. It is clear that the Integrated Cache representation is more space-efficient when the intersection $(\ell_i \bigcap \ell_j)$ is large. However, there is a drawback when compressing I_{ij}. The split of each list ℓ_i in two lists $(\ell_i - (\ell_i \bigcap \ell_j))$ and $(\ell_i \bigcap \ell_j)$ might separate consecutive DocIDs, leading to shorter runs of 1's in the DGap representation; in such a case, $C(I_{ij})$ has a lower compression ratio than $C(\ell_i) + C(\ell_j)$.

Therefore, to determine the impact of the size of the intersection between both terms with respect to the value of δ ratio, we analyze its behaviour as a function of the size of the "Intersection Size" (IS) ratio defined as:

$$IS_{ij} = \frac{|\ell_i \bigcap \ell_j|}{|\ell_i| + |\ell_j|} \tag{5.2}$$

For each pair of terms in the dataset we compute both IS_{ij} and δ ratios. Figure 5.6 shows the results. Values of $\delta_x > 1$ correspond to a worse compression rate of the integrated representation with respect to the compression of the separated lists

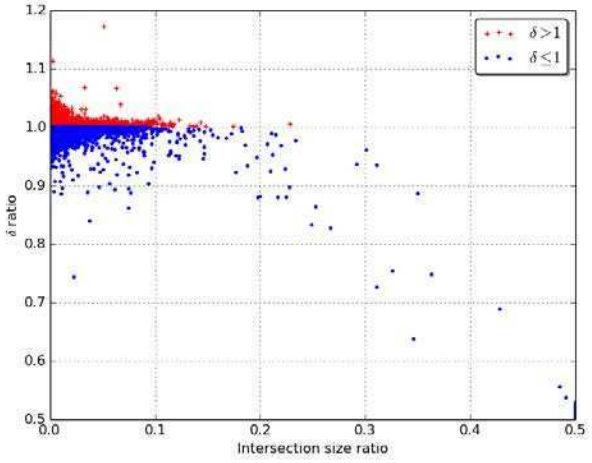

FIGURE 5.6: Integrated cache compression performance: δ_x ratio scatter plot between lists vs integrated representations. x-axis in log scale to get a clearer view.

Examining those values we find that 97% of them occur when the IS_{ij} ratio is smaller than 0.15. This means that the integrated representation is still more space-efficient than the compression of the lists (ℓ_i and ℓ_j) when $IS_{ij} > 0.15$. This observation shows that the size of the intersection is a good criterion to select pairs of terms to insert into the Integrated Cache (the bigger IS_{ij}, the better).

We also compute the efficiency in the use of the space for the Integrated Cache with respect to the separated posting lists in both uncompressed and compressed representations. For different numbers of entries in the cache (that grows in powers of 10), we sum the length of the standard (separated) posting lists ($|\ell_i + \ell_j|$) and the integrated representation ($|Iij|$) respectively. Finally, we calculate the ratio between both of them. Table 5.1 shows the sum of posting lists lengths and δ_x ratio for the uncompressed representations. The entries in the table show that the Integrated Cache achieves a better use of the space (as the IS_{ij} ratio decreases).

# of entries	$\sum \lvert\ell_i\rvert + \lvert\ell_j\rvert$	$\sum \lvert I_{ij}\rvert$	δ_x ratio
10	9,797,866	5,060,928	0.5165
100	44,227,078	23,899,513	0.5404
1.000	143,357,823	108,769,302	0.7587
10.000	457,975,054	404,064,223	0.8823
100.000	706,712,370	653,568,221	0.9248

TABLE 5.1: Sum of posting lists lengths for the uncompressed representation.

The compressed versions of both representations show a similar behaviour (Table 5.2). However, the ratio is slightly worse in this case due to the lower compression performance of the integrated representation $C(I_{ij})$, as a consequence of the DGap values distribution (described above). Figure 5.7 compares the ratio of both representations for the different number of entries in the tables.

# of entries	$\sum C\|\ell_i\| + C\|\ell_j\|$	$\sum \|C(I_{ij})\|$	δ_x ratio
10	720,395	409,170	0.5680
100	4,570,464	2,685,710	0.5876
1.000	19,833,629	16,817,892	0.8479
10.000	81,141,977	77,938,818	0.9605
100.000	133,297,699	129,155,629	0.9689

TABLE 5.2: Sum of posting lists lengths for the compressed representation.

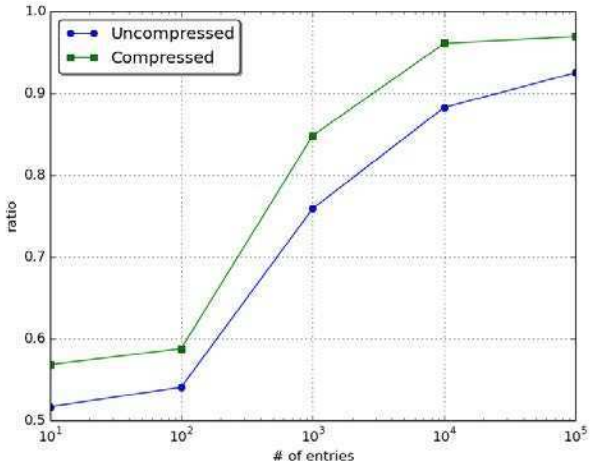

FIGURE 5.7: Integrated cache compression performance: Efficiency ratio (the lower, the better) between lists vs integrated representations. x-axis in log scale to get a clearer view.

5.2 Selecting Term Pairs

In this section we describe different approaches used to select the pairs of terms to fill up the cache. We propose a static posting list cache populated with those lists that maximize a particular function. We consider the $Q_{tf}D_f$ algorithm [10] that is one of the best for maximizing hit rate and a variant of it in which each posting list is weighted according to the $f(t_i) \times |\ell_i|$ product. In this expression, $f(t_i)$ is the raw frequency of term t_i in a query log training set and $|\ell_i|$ is the length of the posting list of term t_i in the reference collection. Hereafter, we refer to this metric as FxS.

We consider several strategies to select the "best" intersections to keep in cache. According to the analysis introduced in the previous section, the "best" intersections to

populate the cache are those that maximize the size of the intersection $\ell_i \bigcap \ell_j$. We also consider the FxS product that weights each posting list in the baseline method.

5.2.1 Greedy Methods

We first consider the simplest approach that starts ordering the posting lists according to their FxS products. Then, it merges lists by pairing together consecutive term pairs as $(1^{st}, 2^{nd}), (3^{rd}, 4^{th}), ..., ((n-1)^{th}, n^{th})$. We refer to this method as *PfBT-seq*. Note that this approach groups "good" terms but it does not take into account the size of their intersections.

The second approach (*PfBT-cuad*) computes the intersection of each possible pair (for all term lists) and then selects the pairs that maximize $(t_i \bigcap t_j)$ without repetitions of terms. This algorithm is time consuming ($O(n^2)$) so, we run it considering only sub-groups of lists that we estimate may fit in cache (according to their size). For example, 1GB cache holds roughly 1000 lists of separated terms for a given collection, so we compute the 500 best pairs and then we fill the remaining space with pairs picked sequentially (using the same criteria as in *PfBT-seq*).

The third approach (named *PfBT-win*) is a particular case of the previous one that tries to maximize the space saving among a group of posting lists. It sets a window of w terms (instead of all terms) and computes the intersection of each possible pair.

The intention behind this approach is to bound the computational cost required to get the best candidate term pairs. Finally, it selects the definitive term pairs using the same criterion as before.

5.2.2 Term Pairing as a Matching Problem

The fourth approach (*PfBT-mwm*) considers the term pairing as an optimization problem, reducing it to the Maximum Weighted Matching (MWM). We formalize the problem as follows:

Let $G(T, E)$ be a graph with a set of vertices T and a set of edges E. Assume that each vertex $t_i \in T$ corresponds to a term. Suppose that exists an edge $e_{ij} \in E$ between each couple of vertex v_i and v_j as well. The weight of each edge (w_{ij}) is given by the size of the intersection $|\ell_i \bigcap \ell_j|$ that exists if and only if $|\ell_i \bigcap \ell_j| > 0$.

For the above graph G, a matching M in G is a set of pairwise non-adjacent edges; that is, no two edges share a common vertex. Given our weight scheme, M is a Maximum

Weight Matching if M is a matching and $\sum_{e \in M} w_{ij}$ is maximal. As a result of this optimal pairing, we populate the cache with the couple of terms corresponding to nodes at both ends of the maximum weighted matching in G.

In our experiments we use the matching algorithm proposed in [46]. This is an exact algorithm that runs in $O(n^3)$ time; however the size of our graphs (thousands of vertices) makes it computationally tractable and the algorithm is executed offline (which reduces the impact of the processing time of this step).

This approach is similar to that proposed in [64] but we apply a slightly different weighting criterion. While in that work the weight e_{ij} measures the benefit of pairing two terms (in number of bits) considering the encoding method used for representing the lists, we directly define the weight as the size of the intersection $\ell_i \bigcap \ell_j$.

5.3 Experiments and Results

We evaluate the integrated cache in two scenarios: in the first one, data is stored in raw (uncompressed) format, while in the second one data is compressed using the PForDelta coding.

5.3.1 Data and Setup

We evaluate the proposed framework against a competitive list caching policy using two real web crawls with different characteristics and a sample of the AOL query log over our simulation framework. We use the UK and WB document collections (Section 3.3.1).

We select a subset of 6M queries to compute statistics and around 2.7M queries as the test set (AOL-1). Then, we filter the file keeping only unique queries. This allows isolating the effect of the *Result Cache* simulating that it captures all query repetitions (in the case of having a cache of infinite size), thus giving a lower bound on the performance improvement due to our cache. This second test file is about 800K queries (AOL-2).

The total amount of memory reserved for the cache ranges from 100MB to 1GB for the UK data set, and from 100MB to 16GB for the WB data set. A cache of 16 GB stores about 60% and 70% of the inverted indexes respectively. For each query we log the total costs incurred using a static version of *List Cache* (filling it with the most valuable posting lists) and the four proposed methods to fill the *Integrated Cache*. In both cases we use the FxS metric for comparison. We set $w = 10$ for the *PfBT-win* method.

Our implementation reserves eight bytes for each posting in the pure term partitions (DocID and frequency information use four bytes each) while the intersection area requires twelve bytes because it stores the frequencies of both terms.

Finally, rather than hit ratio, we consider the overall time needed by the different strategies to process all queries as a performance metric. Experimental evidence shows that substantial savings are possible using the proposed approach.

5.3.2 Integrated Cache with Raw Data

We compare all proposed approaches to select pairs of terms using the two document collections and the two query sets. For the AOL-1 query set we test all the approaches against the baseline, the standard posting list cache sorted according to the FxS score. In our setup, experimental evidence shows that FxS outperforms $Q_{tf}D_f$ when measuring cost. All evaluated strategies outperform the baseline and the best strategy is PfBT-mwm. These improvements increase up to 23% and 38% for the UK (Figure 5.8) and WB (Figure 5.9) collections respectively.

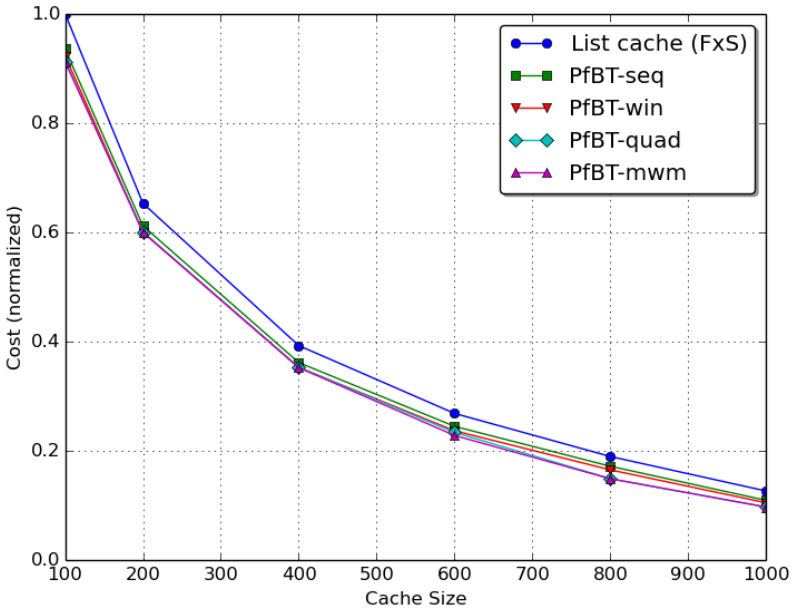

FIGURE 5.8: Performance of the different approaches using the AOL-1 query set and the UK collection (y-axis in log scale to get a clearer view).

Figure 5.10 shows the improvement achievable by the *Integrated Cache* as a function of cache size (for the different term pairing strategies).

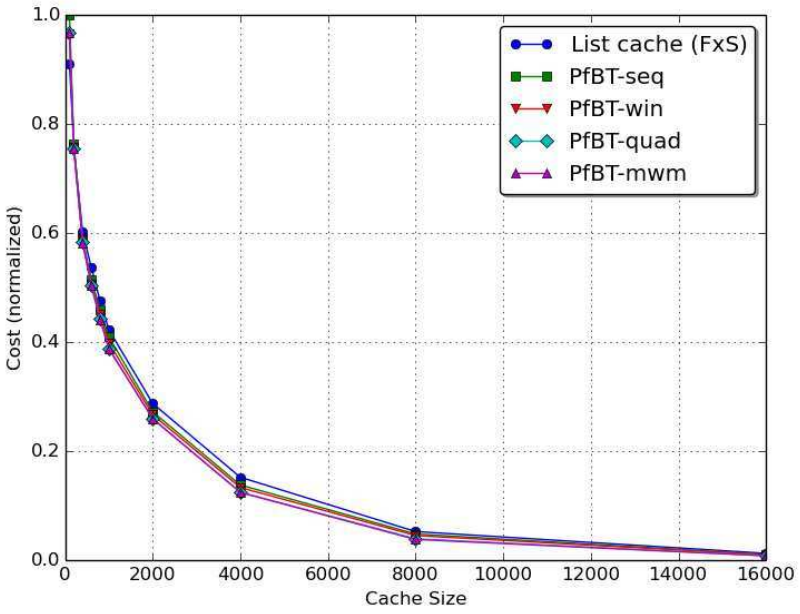

FIGURE 5.9: Performance of the different approaches using the AOL-1 query set and the WB collection.

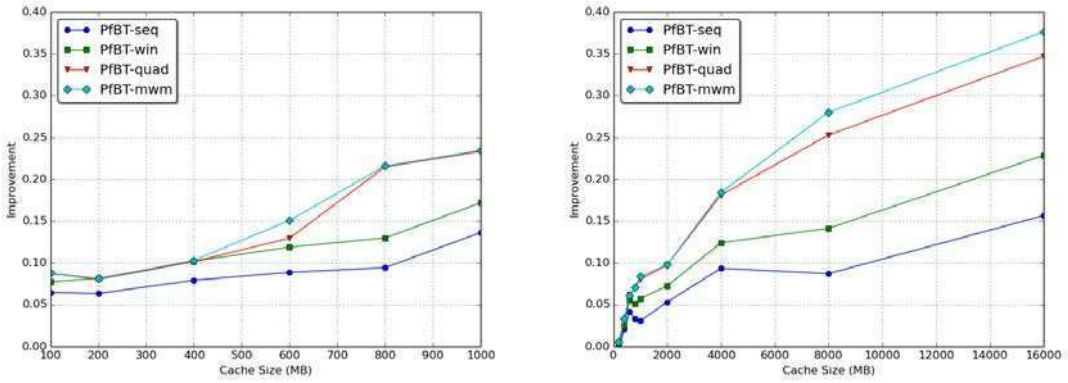

FIGURE 5.10: Improvements obtained by the Integrated Cache vs the baseline (List Cache).

In the second experiment we use the dataset of unique queries (AOL-2) and the best strategy obtained from the previous test (*PfBT-mwm*). Improvements range from 7% up to 22% for the UK collection. The behaviour is again different for the WB collection.

For smaller cache sizes, the performance is worse (or just slightly better in some cases) up to 1GB cache size and it increases up to 30% in the best case (16GB). This is because this collection has longer posting lists and only a few are loaded in smaller caches. These results are shown in Figures 5.11 and 5.12.

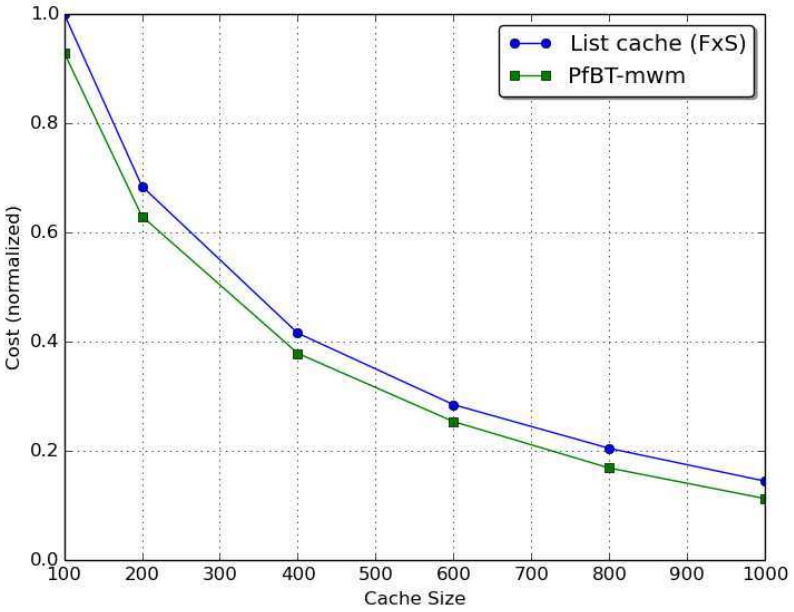

FIGURE 5.11: Performance of the PfBT-mwm approach using the AOL-2 query set and the UK collection.

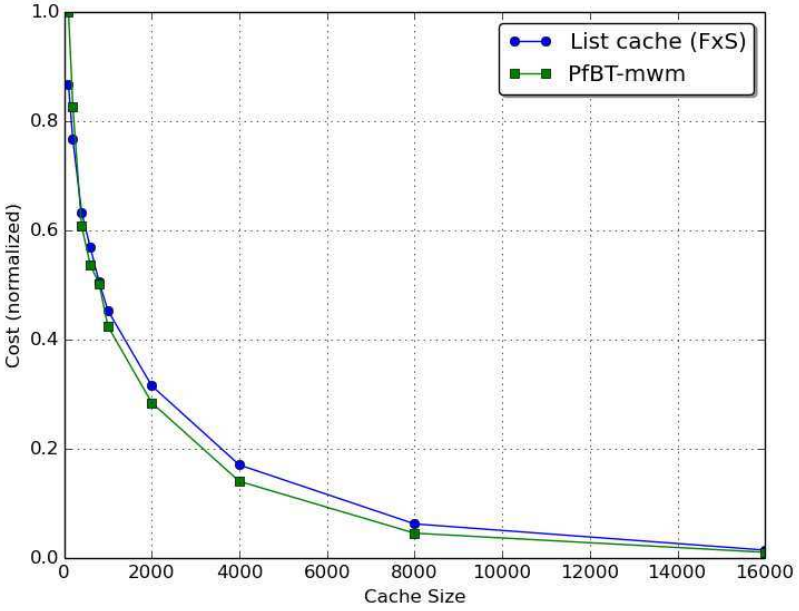

FIGURE 5.12: Performance of the PfBT-mwm approach using the AOL-2 query set and the WB collection.

5.3.3 Integrated Cache with Compressed Data

Compression techniques are used to reduce the size of each inverted list thus enabling the possibility of storing more data in the memory. For this reason, we evaluate the performance of our method when the inverted lists are compressed. To this aim we extend previous approaches to select the best pairs of terms. Namely, we consider *PfBT-mwm-sf*, a variation of the MWM approach, in which the weight of each edge of the graph is $w_{ij} = |\ell_i \bigcap \ell_j| \times f(t_i, t_j)$, where $f(t_i, t_j)$ is the frequency of the pair (t_i, t_j) in the query log.

We use the same objective function in a greedy approach that at each iteration chooses the pair of terms that maximize the value w_{ij}. The main difference in the resulting set of pairs with regards to the *PfBT-mwm* method is that some terms may be repeated in different pairs (e.g. term t_5 in line 3 of Fig. 5.3). However, the redirection strategy in the implementation of the Integrated Cache avoids the use of extra space in the memory. This last approach is named *PfBT-greedy-sf*.

In these experiments we use the WB collection which contains longer posting lists than the UK one that may be compressed still requiring enough space in cache. Figure 5.13 shows the results for the AOL-1 query set. The best strategy in this case becomes *PfBT-mwm-sf* with improvements up to 70% in the best case (cache sizes between 800 MB and 1GB). *PfBT-greedy-sf* is also much better than the baseline but it is worse than *PfBT-mwm-is* for cache sizes greater than 2 GB.

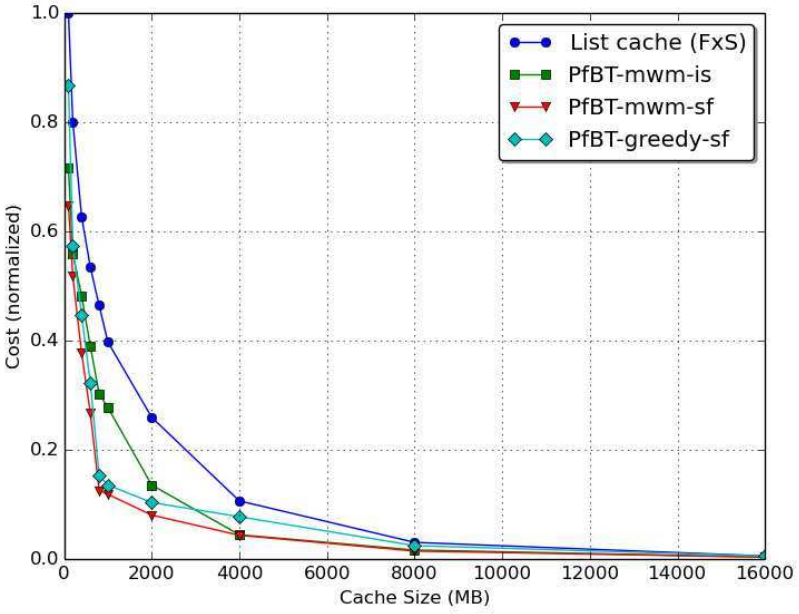

FIGURE 5.13: Performance of the different approaches using the AOL-1 query set.

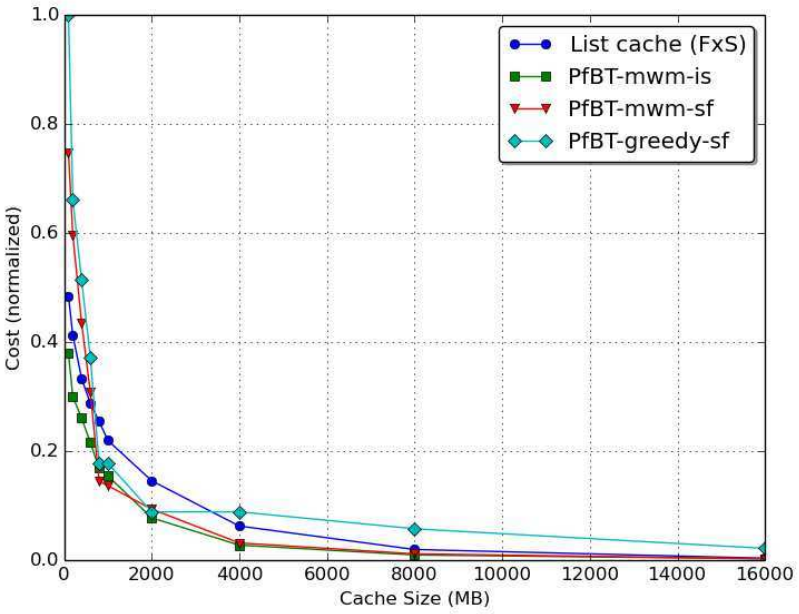

FIGURE 5.14: Performance of the different approaches using the AOL-2 query set.

In the case of the second query set (AOL-2) the behaviour is different. The best strategy is *PfBT-mwm-is* in all cases (up to 57% better than the baseline). The second strategy *PfBT-mwm-fs* only outperforms the baseline for cache sizes greater than 800 MB, while the *PfBT-greedy-sf* do not perform well (on average). A possible explanation is that the greedy approach uses the cache space less efficiently because it allows previously existing terms whose impact is partially compensated by the saving on repeated queries (AOL-1 query set). In this case (AOL-2 query set), there are not extra savings from repeated queries that compensate the *PfBT-greedy-sf* behaviour when selecting term-pairs.

5.3.4 Integrated Cache and Result Cache

To obtain a complete evaluation of the effectiveness of Integrated Cache we consider another set of experiments in which Integrated Cache is combined with a Results Cache. Because of this, we add a Results Cache to our simulation framework using two cache sizes (250K and 500K) and we compare the same strategies used in the previous section. Figures 5.15 and 5.16 show the results.

All the strategies outperform the baseline for the results cache of 250K entries. Both new strategies that include the frequency of the pair in their objective function (*PfBT-mwm-sf* and *PfBT-greedy-sf*) are the best in average except for the last case (16GB of cache size) where the simplest approach gets the best result. According to [100], more

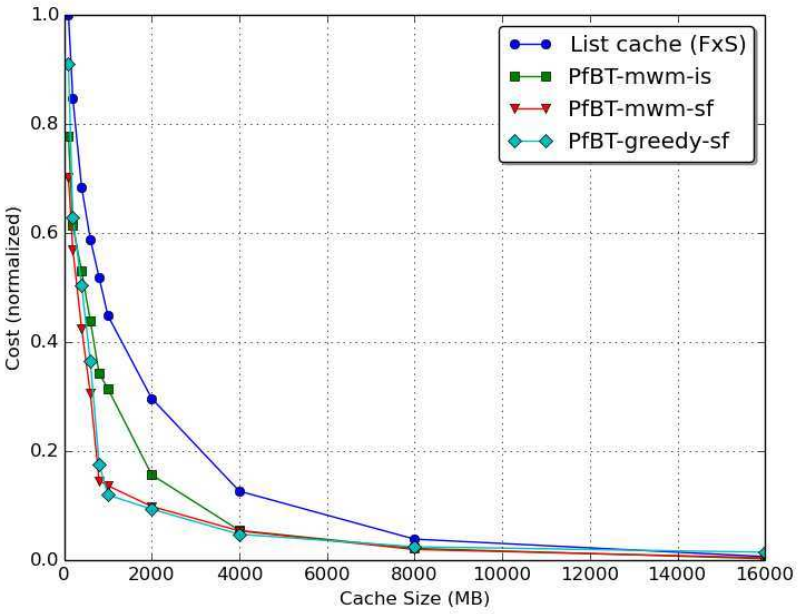

FIGURE 5.15: Performance of the different approaches using the the AOL-1 query set and RC 250k entries.

sophisticated strategies are better when the cache capacity is small, due to the optimized space usage. However, when the cache capacity is big enough a simpler strategy is still useful because the hit rate of the results cache approaches to its upper bound.

Although there are some differences in the remaining case (500K entries), the results show that *PfBT-mwm-sf* and *PfBT-greedy-sf* are again the leading strategies. The best performance is achieved with an Integrated Cache of 4GB, obtaining an improvement close to 70%. When considering all the settings and cache sizes, *PfBT-mwm-sf* becomes the best strategy with a performance improvement around 50%.

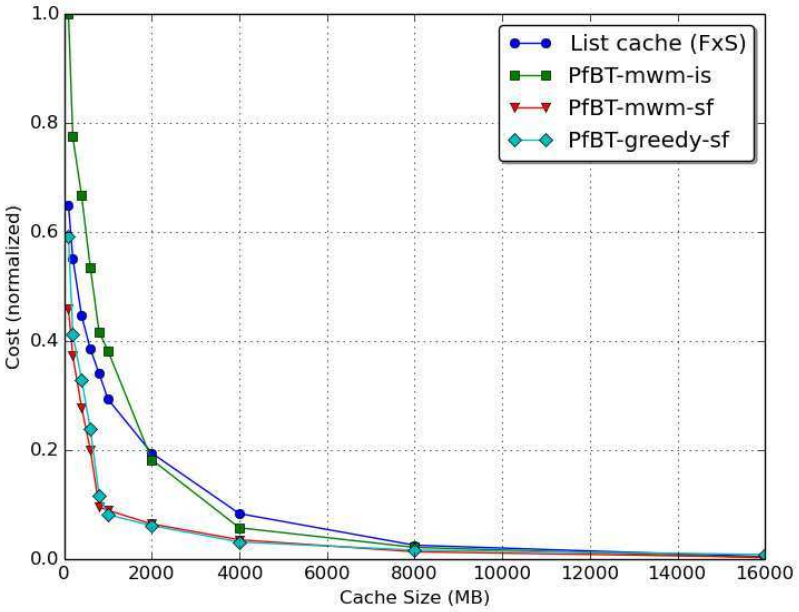

FIGURE 5.16: Performance of the different approaches using the AOL-1 query set and RC 500k entries.

5.4 Summary

We have proposed the *Integrated Cache*, a method to improve the performance of a search system using the memory efficiently to store the lists of term pairs based on a paired data structure along with a resolution strategy that takes advantage of an intersection cache. We consider several heuristics to populate the cache and include an approach based on casting the problem as a maximum weighted matching one.

We provide an evaluation of our architecture using two different document collections and subsets of a real query log considering several scenarios. We also represent the data in cache in both raw and compressed forms and evaluate the differences between them using different configurations of cache sizes. Besides, we consider the existence of a Results Cache in the architecture of the search system that filters out a significant number of repeated queries. The experimental results show that the proposed method outperforms the standard posting lists cache in most of the cases, taking advantage not only of the intersection cache but also of the query resolution strategy.

Caché Integrado de Listas+Intersecciones - Resumen

Como se mencionó en capítulos previos, los cachés de Listas e Intersecciones se implementan a nivel de los nodos de búsqueda. Usualmente, son independientes entre si e intentan ofrecer beneficios desde dos perspectivas diferentes. El caché de Listas alcanza una mayor tasa de aciertos debido a que la frecuencia de los términos individuales es mayor que combinaciones de éstos, pero cada acierto en los últimos genera beneficios mayores debido a que evita el cómputo de dos o más intersecciones entre listas de términos.

Basados en la observación de que algunos términos co-ocurren frecuentemente en diferentes consultas, la motivación aquí es diseñar un caché que pueda capturar los beneficios de ambos, utilizando la misma área de memoria. Se propone, entonces, un caché estático integrado (*Integrated Cache*) que utiliza la memoria de los nodos de búsqueda de forma más eficiente para almacenar listas de pares de términos en una estructura de datos específica, junto con una estrategia de resolución de las consultas que toma ventaja de ésta.

La estructura de datos empleada permite representar las listas de los términos que integran cada par de forma eficiente, utilizando solo un área para la intersección. Concretamente, dados los términos t_i y t_j y sus listas asociadas (ℓ_i y ℓ_j) el almacenamiento del par (t_i, t_j) resulta: $(\ell_i \setminus (\ell_i \bigcap \ell_j))$, $(\ell_j \setminus (\ell_i \bigcap \ell_j))$ y $(\ell_i \bigcap \ell_j)$ en espacios contiguos pero separados. Por lo tanto, se puede retornar el resultado de la intersección o reconstruir la lista de cualquiera de los términos (a partir de computar una unión). Esta propuesta se complementa con un algoritmo que evita almacenar términos duplicados, utilizando una estrategia de redirección específicamente diseñada.

Una cuestión importante aquí es la selección de los pares para poblar el caché. A tal efecto, se evalúan tres opciones *greedy* que consisten en combinar los términos a partir de sus listas, ordenadas de acuerdo a una métrica que resulta competitiva en caching de

Listas. Además, se propone modelar la situación como un problema de optimización combinatoria y se lo resuelve con un algoritmo de *Maximum Weighted Matching*, utilizando dos funciones objetivo diferentes para ponderar las aristas del grafo.

La evaluación de la propuesta se realiza utilizando dos colecciones de documentos, considerando varios escenarios que incluyen los datos sin comprimir y comprimidos y combinaciones con un caché de Resultados. En cuanto a la compresión, se estudia cómo la división de las listas afecta la tasa de ahorro de espacio y bajo qué circunstancias el caché Integrado resulta eficiente. En cuanto al caché de Resultados, la idea es modelar un escenario real de un sistema de búsquedas en el cual un primer nivel de caché en el broker filtra una cantidad significante de consultas repetidas.

Cuando se consideran los datos sin comprimir, se utiliza como referencia (baseline) un caché de Listas que contiene los mejores términos de acuerdo al producto de su frecuencia por la longitud de la lista. Todas las estrategias evaluadas para el caché Integrado superan al *baseline*, siendo la más eficiente aquella que modela el problema como un *matching* en un grafo. Las mejoras alcanzan un 38% en el mejor caso. Si se considera la existencia del caché de Resultados, las mejoras llegan hasta un 22%, mostrando que, inclusive bajo esta configuración, la propuesta resulta útil para reducir tiempo de resolución de consultas.

Para la evaluación de la versión comprimida se utiliza el *codec* PForDelta para el tratamiento de las listas ya que es uno de los enfoques que brinda un buen compromiso entre su tasa de compresión y la velocidad de descompresión. En este caso, la caché Integrada también supera al caché de Listas, incluso cuando se incluye en la arquitectura un caché de Resultados. Considerando todas las configuraciones y tamaños de caché evaluados, la mejor estrategia consiste en seleccionar los pares mediante un algoritmo de *Maximum Weighted Matching* que pondera las aristas utilizando la frecuencia de los pares y la longitud de la intersección entre las listas de los términos, con una mejora en la performance del 50% (en promedio).

Chapter 6

Machine Learning Based Access Policies for Intersection Caches

In Chapter 3 we introduce different strategies to solve a query that take into account the existence of an *Intersection Cache*. Then, we propose the use of the so-called cost-aware caching policies in this context with the aim to reduce the overall query processing time. Roughly speaking, a "caching policy" (a.k.a. eviction or replacement policy) refers to the algorithm and criteria to replace a cached object when the cache is full to make room for a new one. Caching policies ideally evict entries that are unlikely to be a hit or those that are expected to provide less benefit. For example, the popular LRU [78] algorithm evicts the least recently used item from the cache to maximize the hit rate while the Greedy-Dual Size (GDS) strategy [23] uses a cost-aware approach, thus trying to reduce the query processing cost (as we showed in Chapter 4). The "optimal" replacement policy aims to make the best use of available cache space to improve cache hit rates and to reduce the load in servers.

The problem with using only eviction policies in a cache is that all of the items are admitted when a cache miss occurs, thus removing at least one item from memory. This implies admitting some items that will never appear again or are not useful enough for cost savings. To prevent such a situation, the cache eviction policy may be complemented by an *admission policy* that tries to predict the usefulness of the item to be accepted in the cache. We also know that the number of terms in the collection is finite (although big) but the number of potential intersections is virtually infinite and depends on the submitted queries.

It is interesting to note that the use of admission policies is not highly important in every context. For example, let us consider the case of virtual memory systems. This kind of systems uses main memory efficiently by treating it as a cache for an address space

stored in secondary memory (typically, a hard disk), keeping only the active blocks in main memory, and transferring data back and forth between disk and memory as needed, thus managing the whole memory more efficiently. Given the system design, all data blocks must be admitted into the main memory to be used by the operating system, so the use of an admission policy is not a valid option in this case.

The use of admission policies in results caches of search engines has been addressed in the past, however this issue has not received any attention in the context of intersections caches. One important contribution to consider is the work of Baeza-Yates et al. [11] that proposes a cache admission policy based on some query features such as their length and frequency. Basically, that strategy splits the cache area into a two-segment cache (controlled and uncontrolled parts), where one segment is reserved for queries that are considered to be valuable according to an admission policy and the other one is reserved for the remaining queries. This decision is taken on the basis of a threshold that is determined experimentally to prevent singleton queries (i.e. those queries that occur only once in a given query stream) from polluting the controlled part of the cache.

Finally, the recent work of Ozcan et al. [90] introduces an optimization that improves the performance of result caching techniques. Cached results may be stored in two different representations. On one hand, it is possible to store the full result HTML page that contains URLs, document summaries (snippets) and some data that complete the final search result page. On the other hand, an alternative strategy is to cache only the result document identifiers, which take much less space, allowing results of more queries to be cached. Obviously, this second option requires the computation of snippets and the assembly of the final result page. As we described above, this is part of the final phase of the computational process to return the results to the user. These two strategies lead to a trade-off between the hit rate and the average query response latency so they decided to split the result cache into two separate areas: HTML and docID caches. The query results are maintained either in both caches or only in the docID cache. In the last case, this gives a second chance to a query avoiding the full computation in the backend servers thus requiring only the assembling of the HTML response page (at broker level). Another interesting issue in this work is the proposal of a machine learning approach that tries to identify singleton queries on the basis of six query features extracted from query logs. Queries that are classified as singleton are stored only in the docID cache thus increasing the hit rate of the HTML caches and reducing the overall processing cost.

In this chapter we explore a similar idea to our problem of boosting the performance of the Intersection Cache. We introduce an admission policy based on the usage of Machine Learning techniques to decide which items (intersections) should be cached and which ones should not. Basically, our aim is to prevent infrequent or even singleton intersections

(i.e. pairs of terms that appear together only once in the query stream) from polluting the cache. Besides, we incorporate the information of the admission policy into the query resolution strategy by trying to compute only intersections that are likely to appear again in the future, thus increasing the effectiveness of the intersection cache as well. The proposed policy is built on the top of a classification process that predicts whether an intersection is frequent enough to be inserted in the cache.

Herein, we investigate an approach that combines cost-aware intersection caching together with an admission policy that considers the existence of this cache boosted by the aforementioned Machine Learning approach that tries to detect infrequent intersections (not only singletons). This process is done using a reduced set of features to avoid a computation overhead at search nodes level. Finally, our query resolution strategy is also aware of the existence of an intersection cache to rearrange the query terms in a more convenient way.

6.1 Brief Introduction to Query Log Mining

According to [93], the volume of queries submitted to a search engine makes query logs a critical source of information to optimize the precision of results and its efficiency as well. The query distribution, the arrival time of each query and the results that users click on, are valid examples of information derived from query logs that can be analyzed to obtain useful insights of a user's behaviour.

Query log mining is defined as the act of extracting useful information from query log files. This is related to the use and adaptation of data-mining techniques applied to search engine usage data by automatic or semi-automatic means with the aim of extracting implicit previously unknown and potentially useful information from data. Quoting Silvestri [99], query log mining is *"concerned with all those techniques aimed at discovering interesting patterns from query logs of web search engines with the purpose of enhancing either effectiveness or efficiency of an online service provided through the web"*.

Previously submitted queries represent a very important means for enhancing effectiveness and efficiency of search engines. Query logs keep track of information regarding interaction between users and the search engine that contain usage patterns that can be exploited to design new effective methods for enhancing the performance of the system. For instance, this kind of information becomes highly useful to give users query suggestions, spelling corrections or learn how to rank the results. Query log analysis also helps the search engine to improve its result cache efficiency (e.g. enabling the recognition of highly frequent queries).

Basic data-mining techniques such as clustering, classification and association rules have been broadly used to mine web usage data, thus extracting from log files actionable knowledge, like patterns and models of a user's behaviour. For a more thorough overview on this issue, we refer to the work of Silvestri [99], that covers many of the mentioned topics with more details.

In our case, *classification* is the most interesting Machine Learning task used to design a cache admission policy. This is a predictive technique that basically consists of building a model that can assign a label (or a "class") to new instances of any object. Given a collection of labelled items (training set) that are characterized by different attributes (or features) and belong to a specific "class", the task consists of finding a model for the class attribute. Then, using that model, the goal is to assign a label to future (unseen) records as accurately as possible.

6.2 Dynamics of Pairs of Terms in a Query Stream

Before the description of our approach we introduce here a study of the dynamics of the occurrence of term pairs in a query stream. Our aim is to characterize the emergence of new singleton intersections with respect to those with $Freq > 1$ to determine the usefulness of an admission policy. Recall that a single query may generate many intersections; specifically a query of n terms generates $\binom{n}{2}$ intersections. Each of them that has appeared only once after processing q queries (i.e. a singleton) may appear again in a near future, thus leaving this status.

To this extent, we analyze the 12 million queries file in order to understand the dynamics of the appearance of singleton intersections. We split the query stream into bins (or intervals) of 500K queries, compute all possible intersections and derive some trends regarding the number of singletons in each bin and those that remain stable (i.e. $Freq = 1$) across the time. Figure 6.1 shows these results.

The first observation is that the number of singleton and non-singleton intersections remain constant through the bins (in our dataset). This proportion of singleton intersections in each interval is about 69.4% on average (we analyze this behaviour in depth in the next picture). The second observation is that the cumulative frequency through time exhibits a linear growth (with different slopes) in both cases.

We find a slight decline proportional to $1.035x^{-0.03}$ looking into the evolution of the proportion of singleton pairs of terms in the query stream (Figure 6.2). This means that the number of singleton pairs remains large enough to be considered. For example,

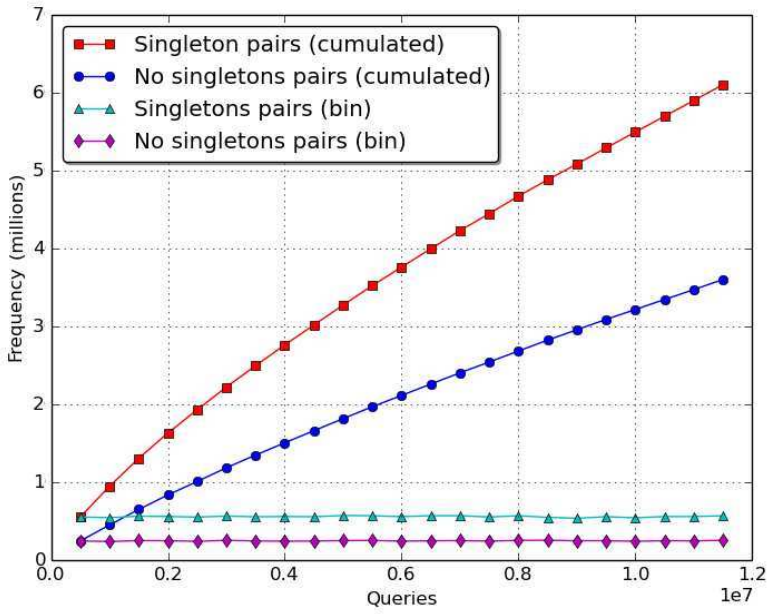

FIGURE 6.1: Number of singleton and non-singleton intersections in the query stream. In both cases, cumulative frequencies show a linear growth proportional to $f(x) = 0.49x + 674,826$ and $f(x) = 0.30x + 248,083$ respectively.

following this trend we may find a proportion of roughly 50.0% after processing 18 billion queries.

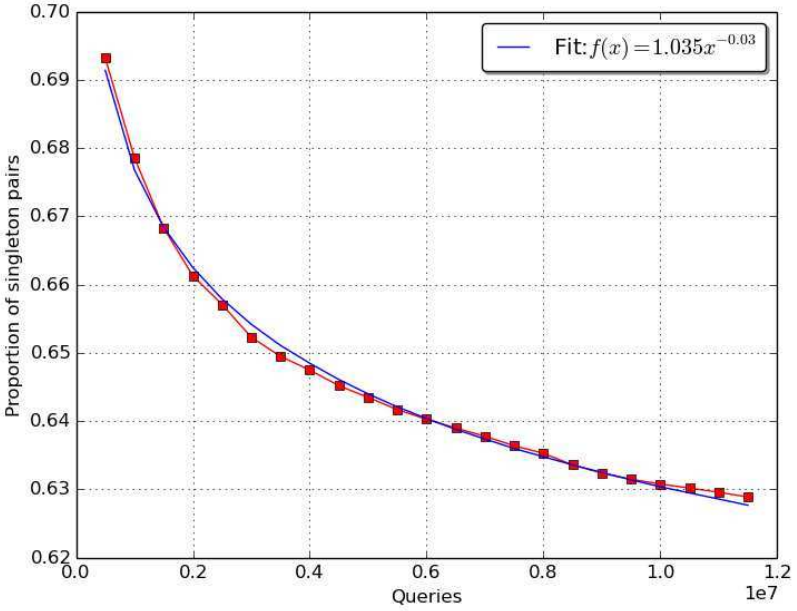

FIGURE 6.2: Evolution of the proportion of singleton pairs in the query stream.

In order to characterize the impact of singleton intersections on the amount of processed data by a search node we compute the sum of the lengths of the posting lists of

both terms (i.e. $|\ell_1|+|\ell_2|$), for the two biggest considered document collections (WB and WS). We also compute the proportion of singletons on the total amount of data and find that the lengths of their posting lists represent about 22.7% and 24.7% for the WS and WB collections respectively. Figure 6.3 shows these results.

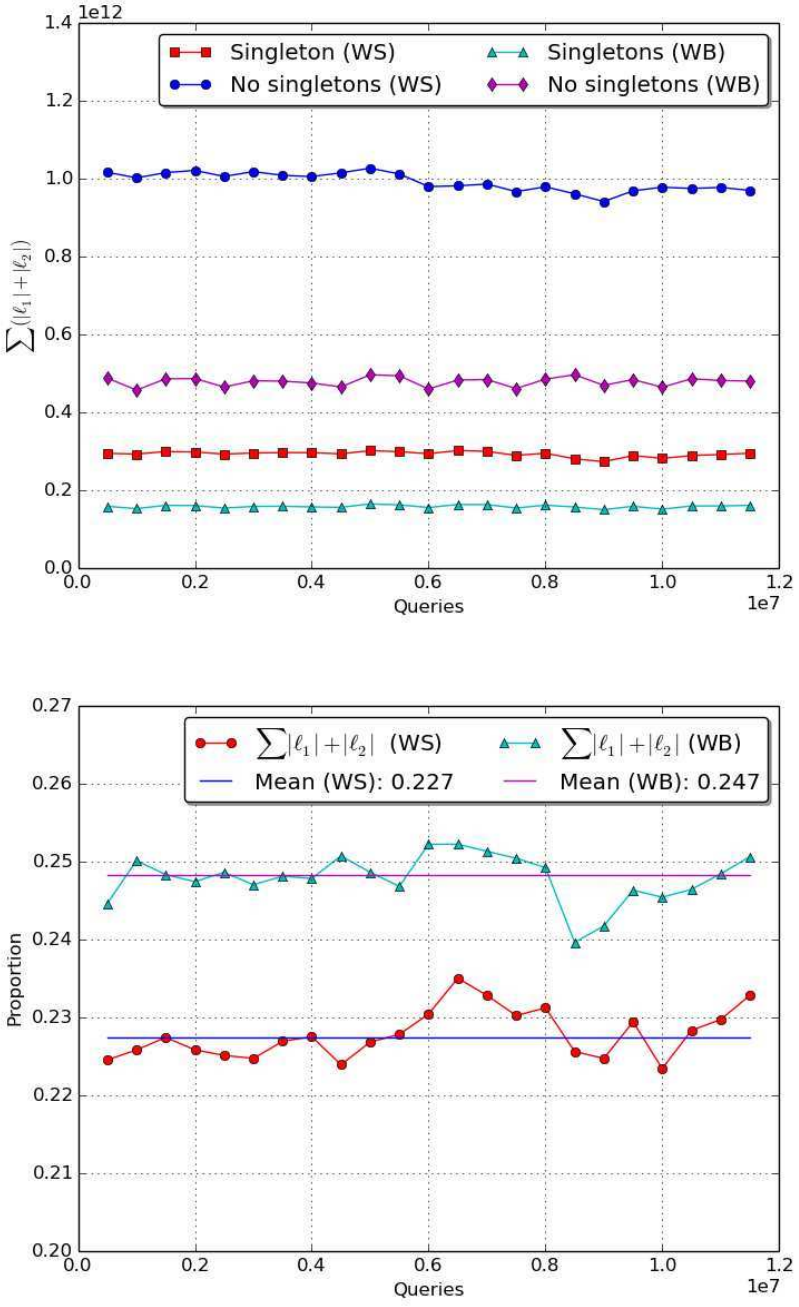

FIGURE 6.3: Sum of the lengths of the posting lists of both terms for WB and WS document collections (top). Impact (proportion) of singletons on the total amount of data (bottom).

We also study the appearance of new pairs of terms in each interval of 500K queries. We find that 50.6% (on average) corresponds to new instances (we exclude the first bin

because all of them are *new* at the beginning). This trend is obviously decreasing and proportional to $f(x) = 27.36x^{-0.258}$ in our sample. Finally, we compute the conversion rate for each pair from singleton intersections to non-singleton ones through the bins. That is, we measure the proportion of intersections that belong to the singletons set at bin b and increase their frequency in the following bin $(b + 1)$, thus no longer considered as a singleton pair. Figure 6.4 shows this result.

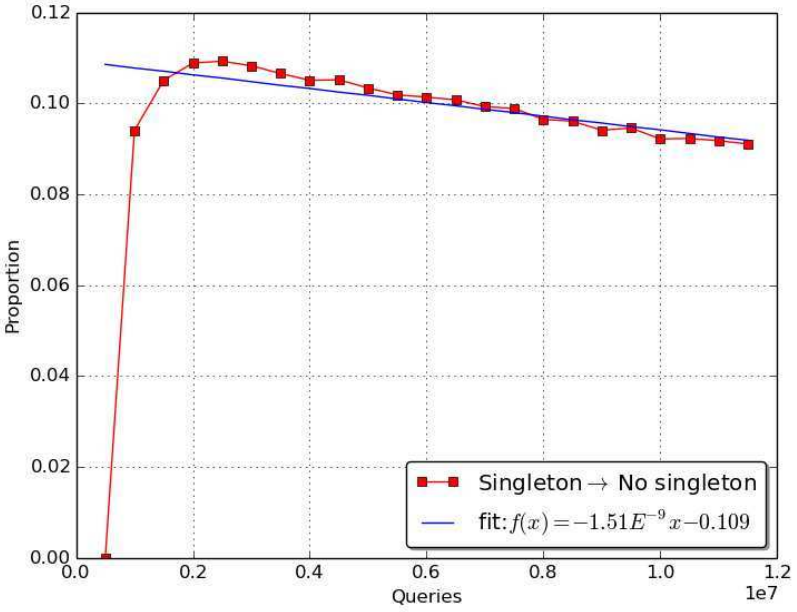

FIGURE 6.4: Conversion rate from singleton to non-singletons through bins.

Excluding the first point that corresponds to the initial bin, we find that the proportion of *converted* pairs is about 9.5% (on average). This behaviour has a decreasing trend proportional to $f(x) = -1.51E^{-9}x - 0.109$. This is particularly interesting because the admission policy may predict some pairs as singletons erroneously that appear again in a near future. To overcome this situation, we propose to handle classification mistakes in a particular way, as we will show in the next sections.

6.3 Machine Learning based Admission Policy

As we previously mentioned, our challenge is to implement an access policy for the Intersection Cache that previously decides which pairs will be allowed into the cache and which ones will not. This policy is implemented at the search node level and runs during the query resolution process. Given a conjunctive query $q = \{t_1, t_2, t_3, ..., t_n\}$ that represents a user's information need (each t_i is a term, and the user wants the list of documents that contains *all* the terms) we adopt a query resolution strategy that first decomposes

the query into pairs of terms. Each pair is checked in the Intersection Cache and the final resolution order is given by considering first the pairs that are present in the cache and afterwards intersecting them with the remaining ones [41]. The intersection cache is dynamically managed using the Greedy-Dual Size (GDS) strategy [23].

Our aim is to identify pairs of terms that appear infrequently (even only once) in the query stream, and therefore caching them does not provide any advantage. The idea is to predict those intersections whose $freq(i) \leq F_T$ to discard them (F_T is the threshold frequency to consider a pair as infrequent).

Our approach works as follows: Given the stream of pairs of terms (intersections) I_n, the admission policy is mapped to a function $f : I_n \rightarrow \{0, 1\}$ that determines, for each pair $i \in I_n$, whether to cache it ($f(i) = 1$) or not ($f(i) = 0$). In our proposal, the function f is built on the top of a machine learning process modeled as a classification problem [1].

Following the *simplicity* design principle (that states that, when there is a doubt during design, choose the simplest solution [24]) we decided to use a minimal number of features to train the classifier. Furthermore, the use of a large number of features requires either the retrieval of data from the inverted index or the precomputation and caching of them. The former requires a large memory budget and the latter is clearly expensive [54]. For example, consider a feature space of 40 values that can be stored in a four-byte data structure and 10 million terms that are stored in the inverted index. Caching these features requires roughly 1.6 GB memory, but the search nodes benefit more from using this memory to cache a larger number of posting lists or intersections. Moreover, the feature information must be updated online, adding additional overhead. The motivation is also due to the fact that the caching process should not affect the overall performance of the search node, even at the expense of lower classification ratios. For that reason, we only consider four features of each pair of terms (t_i, t_j) to build the classifier:

- TF_{ti} and TF_{tj}: The individual term frequencies of t_i and t_j (respectively) in a query log used as a training set.
- DF_{ti} and DF_{tj}: The document frequencies of t_j and t_j in the document collection that the search node holds.

Note that these features can be computed offline and require a small amount of memory. Another set of features is also possible to include, for example, term lengths, joint frequency (of t_i and t_j in a training set), number of different users that submitted a query, an so on. However, we consider a reduced set of them to minimize the processing overhead at running time. We also prefer features of individual terms rather than the

combination of both to be able to predict *unseen* combinations (new pairs of terms that do not appear in the training set). The accuracy of the classification process using more features and its impact on search performance deserve a deeper research.

Then, the search nodes only use the classification rules derived from them. For simplicity, we build a binary classifier that maps each pair of terms into two possible classes (C):

$$C(t_i, t_j) = \begin{cases} 0, & \text{“Infrequent pair”} \\ 1, & \text{“Frequent pair”} \end{cases}$$

A pair (t_i, t_j) is classified as “infrequent” if it is more likely that $freq(t_i, t_j) \leq F_T$. Otherwise, it is classified as “frequent”.

Furthermore, we modify the query resolution strategy using the information of the classifier. We start from the S4 strategy introduced in [41]. This basically takes a query sorted according to the length of the posting lists of its terms (from shorter to longer). Then, it tests all the possible two-term combinations in the cache in order to maximize the chance of a hit and rewrites the query according to this result, giving precedence to the pairs that are found in cache.

We improve this strategy by incorporating the classification decision into the query resolution process in a two-phase procedure. In the first phase, all two-term combinations are generated as in the S4 strategy and the admission policy is applied to determine which pairs are worth to be cached or not. The second phase is similar to the original strategy and because of this all the pairs are looked for in the cache selecting only those that are already cached. The complete process can be detailed as follows:

1. $P_1 \leftarrow$ all two-term combinations.
2. $P_2 \leftarrow$ pairs from P_1 that pass through the admission policies (phase one).
3. $P_3 \leftarrow$ pairs from P_2 that are present in cache (phase two).
4. Rearrange the query using first the pairs from P_3. If not all query terms are covered, complete with the pairs from P_2 that contain the missing terms and complete with the pairs from P_1 if necessary.
5. Finally, add the remaining term if the number of query terms is odd.

At the end of this process, the query is rearranged by considering first those pairs that are likely to appear again, thus preventing less useful ones to pollute the cache.

To illustrate clearly, let us consider the following example: let $q_i = \{t_1, t_2, t_3\}$ and its sorted version according to their lists lengths is $q_i' = \{t_3, t_1, t_2\}$. In the first phase the pairs $(t_3, t_1), (t_3, t_2), (t_1, t_2)$ are tested. Let us assume that the pair (t_3, t_1) does not pass the admission test so only the remaining two pairs $(t_3, t_2), (t_1, t_2)$ are considered next. For the second phase, let us also assume that the pair (t_1, t_2) is already cached so the query is scheduled to be solved like: $(\ell_1, \ell_2) \bigcap \ell_3$.

As we previously mentioned, the classifier uses only four features of each pair in the query, namely $TF_{t1}, TF_{t2}, DF_{t1}, DF_{t2}$, to predict the class that determines whether to cache it or not.

6.3.1 Cumulative-Frequency Control Algorithm

In order to implement the admission policy we try different classical classification algorithms that match our problem, namely Naive Bayes and Decision Trees [1], that are popular supervised learning algorithms. According to [111], these are identified as two of the most influential algorithms in data-mining tasks.

However, as we will show in next sections, the performance of the classification process is limited and it translates into admission mistakes in the cache. For a better understanding of the types of errors that the classifier incurs we use its confusion matrix [60][103], a special contingency table that represents the count of a classifier's predictions with respect to the actual outcome on the labeled learning set. As an example, Table 6.1 shows the confusion matrix of the Decision Tree classifier for the case $F_T = 1$.

		Prediction	
		Frequent	**Infrequent**
Truth	**Frequent**	0.417 TP (True Positives)	0.102 FN (False Negatives)
	Infrequent	0.147 FP (False Positives)	0.334 TN (True Negatives)

TABLE 6.1: Confusion Matrix of the Decision Tree classifier for $F_T = 1$. Each cell corresponds to the proportion of classified instances.

The two errors have different effects in the behaviour of the intersections cache. In the case that an item (pair) is "frequent" but the classifier labels it as "infrequent" the admission policy avoids caching it. The opposite case happens when the item is classified as "frequent" although it is not, and thus it is sent to cache. The former error is more severe than the latter because an item that is likely to appear again is never cached. In the example, the proportion of items that fall in this error type represents around 10% of the total classified items (up to 15% in other configurations). The second error is less severe

because "infrequent" pairs sent to the cache are presumably evicted by the replacement policy in a short period of time.

In order to mitigate the first error we extend our admission policy with a *double-check* condition. We build on the ideas of the 2Q buffer management policy [55] that maintains two queues ($A1$ and Am) to place pages as either *hot* or *cold*. Simplifying the idea, on the first reference to a page, 2Q places it in the $A1$ buffer which is managed as a FIFO queue. If the page is referenced again during its $A1$ residency, then it is moved to the Am queue, which is managed as an LRU queue. If a page is not referenced while on $A1$, it is probably a cold page, and will be removed when this buffer becomes full.

Based on this idea, we develop the double-check for items that the classifier labels as "infrequent". Recall that we define an item as "infrequent" if its cumulative frequency is lower than a certain threshold (F_T). We maintain a FIFO queue (CFC) of size s that accumulates the frequency of a set of pairs. When the classifier labels a pair as "infrequent" a second test is made using the information of CFC queue. If the cumulative frequency of a pair exceeds F_T this pair is cached anyway. We name this strategy "Cumulative Frequency Control" (CFC). We implement two versions of this approach: the first one only checks for the presence/absence of the item in the CFC queue ($freq(i) = 1$). The second version is a general case where we check for a threshold frequency (instead of just presence/absence) as $freq(i) \leq F_T$. Algorithm 3 shows the modified version of our admission policy using the general case of the CFC algorithm.

It is important to note that storing statistics of all possible items over the recent history is considered prohibitively expensive for practical implementations. We also have to recall that computing all possible two-terms combinations over a stream of queries result in a virtually infinite set so we limit the size of CFC queue. Take into consideration that the implementation should be different for $F_T = 1$ or larger. In the former case, we only need to test whether an item is a member of the queue. This may be accomplished using space-efficient probabilistic data structures, such as Bloom filters [18]. For the latter, a more sophisticated technique is required. An interesting approach is to estimate this statistics in a highly efficient manner using sketching techniques for approximate counting [45]. In both cases, the dynamics should be considered in order to detect and control how (and when) these approximate data structures should be cleared.

Algorithm 3: Admission policy with cumulative frequency control

Input: (t_1, t_2): Pair of terms, $TF_{t1}, TF_{t2}, DF_{t1}, DF_{t2}$, CFC queue, F_T
Output: *decision* : $\{0, 1\}$
$pair \leftarrow (t_1, t_2)$
$decision \leftarrow 0$
if *(isFrequent*$(TF_{t1}, TF_{t2}, DF_{t1}, DF_{t2})$*)* **then**
  send *pair* to the second stage
  (or insert *pair* in cache)
  $decision \leftarrow 1$
else
  if *(exists(CFC, pair))* **then**
    $CFC[pair]++$
  else
    insert *pair* in CFC
  end
  if $(CFC[pair] > F_T)$ **then**
    send *pair* to the second stage
    (or insert *pair* in cache)
    $decision \leftarrow 1$
  else
    do nothing with *pair*
  end
end
return *decision*;

6.4 Experiments and Results

6.4.1 Data and Setup

We evaluate the proposed admission policies using the two biggest data collections that we describe in Section 3.3.1 (WB and WS). We use a subset of 8M queries to compute statistics and train the classifier and the remaining ones to test the algorithms. The total amount of memory reserved for the cache ranges from 100MB to 16GB in all cases. In these experiments, we assume that the inverted index resides in the main memory. As we mentioned before, this is a realistic setup for modern search systems. The *extra* available memory is used by the *Intersection Cache*.

Finally, we consider the overall cost incurred by the different strategies to process all queries as a performance metric following the same comparison framework as in the previous chapters.

Classification algorithms setup: As we mentioned above, we select the Naive Bayes (NB) and Decision Trees (DT) classifiers. In our experiments we find that the NB classifier performs poorly to address this problem so we decided to improve the efficiency of DT-based methods by the use of a Random Forest (RF) ensemble [19]. This method

basically works by constructing many decision trees during training time and outputting the class that is the mode of the classes of the individual trees. Unlike single DT which are likely to suffer from high variance or high bias, RF methods use averaging to find a natural balance between the two extremes, thus minimizing the overfitting to the training set.

We test different values of the maximum depth that the trees are allowed to grow and train our classifiers with 8 million of unique pairs of terms of positive and negative examples (i.e. frequent and infrequent pairs). Figure 6.5 shows an example of a decision tree generated for a particular configuration (WB Collection, $F_T = 1$).

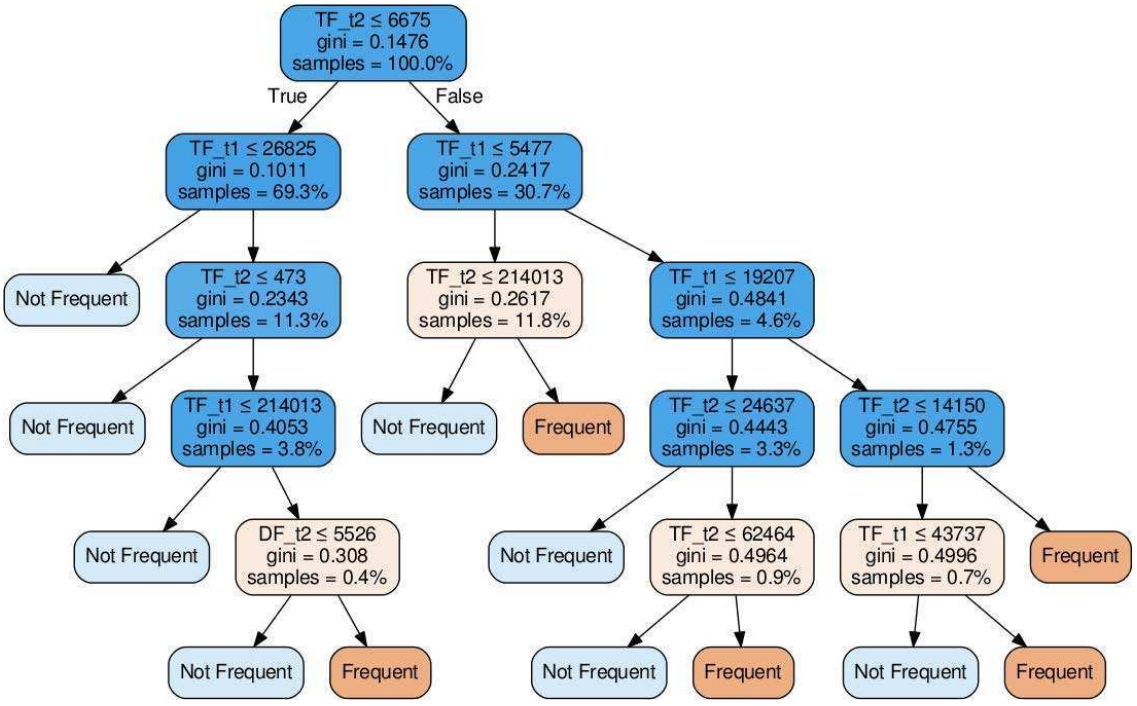

FIGURE 6.5: Example of a decision tree generated using the four aforementioned features (WB Collection, $F_T = 1$)

Decision trees are self–explanatory and each branch of the tree can be mapped into a decision rule that describes the relationship between inputs (features) and targets (classes). Thus, this representation is considered as comprehensible. For example, starting from the root node, decision rules might be built as[1]:

- IF $(TF_{t2} \leq 6675)$ AND $(TF_{t1} > 26825)$ AND $(TF_{t2} \leq 473) \Rightarrow$ "NOT FREQUENT"
- IF $(TF_{t2} > 6675)$ AND $(TF_{t1} > 5477)$ AND $(TF_{t1} > 19207)$ AND $(TF_{t2} > 14150) \Rightarrow$ "FREQUENT"

In general, the two most important features are TF_{t1} and TF_{t2} because they allow to separate the observations in the target classes in most of the cases. This is an interesting

[1] In this example, we derive the rules from the tree directly, and do not eliminate overlapping conditions. This process must be done before the implementation.

observation that may help to build an admission policy based upon different criteria (instead of a machine learning approach).

Then, we also perform a k-fold cross validation. This process basically consists of partitioning data into k randomly chosen subsets (folds) of roughly equal size. Each subset is used to validate the trained model using the remaining $(k-1)$ folds. This process is repeated k times so that each fold is used once for validation.

To asses the performance of the classification process we use the *Precision* (P) and *Recall* (R) metrics. *Precision* (also called *"Positive Predictive Value"* in Machine Learning field) is the proportion of retrieved instances that are relevant. Intuitively, P measures a classifier's ability to minimize the number of false positives. In our case, *Precision* measures the fraction of (true) frequent intersection over the number of instances classified as frequent. P is defined as[2]:

$$P = \frac{TP}{TP + FP}$$

Complementarily, *Recall* (also called "*Sensitivity*") measures the proportion of all positive instances that are correctly identified (e.g., the percentage of frequent intersections correctly classified over all true frequent intersections). Intuitively, R measures a classifier's ability to find all the positive instances. R is defined as:

$$R = \frac{TP}{TP + FN}$$

Finally, the best *Precision* values achieved by different configurations of the classifiers are between 69% and 75% while the best *Recall* values are between 70% and 85%. As this is not good enough, we implement the CFC algorithm (described in Section III-A) to complement the classification process, improving the performance of the admission policy.

6.4.2 Baseline

We initially run different experiments to determine the cost savings achieved when infrequent pairs are not cached. The lower bound of performance (*noap*) corresponds to the intersection cache without the admission policy. Then, as an upper bound, we use a kind of *clairvoyant* approach that *knows* all infrequent pairs according to different values for the frequency threshold, namely $F_T \leq \{1, 5, 10, 14\}$. We experimentally determine $F_T \leq 14$ as the threshold configuration that obtains the highest cost savings.

[2]TP, FP and FN denote 'True Positive', 'False Positive' and 'False Negative' rates respectively, as shown in Table 6.1.

Figure 6.6 shows the results for both collections as the size of the cache varies. As expected, all the *clairvoyant* approaches outperform the baseline. These results reinforce the idea that using an admission policy in combination with the intersection cache offers extra cost savings.

For the WB dataset, average improvements are around 4.6%, 8.9%, 11.8% and 12.6% for $F_T = 1, 5, 10, 14$ respectively. The best performance gain (17.3%) is achieved with $F_T \leq 14$ and 4GB of cache size. The behaviour is quite similar for the WS dataset with improvements around 4.0%, 6.3%, 8.8% and 11.9%. The best case is achieved with the same configuration as in the previous one, giving up to 15.7% of performance boost.

6.4.3 Admission Policy using DT and RF classifiers

In this section we report the performance results when the admission policy described in Section 6.3 is used. We refer to the previous results and select the best *clairvoyant* algorithm ($clair - F_T \leq 14$) as the upper bound.

When using the RF-based classifier (AP-RF), the admission policy $F_T \leq 14$ improves the performance of the query resolution process by 4.6% (on average) for cache sizes up to 2 GB for the WB collection (with a peak of 7.2% and 800 MB of cache space). Looking at the WS collection, the performance is slightly better achieving up to 7.1% of improvement in the same configurations as in the WB collection. Figure 6.7 shows the results.

The behaviour of the admission policy using the DT-based (AP-DT) classifier for the WB collection is slightly worse (4.2% on average and 6.4% in the best case, with $F_T \leq 5$) but this result improves considerably for the WS collection rising up to 9.1% on average ($F_T \leq 14$). The outcome is illustrated in Figure 6.8. This result is consistent with a better (although poor) performance of the classifier. As we mentioned before, the best classification precision achieved is about 75%.

As we mentioned in Section 6.3.1, we design an approach developed specifically to handle classification errors. In the next section we report the results of the admission policy complemented by the the CFC algorithm. This approach leads to better performance improvements, even for large cache sizes.

6.4.4 Admission Policy with CFC algorithm

We use the same configurations as in the previous experiments with the aim of evaluating and testing the new proposal under similar conditions. Specifically, we use the same values of the frequency threshold ($F_T = \{1, 5, 10, 14\}$) to consider a pair as infrequent.

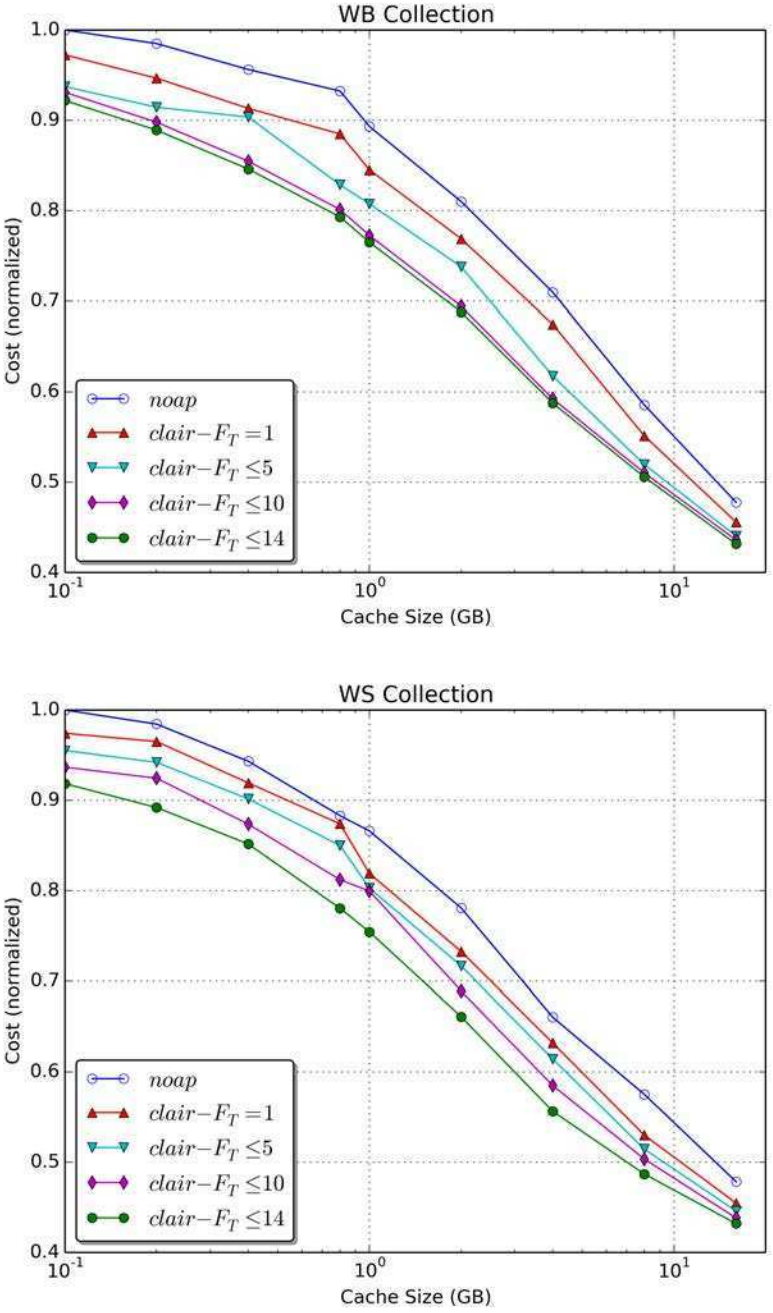

FIGURE 6.6: Performance comparison between the baseline (*noap*) and the *clairvoyant* algorithm with different thresholds ($clair - F_T \leq x$) for the two document collections. (x-axis in log scale to get a clearer view)

We run the classification process again for each case (to simplify, we only run the DT-based classifiers under similar configurations to those used to establish the baselines).

Then, we analyze the behaviour under two different configurations. On the one hand we run the Algorithm 3 considering the presence/absence of an item in the CFC queue. On the other hand, we explicitly count the cumulative frequency of a set of pairs and

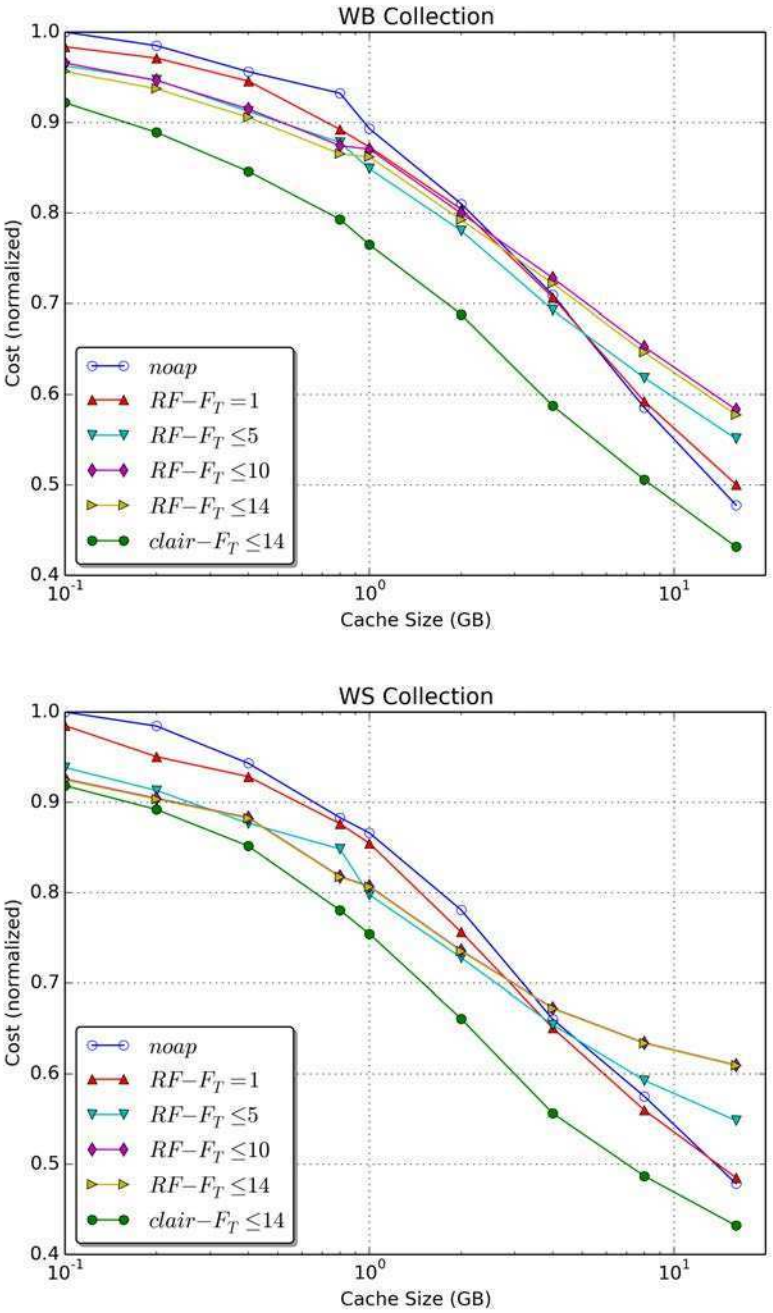

FIGURE 6.7: Performance of the algorithms using the Random Forest classifier for both collections with respect to the lower (*noap*) and upper ($clair - F_T \leq 14$) bounds (x-axis in log scale)

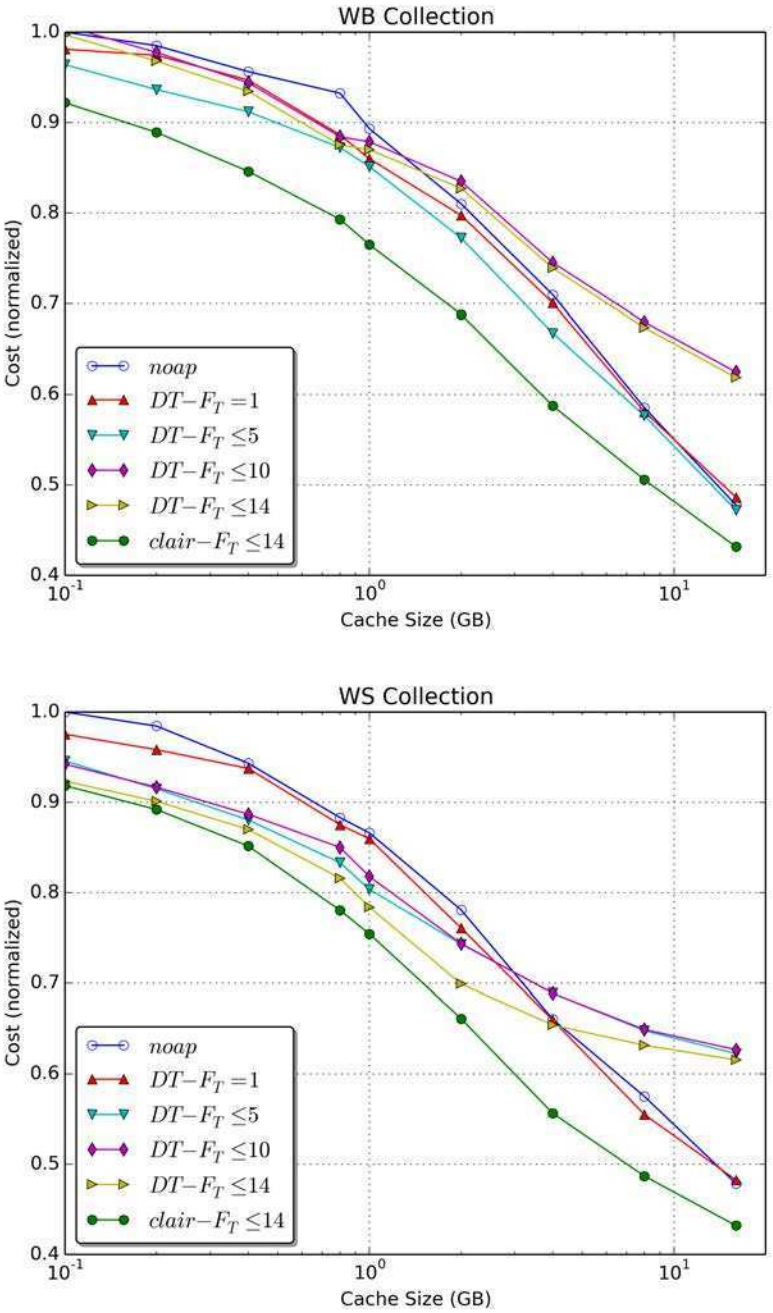

FIGURE 6.8: Performance of the algorithms using the Decision Tree classifier for both collections with respect to the lower (*noap*) and upper ($clair - F_T \leq 14$) bounds (x-axis in log scale)

check the condition $CFC[pair] > F_T$ to consider *pair* as frequent. For this, we use a CFC queue of 300,000 elements.

As a first observation, the CFC algorithm avoids the degradation of the performance for bigger cache sizes reducing the execution times in all the cases. This is clearly viewed in Figures 6.9 and 6.10 (with CFC) with respect to Figures 6.7 and 6.8 (without CFC).

– Results when checking presence/absence of *pair* in CFC queue (AP-DT-CFC-bin)

Figure 6.9 summarizes the results of this policy. Here, we consider again the best *clairvoyant* algorithm ($clair - F_T \leq 14$) as the upper bound. The first interesting observation is that all the algorithms that set $F_T \geq 5$ outperform the baseline for all the cache sizes considered (in both collections).

Specifically, in the case of WB collection, the average gains are around 2.1%, 4.5%, 6.2% and 7.2% (with a top value of 11.4% for 800MB of cache size and $F_T \leq 14$). The behaviour for WS collection is similar, with average gains about 1.7%, 7.4%, 7.8% and 8.8% (top value is 11.9% for 4GB of cache size and $F_T \leq 14$).

– Results when checking the frequency condition (AP-DT-CFC-freq)

The best performance of the admission policy is achieved using this approach (Figure 6.10). In the case of the WB collection, performance improvements (on average) are around 4.2%, 6.7% and 7.7% (top value of 11.8% for 800MB of cache size and $F_T \leq 14$). For the WS collection, average gains are around 8.6%, 9.2% and 7.5% (with a top value of 11.7% for 1GB of cache size and $F_T \leq 10$).

Finally, we plot the best series of each strategy for the two document collections to allow a more clear comparison (Figure 6.11). We also summarize the best results in Tables 6.2 and 6.3. In both cases, columns report the average performance gain with respect to the baseline (*noap*), the higher cost saving and the best configuration respectively. The access policy with the CFC algorithm that checks the frequency condition (AP-DT-CFC-freq) performs better in general, achieving a top improvement around 11.7% for the case where the *clairvoyant* reaches a 13% of cost saving.

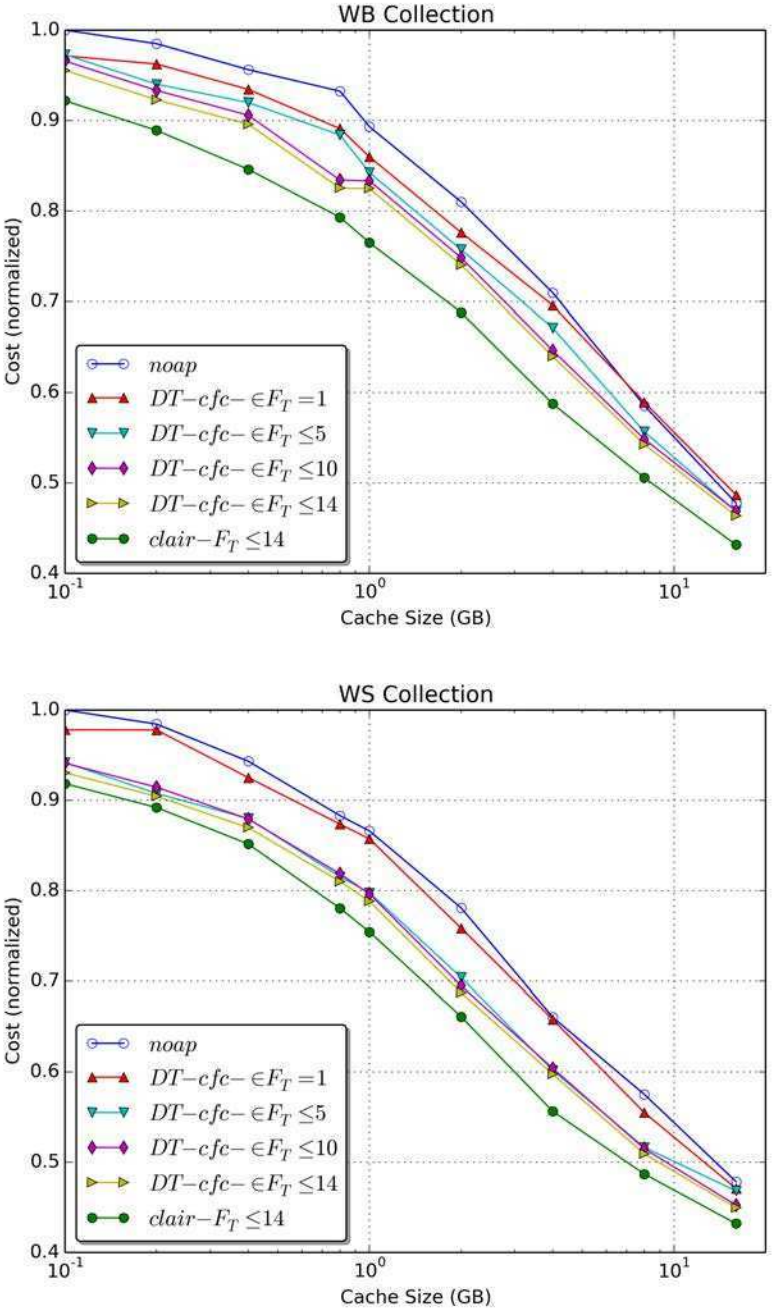

FIGURE 6.9: Performance of the algorithms using the Decision Tree classifier with CFC (checking presence/absence). x-axis in log scale.

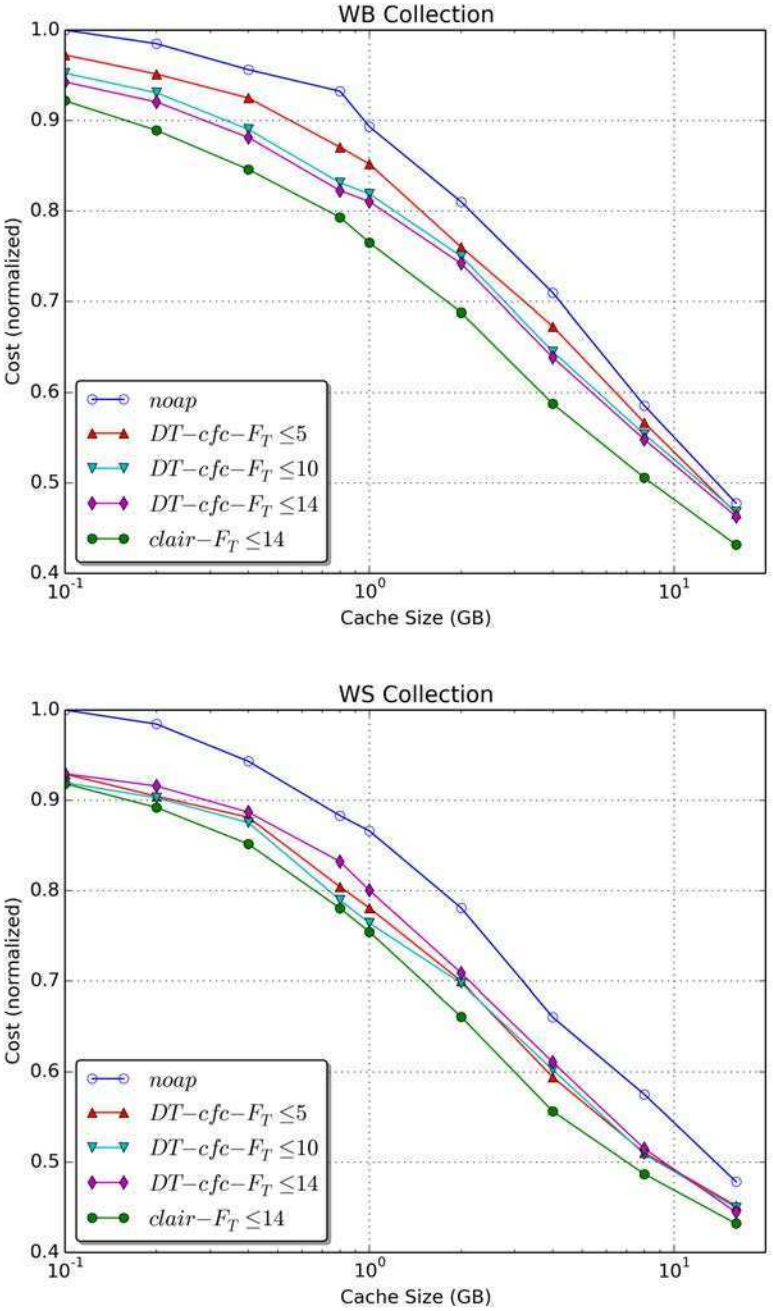

FIGURE 6.10: Performance of the algorithms using the Decision Tree classifier with CFC (checking frequency condition). x-axis in log scale.

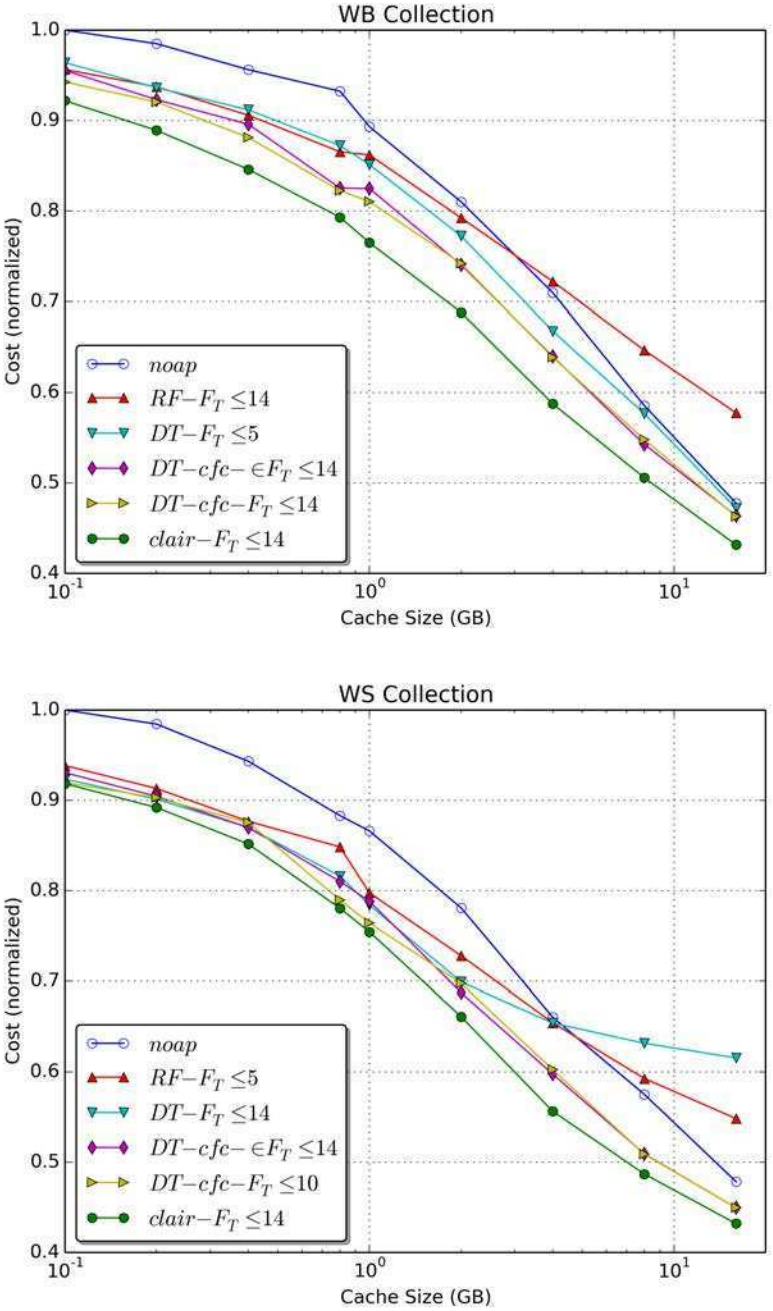

FIGURE 6.11: Performance comparison of the best configuration of each algorithm. x-axis in log scale.

	Average	**Best**	**Config**
AP-RF	**4.6%**	**7.2%**	$F_T \leq 14$
AP-DT	**4.2%**	**6.4%**	$F_T \leq 5$
AP-DT-CFC-bin	**7.2%**	**11.4%**	$F_T \leq 14$
AP-DT-CFC-freq	**7.7%**	**11.8%**	$F_T \leq 14$

TABLE 6.2: Results summary for the WB Collection.

	Average	**Best**	**Config**
AP-RF	**7.1%**	**9.2%**	$F_T \leq 14$
AP-DT	**9.1%**	**10.4%**	$F_T \leq 14$
AP-DT-CFC-bin	**8.8%**	**11.9%**	$F_T \leq 14$
AP-DT-CFC-freq	**9.2%**	**11.7%**	$F_T \leq 10$

TABLE 6.3: Results summary for the WS Collection.

6.5 Summary

In this chapter we propose and evaluate an admission policy for intersection caches based on a Machine Learning approach. We model the admission decision as a classification problem whose main goal is to avoid infrequent pairs of terms to pollute the cache. We also integrate the admission policy into the query resolution strategy and incorporate a Cumulative-Frequency Control algorithm specifically designed to handle classification errors, that improves the cache performance.

The evaluation using a simulation framework and real datasets shows that the proposed admission policy boosts the intersection cache leading to an improvement of the total processing cost close to 12% with respect to the same algorithm without the admission control. In general, the best results correspond to the CFC algorithm that checks the frequency condition.

Políticas de Acceso para Caché de Intersecciones - Resumen

En general, cuando se mencionan políticas de caching se hace referencia al mecanismo utilizado para desalojar un ítem del caché para hacer lugar a uno nuevo, es decir, se refieren a política de reemplazo. Éstas se basan en los diferentes parámetros que caracterizan a los ítems que se desea mantener (frecuencia, patrón de acceso, costo, entre otros). El problema de solo utilizar una política de reemplazo es que todos los ítems son admitidos (incluso aunque no aporten suficiente beneficio y no se repitan nuevamente) y cuando el caché se encuentra completo esto implica el desalojo de un elemento existente. Para prevenir esta situación, existe otra política posible a implementar en un caché que determina previamente si un ítem es suficientemente "bueno" como para ser conservado en memoria. Este tipo de políticas se denominan de "admisión" y complementan a las primeras tratando de predecir la utilidad de aceptar el nuevo ítem en caché.

En este capítulo se propone, diseña y evalúa una política de admisión para el caché de Intersecciones utilizando técnicas de *Machine Learning*. Concretamente, se modela la admisión como un problema de clasificación y se intenta determinar si una intersección entre dos términos es suficientemente frecuente (o no) para ser admitida. Además, se incorpora la decisión de admisión dentro de la estrategia de resolución de la consulta a los efectos de determinar el orden de ejecución de cada paso del proceso.

Como el objetivo es identificar pares de términos que no aparecen frecuentemente, la idea es predecir aquellos ítems i cuya $frequency(i) \leq F_T$, donde F_T es un umbral de frecuencia particular. El enfoque funciona de la siguiente manera: dado un flujo de pares de términos (intersecciones) I_n, la política de admisión se mapea a una función $f : I_n \rightarrow \{0, 1\}$ que determina para cada par $i \in I_n$, si enviarlo al caché ($f(i) = 1$) o no ($f(i) = 0$). Esta función se construye sobre un algoritmo de clasificación binario proveniente del área de Machine Learning que utiliza un mínimo número de características (*features*) a los efectos de que el proceso no incurra en overhead que afecte la performance.

Complementariamente, se diseña e implementa un algoritmo de doble chequeo que maneja los potenciales errores de clasificación, incrementando la performance del conjunto completo. Éste se basa en una cola FIFO que, básicamente, es consultada ante un fallo de admisión y permite corregir la decisión. Se analizan dos casos: en el primero, solo se chequea presencia/ausencia del ítem en la cola y en el segundo se considera la frecuencia acumulada del ítem mientras reside en la misma.

La propuesta se implementa a nivel de los nodos de búsqueda utilizando la estrategia de resolución S4, un caché dinámico basado en el algoritmo Greed-Dual Size (GDS) y se asume que el índice invertido reside completamente en memoria. Para la evaluación se construyeron diferentes clasificadores basados en Árboles de Decisión (*Decision Trees*) y Ensambles (*Random Forests*), modificando los umbrales de frecuencia que determinan si un ítem es frecuente o no lo es. La comparación se realiza, por un lado, respecto del uso del caché de Intersecciones sin política de admisión (cota inferior) y por el otro, un algoritmo "clarividente" que conoce de antemano el resultado y ofrece la mejor performance (cota superior).

Como datos de prueba se utilizan dos colecciones web y un conjunto de consultas reales. Las consultas se dividen en dos porciones, una para el entrenamiento de los clasificadores y otra para las pruebas, siguiendo la metodología habitual del área. Los resultados globales muestran que la política de admisión complementada con el algoritmo de doble chequeo mejora la performance del caché de Intersecciones todos los casos. Las mejoras van del 4.6% al 9.2% en promedio para diferentes configuraciones alcanzando un máximo cercano al 12% con respecto al mismo algoritmo sin control de admisión, utilizando el doble chequeo y considerando la frecuencia del ítem en la cola.

Chapter 7

Conclusions and Future Work

Web search engines need to process huge amounts of data to build sophisticated structures that support search. The ever growing amount of users and information available on the web impose performance challenges to web search engines. Query processing consumes a significant amount of computational resources so that many optimization techniques are required to enable that search engines efficiently cope with heavy query traffic.

One of the most important mechanisms to address this kind of problems is caching, that can be applied at different levels of the WSE architecture. Query result caching and posting lists caching in the context of search engines have become a popular research topic in the last decade. However, Intersection caching has not received much attention. In this thesis, we focus on this problem and propose the use of cost-aware caching policies that may contribute to the performance improvement of search engines.

We first propose different query resolution strategies that take into account the existence of an intersection cache. Then, we introduce a study of cost-aware intersection caching using four query resolution strategies and different combination of caching policies. Basically, we consider static, dynamic and hybrid caching replacement policies adapted to our problem and compare them against common cost-oblivious policies used as a baseline. To this extent, we provide the evaluation under two scenarios: (a) when the inverted index resides on disk and (b) when the whole index resides in the main memory. The overall results show that S4 is the best strategy, and this is independent of the replacement policy used. Let us remember that S4 tests first all the possible two-term combinations in the cache in order to maximize the chance to get a cache hit. Using a disk-resident index, S4 becomes up to 61% better that the remaining policies (on average) and the performance improvement reaches a 24% for memory-resident indexes. In the same way, we can observe that cost-aware policies are better than cost-oblivious policies, thus improving

the state-of-the-art baselines (39%, 24% and 10% better for static, dynamic and hybrid policies, respectively).

As another contribution of this thesis, we propose the Integrated Cache, a method to improve the performance of a search system using the memory efficiently. This cache stores the lists of pairs of terms based on a paired data structure along with a resolution strategy that takes advantage of an intersection cache. We also represent the data in cache in both raw and compressed forms and consider several heuristics to populate the cache. Our evaluation shows that the proposed method outperforms the standard posting lists cache in most of the cases: up to 38% and 53% using uncompressed and compressed data, respectively. This approach takes advantage not only of the intersection cache but also the query resolution strategy.

Finally, we propose and evaluate an admission policy for intersection caches based on a Machine Learning approach. We model the admission decision as a classification problem whose main goal is to avoid infrequent pairs of terms to pollute the cache. We also integrate the admission policy into the query resolution strategy and incorporate a Cumulative-Frequency Control algorithm specifically designed to handle classification errors that improves the cache performance. The evaluation using different frequency thresholds shows that the proposed admission policy boosts the performance of the intersection cache, leading to an improvement of the total processing cost close to 12% with respect to the same algorithm without the admission control.

Different research directions arise from this work. We plan to design and evaluate specific data structures for cost-aware intersection caching. The optimal combinations of terms to intersect and maintain in cache seem to be still an open problem, especially for cost-aware approaches. It would be also interesting to evaluate the use of other techniques such as list pruning and compression because under the scenario where the full inverted index is in the main memory the intersection cache becomes the second level cache in the architecture of a search engine.

In the case of the Integrated Cache, it would be interesting to extend the approach by considering trigrams and other combinations of terms. Besides, the evaluation of different compression encoders according to the distribution of the DocIDs in the integrated representation is an issue for a future work. It would be interesting to model this problem analyzing the space-time tradeoff. Another interesting open question concerns the design and implementation of a dynamic version of this cache. In this case, the admission and eviction policies should contemplate not only the terms but also the pairs.

Finally, different improvements to the admission policy are clearly possible. We plan to use the feedback of our algorithms to improve the classification performance and consider new strategies to detect infrequent pairs more accurately. Another interesting direction is to investigate the dynamics of the infrequent/frequent status of each pair of terms to establish correct time slots to retrain the prediction model in an online fashion. We also plan to define a family of CFC algorithms that adapt better to different data streams by the use of diverse queue strategies instead of the FIFO approach only.

Conclusiones y Trabajos Futuros - Resumen

Los motores de búsqueda web (WSE) necesitan procesar enormes cantidades de datos para construir estructuras sofisticadas que soportan la búsqueda. La cantidad cada vez mayor de usuarios y de la información disponible en la web impone retos de rendimiento a éstos. El procesamiento de consultas consume una cantidad significativa de recursos computacionales de modo que se requieren muchas técnicas de optimización para permitir que los motores de búsqueda hagan frente de manera eficiente a un alto tráfico de consultas.

Uno de los mecanismos más importantes para hacer frente a este tipo de problemas es el almacenamiento en caché (caching), que se puede aplicar a diferentes niveles de la arquitectura de un WSE. El estudio de cachés de Resultados y Listas en el contexto de los motores de búsqueda se ha convertido en un tema de investigación popular en la última década. Sin embargo, el caché de Intersecciones no ha recibido mucha atención. Esta tesis se centra en este problema y propone el uso de políticas de almacenamiento en caché que consideran el costo de los ítems (cost-aware) y que pueden contribuir a la mejora del rendimiento de los motores de búsqueda.

En primer lugar, se proponen diferentes estrategias de resolución de consultas que tienen en cuenta la existencia de un caché de Intersecciones. A continuación, se presenta un estudio de este caché de Intersecciones usando cuatro estrategias de resolución de consultas y diferentes combinaciones de políticas de reemplazo. Básicamente, se consideran políticas de reemplazo estáticas, dinámicas e híbridas adaptadas a este problema y se las compara contra políticas que no consideran el costo (cost-oblivious) usadas como referencia. Además, se brinda una evaluación considerando dos escenarios: (a) cuando el índice invertido reside en el disco y (b) cuando todo el índice reside en la memoria principal.

Los resultados generales muestran que la estrategia S4 es la mejor, y esto es independiente de la política de reemplazo utilizada. Cabe recordar que S4 analiza primero todas las posibles combinaciones de dos términos en caché con el fin de maximizar las posibilidades de obtener un acierto. Considerando el índice residente en disco, S4 supera en

hasta un 61% a las políticas restantes (en promedio) y la mejora del rendimiento alcanza un 24% para el caso del índice en memoria. De la misma manera, se puede observar que las políticas que consideran el costo son mejores que las que no lo hacen, mejorando así el estado del arte en el tema (39%, 24% y 10% mejor para políticas estáticas, dinámicas e híbridas, respectivamente).

Como otro aporte de esta tesis, se propone un caché Integrado para mejorar el rendimiento de un sistema de búsqueda usando la memoria de manera más eficiente. Este caché almacena las listas de pares de términos sobre la base de una estructura de datos existente y se combina con una estrategia de resolución que aprovecha la existencia del caché de Intersecciones. También, se representan los datos en caché tanto en forma no comprimida como comprimida y se consideran varias heurísticas para poblar la memoria caché. La evaluación muestra que el método propuesto supera a un caché de Listas estándar en la mayoría de los casos: hasta un 38% y un 53% para datos no comprimidos y comprimidos, respectivamente. Este enfoque se aprovecha no sólo de la existencia del caché de Listas+Intersecciones sino también de la estrategia de resolución de consulta, reduciendo el tiempo todal de procesamiento.

Por último, se propone y evalúa una política de admisión para un caché de Intersecciones sobre la base de un enfoque basado en Machine Learning. Se modela la decisión de admisión como un problema de clasificación, cuyo principal objetivo es evitar que ciertos pares de términos poco frecuentes contaminen el caché. También, se integra la política de admisión en la estrategia de resolución de consultas y se incorpora un algoritmo de control de frecuencia acumulada diseñado específicamente para manejar los errores de clasificación, el cual mejora el rendimiento del caché. La evaluación utilizando diferentes umbrales de frecuencia muestra que la política de admisión propuesta aumenta el rendimiento del caché de Intersecciones, alcanzando una mejora del costo total de procesamiento cercana al 12% con respecto al mismo algoritmo sin el control de admisión al caché.

A partir de este trabajo surgen diferentes líneas de investigación. Primero, se tiene la intención de diseñar y evaluar estructuras de datos específicas para un caché de Intersecciones que considera el costo. Las combinaciones óptimas de términos a combinar y mantener en caché parecen ser todavía un problema abierto, especialmente para los enfoques que consideran el costo. También, resulta interesante evaluar el uso de otras técnicas como la poda y compresión de listas debido a que en el escenario donde el índice invertido completo se encuentra en memoria, el caché de Intersecciones se convierte en el segundo nivel en la arquitectura del motor de búsqueda.

En el caso de la caché Integrada, es interesante ampliar el enfoque considerando trigramas y otras combinaciones de términos. Además, la evaluación de diferentes codificadores

de compresión de acuerdo con la distribución de los DocID de la representación integrada es un problema a estudiar en el futuro. Es interesante también modelar este problema analizando el compromiso espacio-tiempo. Otra cuestión abierta interesante es el diseño e implementación de una versión dinámica de este caché. En este caso, las políticas de admisión y de desalojo deben contemplar no solamente propiedades de los términos, sino también de los pares.

Por último, diferentes mejoras a la política de admisión son claramente posibles. Se tiene la intención de utilizar la retroalimentación de los algoritmos iniciales evaluados para mejorar el rendimiento del proceso de clasificación y considerar nuevas estrategias para detectar pares poco frecuentes con mayor precisión. Otra dirección interesante es investigar la dinámica de los pares de términos (poco frecuente/frecuente) para establecer intervalos correctos para re-entrenar el modelo de predicción en tiempo de ejecución. También cuenta definir una familia de algoritmos de control de frecuencia acumulada que se adapten mejor a los diferentes flujos de consultas mediante el uso de diversas estrategias de cola (en lugar de usar sólo FIFO).

Bibliography

[1] Charu Aggarwal. *Data Classification: Algorithms and Applications*. Chapman and Hall/CRC Press, 2014. ISBN 9781466586741.

[2] Vo Ngoc Anh and Alistair Moffat. Inverted index compression using word-aligned binary codes. *Inf. Retr.*, 8(1):151–166, January 2005. ISSN 1386-4564.

[3] Vo Ngoc Anh and Alistair Moffat. Pruned query evaluation using pre-computed impacts. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '06, pages 372–379, New York, NY, USA, 2006. ACM. ISBN 1-59593-369-7.

[4] Ioannis Arapakis, Xiao Bai, and B. Barla Cambazoglu. Impact of response latency on user behavior in web search. In *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*, SIGIR '14, pages 103–112, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2257-7.

[5] Diego Arroyuelo, Senén González, Mauricio Oyarzún, and Victor Sepulveda. Document identifier reassignment and run-length-compressed inverted indexes for improved search performance. In *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '13, pages 173–182, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2034-4.

[6] Claudine Badue, Ramurti Barbosa, Paulo Golgher, Berthier Ribeiro-Neto, and Nivio Ziviani. Basic issues on the processing of web queries. In *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '05, pages 577–578, New York, NY, USA, 2005. ACM. ISBN 1-59593-034-5.

[7] Claudine Badue, Ricardo Baeza-Yates, Berthier Ribeiro-Neto, Artur Ziviani, and Nivio Ziviani. Analyzing imbalance among homogeneous index servers in a web search system. *Inf. Process. Manage.*, 43(3):592–608, May 2007. ISSN 0306-4573.

[8] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval: The Concepts and Technology behind Search*. Addison-Wesley Professional, Inc., 2nd edition, 2011. ISBN 0321416910.

[9] Ricardo Baeza-Yates, Liliana Calderón-Benavides, and Cristina González-Caro. The intention behind web queries. In *Proceedings of the 13th International Conference on String Processing and Information Retrieval*, SPIRE'06, pages 98–109, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-45774-7, 978-3-540-45774-9.

[10] Ricardo Baeza-Yates, Aristides Gionis, Flavio Junqueira, Vanessa Murdock, Vassilis Plachouras, and Fabrizio Silvestri. The impact of caching on search engines. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '07, pages 183–190, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-597-7.

[11] Ricardo Baeza-Yates, Flavio Junqueira, Vassilis Plachouras, and Hans Friedrich Witschel. Admission policies for caches of search engine results. In *Proceedings of the 14th International Conference on String Processing and Information Retrieval*, SPIRE'07, pages 74–85, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-75529-2, 978-3-540-75529-6.

[12] Ricardo Baeza-Yates, Aristides Gionis, Flavio P. Junqueira, Vanessa Murdock, Vassilis Plachouras, and Fabrizio Silvestri. Design trade-offs for search engine caching. *ACM Trans. Web*, 2(4):20:1–20:28, October 2008. ISSN 1559-1131.

[13] Sorav Bansal and Dharmendra S. Modha. Car: Clock with adaptive replacement. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST '04, pages 187–200, Berkeley, CA, USA, 2004. USENIX Association.

[14] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, March 2003. ISSN 0272-1732.

[15] László Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.*, 5(2):78–101, June 1966. ISSN 0018-8670.

[16] Azer Bestavros, Robert Carter, Mark Crovella, Carlos Cunha, Abdelsalam Heddaya, and Sulaiman Mirdad. Application-level document caching in the internet. Technical report, Boston University, Boston, MA, USA, 1995.

[17] Bodo Billerbeck, Adam Cannane, Abhijit Chattaraj, Nicholas Lester, William Webber, Hugh E. Williams, John Yiannis, and Justin Zobel. Rmit university at trec 2004. In *Proceedings Text Retrieval Conference (TREC), Gaithersburg, MD, November 2004. National Institute of Standards and Technology Special Publication*, 2004.

[18] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970. ISSN 0001-0782.

[19] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001. ISSN 0885-6125.

[20] Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the Twelfth International Conference on Information and Knowledge Management*, CIKM '03, pages 426–434, New York, NY, USA, 2003. ACM. ISBN 1-58113-723-0.

[21] B. Barla Cambazoglu, Aytul Catal, and Cevdet Aykanat. Effect of inverted index partitioning schemes on performance of query processing in parallel text retrieval systems. In *Proceedings of the 21st International Conference on Computer and Information Sciences*, ISCIS'06, pages 717–725, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-47242-8, 978-3-540-47242-1.

[22] 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 *Proc. of the third ACM Int. Conf. on Web search and data mining*, WSDM '10, pages 411–420, USA, 2010. ISBN 978-1-60558-889-6.

[23] Pei Cao and Sandy Irani. Cost-aware www proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*, USITS'97, pages 18–18, Berkeley, CA, USA, 1997. USENIX Association.

[24] Brian Carpenter (Ed). Architectural principles of the internet. RFC 1958 (Informational), June 1996.

[25] Carlos Castillo. Effective web crawling. *SIGIR Forum*, 39(1):55–56, June 2005. ISSN 0163-5840.

[26] Carlos Castillo, Debora Donato, Luca Becchetti, Paolo Boldi, Stefano Leonardi, Massimo Santini, and Sebastiano Vigna. A reference collection for web spam. *SIGIR Forum*, 40(2):11–24, December 2006. ISSN 0163-5840.

[27] Matteo Catena and Nicola Tonellotto. A study on query energy consumption in web search engines. In *Proceedings of the 6th Italian Information Retrieval Workshop, Cagliari, Italy, May 25-26, 2015.*, 2015.

[28] Matteo Catena, Craig Macdonald, and Iadh Ounis. On inverted index compression for search engine efficiency. In Maarten de Rijke, Tom Kenter, ArjenP. de Vries,

ChengXiang Zhai, Franciska de Jong, Kira Radinsky, and Katja Hofmann, editors, *Advances in Information Retrieval*, volume 8416 of *Lecture Notes in Computer Science*, pages 359–371. Springer International Publishing, 2014. ISBN 978-3-319-06027-9.

[29] Surajit Chaudhuri, Kenneth Church, Arnd Christian König, and Liying Sui. Heavy-tailed distributions and multi-keyword queries. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '07, pages 663–670, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-597-7.

[30] J. Shane Culpepper and Alistair Moffat. Compact set representation for information retrieval. In *Proc. of the 14th International Conf. on String Processing and Information Retrieval*, SPIRE'07, pages 137–148, Berlin, Heidelberg, 2007. ISBN 3-540-75529-2, 978-3-540-75529-6.

[31] Edleno S. de Moura, Célia F. dos Santos, Daniel R. Fernandes, Altigran S. Silva, Pavel Calado, and Mario A. Nascimento. Improving web search efficiency via a locality based static pruning method. In *Proceedings of the 14th International Conference on World Wide Web*, WWW '05, pages 235–244, New York, NY, USA, 2005. ACM. ISBN 1-59593-046-9.

[32] Jeffrey Dean. Challenges in building large-scale information retrieval systems: Invited talk. In *Proceedings of the Second ACM International Conference on Web Search and Data Mining*, WSDM '09, pages 1–1, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-390-7.

[33] Peter J. Denning. The locality principle. *Commun. ACM*, 48(7):19–24, July 2005. ISSN 0001-0782.

[34] Shuai Ding and Torsten Suel. Faster top-k document retrieval using block-max indexes. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '11, pages 993–1002, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0757-4.

[35] Shuai Ding, Josh Attenberg, Ricardo Baeza-Yates, and Torsten Suel. Batch query processing for web search engines. In *Proc. of the Fourth ACM International Conf. on Web Search and Data Mining*, WSDM '11, pages 137–146, New York, NY, USA, 2011. ISBN 978-1-4503-0493-1.

[36] Wolfgang Effelsberg and Theo Haerder. Principles of database buffer management. *ACM Trans. Database Syst.*, 9(4):560–595, December 1984. ISSN 0362-5915.

[37] P. Elias. Universal codeword sets and representations of the integers. *IEEE Trans. Inf. Theor.*, 21(2):194–203, September 2006. ISSN 0018-9448.

[38] Tiziano Fagni, Raffaele Perego, Fabrizio Silvestri, and Salvatore Orlando. Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Trans. Inf. Syst.*, 24(1):51–78, January 2006. ISSN 1046-8188.

[39] Esteban Feuerstein. *LATIN '95: Theoretical Informatics: Second Latin American Symposium Valparaíso, Chile, April 3–7, 1995 Proceedings*, chapter Paging more than one page, pages 272–285. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995. ISBN 978-3-540-49220-7.

[40] Esteban Feuerstein. *On-line Paging of Structured Data and Multi-Threaded Paging.* PhD thesis, Universitá degli Studi di Roma 'La Sapienza', 1995.

[41] Esteban Feuerstein and Gabriel Tolosa. Cost-aware intersection caching and processing strategies for in-memory inverted indexes. In *In Proc. of 11th Workshop on Large-scale and Distributed Systems for Information Retrieval*, LSDS-IR'14, New York, 2014.

[42] Esteban Feuerstein, Mauricio Marin, Michel Mizrahi, Veronica Gil-Costa, and Ricardo Baeza-Yates. Two-dimensional distributed inverted files. In *Proceedings of the 16th International Symposium on String Processing and Information Retrieval*, SPIRE '09, pages 206–213, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03783-2.

[43] Esteban Feuerstein, Veronica Gil-Costa, Michel Mizrahi, and Mauricio Marin. Performance evaluation of improved web search algorithms. In *Proc. of the 9th Int. Conf. on High performance computing for computational science*, VECPAR'10, pages 236–250, Berlin, Heidelberg, 2011. ISBN 978-3-642-19327-9.

[44] Esteban Feuerstein, Veronica Gil-Costa, Mauricio Marin, Gabriel Tolosa, and Ricardo Baeza-Yates. 3d inverted index with cache sharing for web search engines. In *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, pages 272–284, Berlin, Heidelberg, 2012. ISBN 978-3-642-32819-0.

[45] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and et al. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *IN AOFA '07: Proceedings of the 2007 International Conference on Analysis of Algorithms*, 2007.

[46] Zvi Galil. Efficient algorithms for finding maximum matching in graphs. *ACM Comput. Surv.*, 18(1):23–38, March 1986. ISSN 0360-0300.

[47] Qingqing Gan and Torsten Suel. Improved techniques for result caching in web search engines. In *Proceedings of the 18th international conference on World wide web*, WWW '09, pages 431–440, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-487-4.

[48] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. Compressing relations and indexes. In *Proceedings of the Fourteenth International Conference on Data Engineering*, ICDE '98, pages 370–379, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8289-2.

[49] S.W. Golomb. Run-length encodings. In *IEEE Trans. Info. Theory*, volume 12, 1966.

[50] A. Gulli and A. Signorini. The indexable web is more than 11.5 billion pages. In *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web*, WWW '05, pages 902–903, New York, NY, USA, 2005. ACM. ISBN 1-59593-051-5.

[51] Jun Hirai, Sriram Raghavan, Hector Garcia-Molina, and Andreas Paepcke. Webbase: A repository of web pages. In *Proc. of the 9th International World Wide Web Conf. on Computer Networks*. North-Holland Publishing Co., 2000.

[52] Arun Iyengar, Lakshmish Ramaswamy, and Bianca Schroeder. Techniques for efficiently serving and caching dynamic web content. In *Web content delivery*, pages 101–130. Springer, 2005.

[53] Bernard J. Jansen, Danielle L. Booth, and Amanda Spink. Determining the informational, navigational, and transactional intent of web queries. *Inf. Process. Manage.*, 44(3):1251–1266, May 2008. ISSN 0306-4573.

[54] Myeongjae Jeon, Saehoon Kim, Seung-won Hwang, Yuxiong He, Sameh Elnikety, Alan L. Cox, and Scott Rixner. Predictive parallelization: Taming tail latencies in web search. In *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*, SIGIR '14, pages 253–262, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2257-7.

[55] Theodore Johnson and Dennis Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc. ISBN 1-55860-153-8.

[56] Simon Jonassen. Efficient query processing in distributed search engines. *SIGIR Forum*, 47(1):60–61, 2013.

[57] Simon Jonassen, B. Barla Cambazoglu, and Fabrizio Silvestri. Prefetching query results and its impact on search engines. In *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '12, pages 631–640, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1472-5.

[58] Anna R. Karlin, Mark S. Manasse, Larry Rudolph, and Daniel D. Sleator. Competitive snoopy caching. *Algorithmica*, 3(1):79–119, 1988. ISSN 1432-0541.

[59] Rachid El Abdouni Khayari, Michael Best, and Axel Lehmann. Impact of document types on the performance of caching algorithms in www proxies: A trace driven simulation study. In *Proceedings of the 19th International Conference on Advanced Information Networking and Applications - Volume 1*, AINA '05, pages 737–742, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2249-1.

[60] Ron Kohavi and Foster Provost. Glossary of terms. *Machine Learning*, 30(2-3): 271–274, 1998.

[61] Roberto Konow, Gonzalo Navarro, Charles L.A. Clarke, and Alejandro López-Ortíz. Faster and smaller inverted indices with treaps. In *Proc. of the 36th Int. Conf. on Research and Development in Information Retrieval*, SIGIR '13, pages 193–202, New York, NY, USA, 2013. ISBN 978-1-4503-2034-4.

[62] Tayfun Kucukyilmaz, Ata Turk, and Cevdet Aykanat. *Memory Resident Parallel Inverted Index Construction*, pages 99–105. Springer London, London, 2012. ISBN 978-1-4471-2155-8.

[63] Ravi Kumar, Kunal Punera, Torsten Suel, and Sergei Vassilvitskii. Top-k aggregation using intersections of ranked inputs. In *Proceedings of the Second ACM International Conference on Web Search and Data Mining*, WSDM '09, pages 222–231, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-390-7.

[64] Hoang Thanh Lam, Raffaele Perego, Nguyen Thoi Quan, and Fabrizio Silvestri. Entry pairing in inverted file. In *Proc. of the 10th International Conf. on Web Information Systems Engineering*, WISE '09, pages 511–522, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-04408-3.

[65] Ronny Lempel and Shlomo Moran. Predictive caching and prefetching of query results in search engines. In *Proceedings of the 12th international conference on World Wide Web*, WWW '03, pages 19–28, New York, NY, USA, 2003. ACM. ISBN 1-58113-680-3.

[66] H Levene. Contributions to probability and statistics: Essays in honor of harold hotelling. pages 278–292. Stanford University Press, 1960.

[67] Ruixuan Li, Chengzhou Li, Weijun Xiao, Hai Jin, Heng He, Xiwu Gu, Kunmei Wen, and Zhiyong Xu. An efficient ssd-based hybrid storage architecture for large-scale search engines. In *Proceedings of the 2012 41st International Conference on Parallel Processing*, ICPP '12, pages 450–459, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4796-1.

[68] Jimmy Lin and Andrew Trotman. Anytime ranking for impact-ordered indexes. In *Proceedings of the 2015 International Conference on The Theory of Information Retrieval*, ICTIR '15, pages 301–304, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3833-2.

[69] Tie-Yan Liu. Learning to rank for information retrieval. *Found. Trends Inf. Retr.*, 3(3):225–331, March 2009. ISSN 1554-0669.

[70] Xiaohui Long and Torsten Suel. Three-level caching for efficient query processing in large web search engines. In *Proceedings of the 14th international conference on World Wide Web*, WWW '05, pages 257–266, New York, NY, USA, 2005. ACM. ISBN 1-59593-046-9.

[71] Qiong Luo, Jeffrey F. Naughton, and Wenwei Xue. Form-based proxy caching for database-backed web sites: Keywords and functions. *The VLDB Journal*, 17(3): 489–513, May 2008. ISSN 1066-8888.

[72] William L. Lynch, Brian K. Bray, and M. J. Flynn. The effect of page allocation on caches. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, MICRO 25, pages 222–225, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press. ISBN 0-8186-3175-9.

[73] Craig Macdonald, Iadh Ounis, and Nicola Tonellotto. Upper-bound approximations for dynamic pruning. *ACM Trans. Inf. Syst.*, 29(4):17:1–17:28, December 2011. ISSN 1046-8188.

[74] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008. ISBN 0521865719, 9780521865715.

[75] Mauricio Marin, Carlos Gomez-Pantoja, Senen Gonzalez, and Veronica Gil-Costa. Scheduling intersection queries in term partitioned inverted files. In *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, Euro-Par '08, pages 434–443, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-85450-0.

[76] Mauricio Marin, Veronica Gil-Costa, and Carlos Gomez-Pantoja. New caching techniques for web search engines. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 215–226, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-942-8.

[77] E.P Markatos. On caching search engine query results. *Comput. Commun.*, 24(2): 137–143, February 2001. ISSN 0140-3664.

[78] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Syst. J.*, 9(2):78–117, June 1970. ISSN 0018-8670.

[79] Nimrod Megiddo and Dharmendra S. Modha. Arc: A self-tuning, low overhead replacement cache. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, FAST'03, pages 115–130, Berkeley, CA, USA, 2003. USENIX Association.

[80] Sergey Melink, Sriram Raghavan, Beverly Yang, and Hector Garcia-Molina. Building a distributed full-text index for the web. *ACM Trans. Inf. Syst.*, 19(3):217–241, July 2001. ISSN 1046-8188.

[81] Christian Middleton and Ricardo Baeza-Yates. A comparison of open source search engines, 2005.

[82] Alistair Moffat, William Webber, and Justin Zobel. Load balancing for term-distributed parallel retrieval. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '06, pages 348–355, New York, NY, USA, 2006. ACM. ISBN 1-59593-369-7.

[83] Alexandros Ntoulas and Junghoo Cho. Pruning policies for two-tiered inverted index with correctness guarantee. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '07, pages 191–198, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-597-7.

[84] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. An optimality proof of the lru-k page replacement algorithm. *J. ACM*, 46(1):92–112, January 1999. ISSN 0004-5411.

[85] Giuseppe Ottaviano and Rossano Venturini. Partitioned elias-fano indexes. In *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*, SIGIR '14, pages 273–282, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2257-7.

[86] Giuseppe Ottaviano, Nicola Tonellotto, and Rossano Venturini. Optimal space-time tradeoffs for inverted indexes. In *Proceedings of the Eighth ACM International*

Conference on Web Search and Data Mining, WSDM '15, pages 47–56, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3317-7.

[87] Rifat Ozcan, Ismail Sengor Altingovde, and Özgür Ulusoy. Static query result caching revisited. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 1169–1170, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-085-2.

[88] Rifat Ozcan, Ismail Sengor Altingovde, and Özgür Ulusoy. Cost-aware strategies for query result caching in web search engines. *ACM Trans. Web*, 5(2):9:1–9:25, May 2011. ISSN 1559-1131.

[89] Rifat Ozcan, I. Sengor Altingovde, B. Barla Cambazoglu, Flavio P. Junqueira, and ÖZgüR Ulusoy. A five-level static cache architecture for web search engines. *Inf. Process. Manage.*, 48(5):828–840, September 2012. ISSN 0306-4573.

[90] Rifat Ozcan, Ismail Sengor Altingovde, B. Barla Cambazoglu, and Özgür Ulusoy. Second chance: A hybrid approach for dynamic result caching and prefetching in search engines. *ACM Trans. Web*, 8(1):3:1–3:22, December 2013. ISSN 1559-1131.

[91] Greg Pass, Abdur Chowdhury, and Cayley Torgeson. A picture of search. In *Proceedings of the 1st international conference on Scalable information systems*, InfoScale '06, New York, NY, USA, 2006. ACM. ISBN 1-59593-428-6.

[92] Stefan Podlipnig and Laszlo Böszörmenyi. A survey of web cache replacement strategies. *ACM Comput. Surv.*, 35(4):374–398, December 2003. ISSN 0360-0300.

[93] Baeza-Yates Ricardo, Castillo Carlos, Junqueira Flavio, Plachouras Vassilis, and Silvestri Fabrizio Silvestri. Challenges on distributed web retrieval. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 6–20, 2007.

[94] Knut Magne Risvik, Yngve Aasheim, and Mathias Lidal. Multi-tier architecture for web search engines. In *Proceedings of the First Conference on Latin American Web Congress*, LA-WEB '03, pages 132–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2058-8.

[95] Cristian Rossi, Edleno S. de Moura, Andre L. Carvalho, and Altigran S. da Silva. Fast document-at-a-time query processing using two-tier indexes. In *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '13, pages 183–192, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2034-4.

[96] Paricia Correia Saraiva, Edleno Silva de Moura, Novio Ziviani, Wagner Meira, Rodrigo Fonseca, and Berthier Riberio-Neto. Rank-preserving two-level caching for scalable search engines. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '01, pages 51–58, New York, NY, USA, 2001. ACM. ISBN 1-58113-331-6.

[97] Fethi Burak Sazoglu, Berkant Barla Cambazoglu, Rifat Ozcan, Ismail Sengör Altingövde, and Özgür Ulusoy. A financial cost metric for result caching. In *The 36th International ACM SIGIR conference on research and development in Information Retrieval, SIGIR '13, Dublin, Ireland - July 28 - August 01, 2013*, pages 873–876, 2013.

[98] E. Shurman and J Brutlag. Performance related changes and their user impacts. In *Velocity '09: Web Performance and Operations Conference*, 2009.

[99] Fabrizio Silvestri. Mining query logs: Turning search usage data into knowledge. *Found. Trends Inf. Retr.*, 4(1-2):1–174, January 2010. ISSN 1554-0669.

[100] Gleb Skobeltsyn, Flavio Junqueira, Vassilis Plachouras, and Ricardo Baeza-Yates. Resin: a combination of results caching and index pruning for high-performance web search engines. In *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '08, pages 131–138, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-164-4.

[101] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, February 1985. ISSN 0001-0782.

[102] A.j. Smith. Cache memories, 14. In *Computing Surveys*, 1982.

[103] Marina Sokolova and Guy Lapalme. A systematic analysis of performance measures for classification tasks. *Inf. Process. Manage.*, 45(4):427–437, July 2009. ISSN 0306-4573.

[104] Shirish Tatikonda, B. Barla Cambazoglu, and Flavio P. Junqueira. Posting list intersection on multicore architectures. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, SIGIR '11, pages 963–972, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0757-4.

[105] Anthony Tomasic and Hector Garcia-Molina. Caching and database scaling in distributed shared-nothing information retrieval systems. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 129–138, New York, NY, USA, 1993. ACM. ISBN 0-89791-592-5.

[106] Jiancong Tong, Gang Wang, and Xiaoguang Liu. Latency-aware strategy for static list caching in flash-based web search engines. In *Proceedings of the 22Nd ACM International Conference on Conference on Information & Knowledge Management*, CIKM '13, pages 1209–1212, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2263-8.

[107] Howard Turtle and James Flood. Query evaluation: Strategies and optimizations. *Information Processing and Management*, 31(6):831–850, November 1995. ISSN 0306-4573.

[108] W. Webber and A. Moffat. In search of reliable retrieval experiments. In *ADCS 2005, Proceedings of the Tenth Australasian Document Computing Symposium, December 12, 2005*, pages 26–33, 2005.

[109] Hugh E. Williams and Justin Zobel. Compressing integers for fast file access. *The Computer Journal*, 42:193–201, 1999.

[110] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing gigabytes (2nd ed.): compressing and indexing documents and images*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999. ISBN 1-55860-570-3.

[111] Xindong Wu, Vipin Kumar, J. Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J. McLachlan, Angus Ng, Bing Liu, Philip S. Yu, Zhi-Hua Zhou, Michael Steinbach, David J. Hand, and Dan Steinberg. Top 10 algorithms in data mining. *Knowl. Inf. Syst.*, 14(1):1–37, December 2007. ISSN 0219-1377.

[112] Hao Yan, Shuai Ding, and Torsten Suel. Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 401–410, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-487-4.

[113] Neal Young. *Competitive Paging and Dual-guided On-line Weighted Caching and Watching Algorithms*. PhD thesis, Princeton University, Princeton, NJ, USA, 1991. UMI Order No. GAX92-02525.

[114] Neal E. Young. On-line file caching. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '98, pages 82–86, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics. ISBN 0-89871-410-9.

[115] Jiangong Zhang, Xiaohui Long, and Torsten Suel. Performance of compressed inverted list caching in search engines. In *Proceedings of the 17th international conference on World Wide Web*, WWW '08, pages 387–396, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-085-2.

[116] Justin Zobel and Alistair Moffat. Adding compression to a full-text retrieval system. *Softw. Pract. Exper.*, 25(8):891–903, August 1995. ISSN 0038-0644.

[117] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), July 2006. ISSN 0360-0300.

[118] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-scalar ram-cpu cache compression. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, page 59, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2570-9.