An Adaptive Index Recommendation System (AIRs) on Document-Based Databases

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation
von

Parinaz Ameri
aus Bam, Iran

Tag der mündlichen Prüfung: 20.07.2017

Erster Gutachter:	Prof. Dr. Achim Streit
Zweiter Gutachter:	Prof. Dr. Andreas Oberweis

Erklärung zur selbständigen Anfertigung der Dissertationsschrift

Hiermit erkläre ich, dass ich die Dissertationsschrift mit dem Titel

An Adaptive Index Recommendation System (AIRs) on Document-Based Databases

selbständig angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Regeln zur Sicherung guter wissenschaftlicher Praxis am Karlsruher Institut für Technologie (KIT) beachtet habe.

Ort, Datum Parinaz Ameri

To Hamzeh, my true friend
and
to my beloved parents

Acknowledgements

I would like to begin my acknowledgements by thanking Prof. Dr. Achim Streit, my supervisor, for the continuous support of my PhD studies and research. I would also like to thank Prof. Dr. Andreas Oberweis for co-supervising this thesis.

Besides, I would like to thank Dr. Jörg Meyer for his advice throughout my work and for his comprehensive effort in proofreading the drafts of this thesis. Additionally, I would like to thank Mr. Nico Schlitter for his friendly encouragement and support during my doctoral study.

I would like to thank my parents for their unconditional and long lasting love. Finally, I would like to thank my true friend, my husband, for his devoted attention, kindness, and compassion he has shown during the past years it has taken me to finalize this thesis.

Zusammenfassung

Aufgrund stetig wachsender Datenmengen und einer gesteigerten Vielfalt an Datenbankanwendungen zeigen aktuelle Studien ein steigendes Interesse an Assistenzsystemen, welche beim Design von Datenbanken und insbesondere bei der Auswahl von Indexmengen unterstützen. Ziel dieser Arbeit ist es, eine tiefgreifende Analyse von individuellen Indexmengen zu ermöglichen, um jene Menge zu identifizieren, welche für ein Arbeitsaufkommen den größten Nutzen bei möglichst geringen Wartungskosten aufweist.

Beschränkungen, welche durch die Umwelt und das eigentliche Arbeitsaufkommen auferlegt werden, erschweren eine solche Analyse. Selbst wenn die zugrunde liegenden Speicherrestriktionen es zuließen, wäre das Erzeugen aller möglichen Indexmengen aufgrund des durch Schreiboperationen verursachten Wartungsaufwands jedoch keine Option.

Obwohl das Index Select Problem (ISP) seit Jahrzenten untersucht wird, vereinfachen viele Studien einige Aspekte, indem sie sich lediglich auf ein Lesezugriffsszenario beschränken oder Fortschritte bei der Anfrageoptimierung und der Verschränkung von Indexen ignorieren. Darüber hinaus wurden die Studien größtenteils für relationale Datenbanken durchgeführt.

Diese Arbeit nähert sich den vernachlässigten Aspekten dieses komplexen Optierungsproblems durch die Entwicklung eines mathematischen Modells, welches die relevanten Faktoren berücksichtigt. Das entstandene Modell wird anhand von Dokument-basierten Datenbanken untersucht. Dabei wird klar, dass das Anwachsen des Anfragelastumfangs und der gespeicherten Datenmengen die Entwicklung neuer Methoden erfordert, um die Anzahl der durch den Optimierer zu evaluierenden Indexmengen zu reduzieren ohne dabei jedoch die relevanten Indexmengen zu verlieren.

Der zu betrachtende Suchraum wird verkleinert, indem nur die relevanten Indexe für die am häufigsten vorkommenden und langlaufenden Datenbankanfragen als Index-Kandidaten berücksichtigt werden. Eine ähnliches Vorgehen wird auf Indexkombinationen angewendet. Anstatt alle möglichen Indexkombinationen zu berücksichtigen, werden lediglich jene verwendet, die vom gegebenen Arbeitsaufkommen auch genutzt werden können. Eines der Ziele dieser Arbeit ist die Entwicklung und zugehörige Komplexitätsbetrachtung eines Algorithmus, der diese beschränkten Kombinationen identifiziert und zugleich die Kosten anhand des indexverschränkenden Anfrageoptimierers verifiziert.

Die Verifikation durch den Anfrageoptimierer erhöht die Last auf dem Datenbanksystem zusätzlich. Um diese Last zu reduzieren wird eine virtuelle Umgebung basierend auf repräsentativen Stichproben des ursprünglichen Datensatzes erzeugt. Da jedoch die Größe der Indexe nicht direkt hergeleitet werden kann, wird eine theoretische Abschätzungsmethode mit Bezug zum ursprünglichen Datensatz entwickelt.

Das Fehlen eines geeigneten Benchmark-Werkzeugs, welches anstelle des gesamten Datenbanksystems lediglich das Index Recommendation System untersucht, wird im Kontext nicht-relationaler Datenbanken adressiert. Neben geeigneten Metriken wird ein generischer Arbeitslast-Generator vorgestellt, mit dem die in dieser Arbeit erzeugten synthetische Datensätze und Arbeitslasten erzeugt werden.

Schließlich werden sowohl die individuellen Lösungen als auch das Gesamtsystem hinsichtlich ihrer Leistungsfähigkeit bewertet. Für diese Evaluierung werden genau jene meteorologischen Datensätze und deren zugehörige Arbeitsaufkommen genutzt, welche diese Arbeit ursprünglich motivierten. Die Resultate zeigen die Effizienz der vorgeschlagenen Lösungen.

Abstract

Due to the increase in the amount of data volume and variety of application workloads that databases should handle, recent studies show a raising interest in the utilization of automatic physical database design assisting systems, more specifically index design systems. The aim of this thesis is to enable thorough investigations of individual sets of indexes for any particular workload to find the most profitable set that has the least maintenance cost on the targeted database.

The nature of such an analysis is complicated due to the constraints imposed by the environment and the workload itself. Because of the indexes' maintenance cost for any write query to the database, creating all possible indexes would not be an option, even if the storage limitations allowed this materialization.

Although the Index Selection Problem (ISP) is a well-established field of research for decades, many of the studies simplified some aspects of the problem like restricting the workloads to read-only scenarios or ignoring the advances in query optimizer capabilities to intersect various indexes to execute one query. Additionally, this problem has mostly been studied in the context of relational databases.

This thesis focuses on covering the missing aspects of this complicated optimization problem by formulating a mathematical model considering the relevant factors. The model is then evaluated with to document-based database types. The growth of workloads and datasets requires the development of new methods to reduce the number of index sets that should be evaluated by the optimization model with an insurance of not eliminating the relevant index sets from the evaluation.

The search space is reduced by considering only the relevant indexes of the most frequent queries and the ones with long run-time as the candidate indexes. A similar idea is applied to index combinations. Instead of using all possible candidate index combinations, the index recommendation system only regards combinations that can be utilized by the given workload. One of the aims of this thesis is the development of an algorithm and the analysis of its complexity that extracts these limited combinations and simultaneously verifies their costs by a query optimizer with index intersection capability.

The verification process by the query optimizer enforces additional load on the in-production system. To reduce this load, a virtual environment with the assistant of representative samples of the original targeted datasets is created.

However, since the size of indexes on the original dataset can not be directly extracted from the sample index sizes, a theoretical method is developed to estimated the approximated original size of indexes.

Finally, the challenge of absence of a proper benchmarks for targeting the index recommendation system rather than evaluating the whole database performance, especially compared to non-relation databases, is addressed. Besides the introduction of proper metrics, the development of a generic workload generator solution allows defining various synthetic datasets and workloads that are used through out the whole thesis.

At the end, individual solutions as well as the overall performance of the entire system is evaluated. This evaluations are performed with the help of the real dataset and workload use-cases from the meteorological projects that motivated this thesis. The evaluation results show the effectiveness of the proposed solutions.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Databases are one of the major means to organize and manage data. Although relational databases have been the dominant type of databases, their deficiency in management capabilities for non-traditional applications such as image processing and web applications lead to the development and usage of non-relational databases. Among various types of databases, the document-based type is a widely utilized type.

Despite their difference, both relational and document-based databases take advantage of indexes to provide fast access to the data. Indexes are data structures that can be created using one or more attributes of a dataset. Indexes contain a copy of the data of its corresponding attributes mostly in form of a tree that can be searched efficiently. The existence of proper indexes reduces the search time of read queries. However, the materialization of all of the possible indexes might exhaust the storage limitations. Also, the presence of indexes introduces additional maintenance costs for write operations, because in addition to execute the write operation all of the relevant indexes should also be updated. Therefore, there is a trade off between fast access by creating more indexes and tolerating maintenance cost by updating them. This trade off decision is an acknowledged NP-complete problem in the literature known as *Index Selection Problem* (ISP).

The growth in databases and their workloads results in more interest of atomized systems to help managing the database physical design decision such as choosing the proper set of indexes. Any automated adaptive approach that aims for recommending the most beneficiary set of indexes to minimize the cost of running all of the queries in a workload should solve this problem. It has been proven that the ISP can be solved in practice by considering some heuristics. Despite extensive studies on this problem, there are still many aspects that require more comprehensive research. The problem of recommending indexes has been studied in the context of relational database in literature. The context of this thesis is the management of the physical database design of a meteorological databases by introducing required indexes in an instance of document-based databases. Among all of the contributions, this thesis focuses on studying the index recommendation problem in the context of document-based databases. The main differential data models, query languages and query optimizer behaviours between these database types are studied and discussed.

These studies then empower the reduction of the search space of the index recommendation problem by the extraction of relevant candidate indexes. In the following, the major research questions and the main contributions of this thesis are explained.

1.1 Main Contributions

This thesis necessarily covers a broad-spectrum of topics to enable the design of an adaptive index recommendation system for various workloads. Although the contributions achieved in this thesis can be considered separately, the general focus of all of them is to recommend the most optimal and beneficiary set of indexes efficiently. These contributions are then put together to design an integrated framework known as *Adaptive Index Recommendation system* (AIRs). In essence, this thesis addresses the following research sub-topics:

1.1.1 Reducing the Search Space of the Index Recommendation Problem

The search space of the index recommendation problem grows exponentially by the number of attributes in the targeted dataset. To reduce this search space and extract the relevant indexes for a specific workload, two main heuristics are developed and utilized by AIRs.

One is the development of a *Frequent-Long* algorithm that identifies only the queries that are either issued frequently or their run-time takes longer than a configurable threshold as relevant queries. The extraction of the candidate indexes of these queries is done by a syntax-based analysis of the queries. The syntax-based analysis is developed based on the behavior of the query optimizer and the query language of the targeted database.

The other one is to limit the evaluation of sets of indexes to the so-called *atomic* ones. An atomic set of indexes is defines as a set of indexes that all of them are used to run some query in the workload. For the extraction of these atomic index sets an *Index Benefit Graph* algorithm is utilized. This algorithm provides a top-down approach to determine the set of atomic indexes.

1.1.2 Formulation of a more Profitable Index Sets as an Objective Function

The benefit and maintenance cost of each set of candidate indexes should be evaluated with regard to the queries in a workload that are issued in a specified period of time. It necessitates the formulation of a cost function with regard to all of the relevant criteria to determine the most beneficiary set of indexes.

The developed formula considers the effect write operations and the cardinality of the dataset attributes. Additionally, to ensure that only substantial indexes for the query optimizer are recommended, the core cost evaluation process to feed as input to the objective function is done by the query optimizer.

This objective function can be then maximized by any optimization method to find the index set with highest benefit and lowest maintenance cost for a particular workload. While query optimizer should be make its optimized decision fast, the optimization process of the index recommendation can use longer time frames to make its decision. Thus, more precise optimization methods can be utilized by an index recommendation system which guarantee finding the global optimal index set.

1.1.3 Evaluation of the Index Sets without Overloading the Query Optimizer

The execution of the cost evaluation process by means of the query optimizer enforces a large amount of overload on the in-production system. It requires that the candidate indexes are available to the query optimizer.The materialization of all candidate indexes for a large dataset is a very resource-consuming process. Also, the large number of necessary calls to the query optimizer for the evaluation process can interfere with the performance of the database in response to its applications.

Utilization of a virtual environment that consists of a representative sample of the targeted dataset is presented as a solution to this problem. By materializing indexes on the representative sample set, they are available to the query optimizer for the cost evaluation process of running each query. In this case, the size of each index on the original dataset must be estimated. This size can be estimated from the number of documents containing the attributes of the index and their corresponding value types.

1.1.4 Evaluation of the Performance of the Designed Solution

Despite the extensive studies on the index selection problem, there are only few unified methods to evaluate the results which are not standardized especially for document-based databases. Other approaches are mostly evaluated by a set of standardized benchmarks that are targeted to evaluate the performance of the databases themselves rather than a specialized system like index recommendation.

Also, the standard methods utilized by other approaches are designed for the relational datbases and the SQL-like query languages. The evaluation of the developed solutions, in particular in relation to a document-based database model necessitates the definition of the baseline, evaluation metric and a proper dataset and workload.

1.2 Outline of the Thesis

The rest of this thesis is organized as the following:

In Chapter 2, the management of some meteorological datasets in a document-based database is provided as the motivation for this thesis. These projects establish a use-case for the index recommendation system. Some of the descriptions and studies on the provided data and use-cases of these projects are published in:

- P. Ameri, et al., “On the application and performance of MongoDB for climate satellite data“, In TrustCom (2014), pp. 652-659,

- R. Lutz, P. Ameri, et al., “Management of Meteorological Mass Data with MongoDB“, In EnviroInfo (2014), pp. 549 -556,
- M. Szuba, P. Ameri, et al., “A Distributed System for Storing and Processing Data from Earth-observing Satellites“, In CCGrid (2015), pp. 169-174.

Hence, a brief description of some of their corresponding databases and workloads is also given. As the next step to set up the scene, a brief and general introduction is provided. This section is followed by a more precise discussion on the criteria of index recommendation. Some of the material presented in this section are already published in:

- P. Ameri, “Database Techniques for Big Data“, In "Big Data: Principles and Paradigms" published (2017), Chapter 6.

Chapter 3 is dedicated to discussions of the related work. In the context of each of the research sub-topics, the major influential related works are presented. The possible inspiration of each of these works on the current thesis as well as their shortcomings and the differences between the approaches are discussed. Some parts of these discussions are first published in:

- P. Ameri, “On a Self-Tuning Index Recommendation Approach for Databases“, In ICDE (2016), pp. 201-205.

In Chapter 4, the overall problems of designing a unified index recommendation system are explained. Then, the general modular architecture design of the AIRs is presented and the requirement of having each of the modules and their particular rule are clarified. Also, the necessity of having a virtual environment for the evaluation and the strategic design of this environment in AIRs are discussed. In the rest of this chapter, various essential practical heuristics that are considered to reduce the complexity of the search space are analyzed. As the next step, the model to estimate the size of indexes without creating them on the original dataset is presented. The initial ideas of the architecture design of the index recommendation system are published in:

- P. Ameri, et al., “On a new approach to the index selection problem using mining algorithms“, IEEE BigData (2015), pp. 2801-2810.

Chapter 5 is devoted to the theoretical analyses of various aspects in design of the index recommendation system. At first, the essential factors to formulate an objective function and the mathematical representation of this function in the chosen optimization method is presented. Then, the problem of having large search space of candidate index sets that are the input of the objective function is discussed. As a solution, the developed algorithm to extract the so-called *atomic configuration* is presented and its complexity is analyzed. The initial formulation of the objective function is published in:

- P. Ameri, “Challenges of Index Recommendation for Databases“, In GVDB (2016), pp. 10-14.

Finally, the evaluation of the performance of each developed solution and the overall performance of the AIRs are presented in Chapter 6. At first, the difficulties of benchmarking the index recommendation system specially in relation to a document-based database are discussed. In that regard, the established methods for the evaluation are presented. In addition, the developed workload generator solution is introduced that enables generation of synthetic database and workloads that are used to evaluate various aspects throughout the whole thesis. In the rest of this chapter, the data sources that are used for the various evaluations are presented. The chosen datasets and workloads of individual meteorological projects are introduced in more details. With help of these datasets and workloads and the proposed methodologies, distinct solutions of AIRs for index recommendation problem and its overall performance to recommend the optimal set of indexes are verified. Some of the materials that are used in this chapter, such as the precise presentation of the synthetic workload generator, the necessity to develop such solution and its specific grammar are published in:

- P. Ameri, et al., “NoWog: A Workload Generator for Database Performance Benchmarking“, In DataCom (2016), pp. 666-673.

After discussion and evaluation of the contributions to the index recommendation problem, the thesis is concluded in Chapter 7. In this chapter, first the achievements of the thesis are summarized and then the results are confronted with the goals of the study. At the end an outline for future work is given.

Chapter 2

Setting the Scene

A crucial step in the database design and management is to identify and adjust the proper indexes to the incoming workload of the corresponding applications. As a result of the increasing complexity in both query processors and database applications, the decision-making process to adjust the system with a proper set of indexes gets more intricate. Consequently, there is a high demand for adaptive recommendation systems that can systematically examine the space of alternative solutions and assist with the database tuning process [1].

This chapter, first, presents the applications from some meteorological databases that required to be managed by a document-based database in Section 2.1. The management of the corresponding workload of these databases and the distribution of their queries lead to the recognition of the need for an adaptive index recommendation system. A brief summary of the required background knowledge to go through this thesis is provided in Section 2.2. Additionally, an introduction to the index recommendation topic and its challenges is given in Section 2.3. At last, Section 2.4 introduces the specific challenges of the index recommendation problem in association with document-based databases.

Most of the content written in this chapter is published in [2], [3], [4], [5], [6] and [7].

2.1 Motivation: The Meteorological Application's Database

The motivation for the work presented in this thesis originated from the management of databases for several meteorological applications in the context of Earth and Environment Data Life Cycle Lab [8] of the Large-Scale Data Management and Analysis project [9]. Their datasets consist of large schemaless data with scalar and multi-dimensional array values. The structure of these datasets are described in detail in [2], [3], [4].

The first example comes from a meteorological application with satellite data (sat1) [2] and [4]. Modern remote sensing techniques implemented and launched on satellite platforms like MIPAS [1] [10] on ENVISAT [11] or MLS [2] [12] on EOS-AURA [13] generate enormous

[1]The Michelson Interferometer for Passive Atmospheric Sounding (MIPAS)

[2]The Earth Observing System Microwave Limb Sounder (EOS MLS)

amounts of global atmospheric data comprising rich, long time continuous measurement sets to be maximally exploited by adequate data processing in a finite time-frame. Ideally, these real world measurements are accompanied by atmospheric chemistry or climate models (e.g., ECHAM [1]/MESSy [2] [14]), which simulate the physical processes and scan the relevant parameter fields to achieve a deeper understanding of the atmosphere by comparison with the measurements [15]. Moreover, data from (often many) different satellites have to be correlated and cross-validated to estimate inherent observation and instrument uncertainties. From the original datasets, derived products are generated which themselves are subject to the aforementioned processing procedures and are typically the main objects of further scientific exploration.

As a result of the different nature and origins of the products, the datasets from all these fields and within a single field are disseminated in very different formats. As an additional complexity, even data equally labeled according to labeling conventions (like CF [16] for netCDF [17] datasets) can have differing actual content (e.g., time often has different reference points). So, there is a need to effectively store the data in a common area for efficient access and also harmonize the datasets or even to find a common denominator for the base parameters to be stored, as these different sets deliver very different quantities. This set of base parameters should be generic for the diverse applications required for the data collections.

We use the IMK [3]/IAA [4] MIPAS/ENVISAT scientific dataset as a sample project, together with additional datasets from other satellites (esp. AURA-MLS). The minimum base parameter set where all data is indexed on consists of temporal and spatial coordinates. Figure 2.1 shows only the spatial distribution of data recorded both by MLS and MIPAS instruments for the duration of one day.

Traditionally analyses of climate data are based on large datasets at the edge of technological and financial feasibility, typically stored in file hierarchies. There are some metadata indexes added in a single server database for each file for crude searching capabilities. On the application side, particular file input modules and specialized visualization software in high-level programming languages (e.g., Java or IDL) for a posteriori plot analysis were prevalent [18]. For the sheer variety of the applications, a lot of detailed knowledge and programming expertise was needed to change or extend such software often composed of many modules in different programming paradigms and languages.

However, recent developments in information technology and rapid price decline make it now possible to index the complete datasets in large, distributed databases and exhaust the full operational capacities of such configurations. Computing power not only in several CPU cores, but even more in the cores of graphic cards in the teraflop range is now ubiquitous so that a lot of pre-processing, conversion, and additional storage (that previously should

[1]ECMWF Hamburg: atmospheric general circulation model by Max Planck Institute for Meteorology in Hamburg, Germany

[2]The new Modular Earth Submodel System (MESSy)

[3]5Karlsruhe Institute of Technology (KIT), Institute for Meteorology and Climate Research (IMK), Germany

[4]6CSIC, Instituto de Astrofsica de Andaluca (IAA), Spain

be done in advance) can now be integrated on the fly in the viewing pipeline of climate analysis. It radically changes the approach to scientific exploration, as real-time viewing of several months datasets and even complex online animation is now within reach.

When it comes to choosing platforms and methods to handle Big Data, several options exists; some of them are: Cloud computing platforms [19], MapReduce [20] and scalable storage systems or massively parallel processing databases [21]. To handle metadata, database with their indexing facilities are known to be more useful than filesystem storages [22]. Additionally, modern databases facilitate the storage of variety of data structures such as non-scalar values and images. The goal is also to store the data itself into the database instead of files.

Therefore, for a design architecture to handle Big Data, scalability of the system must be considered, so that the system can scale over a cluster consisting of several nodes. Based on the CAP (Consistency, Availability, Partition tolerance) theorem [23] traditional relational databases, whose functionality is based on ACID (Atomicity, Consistency, Isolation, Durability) [24] properties, are not partition tolerant. Also, according to the CAP theorem, scalability of traditional databases is a big issue. Moreover, given in our special use case there are more read queries to the database than write queries, and delete queries are considered to happen rarely, being bound by all ACID properties was not only unnecessary but

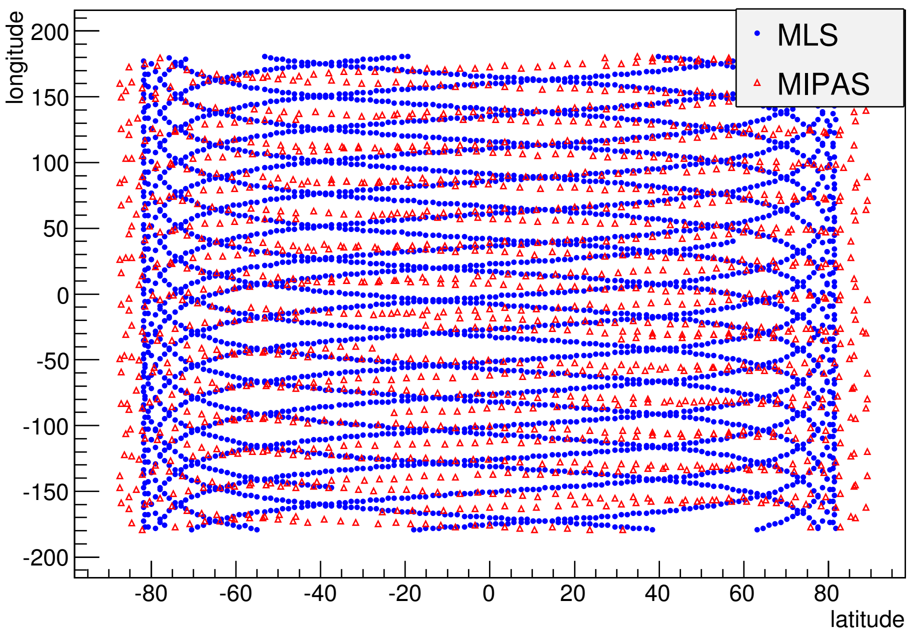


Figure 2.1: Comparison of the spatial distribution of the data measured by MLS and MIPAS instruments during one day.

also incompatible with availability and performance, as well as scalability requirements [25].

The above-mentioned characteristics of the meteorological datasets suggests utilizing a non-relational database. Whereas most of the non-relational databases cover schemaless data structures and facilitate scalability requirements, an instance of a document-based NoSQL database (i.e. MongoDB) is chosen for these applications. Document-based databases assist with handling multi-dimensional array values and nested-document concepts. Additionally, their *JavaScript Object Notation* (JSON) [26]- or *Extensible Markup Language* (XML) [27]-based documents support web applications such as the considered meteorological applications. There are more features provided by this database that is particularly useful for the particular meteorological satellite use case, such as spatial indexing, documents based on JSON which ease the use of GEOJSON [28] match, and also the possibility to have parallel access to the database.

Figure 2.2a depicts the distribution of queries to all of the collections containing data for a *satellite* application [2], [4]. This application does not have a heavy workload. The distribution of queries forms a read-mostly workload for this application. Over the course of seven months, there are only four peaks of insertion to all of the collections of this application and only a small distribution of update operations to one of the collections. (For further description see WKL_1 and WKL_2 in Section 6.2). Therefore, since this is a read-mostly workload, it can benefit a lot from having proper indexes in place.

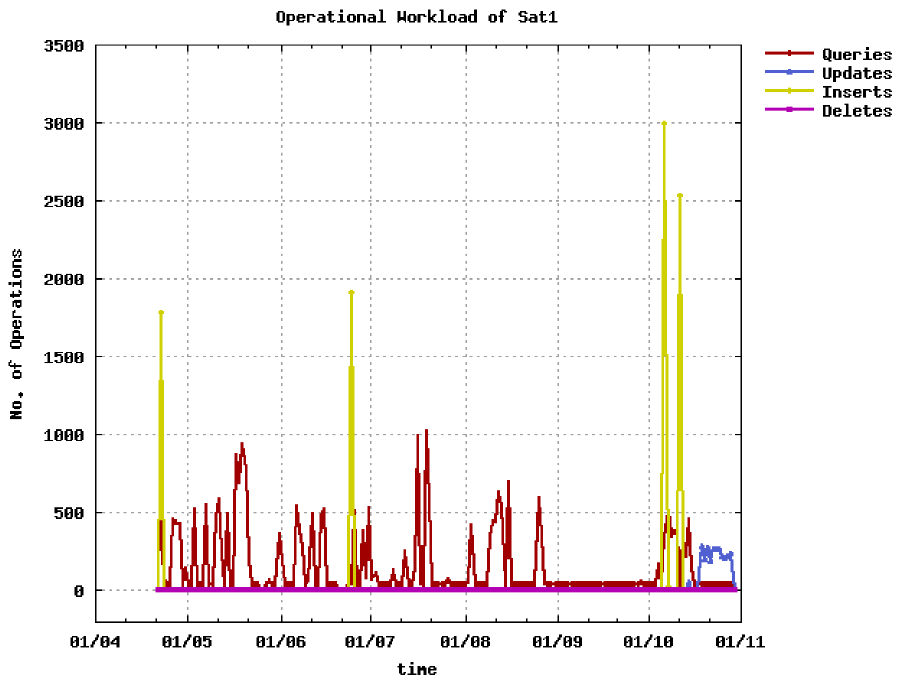
Whereas Figure 2.2a illustrates the frequency of all of the queries, Figure 2.2b exhibits the scattering of unique search queries. Note the unique query that suddenly appears in June of that year - with uniqueQueryID 1 - in a high number (3065 times) and disappears in the following months again. Such queries are usually related to a temporary usage of the data in the application. In many situations, these queries are issued without informing the database system administrator to generate new proper indexes. An automatic index recommendation system, which monitors the incoming workload and accordingly adjusts corresponding indexes to improve the performance of user applications.

Another motivational use-case example is the application of the Gimballed Limb Observer for Radiance Imaging of the Atmosphere (GLORIA) [1]. GLORIA is a recently developed unique atmospheric remote sensing instrument that bridges the gap from scanning to imaging in the infrared spectral domain. This is realized by the combination of a classical Fourier transform spectrometer (FTS) with a 2-D detector array tailored to the FTS needs. GLORIA is designed to operate on the high altitude research platforms aircraft (HALO) [30] and stratospheric balloons [31].

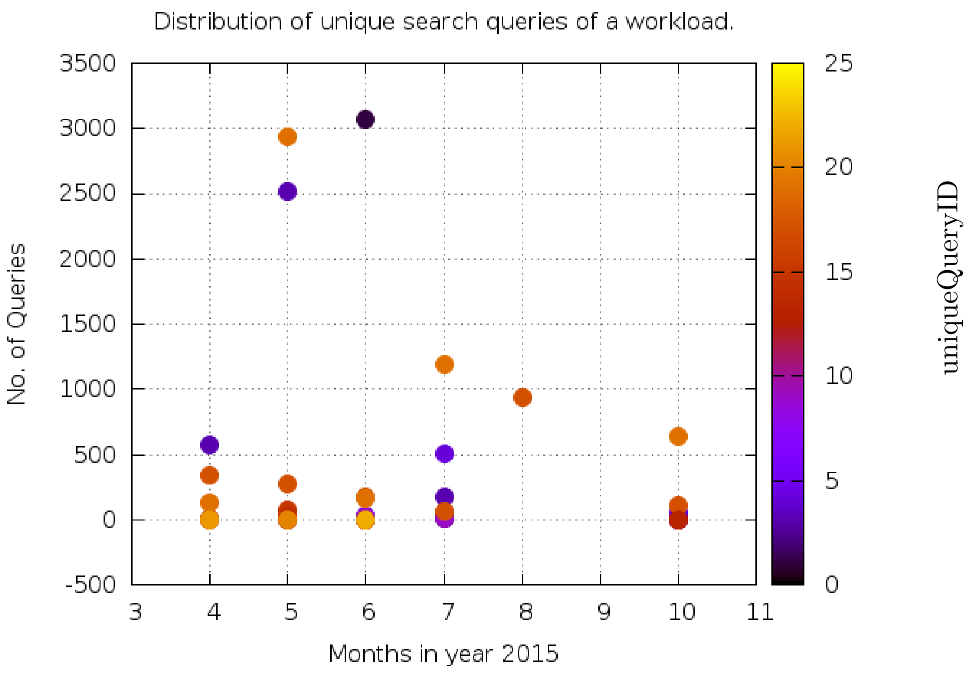
GLORIA's data consist of three major parts: the pre-processed measurement data, the processing configuration data, and the final result data (spectra, trace gas distributions, etc.). The first objective is to store the pre-processed data which is the base for further processing and visualization.

Figure 2.3 shows the data structure of a pre-processed GLORIA dataset provided by

[1]GLORIA is a joint development of the two national research centers of the Helmholtz Association Karlsruhe Institute of Technology (KIT) and Forschungszentrum Jülich (FZJ). Detailed information about the instrument is presented at [29].



(a) Meteorological satellite application workload over the course of more than half a year. This workload is *read* mostly, with some insertion peaks and few frequent *updates* at the end. The has been no *delete* operation in this workload.



(b) Distribution of 22 unique search queries from April to October of 2015 in the satellite workload. Each colored label indicates a uniqueQueryID for each of the search queries of the satellite workload.

(c)

Figure 2.2: A read-mostly meteorological workload of satellite applications and the distribution of their search queries.

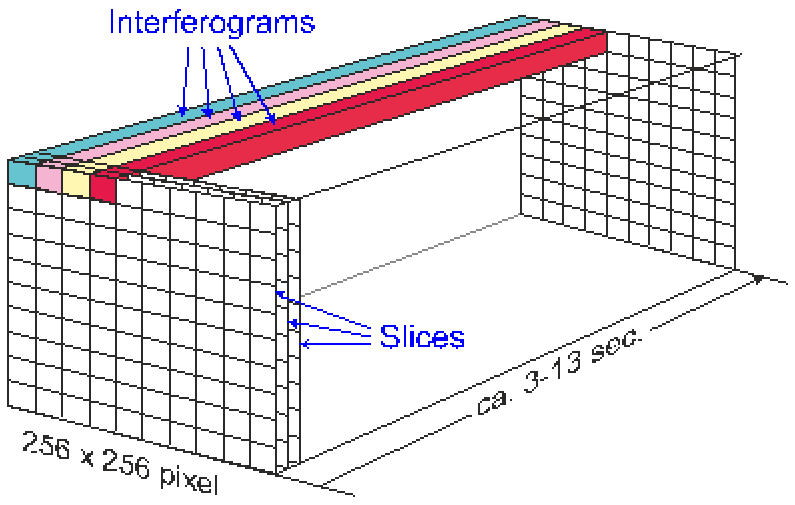


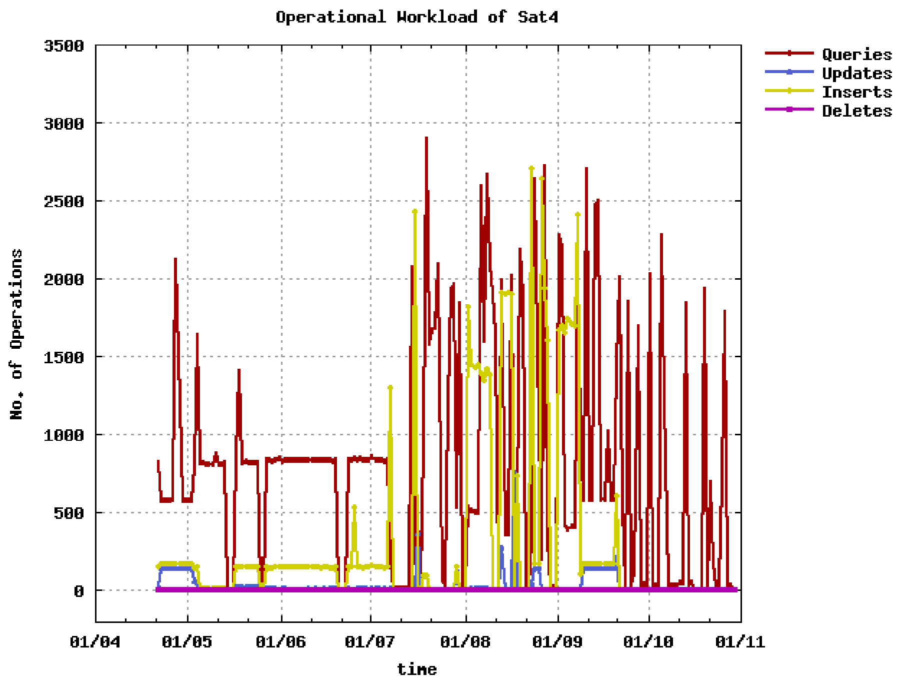
Figure 2.3: The structure of the files of a GLORIA dataset which is known as **cubes**. Each cube contains a measurement sequence of about 3 to 13 seconds. A measurement sequence is a sequence of recorded images which are called **slices**. A single slice is delivered in a time period of 15ns-300ms by a detector field with 256×256 pixels.

the measurement campaigns on board of HALO. A flight delivers a dataset and may take several hours. The files of a dataset are called "cubes" and contain a measurement sequence of about 3 to 13 seconds. A measurement sequence is a sequence of recorded "images" (slices). A single slice is delivered in a period of 15ns-300ms by a detector field with 256x256 pixels. Depending on the scientific needs, only a subset of the full detector array is used for performing the measurements.

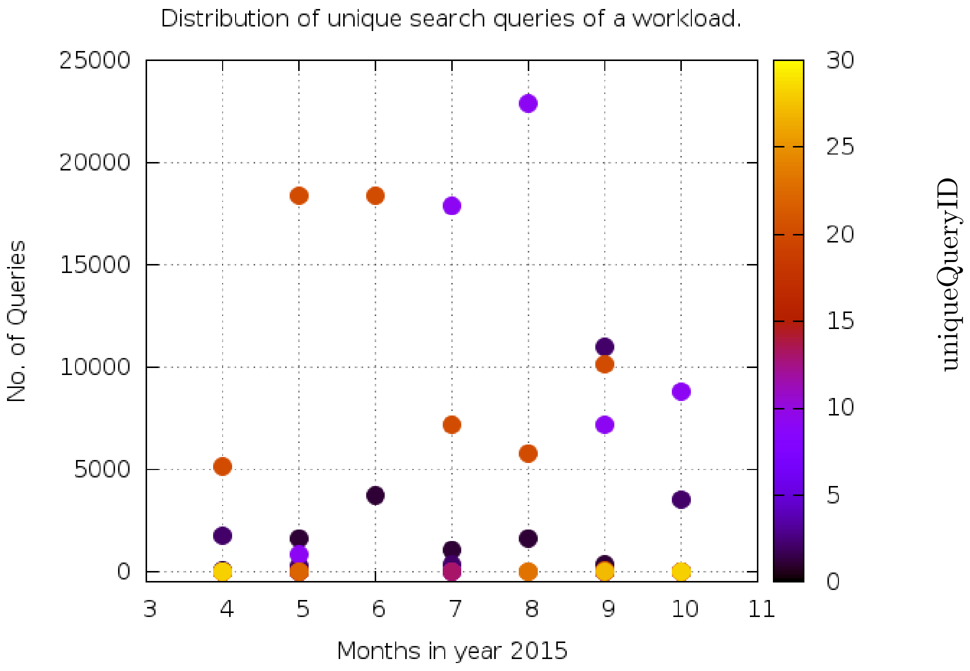
A HALO flight in 2012 took 12 hours and produced about 11.500 datasets ($\approx$ 2.2 TB). Scientists are particularly interested in processing interferogram data. Since GLORIA dataset files contain the ordered list of slices, the datasets must be converted to be stored in MongoDB based on the resulting interferograms.

The first approach to manage GLORIA data was reuse of the software environment of its predecessor instrument MIPAS which is available on aircraft, balloon and for ground-based measurements. Furthermore, MIPAS provided a huge amount of data as one instrument aboard the European environmental satellite ENVISAT.

The software environment for the MIPAS aircraft and balloon experiments is based on Firebird [32], a free parallel development of Borland's^{TM} relational database Interbase. The database manages the metadata of the measurements with links to the data files which reside on the file system. The data generated by GLORIA is approximately hundred times the data produced by MIPAS. For this reason, the data management has to be reconfigured.



(a) Meteorological GLORIA application workload. The mixture of operations in this workload in more than satellite workload. However, there is no *delete* operation in workload as well.



(b) Distribution of 28 unique search queries in GLORIA workload from April to October of 2015. Each colored label indicates a uniqueQueryID for each of the 28 unique search queries of GLORIA workload.

(c)

Figure 2.4: The workload of GLORIA application and the distribution of its search queries.

Additionally, several fundamental new requirements exist for the data management which must be fulfilled by the new environment. The most important requirements are:

- Access to data and metadata within one single database query
- Sustainable long-term usage of well-established database to manage the data
- The data schema should be open for enhancements and modifications
- Horizontal scalability
- Possibility of array-oriented storage of datasets

They all provided reasons to again use the same document-based database (i.e. MongoDB) for the data management of the *GLORIA* application [3]. Figure 2.4 depict the corresponding workload distribution for the *GLORIA* application. Two individual workloads of this application are described in more details as WKL_3 and WKL_4 in section 6.2. The distribution of all queries in Figure 2.4a suggests that this workload has similar characteristics as the *satellite* workload, but there are more queries issued per time unit. Such heavy workload might require many indexes to be created. As machines that host database systems have a limited storage capacity, the creation of all required indexes might not be possible. Therefore, there should be an optimization method to choose the most beneficiary indexes for the whole workload.

The spread of unique search queries is depicted in Figure 2.4b. This figure displays few search queries that are issued to the system on a more regular basis over the course of several months. However, for some of these regularly issued queries to the system, the number of issuances is not that high. Normally, the more frequent a query is, the higher is its need to take advantage of the presence of its corresponding indexes. This fact already suggests a strategy to prioritize indexes.

In this section, the requirement for an adaptive index recommendation system in conjunction with the considered document-based database is identified. In the following of this section, a brief background knowledge on query processing procedure in databases is given. Section 2.3 introduces the criteria of an index recommendation system. Then, the specific criteria of such systems on document-based databases are discussed.

2.2 Background

This section is dedicated to introducing the main concerns and criteria of designing an index recommendation system. To do so, first, some fundamental components of any database system are presented. Then, the role of the query optimizer as the key component to evaluate query plans is explained. Eventually, the major components of an index recommendation system are defined similarly to the structural elements of a query optimizer.

Figure 2.5 illustrates the key components of modern database architectures. Depending on the database type and the query language of the database, the *User Interface* contains a specific protocol that parses each query and translates it to form a *parse tree* that is recognizable by the *Query Processor*. The *Query Processor* layer receives the request from the protocol layer and makes a decision about how to further process the query. At this level, the query is examined both syntactically and semantically. Query processing is done in two different steps: 1- query optimization and 2- query execution [33, Chapter 19]. The *Storage Engine* store and addresses the actual data depending on the data model of the database.

The well-known relational databases store data in tables (relations) where each row (tuple) represents a full object that the columns indicate its characteristics. Each cell of the table should contain scalar values based on the *First Normal Form* (1NF) [34]. Whereas the newly emerged database models such as document-based, graph-based, key-value stores and column-based databases, each have different model of storing data. The document-based databases that are the focus of this thesis store the data in documents instead of tables. Section 2.4.1 discusses this data model in more detail.

The *Query Optimizer* is a key component in a database. It receives the parsed query as input and is responsible for preparing an efficient plan to execute the query [33, Chapter 19]. To fulfill this purpose, the query optimizer searches a vast space of alternative plans. Eventually, it extracts the plan that based on the query optimizer evaluations is anticipated to be least expensive. The output of the query optimizer is known as the *Execution Plan* that defines strictly how a query should be executed.

This execution plan is commonly a tree of *physical operators* that is passed to the *Query Execution Engine* as its input to run the query. The input to each physical operator is a series of documents (or tuples in the case of relational databases), and after application of its role on these documents, this operator passes the subset of the input documents that satisfied the criteria of the operation. Examples of physical operators are *index scan*, *sort*, and *project* [35]. Each operator returns its output to the next physical operator that is in an upper layer of the plan tree. The root operator node of the execution plan tree returns the results for the query. To find proper physical operators and their order in the execution tree to obtain the least expensive plan, the query optimizer must evaluate many alternative plans.

Indexes are data structures that can be created using one or more attributes of a dataset. Indexes contain a copy of the data of its corresponding attributes mostly in form of a tree,

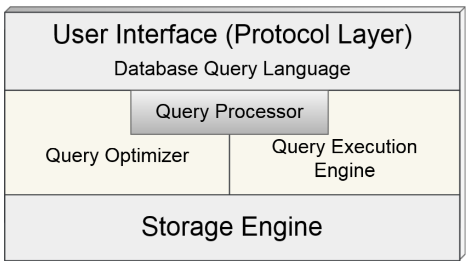


Figure 2.5: The high level architecture of the key components of a typical database.

i.e. B-tree [36] or R-tree [37], that can be searched efficiently. Additionally, the leafs of these trees usually include either a direct link to where the data is copied from or the address of the low-level disk block where the actual data is stored. Therefore, the existence of proper indexes reduces the search time of read queries. However, each index enforces further costs of storage space and writes to the system in case of any update of the data to maintain the index.

In the presence of several indexes in the system, the query optimizer evaluates the cost of running the query with each of such indexes. The benefit of indexes are evaluated based on their *selectivity* as described in Definition 2.2.2. Therefore, query optimization can be seen as a complex search problem that is characterized by three components: 1- *Search space*, 2- *Cost model* and 3- *Enumeration strategy* [1].

Definition 2.2.1. ***Cardinality***:
The *cardinality* of an attribute a in a data set is defined as the number of unique values of that attribute and is represented as CA_a [33, Chapter 3].

Definition 2.2.2. ***Selectivity***:
For any index I, its *selectivity* is denoted by SL_I and defined as the *cardinality* of I's corresponding attributes CA_a over the total number n of documents in the collection, $SL_I = \frac{CA_a}{n}$ [33, Chapter 19].

In the query optimization process, the *search space* defines the set of execution plans that should be considered by the query optimizer. The *cost model* evaluates the consumption of resources by every element of a particular plan and assigns a cost to it. To traverse all of the possible alternative plans in the search space, an efficient *enumeration strategy* is needed [40].

The query optimizer carries out the complex task of evaluating alternative plans by considering the set of already available indexes in the system. Since query optimization is an expensive process, usually to prevent extensive use of it the winning plan is saved as a cache entry. For as long as there is no change in the set of available indexes in the database, the query processor avoids triggering the query optimizer for similar queries to the system. The decision-making process of the query processor is illustrated in Figure 2.6.

The query optimizer bases its decision on the statistics that are typically available to it from the dataset. This statistical information is advantageous to construct an execution plan by estimation of the query result size. These statistics usually contain information about data distribution, as well as a number of distinct values in each attribute (column) [41]. These statistics solely approximate the distribution of values in an attribute. The availability of the statistical information is beneficiary to derive the cardinality of range and join predicates which is one of the main features in the cost-based query optimizers [1, Chapter 2]. A large uncertainty in the estimation of the results size may impair the validity of the decision taken by the query optimizer [42].

Various techniques are suggested in the literature to estimate the results size of a query such as utilization of histograms [43], sampling [44] and parametric techniques [45]. Most database systems save the statistical information about the data distribution of each attribute in histograms [42]. The major advantages of utilizing histogram over other techniques are that for most real-world database they produce an estimation with low uncertainty and occupy a fairly small amount of storage [42].

In a one dimensional histogram, the values of an attribute are divided into some *buckets*. Each bucket is associated with some aggregated information also known as partitioning rule. The aggregated information associated with each bucket as well as the procedure utilized to choose the boundaries for the buckets results in different categories of histograms, such as *equi-width*, *equi-depth*, *max-diff* and *end-biased* histograms [1, Chapter 2]. The accuracy of the histogram, as well as memory usage of the histogram, are affected by the number of chosen buckets.

Independent of their types, histograms represent a compressed value distribution for each given attribute. In general dealing with such a compact data structure requires a couple of simplifying assumptions. In particular, in the case of missing or stale statistics, the query optimizer should apply some simplifying assumptions. For example, in such cases, the query optimizer assumes the values represented in a bucket are distributed uniformly

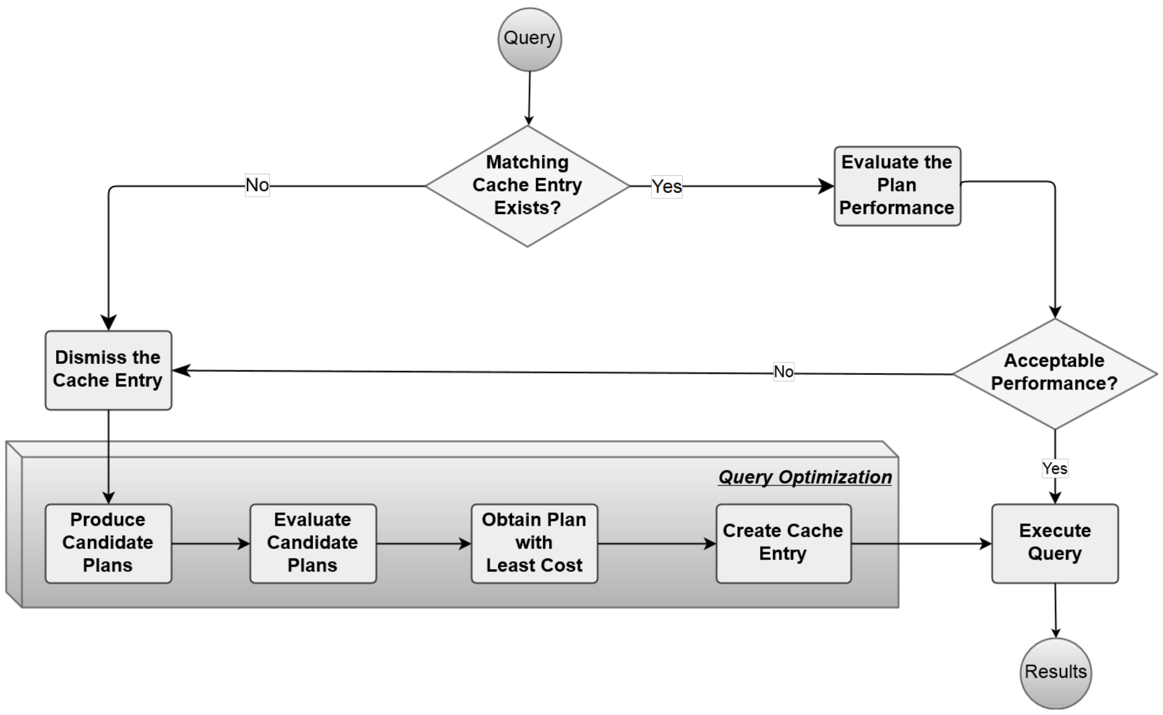


Figure 2.6: The decision tree of a typical query processor functionality regarding utilization of the cache entries or activating the query optimization process. If a cache entry of a plan with satisfiable performance for a particular query already exists and the set of available indexes are not altered, the query processor does not require to trigger the expensive process of query optimization. (Adapted from Figures in [38] and [39])

in the bucket domain even though the attribute contains skewed data. When there are no multi-dimensional statistics available, the assumption of the query optimizer is that predicates in different columns are independent [1, Chapter 2]. Despite the widely usage of one dimensional histograms, multi-dimensional histograms are utilized rather rarely.

After building the histograms, statistical analysis with the purpose of determining stale or missing values are frequently run to verify the stored statistics. The validation process may include the execution of the query or a part of it on a sample set of the input dataset [41]. The sampling methods should be adequately accurate to validate the stored statistics. Many studies have been conducted in the literature on how to obtain an optimal sample size that leads to a more accurate and precise estimation of the result size of the query [41], [46], [42].

Usually, the creation of an index with leading key attribute A_1 also returns a histogram on that attribute. As the index includes all values for an attribute in sorted order, the histogram creation can be conveniently added on top of index construction. As a result, by the construction of an index, the query optimizer is also given better estimates for particular predicates, which can influence the chosen execution plan [1, Chapter 2].

Consequently, whereas the query optimizer of the database is responsible for carrying out the complex task of choosing the most efficient plan considering the existing indexes in the system, often an even more crucial task is to provide a set of proper indexes to the query optimizer. Unless there is a corresponding proper index for a query, the query optimizer is forced to select a full collection scan as the winning plan.

The database administrators usually utilize a series of heuristics that are developed over time as the norm to choose the appropriate index for a corresponding query [47]. However, heuristics are not sufficient anymore due to the advancements in the query optimizer designs and establishment of different database types with distinct query languages. This method of manually choosing and creating indexes requires experienced administrators with extensive knowledge of the system and frequent monitoring of the database workload to adjust the proper indexes. This is not scalable for large datasets and complex workloads.

As a result, there is a high demand for assisting tools that can monitor the database workload continuously, identify a candidate set of indexes based on a comprehensive analysis of the way the query optimizer works, and recommend the most proper set of indexes. Such an index recommendation system facilitates the adaptation of the database to the detected changes in the workload. A concrete definition of the task for an index recommendation system and a brief overview of the design concerns for such system are presented in next section.

2.3 Introduction to Index Recommendation Criteria

The definition of the task that an *Index recommendation* system should accomplish is described in Problem 2.3.1.

Problem 2.3.1. ***Index Recommendation Problem:***
Given a workload $W = \{q_1, q_2, \cdots, q_n\}$, where each q_i is a query, and given a total storage

limitation S acquire the subset $C = \{I_1, I_2, \cdots, I_m\}$ of all possible indexes - known as a configuration- such that 1- $\sum_j size(I_j) \leq S$ and 2- $\sum_i cost(q_i, C)$ is minimized, where $cost(q_i, C)$ is the estimated cost of the execution plan for query q_i by the query optimizer when all and only indexes in configuration C are accessible for it.

Therefore, the index recommendation system should be able to evaluate a large set of alternative plans per query in the workload and extract the most efficient one. This operation is also known as *index selection* in the literature and Douglas Comer in [48] proves that it is a NP-complete [49] problem.

Based on the definition of the *Index Recommendation* in Problem 2.3.1, it can be characterized similar to the query optimizer tasks. Therefore, despite different considerations, the index recommendation problem can be analyzed regarding the same components as query optimization problem, by characterizing the search space, a cost model and choosing an efficient enumeration technique. Based on the challenges appeared in each of these steps, I developed solutions that are presented throughout this thesis under an integrated system named *Adaptive Index Recommendation system* (AIRs). A brief description of the most significant challenges and considerations in the above-mentioned steps and their corresponding developed solutions are as the following:

1. The search space of the index recommendation problem consists of various subsets of all possible indexes.

   Problem 2.3.2. ***Search Space Problem:***
   The number of possible combinations out of the set of all indexes is enormous. Therefore some strategies to extract the necessary fraction set of all index combinations are required.

2. In contrast to the cost model of the query optimizer that is used to optimize the cost of one query, the cost model of an index recommendation system should optimize the cost for the entire workload of queries to the database at least for a defined period. A recommended index that is disregarded by the optimizer should be avoided.

   Problem 2.3.3. *Evaluation of the profit of any set of indexes concerning the incoming workload requires modeling an objective function.*

3. Even by reducing the number of candidate indexes, the evaluation of the index combinations by the query optimizer can exert a lot of load on the in-production system. The main reason is to obtain the statistics regarding the cost of execution of any query with each particular index from the query optimizer, that index should be created. Adapting an appropriate strategy to extract the index statistics without overloading the database system is substantial.

   Problem 2.3.4. *Extraction of the corresponding statistics regarding any set of indexes from the query optimizer enforces lots of additional load on the database system.*

4. Similar to the functionality of a query optimizer, traversing the space of alternative plans with an efficient enumeration technique is essential. To ensure rapidity of the enumeration process, query optimizers mostly rely on *greedy* algorithms, which do not guarantee to always find the global optimal solution [50]. Although the speed of the enumeration technique for the index recommendation can be a factor, there can be a trade-off between speed of the algorithm to the extraction of the globally optimal solution.

   Problem 2.3.5. *Extract the globally optimal solution to the objective function of the index recommendation system.*

5. Due to the utilization of individual and non-standardized procedural languages by document-based databases instead of the declarative SQL [51], as well as a different usability of these databases, the standard benchmarking convention of the database society, such as TPC [52], are not practically helpful. Therefore other methods should be applied to determine the quality of the solution presented by AIRs.

   Problem 2.3.6. *Benchmarking non-relational and specially non-SQL related systems is not yet fully standardized.*

The solutions of the above-mentioned problems are applied in different segments of the AIRs and are discussed throughout this thesis. Chapter 2.4 is dedicated to specifying the differences in studying the index recommendation system on a document-based database compared to a relational database.

2.4 Index Recommendation Specifications on Document-Based Databases

Unlike well-known traditional relational databases that store data in *relations* (tables), the data model of document-based models is found on storing data in *documents*. This fundamentally different data model enforces many distinctions between these databases that can affect databases as well as tools that interact with databases. Therefore, the first crucial step towards the design of an index recommendation system is to study these differences and their possible effect as well as the impact of all common potential parameters.

This section is consecrated to define these differences and their influence in designing the AIRs. In section 2.4.1, the basic data model of a typical document-based database and its influence on the indexing process are introduced. Section 2.4.2 presents the description of the non-standard procedural query language that is later exploited throughout this thesis. The assumptions about the behavior of the query optimizer of the database under study are defined in Section 2.4.3.

2.4.1 Data Model and Indexing

The principle idea for document-based databases is to store data in *documents*. A *document* is usually in a markup standard format like JSON, or XML. Each document contains several *attributes* that are build by an arbitrary name. Attributes are associated with some *values* (similar to key-value structure). Documents are classified together in directories that can be named as *collections*.

Document-based databases are *schemaless* [53]. Therefore, neither the attributes nor the type of their values is predefined. As a result, the appearance of any attribute in a document is optional, and the documents in the same collection are not bound to contain a similar set of attributes. Even for two documents that have the same set of attributes, the type of their associated values are not forced to be identical. Unlike in the relational data model, the value in a document-based model can range from any scalar value to a list (array) or even a whole other document inside the current document (*nested documents*).

The concept of nested documents is considered as a replacement for *JOIN* operations in relational databases [5] as expressed in Assumption 2.4.1. This is one of the key solutions of document-based databases to increase the performance. The idea is that a document, including all its nested documents, is stored in a single location in persistent memory. Accordingly, even if the database store the data on distributed hosts, once queried the documents can be fetched from one physical host where they are stored. Hence, the number of I/O operations are reduced.

Assumption 2.4.1. *There is no JOIN operation used in document-based databases to answer a query.*

Despite all these differences in the data model of document-based and relational databases, the construction of indexes is essentially alike. Indexes are usually B-tree or B+-tree structures with search, insert and delete time complexity of $O(\log n)$ [54]. Each tree is constructed with the values of an attribute or a set of attributes depending on the chosen keys in the corresponding index. The leaves of these trees form a sorted list of these values. These leaves contain pointers that indicate the physical address on disk where the values are stored.

Indexes are built on collection level and can contain one single attribute or a combination of attributes (*compound indexes*). If the information that is requested is not entirely stored in the index, or in other words if an index does not contain all of the attributes that appear in projection part, the whole document should be fetched from disk. To prevent such expensive fetching, it is best to consider including more attributes in the form of compound indexes that are appearing more often in one query.

In relational databases, another type of indexes is commonly used known as *clustered index*. Instead of containing a pointer to the actual records, the cluster indexes include the actual record itself. Base on the structure of document-based databases and them being schemaless, the assumption is that no clustered index can be made on them.

Assumption 2.4.2. *Building clustered indexes is not possible in document-based databases.*

The fundamentally different data model of document-based databases requires its specific query language to access and manipulate data. Section 2.4.2 presents a definition for this specific query language that is used throughout this thesis.

2.4.2 Procedural vs. Declarative Query Language

The conventional declarative *Structured Query Language* (SQL) is defined based on relational algebra [55]. It is widely used to interact with relational databases and to operate on tables. The essentially different data model of document-based databases calls out for its specific query language. Due to the structure of documents, normally a procedural query language is defined to access their contents. Despite the similarity of the data model and utilization of a typically procedural query language, there is no unified query language utilized by all document-based databases [5]. Therefore, in this section, a definition of the query language that is used throughout this thesis is presented. This language covers all of the relevant aspects of targeted database query language, but it is general enough to describe the query language of any other database.

The content of documents can be presented in tree structures. These contents can be searched and accessed by the path to their attributes. Therefore, a procedural language that specifies exactly how to navigate to the data is commonly utilized by document-based databases. These languages mainly consist of four basic operations to create, retrieve, update and remove data. Based on Assumption 2.4.1, there is no join operation. These operations are covered by *insert, read, update and delete* operations in the language. These operations can be constructed by *search-predicate*, *projection-predicate*, *sort-clause*, *modification-predicate* as shown in Code 2.1 and 2.2.

Code 2.1: The definition of the retrieval operation in a procedural language of document-based databases.

```
db.coll.READ({search-predicate}{projection-predicate}).SORT({
   sort-clause})
```

Code 2.1 represents the definition of a *read* operation in this procedural language that is used to retrieve data. The *db* and *coll* are respectively indicators for the targeted database and collection that can be replaced by the intended names. The *search-predicate* contains conditions that should be fulfilled when searching for the intended documents. These conditions can be formulated by a combination of *equality* or *range* condition on each intended attribute. Also, equality or range conditions on multiple attributes can be combined by any *conjunctive operation* as *OR* (indicated with $\vee$) and *AND* (indicated with $\wedge$). If this predicate remains empty, all documents in the collection are searched.

The *projection-predicate* in Code 2.1 contains the attributes that should be returned from the documents that match the conditions of the *search-predicate*. The specification of the *projection-predicate* is optional. In case it is not specified, the whole documents that satisfy the conditions are returned. Each *READ* operation can potentially be combined with a *SORT* operation as shown in Code 2.1. The sorting process is done based on the attributes appeared in the *sort-clause*. An example of a read operation is presented in Example 2.4.1.

In Example 2.4.1 - as well as all the other example in this thesis - certain values are identified with capital letters. Also, the path to any attribute b_j of a document nested under attribute a_i is shown with a dot separator as $a_i.b_j$. The same path specification applies for indexes of arrays. The *loc.coordinate.1* in the *search-predicate* of Example 2.4.1 represents the path to the value of the second element in the array of value associated with attribute *coordinate*. Where the attribute *coordinate* itself is an attribute of the document that is assigned as value to attribute *loc*.

Example 2.4.1. db.coll.**READ**({"att-time" $>$ TB $\vee$ "loc.coordinate.1" $=$ LOC}
{"sun-shine"}).**SORT**({"att-time"})
The *search-predicate* of this query determines conditions to find the documents with their *att-time* attribute is greater than the given time as *TB* or the value of their *loc.coordinate.1* element is equal to the given location as *LOC*. The matching documents are first sorted based on the value of their *att-time* attributes. Then only the *sun-shine* attribute of each document and its corresponding value is returned.

Code 2.2 represents the definition of any *WRITE* operation in the procedural language. The *WRITE* command is an indicator that can be replaced by any of the following operations: *UPDATE*, *INSERT*, or *DELETE*. The *search-predicate* of this operation, similar to the *READ* operation defines the conditions to find the intended documents. Insert operations do not contain any search part. Therefore, the *search-predicate* in insert operations is empty , as shown in Example 2.4.2.

Example 2.4.2. db.coll.**INSERT**({}, {"normal-array"=[N1, N2, N3],
"nested-doc"={"nested-att"=NA}})
According to this insert query, a document with two attributes *normal-array* and *nested-att* is created. The *normal-array* contains as value an array of three elements and the *nested-att* a nested document with one attribute.

Code 2.2: The definition of the write operations in the procedural language of document-based databases. The WRITE operation can be replaced by an update, insert or a delete operation. The search-predicate is empty for an insert operation.

```
db.coll.WRITE({search-predicate}{modification-predicate})
```

The *modification-predicate* in Code 2.2 contains attributes and values that should be altered in the matching documents. If the specified attribute in this predicate already exists in the document, only the value of it is updated to the newly determined value. However, if the attribute does not exist in the matching documents, this attribute and its corresponding values are added to the document, as discussed in Example 2.4.3.

Example 2.4.3. db.coll.**UPDATE**({ BM < "match-att" < EM }
{"first-att" = FA, "match-att" = SA})
This update query searches for all of the documents where attributes *match-att* is greater than given value *BM* and less than *EM* value. With the assumption that attribute *first-att* did not exists in this document, this operation adds this attribute with the given value *FA* to the document. It also replaces the value of attribute *match-att* with the value *SA*.

If the *WRITE* operation is *delete*, the specified attributes in the modification part are removed from the matching documents. However, if the *modification-predicate* of a delete operation is empty, the whole matching document is removed.

Example 2.4.4. db.coll.**DELETE**({"any-att" = A})
This delete operation searches for a document where the attribute *any-att* is equal the give value *A*. It then drops the entire matching document.

The different data model and the assumptions that were defined in Section 2.4.1 required for a different query language for document-based databases. The definitions of the procedural query language presented in this section are used all through this thesis. The differences are not limited to the query language, but also to the rules of thumbs that the query optimizer of the document-based databases is applying to produce the optimal plan. These major considerations of the query optimizer of document-based databases are explained in Section 2.4.3. All of these differences also result in disqualification of the standard benchmarks of relational databases to evaluate document-based systems. This issue is discussed in Section 6.1.

2.4.3 Query Optimizer Behaviour

The application of a different data model and query language impose special query optimization considerations on the query optimizer of document-based databases. Although the basic functionality of the query optimizer as described in Figure 2.6 remains the same, the rules that are applied to produce and evaluate candidate plans are different. The selection of the candidate indexes in the *index recommendation system* should be carried out with regards to the rules of the query optimizer (see Section 4.3). These rules are directly dependent on the set of *physical operators* that the *Query Execution Engine* is capable to perform. However, unlike traditional relational databases, NoSQL databases in general, and document-based databases in particular do not have a unified query language and thus a unified basic set of physical operators. Therefore, it is requisite to define the set of optimization rules evaluated by the targeted database query optimizer. In the rest of this section, first the type of some of the optimization tasks for a document-based query optimizer are described. Then, the assumption of the main rules applied by a typical document-based query optimizer to choose a plan are introduced.

The search space of a query optimizer consists of the alternative plans for execution of a given query. The construction of these alternative plans is dependent on the series of *physical operators* supported by a *Query Execution Engine*.

The assumption is that the document-based databases eliminate usage of *JOIN* operators (Assumption 2.4.1). Also, the indexes are constructed on collection level. As a result, no query accesses multiple collections at the same time. Therefore, all of the query optimization tasks concerning *JOIN* operations, e.g. utilization and operator reordering of *Loop Join*, *Merge Join*, or *Hash Join* [1, Chapter 2], are irrelevant for a document-based query optimizer.

However, there remain other sources of variety for a query optimizer to generate alternative plans. One of the most important ones is the choice of the *access path.* If no indexes are available for a query, the default plan generated by the query optimizer is to access the intended data by performing a full collection scan. Modern query optimizer can utilize single- or multi-attribute (compound) indexes. In the presence of various indexes, more alternative plans can be constructed by performing an index seek utilizing the indexes.

The plans are constructed by prioritizing the set of rules described in Definition 2.4.1. Then the cost of each index is evaluated by the optimizer based on the *selectivity* of that index (see Definition 2.2.2). The lower the selectivity, the more efficient the index is.

Definition 2.4.1. ***The Query Optimizer Selection Rules***:
The selection of indexes is performed by considering the following rules:

1. If the query includes a *sort-clause*, try to sort using an index.

2. Satisfy conditions in the *search-predicate* of the query utilizing indexes.

3. If the query contains a *range condition* or *sort-clause*, choose the index which its last attribute can satisfy the range or sort.

The first rule indicates that execution of a *SORT* operation utilizing an index is the priority. Both single- or multi-attribute indexes can be used to satisfy the *sort-clause.* The same is true to satisfy the second rule for attributes in the *search-predicate.*

However, if the query does not contain any *projection-predicate* or the index does not contain the information for the specified attributes in the *projection-predicate*, the whole matching document should be fetched from disk to memory. This process slows down the response time with several order of magnitudes. Therefore, it can be helpful to benefit from multi-attribute indexes that cover more information.

Additionally, the second rule points out another capability of the query optimizer. For queries with more complex predicates, the query optimizer is also capable of utilizing several indexes simultaneously with what is called *intersection of indexes.* The idea behind *index intersection* is to exploit multiple-conditions *selectivity* by simultaneously scanning the single indexes of each condition.

Example 2.4.5. Consider the following conjunctive condition: $A_1 = 10 \wedge A_2 =$"string", where A_1 and A_2 are two single attributes in documents of a collection. The assumption is that there are single attribute indexes available on attribute A_1 and A_2. By exploiting these indexes, a set of pointers to the documents that fulfill each condition can be obtained.

The query optimizer applies many rules. The rules that are more relevant to the decision making and extraction of criteria in an index recommendation system are described in this section.

2.5 Summary

In this chapter, a description of the application of meteorological databases that provided the motivation for the research for an adaptive index recommendation system on the document-based database is presented. The specific requirements of these applications demanded the usage of document-based databases. The distribution of different operations through time and the unique queries in the corresponding workload of these applications raised the awareness about the lack of an adaptive index recommendation system especially with the considered document-based database.

To help the reader to navigate through thesis, a summary of the background knowledge representation and a briefing about the challenges of index recommendation problem are given. Then, to lay the basics for AIRs design, the distinctions between traditional relational databases and document-based ones regarding design an index recommendation system are introduced.

For this purpose, the data model of the document-based databases is described. Due to the diversity of the models between different document-based databases, the assumptions of the behavior of a typical document-based database that the research in this thesis is based upon are introduced.

It is deduced that the entirely different data model of the document-based database requires other query language than the standard SQL. As there is no unified query language for NoSQL databases in general, and document-based databases in particular, the query language that is utilized throughout this thesis is described.

As a result of the different data model and query language, the tasks of the query optimizer also differs for document-based databases. The fundamental rules that are assumed a typical document-based database applies are presented.

At last, based on these characteristics, the parameters that must be considered in the design of an adaptive index recommendation system concerning a document-based database are presented. Different segments of the index recommendation system contain various solutions containing these parameters.

Chapter 3

Related Work

Indexes are physical structures that can significantly increase the performance of the database. The design of these physical structures along with the capabilities of the query optimizer and engine of the targeted database determine the efficiency of a query execution. Automatic recommendation of required indexes is non-trivial. The significance of this matter has engaged several scientific [56], [57], [58] as well as commercial study groups [59], [60] for years. In this chapter, some of the related work in the field are discussed. Also, more recent and leading studies on the subject are mentioned and are discussed in context of this thesis. Due to the broadness of the aspects that index recommendation is associated with, the related work is discussed concerning each issue in the rest of this chapter.

Parts of the following text is published in [61].

3.1 Complexity of Index Recommendation Problem

In 1978, D. Comer investigated the complexity of the selection of a set of indexes out of all possible indexes [48]. Since the focus of his work is merely on the proof of difficulty of the *Optimum Index Selection Problem* (OISP), a simplistic form of ISP is defined as the following: For any given file F with n records, k attributes, and a given integer p, is there an index set for F with size not larger than p? Even with the assumption that the attributes of the file are not to be combined, Comer reduces the NP-complete problem *Satisfiable with exactly 3 literals per clause* (SAT3) [62] to OISP. This proves that OISP is an NP-complete problem and there is no known algorithm to solve this problem in less than exponential run-time for any arbitrary inputs.

Then, Piatetsky-Shapiro in [63] also by reducing the problem of selection of secondary indexes to the *Minimum Set Cover* [49] proved that it is an NP-complete problem.

However, Chaudhuri et al. in [64] investigated the selection of clustered and non-clustered indexes with the assumption of two constraints: 1- the additional storage required to build the proposed indexes should not exceed the storage limit, 2- for any given table in the database, it is not possible to build more than one clustered indexes. By reduction of the non-clustered index selection problem to the k-densest sub-graph [65], they proved that it is an NP-hard problem.

Although in Section 2.4.1 we assume that building cluster indexes is not possible in the collections of the document-based databases (see Assumption 2.4.2), according to the above-mentioned studies, any solution to the recommendation of the sub-set of proper indexes is computationally hard.

In general, the index recommendation problem can be deliberated as a complex search problem that has a *search space*, a *cost model* to evaluate each solution, and an *enumeration method* to traverse the space of possible solutions. The steps taken in the advancement of any of these areas of research over assisting physical design tools such as index recommendation systems are discussed in the following of this section.

3.2 Search Space of the Index Recommendation System

The search space of the index recommendation system is composed of candidate indexes. There are various methods to select the candidate indexes. A common method to extract the candidate indexes is to execute a syntactic analysis of the query [66], [67], [68]. This parsing can be done on the query string and can extract the indexable attributes from the parse tree which is known as *Parsing-Based Approach* or from the execution plan generated by the query optimizer which is known as *Execution Plan-Based Approach* [1, chapter 4].

The primary studies only included the selection of single-attribute indexes [69], [70], [71], [72]. However, considering the ability of all modern query optimizers in processing multi-attribute indexes, it is important to recommend them as candidates when required. There are different methods to construct a proper subset of multi-attribute candidate indexes for a workload such as:

1. iterating from single-attribute indexes to multi-attribute candidate indexes [66],
2. clustering single-attribute indexes in various queries together [67],
3. utilization of a Frequent Itemset algorithm [73] to determine single- and multi-attributes at the same time [68].

Not only to build the proper set of multi-attribute candidate indexes, but also to consider the candidate indexes only for frequent queries of the workload, the Frequent Itemset algorithm is adapted in this thesis as well. However, to not ignore the queries with long run-time, the candidate indexes of such queries are also added to the candidate set of indexes.

Additionally, some index transformation techniques can be used to consider sub-optimal candidate indexes that might fit better into the environmental constraints such as storage limitation. One of these transformation techniques that is leveraged in this thesis as well is *index merging* [74]. The idea of *index merging* is to combine two indexes that their first starting key is similar with a utilization of the starting key and combination of rest of their attributes.

Furthermore, typically any subset of these candidate indexes should be evaluated in the index recommendation process. These sub-sets of indexes are known as *configurations.* Depending on the number of candidate indexes, the number of all possible configurations is enormous. Utilization of the *atomic configuration* concept [75] assists in reducing the search space of the index recommendation problem. The atomic configurations are configurations that all of their indexes are used for the execution of some query. The execution cost of queries for any other configuration can be calculated based on the execution cost of the atomic configurations. Since the atomic configurations are only a fraction of all possible configurations, leveraging this concept results in a major reduction in the search space of the index recommendation problem.

3.3 Cost Model

As discussed before, the index recommendation problem can be seen as a complex search problem with a possibly large search space of configurations. Although there are various methods how to find the solution for this problem, the fundamental common requirement of them all is the ability to evaluate the expected data access cost for each given query in the presence of every configuration in the search space. Therefore, a cost model should be designed to promote the configurations with a better fit.

Many studies on the index recommendation problem merely focused on designing a cost model that defined the benefit of indexes for read queries with storage constraint [67], [66]. However, an accurate cost model should take into account the maintenance cost of indexes in the presence of update operations as some suggested models in [76], [68]. The early studies on the index recommendation problem tended to develop an external cost model to independently evaluate different sets of indexes [69], [70], [71], [72], [77].

However, the efficiency of any set of recommended indexes for a given query depends on the design and efficiency of the query optimizer of the targeted databases. Therefore, a major trend in the design of index recommendation systems is to communicate with the query optimizer of the targeted database to estimate the worth of any set of indexes for a given query [78], [79], [66], [68].

Utilization of the query optimizer to estimate the profit of any set of indexes has several advantages such as: 1- the recommended index sets are assured to be used by the query optimizer to execute the query, 2- the recommendation system benefits from all of the performance optimization aspects that the query optimizer takes into consideration, and 3) any modification in the cost model of the query optimizer is automatically included in the index recommendation system [80]. This way, the capability to run the query with multiple indexes by intersecting them together is also taken into account [64].

3.4 Query Optimizer Considerations

The *search space* of any query optimizer depends on the number of *physical operators* that the database engine can perform. In query optimizers of relational databases, the *search* *space* also revolves around the number of equivalent algebraic transformations that can be carried out by the database engine. Many of these transformations are by some means related to properly placing the *JOIN* operations in sub-plans [1, chapter 2]. Examples of such considerations are the determination of the order of *GROUP-BY* and *JOIN* clauses after each other [81] and simplification of the execution of an *OUTERJOIN* by pulling it above a block of *JOIN* operations [82].

All of the above-mentioned optimizations in the *search space* are tightly associated with the relational data model and normally their *SQL* language. However, since the document-based databases mostly eliminate the *JOIN* physical operators [53, Chapter 5], these transformations are not considered in their optimizers.

Nevertheless, there are many other transformations in the *search space* of the query optimizer that are relevant for both relational and document-based databases such as *access path selection* [83]. Naturally, the considerations of the query optimizer regarding the common transformation for relational and document-based databases are specific to the data model and the query language of that database. This is also true for the case of the *cost model* and the *enumeration strategy* of the query optimizer.

In the case of the enumeration strategy, the query optimizer of the document-based databases mainly follow the *extensible optimizer* paradigm [84] that mostly relies on the *Volcano/Cascades* optimization framework [85]. However, in a document-based database, the rules of the query optimizer are defined based on the criteria of the data model and language. An example of the rules applied in the query optimizer of a particular document-based database is given in [86, Chapter 7].

By the time of writing this thesis, to the best of my knowledge, all of the published scientific studies on the index recommendation problem are applied to relational databases[1]. The behavior of the query optimizer and the rules that it applies to determine the optimal plan directly affect the syntactic methods used in an index recommendation system to extract candidate indexes and the estimated costs of running a query with a distinct set of indexes.

3.5 Reduction of Load on the Query Optimizer

Despite all of its advantages, obtaining the cost of running queries under various index sets from the query optimizer puts lots of load on the database system. This extra load can affect the performance of the database in response to its application queries. Therefore, many studies have been conducted to reduce this load on the query optimizer.

One well-known method to lessen the load of the evaluation of queries by the query optimizer is the implementation of a *What-If optimization* environment [89]. This environment enables the simulation of the existence of a hypothetical set of indexes in the database management system. Therefore, the process of optimization of different queries can be done

[1]The author is aware of the attempt to adapt AutoAdmin [59] to be applied to their cloud-based document-based database *DocumentDB* [87] in Microsoft Azure according to [88]. However, by the date of publication of this thesis no scientific paper is published about this process.

with the assumption of the presence of that set of indexes without materializing those indexes in reality. For any given query (read or update) and a set of indexes, the *What-If optimizer* produces the cost estimation of the optimal execution plan for the query with the index set. Such environment is extensively utilized by index recommendation systems and thus is adapted into many modern databases [90].

Nevertheless, the *What-If* optimization is an expensive process and is itself often the bottleneck in the index recommendation process. Therefore, an enhancement study is conducted to reduce the overhead of the *What-If* optimization by a technique called *INdex Usage Model* (INUM) [57]. INUM is designed to function in cases where a large number of *What-If* optimizations should be performed for the same statement. It is considered as a fast alternative to the *What-If* optimization. INUM functions in two phases:

1. For a given query q, it makes few carefully chosen calls to the *What-If* optimizer to obtain the potential set of optimal execution plans of q. INUM caches these plans and utilizes them when any *What-If* optimization regarding q is required.

2. It transforms each of the optimal cached plans of phase 1 to use the given set of indexes as the access plans. INUM introduces the transformed plan with the lowest cost as the optimal plan [90].

However, the usage of the *What-If* optimizer or *INUM* requires the development of an additional component within the database. This *What-If* component should collect the same statistics about the data set as the ones gathered by the query optimizer to build the necessary histograms [91]. As an alternative solution, the logic of gathering statistics is leveraged in the designed solution of this thesis, to develop a *virtual environment* consists of a sample set of the original data set where all of the candidate indexes can be materialized once. Then the cost of running the query with each configuration can be simulated in this environment. This design is significant because index recommendation systems typically rely on the design of the database system itself, such as the existence of a *What-If* interface [92]. However, the *virtual environment* design enables the index recommendation system to cooperate with any database without requiring to alter internal components of the database.

Since the estimation obtained from the query optimizer relies on the statistics that it gathers from the dataset (i.e. histograms and cardinality of attributes), it is imperative that the sample set carries approximately the same statistical ratio for all attributes in the dataset. This approach is inspired by and based on the idea of utilization of random sampling for the approximation of histograms provided to the query optimizer to estimate the number of results of a query [93], [94]. Piatetsky-Shapiro and Connell prove in [95] that for any single given query, just a small sample size is enough to estimate an accurate histogram with high probability for that query. However, for the purpose of building the *virtual environment* with a representative sample for the index recommendation system, a histogram should be derived that is reasonably adequate for all of the queries in the workload or a large number of them. Gibbons et al. in [46] determined a bound on the requisite sampling size which is independent of the distribution. The focus of their work is on maintaining histograms incrementally. Chaudhuri et al. in [96] introduce an error metric that leads to a specification of a much stronger bound on the crucial sample size. The results of this method are adapted to conduct the study on the essential optimal bound on the trade-off between the size of samples and the derived error on the histograms.

Additionally, to reduce the overhead of calls to the query optimizer - regardless of *What-If* or our *virtual environment* - the concept of *atomic configurations* is utilized. As mentioned earlier in this section, the execution cost of queries for any configuration can be computed based on the execution cost of the atomic configurations. Therefore, the number of calls to the query optimizer decrease, because only these atomic configurations should be evaluated.

There are various methods to extract the atomic configurations. Chaudhuri and Narasayya in [75] discuss a static strategy based on the restrictions of the query processors. The idea is that the atomic configurations can be constructed upon characteristics of the query processor such as defining the maximum number of indexes that can be intersected to run a query or number. In contrast to exploit a static method, an adaptive strategy is further developed in this thesis.

3.6 Enumeration Technique

The configurations in the search space of the index recommendation problem should be traversed efficiently to be evaluated according to the designed cost model. The enumeration techniques can generally be classified into two categories: *bottom-up* and *top-down* approaches [1, Chapter 6]. Each of these approaches has its advantages and disadvantages.

The usage of a *hill-climbing approach* also known as *Greedy algorithm* [97, Chapter 16] is an example of a *bottom-up* approach that is commonly used to solve the index recommendation problem [68], [72], [70], [75]. The ascending greedy algorithm starts from an empty set of candidate indexes and incrementally (greedily) adds more candidate indexes attempting to maximize the benefit of the indexes. This process terminates as the benefit starts to decrease or the storage limit is reached. Despite its relatively fast process, the greedy algorithm does not guarantee to always find the optimal global solution. It might stop at a local maximum solution. This approach is the common solution used in the Microsoft's *AutoAdmin* index selection tool [59].

Another commonly used *bottom-up* approach to find the final configuration from the set of candidate indexes is to assimilate it into a form of the *Knapsack* problem [98]. The studies on the index selection problem conducted in [99], [66], [100], [64], [101] and [102], all formulated it as a *Knapsack* problem where each index represents an object, the index storage size is considered as the weight of each object, the chosen cost of running the workload determines the benefit objective function, and the total storage limit is the knapsack size. Then the problem is often solved by a greedy approach. This approach is the core of the index selection tool of *DB2 Advisor* [60].

Additionally, derivation of randomized methods such as *Genetic algorithm* [103] are also adapted to formulate the index recommendation problem. Kratica et al. in [76] defined the primary population for the Genetic algorithm to be the candidate index set. The objective function to optimize is similar to previous approaches. This method evaluates the cost of running the workload with different configurations. Then the combinatorial manufacture of the final configuration is performed by mutation, crossover, and selection genetic operators [68].

In general, the *bottom-up* approaches are more beneficiary in cases where the storage capacity is limited to a small size. However, in modern, scalable systems, often the storage is not tightly restricted. On the contrary, the *top-down* approaches start with a globally optimal solution that normally exceeds the storage limit. Then, they gradually temper this initial configuration so that it will fit into the storage constraint. The top-down approaches are more desirable, especially in cases where the storage capacity is not that low. Through a relaxation of constraints, the top-down approaches determine measures of approximate optimal solutions [104]. Therefore, they provide information about more efficient configurations than the final recommended one, that does not fit into the constraints. This information can be valuable to make better decisions for resource management of the system such as required increase in the disk or memory storage of the database environment.

A remaining open research question is if hybrid schemes build upon the approaches mentioned above can improve the index selection problem [1, Chapter 6]. Additionally, as noted before, most recent advances in index recommendation approaches all suffer from a common drawback: they do not guarantee the optimality of the final solution [64]. In most of the approaches above, the final recommended configuration can be a locally optimal solution instead of a globally optimal one.

Therefore, in the course of this thesis, the search space of the index recommendation is first narrowed down with the assistance of a top-down approach in the form of *Index Benefit Graph algorithm* [72]. Then to assure the discovery of the global optimal solution, the index recommendation problem is formulated in *Integer Linear Programming* [105] which is traversed by the *Branch-and-Bound algorithm* [106].

The usage of the adapted *Index Benefit Graph algorithm* provides all of the advantages of top-down approaches in this work. An important aspect that is covered is the possibility of interaction of indexes [107]. Interaction of indexes in the form of index intersection is an important common functionality of modern database optimizers to answer a query. The coverage of index intersections can only be estimated utilizing top-down approaches. In bottom-up approaches where the cost of running a query with each index should be defined individually and before the enumeration, it is not possible to cover the index intersection possibility for queries. The complexity of the adapted *Index Benefit Graph* algorithm to extract the index intersections is a subject of study in this thesis.

The usage of an *Index Benefit Graph* in combination with the usage of *Frequent Itemset* and long query strategy helps to restrict the size of the search space for the index recommendation problem. Hence, the utilization of the *Integer Linear Programming* becomes practical.

The *Integer Linear Programming* (ILP) is itself an NP-hard problem [108]. However, for a small search space, it can return the optimal global solution quickly. Papadomanolakis and Ailamaki in [56] presented a formulation of the index selection problem in ILP. However, the cost model of that design for write operations is not complete. This thesis provides a complete formulation of index recommendation problem in ILP that takes into account not only the effect of updates and the ratio of read to write, but also the selectivity of candidate indexes. Also, a sensitivity analysis of this model is provided.

3.7 Benchmarking Challenges

Despite the extensive research that has been conducted on the index recommendation problem over the past years, not much attention is paid to systematic evaluation methodologies to determine the quality of various index recommendation systems [1]. To perform such evaluations, standard benchmarks are required. However, the defined generic benchmarks do not target specifically to validate the index recommendation solutions. Therefore, for each index recommendation solution, a special set of experiments is defined to verify the solution. There have been some efforts to design frameworks to benchmark index recommendation systems that are discussed in the rest of this section. The common aspect among all of them is that their database type is relational with a corresponding workload of *SQL* queries. Therefore, the evaluation of an index recommendation solution for document-based databases projects additional challenges.

The generic database benchmarks are designed to provide a way to execute the same set of tasks on different systems so that the performance results of these systems are comparable with each other [109]. Designing a fair benchmark is a delicate task. An inadequate benchmark can leave the door open for ”gaming the benchmark” [1, Chapter 12]. Each benchmark has three fundamental components: 1- *evaluation metric*, 2- *baseline definition*, 3- *the database/workload.* Each of these components must be carefully chosen according to the *System Under Test* (SUT) [110]. For the generic database benchmarks that are targeting the performance of the system as a whole the *evaluation metric* is usually a single quantity metric such as the *throughput* of the system. These benchmarks also determine the *baseline* and the *database/workload.* Then, their execution on several different systems allows comparing the results of their performances with each other [111].

There are many generic benchmarks defined to evaluate the performance of relational databases. The *Transaction Processing Performance Council* (TPC) [52] is a well-known benchmark defined by a consortium of vendors. TPC consists of a series of benchmarks that are corresponding to different well-established applications of relational databases. They even introduce modern benchmarks to cover more recent relational applications such as support systems and web commerce [112]. The performance of the whole system is evaluated then with the aid of a single metric.

Many index recommendation solutions utilized one or more TPC datasets and a variation of their workloads to validate the performance of their solutions [66], [79], [58], [68], [56], and [67]. These solutions, mostly, used a single value named *percentage improvement* to evaluate the quality of their index recommendation solutions. This number indicates how efficient costs of running the queries of the given workload are with the recommended set of indexes in comparison to the baseline set of indexes.

However, to assess the performance of specific database related systems such as an index recommendation system, more accurate evaluation metrics, and accurate baselines should be defined which are dedicated to index recommendation evaluation. Hence, each of the mentioned solutions to the index recommendation problem utilized a variation of one of the TPC benchmark and yet none of them used the same set of benchmarks. Consequently, though all of those solutions are developed for a relational database, their results are not directly comparable with each other.

There have been few efforts to design a common framework for the evaluation of index recommendation systems. Consens et al. in [111] suggest a framework to assess the effectiveness of the automated index recommendation systems that is known as *Toronto Autonomic Benchmark* (TAB). TAB defines its goal as enabling a comparison of the results of one or more index recommendation systems with each other that are running on the same database system, in contrast to the comparison of the results of these systems across multiple database systems. TAB introduces an *evaluation metric* that carries more detailed information about the quality of the index recommendation solution performance. This metric requires an input time and defines the quality of each particular set of recommended indexes as the number of queries in the workload that can be executed within this given time. This metric allows for a side-by-side comparison of the quality of the results of multiple index recommendation systems. It also supports performing a specific goal-oriented evaluation such as the execution of a fraction of queries in a sub-second time [113]. The *baseline* of TAB is proposed to be a set of all single-attribute indexes. For their *database/workloads*, they use both real and synthetic databases that can appropriately scale to the capacity of the available resources. For their realistic scenario, TAB utilizes the *Non-redundant REFerence protein* (NREF) database published by the *Protein Information REsource* [114] and for synthetic workloads a combination of TPC-H [115] and a skewed version of the TPC-H [116]. However, they only utilize retrieval queries and do not consider update queries.

N. Bruno in [113] had a critical look at the TAB benchmarks. The first concern is about the *evaluation metric* of TAB that works with the actual run-time of the queries. He argues that the execution of a query depends on many factors such as the query optimizer, query processor, and even the conditions of the underlying operating system. Therefore, such evaluation metric as introduced by TAB can be beneficent when dealing with assessing the full database system. Nevertheless, for a pure evaluation of the index recommendation systems, it is important to freeze any other external variables [117]. Thus, Bruno argues for utilizing the estimated execution cost by the query optimizer. Additionally, he claims another drawback of TAB's evaluation metric is that it only reveals information about the quality of the index recommendation solution for all of the queries of the workload rather than for each single query. Therefore, he recommends an evaluation metric that determines the quality of the recommended set of indexes in comparison to the baseline or any other set of indexes for every single query. He claims that this metric can be used as a complementary metric to TAB's metric to evaluate the index recommendation solutions. Regarding the *baseline* definition, Bruno also strongly argues against the usage of single-attribute indexes.

He introduces the design of a proper *database/workload* to evaluate the quality of index recommendation systems as an open research question. He explains that any beneficiary database/workload of a benchmark should be constructed from at least one of the following three 'buckets': Micro-benchmarks, Synthetic benchmarks or Real benchmarks.

Finally, Schnaitter and Polyzotis proposed a benchmark to assess the online index recommendation systems in [92]. Similar to the last two approaches, they also assumed that the data is stored in a relational database. Then, they exploit two *evaluation metrics* for their study. The first metric measures the total cost of the materialization of each set of indexes and the cost of running the queries of the workload under the current materialized set of indexes. Like the strategy proposed in [113], they also argue to utilize the estimated cost by the query optimizer instead of the actual running cost of the query. Therefore, their metric is not influenced by any mismatch between the statistics of the query optimizer and the real execution cost in the environment. As a complementary *evaluation metric*, they also capture the total wall clock time to execute the whole workload. For both of these metrics, they use the system that contains no indexes as their *baseline*. At last, they utilized the TPC-H, TPC-C, and TPC-E from the TPC benchmark suits and the NREF data set as their *database*. As of the *workload*, they use three different templates, which indicate various levels of workload complexity. Then, from each of these templates, the tables that are engaged in each query are selected randomly, with a selection probability proportional to the cardinality of the participating tuple. This approach to generating the workload is based on the common assumption that the query distribution pursues the data distribution. Certainly, an alternative approach to the random query usage is to benefit from the standard set of queries that are designed for each of the chosen datasets.

The common target of all of the above approaches is index recommendation solutions for relational databases. The evaluation of NoSQL database systems is even more challenging since there is not yet a generic standard benchmark designed to operate in both relation and all of the different types of NoSQL databases. The design of a uniform set of benchmarks to cover all database types requires a very careful and precise study. Due to the fundamental distinctions between data models and query languages of the NoSQL databases to each other and the traditional relational databases, the TPC benchmarks can not be directly used for other database types.

Even a proper mapping of the normalized data structure of the relational TPC benchmarks and their corresponding queries to the highly denormalized structures of NoSQL databases is a complex task [6]. Up until now, the efforts to map some of the TPC benchmark suits to document-based data model normally suffer from one of the following issues: they either do not consider the capability to replace linked data with nested documents [118], or they overuse the embedding ability [119]. Eventually, a proper translation of each query is required [120].

An attempt to provide a general framework that can produce data sets and workloads for many different database types is the development of the well-known Yahoo Cloud Serving Benchmark (YCSB) [121]. However, the common standardized *evaluation metric*, *baseline* and set of *database/workload* are still an open research question.

Therefore, the design of the methods to evaluate effectiveness of the AIRs proposed solutions required major attention. Sections 6.1 and 6.2 represent the proposed *evaluation metric*, *baseline* and the designed synthetic and real *database/workload* scenarios to evaluate the AIRs.

Table 3.1 lists a summary of some of the significant characteristics related to an index recommendation system for significant approaches to the index recommendation problem.

	[66] DB2 Advisor	*[79] Microsoft SQL Server 2005*	*[68] A Data Mining Approach*	*[56] An ILP Approach*	*[67] A Clustering Approach*	*[76] A Genetic Algorithm*	*AIRs*
Data Model	Relational	Relational	Relational	Relational	Relational	Relational	Document-based
Query Language	SQL	SQL	SQL	SQL	SQL	SQL	Procedural Query Language
Communicate with Query Optimizer	✓	✓	—	✓	✓	—	✓
Heuristic to Extract Queries	—	—	Frequent Itemsets Output	—	K-Means Clustering Output	—	Frequent-Long & Merged Output
Merging	✓	—	—	—	—	—	✓
Reduction of Search Space	SAEFIS & BFI Algo.	Atomic Configs	Frequent Queries	Atomic Configs	Clustering Similar Queries	Atomic Configs	Atomic Configs &Frequent-Long Alg.
Enumeration Technique	Knapsack Formulation & Greedy Search	Greedy Search	Greedy Search	Integer Linear Programming	—	A Genetic Algorithm	Integer Linear Programming
Garantee Optimal Solution	—	—	—	✓	—	—	✓
Reduction of Load on the Optimizer	Single Call Algorithm	What-If Optimizer	—	INUM	—	—	Virtual Env. Based on Sampling
Evaluation Scenario	Versions of TPC-D suits	Versions of TPC-D suits	Versions of TPC-H and TPC-D suits	TPC-H suit	TPC-R suit	Randomly Generated ISPs by [117]	Mix of Real and Synthetic Generated by NoWog

Table 3.1: Comparison between well-known solutions to the index recommendation problem.

Chapter 4

The Adaptive Index Recommendation System

Besides the theoretical complexities of index recommendation, there are many applied and practical issues that should be considered when designing the architecture of a unified system. This chapter is devoted to the discussion of such practical issues as well as the explanation of heuristics used to design solutions in AIRs.

At first, the criteria that are considered to design the AIRs are presented in Section 4.1.

As presented in Problem 2.3.2, one major concern in the design of an index recommendation system is to reduce the search space of the candidate indexes. Section 4.3, in addition to presenting the architecture of AIRs declares the heuristics that are taken into account to reduce the search space.

Another major practical design concern is to reduce the overhead of calls to the query optimizer as stated in Problem 2.3.4. The designed solution of AIRs to address this problem is discussed in Section 4.2.

Materialization of indexes on the original datasets is too resource-consuming and the size of each candidate index can not be directly measured. Therefore, the size of indexes should be estimated from the number of documents containing the corresponding attributes that construct the index. As discussed in Section 4.4, this estimation can be done based on the type of the value that the attribute contains.

Some of the ideas presented in this chapter are published in [7], [61], and [122].

4.1 Identification of Relevant Criteria

Based on the task defined for an index recommendation system in Section 2.3 and the characteristics and requirements of each of its sub-challenges, the parameters that should be considered in the design of an index recommendation system can be extracted. This section is dedicated to explaining the extracted parameters. A concise description of where and how these parameters are taken into account in the unified design of the AIRs is also presented.

The design of an index recommendation system is dependent on three general conditions of the targeted database: 1) the initial state of the database at the time of activating the recommendation system, 2) the capacity of the environment where the database instance is running within, and 3) the type of incoming workload to the database. The exact relevant parameters in the design of the adaptive index recommendation system are chosen regarding each of these conditions. Table 4.1 represents a summary of all of these relevant parameters in correspondence to each general condition.

The logic behind consideration of each of these parameters are discussed in the following:

1. **Initial State of the Database**: the AIRs can only recommend indexes based on its analysis of the database system at the time that AIRs is triggered. The state of the database at this time is known as the *initial state* of the database. This initial state can be defined by the following parameters:

    - ***Database Type***: the type of a database is defined by its data model, query language, and the query optimizer behavior. The differences in the behavior of the query optimizer and the query language of various databases have a direct impact on the syntactic analysis of the queries to extract candidate indexes. The required distinctive syntactic analysis is discussed in Section 4.3.
    - ***Cardinality of the Database Attributes***: the cardinality of attributes in the contemplated datasets determine the selectivity of the chosen indexes. A designed strategy of AIRs to reduce the overhead of calling to the query optimizer is to work with a sample of the dataset. The query optimizer chooses between different indexes based on their selectivity that is extracted from the statistics of the dataset.

      Therefore it is important to extract a representative sample of the dataset, especially in terms of cardinality of the attributes. Hence, the cardinality parameter impacts the practical design of the AIRs as debated in Section 4.2.
    - ***Already Existing Indexes***: the targeted database at its initial state might already contain some indexes. The execution of the analyzed workload with the recommended indexes proposed by AIRs should not be more expensive than its execution with the set of already existing indexes. This aspect is considered in Section 4.2 where an algorithm is developed for the materialization of the set of indexes (Algorithm 4.1).

2. **Database Environment**: the capacity of the environment where the targeted database is running determines a limitation on the available resources. These resources consist of available storage, processing units, etc. However, for parametrization of an index recommendation system design, it is important to consider the storage restrictions. By strongly advising to run the AIRs on a separated system as the original database, the processing power of the environment becomes irrelevant as a parameter in the study.

- ***Storage Limitation***: to be able to materialize the proposed indexes, their required storage capacity should not exceed the available storage threshold. Also, to benefit fully from the advantages of having an index in the system, the index should fit into the memory.

  Therefore the available free storage capacity of the database environment implies an upper limit for the total size of all recommended indexes. This matter is introduced as a constraint for the objective function presented in Section 5.1. Additionally, since materializing indexes is an expensive process, the size of the candidate indexes should be estimated. The related study to estimate the size of each index is covered in Section 4.4.

3. **Type of Input Workload**: the creation of indexes that are not beneficiary for any query occupy unnecessary storage capacity. Therefore, it is important to create the indexes with regard to the characteristics of the queries in the workload.

   These characteristics can be described by the following parameters:

   - ***Read to write Ratio***: the existence of relevant indexes is purely beneficiary for read queries. However, any time an update, insert or a delete query is issued, all relevant indexes should be updated. Therefore, the more write operations in the workload, the more maintenance costs for indexes. Therefore, the calculation of the benefit and maintenance cost of each index with respect to the ratio of read to write operations in the workload is an important parameter in the design of an index recommendation system. The design of the objective function in Section 5.1 is targeted to cover this issue.
   - ***Frequency of Individual Queries***: the importance of the queries in a workload is not equal to each other. The more frequent a query is issued, the more important that query is in the workload.

     Therefore, a weighting strategy based on the frequency of the queries is developed in Section 4.3.1.
   - ***Queries with Long Run-Time***: weighting queries merely based on their frequency can result in the elimination of queries that can not be issued more frequently due to their very long run-time. In fact, these are the queries that need indexes the most.

     Therefore, as discussed in Section 4.3 the queries with long run-time should also be treated specially by the design of the index recommendation system.

In this section, the effective parameters deliberated to design the index recommendation system are introduced. These parameters are placed in various segments of the recommendation system. Next, the integrated architecture design of AIRs that is constructed to cover all these criteria is presented.

General Relevant Conditions	***Relevant Parameters***
Initial State of the Database	• Database Type • Cardinality of the Dataset Attributes • Already Existing Indexes
Database Environment	• Storage Limitation
Type of Input Workload	• Read to Write Ratio • Frequency of Individual Queries • Queries with Long Run-Time

Table 4.1: General considerations and parameters that effect the design of the Adaptive Index Recommendation system (AIRs).

4.2 The Architecture Design

The architecture of the *Adaptive Index Recommendation System* (AIRs) has a modular design which is illustrated in Figure 4.1. The modules of the AIRs are segmented into two sections. On the left-hand side, the module and components run regularly to monitor changes in the database workload and data set. A profile of the incoming workload to each dataset and a sample of the dataset are recorded by these components. The modules on the right-hand side in Figure 4.1 run when the recommendation process is triggered. After the evaluation of the workload and its corresponding dataset, these modules recommend a configuration as the set of optimal indexes for the targeted database. AIRs is connected to the targeted in-production database. The recommended configuration can either be created directly on the in-production system or be suggested to the user. This section is dedicated to explaining practical considerations for the design of such an adaptive index recommendation system.

Although running the AIRs on the same system as the in-production (target) database is possible, it is not recommended. Hosting AIRs on that same infrastructure means that

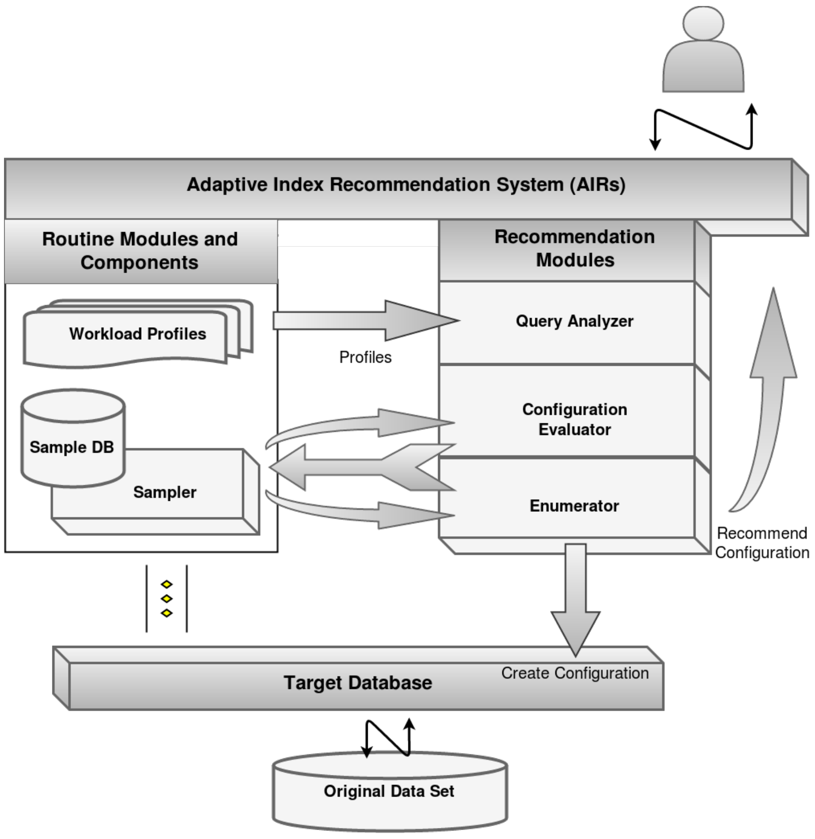


Figure 4.1: The Architecture of the Adaptive Index Recommendation System (AIRs).

it competes for the same valuable resources with the target database and possibly other applications. Since AIRs connects to the target database only to extract the *Workload Profiles* and to update its *samples*, the trade-off between transferring just small samples of data and workload profiles to a remote machine and preserving resources for the original database pays off.

The AIRs modules and components are introduced in the following under their particular categories:

1) Routine Modules and Components: these segments gather information from the *Original Data Set* on a regular basis as shown in Figure 4.1. They operate either on periodic cycles or after receiving a triggering signal by the user.

- ***Workload Profiles Component:*** this component contains profiles of all of the data manipulation queries (search, update, insert, and delete) issued to the targeted database *collection*. The corresponding run-time timestamps of each query is also recorded. A selection of these queries in a specific time interval for a particular collection can shape the workload. This workload is used as input to the *Query Analyzer* module, once the recommendation process on the targeted collection is triggered.

- ***Sample DB Component:*** this component is actually a similar database instance as the in-production one (specified as *Target Database* in Figure 4.1). It stores the *samples* of the *Original Data Sets*. Additionally, the query optimizer of this database instance is used to evaluate the costs of running each query with different configurations. Therefore, it is important that the sample database has the same query optimizer version as the in-production database. If AIRs runs on the same system as the target database, the original database instance itself would be used as this component. This case is not recommended because of the occupation of available storage to the database by AIRs and the extra load that evaluations of AIRs put on the target database. In either case, the evaluation process of AIRs on the sample is carefully designed in way that it does not alter the incoming workload records or the actual data set of the targeted database.

- ***Sampler Module:*** this module extracts a sample of each of the targeted database collections and stores them into the *Sample DB* component. The queries to extract the representative sample are designed in a way that they do not influence the workload of the targeted collections in the future runs of index recommendation process. This module can be triggered at the same time as the *Recommendation Modules* which increases the run-time duration of the AIRs. However, this module can be set to run regularly. Based on the size of the sample and the number of write operations to the corresponding collection, the related samples should be refreshed. Many practical heuristics are utilized in the design of this module that are discussed in more details later in this section.

2) Recommendation Modules: these modules run in sequence only if the recommendation process is triggered. The series of these modules extracts the *workload* for the selected database collection in a particular time interval. As shown in Figure 4.2, the output of this series is the recommended index configuration.

- ***Query Analyzer:*** this module is responsible for the analysis of workload queries and the extraction of the most relevant attribute sets from them to construct candidate indexes. The output of this module is the union set of all candidate indexes for each relevant query (*Enumeration Space*). This set is passed as input to the next module in this series. The Query Analyzer module consists of three internal sub-modules as: 1) *Cleanser*, 2) *Miner*, and 3) *Merger*. The extraction of the candidate indexes from relevant queries plays a key role in the reduction of the enumeration space and thus for the scalability of the whole system. The approaches taken in each of sub-modules of the *Query Analyzer* module are discussed in detail in Section 4.3.

- ***Configuration Evaluator:*** the goal of this module is to efficiently evaluate combinations of indexes from the enumeration space and to derive so-called *Atomic Configurations*. For this purpose, the *Index Benefit Graph* (IBG) method (see Section 5.2) is implemented in this module. Also, the storage costs of each index configuration is estimated based on the estimation techniques described in Section 4.4. The storage cost along with scan and update cost of each extracted atomic configuration are passed to the *Enumerator* module.

- ***Enumerator:*** this module contains the implementation of the objective function within the proper enumeration technique. The objective function implemented in this module is the one described in Section 5.1.2. As discussed in that section, the enumeration technique used in this module is the Branch-and-Bound in Integer Linear Programming. The output of this module is the *Recommended Configuration.* If the algorithm runs to its completion, then the recommended set of indexes is the optimal solution, i.e. the global maximum of the objective function. However, if the running process is interrupted before the optimal solution is obtained, this module returns a sub-optimal configuration with its quality gap as later defined in Equation 5.10d.

As the enumeration process is finished, the recommended configuration can be created on the targeted database. There are more factors to be considered before materializing the recommended configuration. This process is described in Algorithm 4.1. If the targeted database does not have any indexes, the index recommendation process is completed. However, it often happens that the targeted collection already has a set of indexes which form an *Existing Configuration* C_E. In that case, the quality of the *Recommended Configuration*

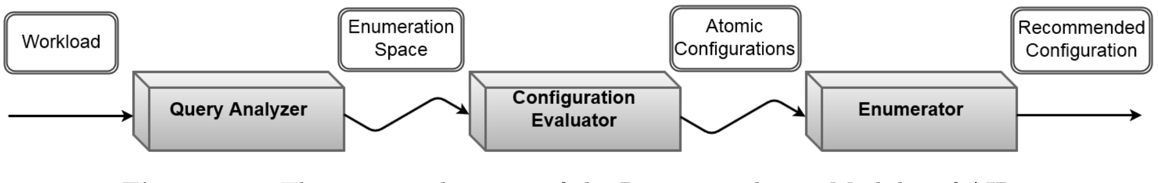


Figure 4.2: The input and output of the Recommendation Modules of AIRs.

C_R should be compared to C_E. This quality is compared to the benefit of each of these configurations for the whole workload. Let the benefit of C_R be Z_R and the benefit of C_E for the workkload be Z_E. Only if Z_R is greater than Z_E by at least the *threshold* τ, the new configuration should replace the existing one. Introducing this threshold ensures that the benefit gained by creating the new configuration is worth the *materialization cost* of configurations (creation of new and removal of old indexes). To minimize the cost of index materialization, only the indexes in C_E that are not common between already existing and recommended configurations are dropped and only the none-common indexes of C_R would be created.

Algorithm 4.1 Configuration_Materialization(C_R, C_E)

```
$Z_R \leftarrow$ evaluate_configuration($C_R$)
$Z_E \leftarrow$ evaluate_configuration($C_E$)
if $Z_R - Z_E > \tau$ then
    common $\leftarrow C_R \cup C_E$
    drop_indexes($C_E$ - common)
    create_indexes($C_R$ - common)
end
```

To enable the query optimizer to evaluate execution plans with an arbitrary set of indexes from the *enumeration space*, those indexes should be materialized on the dataset. Part of the index materialization process is also to obtain and create related statistics of that index. These statistics contain various information such as the cardinality of the predicates that contain the specified key attributes in the index. These statistics are added to a catalogue utilized by the query optimizer. Many database store these statistics in form of histograms [1, Chapter 2]. Since the creation process of each index contains structuring the values of the corresponding key attributes in a sorted order, the histogram creation procedure can be carried out simultaneously. Histograms represent the data distribution. The cardinality of each query predicate can be calculated from these histograms. Therefore, by materializing each index, the query optimizer is given a better estimation of the certain predicates in queries. Consecutively, these statistics can led to a choice of a different execution plan by the query optimizer. A common concern in utilization of one-dimensional histograms in relational databases, especially in conjunction with SQL-based query languages, is that they do not return proper results for *join* queries [123]. However, they perform excellent in response to range queries. Since document-based databases mostly eliminate *join* queries, they can easily provide the necessary statistics with one-dimensional histograms.

The cost evaluation process of arbitrary index sets requires those indexes to be created and be removed after the evaluation is done. However, materialization of all candidate indexes on the original datasets is both time- and resource-consuming. If the database is capable of generating an execution plan with applying some filters that do not consider existence of some of the indexes, all of the indexes in the *enumeration space* can only be created and removed once. Without filtering capability of the database, the indexes should be created and removed more often for each configuration. Nevertheless, even one time creation and deletion of all of the indexes in the *enumeration space* can be a heavy load on the in-production system, especially if it contain many documents. Additionally, the evaluation process contains many calls to the query optimizer to return the cost of running each query with various index sets. These queries produce additional queries load on the database to the normal application queries. Therefore, for a database with heavy workload, this evaluation process can affect the performance of the database negatively.

A solution to this problem is to apply the index materialization and cost evaluation procedure for the index recommendation to a virtual environment, instead of the in-production system. This virtual environment consists of representative samples of the original targeted datasets. The *Sampler* module of Figure 4.1 is responsible to collect the representative samples and store them in the *Sample DB* component.

On the one hand, the idea of utilizing a virtual environment is similar to the basic idea of the well-known *What-if* component (see Chapter 3). The difference is that in the *What-if* environment, instead of creating the indexes the related histograms of those indexes are injected to the catalogue of relevant statistics for the query optimizer. Therefore, instead of actually materializing the indexes, their hypothetical existence is considered by the query optimizer. For the creation of these hypothetical indexes, still their relevant statistics should be extracted to build the histograms.

On the other hand, the idea of utilizing a representative sample of the original collection as a virtual environment is similar to the idea of the construction of histograms with the help of samples. This idea is practiced by many databases to construct their histograms. Histograms are an approximation of the data distribution of the values in each attribute. Hence, a histogram created for a uniform random sample of the original collection returns relatively similar results as the one constructed for the original collection [1, Chapter 5]. One concern regarding the construction of a sample is how large the size of the sample should be. Chaudhuri et al. in [96] proved a sample between 1 % to 10 % of the original dataset size results in histograms similar to the ones on the original collection.

Therefore, a uniform sample can be extracted from the original dataset. For the purpose of reducing the amount of data transformation, AIRs constructs a representative sample with 5 % of the number of documents in the original collection. To extract the representative sample of each targeted collection, this module performs a full collection scan on it. Like any other query, these queries are also stored in the related *Workload Profile* of that collection. However, the full collection scan queries are eliminated by the *Query Analyzer* module in the process of extracting relevant queries. Therefore, the sampling procedure does not influence the workloads of the targeted database and the results of the future runs of the index recommendation process.

4.3 Exploration of Candidate Indexes

The number of all possible indexes for a data set typically makes a too large search space to be exploited by the enumeration techniques. Utilizing adequate heuristics assist in reducing the size of the search space. In this section, the heuristics utilized in AIRs, which consists of a syntactical strategy and considerations of the most frequent queries and merging indexes, are discussed.

The number of all possible indexes on a data set collection is dependent on the number of unique attributes of the collection and available *index types* (e.g. ascending, descending, geospatial, text, etc.) in the particular database instance. Let t be the maximum number of unique attributes in documents of the targeted collection. Also assume that θ is the number of index types that the database instance supports. Therefore, the number of all possible indexes N_{idx} on that collection can be obtained as:

$$N_{idx} = \sum_{\gamma=1}^{t} \frac{\theta^{\gamma} t!}{(t-\gamma)!}. \tag{4.1}$$

There are t possible choices of indexes fro the first field of the document. Coordinately, $(t-1)$ choices remain for the second field. By adding more fields to the combination, the total number of possible indexes on those fields grows as $t(t-1)(t-2)...(t-\gamma+1)$ that is equivalent to $\frac{t!}{(t-\gamma)!}$.

The number in Equation 4.1 grows exponentially with the number of attributes in the dataset. This growth not only produces an enormous search space for the enumeration technique but also occupies a lot of storage space. Additionally, having all of these indexes in place not only might not be useful, but they can potentially be even harmful due to their necessary maintenance cost in the case of many *write* operations to the system.

The goal is to reduce this number by only considering indexes on fields that appear in queries of the workload. This is achieved by syntactical analysis of the corresponding workload. The input to the *Query Analyzer* is the extracted *Workload* in the specified time interval. This input is first passed to the *Cleanser* module, as shown in Figure 4.3. This module is responsible for performing a syntactic analysis of the query string and extracting the *indexable* attributes which are defined in Definition 4.3.1. This extraction is based on a set of rules described in Section 2.4.3 that the query optimizer applies to selecting the proper indexes.

Definition 4.3.1. ***Indexable***:
The *indexable* attributes are the ones specified in the search-predicate and the sort-clause of any query.

The attributes in the replacement-predicate of write queries do not benefit from having indexes (see the procedural language structure in Section 2.4.2).

After parsing each query, the *Cleanser* module obtains all attributes with the *equality* and *range* predicates from the *parse tree* of the *query processor*. For each query with a single attribute in the search-predicate, a set containing that attribute is added to the

Repeat Att List output of the *Cleanser*. If the search-predicate of the query contains more than one attribute, the structure of the obtained set of attributes depends on the conjunctive operator associated with that part. For a query with an *AND* conjunctive operator, a list of attributes including all attributes in the search-predicate is added to the output set. Whereas for a query with an *OR* conjunctive operator, *index intersection* is used. As a result, two separate lists containing each attribute in the predicate is added to the output set.

Example 4.3.1. Assume the search-predicate is as the following: $(A_1 > 10 \wedge A_2 = 20)$. In this case, the obtained set of attributes is $\{\{A_1,\ A_2\}\}$. However, for a query with an "OR" conjunctive operator like:$(A_2 = 200 \vee A_3 \leq 12)$, two separated list as $\{\{A_2\}, \{A_3\}\}$ each containing one of the attributes are added to the *Repeat Att List*. The resulting *Repeat Att List* of these two queries is $\{\{A_1.A_2\}, \{A_2\}, \{A3\}\}$.

Many heuristics can be used to reduce this space. The *Query Analyzer* module contains the designed heuristics for AIRs. These strategies include defining a fraction of queries in the workload as *relevant queries* that are:

- repeated more frequently during the requested time interval,
- are taking too much time (more than a threshold) to receive their answers

during the requested time interval. The idea of only considering frequent and long queries is named as the *Frequent-Long* strategy.

The *Frequency Miner* in Figure 4.3 exhibits the sub-modules of *Query Analyzer* where the first strategy is implemented. The logic behind this module is explained later in this section. The second strategy is implemented within the *Cleanser* module itself. This module

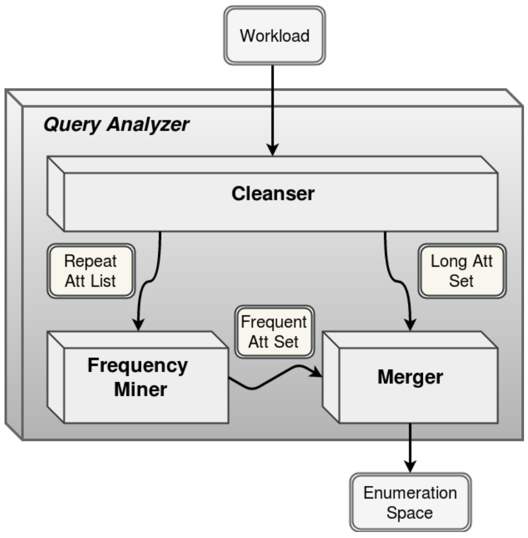


Figure 4.3: The input and output for sub-modules of the AIRs Query Analyzer module.

produces two sets of outputs: 1) *Repeat Att List* that is passed as input to the *Frequency Miner* and 2) *Long Att Set* that is passed to the *Merger* module. Whereas the first one includes the set of indexable attributes from all queries, the second one contains the same information only from *long queries.*

Definition 4.3.2. ***Long Query***:
Any query q_i with run-time above a threshold μ is known as a *long query.*

Since a query that takes too long to retrieve its answer would not be issued that frequently, it is necessary to gather the *Long Att Set* in addition to the *Frequent att Set.* Whereas such queries are in urgent need to be indexed, the strategy of most repeated attribute sets in *Frequency Miner* eliminate the corresponding list of attributes for these queries. Passing the *Long Att Set* the module after *Frequency Miner* ensures that these queries get a fair chance of being indexed.

In the rest of this section, the most frequency strategy and the merging of indexes to cover less favorable indexes with better compromising benefit are explained respectively.

4.3.1 Frequency Query Strategy

After extracting the list of indexable attributes of each query, the candidate set of indexes can be defined based on different methods. AIRs builds its initial candidate set based on a heuristic of most frequency queries to the system. The assumption is that rarely issued queries to the system are less likely to be issued again. Therefore, the candidate index set can be built without considering the attribute set of such occasional queries.

The *Frequency Miner* module of the *Query Analyzer*, shown in Figure 4.3, contains the implementation of this logic. To construct this logic, AIRs benefits from the *Frequent Itemset* algorithm [124].

Two basic elements of this algorithm are *items* and *itemsets.* In this specific use-case, each query in the workload forms an *itemset* and any of its individual attributes is an *item* of that itemset. Therefore, the workload W is the list of all of the itemsets. The level of significance of each subset of items can be measured with its *support*, which is defined in Definition 4.3.3. Based on the associated support of each itemset, the *frequent itemsets* can be signified based on the Definition 4.3.4.

Definition 4.3.3. ***Support***:
Let X be a subset of items and W be the list of all itemsets with size n. Then, the *support* s in association with subset X is defined as the portion of itemsets ω that contains subset X:

$$s(X) = \frac{|\omega \in W; X \subseteq \omega|}{n}. \tag{4.2}$$

Definition 4.3.4. ***Frequent itemset***:
Given a number ξ as the support threshold, a subset M of items is known as *frequent*, if its associated support s is greater than or equal the support threshold: $s \geq \xi$.

The advantage of using *frequent pattern mining* is that it extracts all the single and multi-attribute frequent patterns as the potential relevant indexes. The multi-attribute patterns are only selected if all of their single attributes are above the specified threshold ξ. Therefore, the risk of eliminating valuable single-attribute indexes is prevented.

This algorithm receives the *Repeat Att List* which contains lists of indexable attributes of each query. The output of this algorithm is a set of key-value pairs with the frequent attribute patterns as key and their frequency number as value. By sorting this list in descending order of its values, it represents a prioritized set of most frequent attribute patterns. This output which is shown as *Frequent Att Set* in Figure 4.3 is passed to the *Merger* module. The functionality of the *Merger* module is discussed in Section 4.3.2.

4.3.2 Index Transformation through Merging

The first step in the *Merger* module is to unify its two input sets as: *Frequent Att Set* $\cup$ *Long Att Set*. The *merging* logic of this module is performed on this union set.

The implemented logic in the *Merger* module is based on the idea that query optimizers are capable of combining indexes in the form of intersection and union of two indexes to answer queries. However, constructing these transformed shapes of indexes on the fly by the query optimizer rises the costs of processing the query. Therefore, if there are many queries in the workload that can benefit from the transformed shape of indexes, it is worth adding these *transformed indexes* to the candidate set of indexes and evaluate their benefit rather. The *transformed indexes* set is constructed from the union of the two input sets of the *Merger* module. The transformed shape considered in AIRs is the *merge* of two indexes with similar leading keys as defined in Definition 4.3.5 and 4.3.6.

Definition 4.3.5. ***Index Merge***:
Assume index I_1 consists of the leading attribute A_1 and a set of following attributes denoted as R_1, that can be shown as $I_1 = S(A_1|R_1)$. In the same way index $I_2 = S(A_2|R_2)$. Then, the merge of I_1 and I_2 is defined as an index with the leading attribute of I_1 as its leading key and a union of all attributes in I_2 with following attributes of I_1. The merge of two indexes I_1 and I_2 is denoted as $I_1 \oplus I_2 = S(A_1|(R_1 \cup A_2 \cup R_2) - A_1)$.

A merge of two indexes results in an index that is usually smaller than the sum of two original indexes while it maintains many of their qualifications to answer queries. Index merging is not commutative in the order of indexes, i.e. $I_1 \oplus I_2 \neq I_2 \oplus I_1$ because $I_2 \oplus I_1 = S(A_2|(R_2 \cup A_1 \cup R_1) - A_2)$.

An execution plan for a query constructed by the two original indexes has usually smaller cost than the one with the merged index. Therefore, when the two original indexes are present, the merged index most probably is not the first choice of the query optimizer to run a query. However, in the case of limited storage capacity, the overall benefit of keeping the merged index might exceed the benefit of the two individual indexes in the optimization process.

Hence, it is possible to merge all indexes in the candidate set of indexes with each other. However, for a candidate set with n indexes, adding the merged form of all indexes results in a factor of $n(n-1)$ growth in the size of this set. This large number also projects a lot of overhead to evaluate the cost of these indexes.

Therefore, as an optimization in the heuristic of AIRs, only merging of indexes that share a leading attribute key is considered which is defined in Definition 4.3.6. The algorithm of this heuristic that is implemented in the *Merger* is described in Algorithm 4.2.

Definition 4.3.6. ***Index Merge with Leading Keys***:
Given $I_1 = S(A_1|R_1)$ and $I_2 = S(A_1|R_2)$, the merge of indexes with shared leading attribute key follows from Definition 4.3.5 as: $I_1 \oplus I_2 = S(A_1|(R_1 \cup R_2) - A_1)$.

Algorithm 4.2 Merge_Indexes(E)

```
Result: The enumeration space with merged indexes.
foreach $I_i, I_j \in E, i \neq j$ do
    $A_i \leftarrow$ first_att($I_i$)
    $A_j \leftarrow$ first_att($I_j$)
    if $A_i == A_j$ then
        $E \leftarrow E \cup$ merge($I_i$, $I_j$)
    end
end
return $E$
```

As a result, the *merge* function in the Algorithm 4.2 produces all of the possible combinations out of merging the leading attributes of the input indexes I_i and I_j. The output of this algorithm is a set of all single and multi-attribute candidate indexes for frequent and long queries, as well as merged indexes of them. This output builds the *Enumeration Space*, shown in Figure 4.3, which is the general output of the *Query Analyzer* module.

As summary, in this section the architecture design of the *Adaptive Index Recommendation System* (AIRs) is described. The practical difficulties of extracting proper candidate indexes and the heuristics of AIRs in this regard is explained. These heuristics include considering most frequent queries and the ones with long run-time responses as relevant queries. The candidate indexes are extracted from these queries. By merging the candidate indexes that share the first leading attribute with each other and adding them to the set of candidate indexes, the enumeration space of the index recommendation system is prepared.

4.4 Storage Estimation

For the optimization of recommended indexes the storage constraint should be taken into account. Thus, the required storage for each index should be given. Since the indexes of each configuration are built only on the sample data set, the size of each index on the original data set should be estimated. There are at least two approaches to obtain this estimation: based on a thorough study of the index sizes and different index combinations, or based on theoretical estimate of size of all indexes. In the following of this section, the pros and cons of each of these methods are discussed. The section is concluded by a description of the chosen method to estimate the size in my approach.

Typically, the database generated metadata, data, and indexes are stored in *Data Files*. Each Data File is build up of multiple logical containers named *Extents* to store data and indexes. Databases use several optimized methods to store data, one of which is to reserve extra storage space for future data addition. This additional space is influenced by configurations of the database instance and the environment where the instance is running on, such as the filesystem block size.

There are different measures related to the storage consumption of data and indexes. In general, most of the measures in current databases include an additional storage space. The other measure might include the size of all data extents in the database. This measure is larger than the first one because it contains the vacated space by deleted or moved data and space yet to be used.

Table 4.2 shows the essence of the problem when we deal with obtaining the index size from the sample. This table presents the index size of three indexes. The first one is structured on an attribute containing single integer values and the second on arrays of integer values. The third one is a compound index containing these two attributes. The original index sizes of these attributes differ from each other. However, the size of these indexes on a sample of twenty documents is the same. Databases usually allocate more storage than initially needed for data for more efficient memory management. As a result,

Existing Indexes	I_1 ***Single***	I_2 ***Single***	$[I_1, I_2]$ ***Compound***
Value Type	Int on single value	Int on array of values	Int-Int
No. Documents in Original set	40,000	40,000	40,000
Original Index Size (Bytes)	233,472	1,130,496	1,679,360
No. Documents in Sample set	20	20	20
Sample Index Size (Bytes)	16,384	16,384	16,384

Table 4.2: Comparison of index storage consumption on original data set and a sample set for attributes with single and array of values. The original set of documents are created by NoWog. The original set for the case of single index on I_1 contains only one attribute with integer value. For the single index on I_2 the dataset contains only one attribute with array value of average size 10 integer elements. Only in case of the compound index, the data set contains two attributes which each of them has integer as their values. The results of this simple comparison show the practical optimizations implemented in the database design to preallocate blocks of storage. Despite the obvious dependency of the index size to the type of values for indexes built on the original dataset, the size of the indexes on the sample are the same. So, a direct estimation of the original index size from its sample size is not possible.

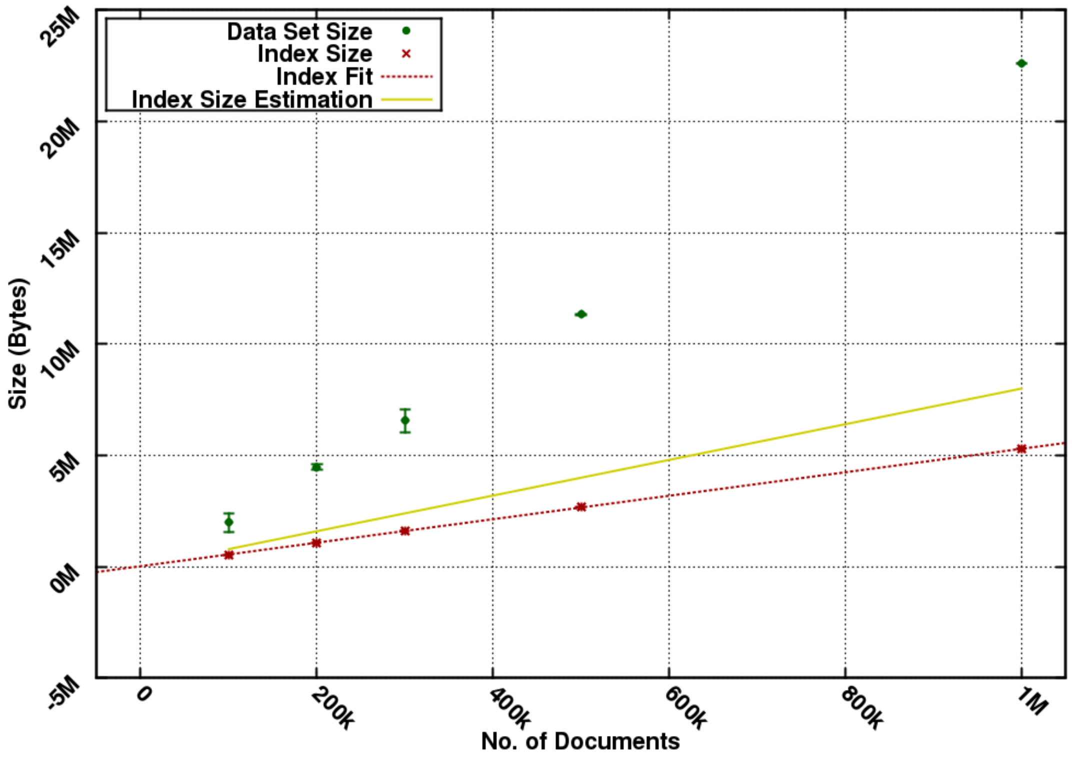


Figure 4.4: Index size growth pattern for attributes with ***Integer*** value types to the number of documents in the dataset. The theoretical estimation is obtained with the assumption o f $\tau_{int} = 8\ Bytes$.

for a sample with a small number of documents, the size of all of these indexes seem to be the same. Therefore, it is not possible to deduce the original index sizes from information provided by the sample. Thus, to pass a legitimate size for the constraint calculation in Chapter 5, the size of original indexes should be estimated.

The first estimation model is based on the assumption that there is an approximately linear dependency of the index size to the number of documents. The parametrization of the linear curves needs to be done for each index type. Once a set of parametrizations is obtained from various test databases, the size of a new index can be estimated just from the number of documents and types of attributes to be indexed.

Figures 4.4, 4.5, 4.6, and 4.7 depict the index size with respect to the growth in size of documents for various individual types of values. Each these figures shows the trend of the index growth for syntactical datasets (generated by NoWog discussed in Section 6.1) which each consists of only a single attribute. The type of the value differs in each figure as integer, boolean, nested documents and array. Values are generated randomly. Therefore, the generation of the dataset is repeated ten times. The error bars of the *Data Set Size* markers that are indicated with green markers are produced by calculation of the standard

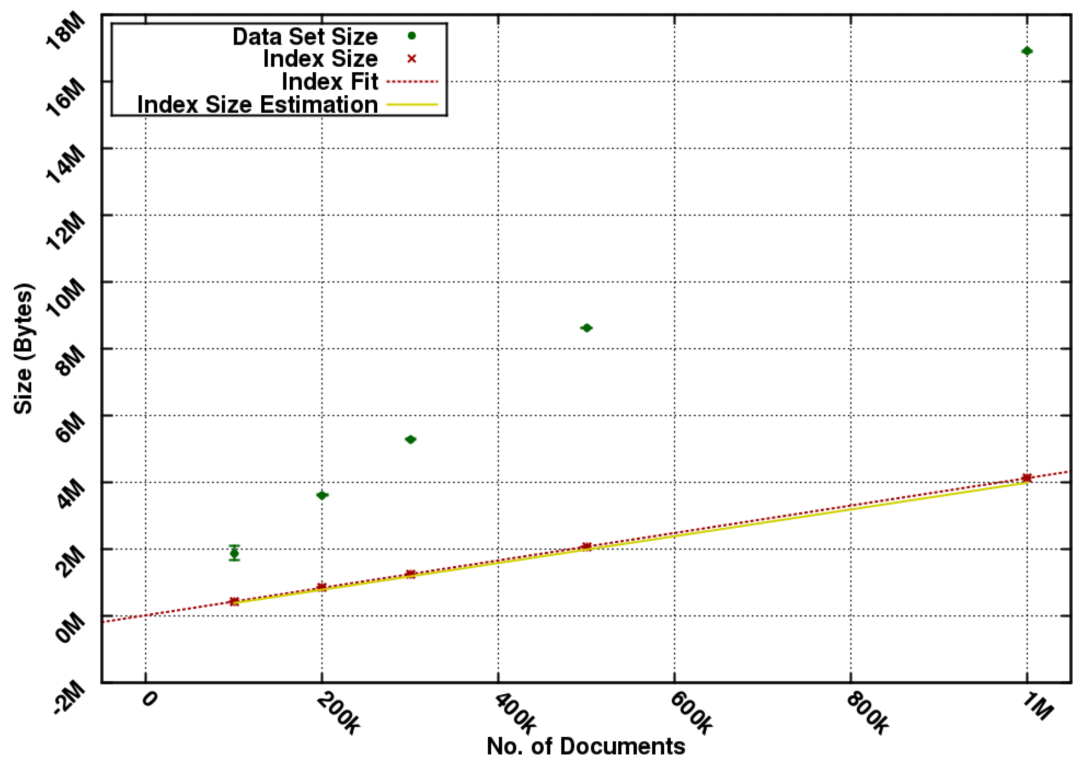


Figure 4.5: Index size growth pattern for attributes with ***Boolean*** value types to the number of documents in the dataset. The theoretical estimation is obtained with the assumption of a block size $\tau_{int} = 4\ Bytes$.

deviation of ten dataset generation in each figure. The actual *Index Size* on each of these unique attributes are depicted with red crosses in each figure. The *Index Fit* shows a linear fitting line for the growth of the index size. Since a negative intercept for the size of index and data set does not have any logical interpretation, the fitting formula is defined as the following:

$$f(x) = m * x + c^2. \tag{4.3}$$

Therefore, the fitting line is forced to have a non-negative intercept.

Since the slope of this fitting line for each of the attribute types is a fixed number, the information of this line can be used to estimate the size of the index merely based on the number of documents in the original dataset. For example, the Figure 4.4 shows the trend of index growth for a data set that contains one attribute with random integer values. Since the fitting line passed through all of the markers and the error bars are negligible, it shows that the index size growth has a good linear dependency to the number of documents that contain the corresponding attribute. Therefore the corresponding slope and intercept can be used to estimate the size of single attribute indexes.

However, this approach is tightly coupled to the database instance and its particular

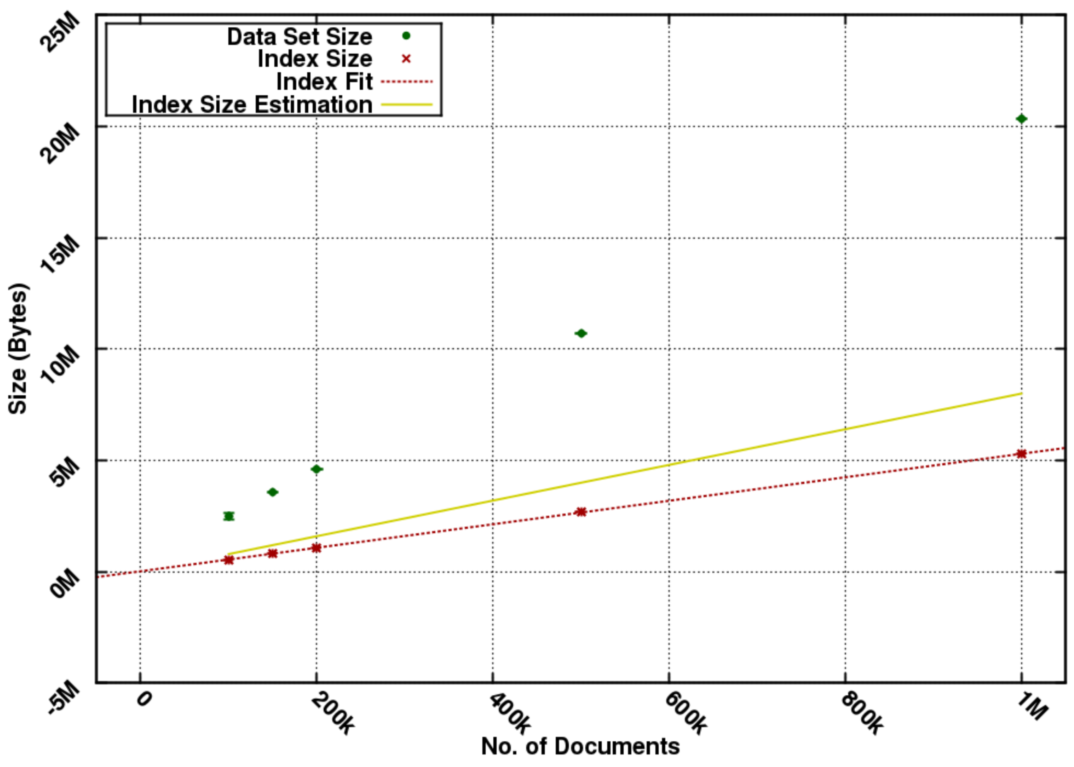


Figure 4.6: Index size growth pattern for attributes to the number of documents in the dataset. Each attribute contains three layers of ***Nested Documents*** which at deepest layer includes one attribute with integer value type. The theoretical estimation is calculated based only on assuming a block size of $\tau_{int} = 8\ Bytes$ for an integer value.

configurations and optimization methods. Therefore, the second approach is defined based on theoretical knowledge of the required size to store indexes [125] which is indicated by the yellow *Index Size Estimation* line in each of the Figures 4.4, 4.5, 4.6, and 4.7.

If a set of documents that all have a particular attribute contains n members, the required size for storing the index can be estimated by

$$S_{single} = (\tau_{type} * n) \tag{4.4}$$

where τ_{type} is the arbitrary block size that can be associated with each type of value. For example, for an integer value in Figure 4.4, it is defined as $\tau_{int} = 8\ Bytes$. Whereas Figure 4.5 with boolean type contains $\tau_{bool} = 4\ Bytes$. As shown in Figures 4.4, 4.5, 4.6, and 4.7, this calculation provides a theoretical estimation for the index size.

Usage of a theoretical estimation has the advantage of being less dependent on hardware- and implementation-specific properties. However, the disadvantage of using a theoretical estimation is that if the distance between the estimated and the actual index size is large, the cost of index storage consumption will be overestimated. As a result, a necessary index might not be created.

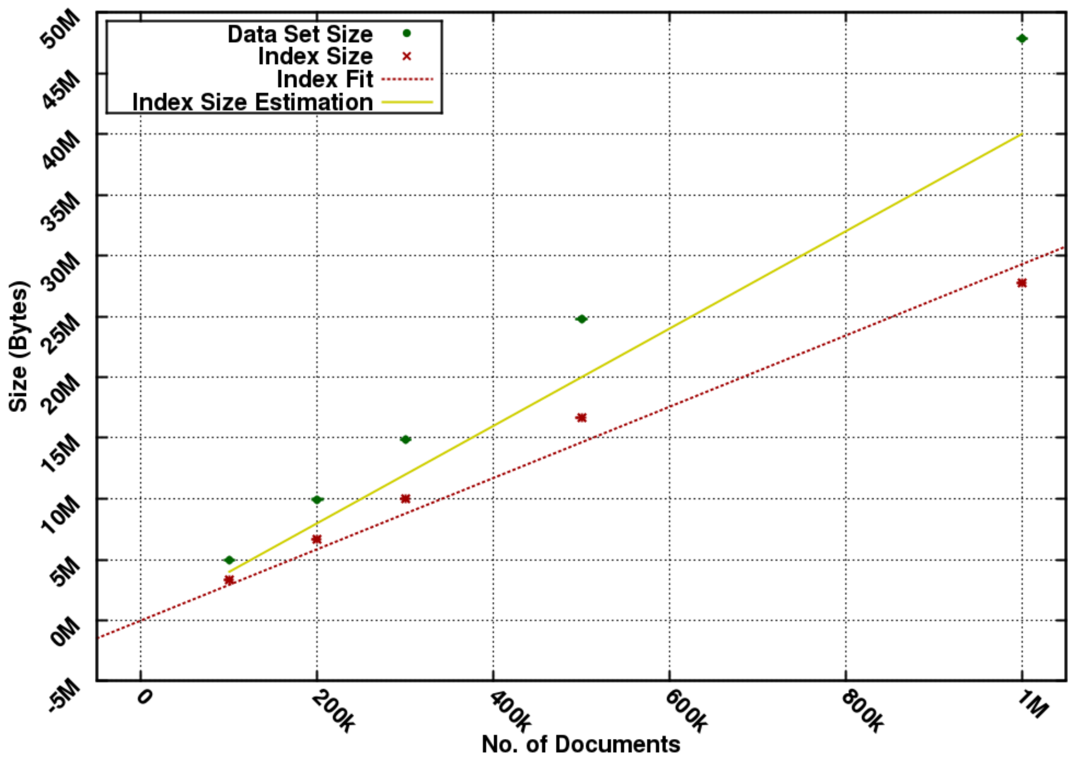


Figure 4.7: Index size growth pattern for attributes with ***Array*** of integers as value type to the number of documents in the dataset. The theoretical estimation in calculated based on the equation 4.5 and the block size assumption of $\tau_{int} = 8\ Bytes$. The dataset is generated with random array size generator that as average converge to a fix number (in this case ten). Therefore, the *Index Size Estimation* shapes a straight line. It also explain the deviation of real *Index Size* markers from the *Index Fit* line.

There are some additional concerns involved to calculate theoretical estimation for indexes build on attributes with array value types. The documents of Figure 4.7 contain arrays of an integer for each of the tested attributes. In this case, the equation to estimate the index storage consumption can be defined as:

$$S_{array} = \tau_{type} * \left(\sum_{j=1}^{n} l_j \right) = \tau_{type} * n * \bar{l} \tag{4.5}$$

where l_j is the length of each array value of the attribute. The workload is generated such that the average length of array values is a given number. This reason explains the straight line for the theoretical estimation in Figure 4.7 in contrast to the slight curve of the actual index sizes that has a slight deviation from it fitting line. Therefore, theoretical estimation for index size of an attribute with array value type is done based on the maximum array size of that attribute.

Additional information about the index estimation can also be obtained by comparison of Figures 4.4 and 4.6 to each other. The index in Figure 4.6 is generated on an attribute in a three layer deep nested document that contains integer values. The similarity between the pattern of these figures shows that having layers of nested documents does not have any direct influence on the required storage consumed by indexes. Therefore, the estimation of index size for such attributes can also be merely based on the type of value ($S_{nested} = S_{single}$).

The same approaches can also be applied when the size of a compound index should be estimated. At least three methods were considered to estimate the size of a compound index:

1. find a typical pattern for the compound index size growth about the number of documents in the database,

2. utilization of sum of the storage sizes of single indexes,

3. utilization of sum of the theoretical storage sizes of single indexes

as a theoretical estimation.

Each of these methods has their advantages and disadvantages that are explained in the rest of this section. In this regard, Figures 4.8, 4.9, 4.10, 4.11, 4.12, and 4.13 are produced to demonstrate the associated measurements to each of these methods. For each of these figures a particular dataset is used that consists of documents with two attributes containing various data types. Each of these figures includes information about the size of two single indexes that are built on each of the attributes of the database as well as the size of the compound index constructed on both of these attributes. The fitting line that is drawn for each of single and compound indexes shows the linear pattern for the size of indexes with increasing number of documents. These figures contain indicator lines produced by summing up the size of single indexes as *Sum of Single Indexes* and by summing up the theoretical size of the single indexes as *Compound Index Size Estimation.*

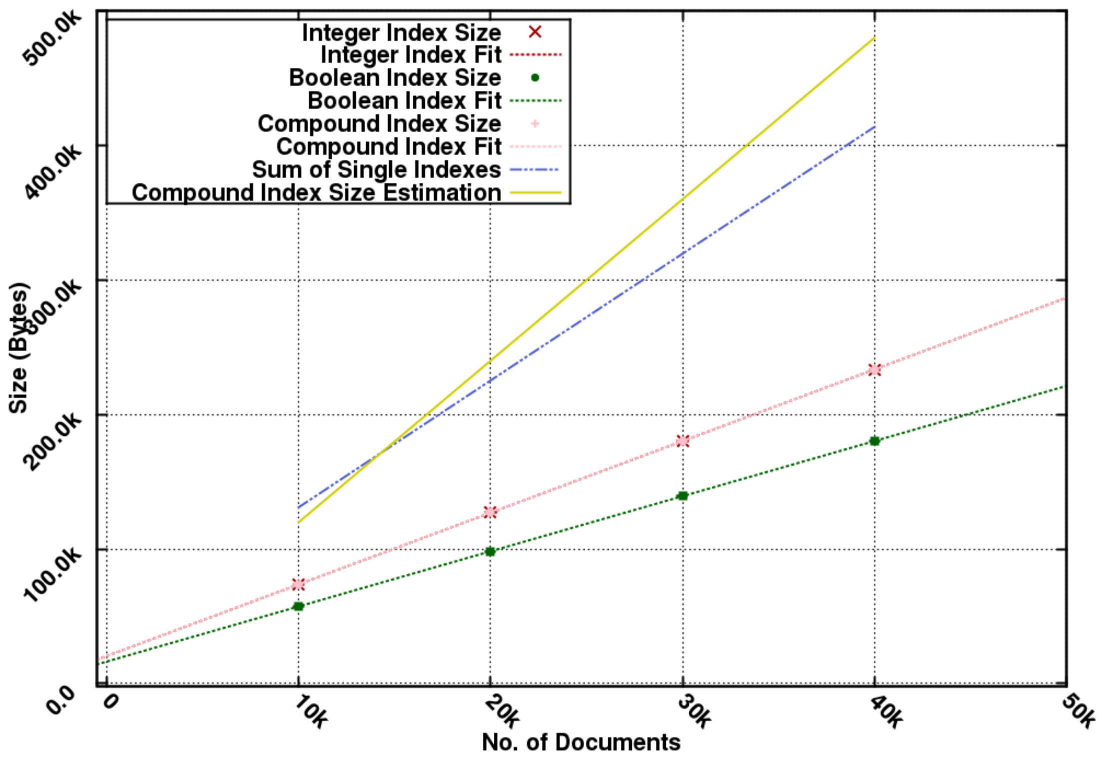


Figure 4.8: Int-Bool: Comparison of index size growth pattern between two single indexes and a compound index with increasing number of documents. The dataset consists of documents with two attributes. The values of one attribute are integer numbers and the other booleans. A single index is built on each of these attributes, and the compound index contains both of these attributes. The fact that the *Compound Index Size* markers overlap with the *Integer Index Size* crosses shows that the storage optimizations implemented in the targeted database almost ignore the size of the boolean attribute when compounding it with an integer. The *Sum of Single Indexes* is obtained by adding up the size of two single indexes which provides a theoretical estimation for the size of compound indexes. The *Compound Index Size Estimation* depicts the resulting theoretical estimation line obtained by Equation 4.6, which offers an alternative theoretical estimation in this case.

The first assumed estimation method is to use the information gathered by the linear pattern of the compound indexes for various value type combinations. However, the compound index size of Figures 4.8, 4.9, 4.10, 4.11, 4.12, and 4.13 shows that the compound index size has a different growth ratio depending on the various data types of the attributes. It happens due to the storage-related optimization strategies implemented in the targeted database. As an example, in the case of boolean value types, Figures 4.8, 4.9, and 4.10 show that the compound index size of a boolean attribute and any other type has almost the same size of the single attribute of the other type. Also, comparison of Figure 4.11 and 4.13 confirms the conclusion that only the size of attributes in nested documents depends on the type of the value. The nested level access path has an insignificant role in storage space. The compound indexes in all of these cases show a linear fitting. However, to use the compound fitting line information, an analysis of higher orders of compound attributes with various value types than just two attributes is needed. If we do not consider any restriction on the number of attributes that can be combined in a compound index, the number of possible combinations of different value types that should be investigated is very large.

Therefore, an alternative approach is to consider the fact that compound indexes are a combination of two or many single indexes. Therefore, by having the estimation of the behavior of single indexes, the size of compound indexes can be estimated by summing up the size of each single value type for the corresponding attributes. With this method, the number of cases that should be investigated reduces to as many types of values considered by the targeted database (in this case four).

The storage estimation based on this approach is shown in Figures 4.8, 4.9, 4.10, 4.11, 4.12, and 4.13 as *Sum of Single Indexes*. In almost all of these cases, the database under study uses less storage space for compound indexes than the sum of the single attribute index sizes. The only exception is Figure 4.12 where sum of the values is an underestimation of the actual compound index size. Hence, this method does not provide a theoretical estimation in all of the cases.

Additionally, the problem of being dependent on the environment and the optimizations implemented in the database still apply to this method. Furthermore, since this method generates sizes for compound indexes that are different from the actual compound index sizes, the resulted storage estimation might play an unfair role in favor or disadvantage of compound indexes.

Thus, to have an environment-independent solution, a theoretical estimation for the compound indexes can be obtained by the following equation:

$$S_{compound} = n * \sum_{i \in P} \tau_i, \tag{4.6}$$

where P is the set of single value types of each compound index attribute. n is the number of documents that contain both of the corresponding attributes of a compound index. The only requirement of this method is to specify the size of each value type as an input. By adapting this approach, similar estimation methods are used for both single and compound index size estimations. Therefore, the chance of assigning an inequitable

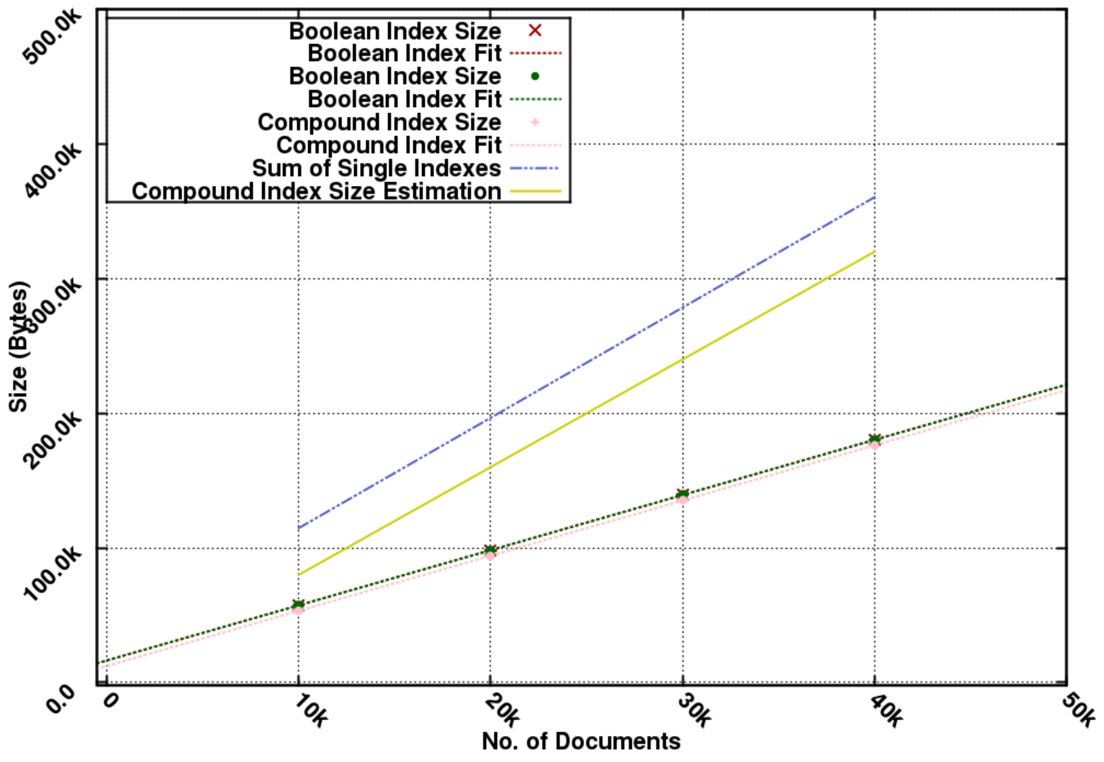


Figure 4.9: Bool-Bool: Comparison of index size growth pattern of two single indexes each on an attribute with boolean values and a compound indexes built on both of them with increasing number of documents. The indicator markers of both *Boolean Index Sizes* and the *Compound Index Size* are all overlapping in this case which reveals the storage optimization implementation strategy of the targeted database regarding boolean values. The theoretical estimation of the compound index size obtained by Equation 4.6 is depicted with *Compound Index Size Estimation* line which in this case is less than the size estimated by the *Sum of Single Indexes*. However, the theoretical line still provides a theoretical estimation of the storage estimation.

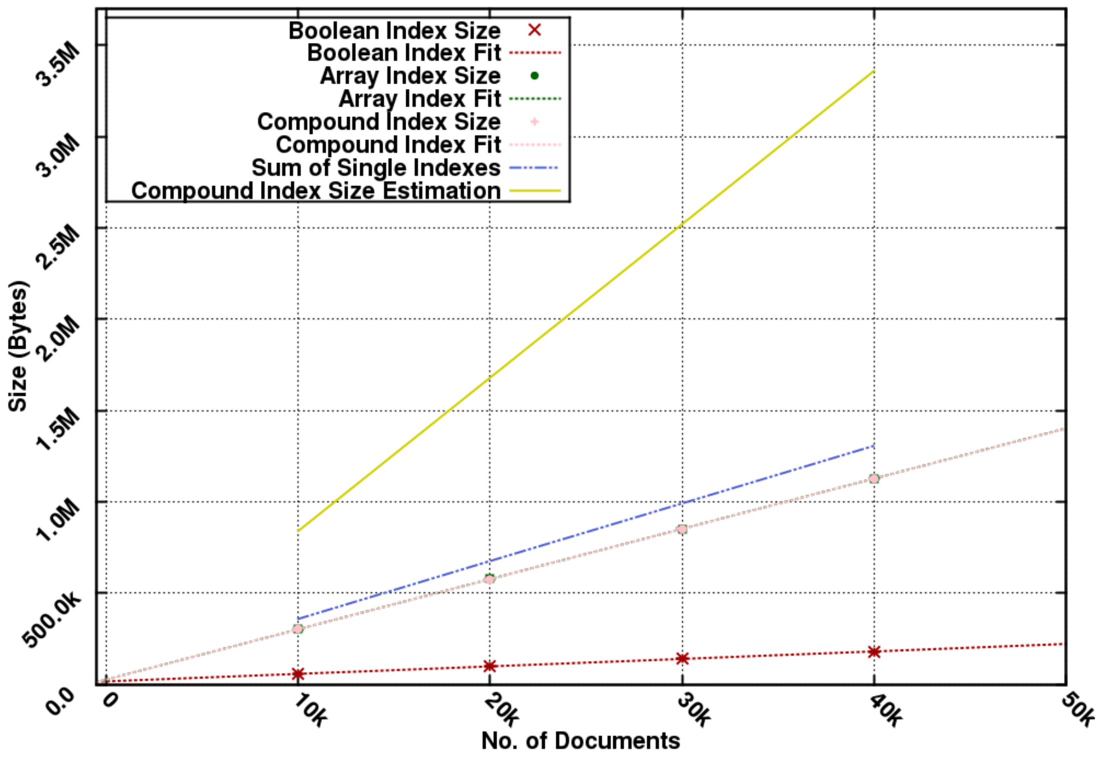


Figure 4.10: Bool-Array: Index size growth patterns on a dataset with two attributes, one containing boolean values and the other array of integers. The overlap of index size marker of the single index on arrays and the compound index again shows the implemented storage optimization strategy in the target database to consider the size of boolean index types in compound indexes negligible. The *Sum of Single Indexes* then provide a slightly higher estimation of compound index size than the actual index. The theoretical estimation shown with line *Compound Index Size Estimation* is calculated by Equation 4.7.

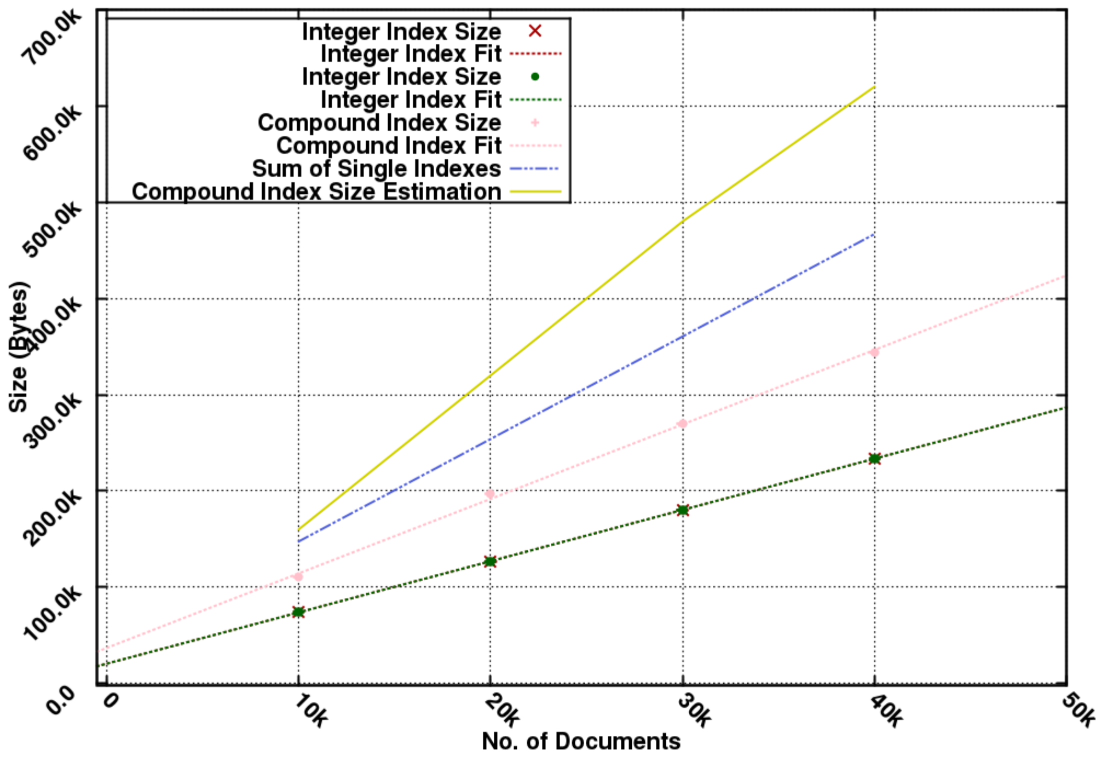


Figure 4.11: Int-Int: The comparison of the index size growth patterns on a dataset with two attributes each containing integer values. The fact that the fitting line for both single indexes and the compound index sizes passes through actual most of the error bar regions for actual markers indicates that the linear fitting is appropriate. The difference between the actual *Compound Index Fit* and the *Sum of Single Indexes* line shows the implemented storage optimization strategies in the targeted database which makes both of these estimation methods environment dependent. The theoretical *Compound Index Size Estimation* estimation is calculated by Equation 4.6.

estimation in favor or disadvantage of one index again reduces.

Example 4.4.1. For a dataset of $n = 10K$ documents, according to the Equation 4.6, the estimated index of a compound index on two attributes one containing integer ($\tau_{int} = 8$) and the other boolean ($\tau_{bool} = 4$) value types is calculated as:
$S_{compound} = n * (4 + 8) = 120K$ Bytes. This results is shown in Figures 4.8.

In order to include array attribute considerations in the compound index theoretical estimation, the Equation 4.6 changes itself to:

$$S_{compound\_array} = n * \sum_{i \in P} (\tau_i * \max(l_i)), \tag{4.7}$$

where l_i is the length of the array of values in each document with an attribute of type

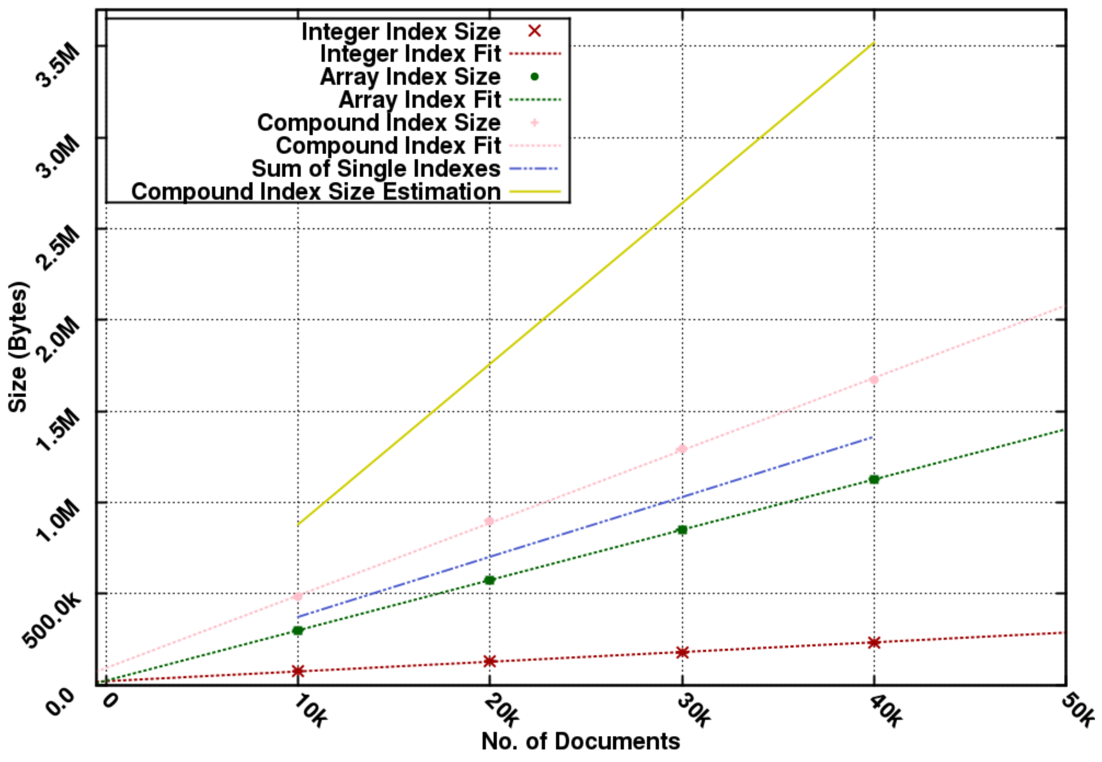


Figure 4.12: Int-Array: Comparison of the index size growth pattern with the number of documents on a dataset consisting of two attributes. One attribute contains integer values and the other arrays of integer values. This is the only case where the *Sum of Single Indexes* provides a less estimated size for the compound index than the actual size of that index, due to the storage optimization strategies implemented in the targeted database. This measurement provides another reason besides being environment-dependent for not utilizing the sum of single indexes as the theoretical estimation. However, the theoretical *Compound Index Size Estimation* line obtained by Equation 4.7 provides a proper theoretical estimation.

i. This length for scalar values is one. Since document-based databases are schemaless, one attribute might contain different value types in various documents. In such cases, the practical solution is to consider the type with larger fitting line slope and use that to estimate the storage required for that attribute. Thus obtaining a theoretical estimation is guaranteed.

Example 4.4.2. Assume a dataset with $n = 20K$ documents and two attributes, one containing boolean values ($\tau_{bool} = 4, \max(l_{bool}) = 1$) and the other arrays of integer values ($\tau_{int} = 8$) with maximum length of ten elements in an array ($\max(l_{int}) = 10$). The estimated size of a compound index on these two attributes can be obtained according to Equation 4.7 as:
$S_{compound_array} = n * (\tau_{int} * \max(l_{int}) + \tau_{bool} * \max(l_{bool})) = 20K * (8 * 10 + 4 * 1) = 1,680K$.

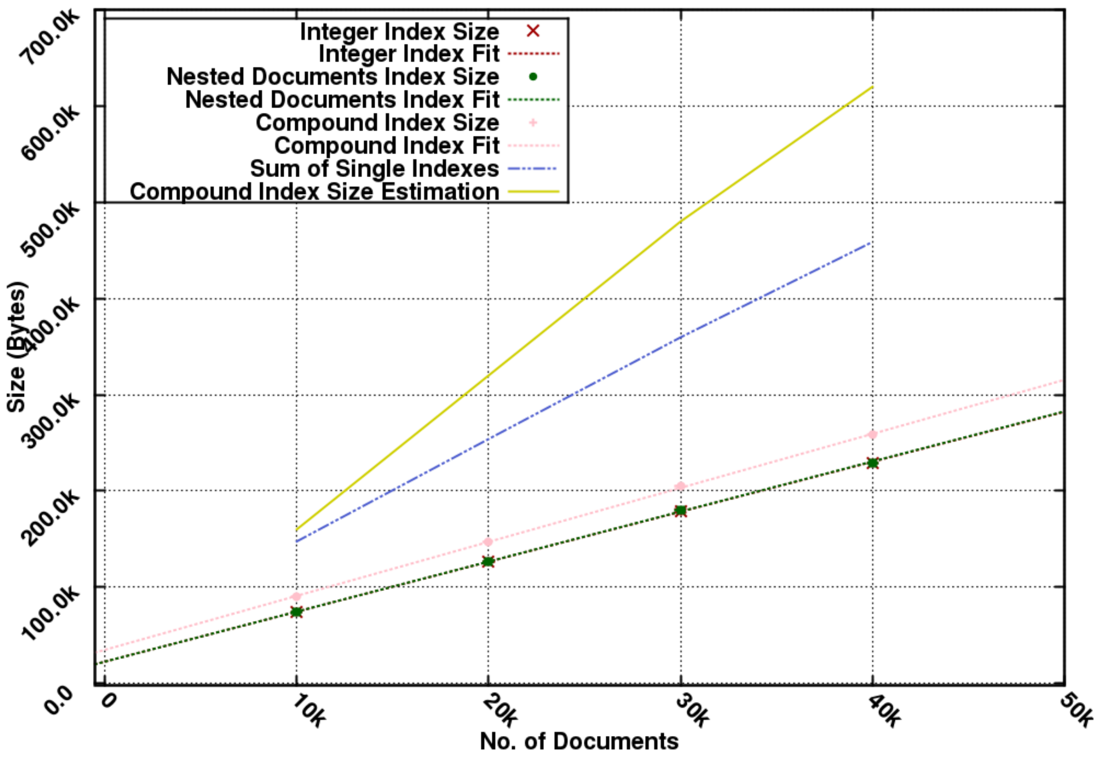


Figure 4.13: Int-Nested: Comparison of the index size growth patterns of single and compound indexes to the number of documents in a dataset with one containing integer values and another attribute containing three layers of nested documents which then include an attribute with an integer value. The fact that the single index size markers on these two attributes overlap reveal that only the internal value type of nested documents are playing a part in the index size. Therefore, to calculate both the *Sum of Single Indexes* and the theoretical estimation of *Compound Index Size Estimation* obtained by Equation 4.6, two integer value types are considered. The results of this figure are similar to the results of Figure 4.11

This results is illustrated in Figure 4.10.

In this section, the importance and challenges of estimating the needed storage capacity of indexes are discussed. Since the size of indexes for each configuration index can not be directly measured on the chosen sample set, where the indexes are built, the actual size of indexes on the original dataset should be estimated. For such an estimation, several methods are proposed, and the pros and cons of each of them are discussed. The considered methods are as the following:

- parametrization of index size based on number of documents containing the corresponding attributes of that index,
- theoretical usage of a defined theoretical estimation and the attribute type.

To have a less environment-dependent approach, the usage of the theoretical estimation is chosen as the most practical and fair estimation method. This theoretical estimation is defined based on the type of values that the single attributes indexes contain. The theoretical estimation for the compound indexes are calculated as the sum of this theoretical estimation of the single index of each of the attributes in a compound index. Due to the storage optimization methods that might be implemented in each targeted database, the estimation of the indexes by the proposed theoretical estimation method can differ from the actual index sizes. However, this approach provides a uniform method to define a theoretical estimation on the index size estimation which is not expected to play in favor or disadvantage of any of the indexes.

4.5 Summary

There are several aspects that had to be studied before finalizing this architecture. Therefore, the extraction of the relevant factors is an important part of the study. Based on these factors, then the architecture of the adaptive index recommendation system is designed. To design the integrated index recommendation system, it is required to consider various practical considerations. Each of modules of AIRs contains the implementation of at least one of these practical studies and aspects.

A summary of the contributions that are debated in this chapter are as the following:

- The modular architecture design of an adaptive index recommendation system
- Development of the representative sample-based virtual environment to reduce the load of the index evaluation part
- Utilization of the Frequent-Long strategy to reduce the search space for the index recommendation problem
- Adaption of merging algorithm to provide a chance for the evaluation of the transformed set of indexes
- Development of a strategy to estimate the approximated size of indexes without creating them

The comprehensive discussion over the theoretical studies that are conducted to develop the index recommendation system, are deliberated in the next chapter.

Chapter 5

Cost Model for Configuration Evaluation

This chapter discusses most of the theoretical issues regarding the design of the index recommendation system.

As discussed in Problem 2.3.3 of Section 2.3, the profit of any set of indexes for a given workload can be examined with a proper objective function. The essential theoretical complexity of the index recommendation problem can be formulated with an objective function with most of the identified parameters presented in Section 4.1. Maximization of this objective function should result into the extraction of the optimal solution set of indexes for a workload. The formulation of this objective function and an efficient enumeration technique to traverse the search space of candidate indexes for this function are presented in Section 5.1.

As stated in Problem 2.3.2 in Section 2.3, even for a moderate sized set of candidate indexes, the number of possible combinations that should be traversed can be very large. Also, an enormous number of calls are made to the query optimizer that projects a heavy load on the in-production system to process all possible configurations. AIRs addresses a solution for this matter based on the concept of *atomic configurations*. This solution, its corresponding algorithm, and the analysis of its complexity are discussed in Section 5.2. The chosen traverse method as a solution to the Problem 2.3.5 of Section 2.3 to find the globally optimal set of indexes is presented in Section 5.1.1.

The initial formulation of the problem that is presented in this chapter is published in [122].

5.1 Objective Function Formulation

As discussed in Section 4.1, to determine the optimal set of indexes for a workload, several conditions and parameters should be taken into account. In general, these parameters depend on: 1) the initial state of the database, 2) the type of input workload and 3) the environment where the database is running (see Table 4.1). Formulating these parameters into an objective function is necessary to find the best set of indexes. By maximization of this objective function through an efficient enumeration technique, the optimal set of indexes for that particular workload under the specified conditions can be recommended. This section is dedicated to the formulation of an objective function in an adequate optimization method.

The introduction of indexes reduces the response time of the database to *read* queries. An appropriate selection of indexes reduces the number of documents that should be scanned to retrieve necessary information for the query. Therefore, there is a *benefit* associated with each index that serves a search query. However, the existence of each index introduces a maintenance cost in case of any write operation into the system. This cost is caused by required updates that should be made into corresponding indexes in the system. As an example, the search-predicate of an update operation can benefit from a suitable index. However, the update-predicate enforces adjustments into that index and all of the other indexes in the system with the corresponding attributes.

Let workload W consists of n queries as $W = \{q_1, q_2, ..., q_n\}$. Also, consider there are m indexes in the full set of candidate indexes as $E = \{I_1, I_2, ..., I_m\}$. This set includes all single- and multi-attribute indexes that correspond to the attributes in the most frequently used queries as well as merged indexes which are respectively discussed in Section 4.3.1 and 4.3.2. The associated size of each index is denoted by s_j for $j \in [1, 2, ..., m]$.

Definition 5.1.1. ***Configuration***:
A set of indexes $C_k = \{I_{k1}, I_{k2}, ...\}$ so that $C_k \subset E$ is called a *configuration.*

As there are 2^m possible configurations, I restrict my work to a subset of all configurations that contain index sets that are used by some query from the workload. Such configurations are known as *atomic configurations* and are defined in Definition 5.1.2.

Definition 5.1.2. ***Atomic Configuration***:
A configuration C_k is an *atomic configuration* for a query q_i, if an execution plan of q_i is possible that uses all of the indexes in C_k. A configuration is also defined as an atomic configuration for a workload W of all queries, if it is atomic to at least one query in that workload [75].

The objective of the index recommendation system is to determine the optimal set of indexes C_k that has highest *benefit* for the workload under the storage constraint applied by the environment after considering the corresponding maintenance cost. Therefore, in the presence of the configuration C_k, the objective function for a given query q_i from the workload W can be defined as:

$$Z^{(i)}(C_k) = \texttt{Ben}^{(i)}(C_k) - \texttt{Maint}^{(i)}(C_k), \tag{5.1}$$

where $\texttt{Ben}^{(i)}(C_k)$ and $\texttt{Maint}^{(i)}(C_k)$ are respectively the benefit and maintenance cost terms.

Definition 5.1.3. ***Benefit***:
The *benefit* of an index I_j for the query q_i is defined as the reduction in the cost of running q_i, utilizing I_j in comparison to the cost without any index.

Thus, the $\texttt{Ben}^{(i)}(C_k)$ term can be defined as the difference between the number of documents that should be scanned in order to retrieve the required information for query q_i by using indexes in configuration C_k and the same number while no index exists in the system. However, since indexes typically reduce the number of scanned documents by a few order of magnitude, the logarithmic cost is considered.

Let the cost of scanning the database to run a query q_i by using configuration C_k be denoted as $Cost_s(q_i, C_k)$. This cost is equal to the logarithm of the number of documents that should be scanned to retrieve necessary information for q_i by using indexes in configuration C_k. Then the $Ben^{(i)}(C_k)$ can be defined as the difference between $Cost_s(q_i, C_k)$ and the cost of running the query while no index exists in the system, which is shown as $Cost_s(q_i, \emptyset)$. In mathematical form, the benefit of configuration C_k for running the query q_i is defined as the following:

$$\texttt{Ben}^{(i)}(C_k) = Cost_s(q_i, \emptyset) - Cost_s(q_i, C_k). \tag{5.2}$$

When no index is present in the system, all of the documents must be scanned to execute a query. The number of scanned documents utilizing any set of indexes cannot exceed the total number of documents. Therefore, the benefit of configuration C_k for a query q_i is greater if the difference between the costs in Equation 5.2 is larger. In the worst case, the benefit of the configuration C_k for query q_i is zero which happens in the case that q_i does not use any index of C_k. The definition of Equation 5.2 is also valid for the search-predicate of write operations such as update and delete that benefit from having relevant indexes in the system.

However, there is another cost associated with configuration C_k if the query q_i is a *write* operation (i.e. update, delete or insert). This cost is related to updating the relevant indexes of configuration C_k that shapes the core of $\texttt{Maint}^{(i)}(C_k)$ in Equation 5.1. For any write query q_i in the workload, each index I_j in C_k should be updated if the update-predicate of the write query q_i has at least one common attribute with attributes in that index. Equation 5.3a represents this concept:

$$\texttt{Maint}^{(i)}(C_k) = \sum_{I_j \in C_k} Maint^{(i)}(I_j), \tag{5.3a}$$

where $\texttt{Maint}^{(i)}(I_j)$ is defined as:

$$\texttt{Maint}^{(i)}(I_j) \equiv F_{ij} = f_{ij} * Cost_u(j) \tag{5.3b}$$

where $f_{ij} \in \{0, 1\}$ is a coefficient that indicates the existence of common attributes in index I_j and query q_i as described in Equation 5.3c and $Cost_u(j)$ is the cost of updating the existing indexes as defined in Equation 5.3d. The indicator coefficient f_{ij} is one if there is at least one common attribute in the search-predicate of the query q_i and index I_j, and is equal zero otherwise:

$$f_{ij} = \begin{cases} 1 & I_j \bigcap q_i \neq \emptyset \\ 0 & \textit{otherwise.} \end{cases} \tag{5.3c}$$

The $Cost_u(j)$ is associated with the time complexity of updating any index in the corresponding configuration. Index structures, even in document-based databases, are mostly in the form of a B-tree. The complexity of the insertion operation in a B-tree is of order $O(\log n)$ [126], where n, in this case, is the number of nodes in the tree. The size of a B-tree for an index depends on the `cardinality` of its attributes (see Definition 2.2.1). Therefore, the associated cost of update operations $Cost_u(j)$ is defined as:

$$\forall j : I_j \in C_k \qquad Cost_u(j) = \log CA_{a_h}, \tag{5.3d}$$

where CA_{a_h} is the cardinality of the attribute indexed by index I_j. If index I_j is a single-attribute index, obtaining its corresponding cardinality CA_{a_h} is straightforward. However, for a multi-attribute index, the educated guess is to use the cardinality of an attribute that has the maximum cardinality among attributes of that index. In case of a failure to obtain the cardinality, a common practice is to consider the number of documents in the dataset as the cardinality of the index. Therefore, the cardinality that is considered for the cost calculation in Equation 5.3a is defined as:

$$\forall h : h \geq 1, a_h \in I_j \qquad CA_a \leq \min\{\ \{\max_h a_h\}, \#docs\}. \tag{5.3e}$$

where a_h is an attribute in I_j and $\#docs$ is the total number of documents in the collection.

Now, Equation 5.1 can be solved to obtain the profit of using configuration C_k to run query q_i. The goal is to find the single configuration that maximizes the profit for each query. Therefore, the objective function is to find the single configuration that maximizes the profit for the whole workload W and can be defined as the following:

$$Z(C_k) = \sum_{i=1}^{n} Z^{(i)}(C_k). \tag{5.4}$$

Moreover, the optimal solution should maximize Equation 5.4 considering the constraint of the total available storage size S. This constraint can be defined as the following:

$$\forall j : I_j \in C_k \qquad \sum_{j=1}^{m} s(I_j) \leq S. \tag{5.5}$$

To investigate all possible cases and to find the optimal solution, the objective function should be formulated in the form of a proper optimization model. The rest of this section contains a brief discussion of the reasons for the choice of a particular enumeration technique (Section 5.1.1). Then, in Section 5.1.2, the reformulation of the objective function into the chosen optimization model is presented.

5.1.1 Optimization Model and Enumeration Technique

To extract the optimal configuration from Equation 5.4, all configurations should be traversed effectively. Configurations are formed by a union of all possible combinations of indexes in the initial candidate index set for each query, which is known as the *enumeration space*. The number of all possible configurations, even for a moderate size of candidate index set, can get very large. Therefore, it is important to apply a proper heuristic to reduce the enumeration space.

Based on the approach explained in Section 4.3, the enumeration space contains all of the frequently repeated patterns of single- and multi-attribute indexes and the required merged indexes of them.

Nevertheless, the number of configurations that should be traversed is usually large enough so that an exhaustive search would not be feasible and waste many resources. Thus, it is important to apply an effective optimization method with an appropriate enumeration technique to find the optimal configuration.

As discussed in Chapter 3, there are already many optimization methods that are used to solve similar objective functions for index recommendation. The optimization methods take advantage of two main enumeration categories: bottom-up and top-down enumeration techniques.

The bottom-up approach starts with an empty set of indexes as of the initial configuration and gradually adds indexes to this set. Many approaches to the index recommendation problem formulate this problem with different forms of bottom-up models, such as the knapsack problem [66] or greedy hill-climbing problem [75], [127]. The bottom-up enumeration strategies search the enumeration space for a subset that is a perfect fit to satisfy the storage constraint and has the maximum profit for running the workload.

These bottom-up techniques are most beneficial when the storage limitation is small because the final configuration is likely to consist of only a few indexes. On the other hand, the bottom-up approach requires that each primary element of the system be specified in detail. Therefore, it is necessary to define the profit of each index before the implication of the enumeration technique. However, since modern query optimizers are usually capable of intersecting indexes with each other, granting a predefined and fixed profit to each index results in an inaccurate evaluation of configurations. Also, given the merged indexes are not the first choice of the query optimizer to run a query when the original indexes are also present, these approaches do not provide an exact benefit evaluation for merged indexes. Additionally, neither of the techniques mentioned above guarantee to find the optimal global solution, and they might end up at the locally optimal solution [1]. Therefore, it is worth trading off more optimization time for a better solution.

In contrast, top-down approaches begin with large initial configurations that might overflow the storage threshold. Then, they gradually remove the low-impact indexes from this configuration till the configuration fits into the storage constraint. The top-down approaches have several advantages, especially if the storage limitation is not that tight. Top-down approaches provide information about approximately optimal solutions through a relaxation of the constraints. Therefore, they provide additional information about more efficient configurations than the recommended one which do not fit into the storage limitation. This information in part can be helpful in decision making for the resource management of the system (e.g. the increase of disk or memory storage of the current database environment).

Considering that the document-based databases are mostly favorable for their easy to scale solutions, it is natural to assume that the available storage size is not that tight for their instances. Therefore, it is decided to take advantage of the top-down algorithms.

Also, to make sure that the algorithm returns the globally optimal solution, working with a *deterministic optimization method* [128] is chosen. Additionally, the fact that creating a partial of an index does not have practical meaning results in discrete feasible sets. Consequently, the *Integer Linear Programing* (ILP) is chosen as the optimization method that internally benefits from a top-down enumeration technique, i.e. a Branch-and-Bound algorithm (see Section 5.1.2). Therefore, the algorithm guarantees to return the optimal solution when it finishes running entirely. However, if the algorithm is interrupted due to run-time limitations, a suboptimal solution with a quality indicator is returned that is based on the distance of this solution from the optimal one.

In the rest of this section, the formulation of the general objective function in Equation 5.4 and its constraint in Equation 5.5 as Integer Linear Programming method will be presented.

5.1.2 Integer Linear Programming Formulation of the Objective Function

The purpose of this subsection is to formulate the objective function 5.4 and the constraint 5.5 in the Integer Linear Programming. To define the objective function with ILP, two condition variables x_{ik} and y_j are introduced. For each query q_i, there are p configurations to be investigated and the query should be evaluated by each configuration C_k at a time. Thus, the decision variable x_{ik} can be defined as the following:

$$\forall i,k:\ 1 \le i \le n,\ 1 \le k \le p \qquad x_{ik} = \begin{cases} 1 & q_i \textit{ uses } C_k \\ 0 & \textit{otherwise.} \end{cases} \tag{5.6}$$

Moreover, as described by Equation 5.3a, there are few of m candidate indexes built in any configuration C_k. Therefore, another decision variable y_j can be defined as:

$$\forall j:\ m \ge 1, I_j \in C_k \qquad y_j = \begin{cases} 1 & I_j \textit{ is built} \\ 0 & \textit{otherwise.} \end{cases} \tag{5.7}$$

Utilizing these conditional variables, the objective function 5.4 can be rewritten as:

$$\forall j: I_j \in C_k \qquad Z = \sum_{i=1}^{n}\left(\sum_{k=1}^{p} b_{ik}\cdot x_{ik} - \sum_{j=1}^{m} A_i \cdot F_{ij}\cdot y_j\right) \tag{5.8}$$

where $b_{ik} \equiv Ben^{(i)}(C_k)$ is the benefit defined in Equation 5.2 and F_{ij} corresponds to the maintenance costs that is defined in Equation 5.3a. A_i is introduced so that the maintenance cost is only calculated for write queries and is defined as:

$$A_i = \begin{cases} 1 & q_i \textit{ is write query} \\ 0 & \textit{otherwise.} \end{cases} \tag{5.9}$$

The optimal solution of the Equation 5.8 is denoted by Z^*. This solution should be obtained by considering the proper constraints of the system that are expressed in the series of Equations 5.10a-d.

As described in Equation 5.5, the first requirement to consider is that the sum of the size of all indexes in the chosen configuration C_k should not exceed the total amount of available storage S. Equation 5.10a is defined to satisfy this constraint:

$$\forall j : I_j \in C_k \qquad \sum_j s_j \cdot y_j \leq S. \tag{5.10a}$$

Another constraint is based on the fact that the database should use only one configuration at a time to execute any query q_i. The constraint 5.10b represent that the existence of any two configuration at the same time is mutually exclusive:

$$\forall i : 1 \leq i \leq n \qquad \sum_{k=1}^{p} x_{ik} \leq 1. \tag{5.10b}$$

The *less than* sign in the mutually exclusive condition of Equation 5.10b suggests that the case without any of the p (atomic) configurations to run the query q_i is also possible. By changing this sign to equality, the condition restricts to not include such case. However, it is desirable to cover such a case, because for any workload W, there might be some query q_i that does not profit from any index set.

The next constrain originates from the dependency between the configuration C_k and its corresponding indexes. For the database to be able to utilize any configuration C_k to execute a query q_i, all of the indexes of that configurations should be build. Therefore, the variable x_{ik} suggests the query i using the configuration k is dependent on the variable y_j for all of the indexes in that configuration. The conditional constraint 5.10c formulates this requirement:

$$\forall i : 1 \leq i \leq n, \quad \forall k : 1 \leq k \leq p, \quad \forall j : I_j \in C_k \qquad x_{ik} \leq y_j. \tag{5.10c}$$

The last constraint follows the nature of decision variables x_{ik} and y_j as described in Equations 5.6 and 5.7 respectively. These constraints represent the binary nature of the introduced decision variables. The usage of integer decision variables for the case of index recommendation system makes perfect sense; because building half of an index does not have a practical meaning. Also, it is not possible for a query to use a fraction of an atomic configuration. The Equation 5.8 can be solved by ILP under the constraints in series of Equations 5.10a-d.

By formulating the problem as ILP model, some known methods can be applied to solve the objective function for the `enumeration space` of the index recommendation problem.

The optimal solution (x^*_{ik}, y^*_j) for the objective function and its corresponding optimal value Z^* can be obtained by the ILP approach.

When the enumeration space is small, the ILP can quickly return the optimal solution [1]. However, finding a feasible solution for integer linear program problems is NP-hard [97]. Therefore, in general, there is no known algorithm to solve an ILP problem in polynomial time. One standard approach to solving ILP problems is first to find a relaxation that is numerically more feasible to solve and to obtain an approximation of the optimal solution for ILP.

By dropping the integer restriction on the variables of an ILP problem, it converts to a linear program. Nonetheless, there are efficient algorithms that can solve Linear Programs(LP) such as the Simplex algorithm [97]. By removing the integer constraint, a relaxation of the ILP model is applied which is known as *LP relaxation* [108].

The reason to use the LP relaxation is not only because it can be done efficiently, but also because the optimal solution of the ILP is never better than the optimal solution of its LP relaxation [108, chapter 6]. Meaning that in the case of maximizing the objective function, the solution to the linear problem (x^0_{ik}, y^0_j) has a better or similar profit to the ILP optimal solution or $Z^* \leq Z^0$. As a result, in maximization cases, LP relaxation solution Z^0 provides an upper bound for the optimal value of the ILP objective function Z^*. Utilizing the LP relaxation provides much useful information such as finding sub-optimal solutions with excellent performance, but with larger size than the available storage limitation. The Z^0 is a fractional value. The optimal solution Z^* cannot just be obtained by rounding Z^0 down. Usually, an enumeration method should be used to find the feasible solutions for the ILP problem.

There are several enumeration methods to solve the ILP problem, such as the Branch-and-Bound (B&B), the Cutting Plane, and the Branch-and-Cut methods [105]. The Branch-and-Bound is the most common method for enumeration method to solve ILP problems and involves solving multiple LP relaxations.

Branch-and-Bound builds a tree of solutions that starts with solving the objective function with LP relaxation. Therefore the value of the root node of a B&B tree is the upper bound Z^0 to the ILP problem. Based on the heuristic that is used in the algorithm (e.g. rounding down from fractional solution) the best integer solution $(x^{int}_{ik}, y^{int}_j)$ and its value Z^{int} can be obtained. The upper bound value Z^0 can be used to measure the distance of any best integer solution to the optimal solution. This distance is calculated as:

$$d(Z^0, Z^{int}) = \frac{Z^0 - Z^{int}}{Z^0}. \tag{5.10d}$$

Based on this distance, the quality gap of each solution to the optimal solution can be defined as $d(Z^0, Z^{int}) * 100$. Since solving multiple LP problems can take too long, utilizing the B&B algorithm has the advantage that the execution of the algorithm can be stopped at any level. Then a suboptimal feasible solution is returned with a measurement of its quality that is defined based on Equation 5.10d. Additionally, if this algorithm is run to its completion, it returns the optimal solution Z^*.

The objective of an index recommendation system is presented in its mathematical form in this section. The objective function is formed by considering parameters to cover the state of the database, the environment where a database is running in and the entry workload. After a brief discussion of the pros and cons of different optimization methods, the ILP is chosen to solve the objective function. The objective function and its constraints in ILP are formulated.

Solving ILP is done by utilizing LP relaxation in the form of the Branch-and-Bound enumeration method. However, it is shown that the run-time for the B&B algorithm to find the optimal solution can grow exponentially by the number of configurations that should be investigated. Hence, it is beneficial and necessary to reduce the number of configurations as much as possible. The next section presents the approach to find the minimum number of basic configurations that should be investigated by the optimization method discussed in this section.

5.2 Index Benefit Graph for Search Space Reduction

The query optimizer should evaluate the cost of running each query in the presence of a particular index configuration. Therefore, to provide all of the cost evaluations for the optimization of the indexes in Section 5.1, an enormous number of configurations should be evaluated. Evaluating each of these configurations puts a heavy load on the optimizer. One popular approach to reducing the number of calls to the optimizer is based on the concept of atomic configurations. This section presents the idea to find all of the atomic configurations by building a graph of the chosen indexes by the optimizer and evaluating their costs that are provided by the optimizer. This method results in the reduction of the number of calls to the query optimizer that reduces a load of index recommendation system on the in-production system. Consecutively, the number of configurations that should be evaluated by the enumeration method (see Section 5.1) is massively restricted.

Let E be the set of m candidate indexes as $E = \{I_1, I_2, \dots, I_m\}$ for a workload Q that consists of n unique queries as $Q = \{q_1, q_2, \dots, q_n\}$ out of all queries in workload W. Each subset of the candidate indexes forms a configuration that should be evaluated by the enumeration method for any of the n queries. Therefore the number of configurations (N_{wkl_config}) that should be evaluated depends on the m number of indexes in the initial candidate index set E and n number of unique queries. It is calculated in the following:

$$N_{wkl\_config}(n, m) = n * \sum_{j=1}^{m} \frac{m!}{(m-j)!\,j!}. \tag{5.10e}$$

This number grows exponentially with the number of candidate indexes in the E set.

Example 5.2.1. For only five individual queries ($n = 5$) and ten candidate indexes ($m = 10$), the number of configurations that should be evaluated is: $N_{wkl\_config}(n, m) = 3,855$.

For the evaluation of the cost of running any of the queries in Q with each of these configurations, many calls to the query optimizer are needed. It means, to provide the necessary cost estimations to optimize the objective function in Section 5.1, a heavy load is projected on the query optimizer to evaluate each query under each configuration. Thus, developing a method to reduce the number of configurations and reducing the number of calls to the query optimizer will result in a better performance for the index recommendation system. One interesting approach to significantly reduce the number of configurations is based on the concept of *atomic configurations* [75].

The cost of non-atomic configurations can be obtained from the cost of these atomic configurations. Chaudhuri et al. in [75] prove that for running the query q_i with any configuration C, there is an atomic configuration C_k such that the $Cost_{scan}(q_i, C) = Cost_{scan}(q_i, C_k)$.

The atomic configurations represent a small fraction of the whole possible number of configurations. Since this number of configurations constructs the search space for the optimization method, this concept results in the reduction of the search space. Also, each method that is used to identify the atomic configurations requires some calls to the optimizer. Although the eventual number of calls to the optimizer will be reduced by the atomic configuration concept, it is important to use an efficient and effective method to extract the atomic configurations.

To extract the atomic configurations, the concept of *Index Benefit Graph* (IBG) [72] as an adaptive strategy is leveraged and it is further developed for an optimizer with index intersection property. This strategy provides a method to extract atomic configurations by interacting with the query optimizer (see Section 6.3.2). Section 5.2.1 is dedicated to the construction of an IBG and its corresponding algorithms to extract cost evaluations of a given query. Then, the complexity of the algorithm is discussed in Section 5.2.2.

5.2.1 Construction of an Index Benefit Graph

Definition 5.2.1. ***Index Benefit Graph*** (IBG):
An *Index Benefit Graph* is a directed acyclic graph [129] that is constructed per any given query. It is constructed by a top-down approach that starts projecting the query to all of the indexes in the enumeration space. A complete IBG contains all of the atomic configurations for its corresponding query under the give enumeration space.

Each IBG contains the following elements:

- The label of each edge represents the set of available indexes (configurations).
- Each node name indicates the set of chosen indexes by the query optimizer.
- The label of each node contains various costs such as storage size of chosen indexes, the number of scanned documents, etc.

Examples of IBG graphs are depicted in Figures 5.1, 5.4 and 5.2. The Algorithm 5.1 represents the top-down approach to build an IBG.

Algorithm 5.1 Build_IBG(q_i, E)

```
Result: Root node of the IBG for the query $q_i$ and the initial candidate index set $E$.
$C' \leftarrow$ chosen_indexes_by_optimizer($q_i$, $E$)
$Cost_{C'} \leftarrow$ costs_of_chosen_indexes($q_i$, $E$)

if $node\_name = C' \wedge incoming\_edge = L$ then
    add $E - I_j$ as edge($N$,$N'$)
    return 0
else
    $N \leftarrow$ make_node($C'$, $Cost_{C'}$)
    foreach $I_j \in C'$ do
        $N' \leftarrow$ Build_IBG($q_i$, $E - I_j$)
        add $E - I_j$ as edge($N$,$N'$)
    end
    return $N$
end
```

Let $E = \{I_1, I_2, \ldots, I_m\}$ be the set of initial candidate indexes. It is the initial configuration for all of the n index benefit graphs of queries in the set of unique queries Q of the workload W. Also, it is the label of the first edge of the IBG. According to Algorithm 5.1, the IBG for the query q_i is constructed first by letting the query optimizer select the index set C' from the initial candidate index set E. Also, the costs of running q_i with configuration C' are assigned to $Cost_{C'}$. The $Cost_{C'}$ contains an execution cost, e.g. logarithm of number of scanned documents, and storage cost, e.g. the sum of index size of configuration C'. A node of the graph is built with the C' as its name and the $Cost_{C'}$ as its label. Then $|C'|$ sub-graphs are spawned starting with $E - C'_i$ for $1 \leq i \leq |C'|$. Algorithm 5.1 stops processing a configuration when a node exists in the graph that has the same name as the candidate set of that configuration and the same incoming edge label as the available set of indexes in that configuration.

The process of building the IBG is finished for each query when either no more indexes are chosen by the optimizer to run the query, or the set of available indexes for the configuration is empty. In either case, the cost of running the query at the end of the IBG is equal to the cost of running the query without any index.

Pruning the chosen indexes and evaluating the rest of the available candidate indexes in IBG method is a good strategy to make sure merged indexes (see Section 4.3.2) get a fair chance to get picked and evaluated by the query optimizer. These indexes might not be the first choice of the query optimizer but might satisfy the objective function better. Note in Algorithm 5.1 that nodes can potentially be duplicated. This situation happens under the condition that the query optimizer picks the same set of indexes to run the query under two different configurations.

Example 5.2.2. An example of such a graph for a simple *search* query with only a few attributes in its select-predicate is shown in Figure 5.1. The root node name shows an empty set that indicates the existence of no index. The cost in the label shows the Number of canned Documents (ND) for running this query with no index and the storage of no index (S_0). If the query was an "update," the cost label could contain more information such as `update cost`. The label of the first edge shows the initial configuration C that is available for this query. Although after choosing I_5, a set of $\{I_1, I_4\}$ indexes are still available, since the optimizer apparently does not choose any of them to run the query, no new node is added to the graph.

The execution of the simple query in Figure 5.1 did not require any index intersection and there is only one index used at each *level* (Definition 5.2.3). Therefore, the structure of the graph is simple.

However, query optimizers normally utilize techniques such as *index intersection* and *index union* to answer multi-condition queries more efficiently. These are fundamental operations in query processing in databases [130] (see Example 2.4.5).

The evaluation of the conjunctive condition will be done through the intersection of the set of pointers to the documents. If all of the projection attributes are available via index scan, this index intersection reduces the need to execute a full collection scan. Therefore,

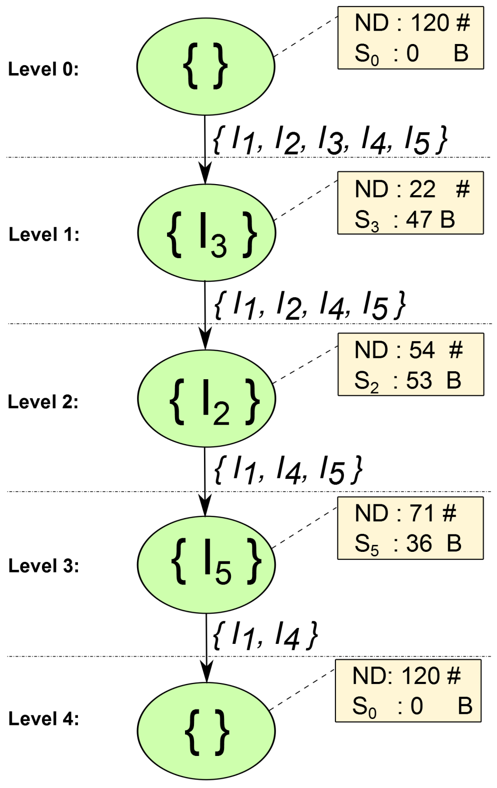


Figure 5.1: Index Benefit Graph (IBG) structure for the execution plan of a query that required no index intersection.

it can result in a significant performance improvement.

If the query optimizer uses an index intersection technique to run the query, the structure of the graph gets more complex. It can either utilize the entire index for the index intersection or the *index prefix* that is introduced in Definition 5.2.2:

Definition 5.2.2. ***Index Prefix***:
The *index prefix* consists of one or more attributes from the beginning of a multi-attribute index. For index I defined as $I = S(A|R)$ where A and R are the set of multiple attributes, A is the index prefix.

Example 5.2.3. An example of such case is illustrated in Figure 5.2. This figure depicts an IBG for a query from one syntactic workload with a conjunctive selection condition like: $A_1 = \mathit{int\_value}\ \mathit{OR}\ A_1 = \mathit{string\_value}$. The query processor of the database under study intersects up to two indexes to answer the query. The node names indicate the chosen indexes for running the query. They show that at least two indexes are used to run this query at each stage. E illustrates the entire set of candidate indexes that are available to the query at first. Then, each branch shows a situation that one of the chosen indexes are eliminated from the initial set E. It means, for a node with chosen index set $\{I_i, I_j\}$ and incoming edge label $E - \{I_k\}$, there are two further configurations that must be evaluated: 1) $E - \{I_k, I_i\}$ and 2) $E - \{I_k, I_j\}$. These form the labels of the outgoing edges. For simplicity, the cost labels are omitted from Figure 5.2. This example also contains many duplicated nodes that have different incoming edge labels at different levels.

To explain the process of extracting the *atomic configurations* from any IBG, first the concept of *level* in IBG should be introduced as in Definition 5.2.3. Algorithm 5.2 describes the method to elect the atomic configurations.

Definition 5.2.3. ***Level***:
The *level* in the IBG is defined as the distance of each node to the root node. This distance is defined as the shortest path between node N_f to the root node N_0.

Algorithm 5.2 Get_Atomic_Configurations(IBG_i)

```
Result: Set of atomic configurations for query $q_i$ with index benefit graph $IBG_i$ with $N_f$
        nodes
$p \leftarrow$ Longest_Distance_to_Root($N_f$)
$A \leftarrow$ set()
foreach $1 \leq l_v \leq p$ do
    $C \leftarrow$ Least_Cost_Node($l_v$, $N_v$)
    if $C \neq \{\ \}$ then
        add $C$ to $A$
    end
end
return $A$
```

The *Longest_Distance_to_Root* method of this algorithm finds the maximum number of levels p for the given IBG_i by finding the node with the longest distance to the root node. Since the distance is defined based on the shortest path, p is equal to the number of edges of the node with the longest path to the root node. According to the Algorithm 5.2, the most beneficial chosen-set (node in IBG) at each level l_v can be extracted by comparing the costs of these sets (such as the number of scanned documents) and picking the node with the highest profit.

Example 5.2.4. The atomic configurations for the query in example 5.2.2 are: $\{I_3\}, \{I_2\}$,

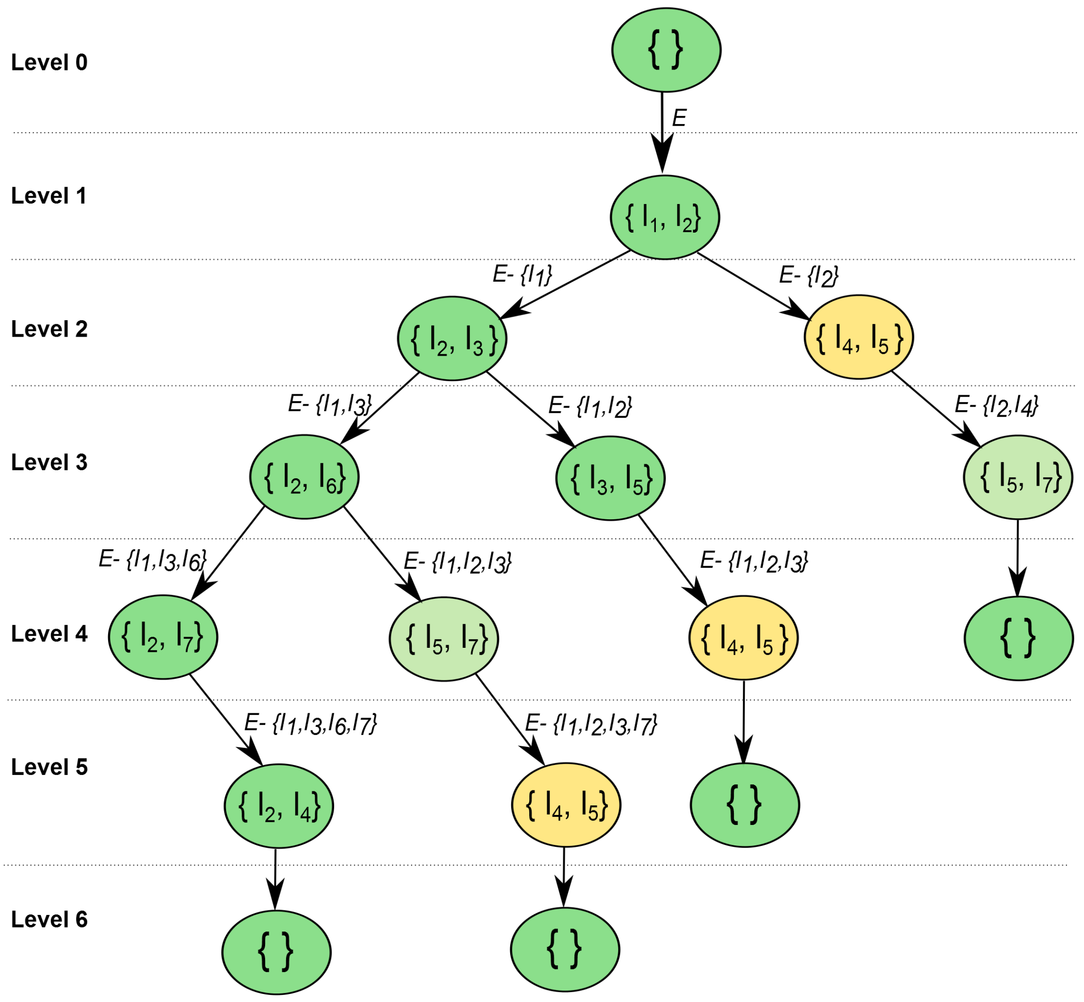


Figure 5.2: Index Benefit Graph (IBG) structure for the execution plan of an example query that required index intersection. The query processor is assumed to be capable of intersecting up to two indexes with each other.

and $\{I_5\}$.

In the case of a tie, further costs (such as update cost for an update query or storage cost) is examined. If these costs can also not remove the tie, the node that is repeated the most in the whole IBG will be chosen. This algorithm makes sure that no empty set representative node is added to the set of atomic configurations. However, even if the cost of a winning atomic configuration is similar to the cost of the root node which represents the cost of running the query without any index, that configuration should be added to the atomic configuration set. This consideration is important for situations that a small sample of the data set is chosen and the costs are evaluated on such sample and mostly result in similar costs for many nodes. (see Section 4.2).

In the rest of this section, the complexity of extracting atomic configurations is discussed.

5.2.2 Complexity Analysis of IBG

First, few assumptions are made to analyze the behavior of the IBG algorithm in evaluating the configurations and extracting the atomic ones. Then the complexity of this algorithm in best and worst case scenarios are calculated with and without considering index intersection.

The first assumption is about the maturity of the query optimizer. It relies on the assumption that the query optimizer is capable of finding the optimal plans. Frank et al. in [72] describe some of the properties of a mature query optimizer. One of these properties is described in Assumption 5.2.1:

Assumption 5.2.1. *For each query q_i, if the query optimizer chooses the set C of indexes from among the available set of indexes E, it chooses the same set C from among any subset of E that contain all the indexes of C.*

One extreme case occurs when no index is chosen from the initial set of candidate indexes. For the optimization purposes, such cases are irrelevant. Assumption 5.2.2 has been made to prevent the consideration of such irrelevant index set optimizations:

Assumption 5.2.2. *For each query q_i in the workload W, there is at least one index in the set of candidate indexes E that if it exists, it would be used by the optimizer to execute that query.*

5.2.2.1 Complexity of IBG with No Index Intersection

Considering the Assumption 5.2.2, the least number of calls to the optimizer happens when each q_i query can only be executed by one index in the initial set of candidate indexes E. Therefore, for each query, two calls to the optimizer are made:

1. to extract the query execution plan after creating all candidate indexes,

2. to extract the execution plan after removing the only satisfiable index from the set of available indexes (which in this case, returns a full collection scan).

Therefore the least number of calls N_{min_C} to the optimizer for the case of no index intersection is $2n$, where n is the number of unique queries $|Q|$ of workload W. In this particular case, there is always only one atomic configuration per query. Therefore, the least number of atomic configurations is:

$$N_{min\_A} = \frac{1}{2} * N_{min\_C} = n. \tag{5.10f}$$

Example 5.2.5. For example with five unique queries $n = 5$, the process of extracting atomic configurations requires only 10 calls to the optimizer and 5 atomic configurations each containing one index are extracted.

In the same way, if the IBGs of none of n queries of set Q contain index intersection, the maximum number of calls to the optimizer N_{max_C} can be calculated by the following formula:

$$N_{max\_C}(n, m) = n * m \tag{5.10g}$$

where m is the number of indexes in the initial candidate index set. This special case happens when all of the queries in set Q could be executed using any of the m indexes in the candidate index set.

In the case of no index intersection, any node other than the empty set representatives presents an atomic configuration. Thus, the maximum number of atomic configurations N_{max_A} as defined in Equation 5.10h can be obtained by:

$$N_{max\_A}(n, m) = N_{max\_C}(n, m) = n * m. \tag{5.10h}$$

Example 5.2.6. For the example of five unique queries $n = 5$ and ten candidate indexes $m = 10$, the highest number of calls to the optimizer is equal to the number of the atomic configurations for the whole workload which is: $N_{max\_A}(n, m) = N_{max\_C}(n, m) = 50$.

Comparing this number to 3,855 configurations - if no IBG optimization was applied - shows even the largest number of configurations that should be evaluated is at least a few orders of magnitude less than the original search space. The same is true for the reduction of load on the query optimizer by scaling down the number of required calls to the optimizer using IBG.

Table 5.1 contains the asymptotic notation indicating the limiting behavior of the IBG algorithm in best and worst case. The best complexity for the schenario without intersection is extracted based on the Equation 5.10f and the worst case scenario is based on Equation 5.10h.

5.2.2.2 Complexity of IBG with Index Intersection

So far the assumption was that either one index is sufficient to retrieve the required data for a query or the query optimizer is essentially not capable of performing query execution with more than one index per query. However, modern database query processors are mostly capable of merging two or more indexes to serve multi-condition queries.

The least number of calls to the optimizer for a given workload can be obtained by considering Assumption 5.2.2, Assumption 5.2.3 and Assumption 5.2.4.

Assumption 5.2.3. *The query optimizer intersects the maximum number of indexes in the enumeration space to execute the query.*

Assumption 5.2.4. *If any of the intersected indexes that are chosen to execute the query are removed, the query optimizer prioritizes scanning the whole collection over utilizing any other index to run the query.*

Figure 5.3 illustrates the IBG of such a scenario. In this case, the number of calls to the optimizer can be obtained as:

$$N_{min\_CI}(n, m, k) = \begin{cases} n * (k+1) & m > k \\ n * (m+1) & m \leq k, \end{cases} \tag{5.10i}$$

where k is the maximum number of indexes that the query optimizer is capable of intersecting with each other to execute a query. Under such condition, there is only one atomic configuration per query. Therefore, the number of atomic configurations is obtained by:

$$N_{min\_AI} = n. \tag{5.10j}$$

The *Best Case* complexity of the IBG with intersection scenario in Table 5.1 is calculated base on Equation 5.10i and 5.10j. With the Assumption 5.2.3 and consideration that the query optimizer might be able to intersect arbitrary number of indexes, it is safe to only consider the case of $m \leq k$ that results into a complexity of $O(mn)$. However, the case presented in Table 5.1 covers more general conditions.

On the other hand, Figure 5.4 illustrates the case of an index benefit graph for a maximum number of calls to the optimizer. The query optimizer can intersect up to a k indexes to run a query. The maximum number of calls happens not only by considering the Assumption 5.2.2 but also by considering Assumption 5.2.5 as:

Assumption 5.2.5. ***Worst case**: The query optimizer can run query q_i with each of the m indexes in the initial candidate index set E, and also with any of their intersections.*

Complexity / ***Algorithm***	***Best Case***	***Worst Case***
IBG without Intersection	O(n)	O(mn)
IBG with Intersection	O($min(k,m)n$)	O($n2^{max(k,m)}$)

Table 5.1: The complexity of the Index Benefit Graph algorithm in asymptotic notation. The algorithm evaluates different chosen configurations and extracts the atomic configurations for each query.

As shown in Figure 5.4, after building all of the indexes in the initial candidate set E, one call to the optimizer is made at *Level 1* to extract the best combination of maximum k number of indexes from among m available ones. For a case where $m \geq k$, there are $\frac{m!}{k!(m-k)!}$ possible combinations of k intersections that the query optimizer chooses only one with the best cost from them. Thus, *Level 1* contains only one call to the query optimizer even for the case of $m > k$.

As a result, *Level 2* contains k nodes. Each of these nodes requires one call to find the cost of the chosen set of indexes which sums up to k calls to the query optimizer at *Level 2*. What happens in the next level depends on the size of m in comparison to k. If the number of indexes in the initial candidate set is greater than k, then at the third level of Figure 5.4 instead of $\frac{k!}{(k-2)!2!}$ calls to the optimizer, k^2 calls should be made. This multiplication of k at each level will continue for $(m-k)$ levels. This pattern continues until one level after where exactly k indexes remain to intersect with each other (*Level 3* in Figure 5.4). At this level, each of k nodes results in $(k-1)$ evaluation call. However, not all of the resulted nodes are unique. The duplicated numbers also have a similar set of enumeration space. Therefore the number of calls reduces to their half. The number of unique choices can be obtained by the binomial coefficient with $\binom{k}{j}$, where j grows incrementally at each level.

Consecutively, for a workload of n unique queries, the number of calls to the optimizer does not only depend on n and m, but also on k. It can be obtained as:

$$N_{max\_CI}(n,m,k) = n * \left(k^{m-k} \sum_{j=0}^{k-1} \binom{k}{j} + 1 \right), \quad where \quad m > k, \ k > 1 \tag{5.10k}$$

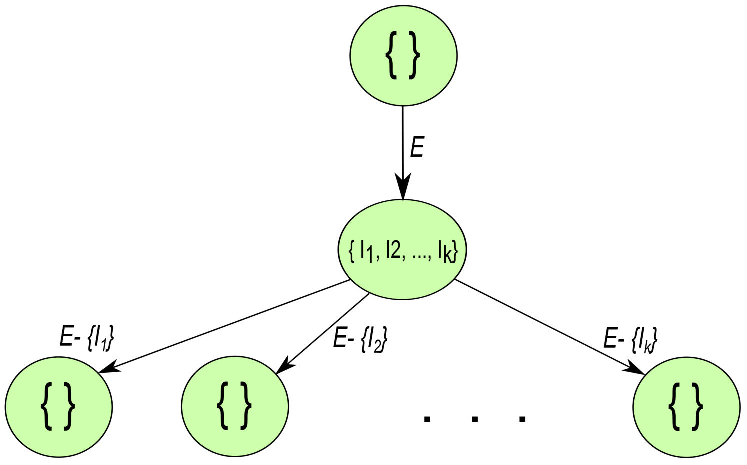


Figure 5.3: Best case scenario in case of the IBG with intersection. The assumption is by removing any of k intersected indexes, the cost of scanning the whole collection would be less than using the new configuration.

and N_{max_CI} is the number of calls to the optimizer in case of using index intersection for $k \geq 1$. Note that this equation leverages to Equation 5.10g for the case of $k = 1$, only if the number of choices at *Level 1* is considered as $(N_{max\_CI} * \frac{m!}{k!(m-k)!}) - 1$.

If m is less than the number of indexes that the optimizer can intersect, then N_{max_CI} does not depend on k anymore. It can be obtained by Equation 5.10l. This formula is formed with the assumption that all indexes in the candidate index set are intersected at first level to answer the query and the optimizer can even run the query with any of them:

$$N_{max\_CI}(n, m) = n * \left(\sum_{j=0}^{m-1} \binom{m}{j} + 1 \right) \quad \text{where } m \leq k, \;\; k > 1. \tag{5.10l}$$

Example 5.2.7. An example of $m < k$ with ten indexes in the enumeration space $m = 10$ and five individual queries $n = 5$ is: $N_{max\_CI} = 5,110$.

In any case, this number can be enormous. Nonetheless, this number is way greater than the usual calls to the optimizer (see Section 6.3.2). The reason is that usually, there are few queries in each workload that require index intersection. Even for such queries often,

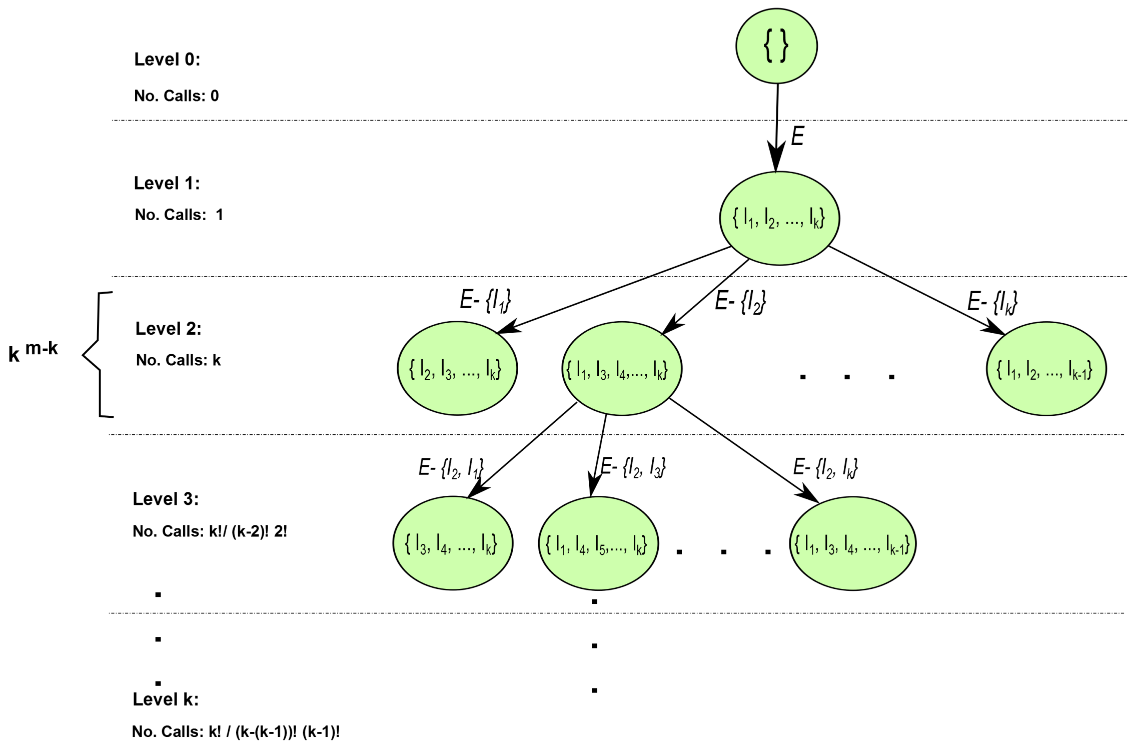


Figure 5.4: IBG structure for the case of maximum number of calls to the query optimizer when the query optimizer is capable of intersecting k indexes to respond to a query and $m > k$.

by removing few of the chosen indexes from the available index set, the query optimizer prioritizes a full collection scan (see the example of Figure 5.2). All of these conditions results in having much lower number of calls to the query optimizer than what is obtained by any of Equation 5.10k or 5.10l.

Based on the fact that only one node in each level is chosen as the atomic configuration and the level $k+1$ contains only empty set representative nodes, the maximum number of atomic configurations with index intersection is obtained by:

$$N_{max\_AI}(n,k) = \begin{cases} n * k & m > k \\ n * m & m \leq k. \end{cases} \tag{5.10m}$$

Equation 5.10k, 5.10l and 5.10m are considered to deduct the *Worst Case* of the IBG algorithm with intersection scenario in Table 5.1. This asymptotic notation is calculated by considering the worst case for the growth of the binomial coefficient [131].

Example 5.2.8. Although the Assumption 5.2.1 is expected to be right for a well-behaved optimizer, it is shown in Figure 5.2 that not all query optimizers behave by this property. Figure 5.2 contains some examples of the situation where the same set of indexes are chosen to run the query under a different subset of available indexes. Consider the node $\{I_5, I_7\}$ that is duplicated in two places. First glance shows that this node is chosen under different sets of available indexes. However, following this node in each of the branches displays that the property 5.2.1 is not met by the query processor of the database under study. The node $\{I_4, I_5\}$ in the right side branch is chosen before node $\{I_5, I_7\}$. But in the other branch, the order of picking these two sets is vice versa. This situation can happen under the circumstances that the cost of running the query under these two configurations is almost the same and the query optimizer uses the first index in its catalog list. Such a situation makes an example of a tie between these two nodes at level 4 of the IBG. If none of the costs of the nodes can break the tie, the $\{I_4, I_5\}$ that is repeated the most in the IBG will be chosen as the atomic configuration at that level. Although in such a situation the number of calls to the optimizer utilizing the index benefit graph might be more than the situation that meets the property 5.2.1, still the number of calls to the query optimizer is restricted. Besides, using the IBG strategy, especially in such case with many similar nodes, helps to ensure that all necessary configurations are evaluated.

In this section, a solution to minimizing the large search space of the enumeration method to optimize the objective function is presented. This solution is based on the concept of atomic configurations and the fact that the cost of all other configurations can be derived from atomic configurations. To find these atomic configurations, a method called Index Benefit Graph (IBG) is introduced. Utilizing the index benefit graph also helps to make sure that indexes that might not be the immediate choice of the optimizer to run a query (such as merged indexes) also get a chance to be evaluated. The process of exploiting the IBG for the case that the query optimizer is capable of intersecting indexes is introduced and discussed.

It is shown that the number of necessary calls to the query optimizer to evaluate the appropriate configurations to build IBG, even for worst cases is much more restricted than evaluating all configurations. Although in reality the number of calls to the query optimizer is restricted (see Section 6.3.2), the study of the worst case scenario with index intersection shows that they can potentially get very large. Such large number of evaluation calls can affect the performance of the in-production database system. Therefore, the necessity to introduce a virtual environment where the communication with the query optimizer has a minimum effect on the fulfillment of the in-production system rises. Such a environment is introduced in Section 4.2.

5.3 Summary

To properly solve the complex problem of selecting proper set of indexes with consideration of all of the relevant factors a thorough mathematical model is required. This chapter contains the mathematical formulation of AIRs objective function. The maximization of this function results in finding an optimal configuration that its profit is maximum for the particular workload under the given storage limitation.

Additionally, to assist the efficiency of the enumeration technique, it is important to reduce the search space of the problem in the best way possible. The usage of atomic configurations reduces the number of configurations that should be traversed by the enumeration technique. Therefore, the Index Benefit Graph algorithm is developed to extract these atomic configurations.

The contributions of this chapter can be summarized as the following:

1. Definition of the objective function considering many parameters including:
   - negative influence of each index for update operations
   - ratio of read to write
   - consideration of the influence of each attribute and its cardinality
2. Formulation of this objective function in Integer Linear Programming to ensure that an optimal solution will eventually be obtained
3. Development and analysis of the complexity of the Index Benefit Graph algorithm for databases with index intersection capability

In the next chapter, the evaluation studies of the solutions developed in each segment of AIRs are presented.

Chapter 6

Performance Studies

The issues regarding the lack of standards and well-defined benchmarks for NoSQL databases in general, and document-based databases in particular, are discussed in this chapter. Additionally, the performance evaluation of individual solutions developed in different segments of AIRs is presented.

Section 6.1 contains a brief discussion on the difficulties of evaluating the performance of document-based database related systems, already stated in the Problem 2.3.6. Also, the specific research questions that lead to the development of a workload generator tool and its characteristics are discussed in this section. The basic metric and baseline to specifically evaluate the index recommendation systems are explained. The continuation of the discussion on the lack of standard benchmarks results in the definition of certain synthetic workloads. Then, Section 6.2 introduces some of the workloads chosen from the real-world meteorological applications. These real-world workloads are exploited for various performance studies in Section 6.3. The results of the evaluation of different solutions of AIRs and its overall performance are presented.

Parts of the following text and figures are published in [6].

6.1 Benchmarking Components and Workload Generator for Document-Based Databases Performance Analysis

To measure the performance of any database-related systems in comparison to similar systems, well-defined and standardized benchmarks are needed. There are some traditionally well-established applications associated with relational databases. Correspondingly, there are standard benchmark workloads defined that are widely used to compare the performance of these databases and their related systems to each other. However, considering the fundamental differences between relational and non-relational databases, there is no uniformly defined set of benchmarks to evaluate the performance of both of these database systems. Additionally, the defined metrics of these macro-benchmarks is intended to assess the overall performance of the database system, not specific database-related systems such as index recommendation systems. The rest of this section contains the discussion about and the definition of the necessary baseline, metrics, and workloads to evaluate the performance of the index recommendation system.

Each benchmark consists of three essential components: 1) ***Baseline definition***, 2) ***Evaluation Metric***, and 3) ***the Database/Workload*** [113]. A series of well-established benchmarks are available to evaluate the overall performance of relational databases regarding their various well-defined applications. The evaluation metric for these benchmarks that aim the performance of the whole database system is normally *throughput*. The Transaction Processing Performance Council (TPC) [52] provides a series of such application benchmarks defined by a consortium of vendors. In addition to traditional relational applications such as Online Transaction Processing (OLTP) or Online Analytical Processing (OLAP) [132], there are even TPC benchmarks introduced to cover newer applications such as support systems and web commerce [112]. The web commerce application can be considered as a typical application for document-based databases as well.

However, due to the major differences in the data models and query languages of the document-based databases and relational ones, these TPC benchmarks are not directly applicable to evaluate the performance of document-based related systems. Also, a mapping from the normalized relational model of the TPC benchmarks to the denormalized model of document-based databases is complex and despite the efforts, a proper mapping between these models is missing [120]. In general, the design of such general-purposing benchmarks requires federated work to define standards which is out of the scope of this thesis.

Even if there were already such generalized benchmark to assess the performance of relational and non-relational database, these macro-benchmarks would not be sufficient to properly evaluate database-related systems with specific functionalities such as index recommendation ones. Despite the extensive research on the index recommendation systems, there is a very limited number of studies on how to properly benchmark these systems [111], [113], [92]. Therefore, as part of my research, it is necessary to define an appropriate evaluation baseline, metric and workload/databases to assess the performance of the designed index recommendation solutions.

As discussed in Section 3.7, Consens et al. in [111] proposed to use all of the indexes related to single-attribute queries as the baseline to compare the performance. However, N. Bruno based on his research presented in [113] argues that such baseline is not a good baseline in general and in specific for decision support workload that demands aggregation and filtration of many attributes. Based on this research and also to be aligned with the research conducted in [92], the baseline for the evaluations of this thesis is considered to be as defined in Definition 6.1.1:

Definition 6.1.1. ***Base Configuration as Baseline***: The *Base Configuration* is the configuration containing no index other than the mandatory indexes of the system, e.g. index on primary keys. This configuration is independent of the workload and can be easily reproduced. Therefore, it can be defined as the *baseline* for the index recommendation system performance. This configuration occupies the least amount of storage and has the worst cost for read-only workloads.

The next crucial component of a benchmark or any evaluation method are *Evaluation Metrics.* Typically, in most of the studies conducted on index recommendation systems a metric named *Percentage Improvement* is utilized to determine the quality of the results. The definition of *Percentage Improvement* is presented in Definition 6.1.2. It is a numerical measure to compare the quality of the recommended optimal solution to the determined baseline.

Definition 6.1.2. ***Percentage Improvement***:
For each given baseline configuration C_0, the *percentage improvement* of a recommended configuration C_i of the workload W is defined as:

$$percentage\ improvement = (1 - \frac{cost(C_i, W)}{cost(C_0, W)}) * 100. \tag{5.10a}$$

Negative values for percentage improvement indicate that the recommended configuration is less efficient than the baseline configuration [1, Chapter 12].

Although this single number can be utilized conveniently to compare the quality of recommended solutions, Consens et al. argued in [111] that a more precise metric that can give more detailed information about the performance of index recommendation systems is required. Therefore, they proposed an additional metric M in the way that for a given workload W and a given configuration C the quality metric $M_{C,W}$ is defined as the ratio of the number of queries of workload W that can be executed in the input time t, and the total number of queries in the workload $|W|$:

$$M_{C,W}(t) = \frac{|run\_time(q)| \leq t}{|W|}. \tag{5.10b}$$

According to the definition of the M metric, the $run\_time(q)$ is the actual run-time of each query. Utilization of the actual execution time of a workload in the metric can be beneficiary when the evaluation of the whole database performance, including e.g. the query optimizer, query processor and even the underlying operating system is intended [113]. However, when the target is to isolate the index recommendation system for assessment, the run-time of the whole queries is not the best indicator. Instead, the cost of each query estimated by the query optimizer can be utilized.

Nevertheless, the M metric reveals information about the quality of a configuration for the whole workload, but it does not give any information about its quality regarding every single query. A complementary evaluation metric I can be defined as in Definition 6.1.3 to disclose the quality of the recommended optimal configuration for each query in the workload regarding the maintenance and storage cost.

Definition 6.1.3. *I* ***Quality Metric***:
For any query q_i of workload W with any two given configurations C_1 and C_2, a *difference value* $v_i = cost(q_i, C_2) - cost(q_i, C_1)$ can be calculated where $cost(q_i, C_1)$ is the number of scanned documents for q_i in presence of C_1. The v_i shows the gain of C_1 over C_2. The

evaluation metric I_{C_1,C_2} is a set of all *difference values* of the queries in the workload with the two given configurations C_1 and C_2. For a workload with n queries the *I Quality Metric* is presented as:

$$I_{C_1,C_2}(W) = \{v_1, \cdots, v_n\}. \tag{5.10c}$$

A positive *difference value* v_i indicates that C_1 was a better configuration for running q_i than C_2 and vice versa. If configuration C_2 be replaced by the baseline configuration C_0, then the different value presents the gain of C_1 to the baseline configuration. The I metric is used then to investigate the quality of the proposed optimal configuration for a given workload.

The third component of each evaluation method (benchmark) is the definition of the *database/workload*. In [113], Bruno argues that any beneficiary *database/workload* of an evaluation method should be build from at least one of these three 'buckets':

- Micro-benchmarks: assess various abilities of the underlying database,
- Synthetic benchmarks: contain complex workloads to exercise the entire capabilities of the database,
- Real benchmarks: cover the possible delicate scenarios that might be neglected in the past two buckets.

To evaluate the proposed solution to the index recommendation system on *real benchmarks*, few of the datasets and workloads of the meteorological projects are used which are introduced in Section 2.1. The chosen real database and their corresponding workloads are discussed in more details in Section 6.2.

Additionally, to enhance my research with studying the effect of different parameters on the index recommendation process, e.g. the effect of having indexes on attributes with different levels of nested documents or different ratio of read-to-write, and to extract the parameters explained in Section 4.1, a combination of different micro- and synthetic queries should be defined and investigated. Therefore, based on the criteria under study, a flexible tool to produce various synthetic datasets and workload solutions called "Not only Workload generator" (NoWog) is developed.

To be able to cover the diversity of query languages of different databases, a unified grammar is designed for NoWog that is presented in Appendix A. This grammar provides a means for users to flexibly define a set of queries with arbitrary attributes in a workload. This grammar facilitates the specification of the distribution and frequency of execution of each query in a given time period. It also supports various operations that combine database entities such as *JOINs* or nested documents. NoWog maps this set of rules in its unified grammar to the query language of the targeted back-end database. The details about the functionality of NoWog are discussed and published in [6].

NoWog provides sets of predefined scenarios that correspond to some typical characteristics of applications such as being read-mostly, or update-intensive. However, the definition of standard benchmark workloads demands federated work to establish common criteria. Such a definition of a standard database/workload is out of the scope of my research and this thesis. Instead, NoWog is utilized to generate a series of synthetic workloads to investigate the effect of various parameters and evaluate the performance of the AIRs solutions. Some of the examples of these datasets and workloads were used in Section 4.4 to study the effect of various data types on the storage size. Many other datasets and workloads are generate by this tool to study various aspects of AIRs' performance.

By specifying a definition for each of the three essential components of a benchmark, a testbed framework is prepared to check out the performance of the target system. The target is to estimate the performance of solutions proposed in the Adaptive Index Recommendation system.

6.2 Real Meteorological Data Sources

As discussed in Section 2.1, the real-world applications in hand come from specific meteorological use cases. Two of these applications (*satellite* and *GLORIA*) are briefly introduced. Their corresponding general workloads are depicted in Figure 2.2 and Figure 2.4, respectively. In this section, two specific workloads of each of these applications are described in more detail. Each of these workloads is intended for one collection. These workloads are then used as real-world scenarios to test AIRs.

Table 6.1 contains statistics about four workloads each of which is issued against one collection in the application database. Workload WKL_1 and WKL_2 belong to the *satellite* application, whereas workload WKL_3 and WKL_4 are taken from the *GLORIA* application (see Section 2.1). For each workload, the number of documents in their corresponding collection is given in Table 6.1. The total number of queries as well as the number of individual operations (i.e. *search*, *update* and *insert*) to each collection are also indicated in this table. As Figure 2.2a and Figure 2.4a suggest none of these workloads contain any *delete* operation. These snapshots of the datasets and their corresponding workloads are used to assess the performance of AIRs in different evaluation scenarios.

The collection related to workload WKL_1 originally contains documents with information about the position of the satellite, e.g. geolocation, in the form of nested documents containing latitude and longitude, and the environmental factors, e.g. sun elevation at the time of each measurement. The workload related to this collection is an update-intensive workload with some *search* queries. A typical *search* query in this workload is shown in Example 6.2.1. Each *update* operation, typically, searches for a document with specific `version_number` and `geo_id`. Then, it sets new attributes with mostly large multi-array values for the found document. Example 6.2.2 shows the structure of a typical *update* operation on this collection.

Example 6.2.1. Typical *search* query of WKL_1:
db.coll.**READ**($\{TB < zpt\_time < TE \wedge 0 < sun\_elevation \wedge$
$geo\_id = GID \wedge version\_number = VN\}$).**SORT**({zpt_time})

Example 6.2.2. Typical *update* query of WKL_1:
db.coll.**UPDATE**({ $version\_number = VN \wedge geo\_id = GID$},
{ $tangents = [T1, T2, \cdots]$, $cloud\_index = [[N1, N2, \cdots], [M1, M2, \cdots], \cdots]$,
$tang\_time = [\{date : D1\}, \{date : D2\}, \cdots]$ })

The related collection to workload WKL_2 contains simpler documents with information about the satellite position of each measurement at any recorded time. WKL_2 is a read-only workload. Each *search* query inquiries for the documents containing time and longitude (stored in nested documents) in a specific range. An example of such a query is given below in Example 6.2.3.

WKL_3 contains very simple *search* queries based on only one attribute as shown in Example 6.2.4. The *_id* attribute in this query is actually the primary key defined in the database that is by default indexes. Therefore, AIRs only considers the other single attribute of this query. However, WKL_3 is a write-intensive workload with many insertions of documents into its corresponding collection. Unlike the simple *search* queries of this workload, the inserted documents are relatively large and contain many attributes with a nested document structure.

The last real-world workload is WKL_4 from the *GLORIA* application. This workload is like WKL_1, an update-intensive workload. However, this workload contains about twice as many *search* queries as *update* operations which are issued on a relatively small set of documents in the collection. The typical *search* query of this workload includes an inquiry only on one attribute, similar to Example 6.2.4. An example of its typical update query is displayed in Example 6.2.5. This operation replaces the value of the already existing attributes.

Example 6.2.3. Typical *search* query of WKL_2:
db.coll.**READ**({$BLONG < loc.coordinate.0 < ELONG \wedge BT < time < ET$}).**SORT**({$time$})

Example 6.2.4. Typical *search* query of WKL_3:
db.coll.**READ**({$queue = QID$, $\_id = PRKEY$})

Example 6.2.5. Typical *update* query of WKL_4:
db.coll.**UPDATE**({$queue = QID \wedge routing\_key = RID \wedge exchange = EID$},
{$queue = new\_QID, routing\_key = new\_RID, exchange = new\_EID$})

Workload	***#Documents***	***#Total Queries***	***#Search***	***#Update***	***#Insert***
WKL_1	4,707,583	2,787	195	2,592	0
WKL_2	1,200,000	2,735	2,735	0	0
WKL_3	285,025	124,898	65,137	0	47,142
WKL_4	123	17,397	8,611	4,398	0

Table 6.1: Number of operations in the real workloads of meteorological applications and their corresponding dataset size.

6.3 Evaluations with Real Data Sources

This section represents evaluations of some of the solutions developed through out this thesis as well as the overall performance of AIRs results. For this purpose an experimental environment is set up utilizing a server with two Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz with sixteen physical cores. However, running the AIRs is limited to use a single core at each run-time for evaluating a workload. Accordingly, the storage limitation is considered as half of the available memory size to each processor. This choice for the storage limitation is made based on two regards: 1) the indexes are most beneficiary if they can reside in memory, 2) at least half of the memory should be dedicated to retrieve the corresponding data into memory. In this environment 128 GB memory is available that half of it (64 GB) is considered to be the storage constraint.

For the evaluation of the effectiveness an instance of MongoDB v3.2 [133], which is a well-known document-based database, is installed on this server. For these experiments, the AIRs is also running on the same server as the database is running. Although this is not the recommended set-up of the AIRs, it is done to prove the functionality of it even in such a set up scenario.

All of the evaluation studies presented in the following of this section are run in this environment.

6.3.1 Assessment of Effectiveness of Candidate Indexes Exploration Strategies

The number of unique attributes in each collection indicates the number of possible combination of indexes that can be built on that collection. The strategies explained in Section 4.3 are applied by the AIRs to extract the number of the prime candidates based on the workload unique queries.

Table 6.2 exhibits the results of the application assessment of these strategies on the real meteorological datasets and workloads as described in Table 6.1. This table contains the maximum number of unique attributes ($\#Unique\ Atts$) in the corresponding dataset of each workload. Any individual attribute of a sub-document is counted as a unique attribute.

Example 6.3.1. The document in Code 6.1 contains two individual attributes: "geo_loc.geo_lon" and "geo_loc.geo_lat". Thus ($\#Unique\ Atts = 2$).

Code 6.1: Example of individual attributes in a nested documents.

```
{"geo_loc":
           {"geo_lon": 42.3021,
            "geo_lat": 66.0637
            }
 }
```

The number of all possible index combinations (*#All Combinations*) is calculated with Equation 4.1. This number is obtained by assuming that there are only two index types available for each attribute ($\theta = 2$). The number of unique queries (*#Unique Queries*) contains the sum of unique search and update operations in the workload. The column (*#Frequent-Long Items*) includes the number of candidate indexes extracted from the frequent and long queries (see Definition 4.3.4 and 4.3.2). The number of candidate indexes after application of the merge strategy is demonstrated in column (*#Enum Space*), which generates the final set of candidates indexes of the *Query Analyzer* of AIRs (see Figure 4.3).

The support threshold configured for the Frequent Itemsets, in this case, is 0.2. According to the statistics of these workloads shown in Table 6.1 and 6.2, the number of unique queries of these workloads is much smaller than the number of issued queries. Therefore, replacement of the chosen low support threshold with a higher one would not change that much in the results of the *Frequent-Long Items* for these workloads.

The results of this table indicate how large the number of all possible index combinations, even for a modest number unique attributes, can get (over 3 Billion for only 12 attributes). Comparison of some possible combinations and the number of candidates in the final enumeration space shows the necessity of applying strategies to reduce the search space of candidate indexes.

A comparison of the total number of queries of each workload in Table 6.1 with the final set of candidate indexes in Table 6.2 reveals that the candidate selection strategies of AIRs are independent of total queries in the workload. These strategies are only dependent on the number of unique queries in the workload and their complexities. The complexity of the query increases the number of attributes in its search-predicate, having range or equality predicates and attributes in the sort-clause.

Consider workload *WKL_2* and *WKL_4* as examples which the former consists of only one unique query and the latter contains two unique queries. The only query of *WKL_2* is described in Example 6.2.1. The output of the enumeration space for this query is a set as shown in Code 6.2. Workload *WKL_4* includes a simple single-attribute search query and an update query as described in Example 6.2.5. There are fifteen candidate indexes in the enumeration space of this query. Some of these candidates are shown in Code 6.3. All candidates are either frequent or long. Neither of the final candidates is constructed by

Workload	***#Unique Atts***	***#All Combinations***	***#Unique Queries***	***#Frequent-Long Items***	***#Enum Space***
WKL_1	12	$> 3.2 \times 10^{10}$	7	35	46
WKL_2	4	632	1	3	3
WKL_3	2	12	1	1	1
WKL_4	4	632	2	15	15

Table 6.2: The effectiveness of the Frequent-Long itemsets and merging strategies in reducing the number of possible attribute combinations to build candidate indexes. The results of frequent itemsets are obtained by a *support* = 0.2.

merging of attributes.

Code 6.2: Enumeration space output for the unique query of WKL_2 which contains three candidate indexes out of the frequent attributes of the query. Since they are all extracted from one query, the frequency number is similar for all of them and is equal to number of total queries of this workload.

```
{ (["loc.coordinates.0"], 2735),
  (["time"], 2735),
  (["loc.coordinates.0", "time"], 2735) }
```

Code 6.3: Some of the candidate indexes in the Enumeration space of WKL_4 which contains fifteen members. Each tuple contains the candidate index and the frequency of appearance of that pattern in the workload. The order of the keys designed by prioratizing search-predicates over sort-clause, quality over range predicate and the cardinality of the attributes in the dataset.

```
{ (["exchange"], 13009), (["routing_key"], 4398),
  (["queue"], 4398), (["routing_key", "exchange"], 4398),
  (["exchange", "queue"], 4398),
  (["routing_key", "queue"], 4398),
  (["routing_key", "exchange", "queue"], 4398), ...}
```

Workload *WKL_1* is the only real-world meteorological example under which the merge module of the *Query Analyzer* adds more candidates to shape the final *Enumeration Space*.

A comparison of the number of indexes in the *Enumeration Space* after applying the *Frequent-Long* and merging strategies to all possible combinations shows a reduction of at least 12 times and at most about 7.0×10^8 in the number of candidates.

6.3.2 Measurement of IBG Algorithm Effectiveness

The run-time complexity of the IBG algorithm is discussed in Section 5.2.2. According to that asymptotic behavioural analysis, the IBG algorithm in the worst-case can grow exponentially. However, the claim is that in real cases, the solution to the IBG algorithm is feasible.

Table 6.3 presents the evaluation of the effectiveness of the IBG algorithm for the real meteorological workloads of Section 6.2. The *#All Configs* column of Table 6.3 represents all of the possible configurations that can be build for each workload. The results in this column are calculated by Equation 5.10e given the number of indexes in the *Enumeration Space* of each workload listed in Table 6.2. Despite the reduction in the number of candidate indexes that by extracting the attributes of the *Frequent* and *Long* queries in the previous step, the number of configurations that can be built by these candidate indexes can be very large. The growth of the number of configurations that should be evaluated by the enumeration method is exponential. Compare the *#All Configs* for WKL_4 with only 15 candidate indexes to WKL_1 with 46 candidates. Also, collecting the required cost of running the queries with each of these configurations enforces many additional calls to the query optimizer. The load that this large number imposes on the whole system is unacceptable.

The *#Atomic Configs* column in Table 6.3 shows the number of atomic configurations extracted by the IBG algorithm. These are the atomic configurations for the whole workload. The *#Call to Optimizer* displays the number of calls that are made to the query optimizer to extract the necessary cost evaluations for all of the atomic configurations in the workload. Correspondingly, the *Avg. #Calls* is the average number of calls and the *Max. #Calls* shows the maximum number of calls that is made per unique query.

The comparison of the number of atomic configurations extracted by the IBG algorithm to all possible configurations of the enumeration space shows the effectiveness in reducing the search space for the optimization method as discussed in Section 5.1.2. The amount of the overall calls to the query optimizer for the whole workload also suggests that in practice, the run-time of the IBG algorithm is feasible. According to these evaluations, the highest number of maximum calls per query to the query optimizer happens in the case of one complex sample query of WKL_1 that calls thirty-three times to evaluate a query. The average number of calls to the query optimizer per query is not large. These results suggest that in real-world examples, the number of calls to the query optimizer is not even close to the worst case scenarios of IBG algorithm discussed in Section 5.2.2 (see Table 5.1).

The outcomes of the assessment of the IBG algorithm for the real meteorological workloads demonstrate its feasibility to extract the atomic configurations and their corresponding cost estimations even for workloads with rather complex queries.

6.3.3 Recommended Configuration Optimality Evaluation

The optimality of the overall proposed solution of the index recommendation system can be measured with metrics that are introduced in Section 6.1. In the following of this section, the optimality of the recommended solution by AIRs is investigated for each of the Meteorological workloads introduced in Table 6.2.

For the first meteorological workload (WKL_1), AIRs recommends a configuration that consists of three indexes as the optimal solution for this workload. Although this workload

Workload	***#All Configs***	***#Atomic Configs***	***#Call to Optimizer***	***Avg. #Calls***	***Max. #Calls***
WKL_1	$7.036\,874\,417\,77 \times 10^{13}$	73	81	≈12	33
WKL_2	6	3	3	3	3
WKL_3	1	1	1	1	1
WKL_4	32,766	16	16	8	15

Table 6.3: The effectiveness of the IBG algorithm is in calling feasible number of times to the query optimizer, despite its high theoretical worst-case. It also significantly reduces the number of configurations that should be evaluated in the next optimization step.

contains about twelve times more update operations than read operations, still these indexes are proposed to be created.

To show that these indexes are actually improving the execution of the workload, several measurements are considered. Figure 6.1 illustrates the run-time measurement of this workload. The run-time is measured once without any index in the system to make the baseline, and the other time with having the indexes from the recommended optimal configuration. The run-time of the intended workload for each of these cases is measured five times. The standard deviation of these measurements is presented as the error bar of the run-time measurements.

The overall *Percentage Improvement* for this workload with the given recommended configuration is 21.05 %. Figure 6.2 illustrates the *difference values* of the quality metric. It shows the recommended set of indexes for five chosen queries of workload WKL_1. Each of these queries happen to be representative of a unique query in this workload. According to the results in this figure, despite high improvement in cost of running two of these queries, the recommended set of indexes are not helpful to improve execution cost of three other queries. This can cause the decreasing overall *Percentage Improvement* with recommended set of indexes for this workload.

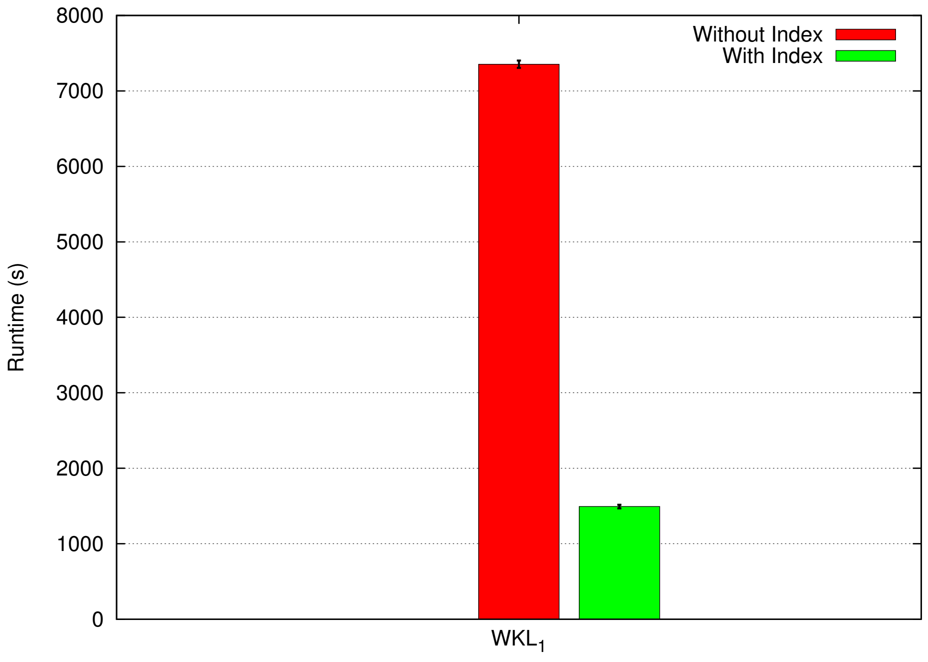


Figure 6.1: WKL_1: Run-time measurement of WKL_1 which includes about twelve times more update operations than read operations. The run-time is measured once without any index (as baseline measurement) and another time with the recommended indexes.

Surely, adding more indexes to support the other queries of the workload can improve the *Percentage Improvement*. However, adding more indexes would enforce more maintenance

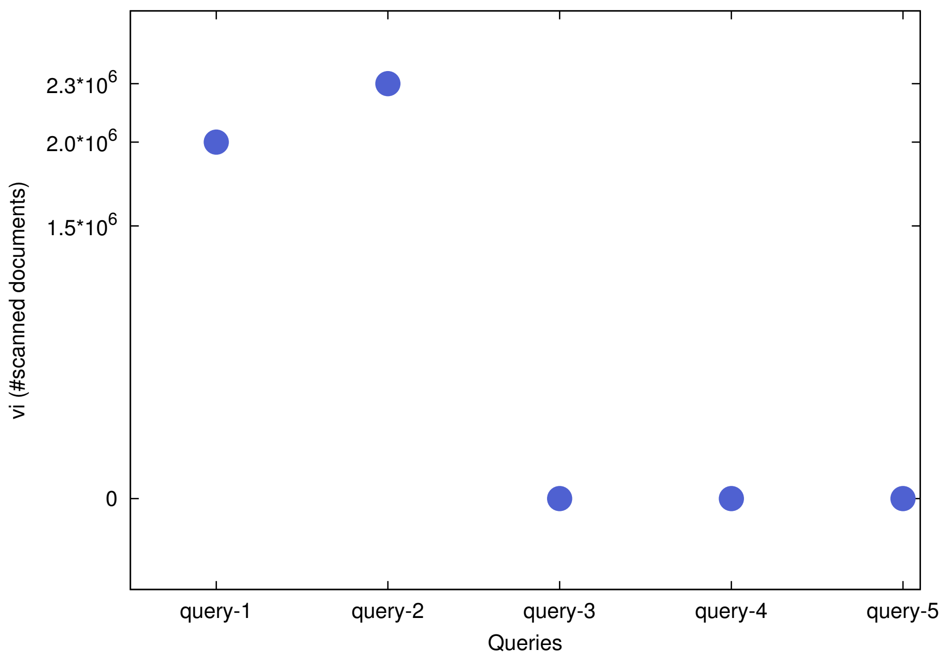


Figure 6.2: Different values of five chosen queries of WKL_1. The chosen queries are each a representative queries of a unique query in this workload. These results show that the recommended set of indexes by AIRs improve the running cost of two of the queries, but are not used to run the other three.

and storage cost. Since the measured run-time only includes the required time for the database to search for the relevant documents and transfer the intended data, the necessary time of updating the indexes is not included in the run-time. Therefore, this cost is not directly visible in the run-time measurement.

Therefore, with the available metrics, it is still difficult to show the optimality of the recommended solution by the AIRs for the workloads with update operation. Still demonstration of this optimality is easily feasible for a read-only workload.

For a read-only workload such as WKL_2, the optimality of each configuration depends only on the benefit that it has for search queries according to Equation 5.8. The only constraint is the size of the indexes. Since the definition of benefit is based on the number of documents that should be scanned to find the relevant document, it has a direct relation to the time required to find the documents. Therefore, run-time measurement can be representative of the optimality of the solutions for such read-only workloads.

As shown in Table 6.2, the enumeration space of this workload contains only three candidate indexes that are presented in Code 6.2. In the experimental environment with not so tight storage limitations, AIRs recommends a configuration containing one single index on the *loc.coordinates* and the compound index.

The run-time measurement of the second workload (WKL_2) is depicted in Figure 6.3.

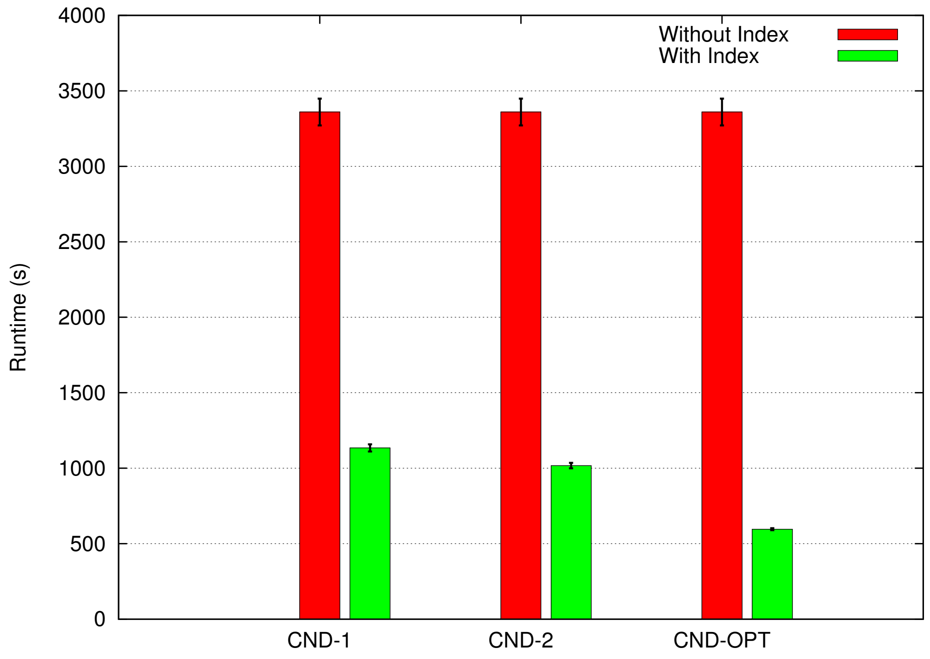


Figure 6.3: WKL_2: Run-time measurement of optimal and sub-optimal candidates of WKL_2. Since this is a read-only workload, its run-time improvement is a good indicator for the optimality of the solution. The run-time improvement utilizing two sub-optimal configurations CND-1 and CND-2 is less than the one with optimal recommended configuration CND-OPT.

Figure 6.3 not only shows the run-time comparison of the base configuration (no indexes) to the optimal recommended configuration (CND-OPT), but also the run-time of the other two candidate configurations with single indexes of the enumeration space that are indicated with CND-1 (for *time* index) and CND-2 (for *loc.coordinates* index). This comparison clearly shows the optimality of the benefit of the recommended configuration for the workload. The *Percentage Improvement* for this workload with the optimal recommended configuration is 91,61%.

Despite the rather large number of insert operations, AIRs recommends to create the single possible configuration with single attribute index. Figure 6.4 shows the run-time improvement of this workload in the presence of the recommended optimal configuration *CND-OPT*. The run-time only reflects the required time to search and retrieve data, but it does not reflect necessarly time to insert documents.

Even without considering the time for insert operations, the run-time improvement of this workload is high. The *Percentage Improvement* of the WKL_3 with consideration of only read queries is equal to 99,99%. This is surprisingly more than the run-time improvement and *Percentage Improvement* of WKL_2 which is a read-only workload. The reason is that workload WKL_2 consists of very large range queries that require retrieval and trans-

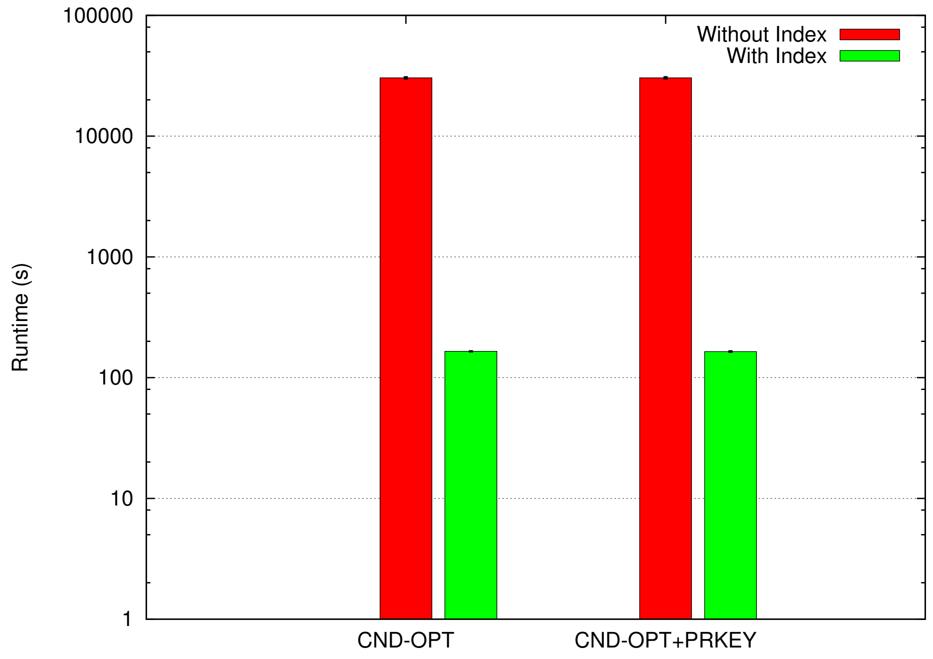


Figure 6.4: WKL_3: Run-time measurement of optimal solution and its combination with the primary key for WKL_3 shown in logarithmic scale. Since the read queries of this workload contain equality predicates, the run-time improvement with introducing the optimal index is very high. The comparison of the single attribute recommended index CND-OPT and its combination with the primary key does not show any difference in the run-time improvement. This is result was expected because of the capability of the database to intersect the indexes and the fact that the primary index is by default indexed.

fers of lots of data. On the contrary, workload WKL_3 contains equality queries that with utilization of the index can pin point to the exact document.

Figure 6.4 also depicts the run-time improvement with the assumption of creating a compound index exploiting both the optimal index attribute and the primary key attribute. This comparison is done, to investigate the correctness of the AIRs design decision to eliminate the primary key from being recommended. This investigation seems necessary, because as shown in Example 6.2.4, the typical query of WKL_3 contains this primary key that is by default indexed in the database. Comparison of the run-time with utilization of this compound index and the single optimal index shows no difference. With the capability of the database in intersecting existing indexes and having the primary key indexed by default, this result was expected.

Workload WKL_4 has interesting characteristics. It consists of a mix of read and update operation and a rather small corresponding dataset as represented in Table 6.1. The update queries are frequent rewrites values of various existing attributes as presented in Example 6.2.5.

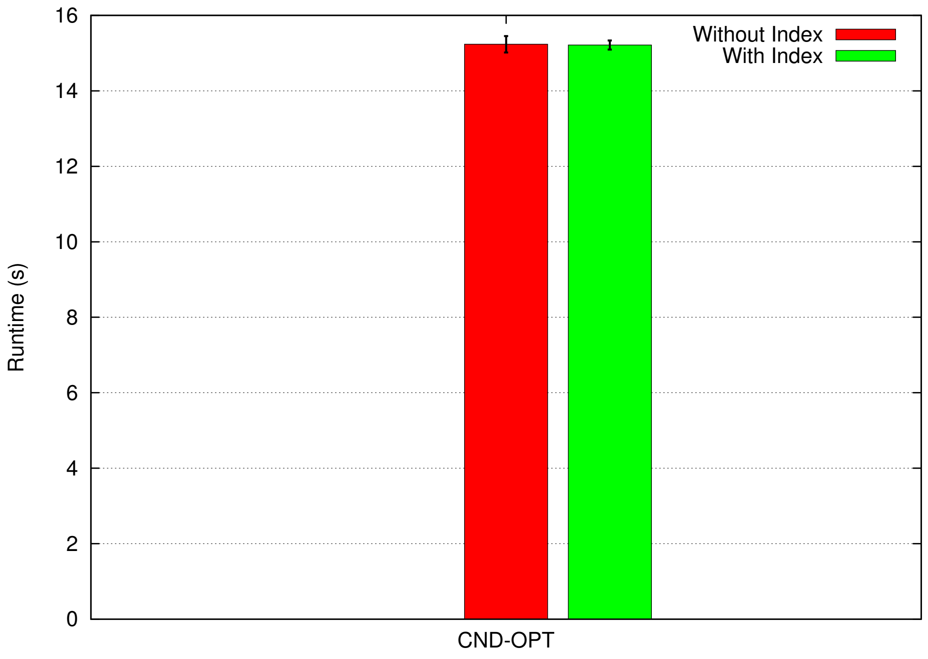


Figure 6.5: WKL_4: Run-time measurement of the optimal solution for WKL_4. Since this rather large workload of read and update operations are issued on a small dataset, even its run-time without any index is small. Therefore, to prevent from introducing additional maintenance costs due to large number of updates in the workload, AIRs recommend to run this workload without any index.

Despite various possibilities of candidate indexes in the enumeration space of this workload as mentioned in Table 6.2, the optimal solution recommended by AIRs is to run this workload without any index. The reason is that the corresponding dataset of this workload contains a fairly small number of documents. Since the database engines are powerful enough to search through hundreds of documents quickly, introducing an index for such a small dataset does not improve search queries significantly. This is also noticeable from the small required run-time of this rather large workload even without having any index as shown in Figure 6.5. Since this workload has a heavy amount of update, introducing any index would just create more maintenance cost for the workload.

The results of this part show the optimality of the configuration recommended by AIRs as well as the improvement in running costs of each of these meteorological workloads causes by this solution .

6.4 Summary

To enable evaluation of developed solutions of AIRs, the proper baseline and metrics are defined. Accordingly, some of the real meteorological workloads and their corresponding datasets are chosen to evaluate the performance of these solutions.

As the first solution, the effectiveness of the *Frequent-Long* strategy in extraction of proper indexes is evaluated. The results show a reduction in the number of enumeration space candidates for even up to 7.0×10^8 times.

The evaluation of IBG algorithm shows its feasibility and applicability in real-world applications. Also, with the help of the developed evaluation framework, the optimality of the proposed solutions of AIRs for real meteorological applications are investigated.

Contributions of this chapter can be listed as the following:

- Introduction of proper metrics to enable direct performance evaluation of index recommendation system,

- Development of a generic workload generator to produce synthetic workloads for various evaluation purposes.

Chapter 7

Conclusion

Database indexes are the common structures to enable fast access to the data and thus speed up the process of search and retrieval of data. However, the materialization of each index enforces maintenance cost of updating them in case of any write to the system. The selection of proper set of indexes for a workload is non-trivial. The focus of the work presented in this thesis is to find an efficient approach and formulation to solve this NP-complete problem.

The motivation to pursue the research presented in this thesis originated from the management of databases for several meteorological applications. Their datasets consist of large schemaless data with scalar and multi-dimensional array values. This data is stored in a document-based database that assists with handling multi-dimensional array values. The management of the corresponding workload of these databases and the distribution of their queries lead to the recognition of the need for an adaptive index recommendation system.

As the first step, the index recommendation problem in the context of document-based databases is discussed. In this regard, the relevant criteria of the document-based databases to develop a framework for automatically solving the index recommendation problem are investigated. Based on the extracted relevant criteria and developed solutions to the index recommendation problem, the Adaptive Index Recommendation system (AIRs) is developed. This framework is in direct communication to the query optimizer of the targeted database.

To solve the index recommendation problem and to design AIRs, various practical and theoretical solutions are developed. To reduce the search space of the index recommendation problem, the *Frequent-Long* strategy is established utilizing the Frequent Itemset algorithm.

Another step to reduce the search space of this problem even further is based on utilizing the atomic configuration concept. This concept limits the search space to those configurations that are at least used by one query in the database. To extract these configurations, the Index Benefit Graph algorithm is developed. The analysis of complexity of this algorithm for databases with index intersection capability shows an exponential growth in worst-case scenario. However, in the evaluation process with real meteorological databases the feasibility of this algorithm is shown.

With help of this algorithm, the number of calls to the query optimizer is drastically reduced. These calls produce extra load on the query optimizer. To avoid overload on the in-production system due to the execution of the cost evaluation process by means of the query optimizer, a virtual environment that consists of a representative sample of the targeted dataset is presented. By materializing indexes on the representative sample set, they are available to the query optimizer for the cost evaluation process of each running query. Without materializing the indexes on the original dataset, their storage size is not directly measurable. Therefore, a method to approximately estimate the size of indexes is presented that is merely based on the number of documents containing their attributes and their corresponding value type.

Then, the mathematical formulation of an objective function to model the index recommendation problem is presented. This function is designed to cover all of the relevant criteria of the index recommendation problem. These criteria are related to both dataset characteristics, such as cardinality of attribute values, and workload characteristics, such as update cost of indexes for write operations. To guarantee finding the global optimal configuration, this objective function is formulated in the Integer Linear Programming that utilizes Branch-and-Bound enumeration technique to enumerate the space of candidate indexes.

Finally, to enable performance evaluation of the index recommendation system rather than the whole database performance, proper metrics are introduced. Then, the performance of the AIRs for some of the real meteorological workloads is evaluated with the help of these metrics. Additionally, a generic workload generator is introduced to allow defining various synthetic datasets and workloads that are used throughout the whole thesis.

However, as a result of utilization of various evaluation methodologies by different studies and their focus on relational databases, any direct comparison between the results of performance evaluation of AIRs and the other studies are inequitable. This provides a ground for further research possibilities.

7.1 Future Extensions

There is a vast potential for further studying the developed solution of this thesis to the index recommendation problem. The modularity of this solution allows research in complexity, performance and usability of additional methods.

So far, the research was restricted to the development and evaluation of the solution to the index recommendation problem regarding document-based databases. However, this solution is generic enough to be applicable to any other database types. Some changes are required to adapt this solution to the capabilities of the query optimizer and language of the targeted database. Doing so would provide an opportunity for a fair comparison between the performance studies of this solution on various databases.

Such comparison has not been possible due to the fundamental difference between the investigated document-based data model and the relational data model used by the majority of other studies. The lack of generic benchmarking methodology between relational and document-based models is also another problem that can be tackled in the future for further studies.

Despite the effort in this thesis to develop proper metrics to evaluate only the performance of the index recommendation system rather than the overall functionality of the database, a more precise metric that can cover the maintenance and storage costs of the indexes for the workload is also a subject for more investigations.

Appendix A

NoWog Grammar

Code A.1: EBNF of the generic language of NoWog. This language is used to determine the characteristics of the workload that the user want to be generated.

```
rule_set, "=", "{", rule, "{", rule, "}", "}", ";"
rule, "=", rule_ID, ":",
                        "{", quadruple, "=", absolute,
                            "}", ";"

quadruple, "=", read, ",", write, ",", sort, ",",
                time_period ";"
read,  "=", "(", "{",
                     { read_phrase  }, "}", ")" | "ALL", ";"
write, "=", "(", "{",
                     { write_phrase }, "}", ")" | "NULL", ";"
sort, "=", "(", "{",
                    { attribute, ":", sort_op }, "}", ")" |
                    "NULL", ";"
time_period, "=" minute, "-", minute, ";"
read_phrase, "=", "(", attribute, ":", read_type )", ";"
write_phrase, "=", "(", attribute, ":", write_type ")", ";"
read_type, "=", bool_match | text_read  | number_read  |
                array_read  | document_read, ";"
write_type, "=", bool_match | text_write | number_write |
                array_write | document_write, ";"

rule_ID, "=", identifier, ";"
attribute, "=", identifier, ";"
bool_match, "=", "True" | "False", ";"
number_read, "=", num_match | range_op | geo_op, ";"
```

```
number_write, "=", num_match, ";"
array_read, "=", arr_read_op,  ".", [ arr_read_type ], ";"
array_write, "=", ( arr_write_op | "Array" ), ".",
                    arr_write_type, ";"
document_read, "=", read_phrase,  { read_phrase  }, ";"
document_write, "=", write_phrase, { write_phrase }, ";"

absolute, "=", distribution, "(", { arguments, "," },
               total, ")", ";"
distribution, "=", "uniform" | "normal", ";"
arguments, "=", float_number, ";"

text_read, "=", "text_read", ";"
text_write, "=", "text_write", ";"
num_match, "=", "num_match", ";"
range_op, "=", "range_op", ";"
geo_op, "=", "geo_op", ";"
arr_read_op, "=", "arr_read_op", ";"
arr_write_op, "=", "arr_add_op" | "arr_remove_op", ";"
arr_read_type, "=", "Bool" | "Num" | "Text" | "range_op", ";"
arr_write_type, "=", "Bool" | "Num" | "Text", ";"

sort_op, "=", "1" | "-1", ";"

minute, "=", digit, { digit }, ";"
identifier, "=", ( letter | "_"), { letter | digit | "_" },
   ";"
float_number, "=", [ "-" ], digit, { digit },
                   [ ".", digit, { digit } ], ";"

digit, "=", "0" | "1" | "2" | "3" | "4" | "5" | "6"
           | "7" | "8" | "9"
letter, "=", "A" | "B" | "C" | "D" | "E" | "F" | "G"
           | "H" | "I" | "J" | "K" | "L" | "M" | "N"
           | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
           | "V" | "W" | "X" | "Y" | "Z" | "a" | "b"
           | "c" | "d" | "e" | "f" | "g" | "h" | "i"
           | "j" | "k" | "l" | "m" | "n" | "o" | "p"
           | "q" | "r" | "s" | "t" | "u" | "v" | "w"
           | "x" | "y" | "z"
```

Appendix B

Glossary

- Atomic Configuration: any configuration with the property that all of its indexes are used by some query in the workload is an atomic configuration for that workload.
- Benefit:
- Cardinality: the cardinality of an attribute in a data set is the number of unique values of that attribute. (Definition 2.2.1)
- Configuration: a set of possible indexes. (Definition 5.1.1)
- Collection: a group of documents.
- Conjunctive Operators: OR and And.
- Data File: data, the database generated metadata and indexes are stored in data files.
- Document: any object without references in form of JSON or XML standard format.
- Extents: the logical containers inside each *Data File* to store data and indexes.
- Enumeration Space: the union of all of candidate indexes for each query.
- Indexable Attributes: the attributes specified in the *search-predicate* of a query with addition of the attributes from the *sort-clause*. (Definition 4.3.1)
- Index Intersection: Exploit multiple-conditions selectivity by simultaneously scanning the single indexes of each condition.
- Index Materialization: creation or removal of indexes.
- Level in IBG: The shortest path between any node in IBG and its root node. (Definition 5.2.3)
- Long Query: a query with a run-time longer than a threshold. (Definition 4.3.2)

- Merge of Indexes: merge of two indexes results in an index with the leading attribute of the first index and the union of its following attributes with all attributes of the second index. (Definition 4.3.5)
- NP-hard: a problem Q is said to be NP-hard if all problems in NP are reducible to Q in polynomial time.
- Query Optimizer: a key component in a database that receives the parsed query as input and is responsible for identifying an efficient plan to execute the query.
- Query Processor: the query processor is a component in any database that performs two main task: query optimization and query execution.
- Selectivity: the cardinality of the attributes of an index over the total number of documents in the collection. (Definition 2.2.2)

Bibliography

[1] N. Bruno, *Automated Physical Database Design and Tuning*, 1st ed. Boca Raton, FL, USA: CRC Press, Inc., 2011. ISBN 1439815674, 9781439815670

[2] P. Ameri, U. Grabowski, J. Meyer, and A. Streit, "On the Application and Performance of MongoDB for Climate Satellite Data," *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pp. 652–659, 2014. doi: 10.1109/TrustCom.2014.84

[3] R. Lutz, P. Ameri, T. Latzko, and J. Meyer, "Management of Meteorological Mass Data with MongoDB," in *BIS-Verlag*. BIS-Verlag, 2014, pp. 549–556, ISBN: 978-3-8142-2317-9.

[4] M. Szuba, P. Ameri, U. Grabowski, J. Meyer, and A. Streit, "A Distributed System for Storing and Processing Data from Earth-Observing Satellites: System Design and Performance Evaluation of the Visualisation Tool," in *IEEE/ACM 16th International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2016, Cartagena, Colombia, May 16-19, 2016*. doi: 10.1109/CCGrid.2016.19 pp. 169–174.

[5] P. Ameri, "Database Techniques for Big Data," in *Big Data: Principles and Paradigms*, R. Buyya, R. Calheiros, and A. Dastjerdi, Eds. Morgan Kaufmann, 2017, ch. 6, ISBN: 978-0-12-805394-2.

[6] P. Ameri, N. Schlitter, J. Meyer, and A. Streit, "NoWog: A Workload Generator for Database Performance Benchmarking," in *IEEE International Conference on Big Data Intelligence and Computing*, 2016. doi: 10.1109/DASC-PICom-DataCom-CyberSciTec.2016.120 pp. 666–673.

[7] P. Ameri, J. Meyer, and A. Streit, "On a New Approach to the Index Selection Problem using Mining Algorithms," in *Big Data (Big Data), 2015 IEEE International Conference on*, 2015. doi: 10.1109/BigData.2015.7364084 pp. 2801–2810.

[8] "LSDMA-Earth and Environment DLCL," 2016, Accessed on May 2017. [Online]. Available: https://www.helmholtz-lsdma.de/climatology.php

[9] C. Jung, M. Gasthuber, A. Giesler, M. Hardt, J. Meyer, F. Rigoll, K. Schwarz, R. Stotzka, and A. Streit, “Optimization of data life cycles,” *Journal of Physics: Conference Series*, p. 032047, 2014, doi: 10.1088/1742-6596/513/3/032047.

[10] H. Fischer, M. Birk, C. Blom, B. Carli, M. Carlotti, T. von Clarmann, L. Delbouille, A. Dudhia, D. Ehhalt, M. Endemann, J. M. Flaud, R. Gessner, A. Kleinert, R. Koopman, J. Langen, M. López-Puertas, P. Mosner, H. Nett, H. Oelhaf, G. Perron, J. Remedios, M. Ridolfi, G. Stiller, and R. Zander, “Mipas: an instrument for atmospheric and climate research,” *Atmospheric Chemistry and Physics*, no. 8, pp. 2151–2188, 2008. doi: 10.5194/acp-8-2151-2008

[11] “Envisat web site,” 2014, Accessed on May 2017. [Online]. Available: http://envisat.esa.int/

[12] J. Waters, L. Froidevaux, R. Harwood, R. Jarnot, H. Pickett, W. Read, P. Siegel, R. Cofield, M. Filipiak, D. Flower, J. Holden, G. Lau, N. Livesey, G. Manney, H. Pumphrey, M. Santee, D. Wu, D. Cuddy, R. Lay, M. Loo, V. Perun, M. Schwartz, P. Stek, R. Thurstans, M. Boyles, K. Chandra, M. Chavez, G.-S. Chen, B. Chudasama, R. Dodge, R. Fuller, M. Girard, J. Jiang, Y. Jiang, B. Knosp, R. LaBelle, J. Lam, K. Lee, D. Miller, J. Oswald, N. Patel, D. Pukala, O. Quintero, D. Scaff, W. Van Snyder, M. Tope, P. Wagner, and M. Walch, “The Earth observing system microwave limb sounder (EOS MLS) on the aura Satellite,” *Geoscience and Remote Sensing, IEEE Transactions on*, pp. 1075–1092, 2006. doi: 10.1109/TGRS.2006.873771

[13] NASA Aura Team, “Aura web site,” 2014, Accessed on May 2017. [Online]. Available: http://aura.gsfc.nasa.gov/

[14] P. Jöckel, H. Tost, A. Pozzer, C. Brühl, J. Buchholz, L. Ganzeveld, P. Hoor, A. Kerkweg, M. Lawrence, R. Sander, B. Steil, G. Stiller, M. Tanarhte, D. Taraborrelli, J. van Aardenne, and J. Lelieveld, “The Atmospheric Chemistry General Circulation Model ECHAM5/MESSy1: Consistent Simulation of Ozone from the Surface to the Mesosphere,” *Atmospheric Chemistry and Physics*, pp. 5067–5104, 2006.

[15] B. Funke, A. Baumgaertner, M. Calisto, T. Egorova, C. H. Jackman, J. Kieser, A. Krivolutsky, M. López-Puertas, D. R. Marsh, T. Reddmann, E. Rozanov, S.-M. Salmi, M. Sinnhuber, G. P. Stiller, P. T. Verronen, S. Versick, T. von Clarmann, T. Y. Vyushkova, N. Wieters, and J. M. Wissing, “Composition Changes After the ”Halloween” Solar Proton Event: the High Energy Particle Precipitation in the Atmosphere (HEPPA) Model versus MIPAS Data Intercomparison Study,” *Atmospheric Chemistry and Physics*, pp. 9089–9139, 2011. doi: 10.5194/acp-11-9089-2011

[16] B. Eaton, J. Gregory, B. Drach, K. Taylor, S. Hankin, J. Caron, R. Signell, P. Bentley, and G. Rappa, “NetCDF Climate and Forecast (CF) Metadata Conventions, Version 1.4,” 2009. [Online]. Available: http://www.cgd.ucar.edu/cms/eaton/netcdf/CF-20010629.htm

[17] R. Rew, E. Hartnett, and J. Caron, “netCDF-4: Software Implementing an Enhanced Data Model for the Geosciences,” in *22nd International Conference on Interactive Information Processing Systems for Meteorology, Oceanograph, and Hydrology*, 2006.

[18] R. Lutz, U. Grabowski, T. Beckmann, T. von Clarmann, H. Fischer, B. Funke, N. Glatthor, M. Höpfner, S. Kellmann, M. Kiefer, A. Linden, M. Milz, S. Ressel, T. Steck, G. P. Stiller, G. Mengistu Tsidu, and D.-Y. Wang, “The imk mipas retrieval processor environment,” in *Proceedings of the 11th International Workshop on Atmospheric Science from Space using Fourier Transform Spectrometry, Bad Wildbad, Germany, Oct. 8–10, 2003.* Forschungszentrum Karlsruhe, Institut für Meteorologie und Klimaforschung, 2003. [Online]. Available: http://www.imk-asf.kit.edu/downloads/sat/P_I_13_Grabowski_U

[19] D. Agrawal, S. Das, and A. El Abbadi, “Big Data and Cloud Computing: Current State and Future Opportunities,” in *Proceedings of the 14th International Conference on Extending Database Technology*, ser. EDBT/ICDT ’11. New York, NY, USA: ACM, 2011. doi: 10.1145/1951365.1951432. ISBN 978-1-4503-0528-0 pp. 530–533.

[20] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” *Commun. ACM*, pp. 107–113. doi: 10.1145/1327452.1327492

[21] D. DeWitt and J. Gray, “Parallel Database Systems: The Future of High Performance Database Systems,” *Commun. ACM*, pp. 85–98. doi: 10.1145/129888.129894

[22] R. Sears, C. van Ingen, and J. Gray, “To blob or not to blob: Large object storage in a database or a filesystem?” *CoRR*, 2007. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr0701.html#abs-cs-0701168

[23] E. A. Brewer, “Towards Robust Distributed Systems (Abstract),” in *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC ’00. New York, NY, USA: ACM, 2000. doi: 10.1145/343477.343502. ISBN 1-58113-183-6 pp. 7–.

[24] J. Gray, “Readings in database systems,” M. Stonebraker, Ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1988, ch. The Transaction Concept: Virtues and Limitations, pp. 140–150. ISBN 0-934613-65-6

[25] D. Pritchett, “BASE: An Acid Alternative,” *Queue*, pp. 48–55. doi: 10.1145/1394127.1394128

[26] “JSON,” http://www.json.org/, Accessed on May 2017.

[27] “XML,” http://www.xml.com, Accessed on May 2017.

[28] “GeoJSON,” http://geojson.org/, 2014, Accessed on May 2017.

[29] "GLORIA Internet site of Earth Observation Portal," https://directory.eoportal.org/web/eoportal/airborne-sensors/gloria, Accessed on May 2017.

[30] "Internet site of DLR/HALO," http://www.halo.dlr.de/, Accessed on May 2017.

[31] "GLORIA Internet site of Helmholtz Association," http://gloria.helmholtz.de/, Accessed on May 2017.

[32] "Firebird: The true open source database for Windows, Linux, Mac OS X and more," https://firebirdsql.org/, Accessed on May 2017.

[33] R. Elmasri and S. Navathe, *Fundamentals of Database Systems*, 6th ed. USA: Addison-Wesley Publishing Company, 2010. ISBN 0136086209, 9780136086208

[34] W. Kent, "A Simple Guide to Five Normal Forms in Relational Database Theory," *Commun. ACM*, pp. 120–125. doi: 10.1145/358024.358054

[35] H. Garcia-Molina, J. Widom, and J. D. Ullman, *Database System Implementation.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1999. ISBN 0130402648

[36] G. Graefe, "Modern b-tree techniques," *Found. Trends databases*, pp. 203–402. doi: 10.1561/1900000028

[37] Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, and Y. Theodoridis, *R-Trees: Theory and Applications.* Springer Publishing Company, Incorporated, 2005. ISBN 1852339772, 9781852339777

[38] S.-S. Kim, "Sung-Soo Kim's Blog," Accessed on May 2017. [Online]. Available: http://sungsoo.github.io/2014/05/27/query-processing.html

[39] "Advanced Database Management System - Tutorials and Notes," Accessed on May 2017. [Online]. Available: http://www.exploredatabase.com/2014/09/query-processing-in-database.html

[40] S. Chaudhuri, "An Overview of Query Optimization in Relational Systems," in *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, ser. PODS '98. New York, NY, USA: ACM, 1998. doi: 10.1145/275487.275492. ISBN 0-89791-996-3 pp. 34–43.

[41] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin, "Automatic SQL Tuning in Oracle 10G," in *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, ser. VLDB '04. VLDB Endowment, 2004. ISBN 0-12-088469-0 pp. 1098–1109.

[42] V. Poosala, P. J. Haas, Y. E. Ioannidis, and E. J. Shekita, "Improved Histograms for Selectivity Estimation of Range Predicates," *SIGMOD Rec.*, pp. 294–305. doi: 10.1145/235968.233342

[43] R. P. Kooi, "The Optimization of Queries in Relational Databases," Ph.D. dissertation, Cleveland, OH, USA, 1980, AAI8109596.

[44] R. J. Lipton, J. F. Naughton, and D. A. Schneider, "Practical Selectivity Estimation through Adaptive Sampling," in *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990.*, 1990. doi: 10.1145/93597.93611 pp. 1–11.

[45] C. M. Chen and N. Roussopoulos, "Adaptive Selectivity Estimation Using Query Feedback," in *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '94. New York, NY, USA: ACM, 1994. doi: 10.1145/191839.191874. ISBN 0-89791-639-5 pp. 161–172.

[46] P. B. Gibbons, Y. Matias, and V. Poosala, "Fast Incremental Maintenance of Approximate Histograms," in *Proceedings of the 23rd International Conference on Very Large Data Bases*, ser. VLDB '97. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. ISBN 1-55860-470-7 pp. 466–475.

[47] L. S. Bonura, *The Art of Indexing.* New York, NY, USA: John Wiley & Sons, Inc., 1994. ISBN 0-471-01449-4

[48] D. Comer, "The Difficulty of Optimum Index Selection," *ACM Trans. Database Syst.*, pp. 440–445. doi: 10.1145/320289.320296

[49] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness.* New York, NY, USA: W. H. Freeman & Co., 1990. ISBN 0716710455

[50] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001. ISBN 0070131511

[51] A. Molinaro, *SQL Cookbook (Cookbooks (O'Reilly)).* O'Reilly Media, Inc., 2005. ISBN 0596009763

[52] TPC, 2016, Accessed on May 2017. [Online]. Available: http://www.tpc.org/information/benchmarks.asp

[53] D. McCreary and A. Kelly, *Making Sense of NoSQL: A Guide for Managers and the Rest of Us.* Manning, 2013. ISBN 9781617291074

[54] A. Silberschatz, H. Korth, and S. Sudarshan, *Database Systems Concepts*, 5th ed. New York, NY, USA: McGraw-Hill, Inc., 2006. ISBN 0072958863, 9780072958867

[55] C. J. Date and H. Darwen, *A Guide to the SQL Standard (4th Ed.): A User's Guide to the Standard Database Language SQL.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN 0-201-96426-0

[56] S. Papadomanolakis and A. Ailamaki, “An Integer Linear Programming Approach to Database Design.” in *In ICDE Workshop on Self-Managing Databases*, 2007. doi: 10.1109/ICDEW.2007.4401027

[57] S. Papadomanolakis, D. Dash, and A. Ailamaki, “Efficient Use of the Query Optimizer for Automated Physical Design,” in *Proceedings of the 33rd International Conference on Very Large Data Bases*, ser. VLDB ’07. VLDB Endowment, 2007. ISBN 978-1-59593-649-3 pp. 1093–1104.

[58] S. Idreos, M. L. Kersten, and S. Manegold, “Database Cracking,” in *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*, 2007. doi: 10.1145/2619228.2619232 pp. 68–78.

[59] N. Bruno, S. Chaudhuri, A. C. König, V. R. Narasayya, R. Ramamurthy, and M. Syamala, “AutoAdmin Project at Microsoft Research: Lessons Learned,” *IEEE Data Eng. Bull.*, pp. 12–19, 2011.

[60] D. C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden, “DB2 Design Advisor: Integrated Automatic Physical Database Design,” in *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, ser. VLDB ’04. VLDB Endowment, 2004. ISBN 0-12-088469-0 pp. 1087–1097.

[61] P. Ameri, “On a Self-Tuning Index Recommendation Approach for Databases,” in *IEEE International Conference on Data Engineering*, 2016. doi: 10.1109/ICDEW.2016.7495648 pp. 201–205.

[62] R. Karp, “Reducibility Among Combinatorial Problems,” in *Complexity of Computer Computations*, R. Miller and J. Thatcher, Eds. Plenum Press, 1972, pp. 85–103.

[63] G. Piatetsky-Shapiro, “The Optimal Selection of Secondary Indices is NP-complete,” *SIGMOD Rec.*, pp. 72–75. doi: 10.1145/984523.984530

[64] M. Datar, V. Narasayya, and S. Chaudhuri, “Index Selection for Databases: A Hardness Study and a Principled Heuristic Solution,” *IEEE Transactions on Knowledge Data Engineering*, pp. 1313–1323, 2004. doi: 10.1109/TKDE.2004.75

[65] U. Feige, G. Kortsarz, and D. Peleg, “The Dense k-Subgraph Problem,” *Algorithmica*, p. 2001, 1999. doi: 10.1007/s004530010050

[66] A. Skelley, “DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes,” in *Proceedings of the 16th International Conference on Data Engineering*, ser. ICDE ’00. Washington, DC, USA: IEEE Computer Society, 2000. ISBN 0-7695-0506-6 pp. 101–111.

[67] M. Zaman, J. Surabattula, and L. Gruenwald, “An Auto-Indexing Technique for Databases Based on Clustering,” in *15th International Workshop on Database and*

Expert Systems Applications (DEXA), 2004. doi: 10.1109/DEXA.2004.1333569 pp. 776–780.

[68] K. Aouiche and J. Darmont, “Data Mining-based Materialized View and Index Selection in Data Warehouses,” *CoRR*, 2007. doi: 10.1007/s10844-009-0080-0

[69] E. Barcucci and O. Pinzani, “Optimal Selection of Secondary Indexes,” *IEEE Transactions on Software Engineering*, pp. 32–38, 1990. doi: 10.1109/32.44361

[70] S. Choenni, H. M. Blanken, and T. Chang, “On the Selection of Secondary Indices in Relational Databases,” *Data & Knowledge Engineering*, pp. 207–233, 1993. doi: 10.1016/0169-023X(93)90023-I

[71] A. Caprara, M. Fischetti, and D. Maio, “Exact and Approximate Algorithms for the Index Selection Problem in Physical Database Design,” *IEEE Trans. on Knowl. and Data Eng.*, pp. 955–967. doi: 10.1109/69.476501

[72] M. R. Frank, E. R. Omiecinski, and S. B. Navathe, “Adaptive and automated index selection in rdbms,” in *In Proceedings of International Conference on Extending Database Technology*, 1992. doi: 10.1007/BFb0032437 pp. 277–292.

[73] R. Agrawal and R. Srikant, “Fast Algorithms for Mining Association Rules in Large Databases,” in *Proceedings of the 20th International Conference on Very Large Data Bases*, ser. VLDB ’94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994. ISBN 1-55860-153-8 pp. 487–499.

[74] S. Chaudhuri and V. Narasayya, “Index Merging,” in *Proceedings of the International Conference on data Engineering (ICDE)*, 1999. doi: 10.1109/ICDE.1999.754945

[75] S. Chaudhuri and V. R. Narasayya, “An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server,” in *Proceedings of the 23rd International Conference on Very Large Data Bases*, ser. VLDB ’97. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. ISBN 1-55860-470-7 pp. 146–155.

[76] J. Kratica, I. Ljubic, and D. Tošic, “A Genetic Algorithm for the Index Selection Problem,” in *Proceedings of the 2003 International Conference on Applications of Evolutionary Computing*, ser. EvoWorkshops’03. Berlin, Heidelberg: Springer-Verlag, 2003. doi: 10.1007/3-540-36605-9_26. ISBN 3-540-00976-0 pp. 280–290.

[77] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman, “Index Selection for OLAP,” in *Proceedings of the Thirteenth International Conference on Data Engineering*, ser. ICDE ’97. Washington, DC, USA: IEEE Computer Society, 1997. doi: 10.1109/ICDE.1997.581755. ISBN 0-8186-7807-0 pp. 208–219.

[78] S. Finkelstein, M. Schkolnick, and P. Tiberio, “Physical Database Design for Relational Databases,” *ACM Trans. Database Syst.*, pp. 91–128. doi: 10.1145/42201.42205

[79] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, and M. Syamala, "Database Tuning Advisor for Microsoft SQL Server 2005," in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '05. New York, NY, USA: ACM, 2005. doi: 10.1145/1066157.1066292. ISBN 1-59593-060-4 pp. 930–932.

[80] G. Graefe, "The Value of Merge-Join and Hash-Join in SQL Server," in *Proceedings of the International Conference on Very Large Data Bases*, ser. VLDB '99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999. ISBN 1-55860-615-7 pp. 250–253.

[81] W. P. Yan and P. Larson, "Performing Group-by before Join." Institute of Electrical and Electronics Engineers, Inc., 1994. doi: 10.1109/ICDE.1994.283001

[82] C. Galindo-Legaria and A. Rosenthal, "Outerjoin Simplification and Reordering for Query Optimization," *ACM Trans. Database Syst.*, pp. 43–74. doi: 10.1145/244810.244812

[83] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access Path Selection in a Relational Database Management System," in *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '79. New York, NY, USA: ACM, 1979. doi: 10.1145/582095.582099. ISBN 0-89791-001-X pp. 23–34.

[84] W. J. McKenna, "Efficient Search in Extensible Database Query Optimization: The Volcano Optimizer Generator," Ph.D. dissertation, Boulder, CO, USA, 1993, UMI Order No. GAX93-20458.

[85] G. Graefe, "The Cascades Framework for Query Optimization," *Data Engineering Bulletin*, vol. 18, 1995.

[86] K. Banker, *MongoDB in Action.* Greenwich, CT, USA: Manning Publications Co., 2011. ISBN 1935182870, 9781935182870

[87] M. A. DocumentDB, "DocumentDB Microsoft Azure," Accessed on May 2017. [Online]. Available: https://azure.microsoft.com/en-us/services/documentdb/

[88] M. Azure, "Microsoft Introduces NoSQL Document Database for Microsoft Azure," Accessed on May 2017. [Online]. Available: https://www.infoq.com/news/2014/08/microsoft-azure-documentdb

[89] S. Chaudhuri and V. R. Narasayya, "AutoAdmin 'What-if' Index Analysis Utility." in *SIGMOD Conference*, L. M. Haas and A. Tiwary, Eds. ACM Press, 1998. doi: 10.1145/276304.276337. ISBN 0-89791-995-5 pp. 367–378.

[90] R. Wang, Q. T. Tran, I. Jimenez, and N. Polyzotis, "INUM+: A leaner, more accurate and more efficient fast what-if optimizer," *2013 IEEE 29th International Conference on Data Engineering Workshops (ICDEW 2013)*, pp. 50–55, 2013. doi: 10.1109/ICDEW.2013.6547426

[91] W. Wu, J. F. Naughton, and H. Singh, "Sampling-Based Query Re-Optimization," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: ACM, 2016. doi: 10.1145/2882903.2882914. ISBN 978-1-4503-3531-7 pp. 1721–1736.

[92] K. Schnaitter and N. Polyzotis, "A Benchmark for Online Index Selection," *IEEE International Conference on Data Engineering. ICDE*, pp. 1701–1708, 2009. doi: 10.1109/ICDE.2009.166

[93] P. J. Haas and A. N. Swami, "Sequential Sampling Procedures for Query Size Estimation," *SIGMOD Rec.*, pp. 341–350. doi: 10.1145/141484.130335

[94] F. Olken and D. Rotem, "Random Sampling from Databases," 1993. doi: 10.1007/BF00140664

[95] G. Piatetsky-Shapiro and C. Connell, "Accurate Estimation of the Number of Tuples Satisfying a Condition," *SIGMOD Rec.*, pp. 256–276. doi: 10.1145/971697.602294

[96] S. Chaudhuri, R. Motwani, and V. Narasayya, "Random Sampling for Histogram Construction: How Much is Enough?" *SIGMOD Rec.*, pp. 436–447. doi: 10.1145/276305.276343

[97] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009. ISBN 0262033844, 9780262033848

[98] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations.* New York, NY, USA: John Wiley & Sons, Inc., 1990. ISBN 0-471-92420-2

[99] L. Saxton, V. Raghavan, and M. Ip, "On the Selection of an Optimal Set of Indexes," *IEEE Transactions on Software Engineering*, pp. 135–143, 1983. doi: 10.1109/TSE.1983.236458

[100] Y. A. Feldman and J. Reouven, "A Knowledge-based Approach for Index Selection in Relational Databases," *Expert Systems with Applications*, pp. 15–37, 2003. doi: 10.1016/S0957-4174(03)00003-4

[101] D. Zilio, S. Lightstone, K. Lyons, and G. Lohman, "Self-managing Technology in IBM DB2 Universal Database," in *Proceedings of the Tenth International Conference on Information and Knowledge Management*, ser. CIKM '01. New York, NY, USA: ACM, 2001. doi: 10.1145/502585.502682. ISBN 1-58113-436-3 pp. 541–543.

[102] T. Gündem, "Near Optimal Multiple Choice Index Selection for Relational Databases," *Computers Mathematics with Applications*, pp. 111–120, 1999. doi: 10.1016/S0898-1221(98)00256-9

[103] S. N. Sivanandam and S. N. Deepa, *Introduction to Genetic Algorithms*, 1st ed. Springer Publishing Company, Incorporated, 2007. ISBN 354073189X, 9783540731894

[104] N. Bruno and S. Chaudhuri, "Automatic Physical Database Tuning: A Relaxation-based Approach," in *Proceedings of the ACM International Conference on Management of Data (SIGMOD).* Association for Computing Machinery, Inc., 2005. doi: 10.1145/1066157.1066184

[105] M. Conforti, G. Cornuejols, and G. Zambelli, *Integer Programming.* Springer Publishing Company, Incorporated, 2014. ISBN 3319110071, 9783319110073

[106] D. R. Morrison, S. H. Jacobson, J. J. Sauppe, and E. C. Sewell, "Branch-and-Bound Algorithms," *Discret. Optim.*, no. C, pp. 79–102. doi: 10.1016/j.disopt.2016.01.005

[107] K. Schnaitter, N. Polyzotis, and L. Getoor, "Index Interactions in Physical Design Tuning: Modeling, Analysis, and Applications," in *International Conference on Very Large Data Bases*, 2009. doi: 10.14778/1687627.1687766

[108] A. Schrijver, *Theory of Linear and Integer Programming.* New York, NY, USA: John Wiley & Sons, Inc., 1986. ISBN 0-471-90854-1

[109] J. Gray, *Benchmark Handbook: For Database and Transaction Processing Systems.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992. ISBN 1558601597

[110] R. Jain, *The Art of Computer Systems Performance Analysis - Techniques for Experimental Design, Measurement, Simulation, and Modeling.*, ser. Wiley professional computing. Wiley, 1991. ISBN 978-0-471-50336-1

[111] M. P. Consens, D. Barbosa, A. Teisanu, and L. Mignet, "Goals and Benchmarks for Autonomic Configuration Recommenders," in *ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '05. New York, NY, USA: ACM, 2005. doi: 10.1145/1066157.1066185. ISBN 1-59593-060-4 pp. 239–250.

[112] M. Poess and C. Floyd, "New TPC Benchmarks for Decision Support and Web Commerce," *SIGMOD Rec.*, pp. 64–71. doi: 10.1145/369275.369291

[113] N. Bruno, "A Critical Look at the TAB Benchmark for Physical Design Tools," in *Sigmod Record.* Association for Computing Machinery, Inc., 2007. doi: 10.1145/1361348.1361349

[114] C. H. Wu, H. Huang, L. Arminski, J. Castro-Alvear, Y. Chen, Z.-Z. Hu, R. S. Ledley, K. C. Lewis, H.-W. Mewes, B. C. Orcutt, B. Suzek, A. Tsugita, C. R. Vinayaka, L.-S. L. Yeh, J. Zhang, and W. C. Barker, "The Protein Information Resource: an

Integrated Public Resource of Functional Annotation of Proteins," *Nucleic Acids Research*, p. 35, 2002. doi: 10.1093/nar/30.1.35

[115] "Transaction Processing Performance Council Benchmark H- Decision Support," Accessed on May 2017. [Online]. Available: http://www.tpc.org/tpch/default.asp

[116] S. Chaudhuri and V. R. Narasayya, "TPC-D Data Generation with Skew." [Online]. Available: ftp.research.microsoft.com/users/viveknar/tpcdskew

[117] A. Caprara and J. J. S. González, "Separating lifted odd-hole inequalities to solve the index selection problem," *Discrete Applied Mathematics*, pp. 111 – 134, 1999. doi: 10.1016/S0166-218X(99)00050-5

[118] "TPC-H for NoSQL Performance benchmark," Accessed on May 2017. [Online]. Available: http://blogs.impetus.com/test_engineering/performance_engineering/tpchnosqlperformancebenchmark.do

[119] N. Rutishauser, "TPC-H applied to MongoDB: How a NoSQL Database Performs," 2012, Accessed on May 2017. [Online]. Available: http://www.ifi.uzh.ch/dbtg/teaching/thesesarch/VertiefungRutishauser.pdf

[120] O. Curé, R. Hecht, C. L. Duc, and M. Lamolle, "Data integration over nosql stores using access path based mappings," in *Proceedings of the 22Nd International Conference on Database and Expert Systems Applications*, ser. DEXA'11. Berlin, Heidelberg: Springer-Verlag, 2011. ISBN 978-3-642-23087-5 pp. 481–495.

[121] YCSB, "YCSB wiki," Accessed on May 2017. [Online]. Available: https://github.com/brianfrankcooper/YCSB/wiki

[122] P. Ameri, "Challenges of Index Recommendation for Databases," in *Proceedings of the 28th GI-Workshop Grundlagen von Datenbanken, Nörten Hardenberg, Germany, May 24-27, 2016.*, ser. CEUR Workshop Proceedings, L. Wiese, H. Bitzmann, and T. Waage, Eds. CEUR-WS.org, 2016, pp. 10–14. [Online]. Available: http://ceur-ws.org/Vol-1594/paper3.pdf

[123] C. Estan and J. F. Naughton, "End-biased Samples for Join Cardinality Estimation," in *International Conference on Data Engineering (ICDE'06)*, 2006. doi: 10.1109/ICDE.2006.61. ISSN 1063-6382 pp. 20–20.

[124] T. Hoque, C. K.-S. Leung, and Q. I. Khan, "CanTree: A Tree Structure for Efficient Incremental Mining of Frequent Patterns," *2013 IEEE 13th International Conference on Data Mining*, pp. 274–281, 2005. doi: 10.1109/ICDM.2005.38

[125] S. S. Lightstone, T. J. Teorey, and T. Nadeau, *Physical Database Design: The Database Professional's Guide to Exploiting Indexes, Views, Storage, and More.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007. ISBN 0123693896, 9780080552316

[126] D. Comer, "Ubiquitous B-Tree," *ACM Comput. Surv.*, pp. 121–137. doi: 10.1145/356770.356776

[127] S. Agrawal, S. Chaudhuri, and V. R. Narasayya, "Automated Selection of Materialized Views and Indexes in SQL Databases," in *Proceedings of the 26th International Conference on Very Large Data Bases.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000. ISBN 1-55860-715-3 pp. 496–505.

[128] R. Horst and T. Hoang, *Global Optimization : Deterministic Approaches.* Berlin, New York: Springer-Verlag, 1993. ISBN 3-540-56094-7

[129] C. Godsil and G. Royle, *Algebraic Graph Theory*, ser. Graduate Texts in Mathematics. volume 207 of Graduate Texts in Mathematics. Springer, 2001. ISBN 978-0-387-95220-8

[130] D. Tsirogiannis, S. Guha, and N. Koudas, "Improving the Performance of List Intersection," *Proc. VLDB Endow.*, pp. 838–849. doi: 10.14778/1687627.1687722

[131] S. Tanny and M. Zuker, "Analytic Methods Applied to a Sequence of Binomial Coefficients," *Elsavier.* doi: 10.1016/0012-365X(78)90101-2

[132] S. S. Conn, "OLTP and OLAP Data Integration: a Review of Feasible Implementation Methods and Architectures for Real Time Data Analysis," in *Proceedings. IEEE SoutheastCon, 2005.*, 2005. doi: 10.1109/SECON.2005.1423297. ISSN 1091-0050 pp. 515–520.

[133] MongoDB, "The MongoDB 3.2 Manual¶." [Online]. Available: https://docs.mongodb.com/v3.2/