Abstract of "Correlation-Aware Optimizations for Analytic Databases" by Hideaki Kimura, Ph.D., Brown University, May 2012.

This thesis studies various techniques that exploit correlations between attributes to significantly improve the performance and maintainability of analytic databases. We first show how a correlation-based secondary index developed here achieves index sizes smaller by orders of magnitude than a conventional secondary index. We then illustrate how, in the context of analytic databases, using a secondary index that is strongly correlated with a clustered index performs orders of magnitude faster than the uncorrelated case. Our goal was, then, to exploit these two observations in real database system settings. To meet the above goal, we developed (1) a data structure to store correlations as a secondary index, and (2) a database design tool to produce correlated indexes and materialized views. We further extended its applicability in a few directions, namely: (3) a formulation and optimization of index deployment to achieve faster completion as well as earlier query speed-up, (4) flexible partitioning and sorting techniques to apply the idea of correlations in distributed systems such as MapReduce, and (5) a clustered index structure for uncertain attributes to apply the benefits of correlations to uncertain databases.

Correlation-Aware Optimizations for Analytic Databases

by

Hideaki Kimura

B. Eng., University of Tokyo, 2005

M. Sc., Brown University, 2008

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island

May 2012

This dissertation by Hideaki Kimura is accepted in its present form by the Department of Computer Science as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date ______________ ______________________________
Stanley B. Zdonik, Director

Recommended to the Graduate Council

Date ______________ ______________________________
Ugur Cetintemel, Reader

Date ______________ ______________________________
Pascal Van Hentenryck, Reader

Date ______________ ______________________________
Samuel Madden, Reader
(Massachusetts Institute of Technology)

Approved by the Graduate Council

Date ______________ ______________________________
Peter Weber
Dean of the Graduate School

Acknowledgments

I thank everyone I worked with throughout my PhD project – Carleton Coffrin, Andrew Ferguson, Rodrigo Fonseca, George Huo, Samuel Madden, Alexander Rasin, Meinolf Sellmann, Pascal Van Hentenryck, and Stanley B. Zdonik. Their insightful ideas made the projects exciting and productive.

I also thank all members of the Brown Data Management Group – Yanif Ahmad, Mert Akdere, Natahan Backman, Ugur Cetintemel, Justin DeBarabant, Jennie Duggan, Tingjian Ge, Jeong-Hyon Hwang, Alexander Kalinin, Andrew Pavlo, Olga Papaemmanouil, and Matteo Riondato. They gave me invaluable help on research, career development, and life in the U.S. I could not have finished the PhD degree without them.

Thanks to my hacker friends in Japan, especially Fumiya Chiba, Kohsuke Kawaguchi, Takuo Kihira, Akira Muramatsu, and Kasuga Shinya, with whom I fostered my programming skills and interests in Computer Science. The days spent with them have been the spine that firmly supported me all the way to completion of my PhD.

Special thanks to Jim Gray, whose book on transactional processing charmed me into DBMS studies. His legendary contribution to database research and warm-hearted personality have always been and will continue to be my goal. His loss at sea is a true tragedy. I also thank Masaru Kitsuregawa and his students for their intense work to translate Gray's seminal book into Japanese language, without it I would still have thought that DBMS is "just a tool".

I thank all people in the New England database community at Brandeis, Boston, MIT, and Yale. The C-Store project here opened my eyes to opportunities for innovations in DBMS, even after four decades of research, and led me pursue my PhD in New England. Thanks especially to Samuel Madden for his endless insights and selfless help to members of the community.

Thanks to people I worked with outside of Brown. I especially thank Vivek Narasayya and Manoj Syamala at Microsoft Research as well as Goetz Graefe and Harumi Kuno at HP Labs. They gave me broad experiences and perspectives on what I have not yet studied.

Finally, Stanley B. Zdonik, my primary advisor, has guided me since my first day at Brown until now. His advising style is to listen carefully to his students' interests, encourage them to pursue what they want, and steer them towards success not forcibly but patiently. It is hard to imagine anyone else who even comes close to him in this regard.

Part of the work contained in this thesis was published in the International Conference on Very Large Data Bases (VLDB) [KHR+09, KHR+10, KMZ10] and in the International Conference on Extending Database Technology (EDBT) [KCRZ12].

I also thank the Corinna Borden Keen Research Fellowship, the Hewlett-Packard Laboratories Graduate Research Fellowship, and the National Science Foundation for funding my research.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Databases have been central to computing systems for decades, ranging from banking applications (*On-Line Transaction Processing*, or OLTP) to data analysis (*On-Line Analytic Processing*, or OLAP). With the quick and steady growth in the amount and complexity of data and queries, the performance and maintainability of databases has been always the key issue in building more stable and more advanced systems.

1.1 Database Indexes

Indexing is the most effective and prevalent technique to speed up query executions in databases. Indexes are categorized into two groups; non-clustered (secondary) indexes and clustered (primary) indexes, as depicted in Figure 1.1.

A non-clustered index is an auxiliary data structure that stores pointers to corresponding tuples in the table sorted by the values of the indexed attributes (*index keys*). A clustered index, on the other hand, alters the physical ordering of the table itself to be in the order of index keys.

The two types of indexes have substantially different properties in performance and maintainability. In terms of query performance, a clustered index is generally faster than a non-clustered index on the same index key. This is because the database must follow the pointers from the non-clustered index to retrieve columns not included in the index, paying expensive disk seek costs. However, a table can have only one clustered index because a clustered index is the table itself, while a table can have an arbitrary number of non-clustered indexes.

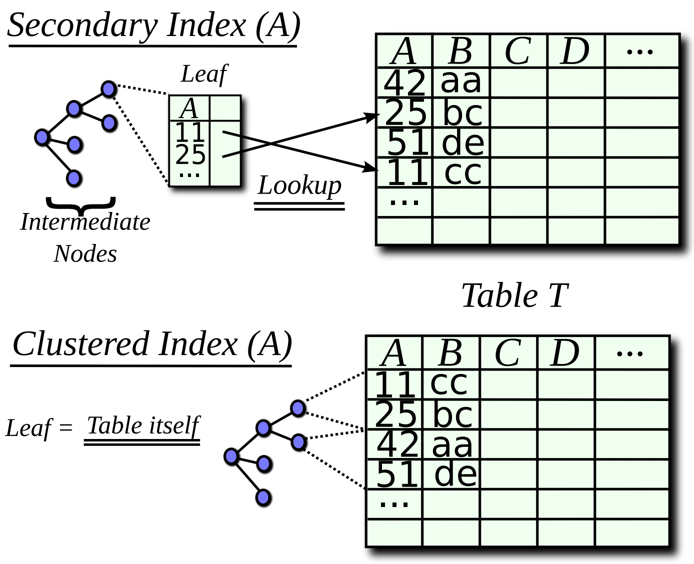

Figure 1.1: Secondary and Clustered Indexes

Some database products support a special type of clustered index called multi-dimensional-clustering (MDC) [PBM+03], which sorts page blocks by multiple index keys. However, it must reduce the granularity of each index key or limit the number of keys because an MDC still has to be aligned on one-dimensional harddisks. The only way to avoid this fundamental limitation is to duplicate the entire (or a subset of the) table as another auxiliary data structure, called a *Materialized View* (MV). Although MVs allow the table to be aligned in multiple orders, the increased space usage causes high overheads for data updates (INSERT/DELETE) and administrative operations such as backups.

Consequently, it is not common to use clustered indexes aggressively. Non-clustered indexes are handy and provide sufficient performance in some cases such as OLTP systems. However, in OLAP systems, queries tend to scan significantly larger ranges, and thus many more tuples, to extract analysis. As random disk seek has remained expensive for decades while the throughput of sequential access has dramatically increased over the years, performing a random disk seek for each tuple read by the query substantially slows down query execution.

Furthermore, the overhead of maintaining a non-clustered index against updates is known to be proportional to the size of the index [WM11]. As OLAP systems usually have millions to billions of tuples, non-clustered indexes significantly damage the database's capability to keep up with updates.

Hence, the common practice in OLAP systems is not to use any non-clustered index but to scan the entire table, especially if the query selectivity (the fraction of data read by the query) is larger than 1% (or even less). This inefficiency of non-clustered indexes prevents OLAP systems from scaling up to larger and more sophisticated analysis.

1.2 Motivation

Many researchers and practitioners have made several efforts to overcome this problem in OLAP systems. One approach is to speed up the full table scan by distributing data horizontally over many harddisks and many machines. Google's BigTable [CDG$^+$08], the DataPath project at Rice University [ADJ$^+$10], and the HadoopDB project at Yale University [ABPA$^+$09] employ this approach to speed up massive data analysis.

Another approach is to store the tables in Column-Store [Aba08]. Column-Store stores each column as a separate file to reduce the size of data accessed per query. It also applies compression on each column file to further reduce the I/O cost to scan the tables [AMF06]. The C-Store project at MIT, Brown and Brandeis University [SAB$^+$05], and its commercialized variant Vertica [Ver], the MonetDB project at CWI [Mon] and its commercialized variant VectorWise [Vec], and BigTable fall into this category. Both of these approaches complement the inefficiency of non-clustered indexes in OLAP systems by not using them and speeding up the table scan instead. However, yet another approach is developed in this thesis: speeding up non-clustered indexes by enhancing and exploiting their ***correlations*** with clustered indexes.

We are the first to study various aspects of this approach in a systematic manner and to observe that the correlation-based approach significantly improves the query performance and maintainability of databases. The approach is also promising in that it is orthogonal to the two existing approaches above and can be used together with them to realize even faster and more scalable databases.

Correlations between attributes in database tables have the potentials to speed up queries substantially while keeping indexes compact. The main theme of this thesis is how to exploit such attribute correlations to realize faster and more scalable analytic databases.

Before digging into the technical discussions of how to exploit such correlations, we first clarify the attribute correlations we want to exploit (Section 1.3), demonstrate their potential benefits (Section 1.4), and then list the technical challenges we need to address to exploit the correlations

(Section 1.5). We provide the overview of the following chapters, each of which addresses the issues.

1.3 What Are Attribute Correlations?

A correlation between attribute sets $A_1, A_2, \ldots, A_n$ in the table T is a frequent co-occurrence of the attributes' values in the table. For example, suppose the LINEITEM table shown in Table 1.1 stores information on each sold item.

Table 1.1: Example of attribute correlations in LINEITEM table. *CommitDate* is strongly correlated with *ShipDate* while it has no correlation with *OrderKey* or *Price*.

OrderKey	CommitDate	ShipDate	Price
123	1999/04/01	1999/04/06	\$1,223
124	1995/09/20	1995/09/19	\$210
125	1997/01/01	1997/01/11	\$6,543
...			

In this table, the values of *CommitDate* (the committed delivery date) and *ShipDate* (the actual shipping date) in a tuple have a strong correlation. The seller ships each item well before the committed delivery date in most cases and, even when delivery is delayed, at least by a week after the committed date.

Thus, the two attributes *CommitDate* and *ShipDate* have a limited set of co-occurring pairs of values shown in Table 1.2. Such a relation between table attributes is called ***Attribute Correlation*** and is the key concern throughout this thesis. Unlike the other attribute pairs (e.g., *CommitDate* and *Price*) that have no correlations, a correlated set of attributes have substantial potential to improve the performance and maintainability of analytic databases.

Table 1.2: Co-occurrences of the values of *CommitDate* and *ShipDate*.

CommitDate	ShipDate
1999/04/01	1999/03/01 $\sim$ 1999/04/08
1999/04/02	1999/03/02 $\sim$ 1999/04/09
...	

1.4 Opportunities for Correlations

Attribute correlations are observed in a variety of domains [BH03, IMH+04] and offer two significant opportunities.

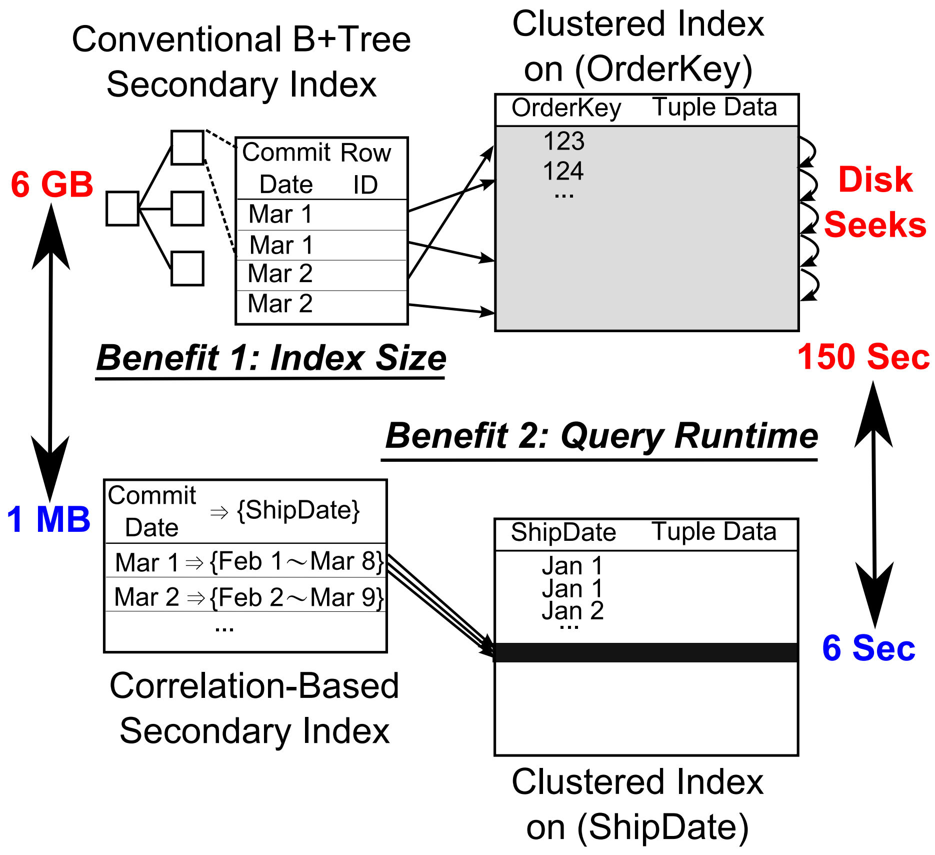

Figure 1.2: Opportunities for Correlations

Suppose we want to access the tuples with a CommitDate value of March 1st. Figure 1.2 illustrates the benefits of correlations by contrasting a conventional method to the proposed method exploiting correlations. The conventional method uses a B+Tree secondary index on CommitDate and a clustered index on primary key (OrderKey), while the proposed method uses a *correlation-based secondary index* on CommitDate and a clustered index on ShipDate.

First, correlations between attributes can be used as a compact secondary index, which is orders of magnitude smaller than the conventional B+Tree indexes. A B+Tree secondary index has to store one index entry for each tuple, consisting of the index key and *pointer* (i.e., RowID or the value of clustered attribute) to the corresponding tuple. Hence, the size of the index becomes quite large for large tables, e.g., 6 GB in TPC-H Scale 20. On the other hand, a correlation-based secondary index stores the co-occurrences of secondary (CommitDate) and clustered (ShipDate) attributes. As long as the attributes have strong correlations, the number of distinct value pairs is quite small and independent of the total number of tuples, e.g., only 1 MB in the above case. Therefore, a correlation-based secondary index is often 1,000 times or more smaller than an equivalent B+Tree secondary index. This compact size has significant benefits in storage consumption, query runtime, update overheads, pressures on the bufferpool and administrative costs.

Second, a secondary index that is strongly correlated with the underlying clustered index performs significantly faster than the uncorrelated case. This is because, in the uncorrelated case, the pointers from the secondary index land at random places in the table. The portions of the table we need to access are completely scattered over the table, so that the database must do several random disk seeks even with the help of bitmap index scan [1]. In analytic databases that have less selective queries, this means thousands or millions of disk seeks, making secondary indexes useless. On the other hand, if the secondary and the clustered indexes are correlated, the pointers land in a small contiguous area of the table. In such a case, the database can answer the query orders of magnitude faster.

These two benefits of correlations have significant potential to improve the performance and maintainability of analytic databases. The goal of this thesis is fully to exploit this potential.

1.5 Challenges in Exploiting Correlations

Despite significant potential, exploiting correlations encounters several technical challenges. Each of these issues corresponds to the package of techniques developed in each chapter below.

1.5.1 How to Detect, Store and Utilize Correlations

The first challenge is how to detect such beneficial correlations and store them as a compact secondary index.

Real databases have a large number of columns. Efficiently finding a strongly correlated set of attributes is not trivial, especially when considering correlations among more than two attributes.

[1] Bitmap index scan is a technique to amortize disk seek costs by sorting the pointers first and then following them in the order.

Also, observe that the CommitDate/ShipDate pair prefers *bucketing* in the above example. As the co-occurring values of ShipDate for a particular value of CommitDate span contiguous 37 days, representing the range of secondary values as a bucket gives a much smaller index size. However, automatically finding the best bucket size requires nontrivial analysis over the correlations.

Finally, we need to determine the best way to construct and maintain the correlations as a secondary index.

We address these issues in Chapter 2 by developing a novel secondary index data structure, *Correlation Maps* (CMs). We then study the algorithm and implementations to detect, store and utilize a wide range of attribute correlations.

1.5.2 How to Enhance Correlations

The second challenge is how to enhance correlations between the secondary and clustered indexes.

Even though there are strongly correlated *attributes*, the benefits of correlations are available only when the secondary and clustered *indexes* are strongly correlated. Hence, a poor choice of clustered indexes yields slower query performances and larger index sizes.

However, choosing the best clustered index is non-trivial. Unlike secondary indexes, one table can have only one clustered index. We must choose the attribute carefully to cluster the table so that as many secondary indexes as possible are benefited. Another option is to build materialized views (MVs) to have different clustering, but this approach brings yet another complication: the trade-off between space consumption and performance.

We tackle these problems in Chapter 3. We develop a novel correlation-aware database design tool that suggests indexes and MVs so that correlations between clustered and secondary indexes are enhanced to improve performance for the given query workload and storage budget. Overall, we achieve up to six times better query performance for a given storage budget compared to the state-of-the-art design tool which does not take correlations into account.

1.5.3 How to Deploy Indexes

The third challenge is how to deploy the suggested indexes efficiently.

Although the design suggested by the aforementioned correlation-aware design tool achieves significantly faster query performance, this design often contains many indexes, and especially many more clustered indexes, because fully utilizing clustered indexes is the key concept of the new design tool.

Deploying indexes, especially clustered indexes, consumes immense hardware resources and takes a long time on large databases. In such a situation, the *order* of indexes to deploy has a significant impact on the query speed-up and the total deployment time. However, optimizing index deployment order is quite challenging because of complex interactions between indexes and the factorial number of possible solutions.

In Chapter 4, we formally define the problem of index deployment order and study several optimization techniques for it. We propose a set of powerful pruning techniques to get the optimal solution and local search methods based on constraint programming (CP) that quickly get near-optimal solutions for larger problems.

1.5.4 Extensions for Distributed Systems

The fourth challenge is how to apply the idea of correlated clustering and secondary indexes to distributed systems.

A wide range of applications nowadays need to store and analyze far larger data than the traditional shared-disk architecture could handle. The only possible approach is to partition and distribute the big data over shared-nothing networks consisting of a large number of commodity servers.

In this setting, we found the significant effect of correlations on query performance again, but for a different reason. Here, the dominating bottleneck in query execution is shipping the data joined or aggregated together to other nodes. Choosing a partitioning key that is well correlated with the join key or aggregate key substantially reduces, or sometimes completely eliminates, the amount of data that have to be transmitted.

In addition to the choice of partitioning, a new key factor is the frequent hardware and software failures in the system due to the large number of machines and commodity hardware. Shared-nothing distributed file systems, such as Hadoop Distributed File System (HDFS), recover the data from replicas of the lost data block in such failures.

Our key idea is to exploit this redundancy in distributed systems to deploy differently partitioned replicas. However, this idea comes with a challenge, namely failure recovery. A different partitioning for each replica could worsen the recovery latency and the risk of permanent data loss.

In Chapter 5, we develop data structures, data placement policies, and analytic models of recovery to solve the issue.

1.5.5 Clustered Index on Uncertain Attributes

The final challenge is how to apply the idea of correlations to an emerging database type, *Uncertain Databases*.

In an uncertain database, an attribute has multiple possible values instead of a single deterministic value. Hence, there is no trivial way to cluster a table on such uncertain attributes. However, without a strongly correlated clustered index, none of the above techniques is meaningful.

We address this issue in Chapter 6 by devising the first clustered index for uncertain databases, *Uncertain Primary Indexes* (UPIs), and a secondary index structure to exploit correlations with UPIs. We discuss its data structure, algorithm and trade-offs between space consumption and query performance in the chapter. We observe that UPIs achieve orders of magnitude faster query performances and also enable secondary indexes to exploit correlations.

1.6 Contributions and Thesis Outline

With the above opportunities and technical issues of correlations in mind, we discuss the techniques to overcome the challenges.

The key contributions in this thesis are:

1. Algorithms and data structure for a fast and compact non-clustered index that detects, stores and exploits correlations **(Chapter 2)**.

2. Theory and implementation of a new correlation-aware physical database designer that enhances and exploits correlations between clustered and non-clustered indexes to speed up the OLAP database **(Chapter 3)**.

3. Formulation and optimization of the index deployment problem on top of CP **(Chapter 4)**.

4. Flexible partitioning and sorting techniques in distributed systems such as MapReduce with emphasis on recoverability guarantees **(Chapter 5)**.

5. A novel clustered index structure geared for uncertain databases that significantly speeds up analytic queries and enables the system to exploit correlations between clustered and non-clustered indexes **(Chapter 6)**.

6. Implementation and empirical analysis of the above techniques.

For better readability, each chapter starts by reviewing the context and prior work related to the techniques proposed in the chapter, we thus do not have a comprehensive related work chapter. Likewise, rather than a single experimental results chapter, we also provide the experimental results and implementation details in each chapter.

Chapter 2

Correlation Maps: *Detecting, Storing and Utilizing Correlations*

In relational query processing, there are generally two choices for access paths when performing a predicate lookup for which no clustered index is available. One option is to use an unclustered index. Another is to perform a complete sequential scan of the table. Many analytical workloads do not benefit from the availability of unclustered indexes; the cost of random disk I/O becomes prohibitive for all but the most selective queries.

It has been observed that a secondary index on an unclustered attribute can perform well under certain conditions if the unclustered attribute is correlated with a clustered index attribute [BH03]. The clustered index will co-locate values and the correlation will localize access through the unclustered attribute to a subset of the pages. In this chapter, we show that in a real application (SDSS) and widely used benchmark (TPC-H), there exist many cases of attribute correlation that can be exploited to accelerate queries. We also discuss a tool that can automatically suggest useful pairs of correlated attributes. It does so using an analytical cost model that we developed, which is novel in its awareness of the effects of clustering and correlation.

Furthermore, we propose a data structure called a Correlation Map (CM) that expresses the mapping between the correlated attributes, acting much like a secondary index. The chapter also discusses how bucketing on the domains of both attributes in the correlated attribute pair can dramatically reduce the size of the CM to be potentially orders of magnitude smaller than that of a secondary B+Tree index. This reduction in size allows us to create a large number of CMs that improve performance for a wide range of queries. The small size also reduces maintenance costs as we demonstrate experimentally.

2.1 Introduction

Correlations appear in a wide range of domains, including product catalogs, geographic databases, census data, and so on [BH03, IMH$^+$04]. For example, demographics (race, income, age, etc.) are highly correlated with geography; price is highly correlated with product industry; in the natural world, temperature, light, humidity, energy, and other parameters are often highly correlated; in the stock market, trends in one security are often closely related to those of others in similar markets.

Recent years have seen the widespread recognition that correlations can be effectively exploited to improve query processing performance [BH03, IMH$^+$04, CFPT94, CGK$^+$99, GSZZ02]. In particular, if a column C_1 is correlated with another column C_2 in table T, then it may be possible to use access methods (such as clustered indexes) on C_2 to evaluate predicates on C_1, rather than using the access methods available for C_1 alone [BH03, CFPT94, CGK$^+$99, GSZZ02].

In this chapter, we focus on a broad class of correlations known as *soft functional dependencies* (soft FDs), where the values of an attribute are well-predicted by the values of another attribute. For example, if we know that the value of `city` is Boston, we know with high probability but not with certainty that the value of `state` is Massachusetts (since there is a large city named Boston in Massachusetts and a much smaller one in New Hampshire). Such soft FDs are a generalization of hard FDs, where one attribute is a perfect predictor of another attribute.

Previous work has observed that soft FDs can be exploited by introducing additional predicates into queries [BH03, GSZZ02] when a predicate over only one correlated attribute exists. For example, if a user runs the query `SELECT * FROM emp WHERE city='boston'`, we can rewrite the query as

```
SELECT * FROM emp
WHERE city='boston'
AND (state='MA' or state='NH')
```

This will allow the query optimizer to exploit access methods, such as a clustered index on `state`, that it would not otherwise choose for query processing. Estimating and improving the performance of such secondary index lookups in the presence of correlations is our primary goal in this work.

In this work, we make three principal contributions beyond existing approaches: first, we describe a set of algorithms to search for soft functional dependencies that can be exploited at query execution time (e.g., by introducing appropriate predicates or choosing a different index). Without such a mechanism it is difficult for the query planner to identify predicates to introduce to exploit a broad array of soft FDs. Our algorithms are more general than previous approaches like BHUNT [BH03] because we are able to exploit correlations in both numeric and non-numeric (e.g. categorical) domains. Our algorithms are also able to identify multi-attribute functional dependencies, where two or more attributes $A_1 \ldots A_n$ are stronger determinants of the value of an attribute B than any of the attributes in $A_1 \ldots A_n$ alone. Consider a database of cities, states, and zipcodes. The pair (`city`, `state`) is clearly a better predictor of `zipcode` than city or state alone, as there are many cities in the US named "Fairview" or "Springfield" but there is typically only one city with a given name in a particular state.

Our second major contribution is to develop an analytical cost model to predict the impact of data correlations on the performance of secondary index look-ups. Although previous work has examined how correlations affect query selectivity, our cost model is the first to describe actual query execution using statistics that are practical to calculate on large data sets. Furthermore, the model is general enough that we use it in our algorithms to search for soft FDs as well as during query optimization. We show that this model is a good match for real world performance.

Our third contribution is to observe that to effectively exploit the correlations identified by any search algorithm (including ours or those in previous work), it may be necessary to create a large number of secondary indexes (one per pair of correlated attributes). Such indexes can be quite large, consuming valuable buffer pool space and dramatically slowing the performance of updates, possibly obviating the advantages gained from correlations. We address this concern by proposing a compressed index structure called a *correlation map*, or CM [Huo07], that compactly represents correlations. By avoiding the need to store an index entry for every tuple and by employing bucketing techniques, we are able to keep the size of CMs to less than a megabyte even for multi-gigabyte tables, thus allowing them to easily fit into RAM.

CMs are simply a mapping from each distinct *value* (not tuple) u in the domain of an attribute A_u to pages in another attribute A_c that contain tuples co-occurring with u in the database. Given a clustered attribute A_c (for which their exists a clustered index), we call attribute A_u the *unclustered* attribute. Queries over A_u can be answered by looking up the co-occurring values of u in the clustered index on A_c to find matching tuples.

We also evaluate the effectiveness of exploiting correlated attributes on several data sets, coming from TPC-H, eBay, and the Sloan Digital Sky Survey (SDSS). We show that correlations significantly improve query processing performance on these workloads. For example, we show that in a test benchmark with 39 queries over the SDSS data set, we are able to obtain more than a 2x performance improvement on 13 of the queries and greater than 16x improvement on 5 of the queries by building an appropriately correlated clustered index on one of the tables. We then show that CMs can capture these same gains during query processing with orders of magnitude less storage overhead. We show that maintenance of CMs (including overheads for recovery) slows the performance of update queries dramatically less than traditional B+Trees. For example, we find that in the Experiment 3 of Section 2.7.2 with 10 CMs or unclustered B+Trees, CMs can sustain an update rate of 900 tuples per second, whereas B+Trees are limited to 29 per second, a factor of 30 improvement.

2.2 Related Work

There is a substantial body of prior work on exploiting correlations in query processing. One can view our work as an extension of approaches from the field of semantic query optimization (SQO); there has been a long history of work in this area [CFPT94, HZ80, Kin81, SO87]. The basic idea is to exploit various types of integrity constraints—either specified by the user or derived from the database—to eliminate redundant query expressions or to find more selective access paths during query optimization.

Previous SQO systems have studied several problems that bear some resemblance to correlation maps. Cheng et al. [CGK+99] describe *predicate introduction* as one of their optimizations (which was originally proposed by Chakravarthy et al [CFPT94] and is the same technique we use in rewriting queries), in which the SQO injects new predicates in the `WHERE` clause of a query based on constraints that it can infer about relevant table attributes; in this case they use logical or algebraic constraints (as in Gryz et al. [GSZZ02]) to identify candidate predicates to insert.

BHUNT [BH03] also explores the discovery of soft constraints, focusing on algebraic constraints between pairs of attributes. The authors explain how to use such constraints to discover access paths during query optimization. For example, the distribution of (`delivery_date` − `ship_date`) in a sales database may cluster around a few common values – roughly 4 days for standard UPS, 2 days for air shipping, etc, that represent "bumps" in delivery dates relative to ship dates. The idea is a generalization of work by Gryz et al. [GSZZ02], who propose a technique for deriving "check constraints," which are basically linear correlations between attributes with error bounds (e.g., $salary = age * 1k \pm 20k$). Godfrey et al. [GGZ01] have also looked at discovering and utilizing "statistical soft constraints" are similar to bumps with confidence measures in BHUNT.

If the widths of the bumps are chosen wisely, BHUNT can capture many algebraic relationships between numeric columns. The authors describe how such constraints can be included in the `WHERE` clause in a query to allow the optimizer to use alternative access paths (for example, by adding predicates like `deliveryDate BETWEEN shipDate+1 and shipDate+3`). Like BHUNT, CMs also are used to identify constraints that can be used to optimize query execution. Unlike BHUNT, CMs are more general because they are not limited to algebraic relations over ordered domains (e.g., BHUNT cannot find correlations between states and zip codes). BHUNT also does not address multi-dimensional correlations or bucketing, which are a key focus of this chapter.

The CORDS system in IBM DB2 [IMH$^+$04] builds on the work of BHUNT by introducing a more sophisticated measure of attribute-pair correlation that captures non-numeric domains. CORDS calculates statistics over samples of data from pairs of attributes that satisfy heuristic pruning rules, and it determines soft (nearly unique) keys, soft FDs, and other degrees of correlation. CMs and CORDS are similar in their measure of soft FD strength and their use of a query training set to limit the search space over candidate attribute sets. Relative to CORDS, our work adds our compressed correlation map structure (CMs), a complete cost model and set of methods to exploit the discovered correlations in query processing using CMs or secondary indices, and a set of techniques to recommend secondary indices / CMs to build. CORDS, on the other hand, focuses on using correlation statistics to improve selectivity estimation during query optimization, and it does not examine how to maintain the information necessary for use during query execution. CORDS also does not find multi-dimensional correlations or explore bucketing. Oracle 11g and PostgreSQL have related statistics [ORA, PSQb], but they are used only for choosing execution plans, too.

Chen et al. describe an approach called ADC Clustering that is related in that it addresses the poor performance of secondary indexes in data warehousing applications [COO08]. ADC Clustering aims to improve the performance of queries in a star schema by physically concatenating the values of commonly queried dimension columns that have restrictions into a new fact table, and then sorting on that concatenation. Though correlations in the underlying data play a major role in the performance of their approach, ADC Clustering does not directly measure correlations or model how they affect query performance.

When paired with an appropriate clustered index, a CM on an unclustered attribute may replace a much larger secondary index structure (such as a B+Tree or bitmap index) by serving as a lossy representation of the index. There has been work on approximate bitmap index structures (e.g. [ACFT06]), where Bloom filters are used to determine which tuple IDs *may* contain a particular attribute value. These techniques do not achieve as much compression as CMs because they represent maps of *tuples* instead of *values*. Also, false positives in approximate bitmaps will result in a randomly scattered set of records that may match a given lookup, whereas bucketing in CMs results in a contiguous range of clustered attribute records that may match a lookup. Work on approximate bitmaps also does not discuss how to choose the index size (the bin width, in our work) to preserve correlations, as we do. Existing work on non-lossy index compression, such as prefix compression [BU77] cannot achieve anywhere near the same compression gains; our experiments with gzip and index compression on our data sets suggest they yield typical size reductions factors of 3–4.

Microsoft SQL Server has a similar technique called *datetime correlation optimization* [MSD]. It maintains a small materialized view to store co-occurring values of two *datetime* columns. When one of the datetime columns is the clustered index key and the other is predicated in a query, the MV is internally used to infer an additional predicate on the clustered index. Though the approach is related to ours, it is unable to capture general correlations. For example, SQL Server is unable to exploit the state-city correlation described in Figure 4 because it supports only datetime columns. In data-warehouse queries, it is unusual for both the clustered key and predicated key to have datetime types. Second, it cannot capture multi-attribute correlations. For example, the (*longitude*, *latitude*)$\rightarrow$*zipcode* example described in Section 6 is inexpressible by SQL Server. Detecting and exploiting such correlations in a scalable way requires a sophisticated analysis. Third, it lacks an adaptive bucketing scheme as described in Section 2.6 to utilize correlations over a wide range of attribute domains. SQL Server always buckets datetime values into month-long ranges. Without an adaptive bucketing scheme, a system would not be able to optimize both the index size (and thus maintenance costs) and query performance unless the attribute domains are evenly and sparsely distributed. Last, SQL Server does not publish an analytic cost model to evaluate the benefit of correlations, which is required to determine the pairs of attributes to exploit. To realize these features, we establish a correlation-aware cost model to evaluate the benefit of correlations and the CM advisor to detect general correlations and design proper bucketing schemes based on workload queries.

2.3 B+Trees and Correlations

We begin with a brief discussion of the costs of conventional database access methods and how they are affected by correlations, before presenting a cost model for predicting the effect of such access methods (Section 2.4) and a discussion of our compressed CM structure (Section 2.5).

For selections when a clustered index is unavailable, a database system has two choices: it may choose to perform a full table scan or use a secondary B+Tree index, if one exists. The cost of each access method depends on well understood factors—table sizes, predicate selectivities, and attribute cardinalities—as well as less well understood factors like correlations. To understand these factors, we begin with a simple cost model and show how it changes in the presence of correlations.

In the following discussion, we assume a table with clustered attribute A_c and secondary attribute A_u on which we query. Table 2.1 summarizes the statistics that we calculate over each table. For the hardware parameters *seek_cost* and *seq_page_cost* the table shows measured values from our experimental platform. We assume that all of the access methods are disk-bound.

Consider a sequential table scan. A table scan incurs no random seek costs, but it must read each page in the table in order of the clustered key. The number of pages p in a table is $\frac{total\_tups}{tups\_per\_page}$. The $cost_{scan}$ of scanning a table is then just $seq\_page\_cost \times p$. We note here that this model is oblivious to external factors such as disk fragmentation and as such underestimates the true cost of a scan in a real database implementation; our numbers show true scan cost to be approximately 10% higher in our implementation.

As our goal in this chapter is to explore secondary index costs, we present cost models for secondary index accesses in the next two sections, and then show how correlations affect such accesses.

2.3.1 Pipelined Index Scan

A secondary B+Tree index is the standard alternative to a table scan, providing an efficient way to access a disk page containing a particular unclustered attribute value. While the B+Tree identifies

Table 2.1: Statistics and parameters used in analytical model.

tups_per_page	Number of tuples that fit on one page.
total_tups	Total number of tuples in the table.
btree_height	Average height of a clustered B+Tree path, root to leaf.
n_lookups	Number of A_u values to look up in one query.
u_tups	Average number of tuples appearing with each A_u value.
seq_page_cost Typical value:	Time to read one disk page sequentially. .078 ms
seek_cost Typical value:	Time to seek to a random disk page and read it. 5.5 ms

the locations where relevant tuples can be found, it cannot guarantee that the tuples are accessed without interleaving seeks. This is because the table may be clustered on a different attribute, and a scan may result in tuple accesses scattered randomly across the physical pages on disk.

In general, if the query executor uses a pipelined iterator model (e.g., performing repeated probes into an index that is the inner relation of a nested loops join) to feed tuples to operators, then a B+Tree operator may need to access unclustered attribute values in an order over which it has no control. If we ignore correlations between the unclustered and clustered attributes, then, each new input value will send the operator on *btree_height* random seeks. The approximate cost of n lookups is then:

$$cost_{uncorrelated} = (n\_lookups)(u\_tups)(seek\_cost)(btree\_height)$$

Since a random seek is so expensive, a pipelined secondary B+Tree operation only makes sense for a small number of specific value lookups. When the set of values to look up is available up front (as in a blocked index nested loops join, or a range selection over a base table), the standard optimization is to sort the index keys before looking them up in the hash table. We call this a *sorted index scan*.

2.3.2 Sorted Index Scan

When the set of all A_u values satisfying the predicate is known up front, the query executor can perform a number of lookups on the unclustered B+Tree and assemble a list of record IDs (RIDs) of all of the actual data tuples in the heap files. The RIDs can then be sorted and de-duplicated. This allows the B+Tree to perform a single sequential sweep to access the heap file, rather than a separate disk seek for each unclustered index lookup. This sweep always performs at least as fast as a sequential scan; it will be faster if it can seek over large regions of the file.

The sorting itself can be implemented in a variety of ways. For example, PostgreSQL uses the index to build a bitmap (with one bit per tuple) indicating the pages that contain records that match predicates [PSQa]. It then scans the heap file sequentially and reads only the pages where corresponding bits are set in the bitmap. In practice, the CPU costs for sorting the offsets is typically negligible compared to the I/O costs saved by the improved access pattern.

2.3.3 The Effect of Correlations

In this section, we show that the performance of a sorted index scan is highly dependent on correlations between the clustered and unclustered values. In particular, a sorted index scan behaves especially nicely when the clustered table value is a good predictor for an unclustered value. To illustrate this, in Figure 2.1 we visualize the distribution of page accesses when performing lookups on an unclustered B+Tree over the `lineitem` table from the TPC-H benchmark. The figure shows the layout of the `lineitem` table as a horizontal array of pages numbered $1 \dots n$. Each black mark indicates a tuple in the table that is read during lookups of three distinct values of the unclustered attribute (either `suppkey` or `shipdate`). The four rows represent four cases (in vertical order):

1. a lookup on `suppkey`; table is clustered on `partkey`
2. a lookup on `suppkey`; table is not clustered
3. a lookup on `shipdate`; table is clustered on `receiptdate`
4. a lookup on `shipdate`; table is not clustered

The `suppkey` is moderately correlated with `partkey`, as each supplier only supplies certain parts. The `shipdate` and `receiptdate` are highly correlated as most products are shipped 2, 4, or 5 days before they are received.

In both cases, where correlations are present, the sorted index scan visits a small number of sequential groups of pages compared to numerous scattered pages when no correlation exists. Particularly striking is the high-correlation case (`shipdate` and `receiptdate`), where the sorted index scan only performs a handful of large seeks to reach long sequential groups of pages. The overall cost of accessing the index on `shipdate` when the table is clustered on `receiptdate` is about 1/20 the cost of accessing it when no clustering is used on the table.

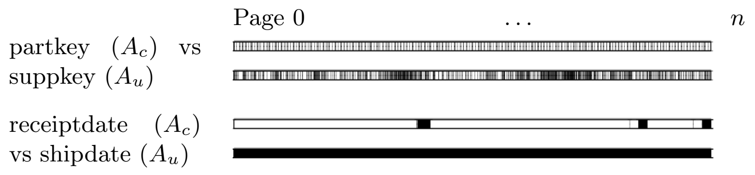

Figure 2.1: Access patterns in **`lineitem`** table for an unclustered B+Tree lookup on A_u (`suppkey`/`shipdate`) with and without clustering on correlated attribute A_c (`partkey`/`receiptdate`).

2.3.4 Experiments

Based on the intuition about the potential benefit of correlations described above, in this section we describe two experiments that demonstrate the actual benefit correlations yield when running queries with unclustered B+Trees We describe our experimental setup and these data sets in more detail in Section 2.7.

Varying the clustered attribute: Figure 2.2 shows the result of an experiment conducted on the SDSS data set to demonstrate that clustering on one well-chosen attribute can speed up many queries.

For this experiment, we devised a simple benchmark consisting of 39 queries, each of which has a predicate over one of the attributes in the PhotoObj table of the SDSS data set with 1% selectivity. This table contains information about the optical properties of various celestial objects, including their color, brightness, and so on. Queries selecting objects with different combinations of these attributes are very common in SDSS benchmarks [GST$^+$02].

We then clustered the table in 39 different ways (once on each of the 39 attributes used in our test queries), and ran all 39 queries over each clustering to measure the benefit that correlations can offer.

Figure 2.2 shows the number of queries that run at least a factor of 2, 4, 8, or 16 times faster than a pure table scan (or an unclustered sorted index scan) when using a secondary index lookup for each choice of clustered attribute. The clustered attribute varies on the horizontal axis. For example, attribute 1 (`fieldID`) is highly correlated with 12 attributes and clustering on it sped up querying on 13 queries by at least a factor of two over a table scan, with 5 of them exhibiting more than a factor of 16 speed-up.

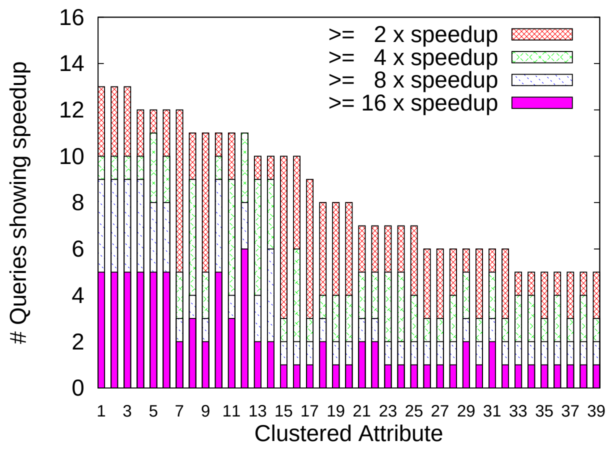

Figure 2.2: Queries accelerated by clustering in PhotoObj table

Introducing correlated clustering: In this experiment we look at the TPC-H attributes shown in Figure 2.1. We again highlight the benefits of a good clustered index choice.

We measured the performance of queries over TPC-H data with two different clustering schemes on the PostgreSQL database. In the first, the `lineitem` table is clustered on `receiptdate`, which is correlated with `shipdate`. In the second, we cluster on the primary key – (`orderkey`, `linenumber`) – which is not correlated with `shipdate`. In both cases, we create a standard secondary B+Tree on `shipdate`. The query used in the experiment is:

SELECT AVG(`extendedprice` * `discount`) FROM LINEITEM WHERE `shipdate` IN [list of 1 to 100 random shipdates]

As the graph in Figure 2.3 shows, the correct choice for the clustered attribute can significantly improve the performance of the secondary B+Tree index. For the uncorrelated case the performance degrades rapidly, reaching the cost of a sequential scan for queries with more than 4 `shipdate`s. This happens because the query on the uncorrelated attribute selects `receiptdate` values that are scattered (approximately 7000 per `shipdate`), so the bitmap scan access pattern touches a large fraction of the `lineitem` table. We have observed the same behavior in other commercial database products. Figure 2.3 also shows that we have a cost model that is able to accurately predict the

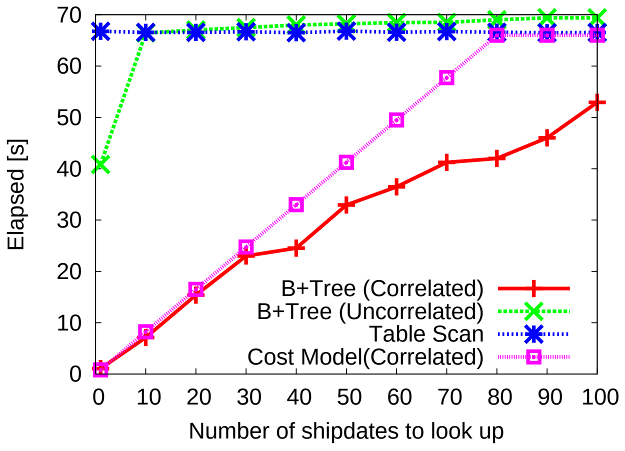

Figure 2.3: Performance of B+Tree with a correlated clustered index on `shipdate` vs. an uncorrelated clustered index

performance of unclustered B+Trees in the presence of correlations – we present this model in detail in the next section.

These experiments show that, *if we had a way to discover these correlations, substantial performance benefits are possible.* Hence, in the next few sections, we focus on our methods for discovering such correlations. In particular, we show how to extend our analytical cost model to capture the effect of correlations (Section 2.4), our algorithms for building compact CM indices (Section 2.5), and finally our approach for discovering correlations (Section 2.6).

2.4 Model of Correlation

In this section, we present our model for predicting the cost of sorted index lookups in the presence of correlations. To the best of our knowledge, this is the first model for predicting query costs that embraces data correlations. As a result, it is substantially more accurate than existing cost models in the presence of strongly correlated attributes.

Table 2.2: Statistics used to measure attribute correlation.

c_tups	Average number of tuples with each A_c value.
c_per_u	Average number of distinct A_c values for each A_u value.

As shown in Table 2.2, we introduce two additional statistics that capture a simple measure of correlation between A_u and A_c. The *c_per_u* value indicates the average number of distinct A_c values that appear in some tuple with each A_u value. The same measure was proposed in CORDS [IMH+04] as the *strength* of a soft FD, where it was used for finding strongly correlated pairs of attributes to update join selectivity statistics rather than building a cost model.

2.4.1 Index Lookups with Correlations

Suppose that we are using a secondary B+Tree that sorts its disk accesses when looking up a set of A_u values as described in Section 2.3.2. For each A_u value v, the query must visit *c_per_u* different clustered attribute values. We need to perform one clustered index lookup to reach each of these clustered attribute values. Once we reach a clustered value, we need to scan at most *c_pages* pages to guarantee finding each tuple containing v. As before, we take the cost of an index lookup to be *btree_height* disk seeks. For each A_c value, we have to scan *c_tups*/*tups_per_page* pages. Finally, as with an uncorrelated sorted index scan, when we scan a large fraction of the file, the access pattern becomes gradually closer to a full table scan. Hence, the index scan is upper bounded by $cost_{scan}$. These observations lead to the following expression for the cost of *n_lookups* on a secondary B+Tree with correlations:

$$\begin{aligned} c\_pages &= c\_tups/tups\_per\_page \\ cost_{sorted} &= \min((n\_lookups)(c\_per\_u)[(seek\_cost)(btree\_height) \\ &\quad + (seq\_page\_cost)(c\_pages)], cost_{scan}) \end{aligned}$$

One simplification in our model is that we ignore the potential overlap between the sets of A_c keys associated with two particular A_u values. In other words, if one A_u value maps to n different A_c values on average, then it is not true in general that two A_u values map to $2n$ different A_c values. Our model may overestimate the number of A_c values involved, and thus the cost of secondary indexes. This is a concern, for example, when evaluating a range predicate over an unclustered attribute that has linear correlation with the clustered attribute (e.g., order receipt dates will overlap heavily for a range of ship dates).

This cost model captures two key facts: first, when both *c_per_u* and *c_pages* are small, the cost of an individual secondary index lookup is not much more than the cost of a lookup on a clustered index (which costs *btree_height* seeks plus a scan of *c_pages*). *c_per_u* will be small when there are few values in the clustered attribute, or when there is a correlation between the clustered attribute and the unclustered attribute. Second, if *c_per_u* is small because the clustered attribute is few-valued, then *c_pages* will likely be large, driving up the cost of each unclustered index access as it will scan a large range of the table.

On the other hand, if *c_per_u* is small due to correlations, *c_pages* is not necessarily large. Hence, if we cluster a table on an attribute that has:

1. A small *c_pages* value and,

2. Correlations to many unclustered attributes (i.e., with a small *c_per_u* value for many unclustered attributes),

then we can expect to be able to exploit this clustering to get good performance from many different secondary indices.

2.4.2 Implementing The Cost Model

To obtain performance predictions from this cost model, we have developed a tool that scans existing tables and calculates the statistics needed by the cost model. Our approach for doing this is quite simple and uses a sampling-based method to reduce the heavy cost of exact parameter calculation.

Given these statistics and measurements of underlying hardware properties, the cost model can predict how much (or if) a given pair of attributes benefits from an unclustered index. The database administrator can use these measurements to choose to build unclustered indexes and to cluster tables on high-benefit attribute pairs that the application is likely to query. In Section 2.7, we present the plots predicted by our cost model alongside our empirical results.

The key measure of correlation that our model relies on is the *c_per_u* statistic. As *c_per_u* is the average number of distinct A_c values for each A_u value, we can calculate its value based on the cardinalities of columns, as follows (this approach is similar to that presented in [IMH+04]). We write the number of distinct values over a pair of attributes A_i and A_j as $D(A_i, A_j)$ and the number of distinct values over a single attribute as $D(A_i)$. Then we can write:

$$c\_per\_u = \frac{D(A_u, A_c)}{D(A_u)}$$

The basic problem of estimating the cardinality of a column has had extensive treatment in both the database and statistics communities, where it is known as the problem of estimating the number of species (e.g. [BF93]).

For estimating single-attribute cardinality, we use the Distinct Sampling (DS) algorithm by Gibbons [Gib01], which computes estimates that are far more accurate than pure sampling-based approaches at a cost of one full table scan. We choose DS over less costly sampling schemes because an error in cardinality estimation for single attributes may cause substantial errors in later database design phases (alternatively, the system catalogs may maintain this statistic accurately).

In Section 2.6 we present our CM advisor that recommends multi-column composite indexes. It is not feasible to use DS for estimating the cardinality of all attribute combinations that our advisor considers, so to estimate composite *c_per_u* measurements we use the Adaptive Estimator (AE) algorithm [CCMN00]. AE estimates composite attribute cardinalities based on a random data sample; we sacrifice some accuracy, but it is very fast because the sample can be kept in memory. These samples are randomly collected during the DS table scan, yielding an optimum random sample as described in [OR95].

2.5 Compressing B-Trees as CMs

We have shown the potential to exploit correlations to make unclustered B+Trees perform more like clustered B+Trees, and we now turn our attention to describe how to efficiently store many unclustered B+Trees in a database system. Our approach uses a compressed B+Tree-like structure called a Correlation Map (CM) that works especially well in the presence of correlations. Compressing secondary indexes is not a new concept: several approaches have been explored in prior work [BU77]. The goals of compression are twofold; first, compression saves disk space, which can be prohibitive when considering numerous indexes on large datasets. Second, compression improves system performance by reducing I/O costs associated with using indexes and by reducing pressure on the buffer pool.

Although disk space is becoming cheaper every day, it is still a limited resource. For large data warehouses in real applications, having many B+Tree indexes can easily require petabytes of disk space and thus does not scale [GST$^+$02]. Existing lossless index compression might achieve up to a 4x reduction in space while, as we show in our experimental section, CMs can reduce index sizes by 3 orders of magnitude. There is an obvious advantage to reducing secondary index sizes – the database requires less disk space and index lookups during query processing consume fewer I/O operations. However, an even more significant improvement comes from reduced index maintenance costs.

Indexes have to be kept up-to-date as the underlying data change through inserts or deletes. Accessing the disk on every update is prohibitively expensive. Thus, the universally applied solution is to keep modified pages in memory and delay writing to disk for as long as possible. Unfortunately, RAM is a much more limited resource than disk space and only a small fraction of a typical B+Tree can be cached. As we show in Experiment 3 of Section 2.7.2, maintaining as few as 5 or 10 B+Trees can lead to a dramatic slowdown in overall system performance. CMs on the other hand can be usually be fully cached in memory as they are quite small; this leads to substantially lower update cost. This means that it is realistic to maintain a large number of CMs, whereas it may not be practical to maintain many conventional B+Trees. In the rest of this section, we describe how CMs are structured, built, and used.

2.5.1 Building and Maintaining CMs

Given that the user wants to build a CM over an attribute $T.A_u$ of a table T (we call this is the *CM Attribute*), with a clustered attribute $T.A_c$, the CM is simply a mapping of the form $u \rightarrow S_c$, where

1. u is a value in the domain of $T.A_u$, and
2. S_c is a set of values in the domain of $T.A_c$ *s.t.* there exists a tuple $t \in T$ of the form $(t.A_u = u,\ t.A_c = c, \ldots)\ \forall c \in S_c$.

For example, if there is a clustered index on `product.state`, a CM on `product.city` might contain the entry "Boston → {NH,MA}," indicating that there is a city called Boston in both Massachusetts and New Hampshire.

The algorithm for building a CM is shown in Algorithm 1. The algorithm works as follows: once the administrator issues a DDL command to create a CM, the system scans the table to build the mapping (line 1). As the system scans the table, it looks up the CM key value in the mapping and adds the clustered index key to the set of key values (line 1). The system tracks the number of times a particular pair of (*uncorrelated*, *correlated*) values occurs using a "co-occurrence" count, which is initialized to 1 (line 1) and incremented as needed (line 1).

The number of times a particular correlated value occurs with each uncorrelated value in the table is needed for deletions. When a tuple t is deleted, the CM looks up the mapping m_{A_u} for the uncorrelated attribute value and decrements the count c for the correlated value $t.A_c$. When c reaches 0, the value $t.A_c$ is removed from m_{A_u}.

The insertion algorithm is very similar to the algorithm for building the table. The main loop (line 1 in Algorithm 1) is simply repeated for each new tuple that is added. Updates can be treated as a delete and an insert.

Since a CM is just a key-value mapping from each unclustered attribute value to the corresponding clustered attribute values, it can be physically stored using any map data structure. This is convenient because database systems provide B+Trees and Hash Indexes that can be used for this purpose. In our implementation, we physically represent a CM using a PostgreSQL table. Whenever a tuple is inserted, deleted, or modified, the CM must be updated as discussed above. Because the CM is relatively compact (containing one key for each value in the domain of the CM attribute, which in our experiments occupy 0.1–1 MB for databases of up to 5 GB), we expect they can generally be cached in memory. Rather than modify PostgreSQL internals, we implemented our own front-end client that caches CMs (see Section 2.7.1). We report the sizes of CMs for several datasets in our experimental evaluation in Section 2.7, showing that they are often much more compact than the equivalent B+Tree.

```
input  : Relation $T$ with attribute $T.A_u$ and clustered index $I$ over attribute $T.A_c$
output: Correlation map $C$, a map from $T.A_u$ values to co-occurring $T.A_c$ values, along with
        co-occurrence count.
$C \leftarrow$ new Map(Value $\rightarrow$ Set) ;
foreach tuple $t \in T$ do
    $m \leftarrow C.get(t.A_u)$ ;
    if $(m.get(t.A_c) = null)$ then
        /* Add fact that $t.A_c$ co-occurred with $t.A_u$ to mapping for $t.A_u$,
           initializing co-occurrence count to 1                              */
        $m.put(t.A_c, 1)$ ;
    end
    else
        /* Increment co-occurrence count for $t.A_c$ in mapping for $t.A_u$      */
        $cnt \leftarrow m.get(t.A_c)$ ;
        $m.put(t.A_c, cnt + 1)$ ;
    end
end
return $C$;
```

Algorithm 1: CM Construction Algorithm

2.5.2 Using CMs

The API for performing lookups on the CM is straightforward; the CM implements a single procedure, `cm_lookup(`$\{v_{u1} \ldots v_{uN}\}$`)`. It takes a set of N CM attribute values as input and returns a list of clustered attribute values that co-occur with $\{v_{u1} \ldots v_{uN}\}$. These clustered attribute values are determined by taking the union of the clustered attribute values returned by a CM lookup on each unclustered value v_{ui}.

Given a list of clustered attribute values to access, the system then performs a sorted index scan on the clustered index. Return values from this scan must be filtered by predicates over the CM attribute, since some values in the clustered index may not satisfy the unclustered predicates – for example, a scan of the states "MA" and "NH" to find records with city "Boston" will encounter many records from non-satisfying cities (e.g., "Manchester".)

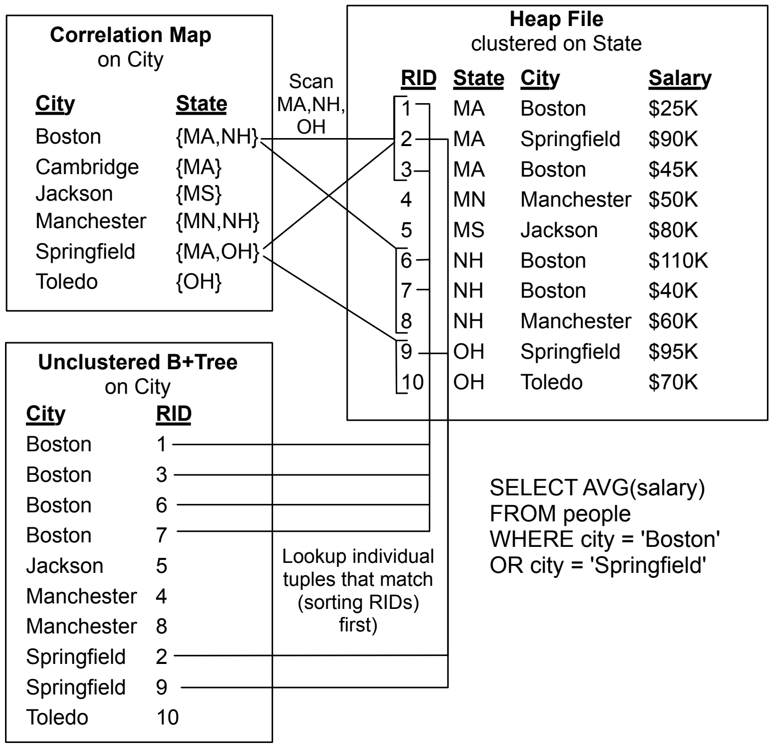

Figure 2.4: Diagram illustrating an example CM and its use in a query plan, and comparing to the use of a conventional B+Tree

Figure 2.4 illustrates an example CM and how it guides the query executor. A secondary B+tree index on `city` is a dense structure, containing an entry for every tuple appearing with each `city`. In order to satisfy the "Boston" or "Springfield" predicate using a standard B+Tree, the query engine uses the index to look up all corresponding `rowids`. The equivalent CM in this example contains all unique pairs (`city`, `state`). To satisfy the same predicate using a CM, the query engine looks up all possible `state` values corresponding to "Boston" or "Springfield". The resulting values ("MA", "NH", "OH") correspond to 3 sequential ranges of `rowids` in the table. These are then scanned and filtered on the original `city` predicate. Notice that the CM scans a superset of the records accessed by the B+Tree, but that it contains fewer entries.

2.5.3 Discussion

CMs capture the correlation between the indexed attribute and the clustered attribute. If the two attributes are well-correlated, each value of the CM attribute will co-occur with only a few values in the clustered attribute, whereas if they are poorly correlated, the CM attribute will co-occur with many clustered attribute values. The degree of compression obtained by replacing a B+Tree with a CM is determined by the degree of correlation as the CM needs to store every unique pair of attributes (A_u, A_c).

We've already seen that CM performance is well-predicted by *c_per_u*. However, there is another condition that affects the performance of correlation maps: they only perform well when the set of relevant clustered attribute values covers a relatively small fraction of the entire table. To see this, consider a correlation with a table clustered on a small-domain attribute, such as gender. Even if the gender attribute is highly correlated with some unclustered attribute, the correlation is unlikely to reduce access costs for most scans of the unclustered attribute, since the system would have to scan about 50% of the table using the CM.

We now briefly describe how CMs can be further compressed through bucketing. An extensive treatment of our approach to bucketing over one or multiple attributes can be found in Section 2.6.

2.5.4 Bucketing CMs

The basic CM approach described in the previous section works well for attributes where the number of distinct values in the CM attribute or the clustered attribute are relatively small. However, for large attribute domains (such as real-valued attributes), the size of the CM can grow quite unwieldy (in the worst case having one entry for each tuple in the table). Keeping a CM small is important to keep the performance benefits outlined above.

We can reduce the size of a CM by "bucketing" ranges of the unclustered attribute together into a single value. We can compress ranges of the clustered attribute stored in the CM similarly. For example, suppose we build a CM on the attribute `temperature` with a clustered index on `humidity` (these attributes are often correlated, with lower temperatures bringing lower humidities). For example, given the unbucketed CM on the left, we can bucket it into the 1^oC or 1% intervals shown on the right via truncation:

$\{12.3^oC\} \rightarrow \{17.5\%, 18.3\%\}$

$\{12.7^oC\} \rightarrow \{18.9\%, 20.1\%\}$ $\{12 - 13^oC\} \rightarrow \{17 - 18\%, 18 - 19\%, 20 - 21\%\}$

$\{14.4^oC\} \rightarrow \{20.7\%, 22.0\%\}$ $\{14 - 15^oC\} \rightarrow \{20 - 21\%, 21 - 22\%, 22 - 23\%\}$

$\{14.9^oC\} \rightarrow \{21.3\%, 22.2\%\}$ $\{17 - 18^oC\} \rightarrow \{25 - 26\%\}$

$\{17.8^oC\} \rightarrow \{25.6\%, 25.9\%\}$

Note that we only need to store the lower bounds of the intervals in the bucketed example above.

The effect of this truncation is to decrease the size of the CM while increasing the number of false positives, since now each CM attribute value maps to a larger range of clustered index values (requiring a scan of a larger range of the clustered index for each CM lookup). We address bucketing in more detail in Section 2.6, and also discuss a sampling-based algorithm we have developed to search for a size-effective bucketing.

The next section describes our *CM Advisor* tool that can identify good candidate attributes for a CM and searches for optimal CM bucketings.

2.6 CM Advisor

In this section, we present our CM Advisor algorithm that searches for good bucketings of clustered attributes and recommends useful CMs to build. There are several reasons why having an automatic designer for CMs is valuable in query processing.

First, a database administrator needs to understand which attributes will benefit from the creation of CMs; although CMs are compact, creating one on every attribute is not possible, especially when allowing composite CMs (over multiple attributes) with different bucketings. Composite CMs are important because there are situations where two attributes can yield stronger correlations with a third attribute than either of the attributes individually. For example an individual `longitude` or `latitude` can occur in many different zipcodes, but a combined (`longitude`, `latitude`) pair lies in exactly one zip code. This property still holds even if longitude and latitude are bucketed using a relatively large bucket size. A traditional (secondary) B+Tree with a composite (`longitude`, `latitude`) key might perform substantially worse than a bucketed CM in such a case as shown in

Experiment 5.

Second, the query optimizer needs to be able to estimate whether a given query should use the CM or not; using the CM adds overhead to query execution. As discussed earlier, CMs work best when a strong correlation exists between the indexed attribute and the clustered attribute. If the correlation is not strong enough, the access pattern using a CM might turn into a sequential scan and thus should not be employed by the query optimizer.

For these reasons, we have developed the *CM Advisor*, an automatic designer for CMs based on the statistics and cost model described in Section 2.4. In this section, we describe how the CM Advisor finds composite correlations from a vast number of possible attribute combinations and proposes promising bucketings that keep the size of the CM small without significantly degrading query performance. Our experimental results show that a well designed composite CM can be both faster than a composite B+Tree index (due to reduced I/O to read from the index and reduced pressure on the buffer pool from index pages) and up to three orders of magnitude smaller.

Before going into the details of the composite CM selection algorithm, we first describe how the CM Advisor chooses possible bucketings for a single attribute that contains many values. We show how to do this for both the clustered and unclustered attribute (on which we build the CM).

2.6.1 Bucketing Many-Valued Attributes

As described in Section 2.5.4, bucketing can dramatically reduce the size of a CM; in particular, bucketing allows the CM Advisor to consider many-valued (even unique) attributes when making CM recommendations. However, we must be careful when choosing bucketing granularity. Very large buckets may result in poor performance by unnecessarily reading large blocks of the correlated attribute, while small buckets produce large data structures, increasing CM access costs (and preventing them from fitting in memory). In this section we describe how our CM Advisor algorithm finds the "ideal" bucketing granularity that strikes a balance between size and performance.

We look at two cases: bucketing the clustered attribute and bucketing the unclustered attributes (the "key" of the CM).

Clustered Attribute Bucketing

If the clustered key is many-valued, the CM structure can become very large even in the presence of a strong correlation between the clustered attribute and the unclustered attribute, since each unclustered attribute value will map to many clustered values. This causes two problems: first, each CM access becomes more expensive due to its size. Second, if we introduce too many query predicates to implement CM scans over clustered attributes (e.g., employing the query rewriting method used in Section 2.7.1), the query plan itself becomes more complex, causing significant overhead in the query optimizer.

To alleviate these problems, the CM Advisor buckets the clustered attribute by adding a new column to the table that represents the "bucket ID." All of the tuples with the same clustered attribute value will have the same bucket ID, and some consecutive clustered attribute values will also have the same bucket ID. The CM then records mappings from unclustered values to bucket IDs, rather than to values of the clustered attribute. CM Advisor performs the actual bucketing during its sequential scan of the table (while computing *c_per_u* statistics). The Advisor begins by assigning tuples to bucket $i = 1$. Once it has read b tuples, it reads the value v of the clustered attribute of the bth tuple. It continues assigning tuples to bucket i until the value of the clustered attribute is no longer v, at which point it starts assigning tuples to bucket $i + 1$ and increments i (this ensures that a particular clustered attribute value is not spread across multiple buckets). This process continues until all tuples have been assigned a bucket.

Wider bucketing causes CM-based queries to read a larger sequential range of the clustered attribute (by introducing false positives), increasing sequential I/O reads but not adding disk seeks. When the bucketing width is chosen well, we have observed that the negative impact of this additional sequential I/O is minimal. To illustrate this, we bucketed the Sloan Digital Sky Survey (SDSS) dataset (see Section 2.7). We then measured the time to run query `SX6`, performing a lookup on two values of the attribute `fieldId`, which is well correlated with the clustered attribute (`ObjID` in this case). We simulated the disk behavior by counting scanned pages and seeks between non-contiguous pages, and then calculated the runtime by applying the statistics in Table 2.1. We varied the bucketing of the clustered attribute from 1 to 40 disk pages per bucket. The results are shown in Table 2.3. We found that performance is relatively insensitive to the bucket size (up to some limit); a value of b such that about 10 pages of tuples map to each bucket appears to work well, taking only about 1 ms longer to read than a bucket size of 1.

Table 2.3: Clustered attribute bucketing granularity and I/O cost

Bucket Size [pgs/bucket]	Pages Scanned	IO Cost [ms]
1	96	15.34
5	105	15.925
10	110	16.25
15	135	17.875
20	140	18.2
40	160	19.5

Bucketing Unclustered Attributes

Bucketing unclustered attributes has a larger effect on performance than bucketing clustered attributes because merging two consecutive values in the unclustered domain will potentially increase the amount of random I/O the system must perform (it will have to look up additional, possibly non-consecutive values in the clustered attribute). This is in contrast to bucketing the clustered attribute, which adds only sequential I/O.

The CM advisor builds equi-width histograms of several different bucket widths from the random data sample (described in Section 2.4.2). Each of these histograms represents one possible bucketing scheme for the attribute under consideration. For a single-attribute CM, the *c_per_u* value for each bucketing can be computed directly from each histogram, by calculating the average number of clustered attribute values that appear in each bin of the histogram, as described in Section 2.4.2. Histograms with fewer, wider bins will have more clustered values per bin and a higher *c_per_u*, whereas histograms with more, narrower bins, will have lower *c_per_u* values. For composite CMs, computing *c_per_u* is more complex, as described in Section 2.6.1. In practice, we find that there is often a "natural" bucketing to the data that results in little increase in *c_per_u* while substantially reducing CM size; we show this effect in our experiments in Section 2.7. Once our CM advisor has constructed all possible histograms (within the bucketing constraints), it iterates through each of them to recommend CMs that will provide good performance, as described in Section 2.6.2 below.

One question that remains unanswered is how to determine the number of different bucketings to consider for each attribute. Our algorithm considers all bucketings that yield between 2^2 and 2^{16} buckets. The bucket *sizes* that we consider scale exponentially. For example, if a column has 100 values, the algorithm considers bucket sizes of $2^1, 2^2, 2^3, 2^4$ and 2^5 (since a bucket size of $2^6 = 64$ yields less than 4 buckets). The limits 2^2 and 2^{16} on the numbers of buckets are configurable, but we found that sufficiently compact bucketing designs often lie within this range in practice.

As another example, Table 2.4 below shows the output from bucketing on the SDSS dataset. Here, CM Advisor outputs attributes such as `mode` and `type`, which are few-valued, without bucketing. For the many-valued attributes `fieldID` and `psfMag_g`, it recommends a series of bucketings that keep the number of buckets in the desired range.

In the case of building a composite CM, we do not directly compute *c_per_u* for each of the single-attribute histograms, but rather pass the possible binnings and the random sample we collected to the composite CM selection algorithm which tries to select a good multi-attribute CM, as we describe next.

Table 2.4: Unclustered attribute bucketings considered for the `SX6` query in the `SDSS` benchmark.

Column	Cardinality	Bucket Widths
`mode`	3	*none*
`type`	5	*none* $\sim 2^1$
`psfMag_g`	196352	$2^2 \sim 2^{16}$
`fieldID`	251	*none* $\sim 2^6$

Bucketing Composite Unclustered Attributes

The number of possible composite CM designs for a given table is very large because there are $\prod_{c=C_1}^{C_N} (Bucketing(c) + 1) - 1$ unique combinations of `N` columns and bucketings (assuming we use the bucketing scheme described above for each attribute). Consider Table 2.4 again. There are two options for the attribute `mode`: whether to include it in the composite CM or not. For `fieldID`, there are eight options: to include it unbucketed, to include it with bucket widths 2^1 through 2^6, or not to include it at all. Similar choices apply for the other attributes. Hence, in total, Table 4 implies $(2 * 3 * 16 * 8) - 1 = 767$ different *candidate designs* for CMs from just 4 attributes. As described in Section 2.4.2 we use Adaptive Estimation (AE) to estimate the combined *c_per_u* for each candidate design. We observed that a sample size of 30,000 tuples gives us reasonably accurate estimates (similar sample size was chosen in [IMH$^+$04]). Using this sample, AE can compute cardinality and bucketing estimates in approximately 5 milliseconds per candidate design.

2.6.2 Recommending CMs

In this section, we explain how the CM Advisor selects (possibly multi-attribute) CMs and bucketings.

Training Queries

A CM can help query execution only when some (or, ideally, all) of its attributes are used as predicates in some query. In other words, an interesting CM design for a query should contain some subset of the predicated attributes. To obtain a set of candidate attributes, our algorithm uses a set of training queries (specified by the DBA) as input. For example, in our SDSS dataset, the DBA might provide the following set of sample queries (alternatively, queries can be collected by monitoring queries at runtime):

Query 1: SELECT ... FROM ... WHERE `ra` BETWEEN
170 AND 190 AND `dec` $<$ 0 AND `mode` $=$ 1
$\Longrightarrow$ { `ra`, `dec`, `mode` }

Query 2: SELECT ... FROM ... WHERE `fieldID` IN (...)
AND `mode` $=$ 1 AND `type` $=$ 6 AND `psfMag_g` $<$ 20
$\Longrightarrow$ { `fieldID`, `mode`, `type`, `psfMag_g` }

Query 3: ...

The goal of the CM Advisor is to output one or more recommended CM designs for each query, along with the expected speed-up factors and CM size estimates. The DBA can then choose which CMs to create. As long as CMs are small, it is reasonable to expect that there will be several CMs on any given table.

Searching for Recommendations

Our CM Advisor exhaustively tries all possible composite index keys and bucketings of attributes for a given training set query. We only consider attributes actually used together in training queries as CM keys. Previous work used similar techniques to prune candidate attribute pairs [BH03, IMH$^+$04]. As most queries refer to a fairly small number of predicates and we only consider a limited number of bucketings (see in Sections 2.6.1 and 2.6.1), this keeps the number of candidate CM designs small. The search space is limited also by removing predicates less selective than some threshold (i.e. >0.5). For the queries in our experiments the longest the CM Advisor ran was about 20 seconds. This is for a query with 5 predicated columns. Given that the CM Advisor is an offline algorithm we believe this is practical.

As we described in Section 2.4, the *c_per_u* value associated with a CM is a good indicator of the expected query runtime improvement. However, a large CM tends to be less useful, even if it has very low *c_per_u*, because it requires too much space to fit into main memory and thus lookup and maintenance become expensive (we further explore the overheads of lookup and maintenance on large index structures in Section 2.7). Therefore simply recommending CM with the lowest *c_per_u* is a poor idea. Instead, our CM designer recommends the smallest CM design within a performance target (defined as a slowdown in query performance relative to an unbucketed design) chosen by the user. We use the cost model developed in Section 2.4 to estimate the performance degradation compared to a secondary B+Tree.

Table 2.5 shows estimated CM sizes for the SDSS dataset, sorted by increased runtime compared to a B+Tree. A performance drop of +3% means that the access method using CM costs 3% more than the access method using a B+Tree for the query. We also show the size ratio of CMs to B+Trees. The CM Advisor recommends the smallest CM design within the user-defined threshold on performance (e.g., up to 10% slowdown compared to a B+Tree), yielding tunable performance. If the Advisor cannot find an unclustered index that is expected to improve performance substantially for a given query, it may recommend that no CM be built.

Table 2.5: CM designs and estimated performance drop compared to secondary B+Trees

Runtime	CM Design	Size Ratio
0%	`psfMag_g`(2^2), `type`, `fieldID`, `mode`	100%
+1%	`psfMag_g`(2^{13}), `type`, `fieldID`, `mode`	24.1%
+3%	`psfMag_g`(2^{14}), `type`, `fieldID`, `mode`	14.6%
+7%	`type`(2^1), `fieldID`	1.4%
+10%	`fieldID`	0.8%
...	...	...

2.7 Experimental Evaluation

In this section, we present an experimental validation of our results. The primary goals of our experiments are to validate the accuracy of our analytical model, to demonstrate the effectiveness of our CM Advisor algorithm, and to compare the performance of CMs and secondary B+Tree indexes.

We ran our tests on a single processor machine with 1G of RAM and a 320G 7200rpm SATA II disk. All experiments were run on PostgreSQL 8.3. We flushed memory caches between runs by using the Linux `/proc/sys/vm/drop_caches` mechanism and by restarting PostgreSQL for each trial. Note that whenever we compare our results to a B+Tree, we are using the standard PostgreSQL

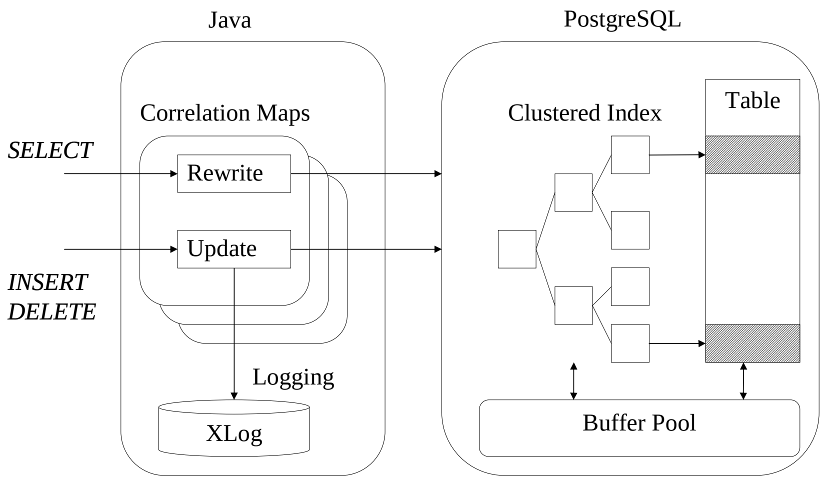

Figure 2.5: Experimental System Overview

secondary index. We also configured PostgreSQL to use a *bitmap index scan* (see Section 2.3.2) when it is beneficial. Because of the flushing and bitmap index scan, we also observed similar performance results with much larger amount of RAM (e.g., 4GB) and data.

2.7.1 System

We prototyped Correlation Maps as a Java front-end application to PostgreSQL as shown in Figure 2.5. All queries are sent to the front-end. Our prototype rewrites SELECT queries to add an IN clause over the clustered attribute. This clause restricts to the clustered attribute values mapped by a predicate on the unclustered attribute. For example, consider the query:

`SELECT * FROM lineitem WHERE receiptdate=`t

If we have a CM over `receiptdate` and a table clustered on `shipdate`, the system might rewrite the query to:

`SELECT * FROM lineitem WHERE receiptdate=`t

        `AND shipdate IN (`$s_1 \ldots s_n$`)`

where $s_1 \dots s_n$ are the `shipdate` values that `receiptdate` t maps to in the CM. PostgreSQL receives queries with the rewritten `IN` clause, which causes it to use the clustered index to find blocks containing matching tuples; by including the original predicate over the unclustered attribute we ensure we receive only tuples that satisfy the original query. For INSERT and DELETE queries, the prototype updates internal CMs as well as table data in PostgreSQL. Although the prototype keeps CMs in main memory and only occasionally flushes to disk on updates, we provide comparable recoverability to a secondary B+Tree index by using a Write Ahead Log (WAL) and flushing the transaction log file during Two-Phase Commit (2PC) in PostgreSQL. We use the PREPARE COMMIT and COMMIT PREPARED commands in PostgreSQL 8.3 to implement the 2PC protocol.

Note that CMs could be implemented as an internal sub-module in a DBMS thereby obviating the need for query rewriting. We expect this would exhibit better performance than the results we present here. We employed the rewriting approach to avoid modifying the internals of PostgreSQL (e.g., its planner and optimizer).

Datasets

Hierarchical Data: The first dataset that we use is derived from eBay category descriptions that are freely available on the web [EBa08]. The eBay data contain 24,000 categories arranged in a hierarchy of sub-categories with a maximum of 6 levels (e.g. antiques $\rightarrow$ architectural & garden $\rightarrow$ hardware $\rightarrow$ locks & keys).

We have populated this hierarchy with unique `ItemID`s. We chose 500 to 3000 `ItemID`s uniformly per category, resulting in a table with 43M rows (occupying 3.5GB on disk). Each category is assigned a unique key value as its Category ID (`CATID`), and the sub-categories for each `CATID` are represented using 6 string-valued fields – `CAT1` through `CAT6`. The median value for the price of each category was chosen uniformly between \$0 and \$1M. Individual prices within a category were generated using a Gaussian around that median with a standard deviation of \$100. Thus, there exists a strong (but not exact) correlation between `Price` and `CATID`. The schema for this dataset is as follows:

$$\text{ITEMS(CATID, CAT1, CAT2, CAT3, CAT4, CAT5, CAT6, ItemID, Price)}$$

TPC-H Data: For our second data source, we chose the `lineitem` table from the TPC-H benchmark, which represents a business-oriented log of orders, parts, and suppliers. There are 16 attributes in total in which we looked for correlations. The table consists of approximately 18M rows of 136 bytes each, for a total table size of 2.5GB at scale 3. The partial schema for this database follows:

$$\text{LINEITEM (orderkey, partkey, suppkey, } \ldots, \\ \text{shipdate, commitdate, receiptdate, } \ldots)$$

SDSS Data: Our third source is the desktop SDSS skyserver [GST+02] dataset which contains 200,000 tuples. We used the fact table `PhotoObj` (shown below) and its partial copy `PhotoTag`.

$$\text{PhotoObj (objID, ra, dec, g, rho, } \ldots)$$

PhotoObj is a very wide table with 446 attributes, while PhotoTag only has a subset of 69 of these attributes. To augment the SDSS dataset to contain a comparable number of tuples to the other datasets, we extended PhotoTag by copying the right ascension (`ra`) and declination (`dec`) windows 10 times in each dimension to produce a 100-fold increase in size (20M rows, 3GB).

2.7.2 Results

In Section 2.3.3 we presented two experiments demonstrating that a secondary index scan performs better when an appropriately chosen clustered index is present and that useful correlations are reasonably common in a real-world data set (SDSS). In this section, we describe the results of a variety of experiments about CMs.

Experiment 1: In our first experiment, we explore the performance implications of using a CM instead of a secondary B+Tree (with an appropriately correlated clustering attribute). Our goal is to demonstrate that CMs capture the same benefits of correlations we showed before. Bear in mind that CMs are substantially smaller than unclustered B+Trees; we measure these size effects and their performance benefit in later experiments. We experimented on the eBay hierarchical dataset clustered on `CATID`. We picked a bucket size of 4096 tuples per bucket for the `Price` attribute (we explain this choice in Experiment 2). We use the following query, varying price ranges as indicated below.

SELECT COUNT(DISTINCT `CAT2`) FROM ITEMS WHERE `Price` BETWEEN 1000 AND 1000+PriceRange

In Figure 2.6, we omit the results for the full table scan as well as for a B+Tree with no correlations, both of which take more than 100 seconds. Here, the CM performs 1s to 4s worse than the secondary index (but still an order of magnitude better than a sequential scan or an index lookup without clustering). This is explained primarily by the increasing number of extraneous heap pages that the CM access pattern reads (which are avoided by the bitmap scan since they do not contain the desired unclustered attribute value), as well as the overhead associated with query rewriting. The observation is that CM performance is competitive, while the data structure is three orders of magnitude smaller (the CM is 0.9MB on disk, the secondary B+Tree is 860MB).

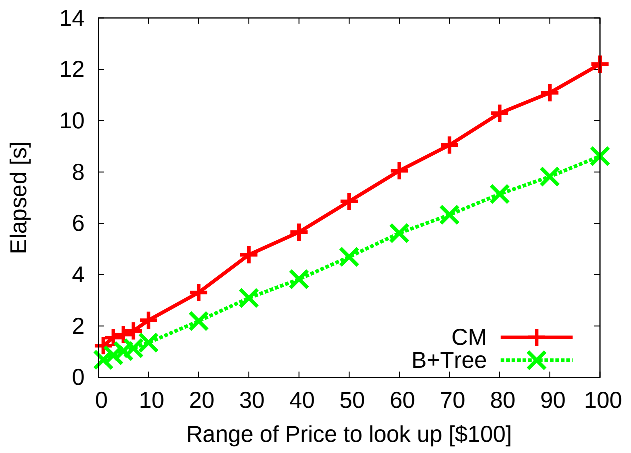

Figure 2.6: Performance of CM and B+Tree index (with correlated clustered attribute) for queries over range of `Price`

Experiment 2: In this experiment, we explore the effects of bucketing. We optimize over bucketing schemes by balancing the performance of the target query and the size of CM. We again use `CATID` as the clustered attribute, but instead of relying on one fixed bucket layout for the unclustered attribute, we vary the bucket size using the approach presented in Section 2.6. We run the query:

SELECT COUNT(DISTINCT `CAT3`) FROM ITEMS WHERE `Price` BETWEEN 1000 AND 1100

The selectivity of this predicate is 6617 rows out of 43M, or 0.000154. In order to evaluate different bucket layouts, we vary the bucket size by powers of two. Therefore, a level of 3 indicates that each bucket holds 2^3 unclustered attribute values.

Looking at Figure 2.7, we see that CM performance is nearly the same as that of the B+Tree up to a bucket level of about 13. With no bucketing, the size of the CM is 350MB, which is already smaller than the PostgreSQL secondary B+Tree (850MB). Observe that as we increase the bucket size, the CM size continues to decrease. It is worth noting that even a CM on a many-valued column like `Price` can become very compact after bucketing.

Figure 2.7 demonstrates a tradeoff between runtime and size. The lookup runtime grows rapidly after the CM hits a particular bucket size. The intuition behind this critical bucket size is the following: if there are two adjacent buckets in the CM that point to the same set of buckets in the clustered index, doubling the CM bucket size has no effect on *c_per_u*. The key bucket size in this example occurs at $2^{13} = 8192$, which is the number of `Price` values closest to the 6617 selected by the range predicate. This shows that there is an "ideal" choice for bucket size that occurs at the knee of the curve.

Experiment 3: In this experiment, we compare the maintenance costs of CMs and secondary B+Trees on eBay data. The table is still clustered on `CATID`, but this time we have multiple CMs and secondary B+Tree indexes on the same columns. We inserted 500k tuples in batches of 10k tuples, which is a standard approach for keeping update overhead low in data warehouses. As shown in Figure 2.8, the total time for inserting 500k tuples quickly deteriorates for B+Trees while for CMs it remains level. Note that we counted all costs involved in maintaining a CM, including transaction logging and 2PC with PostgreSQL.

The reason why the B+Tree's maintenance cost deteriorates for more indexes is that additional B+Trees cause more dirty pages to enter the buffer pool for the same number of INSERTs, leading to more evictions and subsequent page writes to disk. On the other hand, CMs are much smaller than B+Trees and can be kept in memory even when all of their pages are dirty. Therefore, we can maintain a significantly larger number of CMs than B+Trees.

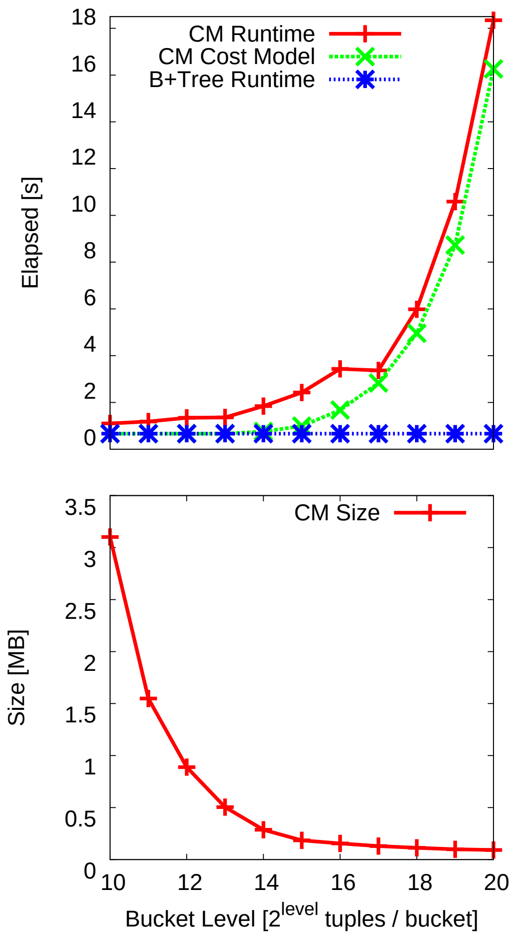

Figure 2.7: Query runtime and CM size as a function of bucket level. The query selects a range of `Price` values.

We also compared the performance of B+Trees and CMs under 50 runs of a mixed workload consisting of INSERTs of 10,000 tuples followed by 100 SELECTs. Here the SELECT query has a predicate on one of `CAT1` to `CAT6`

SELECT AVG(`Price`) FROM ITEMS WHERE `CATX`=X

We randomly chose the predicated attribute and value. The mixed workload gives roughly the same runtime for SELECTs and INSERTs if there is only one B+Tree index. Figure 2.9 shows the total runtime with 5 B+Trees and 5 CMs in the mixed workload, compared against the original INSERT-only workload. The insertion costs on both B+Trees and CMs were higher than the original

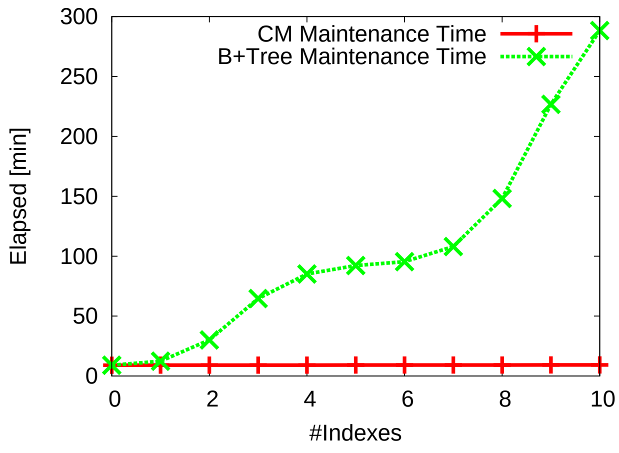

Figure 2.8: Cost of 500k insertions on B+Tree indexes and CMs

workload because SELECT queries consume space in the buffer pool and accelerate the overflow of dirty pages. Interestingly, CMs are faster than B+Trees even for SELECT queries in this mixed workload unlike the read-only workload in Experiment 1 and Experiment 2. This is because SELECT queries over B+Trees frequently have to re-read pages that were evicted from the buffer pool due the many page writes incurred by the updates. In total, 5 CMs are more than 4x faster than B+Trees in the mixed workload; with more secondary indices, the disparity would be more dramatic.

To confirm that correlations benefit both CMs and secondary B+Trees, we also ran the mixed workload for 5 B+Tree indexes after re-clustering the table on `ItemID` (which has no correlation with the predicates). Query performance becomes significantly worse than the times shown in Figure 2.9; the queries take 100x-400x longer because PostgreSQL needs to scan almost the entire table to look up randomly scattered tuples.

In summary, these first experiments demonstrate that properly exploiting correlations can significantly speed up queries both for B+Trees and CMs. However, for multiple B+Trees, maintenance costs quickly deteriorate as the total number of indexes increases. The same effect is not true for CMs because they are so much smaller than B+Trees, and place less pressure on the buffer pool. As a result, we believe CMs provide an ideal way to exploit correlations in secondary index scans.

Experiment 4: In this experiment, we demonstrate that our cost model based on c_per_u captures actual query costs accurately. The data set and clustered key used in this experiment are the same

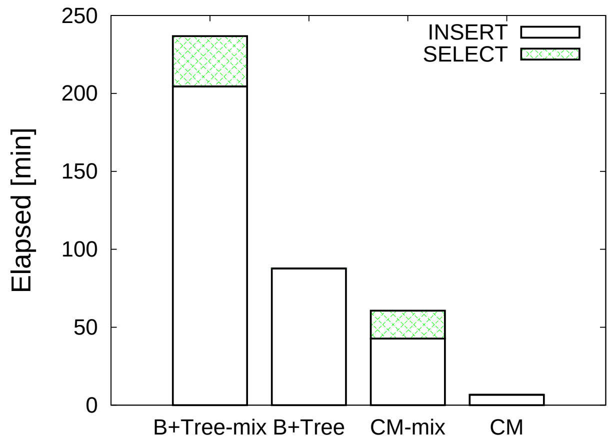

Figure 2.9: Cost of 500k INSERTs and 5k SELECTs on 5 B+Tree indexes and 5 CMs

as in Experiment 1, but we use a different query shown below which has a predicate on `CAT5`:

SELECT AVG(`Price`) FROM ITEMS WHERE `CAT5`=X

In other words, we select over a particular subcategory in the fifth level of the eBay product hierarchy. We build a CM on `CAT5`, which is strongly correlated with `CATID`. We tested different values chosen from the `CAT5` category that exhibited different *c_per_u* counts (ranging from 4 to 145). Our cost model predicts that the CM's performance is primarily determined by how many clustered attribute values the predicated unclustered value corresponds to. As Figure 2.10 shows, this cost model effectively captures the performance of a CM with various *c_per_u* values.

Experiment 5: For our final experiment, we use the SDSS dataset to demonstrate a situation where composite CMs have an advantage over single-attribute CMs as well as secondary B+Tree indexes with a real-world query. This is an example of a non-trivial correlation that was discovered by our CM Advisor tool. The clustered attribute `objID` is correlated strongly with the pair `(ra, dec)`, but the correlation is weaker with each individual attribute. We use the following query, a variant of Q2 from SDSS that identifies objects having blue and bright surfaces within a region.

SELECT COUNT(*) FROM PhotoTag

WHERE `ra` BETWEEN 193.117 AND 194.517

AND `dec` BETWEEN 1.411 AND 1.555

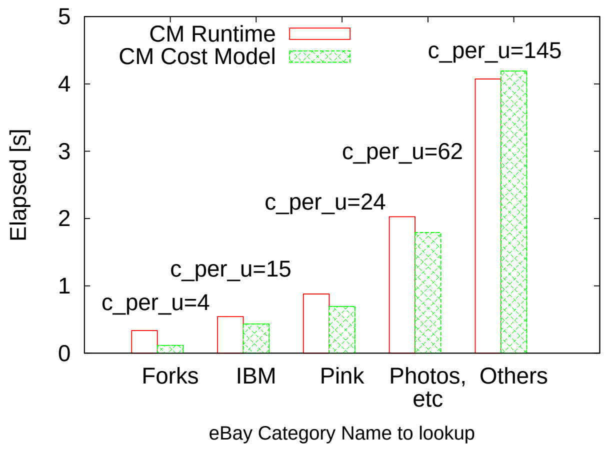

Figure 2.10: CM cost model based on *c_per_u*

AND `g` + `rho` BETWEEN 23 AND 25

We choose the columns and bucket sizes for the CM recommended by the CM Advisor. As we can see in Table 2.6, the composite CM performs much better than a single attribute CM because neither attribute predicts the clustered value but the composition of the attributes does. Both the CM on right ascension and the CM on declination perform worse than the B+Tree index on the pair. However, the CM on the pair of attributes actually performs even better than the B+Tree.

The reason that the composite CM wins is that the B+Tree index performs poorly given multiple range predicates. The secondary index is only used for the range on right ascension, which is the prefix of the compound key. The CM does not have this problem as it is only 699 KB and can be scanned from memory. The size of the secondary index on `(ra, dec)`, on the other hand, is 542 MB.

Table 2.6: Single and composite CMs for an SDSS range query

Index	Bucketing	Runtime[s]	Size[MB]
CM(`ra`)	2^{12}	4.0	0.67
CM(`dec`)	2^{14}	1.7	0.936
CM(`ra, dec`)	2^{14}(`ra`) 2^{16}(`dec`)	0.21	0.699
B+Tree(`ra, dec`)	-	1.12	542

2.7.3 Summary

In this section, we compared the performance of secondary B+Tree and CMs in PostgreSQL on a variety of different data sets and workloads. We showed that CMs and B+Trees can both exploit correlated clustered attributes; that our cost model is a good predictor of performance; and that our CM Advisor can automatically select high performance multi-attribute CMs. We also showed that bucketing can reduce CM size without substantially impacting overall performance, and demonstrated that smaller CMs are substantially cheaper to maintain and keep in main memory, resulting in significantly better overall performance than B+Trees.

2.8 Conclusions

In this chapter, we showed that it is possible to exploit correlations between attributes in database tables to provide substantially better performance from unclustered database indexes than would otherwise be possible. Our techniques exploit correlations by transforming lookups on the unclustered attribute to lookups in the associated clustered index. In order to predict when CMs will exhibit improvements over alternative access methods, we developed an analytical cost model that is suitable for integration with existing query optimizers. Additionally, we described the *CM Advisor* tool that we built to identify correlated attributes and recommend CMs and bucketings that will provide good performance.

Our experimental results over several different data sets validate the accuracy of our cost model and establish numerous cases where CMs dramatically accelerate lookup times over either unclustered B+Trees (without an appropriate clustered column) or sequential scans. We also showed that CMs are much smaller than conventional unclustered B+Trees, making it possible to maintain a large number of them to speed up many different queries. For a workload with updates the compact size of a CM reduces its maintenance overhead over that of the equivalent unclustered B+tree. Based on these results, we conclude that CMs, coupled with our analytical model, have the potential to offer substantial performance gains on a broad class of queries.

We are extending the work in the broader context of *physical database design*. Our work in this chapter assumes a given clustered index. However, if we had the freedom to chose the clustered index (which is fine in a data warehouse) to have stronger correlations with predicated attributes in the workload, we would likely achieve even greater improvement. Towards this goal, the next chapter develops a new physical database designer which chooses a set of materialized views, clustered indexes and CMs so that the correlations between the clustered and unclustered indexes are maximized to optimize the performance for given workload queries within a given space budget. Another extension is to design even more flexible bucketing for skewed value distributions. One possible solution is to consider variable-width buckets that pack more predicated attribute values into a bucket when that bucket has many repeated values for the associated clustered attribute. This approch might further reduce the size of CMs without affecting the query performance.

Chapter 3

Correlation Aware Database Designer: *Designing Correlated MVs and Indexes*

We describe an automatic database design tool that exploits correlations between attributes when recommending materialized views (MVs) and indexes. Although there is a substantial body of related work exploring how to select an appropriate set of MVs and indexes for a given workload, none of this work has explored the effect of correlated attributes (e.g., attributes encoding related geographic information) on designs. Our tool identifies a set of MVs and secondary indexes such that correlations between the clustered attributes of the MVs and the secondary indexes are enhanced, which can dramatically improve query performance. It uses a form of *Integer Linear Programming* (ILP) called *ILP Feedback* to pick the best set of MVs and indexes for given database size constraints. We compare our tool with a state-of-the-art commercial database designer on two workloads, APB-1 and SSB (Star Schema Benchmark—similar to TPC-H). Our results show that a correlation-aware database designer can improve query performance up to 6 times within the same space budget when compared to a commercial database designer.

3.1 Introduction

Correlations are extremely common in the attributes of real-world relational datasets. One reason for this is that databases tend to use many attributes to encode related information; for example, area codes, zip codes, cities, states, longitudes, and latitudes all encode spatial data, using slightly different representations, and these attributes are highly correlated (e.g., a given city name usually occurs in only one or two states.) Similar cases occur in many applications; for example, in a retail database, there might be products (e.g., cars) with manufacturers and models. In the case of cars, a given model is likely made by only one manufacturer (e.g., Ford Escape) for a particular set of years (2000–2009) in a particular set of countries (US), yet these are represented by four different attributes in the database. Correlations also occur due to natural relationships between data; for example, in a weather database, high humidity and high temperatures are correlated, and sunny days are also correlated with hot days. Many other examples are described in recent related work [BH03, KHR$^+$09, COO08].

Previous work has shown that the presence of correlations between different attributes in a relation can have a significant impact on query performance [COO08, KHR$^+$09]. If clustered index keys are well-correlated with secondary index keys, looking up values on the secondary index may be an order of magnitude faster than the uncorrelated case. As a simple example, consider the correlation between city names and state names in a table *People* (`name`, `city`, `state`, `zipcode`, `salary`) with millions of tuples. Suppose we have a secondary index on city names and our query determines the average salary in "Cambridge," a city in both Massachusetts and in Maine. If the table is clustered by state, which is strongly correlated with city name, then the entries of the secondary index will only point to a small fraction of the pages in the heap file (those that correspond to Massachusetts or Maine.) In the absence of correlations, however, the Cantabrigians will be spread throughout the heap file, and our query will require reading many more pages (of course, techniques like bitmap scans, indexes with included columns and materialized views can also be used to improve performance of such queries.) Thus, even if we run the same query on the same secondary index in the two cases, query performance can be an order of magnitude faster with a correlated clustered index. We note that such effects are most significant in OLAP (data-warehouse) applications where queries may scan large ranges, in contrast to OLTP databases that tend to perform lookups of single-records by primary keys.

Moreover, such correlations can be compactly represented if attributes are strongly correlated [BH03, KHR$^+$09]. For example, in Chapter 2, we show that by storing only co-occurring distinct values of the clustered and secondary index keys, the secondary index can be dramatically smaller than conventional dense B+Trees, which store one entry *per tuple* rather than one tuple *per distinct value*.

This means that, by selecting clustered indexes that are well-correlated with predicated attributes, we can reduce the size and improve the performance of secondary indexes built over those attributes. In many cases, these correlations make secondary index plans a better choice than sequential scans.

Although previous work has shown the benefit of correlations on secondary index performance, it has not shown how to automatically select the best indexes for a given workload in the presence of correlations. Conversely, previous work on automated database design has not looked at accounting for correlations. Hence, in this work, we introduce a new database design tool named CORADD (CORrelation Aware Database Designer) that is able to take into account attribute correlations. CORADD first discovers correlations among attributes and then uses this information to enumerate candidate materialized views (MVs) and clustered indexes over them, selecting a set of candidates that offer good query performance in the presence of correlations. To select an optimal set of MVs within a given space budget, CORADD chooses amongst the candidates using an optimized integer linear programming (ILP) technique called ILP Feedback. Finally, it builds compressed secondary indexes on the MVs that take advantage of the strong correlations, offering good performance especially over warehouse-style workloads that can benefit from such indexes. In summary, our contributions include:

- An MV candidate generation method based on query grouping, which identifies groups of queries that benefit from the same MV as a result of correlations;
- Techniques to identify the best clustered attributes for candidate MVs to maximize the benefit from correlations;
- An MV candidate selection method based on integer linear programming and ILP Feedback;
- An implementation and evaluation of CORADD on two data-warehouse benchmarks (SSB [OOC07] and APB-1 [Ola98]), showing that CORADD obtains up to a factor of 6 performance improvement in comparison to a leading commercial product that does not consider correlations.

The rest of the chapter is organized as follows. Section 3.2 summarizes related work on correlations and automatic database design tools. Section 3.3 describes the architecture of CORADD. Section 3.4 describes methods to generate candidate MV designs that exploit correlations. Section 3.5 describes our ILP formulation to pick candidate objects within a given space budget. Section 3.6 describes the ILP Feedback method to adaptively adjust candidate generation. Section 3.7 experimentally compares CORADD with a commercial database designer. Finally, Section 3.8 concludes.

3.2 Background and Related Work

Automatically creating a set of database objects to improve query performance is a well-studied problem in the physical database design literature [ACN00, CN97, PA07]. Most related work takes a query workload as input, enumerates candidate database objects that potentially speed up queries, evaluates the query performance of the objects and then selects objects to materialize.

Most previous work takes a space budget constraint as input to restrict the size of the materialized database objects for two reasons. First, storage is a limited resource. Second, and more importantly, the size of the database is directly linked to its maintenance costs. The cost of inserts or updates rapidly grows as the size of the database grows because additional database objects cause more dirty pages to enter the buffer pool, leading to more evictions and subsequent page writes to disk as shown in the previous chapter..

3.2.1 Exploiting Correlations

Many previous researchers have noted that it is important for query processors to be able to take into account correlations [IMH$^+$04, BH03, KHR$^+$09, COO08]. Chen et al. [COO08] observed that the performance of secondary indexes in data warehousing applications is substantially affected by correlations with clustered indexes. Their Adjointed Dimension Column Clustering aims to improve the performance of queries in a star schema by physically concatenating the values of commonly queried dimension columns that have restrictions into a new fact table, and then creates a clustered index on that concatenation. They report a 10 times or more speed-up when an appropriate clustered index is selected. In Chapter 2, we developed an analytic query cost model that exploits correlations based on a similar observation, which we use to estimate query costs in this chapter. Details of the cost model are in Chapter 2. We reproduce the details of the model in Section 6.6 for the readers' convenience; in short, the cost of a range scan via a secondary index is proportional to the number of distinct values of the clustered index to be scanned. When the secondary index is well-correlated with the clustered index, this number will be small and cost will be low; when it is uncorrelated, this value will be large and cost will be high (close to a sequential scan.) We will refer to this as *our cost model* in the rest of the chapter.

BHUNT [BH03], CORDS [IMH$^+$04], and our work in Chapter 2 discover and exploit correlations in databases. All three use random sampling to find correlations and find many usable correlations across a range of real and synthetic data sets. BHUNT represents discovered correlations as *bumps* and *exception tables* and uses these to rewrite queries. Microsoft SQLServer's *datetime correlation optimization* [1] also stores correlations and uses them to rewrite queries involving dates. In Chapter 2, we developed Correlation Maps (CMs) that use a similar but more general approach; CMs are a type of secondary index that store the clustered index values with which each value of a secondary attribute co-occurs. Then, lookups on the secondary attribute can be performed by scanning the co-occurring clustered attribute values. CMs are thus very compact, as they are a distinct value to distinct value mapping, and can be quite efficient for range queries if the secondary attributes are correlated with a clustered attribute, co-occurring with only a few values.

All of these techniques show that properly clustered primary indexes can improve secondary index performance, but none automatically select the best combination of clustered and unclustered designs to maximize query performance. Previous work [IMH$^+$04, BH03, COO08] does not describe how to search for beneficial correlations, and our work in Chapter 2 only shows how to select beneficial CMs given a clustering (i.e., it does not search for a beneficial clustering.)

Although we believe that any compressed index technique can benefit from our approach, we use CMs in our work here. We believe that CMs are more versatile than similar approaches (CMs support multi-attribute correlations and non-numeric data types). CMs also provide a very flexible compression mechanism and in Chapter 2 we demonstrated that CMs are cheap to maintain even in the presence of heavy inserts.

3.2.2 Candidate Selection

Given a set of candidate database objects (MVs and indexes), a database designer must choose a set of objects to materialize with the goal of maximizing overall query performance within the

[1]`http://msdn.microsoft.com/en-us/library/ms177416(SQL.90).aspx`

space budget. Previous work proposed two different types of selection algorithms: heuristic and optimal. Heuristic algorithms [ACN00, CN97] (e.g., *Greedy(m,k)* [CN97]) often choose a suboptimal set of MVs by making greedy decisions. On the other hand, optimal algorithms like integer linear programming (ILP) [PA07] choose an optimal set in potentially exponential time by exploiting combinatorial optimization techniques. We take the latter approach, further improving the ILP approach from [PA07] in three ways. First, we improve the ILP structure to account for clustered indexes, of which there is only one per table or MV. Next, we propose to use and evaluate a new ILP-related optimization, *ILP Feedback* (see Section 3.6). Lastly, we formulate the design problem so that we do not need to relax integer variables to linear variables, which reduces the error of our approach in comparison to [PA07].

3.3 System Overview

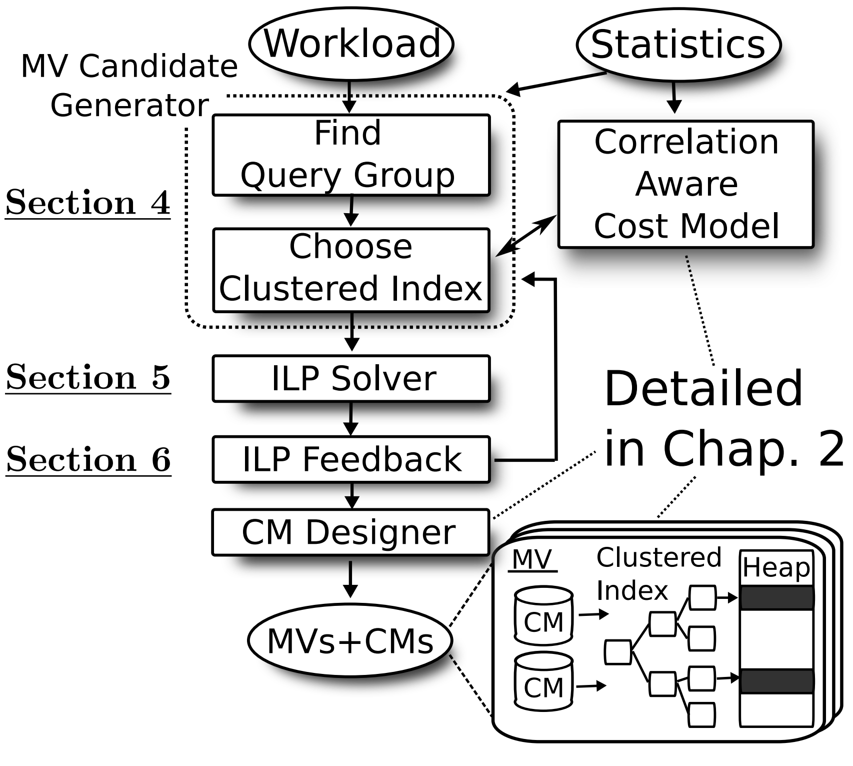

Figure 3.1: CORADD architecture overview

We now describe the design of CORADD, beginning with a system overview. The goal of CORADD is to produce a database design within a given space budget that executes a query workload as quickly as possible – the same goal as previous work [ACN00, PA07] – except that we seek a correlation-aware design, The architecture of CORADD is shown in Figure 3.1. In a nutshell, our design strategy is to choose MVs whose clustered index is well-correlated with attributes predicated in a workload when our cost model identifies cases where such MVs perform significantly better than uncorrelated counterparts.

First, a database workload, expressed as a list of queries that we expect to be executed by the system, is fed into the *MV Candidate Generator* (Section 3.4), which produces a set of MVs. In CORADD, an MV design consists of a *query group* and a clustered index. The query group determines queries the MV can serve and attributes the MV contains. The MV Candidate Generator first selects query groups based on the similarity of their predicates and target attributes, taking into account correlations between those attributes. These query groups are the basis for candidate MV designs, but they do not have clustered indexes at this point. Then, the module produces clustered index designs for the MVs that will take advantage of correlations.

Next, the *ILP Solver* (Section 3.5) selects a subset of the candidate MVs that fit in a given space budget and maximize the performance of the system. As described in Section 3.2, a space budget is also used as a proxy for update cost, allowing CORADD to maximize *read* query performance subject to constraints on update performance or space. We formulate the problem as an integer linear program (ILP), and use an ILP solver to find an optimal solution.

Third, *ILP Feedback* (Section 3.6) sends hints back to the *MV Candidate Generator* to improve both query grouping and clustered index design. This module is inspired by the combinatorial optimization technique called *Column Generation* (CG) [LD05] and provides an efficient way to explore the design space without considering an exponential number of candidate designs. Specifically, the feedback module iteratively selects additional candidate MVs derived from the previous ILP solution and re-solves the ILP to further improve the quality of MV designs.

Finally, the *CM Designer* we developed in [KHR$^+$09] builds CMs on the MVs that are able to exploit correlations between the clustered key and secondary attributes in the MV. Since a CM serves as a secondary index, the output of this stage is a complete database design that is able to answer queries over both clustered and secondary attributes efficiently. For the readers' convenience, a summary of the operation of the CM Designer is reproduced in the previous chapter. Although we have chosen to describe our approach in terms of CMs, CORADD can also work with other correlation-aware secondary index structures like BHUNT [BH03] and SQLServer's date correlation feature.

3.4 MV Candidate Generator

The *MV Candidate Generator* produces an initial set of pre-joined MV candidates, later used by the ILP Solver. Since the number of possible MVs is exponential, the goal of this module is to pick a reasonable number of beneficial MVs for each query. We then use an ILP solver (see Section 3.5) to select a subset of these possible MVs that will perform best on the entire query workload.

As discussed in Section 3.2, secondary indexes that are more strongly correlated with the clustered index will perform better. In generating a database design, we would like to choose MVs with correlated clustered and secondary indexes because such MVs will be able to efficiently answer range queries or large aggregates over either the clustered or secondary attributes. Without correlations, secondary indexes are likely to be of little use for such queries. As an added benefit, correlations can reduce the size of secondary indexes, as described in Section 3.2.1.

A naive approach would be to choose an MV for each query (a *dedicated MV*) which has a clustered index on exactly the attributes predicated in the query. Such an MV could be used to answer the query directly. However, this approach provides no sharing of MVs across groups of queries, which is important when space is limited. The alternative, which we explore in this chapter, is to choose shared MVs that use secondary indexes to cover several queries that have predicates on the same unclustered attributes. We also compare to the naive approach with our approach in the experimental section.

The key to exploiting correlations between clustered and secondary indexes in a shared MV is to choose a clustered index that is well-correlated with *predicates* in the queries served by the MV. Given such a clustered index, secondary indexes are chosen by the CM Designer discussed in Chapter 2 which (at a high level) takes one query at a time, applies our cost model to every combination of attributes predicated in the query and chooses the fastest combination of CMs to create within some space budget for the query.

Because CMs are faster and smaller when they are correlated with the clustered index, the CM Designer naturally selects the most desirable CMs as long as CORADD can produce a clustered index that is well-correlated with the predicates in a query group.

To find such clustered indexes, CORADD employs a two-step process. First, it selects query

Table 3.1: Selectivity vector of SSB

	year	yearmonth	weeknum	discount	quantity
Q1.1	0.15	1	1	0.27	0.48
Q1.2	1	0.013	1	0.27	0.20
Q1.3	0.15	1	0.02	0.27	0.20

Strength (yearmonth → year)=1
Strength (year → yearmonth)=0.14
Strength (weeknum → yearmonth)=0.12
Strength (yearmonth → year,weeknum)=0.19

Table 3.2: Selectivity vector after propagation

	year	yearmonth	weeknum	year,weeknum
Q1.1	0.15	0.15 $(=\frac{0.15}{1})$	1	0.15
Q1.2	0.15 $(=\frac{0.013}{0.14})$	0.013	0.11 $(=\frac{0.013}{0.12})$	0.0162
Q1.3	0.15	0.015 $(=\frac{0.0028}{0.19})$	0.02	0.0028

groups that will form MV candidates by grouping queries with similar predicates and target attributes. Next, it produces clustered index designs for each MV candidate based on expected query runtimes using a correlation-aware cost model. It also produces clustered index designs for fact tables, considering them in the same way as MV candidate objects.

3.4.1 Finding Query Groups

We group queries based on the similarity of their predicates and target attributes. Because each MV can have only one clustered index, an MV should serve a group of queries that use similar predicates in order to maximize the correlation between the attributes projected in the MV. We measure the similarity of queries based on a *selectivity vector*.

Selectivity Vector

For a query Q, the selectivity vector of Q represents the selectivities of each attribute with respect to that query. Consider queries Q1.1, Q1.2, and Q1.3 in the Star Schema Benchmark [OOC07] (SSB), a TPC-H like data warehousing workload, with the following predicates that determine the selectivity vectors as shown in Table 3.1. For example, the selectivity value of 0.15 in the cell (Q1.1, year) indicates that Query 1.1 includes a predicate over year that selects out 15% of the tuples in the table. The vectors are constructed from histograms we build by scanning the database.

Q1.1: `year`=1993 & 1≤`discount`≤3 & `quantity`<25

Q1.2: `yearmonth`=199401 & 4≤`discount`≤6 & 26≤`quantity`≤35

Q1.3: `year`=1994 & `weeknum`=6 &
5≤`discount`≤7 & 26≤`quantity`≤35

Note that the vectors do not capture correlations between attributes. For example, Q1.2 has a predicate `yearmonth`=199401, which implies `year`=1994; thus, Q1.2 actually has the same selectivity on `year` as Q1.3. To adjust for this problem, we devised a technique we call *Selectivity Propagation*, which is applied to the vectors based on statistics about the correlation between attributes. We adopt the same measure of correlation strength as CORDS [IMH+04], namely, for two attributes C_1 and C_2, with $|C_1|$ distinct values of C_1 and $|C_1C_2|$ distinct joint values of C_1 and C_2, $\mathit{strength}(C_1 \rightarrow C_2) = \frac{|C_1|}{|C_1C_2|}$ where a larger value (closer to 1) indicates a stronger correlation. To measure the strength, we use Gibbons' Distinct Sampling [Gib01] to estimate the number of distinct values of each attribute and Adaptive Estimation (AE) [CCMN00] for composite attributes, as in CORDS [IMH+04] and our previous work [KHR+09] (see the previous chapter for the details of cardinality estimation). Using these values, we propagate selectivities by calculating, for a relation with a column set C:

$$\mathit{selectivity}(C_i) = \min_j \left(\frac{\mathit{selectivity}(C_j)}{\mathit{strength}(C_i \rightarrow C_j)} \right)$$

For each query, we repeatedly and transitively apply this formula to all attributes until no attributes change their selectivity.

Table 3.2 shows the vectors for SSB after propagation. Here, `yearmonth` perfectly determines `year`, so it has the same selectivity as `year` in Q1.1. On the other hand, `year` in Q1.2 does not determine `yearmonth` perfectly (but with a strength 0.14). Thus, the selectivity of `year` falls with the inverse of the strength. Put another way, `year` does not perfectly determine `yearmonth`, as each `year` co-occurs with 12 `yearmonth` values. Each `year` value co-occurs with more distinct values of `yearmonth`. CORADD also checks the selectivity of multi-attribute composites when the determined key is multi-attribute (i.e. `year`, `weeknum` in Q1.3).

Grouping by k-means

Our next step is to produce groups of queries that are similar, based on their selectivity vectors. To do this, we use Lloyd's *k-means* [Llo82] to group queries. k-means is a randomized grouping algorithm that finds k groups of vectors, such that items are placed into the group with the nearest mean. In our case, the distance function we use is $[v_1, v_2] = \sqrt{\sum_{a_i \in A} (v_1[a_i] - v_2[a_i])^2}$ where A

is the set of all attributes and v_i is the selectivity vector of query i. We also used k-means++ initialization [AV07] to significantly reduce the possibility of finding a sub-optimal grouping at a slight additional cost. After grouping, each query group forms an MV candidate that contains all of the attributes used in the queries in the group. These k most similar groups are fed into the next phase, which computes the best clustering for each group (see Section 3.4.2).

CORADD considers queries on different fact tables separately. Hence, the candidate generator runs k-means for each fact table with every possible k-value from 1 to the number of workload queries over that fact table. This method assumes each query accesses only one fact table. When a query accesses two fact tables, we model it as two independent queries, discarding join predicates for our selectivity computation.

We studied several other distance metrics and query grouping approaches with different vectors (e.g., inverse of selectivity) but found that the above method gave the best designs. We also note that, because query groups are adaptively adjusted via *ILP feedback* (see Section 3.6) it is not essential for our grouping method to produce optimal groups.

Target Attributes and Weight

As described so far, our grouping method only takes into account predicate selectivities. It is also important to consider which target attributes (i.e., attributes in the SELECT list, GROUP BY, etc) an MV must include in order to answer queries. Consider the following queries from SSB [OOC07]; they have similar selectivity vectors because they directly or indirectly predicate on `year`. However, Q1.2 and Q3.4 have very different sets of target attributes while Q1.1 and Q1.2 have nearly the same set.

Q1.1: SELECT SUM (`price`*`discount`)
    WHERE `year`=1993 & 1≤`discount`≤3 & `quantity`<25

Q1.2: SELECT SUM (`price`*`discount`) WHERE `yearmonth`=
    199401 & 4≤`discount`≤6 & 26≤`quantity`≤35

Q3.4: SELECT `c_city`, `s_city`, `year`, sum(`revenue`)
    WHERE `yearmonthstr` = ‘Dec1997’
    & `c_city` IN (‘UK1’, ‘UK5’) & `s_city` IN (‘UK1’, ‘UK5’)

Now, suppose we have the MV candidates with the sizes and hypothetical benefits shown in Figure 3.2. An MV candidate covering both Q1.1 and Q1.2 is not much larger than the MVs covering each query individually because the queries’ target attributes nearly overlap, while an MV

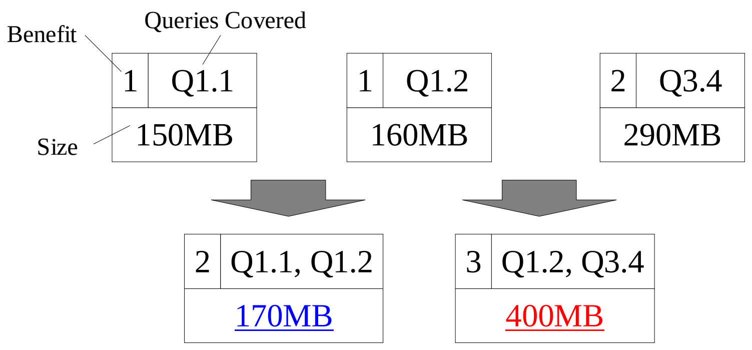

Figure 3.2: Overlapping target attributes and MV size

candidate covering both Q1.2 and Q3.4 becomes much larger than MVs covering the individual queries. Although the latter MV may speed up the two queries, it is unlikely to be a good choice when the space budget is tight.

To capture this intuition, our candidate generator extends the selectivity vectors by appending an element for each attribute that is set to 0 when the attribute is not used in the query and to $bytesize(Attr) \times \alpha$ when it is used. Here, $bytesize(Attr)$ is the size to store one element of $Attr$ (e.g., 4 bytes for an integer attribute) and α is a weight parameter that specifies the importance of overlap and thus MV size. Candidates enumerated with lower α values are more likely to be useful when the space budget is large; those with higher α will be useful when the space budget is tight. When we run k-means, we calculate distances between these extended selectivity vectors, utilizing several α values ranging from 0 to 0.5. We set the upper bound to 0.5 because we empirically observed that α larger than 0.5 gave the same designs as $\alpha = 0.5$ in almost all cases. The final set of candidate designs is the union of the MVs produced by all runs of k-means; hence, it is not important to find α precisely.

3.4.2 Choosing a Clustered Index

The next step is to design a clustered index for each of the query groups produced by k-means. The groups produced in the previous steps consist of similar queries that are likely to have a beneficial clustered index, but k-means provides no information regarding what attributes should form the clustered index key. By pairing clustered indexes with query groups that maximize the performance of the related queries, this step produces a set of MV candidates.

At this point, our goal is not to select the single best design for an MV (this is the job of the ILP described in Section 3.5), but to enumerate a number (t) of possible designs that may work well for the queries. By choosing t possible clusterings for each MV, we provide a parameterized way for our ILP solver to request additional candidates from which it searches for an optimal design. This is important because it does not require us to fix t a priori; instead, our *ILP feedback* method interactively explores the design space, running our clustered index designer with different t values.

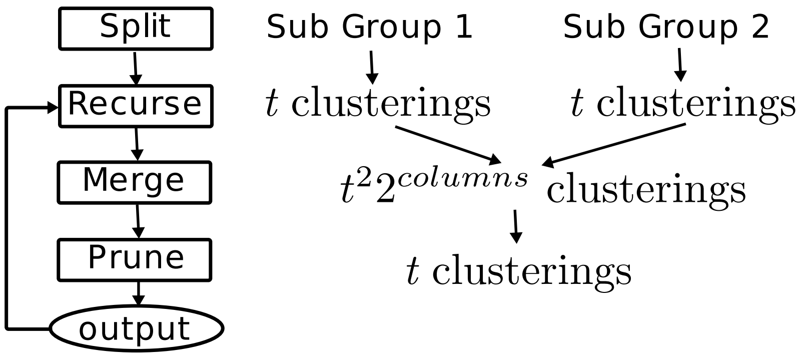

Figure 3.3: Merging method overview

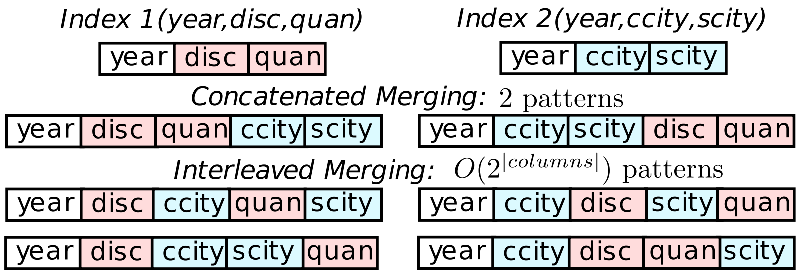
Index 1(year,disc,quan)
Index 2(year,ccity,scity)
year disc quan
year ccity scity
Concatenated Merging: 2 patterns
year disc quan ccity scity
year ccity scity disc quan
Interleaved Merging: $O(2^{|columns|})$ patterns
year disc ccity quan scity
year ccity disc scity quan
year disc ccity scity quan
year ccity disc quan scity

Figure 3.4: Merging via concatenation vs. interleaving

Figure 3.3 illustrates the operation of our clustered index designer. For an MV of only one query, we can produce an optimal design by selecting the clustered index that includes the attributes predicated by the query in order of predicate type (equality, range, IN) and then in selectivity order. We prefer predicates that are less likely to fragment the access pattern on the clustered key (an equality identifies one range of tuples while an IN clause may point to many non-contiguous ranges). We call such candidates *dedicated* MVs and observe that they exhibit the fastest performance for the query. For example, a dedicated MV for Q1.2 is clustered on (`yearmonth`, `discount`, `quantity`).

For an MV with more than one query, we *split* the query group into single-query MVs and *merge* the dedicated clustered indexes, retaining the t clusterings with the best expected runtimes (according to our cost model). This merging approach is similar to [CN99], but differs in that we explore both concatenation and interleaving of attributes when merging two clustered indexes as illustrated in Figure 3.4 while [CN99] considers only concatenation. Although the approach in [CN99] is fast, the queries that benefit from correlations with a secondary index usually see no benefit from a concatenated clustered index key. This is because additional clustered attributes become very fragmented, such that queries over them must seek to many different places on disk. We omit experimental details due to space, but we observed designs that were up to 90% slower when using two-way merging compared to interleaved merging.

Additionally, we reduce the overhead of merging by dropping attributes when the number of distinct values in the leading attributes becomes too large to make additional attributes useful. In practice, this limits the number of attributes in the clustered index to 7 or 8. Furthermore, order-preserving interleaving limits the search space to $2^{|Attr|}$ index designs, rather than all $|Attr|!$ possible permutations.

The final output of this module is t clustered indexes for each MV. We start from a small t value on all MVs; later, *ILP feedback* specifies larger t values to recluster specified MVs, spending more time to consider more clustered index designs for the MVs.

The enumerated MV candidates are then evaluated by the correlation aware cost model and selected within a given space budget by *ILP Solver* described in Section 3.5.

3.4.3 Foreign Key Clustering

In addition to selecting the appropriate clustering for MVs, it is also important to select the appropriate clustering for base tables in some applications, such as the fact table in data warehouse applications (in general, this technique applies to any table with foreign key relationships to other tables.) In many applications, clustering by unique primary keys (PKs) is not likely to be effective because queries are unlikely to be predicated by the PK and the PK is unlikely to be correlated with other attributes. To speed up such queries, it is actually better to cluster the fact table on predicated attributes or foreign-key attributes [KTS+02].

To illustrate this idea, consider the following query in SSB.

Q1.2: SELECT SUM (`price*discount`)
FROM lineorder, date WHERE date.key=lineorder.orderdate &
`date.yearmonth`=199401 & 4≤`discount`≤6 & 26≤`quantity`≤35

One possible design builds a clustered index on `discount` or `quantity`. However, these predicates are not terribly selective nor used by many queries. Another design is to cluster on the foreign key `orderdate`, which is the join-key on the dimension `date` and is indirectly determined by the predicate `date.yearmonth`=199401. This clustered index applies to the selective predicate and also benefits other queries. To utilize such beneficial clusterings, CORADD considers correlations between foreign keys and attributes in dimension tables. We then evaluate the benefit of re-clustering each fact table on each foreign key attribute by applying the correlation-aware cost model. We treat each clustered index design as an MV candidate, implicitly assuming that its attribute set is all attributes in the fact table and the query group is the set of all queries that access the fact table.

Since it is necessary to maintain PK consistency, the fact table requires an additional secondary index over the PK if we re-cluster the table on a different attribute. CORADD accounts for the size of the secondary index as the space consumption of the re-clustered design. While designs based on a new clustered index may not be as fast as a dedicated MV, such designs often speed up queries using much less additional space and provide a substantial improvement in a tight space budget.

The ILP Solver described next treats fact table candidates in the same way as MV candidates, except that it materializes at most one clustered index from the candidates for each fact table.

3.5 Candidate Selection via ILP

In this section, we describe and evaluate our search method to select database objects to materialize from the candidate objects enumerated by the MV Candidate Generator described in the previous section within a given space budget.

3.5.1 ILP Formulation

We formulate the database design problem as an ILP using the symbols and variables listed in Table 3.3. The ILP that chooses the optimal design from a given set of candidate objects is:

$$\textit{Objective}: \quad \textit{min} \sum_{q} \left(t_{q,p_{q,1}} + \sum_{r=2\ldots|M|} x_{q,p_{q,r}} (t_{q,p_{q,r}} - t_{q,p_{q,r-1}}) \right)$$

Table 3.3: Symbols and decision variables

M	Set of MV candidates (including re-clustering designs).		
Q	Set of workload queries.	F	Set of fact tables.
R_f	Set of re-clustering designs for fact table $f \in F$. $R_f \subset M$.		
m	An MV candidate. $m = 1, 2, .., \lvert M \rvert$.	q	A query. $q = 1, 2, .., \lvert Q \rvert$.
S	Space budget.	s_m	Size of MV m.
$t_{q,m}$	Estimated runtime of query q on MV m.		
$p_{q,r}$	r-th fastest MV for query q. $(r_1 \leq r_2 \Leftrightarrow t_{q,p_{q,r_1}} \leq t_{q,p_{q,r_2}})$. $t_{q,m}$ and $p_{q,r}$ are calculated by applying the cost model to all MVs.		
$x_{q,m}$	Whether query q is penalized for not having MV m. $0 \leq x_{q,m} \leq 1$		
y_m	Whether MV m is chosen.		

Subject to:

$$(1)\ y_m \in \{0, 1\} \qquad (2)\ 1 - \sum_{k=1}^{r-1} y_{p_{q,k}} \leq x_{q,p_{q,r}} \leq 1$$

$$(3)\ \sum_{m} s_m y_m \leq S \qquad (4)\ \forall f \in F : \sum_{m \in R_f} y_m \leq 1$$

The ILP is a minimization problem for the objective function that sums the total runtimes of all queries. For a query q, its expected runtime is the sum of the runtime with the fastest MV for q ($t_{q,p_{q,1}}$) plus any *penalties*. A penalty is a slow-down due to not choosing a faster MV for the query, represented by the variable $x_{q,m}$. Condition (1) ensures that the solution is boolean (every MV is either included or is not). Condition (2) determines the penalties for each query by constraining $x_{q,m}$ with y_m. Because this is a minimization problem and $t_{q,p_{q,r}} - t_{q,p_{q,r-1}}$ is always positive (as the expected runtimes are ordered by r), $x_{q,m}$ will be 0 if m or any faster MV for q is chosen, otherwise it will be 1. For example, $x_{q,m}$ is 0 for all m when $y_{p_{q,1}} = 1$ (the fastest MV for query q is chosen). When $y_{p_{q,2}} = 1$ and $y_{p_{q,1}} = 0$, $x_{q,p_{q,1}}$ is 1 and all the other $x_{q,m}$ are 0, thus penalizing the objective function by the difference in runtime, $t_{q,p_{q,2}} - t_{q,p_{q,1}}$. Condition (3) ensures that the database size fits in the space budget. Condition (4) ensures that each fact table has at most one clustered index.

We solve the ILP formulated above using a commercial LP solver. The resulting variable assignment indicates MVs to materialize ($y_m = 1$) and we determine the MV to use for each query by comparing the expected runtime of the chosen MVs. We count the size of CMs separately, as described in Section 3.5.4.

3.5.2 Comparison with a Heuristic Algorithm

To understand the strength of our optimal solution, we compared our ILP solution against Greedy (m,k) [CN97, ACN00], a heuristic algorithm used in Microsoft SQL Server that starts by picking m candidates with the best runtime using exhaustive search and then greedily choosing other candidates until it reaches the space limit or k candidates. As [CN97] recommends, we used the parameter value $m = 2$ (we observed that $m = 3$ took too long to finish). The dataset and query sets are from SSB.

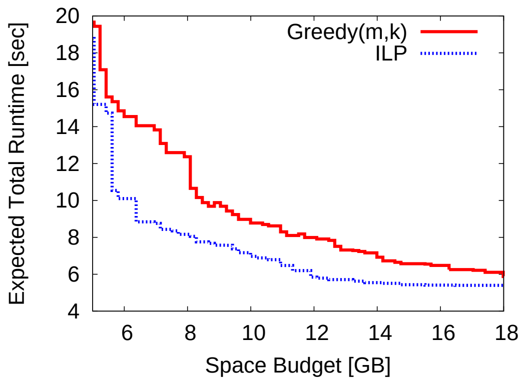

Figure 3.5: Optimal versus greedy.

As Figure 3.5 shows, the ILP solution is 20-40% better than Greedy (m,k) for most space budgets. This is because Greedy (m,k) was unable to choose a good set of candidates in its greedy phase. On the other hand, Greedy (m,k) chooses optimal sets in tight space budgets (0-4GB) where the optimal solutions contain only one or two MVs and the exhaustive phase is sufficient.

3.5.3 Shrinking the ILP

To quickly solve the ILP, CORADD reduces the number of MVs by removing *dominated* MVs which have larger sizes and slower runtimes than some other MV for every query that it covers. For

Table 3.4: MV1 dominates MV2, but not MV3.

	MV1	MV2	MV3	...
Q1	1 sec	5 sec	5 sec	...
Q2	N/A	N/A	5 sec	...
Q3	1 sec	2 sec	5 sec	...
Size	1 GB	2 GB	3 GB	...

example, MV2 in Table 3.4 has a slower runtime than MV1 for every query that can use it (Q1 and Q3), yet MV2 is also larger than MV1. This means MV2 will never be chosen because MV1 always performs better. As for MV3, it is larger than MV1 and has worse performance on queries Q1 and Q3, but it is not dominated by MV1 because it can answer Q2 while MV1 cannot.

For the SSB query set with 13 queries, CORADD enumerates 1,600 MV candidates. After removing dominated candidates, the number of candidates decreases to 160, which leads to an ILP formulation with 2,080 variables and 2,240 constraints. Solving an ILP of this size takes less than one second.

To see how many MV candidates we can solve for in a reasonable amount of time, we formulated the same workload with a varying numbers of candidates. As Figure 3.6 shows, our ILP solver produces an optimal solution within several minutes for up to 20,000 MV candidates. Given that 13 SSB queries produced only 160 MV candidates, CORADD can solve a workload that is substantially more complex than SSB as shown in Section 3.7.

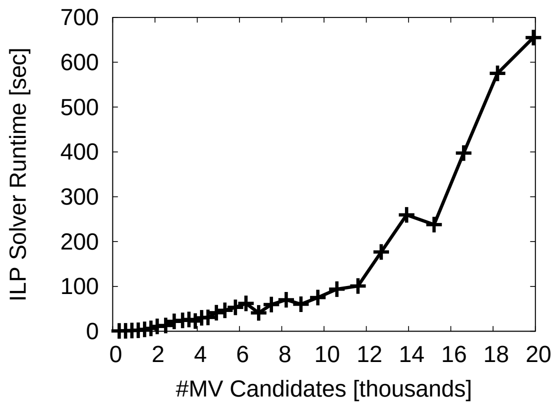

Figure 3.6: LP solver runtime.

Finally, to account for the possibility that each query appears several times in a workload, CORADD can multiply the estimated query cost by the query frequency when the workload is compressed to include frequencies of each query.

3.5.4 Comparison to other ILP-based designers

Papado et al [PA07] present an ILP-based database design formulation that has some similarity to our approach. However, our ILP formulation allows us to avoid relaxing integer variables to linear variables, which [PA07] relies on.

Because compressed secondary indexes are significantly smaller than B+Trees, CORADD can simply set aside some small amount of space (i.e. 1 MB*$|Q|$) for secondary indexes and then enumerate and select a set of MVs independently of its choice of secondary indexes on them (as in *MVFIRST* [ACN00]). This gives us more flexibility in the presence of good correlations. As for [PA07], they must consider the interaction between different indexes; therefore their decision variables represent sets of indexes, which can be exponential in number. For this reason, [PA07] relaxes variables, leading to potentially arbitrary errors. For example, in one experiment, Papado et al converted the relaxed solution to a feasible integer solution by removing one of the indexes, resulting in a 32% lower benefit than the ILP solution. In contrast, our approach substantially reduces the complexity of the problem and arrives at an optimal solution without relaxation.

3.6 ILP Feedback

So far, we have described how to choose an optimal set of database objects from among the candidates enumerated by the MV Candidate Generator. In general, however, the final design that we propose may not be the best possible database design, because we prune the set of candidates that we supply to the ILP. In this section, we describe our iterative *ILP feedback* method to improve candidate enumeration. To the best of our knowledge, no prior work has used a similar technique in physical database design.

Our ILP may not produce the best possible design due to two heuristics in our candidate enumeration approach – query grouping by k-means, and our selection of clustered indexes by merging. Simply adding more query groups or increasing the value of t in the clustered index designer produces many (potentially $2^{|Q|}$) more candidates, causing the ILP solver to run much longer. We tackle this problem using a new method inspired by the combinatorial optimization technique known as delayed column generation, or simply *column generation* (CG) [LD05].

3.6.1 Methodology

Imagine a comprehensive ILP formulation with all possible MV candidates constructed out of $2^{|Q|}-1$ query groupings and $2^{|Attr|}-1$ possible clustered indexes on each of them. This huge ILP would give the globally optimal solution, but it is impractical to solve this ILP directly due to its size. Instead, CORADD uses the MV candidate generator described in Section 3.4 which generates a limited number of *initial* query groupings and clustered indexes.

To explore a larger part of the search space than this initial design, we employ two feedback heuristics to generate new candidates from a previous ILP solution. ILP Feedback iteratively creates new MV candidates based on the previous ILP design and re-solves the ILP until the feedback introduces no new candidates or reaches a time limit set by the user.

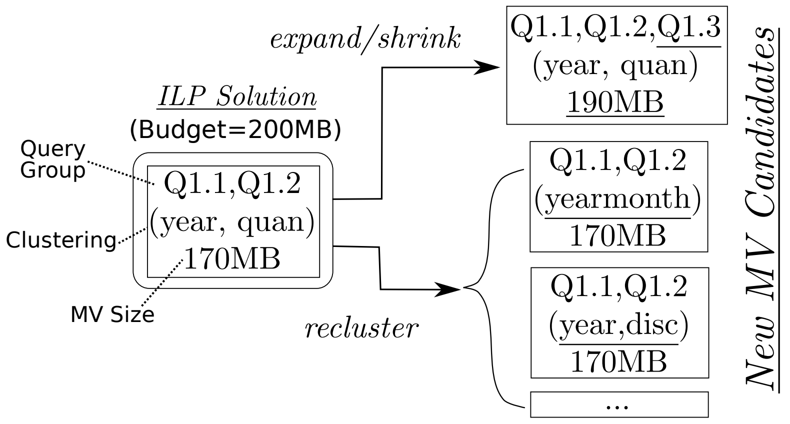

Figure 3.7: ILP feedback

The first source of feedback is to *expand* query groups used in the ILP solution. If the previous solution selects an MV candidate m, expanding m's query group involves adding a new query (by including that query's columns in the MV). We consider an expansion with every query not in the query group as long as it does not exceed the overall space budget. This feedback is particularly helpful in tight space budgets, where missing a good query group that could cover more queries is a major cause of suboptimal designs. Additionally, when m is used in the previous solution but is not chosen to serve some query that it covers (because another MV candidate is faster for that query), we also *shrink* the query group in the hope of reducing the space consumption of m. For example, in Figure 3.7, the initial ILP solution chooses an MV candidate for the query group (Q1.1, Q1.2) with a clustered index on (`year`, `quantity`). As this leaves 30 MB of unused space budget, we add an expanded query group (Q1.1, Q1.2, Q1.3) which does not put the overall design over-budget. If this new candidate is chosen in the next ILP iteration, it will speed up Q1.3.

The second source of feedback is to *recluster* query groups used in the ILP solution. The size of an MV is nearly independent of its choice of clustered index because the B+Tree size is dominated by the number of the leaf nodes. So, when an MV candidate m is chosen in the ILP solution, a better clustered index on m might speed up the queries without violating the space limit. To this end, we invoke the clustered index designer with an increased t-value in hopes of finding a better clustered index. The intuition is that running with an increased t value for a few MVs will still be reasonably fast. This feedback may improve the ILP solution, especially for large space budgets, where missing a good clustered index is a major cause of suboptimal designs because nearly all queries are already covered by some MV. For example, in the case above, we re-run the clustered index designer for this MV with the increased t and add the resulting MV candidates to the ILP formulation. Some of these new candidates may have faster clustered indexes for Q1.1 and Q1.2.

3.6.2 ILP Feedback Performance

To verify the improvements resulting from ILP feedback, we compared the feedback-based solution, the original ILP solution, and the OPT solution for SSB. OPT is the solution generated by running ILP on all possible MV candidates and query groupings. We obtained it as a baseline reference by running a simple brute force enumeration on 4 servers for a week. Note that it was possible to obtain OPT because SSB only has 13 queries ($2^{13} - 1 = 8191$ possible groups); for larger problems this would be intractable.

Figure 3.8 compares the original ILP solution and the ILP feedback solution, plotting the expected slowdown with respect to OPT. Employing ILP feedback improves the ILP solution by about 10%. More importantly, the solution with feedback actually achieves OPT in many space budgets. Note that the original ILP solution could incur more than 10% slowdown if SSB had more attributes with more complicated correlations.

As for the performance of ILP feedback, it took the SSB workload 2 iterations to converge and the approach added only 700 MV candidates to the 1,600 original candidates in the ILP, adding 10 minutes to 17 minutes total designer runtime. Therefore, we conclude that ILP feedback achieves

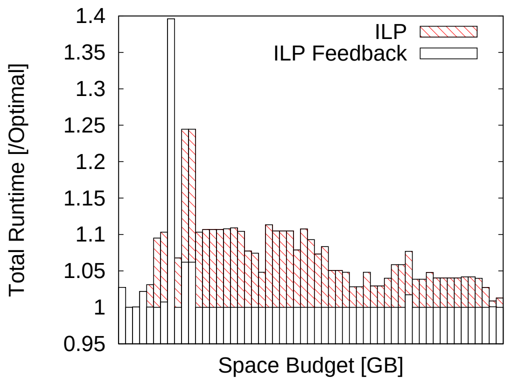

Figure 3.8: ILP feedback improvement.

nearly optimal designs without enumerating an excessive number of MV candidates.

3.7 Experimental Results

In this section, we study the performance of designs that
CORADD produces for a few datasets and workloads, comparing them to a commercial database designer.

We ran our designs on a popular commercial DBMS running on Microsoft Windows 2003 Server Enterprise x64 Edition. The test machine had a 2.4 GHz Quad-core CPU, 4 GB RAM and 10k RPM SATA hard disk. To create CMs in the commercial database, we introduced additional predicates that indicated the values of the clustered attributes to be scanned when a predicate on an unclustered attribute for which an available CM was used (see the previous chapter for the details of this technique.)

We compared CORADD against the DBMS's own designer, which is a widely used automatic database designer based on state-of-the-art database design techniques (e.g., [ACN00, CN97].)

To conduct comparisons, we loaded datasets described in the following section into the DBMS, ran both CORADD and the commercial designer with the query workload, and tested each design on the DBMS. We discarded all cached pages kept by the DBMS and from the underlying OS before running each query.

3.7.1 Dataset and Workload

The first dataset we used is APB-1 [Ola98], which simulates an OLAP business situation. The data scale is 2% density on 10 channels (45M tuples, 2.5 GB). We gave the designers 31 template queries as the workload along with the benchmark's query distribution specification. Though CORADD assumes a star schema, some queries in the workload access two fact tables at the same time. In such cases, we split them into two independent queries.

The second dataset we used is SSB [OOC07], which has the same data as TPC-H with a star schema workload with 13 queries. The data size we used is Scale 4 (24M tuples, 2 GB). For experiments in this section, we augmented the query workload to be 4 times larger. The 52 queries are based on the original 13 queries but with varied target attributes, predicates, GROUP-BY, ORDER-BY and aggregate values. This workload is designed to verify that our designer works even for larger and more complex query workloads.

3.7.2 Results

Experiment 6: In the first experiment, we ran designs produced by CORADD and the commercial designer on APB-1. Figure 3.9 shows the total expected runtime of both designs for each space budget (determined by the cost model) as well as the total real runtime.

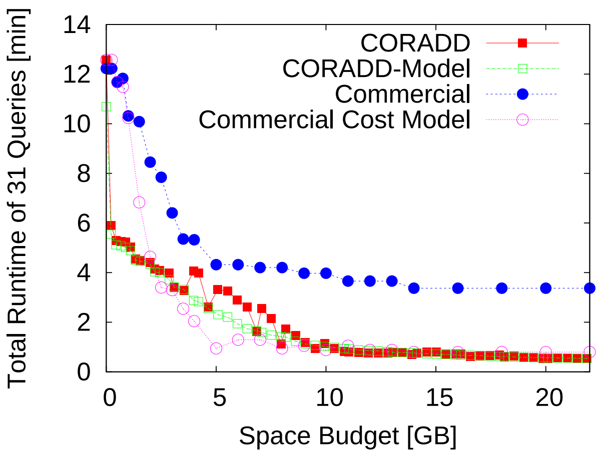

Figure 3.9: Comparison on APB-1.

The expected runtime of CORADD (`CORADD-Model`) matched the real runtime (`CORADD`) very well and, as a consequence, it almost monotonically improves with increased space budgets. Both expected and real runtimes rapidly improved at the 500 MB point where the fact tables are re-clustered to cover queries, and 1 GB to 8 GB points where MVs start to cover queries. After 8 GB where all queries are already covered by MVs, the runtime gradually improves by replacing large MVs with many small MVs that have more correlated clustered indexes. Also at that point, CORADD stops re-clustering the fact table (saving an additional secondary index on the primary key), spending the budget on MVs instead.

Compared with the designs produced by the commercial designer (`Commercial`), our designs are 1.5–3 times faster in tight space budgets (0–8 GB) and 5–6 times faster in larger space budgets (8–22 GB). The commercial designer's cost model estimates the runtime of its designs to be much faster than reality (shown by `Commercial Cost Model`). The error is up to 6 times and worse in larger space budgets where the designer produces more MVs and indexes.

To see where the error comes from, we ran a simple query using a secondary B+Tree index on SSB Scale 20 *lineorder* table. We varied the strength of correlation between the clustered and the secondary index by choosing different clustered keys. Here, fewer *fragments* indicate the correlation is stronger (see the cost model in the previous chapter for more detail). As Figure 3.10 shows, the commercial cost model predicts the same query cost for all clustered index settings, ignoring the effect of correlations. This results in a huge error because the actual runtime varies by a factor of 25 depending on the degree of correlation.

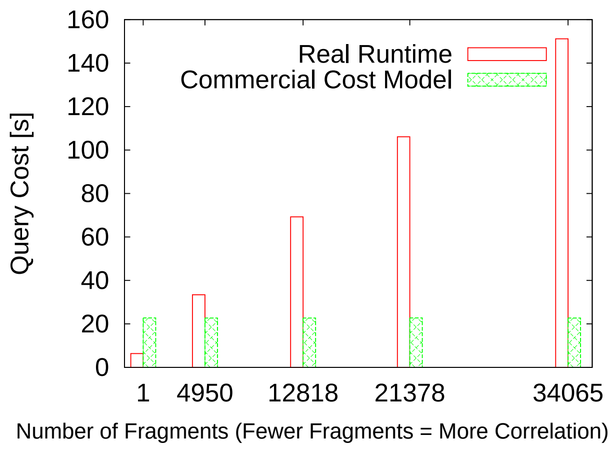

Figure 3.10: Errors in cost model.

Due to its lack of awareness of correlations, the commercial designer tends to produce clustered indexes that are not well-correlated with the predicated attributes in the workload queries, causing many more random seeks than the designer expects. CORADD produces MVs that are correlated with predicated attributes and hence our designs tend to minimize seeks. Also, our cost model accurately predicts the runtime of our designs.

Experiment 7: This experiment is on the augmented (52 queries) SSB. We again ran CORADD and the commercial designer to compare runtimes of the results. This time, we also ran designs produced by a much simpler approach (*Naive*) than CORADD but with our correlation-aware cost model. *Naive* produces only re-clusterings of fact tables and dedicated MVs for each query without query grouping and picks as many candidates as possible. Although this approach is simple, it loses an opportunity to share an MV between multiple queries, which CORADD captures via its query grouping and merging candidate generation methods.

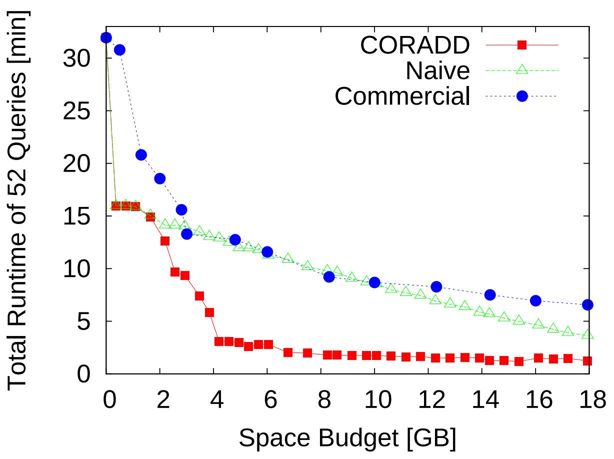

Figure 3.11: Comparison on augmented SSB.

As shown in Figure 3.11, our designs are again 1.5–2 times better in tight space budgets (0–4 GB) and 4–5 times better in larger space budgets (4–18 GB). Even designs produced by *Naive* approach are faster than the commercial designer's in tight space budgets (< 3 GB) because it picks a good clustered index on the fact table, and in larger budgets (> 10 GB) because our cost model accurately predicts that making more MVs with correlated clustering indexes will improve the query performance. However, the improvement by adding MVs is much more gradual than in designs of

CORADD. This is because *Naive* uses only dedicated MVs, and a much larger space budget is required to achieve the same performance as designs with MVs shared by many queries via compact CMs.

Finally, we note that the total runtime of CORADD to produce all the designs plotted in Figure 3.11 was 7.5 hours (22 minutes for statistics collection, an hour for candidate generation, 6 hours for 3 ILP feedback iterations) while the commercial designer took 4 hours. Although CORADD took longer, the runtime is comparable and the resulting performance is substantially better.

3.8 Conclusions

In this chapter, we showed how to exploit correlations in database attributes to improve query performance with a given space budget. CORADD produces MVs based on the similarity between queries and designs clustered indexes on them using a recursive merging method. This approach finds correlations between clustered and secondary indexes, enabling fast query processing and also compact secondary indexes via a compression technique based on correlations. We introduced our ILP formulation and ILP Feedback method inspired by the Column Generation algorithm to efficiently determine a set of MVs to materialize under given space budget.

We evaluated CORADD on the SSB and the APB-1 benchmarks. The experimental result demonstrated that a correlation-aware database designer with compressed secondary indexes can achieve up to 6 times faster query performance than a state-of-the-art commercial database designer with the same space budget.

Chapter 4

A Constraint-Programming Approach for Index Deployment: *Optimizing Index Deployment Orders*

Many database applications today need to deploy hundreds or thousands of indexes on large tables to speed up query execution. Despite a plethora of prior work on selecting a *set* of indexes, no one has studied optimizing the *order* of index deployment. This problem should not be overlooked because an effective index deployment ordering can produce (1) a prompt query runtime improvement and (2) a reduced total deployment time. However, optimizing the problem is challenging because of complex index interactions and a factorial number of possible solutions.

In this chapter we formulate the problem in a mathematical model and study several techniques for solving the index ordering problem. We demonstrate that Constraint Programming (CP) is a more flexible and efficient platform to solve the problem than other methods such as mixed integer programming and A* search. In addition to exact search techniques, we also studied local search algorithms to find near optimal solution very quickly.

Our empirical analysis on the TPC-H dataset shows that our pruning techniques can reduce the size of search space by tens of orders of magnitude. Using the TPC-DS dataset, we verify that our local search algorithm is a highly scalable and stable method for quickly finding a near-optimal solution.

4.1 Introduction

Indexes are the crux of query optimization in databases. The *selection* and *deployment* of indexes has always been one of the most important roles of database administrators (DBAs). As recent software mandates complex data processing over hundreds or thousands of tables, selecting an appropriate set of indexes has become impossible for human DBAs. Therefore, both industry and academia have intensively focused their study on the automatic selection of indexes in physical database design [CN97]. Consequently, every modern commercial database management system (DBMS) ships an automatic design tool as its key component. These design tools support DBAs by suggesting sets of indexes that dramatically improve query execution.

Nonetheless, little effort has been made to study another important aspect of indexing; **deployment**. Deploying indexes is a very costly operation and DBAs give it as much care and attention as possible. It consumes immense hardware resources and takes a long time to complete on large tables. For instance, deploying one index over a table that stores billions of tuples (which is not uncommon at this time) could take days.

Moreover, it is likely that a database requires hundreds of indexes to be deployed due to the growing number and complexity of queries and table schema. For example, a commercial database designer suggests 148 indexes for the TPC-DS benchmark which take more than 24 hours to be deployed even on the smallest (Scale-100) instance. Moreover, business software packages such as SAP use tens of thousands of indexes in their databases, which have to be deployed over thousands of tables and also occasionally re-built on upgrades, migration and so on.

The motivation of this chapter comes from an observation that, during the long process of deploying many indexes over large databases, the order (sequence) of index deployment has two significant impacts on user benefit, illustrated in Figure 4.1. First, a good order achieves prompt query runtime improvements by deploying indexes that yield greater query speed-ups in early steps. For example, an index that is useful for many queries should be created first. Second, a good order reduces the deployment time by allowing indexes to utilize previously built indexes to speed up their

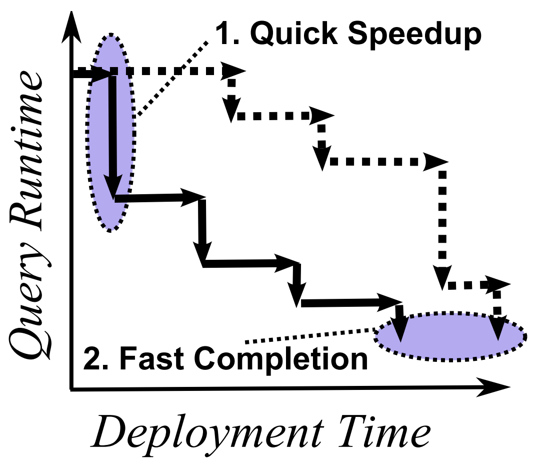

Figure 4.1: Good vs Bad Order

deployment. For instance, an index (ItemID, Date) should be made *after* a wider index (ItemID, Priority, Date) to allow building from the index, not the table. We observe in the TPC-DS case that a good deployment order can reduce the build cost of an index up to 80% and the entire deployment time as much as 20%.

Despite the potential benefits, obtaining the optimal index order is challenging. Unlike typical job sequencing problems [BPN01], both the benefit and the build cost of an index are dependent on the previously built indexes because of **index interactions** described in Section 4.3.2. These database specific properties make the problem non-linear and much harder to solve. Also, as there are $n!$ orderings of n indexes, a trivial exhaustive search is intractable, even for small problems.

One prevalent approach for optimization problems is to quickly choose a solution by a greedy heuristic. However, the quality of a greedy approach can vary from problem to problem and has no quality guarantee. Another popular approach is to employ exact search algorithms such as A* or mixed integer programming (MIP) using the branch-bound (BB) method to prune the search space. However, the non-linear properties of the index interactions yield poor linear relaxations for the BB method and both MIP and A* degenerate to an exhaustive search without pruning.

In this chapter, we formally define the ordering problem as a mathematical model and propose several pruning techniques not based on linear relaxation but on the combinatorial properties of the problem. We show that these problem specific combinatorial properties can reduce the size of the search space by tens of orders of magnitude. We solve the problem using several techniques including, Constraint Programming (CP) and MIP, and show that this kind of problem is easiest to model and has better performance in a CP framework. We then extend the CP model using local search methods to get a near-optimal solution very quickly for larger problems. We evaluate several local search methods and devise a variable neighborhood search (VNS) method building on our CP model that is highly scalable and stable. In summary, our contributions are:

- A formal description of the index deployment order problem
- Problem specific properties to reduce problem difficulty
- Models and algorithms for Greedy, MIP, CP and local search
- Analysis of various solution techniques and solvers
- Empirical analysis on TPC-H and TPC-DS.

To the best of our knowledge, this work is the first to study CP methods in the context of physical database design despite its significant potential as an accurate and scalable design method.

The remainder of this chapter is organized as follows. Section 4.2 reviews the related work. Section 4.3 formally defines the problem of index deployment. Section 4.4 provides several techniques to efficiently solve the problem. Section 4.5 describes our CP model for the problem. Section 4.6 extends the CP model with local search to solve larger problems. Then, Section 4.7 reports the experimental results and Section 4.8 concludes this chapter.

4.2 Related Work

4.2.1 Physical Database Design

Because of the complexity of query workloads and database mechanics, no human database administrator (DBA) can efficiently select a set of database objects (e.g., indexes) subject to resource constraints (e.g., storage size) to improve query performance. Hence, significant research effort has been made both in academia and in industry to automate the task of physical database design [CN97, ZRL$^+$04].

The AutoAdmin project [ACN00] pioneered this field by employing the *what-if* method [CN98] which creates a set of potentially beneficial indexes as hypothetical indexes to evaluate their expected benefit by the database's query optimizer.

Once the benefits of each index are evaluated, the problem of database design is essentially a boolean knapsack problem, which is NP-hard. The database community has tried various approaches to solve this problem. The most common approach is to use greedy heuristics based on the benefit of indexes [CN97] or on their density [ZRL$^+$04] (benefit divided by size). However, a greedy algorithm is not assured to be optimal and could be arbitrarily bad in some cases. Hence, some research has explored the use of exact methods such as mixed integer programming (MIP) [PA07, KHR$^+$10] and A* search [GM99].

Despite the wealth of research in physical database design, no one has studied the problem of optimizing index deployment orders. Almost all prior work in this field considers both the query workloads and the indexes as a *set*. The only exception is [ACN06] which considers a query workload as a sequence, but only considers dropping and re-creating existing indexes to reduce maintenance overhead. Bruno et al.[BC07] mentioned a type of ordering problem as an unsolved problem, but their objective does not consider prompt query speed-ups. Also, they only suggested to use A* or Dynamic Programming and did not solve the problem in [BC07].

4.2.2 Branch-and-Bound

All decision problems, such as the index order problem, can be formulated as tree search problems. Such a tree has one level for each decision that must be made and every path from the root node to a leaf node represents one solution to the problem. In this way, the tree compactly represents all the possible problem solutions. However, exploring this entire tree is no more tractable than exhaustive search. Therefore, many tree search techniques have been developed to more efficiently explore the decision tree.

Branch-and-Bound (BB) is a tree search method which prunes (a.k.a. removes) sub-trees by comparing a lower bound (best possible solution quality) with the current best solution. A* is a popular type of BB search method which uses a user-defined heuristic distance function to deduce lower bounds.

MIP solvers, such as IBM ILOG CPlex, are also based on BB. MIP uses a linear relaxation of the problem to deduce lower bounds, and the pruning power of the MIP is highly dependent on the tightness of the linear relaxation.

BB is efficient when the relaxation is strong, however it degrades as the relaxation becomes weaker, which is often the case for non-linear problem (such as, the traveling salesman problem).

Also, MIP only supports linear constraints, and it is tedious to model non-linear properties using only linear constraints.

4.2.3 Constraint Programming

Similar to MIP, Constraint Programming (CP) does a tree search over the values of the decision variables. Given a model, a CP solver explores the search tree like a MIP solver would. However, there are a few key differences summarized in Table 4.1.

First, CP uses a branch and prune (BP) approach instead of BB. At each node of the tree, the CP engine uses the combinatorial properties of the model's constraints to deduce which branches cannot yield a higher quality solution. Because the constraints apply over the combinatorial properties of the problem, the CP engine is well suited for problems with integer decision variables. Instead of a linear relaxation to guide the search procedure in MIP, CP models often include specialized search strategies that are designed on a problem-by-problem basis [BPN01].

Second, CP does not suffer from the restriction of linearity that MIP models have. This is especially helpful for our problem which has a non-linear objective function and constraints such as nested decision variable indexing.

Third, CP models allow a seamless extension to local search. When the problem size becomes so large that proving a solution's optimality is impossible, the goal becomes getting a near-optimal solution as fast as possible. In this setting, global search techniques (such as MIP and CP) often become impractical because they exhaustively search over every sub-tree that has some chance of containing the optimal solution regardless of how slight the chance is, and how large the sub-tree is. Such exact methods are thus inappropriate to quickly find high quality solutions. On the other hand, local search on top of CP such as Large Neighborhood Search (LNS) [VHM09] combines the pruning power of CP with the scalability of local search.

In later sections, we will contrast these differences more vividly with concrete case studies for modeling and solving the index order problem. Although we find that CP is highly effective for physical database design, to the best of our knowledge this is the first time that CP has been applied to this problem domain.

Table 4.1: MIP and CP Comparison

	MIP	CP
Constraints & Objectives	Linear Only	Linear & Non-Linear
Pruning Method	Branch-Bound & Linear Relaxation	Branch-Prune & Custom Constraints
Non-Exhaustive Search Variant	N/A (Best Solution)	Local Search
Best Suited for	Linear Problems	Combinatorial Problems

4.3 Problem Definition

This section formally defines the index deployment order problem. Throughout this section, we use the symbols, constant values, and decision variables listed in Table 4.2 and 4.3.

4.3.1 Objective Values

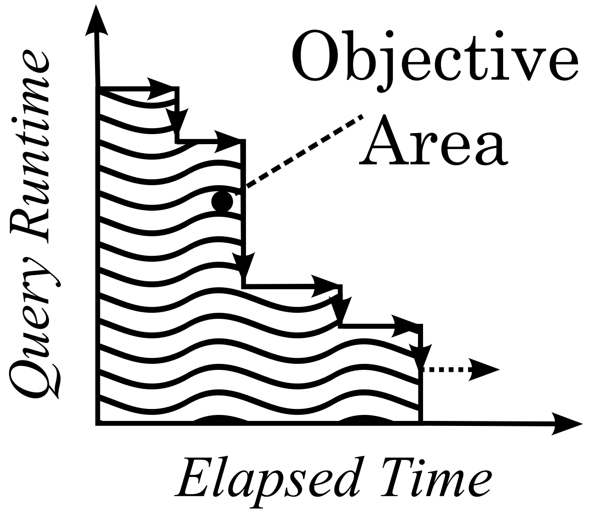

Figure 4.2: Objective Values

Every feasible solution to the problem is a *permutation* of the indexes. An example permutation of indexes $\{i_1, i_2, i_3\}$ is $i_3 \rightarrow i_1 \rightarrow i_2$.

As discussed in the introduction, we want to achieve a prompt query runtime improvement and a reduction in total deployment time. Hence, the metric we define to compare solutions is the area under the improvement curve illustrated in Figure 4.2.

This area is defined by $\sum_i (R_{i-1}C_i)$, the summed products of the **previous** total query runtime and the cost to create the i^{th} index. The previous total query runtime is used because the query speed-up occurs only after we complete the deployment of an index.

Because we would like to reduce the query runtimes and total deployment time, the smaller the area is, the better the solution is. Thus, this objective function considers prompt query speed-ups and total deployment time simultaneously.

Table 4.2: Symbols & Constant Values (in lower letters)

$i \in I$	An index. $I = \{i_1, i_2, .., i_{\|I\|}\}$
$q \in Q$	A query.
$p \in P$	A query plan (a set of indexes).
$plans(q) \in P$	Feasible query plans for query q.
$qtime(q)$	Original runtime of query q.
$qspdup(p, q)$	Speed-up of using plan p for query q compared to the original runtime of q.
$ctime(i)$	Original creation cost of index i.
$cspdup(i, j)$	Speed-up of using index j for index i.

4.3.2 Index Interactions

This section describes the various **index interactions**, which make the problem unique and challenging.

Competing Interactions: Unlike typical job sequencing problems, completing a job (i.e. building an index) in this problem has varying benefits depending on the completion time of the job.

This is because a DBMS can only use one query execution plan at a time. Consider the indexes $i_1(City)$ and $i_2(City, Salary)$ from the following query:

```
SELECT AVG(Salary) FROM People WHERE City=Prov
```

Assume the query plan using i_1 is 5 seconds faster than a full scan while the plan using the covering index i_2 is 20 seconds faster.

The sequence $i_1 \rightarrow i_2$ would have a 5 second speed-up when i_1 is built, and only $20-5 = 15$ second speed-up when i_2 is built because the query optimizer in the DBMS picks the fastest query plan possible at a given time, removing the benefits of suboptimal query plans. Likewise, the sequence $i_2 \rightarrow i_1$ would observe no speed-up when i_1 is built. We call this property *competing interactions* and generalize them by constraint 4.3 in the mathematical model.

Query Interactions: It is well known that two or more indexes together can speed up query execution much more than each index alone. Suppose we have two indexes $i_1(City)$ and $i_2(EmpID)$ for the following query:

```
SELECT .. FROM People p1 JOIN People p2
ON (p1.ReportTo=p2.EmpID) WHERE p1.City=Prov
```

A query plan using one index ($\{i_1\}$ and $\{i_2\}$) requires a table scan for the JOIN and costs as much as the no-index plan $\{\emptyset\}$. A query plan using both i_1 and i_2 ($\{i_1, i_2\}$) avoids the full table scan and performs significantly faster. We call such index interactions *query interactions*. Because of such interactions, we need to consider the speed-ups of the three query plans separately, rather than simply summing up the benefits of singleton query plans.

Build Interactions: As a less well known interaction, some indexes can be built faster if there exists another index that has some overlap with the keys or included columns of the index to be built.

For example, $i_1(City)$ and $i_2(City, Salary)$ have interactions in both ways. If i_2 already exists, building i_1 becomes substantially faster because it requires only an index scan on i_1 rather than scanning the entire table. On the other hand, if there already is i_1, building i_2 is also faster because the DBMS does not have to sort the entire table. We call these index interactions *build interactions* and generalize it by constraint 4.5 in the mathematical model.

This means that the index build cost is not a constant in our problem but a variable whose value depends on the set of indexes already built. Bruno et al. [BC07] also mentioned this effect earlier. In Section 4.7 we show there exist a rich set of such interactions.

Precedence: Some indexes *must* precede some other indexes.

One example is an index on a *materialized view* (MV). A MV is created when its clustered index is built. Non-clustered (secondary) indexes on the MV cannot be built before the clustered index. Hence, the clustered index must precede the secondary indexes on the same MV in a feasible solution.

Another example is a secondary index that exploits *correlation* [KHR$^+$09]. For example, SQL Server supports the *datetime correlation optimization* which exploits correlations between clustered and secondary datetime attributes. To work properly, such an index requires the corresponding clustered index to be built first.

Detection: Some prior work explored a way to efficiently find such interacting indexes [S$^+$09].

In our experiments, we detect interactions by calling the query optimizer with hypothetical indexes as detailed in Section 4.7.

Table 4.3: Decision Variables (in capital letters)

$T_i \in \{1, 2, .., \lvert I \rvert\}$	The position of index i in the deployment order. T is a permutation of $\{1, .., \lvert I \rvert\}$.
R_i	Total query runtime after i^{th} index is made.
$X_{q,i}$	q's speed-up after i^{th} index is made.
$Y_{p,i} \in \{0, 1\}$	Whether p is available after i^{th} index is made.
C_i	Cost to create i^{th} index.

4.3.3 Mathematical Model

Embodying the concepts of index interactions discussed above, the full mathematical model is defined as follows,

$$\text{Objective:} \qquad min \sum_i (R_{i-1} C_i) \tag{4.1}$$

$$\text{Subject to:} \qquad Y_{p,i} = \{T_j \leq i : \forall j \in p\} : \forall p, i \tag{4.2}$$

$$X_{q,i} = \max_{p \in plans(q)} qspdup(p, q) Y_{p,i} : \forall q, i \tag{4.3}$$

$$R_i = \sum_q (qtime_q - X_{q,i}) : \forall i \tag{4.4}$$

$$C_{T_i} = ctime(i) - \max_{j : T_j < T_i} cspdup(i, j) : \forall i \tag{4.5}$$

(4.2) states that a query plan is available only when all of the indexes in the query plan are available. (4.3) calculates the query speed-up by using the fastest query plan for the query at a given time. (4.4) sums up the speed-ups of each query and subtract from the original query runtime to get the current total runtime.

(4.5) calculates the cost to create index i (C_{T_i} because C is indexed by the order) by considering the fastest available ($T_j < T_i$) interaction. For simplicity, this constraint assumes every build interaction is pair-wise (one index helps one other index). So far we have observed this to be the case, but this constraint can easily be extended for arbitrary interactions by doing a similar formulation using X and Y variables.

Given this mathematical formulation, our goal is to find the permutation with the minimal objective value and prove its optimality. However, for large problems where an optimality proof is intractable we are satisfied with a near-optimal solution that can be found quickly.

4.3.4 Discussion

There could be variants of the objective. For example, putting different weights on particular queries can be incorporated by simply scaling up or down runtimes of the queries. Or, one can simply consider $\sum C_i$ as objective to minimize the deployment time like [BC07]. In either case, most of the modeling and pruning strategies in this chapter will be usable with minor modifications.

4.4 Problem Properties

This problem has up to $|I|!$ possible solutions. An exhaustive search method that tests all the solutions becomes intractable even for small problems. Hence, in this section we analyze the combinatorial properties of the problem. Based on the problem specific structure, such as index interactions, we established a rich set of pruning techniques which significantly reduce the search space. This section describes the intuition behind each optimization technique and how we apply it to the problem formulation. The formal proofs, cost analysis, and drill-down analysis of the pruning power of each technique can be found in Appendix B.4.

These techniques are inherent properties of the problem which are independent of a particular solution procedure. In fact, we demonstrate that these techniques reduce the runtime of both MIP and CP solvers by several orders of magnitude in Section 4.7.

4.4.1 Alliances

The first problem property is an *alliance* of indexes that are always used together. We can assume that such a set of indexes are always created together.

Figure 4.3 exemplifies alliances of indexes. The figure illustrates 4 query plans with 6 indexes; $\{i_1, i_3\}$, $\{i_1, i_3, i_5\}$, $\{i_2, i_5\}$, $\{i_4, i_6\}$. Observe that i_1 and i_3 always appear together in all query plans they participate in. Therefore, creating only one of them gives no speed-up for any query. This means we should **always** create the two indexes together. Hence, we add a constraint $T_{i_1} = T_{i_3} + 1$. Same to i_4 and i_6. Note that i_2 and i_5 are **not** an alliance because i_5 appears in the query plan $\{i_1, i_3, i_5\}$

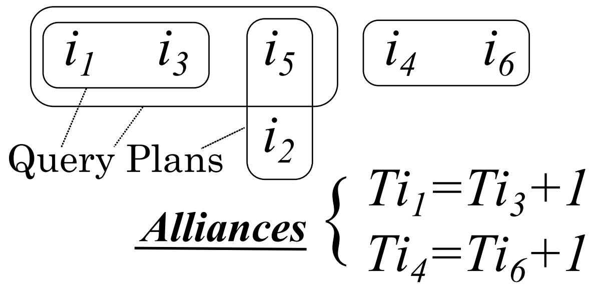

Figure 4.3: Alliances

without i_2. An alliance is often a set of strongly interacting indexes each of which is not beneficial by itself. An alliance of size n essentially removes $n-1$ indexes and substantially simplifies the problem.

4.4.2 Colonized Indexes

The next problem property is a *colonized* index which is a one-directional version of alliances. If all interactions of an index, i, contain another index, j but not vice versa, then i is called a colonized index and should be created after j.

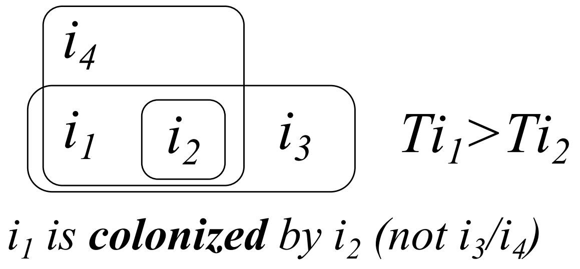

Figure 4.4: Colonized Indexes

Figure 4.4 shows a case where i_1 is colonized by i_2. i_1 always appears with i_2 in all query plans i_1 participates, but not vice versa because there is a query plan that only contains i_2.

In such a case, creating i_1 alone always yields no speed-up. On the other hand, creating i_2 alone might provide a speed-up. Thus, it is always better to build the colonizer first; $T_{i_1} > T_{i_2}$.

Observe that i_1 is not colonized by i_3 or i_4 because i_1 appears in plans where only one of them appears. In fact, if the plan $\{i_1, i_2, i_4\}$ is highly beneficial, the optimal solution is $i_2 \rightarrow i_4 \rightarrow i_1 \rightarrow i_3$, so $T_{i_1} > T_{i_3}$ does not hold. Likewise, if the plan $\{i_1, i_2, i_3\}$ is highly beneficial, the optimal solution is $i_2 \rightarrow i_3 \rightarrow i_1 \rightarrow i_4$, so $T_{i_1} > T_{i_4}$ does not hold.

4.4.3 Dominated Indexes

The next problem property is called a *dominated* index which is an index whose benefits are **always** lower than benefits of another index. Dominated indexes should always be created last.

To simplify, consider the case where indexes have the same build cost and every query plan is used for different queries. For the full formulation without these simplifications, see Appendix B.4.

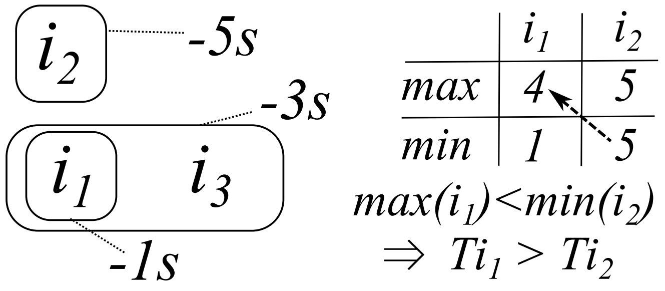

Figure 4.5: Dominated Indexes

Figure 4.5 depicts an example where i_1 is dominated by i_2. The maximum benefit of an index is the largest speed-up we get by building the index. For example, the maximum benefit of i_1 occurs when there already exists i_3, which is $1 + 3 = 4$ seconds. Conversely, the minimum benefit is the smallest speed-up we get by building the index. i_1's minimum benefit happens when there is no i_3 index; only 1 second. On the other hand, both the maximum and minimum benefits of i_2 are 5 seconds.

Hence, the speed-up of building i_1 is always lower than the speed-up of building i_2. As our objective favors a larger speed-up at an earlier step, we should always build i_2 before i_1; $T_{i_1} > T_{i_2}$.

4.4.4 Disjoint Indexes and Clusters

The next problem property is called a *disjoint* index and is an index that has no interaction with other indexes. Such indexes do not give or receive any interaction to affect the build time and speed-up and sometimes we can deduce powerful constraints from them. Figure 4.6 shows an example of a disjoint index i_4 and a *disjoint cluster* $M_1 = \{i_1, i_2, i_3\}$ which has no interaction with other indexes except the members of the cluster.

Suppose we already have a few additional constraints that define the relative order of $\{i_1, i_2, i_3\}$

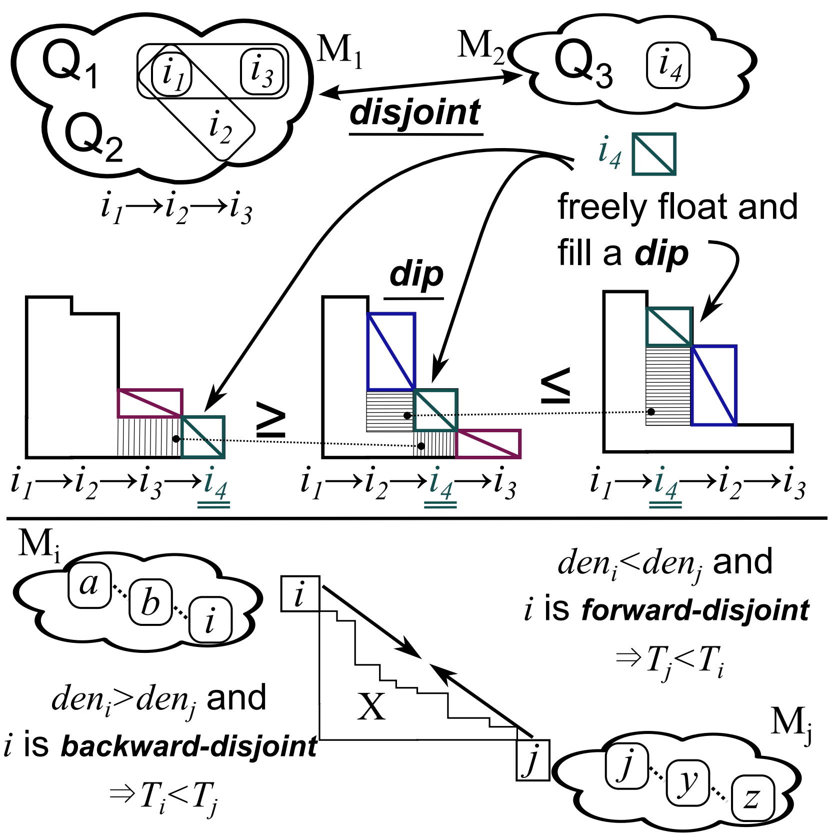

Figure 4.6: Disjoint Indexes and Disjoint Clusters

is $i_1 \to i_2 \to i_3$ and we need to insert i_4 into the order. Among the four possible locations for i_4, we can uniquely determine the best place, which we call the *dip*.

We know the placement of i_4 does not affect the build cost and the speed-up of any index in M_1 because i_4 and M_1 are disjoint. In such a case, we should place i_4 after an index whose *density* (the gradient of the diagonal line; speed-up divided by build cost) is larger than i_4's density and before an index with a smaller density. Otherwise, we can improve the order by swapping i_4 with another index because the shaded area in Figure 4.6 becomes larger when we build an index with a smaller density first. In the example, the best place is between i_2 and i_3, which means $den_{i_1+i_2} > den_{i_4}$, $den_{i_2} > den_{i_4}$ and $den_{i_4} > den_{i_3}$ where den_x is the density of x. We call this location, the dip and there is always exactly one dip.

We can generalize the above technique for non-disjoint indexes when they have special properties which we call *backward-disjoint* and *forward-disjoint*. Consider two disjoint clusters M_i and M_j which contain index i and j respectively. In order to determine whether i precedes or succeeds j in the complete order, we can investigate the interacting indexes of i and j.

i is said to be backward-disjoint regarding j when all interacting indexes of i and j are built after i **or** before j. Conversely, i is said to be forward-disjoint regarding j when all interacting indexes are built before i **or** after j, in other words when j is backward-disjoint regarding i. A disjoint index is both backward and forward disjoint regarding every other disjoint index. Initially most indexes have no disjoint properties, but with the additional constraints from other properties they often become backward or forward disjoint.

An intuitive description of i being backward-disjoint regarding j is that i and j behave as disjoint indexes when we are considering a subsequence $j \rightarrow X \rightarrow i$ for arbitrary X, so i is *disjoint in a backwards order*. Because of the property of disjoint indexes, the subsequence must satisfy $den_i < den_j$ if it is an optimal solution. Thus, if we know $den_i > den_j$, we can prune out all solutions that build j before i. Conversely, if i is forward-disjoint and $den_i < den_j$, then i always succeeds j.

4.4.5 Tail Indexes

Because of the inequality constraints given by the above properties, sometimes a single index is uniquely determined to be the last index. In that case, we can eliminate the index from the problem for two reasons. First, the last index cannot cause any interaction to speed up other indexes either in query time or build time because all of them precede the last index. Second, the interactions the last index receives from other preceding indexes do not depend on the order of other indexes; all the other indexes are already built. Therefore, we can remove the last index and all of its interactions from consideration, substantially simplifying the problem.

We can extend this idea even if there are multiple candidates for the last index by analyzing the possible *tail* index patterns.

For example, in the TPC-H problem solved in Section 4.7.3, i_1 and i_2 turn out to have many preceding indexes and thus the possible orders of them are n (last), $n-1$ (second to last) and $n-2$ (third to last). All possible patterns of the last 3 tail indexes are listed in Figure 4.7. It also shows the last part of the objective area (*tail objective*) for the 3 tail indexes in each pattern (the shaded areas). We can calculate the tail objectives because the *set* of preceding indexes is known therefore, regardless of their orders, their interactions to the tail indexes are determined.

Remember that there are many other preceding indexes before the tail indexes. Therefore, we

Table 4.4: Tail Objectives in TPC-H

Tail	Obj.
$\boldsymbol{i_4 \to i_1 \to i_2}$	**9.7**
$i_4 \to i_2 \to i_1$	9.9
$i_1 \to i_4 \to i_2$	12
$\boldsymbol{i_5 \to i_1 \to i_2}$	**4.0**
$i_5 \to i_2 \to i_1$	4.2
$i_2 \to i_5 \to i_1$	4.5
$\boldsymbol{i_8 \to i_1 \to i_2}$	**6.8**
$i_8 \to i_2 \to i_1$	6.9
$\boldsymbol{i_{11} \to i_1 \to i_2}$	**7.1**
$i_{11} \to i_2 \to i_1$	7.3

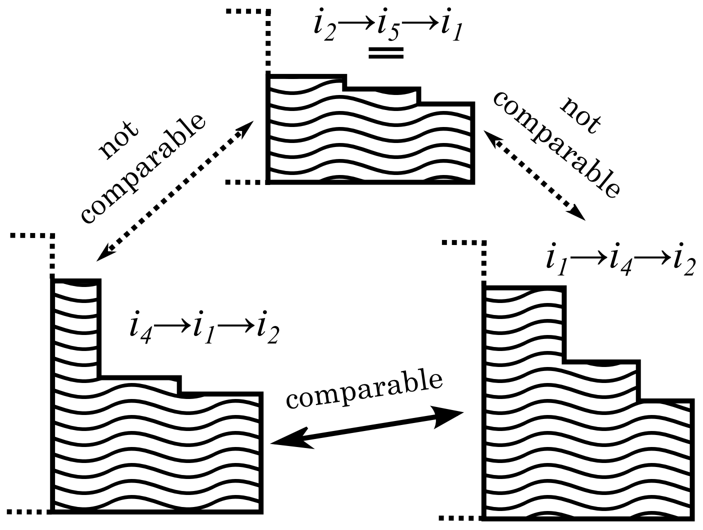

Figure 4.7: Comparing Tail Indexes of Same Index Set in TPC-H

cannot simply compare the tail objectives. For example, the tail objective of $i_2 \to i_5 \to i_1$ in Figure 4.7 is smaller than that of $i_4 \to i_1 \to i_2$. However, because the set of preceding indexes is different, we cannot tell if the former tail pattern is better than the latter.

Nevertheless, we can compare the tail objectives if the set of tail indexes is equivalent. $i_4 \to i_1 \to i_2$ and $i_1 \to i_4 \to i_2$ contain the same set of indexes, thus *the set of preceding indexes is the same too*, which means the objective areas and the order of preceding indexes is exactly the same after we optimize the order of preceding indexes (again, the tail indexes do not affect preceding indexes). Hence, we can determine which tail pattern is better by comparing tail objectives.

Notice that the tail patterns in Figure 4.7 are grouped by the set of tail indexes and also sorted by the tail objectives in each group. The ones with the smallest tail objective in each group are called the *champion* of the group and they should be picked if the set of indexes are the tails.

Now, observe that i_2 appears as the last index in every champion (in bold font) of all groups. This means i_2 is always the last created index in the optimal deployment order because its tail is always one of the tail champions.

4.4.6 Iterate and Recurse

We can repeat the tail analysis by fixing i_2 as the last index and considering a sub-problem without i_2. Not surprisingly, we could then uniquely identify i_1 as the second-to-last index.

Furthermore, by removing the determined indexes (and their query plans) and considering the already introduced inequalities, each analysis described in this section can apply more constraints. Therefore, we repeat this process until we reach the fixed-point. This pre-analysis reduces the size of search space dramatically. In the experimental section, we demonstrate that the additional constraints speed up both CP and MIP by several orders of magnitude.

4.5 Constraint Programming

In this section, we describe how we translate the mathematical model given in Section 4.3.3 into a Constraint Programming (CP) model. We then explain how the problem is solved with a CP solver. To illustrate why CP is well suited for this problem, we will compare the CP model to that of MIP throughout this section.

4.5.1 CP Model

CP allows a flexible model containing both linear and non-linear objectives and constraints. The mathematical formulation presented in Section 4.3.3 can be modeled in standard CP solvers (e.g., COMET) almost identically unlike MIP where the model is more obfuscated (an equivalent MIP model is given in Appendix B.2).

$$\text{Objective:} \quad min \sum_i (R[i-1]C[i]) \tag{4.6}$$

$$\text{Subject to:} \quad alldiffrent(T) \tag{4.7}$$

$$Y[p,i] = \bigwedge_{j \in p} (T[j] \leq i) : \forall p, i \tag{4.8}$$

$$X[q,i] = \max_{p \in plans(q)} (qspdup(p,q)Y[p,i]) : \forall q, i \tag{4.9}$$

$$R[i] = \sum_q (qtime(q) - X[q,i]) : \forall i \tag{4.10}$$

$$C[T[i]] = ctime(i) - \max_j ((T[j] < T[i])cspdup(i,j)) : \forall i \tag{4.11}$$

Objective: Just like the mathematical model, our CP model minimizes the sum of $R[i-1]C[i]$. Although this sounds trivial, MIP cannot accept a product of variables (R and C) as objectives.

The most common technique for linearizing a product of variables in MIP is to *discretize* the entire span to a fixed number of uniform timesteps and define the value of each variable at each timestep as an independent variable [SW92].

However, in addition to losing the accuracy, discretization causes severe problems in performance and scalability of MIP which are verified in the experimental section.

***alldifferent* constraint:** The variable T is given in (4.7) which uses *alldifferent*. This interesting constraint in CP assures all the variables in T are a permutation of their values. The same constraint in MIP would require $|I|^2$ inequalities on elements of T. The CP engine represents it with a *single* constraint which is computationally efficient. This is one of the most vivid examples showing that CP is especially suited for combinatorial problems and how beneficial it is for modeling and optimization purposes.

Logical AND: The AND constraints on Y (4.2) are translated directly into (4.8). Although this sounds trivial, again, it is challenging in MIP. Logical AND is essentially a product of boolean variables, which is non-linear, just as the objective was. Modeling such non-linear constraints causes MIP additional overhead and memory consumption as well as model obfuscation.

MIN/MAX as sub-problem: The constraints on X (4.3) which employ the fastest available speed-up for each query are translated directly into (4.9). Yet again, this is not easy nor efficient in MIP because MIN/MAX is non-linear.

In MIP, this has to be represented as summation of Y and *qspdup* where only one of Y for each query takes the value of 1 at a given time. Some MIP solvers provide min/max constraint and internally do this translation on behalf of users, but the more severe problem is its effect on performance. When MIP considers the linear relaxation of X, min/max constraint yields little insight. Hence, its BB degenerates to an exhaustive search.

Nested variable indexing: The constraints on C (4.5) are translated directly into (4.11). However, this causes two problems in MIP. One is the MIN/MAX as described above, another is the nested variable indexing C_{T_i}. Notice that T is also a variable. Such a constraint cannot be represented in a linear equation. Hence, MIP has to change the semantics of the variable C itself and re-formulate the all of the constraints and the objective calculation.

Additional constraints: Finally, we add the additional constraints developed in Section 4.4 to reduce the search space.

4.5.2 Searching Strategy

CP employs branch-prune (BP) instead of BB used by MIP. These two approaches have very different characteristics. In summary, CP is a *white-box* approach with a smaller footprint as opposed to the *black-box* approach of MIP.

Pruning: CP is able to prune the search space by reasoning over the combinatorial properties of the constraints presented in section 4.5.1. It also utilizes the problem specific constraints we developed in Section 4.4 to efficiently explore only high quality orders. Our experimental results demonstrate that combinatorial based pruning is much more effective for this problem than a BB pruning based on a linear relaxation.

Branching: Users *can* and *must* specify how CP should explore the search space. In our case, we found that it is most effective for the search to branch on the $T[i]$ variables and that a *First-Fail* (FF) search procedure was very effective for solving this problem and proving optimality with very small memory footprint.

A FF search is a depth-first search using a dynamic variable ordering, which means the variable ordering changes in each node of the search tree. At each node the variables are assigned by increasing the domain size. Due to the additional constraints, the domains of the $T[i]$ variables vary significantly. This helps the FF heuristic to obtain optimality.

On the other hand, MIP automatically chooses the branching strategy. This is efficient when the linear relaxation is strong, but, when it is not, the BB search degenerates to an exhaustive breadth-first search which causes large memory consumption and computational overhead. In fact, we observe that MIP finds no feasible solution for large problems within several hours and quickly runs out of memory.

4.6 Local Search

Although CP is well suited for this ordering problem, when there is a large number of indexes with dense interactions between them, proving optimality is intractable. In such a case, our goal is to find a near-optimal solution quickly.

Local search is a family of algorithms for quickly finding high quality solutions. There are many possible local search meta-heuristics to choose from such as, Tabu Search (TS) [GL97], Simulated Annealing, Ant Colony optimization, Large Neighborhood Search (LNS) [VHM09], and Variable Neighborhood Search (VNS). We consider two TS methods, LNS and VNS. TS is a natural choice because it is effective on problems with a highly connected neighborhood (such as this one, where nearly all index permutations are feasible). We also consider LNS and VNS because they are a simple extension of a CP formulation and the CP formulation proved to be very effective on smaller instance sizes.

4.6.1 Tabu Search (TS)

Tabu Search (TS) is a simple method for performing gradient descent on the index permutation. At each step, TS considers swapping a pair of elements in T. To avoid being trapped in local optima and repeating the same swap, TS maintains a *Tabu list*. The elements recently swapped are considered in probation for some number of steps (called *Tabu length*). During those steps, TS does not consider swapping those elements and hopefully escapes local optima.

We implemented and evaluated two Tabu Search methods; TS-**B**Swap (*Best-Swap*) and TS-**F**Swap (*First-Swap*). TS-BSwap considers swapping all possible pairs of indexes at each iteration except the Tabu list, and takes the pair with the greatest improvement. TS-FSwap stops considering swaps when it finds the first pair that brings some improvement.

TS-BSwap will result in better quality while TS-FSwap will be more scalable because quadratic time of checking all pairs may take considerable time in large problems.

4.6.2 Large Neighborhood Search (LNS)

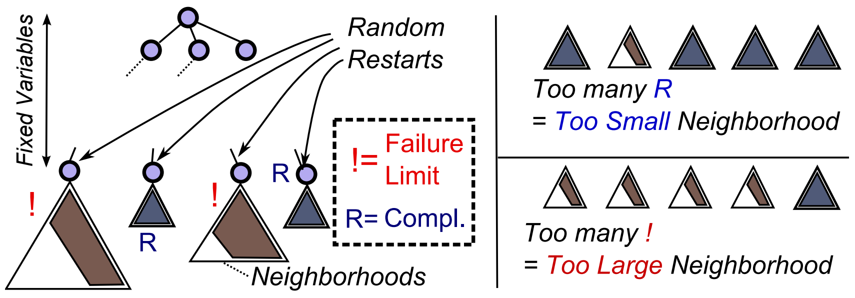

Figure 4.8: Tuning Large Neighborhood Search

Figure 4.8 illustrates how a LNS algorithm executes. A LNS algorithm works by taking a feasible solution to an optimization problem and relaxing some of the decision variables. A CP search is then executed on the relaxed variables while the other variables remain fixed. If the CP search is able to assign the relaxed variables and improve the objective value, then it becomes the new current solution, otherwise the solution is reset and a new set of variables are randomly selected for relaxation (*restart*). Like most local search algorithms, this procedure is repeated until a time limit is reached. In this way, LNS leverages the power of a CP solver to efficiently search a large neighborhood of moves from the current best solution.

The CP model for our LNS algorithm was presented in Section 4.5.1, to complete the picture we need to explain our relaxation strategy. For simplicity we use a very basic relaxation, 5% of the indexes are selected uniformly at random for relaxation. A new relaxation is made if one of these two conditions is met; (1) the CP solver proves no better solution exists in this relaxation; (2) the CP solver has to back track over 500 times during the search (in LNS this is called the failure limit). We found this relaxation size and failure limit effectively drove the search to a high quality solution.

4.6.3 Variable Neighborhood Search (VNS)

One difficulty of a LNS algorithm is how to set the parameters for relaxation size and failure limit. As depicted in Figure 4.8, if they are set too small it is easy to get stuck in a local minimum. If they are too large the performance may degrade to a normal CP approach. Furthermore, different problem sizes may prefer different parameter settings. Our remedy for this difficulty is to change the parameters during search. This technique is well known as Variable Neighborhood Search (VNS) [GK03].

Our VNS approach is to start the search on a small neighborhood and inspect the behavior of the CP solver to increase the neighborhood and escape local minima only when it is necessary. The intuition is, if the relaxation terminates because the CP solver proves there is no better solution, then we are stuck in a local minimum and the relaxation size must increase. However, if the CP solver hits the failure limit without proof, then we should do more exploration in the same size neighborhood, which is achieved by increasing the failure limit. Specifically, we group the relaxations into groups of 20 and if more than 75% of these relaxations were proofs then we increase the relaxation size by 1%, otherwise we increase the failure limit by 20%.

In the experimental section, we find this VNS strategy has two benefits. First it guides the algorithm to high-quality solutions faster than a regular LNS and also consistently found higher quality solutions. Second, VNS is highly scalable and stable even for a problem with hundreds of indexes, which is not the case with the other methods.

4.6.4 Greedy Initial Solution

As described in the introduction, greedy algorithms are scalable but have no quality guarantees. Nonetheless, a greedy algorithm can provide a great initial solution to start a local search algorithm.

To that end, we devise a greedy algorithm which gives a much better initial solution than starting from a random permutation. The key idea of the algorithm is to consider interactions of each index as future opportunities to enable a beneficial query plan that requires two or more indexes. We greedily choose the index with the highest density (benefit divided by the cost to create the index) at each step. Here, the benefit is the query speed-up achieved by adding the index *plus* the potential benefits from interactions. We find query plans that contain the index but are not yet usable because of missing indexes, then equally attribute the speed-up of the query plan to the missing indexes, dividing the benefit by the count of them. For more details and analysis of its quality, see Appendix B.3.

4.7 Experiments

In this section, we study the performance and scalability of each method described in earlier sections via empirical analysis.

4.7.1 Implementation

Our implementation of the CP-based index ordering solution is summarized in Figure 4.9. Given a query workload, we first run a physical database design tool to obtain a set of suggested indexes. Then, we analyze the indexes. To avoid actually creating indexes, we use *what-if* [CN98] interface of the DBMS to hypothetically create each index and evaluate its benefits using the query optimizer. The result is a matrix which stores the benefits and creation costs of all indexes as well as the interactions between them.

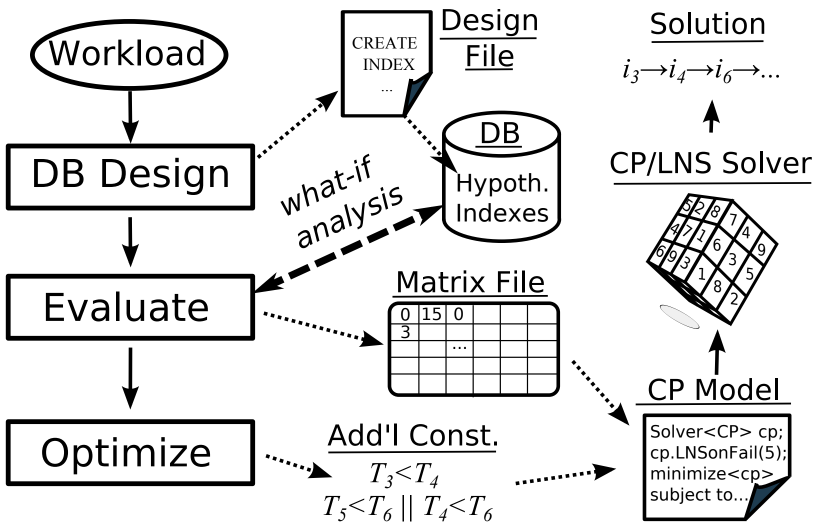

Figure 4.9: System Overview

When formulating this matrix as CP code, we also apply the optimization techniques described in Section 4.4 to get additional constraints to speed up the CP solver. The CP/LNS engine then solves the problem and produces the optimized index deployment order.

We used a popular commercial DBMS and its design tool for the experiments. We also used Comet 2.1 as a CP/LNS solver and ILOG CPlex 12.2 as a MIP solver. All experiments are done in a single machine with a Dual-Core CPU and 2 GB of RAM. CPlex automatically parallelized the MIP on the dual core while CP and local search in Comet only used one core.

4.7.2 Datasets

We use two standard benchmarks as datasets; TPC-H and TPC-DS. Table 4.5 shows the size of each dataset. TPC-DS is a major revision of TPC-H to reflect the complex query workloads and table scheme in real data analysis applications. TPC-DS has many more queries, each of which is substantially more complex and requires several indexes to efficiently process when compared to TPC-H. Hence, the design tool suggested 148 indexes (up to 300 depending on configurations of the tool). There is even a query plan that uses as many as 13 indexes together. We also found a rich set of index interactions in both datasets.

Table 4.5: Experimental Datasets

Dataset	$\lvert Q\rvert$	$\lvert I\rvert$	$\lvert P\rvert$	Largest Plan	#Inter. (Build)	#Inter. (Query)
TPC-H	22	31	221	5 Index	31	80
TPC-DS	102	148	3386	13 Index	243	1363

4.7.3 Exact Search Results

We verified the performance of each method to find and prove the optimal solution with the TPC-H dataset.

We compared the performance of MIP and CP methods with and without the additional constraints, varying the number of indexes (size of the problem). For MIP, we discretized the problem for $|I| * 20$ timesteps. We also varied the density of the problem. *low* density means we remove all suboptimal query plans and build interactions. *mid* density means we remove all but one suboptimal query plan and build interactions with less than 15% effects.

Table 4.6: Exact Search (Reduced TPC-H): Time [min]. Varied the number and interaction density of indexes. VNS: No optimality proof. DF: Did not Finish in 12 hours or out-of-memory.

$\lvert I\rvert$	6	11	13	22	31	16	21
Density	low	low	low	low	low	mid	mid
MIP	<1	11	106	DF	DF	DF	DF
CP	<1	7	214	DF	DF	DF	DF
MIP$^+$	<1					168	DF
CP$^+$	<1					1	DF
VNS	<1						<1?

As can be seen in Table 4.6, neither MIP nor CP could solve even small problems without problem specific constraints, taking time that grows factorially with the number of indexes. By applying the problem specific constraints (denoted by $^+$), both MIP and CP were dramatically improved and took less than one minute to solve all low-density problems. For higher density problems, they took substantially longer because the pruning power of additional constraints decreases. MIP suffered more from the higher density because it results in more non-linear properties discussed in Section 4.5. VNS quickly found the optimal solution in all cases. In the 21 indexes and mid-density problem, VNS found a good solution within one minute and did not improve the solution for 3 hours. This strongly implies the solution is optimal, but there is no proof as the exact search methods did not finish.

4.7.4 Local Search Results

We also studied TPC-H and TPC-DS with all indexes, query plans, and interactions. Because of the dense interactions and many more indexes, the search space increases considerably. Even CP with the problem specific constraints cannot prove optimality for this problem and gets suck in low quality solutions. Hence, we used our local search algorithms to understand how to find high quality solutions to these large problems.

Limited Scalability of MIP: The MIP model suffers severely on these large problems and CPlex quickly runs out of memory before finding a feasible solution with as much as 4 GB of RAM. This is because the denser problem significantly increases the number of non-zero constraints and variables, and CPlex cannot significantly reduce the problem size in the pre-solving step. In fact, over 1 million integer variables remain after pre-solving for problems of this size. This result verifies that a linear system approach does not scale well for the index ordering problem.

TPC-H Results: Due to the limited scalability of CP and MIP, we only evaluated the performance of local search algorithms (TS, LNS, and VNS) described in Section 4.6 on these problems. All the local search methods are implemented in COMET and given the same constraints with the same initial solution using the greedy algorithm from Section 4.6.4.

Figure 4.10 shows the quality (y-axis) of solutions plotted against elapsed search time (x-axis) for the TPC-H dataset. The figure compares the LNS, VNS and two Tabu Search (TS) methods described in Section 4.6.

In this experiment, TS-BSwap achieves a better improvement than TS-FSwap because TS-BSwap considers all possible swaps in each iteration. VNS is comparable to the two Tabu methods while the original form of LNS takes a long time to improve the solution because it cannot dynamically adjust the size of its neighborhood. We also observed that VNS is more *stable* than LNS in that it has less variance of solution quality between runs.

TPC-DS Results:

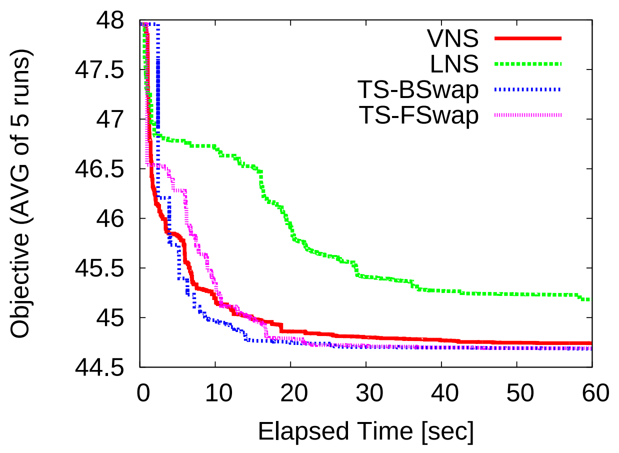

Figure 4.10: Local Search (TPC-H): LNS, VNS and Tabu. (MIP runs out memory)

Figure 4.11 compares VNS with Tabu Search for the TPC-DS dataset. This time, the improvement of TS-BSwap is large but very slow because it takes a very long time (50 minutes) for each iteration to evaluate $\binom{148}{2}$ swaps. VNS achieves the best improvement over all time ranges, followed by TS-FSwap. VNS quickly improves the solution, especially at the first 15 minutes. Considering that deploying the 148 indexes on the Scale-100 instance takes one day, VNS achieves a high quality solution within a reasonable analysis time.

Observations: The result verifies that VNS is a scalable and robust local search method which quickly finds a high quality solutions in all cases. The main reason the TS methods sometimes do not work well is essentially the same as why the LNS with fixed parameters does not perform well. The neighborhood size is fixed and it may be too large with TS-BSwap or too small with TS-FSwap.

It is possible to devise a hybrid Tabu method that dynamically adjusts the tuning parameters (the number of pairs to check, Tabu length, etc) for the problem, but VNS has another important property for avoiding local optima. As VNS relaxes more than two variables at each iteration, it can explore multi-swap neighborhoods that are necessary to influence large sets of interacting indexes.

4.8 Conclusion

In this chapter, we defined and solved the optimization problem of index deployment ordering. We formalized the problem using a mathematical model and studied several problem specific properties

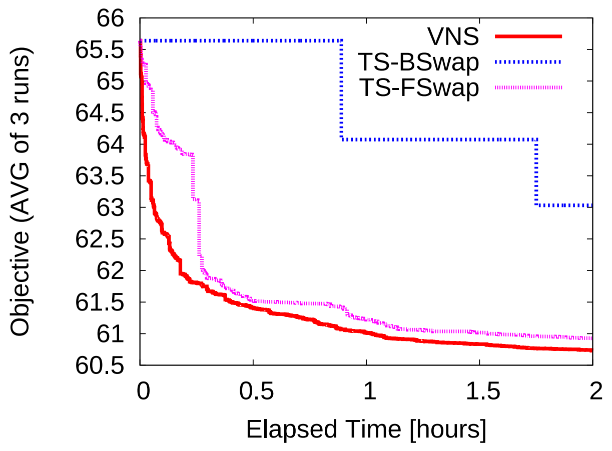

Figure 4.11: Local Search (TPC-DS): VNS and Tabu. (MIP runs out memory)

which increase performance of industrial optimization tools by several orders of magnitude. We developed several approaches for solving the problem including, a greedy algorithm, CP formulation, MIP formulation, and four local search methods. We demonstrated that this problem is best solved by a CP framework and found that our VNS local search method is robust, scalable, and quickly finds a near-optimal solution on very large problems.

4.8.1 Future Work

One open problem is how to jointly solve the index selection problem and index deployment ordering problem. We are currently working on an integrated solution that accounts for the index deployment ordering while choosing a set of indexes to build. The final goal is a database design method, we call *Incremental Database Design* (IDD), which reduces administrative costs for tuning large databases without sacrificing query performance improvements.

Chapter 5

On the Recoverability of Heterogeneously Partitioned Replicas in Distributed Filesystems

MapReduce systems, such as Hadoop, are rapidly becoming popular in big-data analytics. They are backed by shared-nothing distributed filesystems to replicate large objects over thousands of commodity servers for load balancing and fault tolerance. These replicas are partitioned by the same key to assure recoverability among them. However, the choice of partitioning keys has significant impact on query performance when MapReduce aggregates (Reduces) over related records because it needs to transmit a large amount of data among every node (Shuffle) otherwise. Due to the various analyses the user requires, flexible value-based partitioning is essential to improve MapReduce's performance. Here, we propose a new system on top of Hadoop that allows flexible and heterogeneous partitioning in which each replica can have a different partitioning key. The key challenge in this direction is the recovery because a recovery between differently partitioned replicas requires fully reading and repartitioning all blocks of another replica and also is more vulnerable to data loss. Our key contributions are the data structure and data placement policy to reduce the work and risk of recovery as well as an analytic model to predict the risk for a given hardware configuration and physical design. Our simulation experiments show that the techniques significantly improve recoverabilitiy and that our analytic model accurately captures how various factors in hardware and physical design affect recoverability.

5.1 Introduction

Many modern science and business applications generate massive amounts of data that is captured on many storage spindles in parallel. To name a few, human gene sequencing, astrophysics, data mining in social networks and e-commerce. These applications are struggling with processing the enormous data volumes they generate. Such big datasets are partitioned and distributed over a large number of machines to achieve high scalability. A MapReduce system such as Hadoop is becoming a common platform for distributed data analytics in this setting.

MapReduce systems make possible high scalability and a flexible programming model for both structured and unstructured data. The data is partitioned into small blocks (e.g., 64 MB chunks) and placed on a data node in the network. The MapReduce network typically consists of multiple racks each of which contains tens of commodity machines. Due to the large number of commodity machines, failures are common. MapReduce systems replicate every block (typically three times) and store them on different machines for high availability without permanent data loss or costly recovery.

The major bottlenecks in MapReduce systems are disk and network I/O. Unlike relational databases, current MapReduce systems such as Hadoop lack optimized data-access methods such as indexes, materialized views, and vertical partitioning (column-store). Scanning all of the data from disk involves massive disk I/O. MapReduce systems apply horizontal partitioning on the data, but they allow only an implicit partitioning based on the order in which data is loaded: they cannot have redundant copies of the data each partitioned and sorted differently.

When MapReduce systems need to process multiple datasets that are dependent on one another, they must transmit a large amount of data over the network to make sure they have all the related data at the same node. This network communication happens in the Shuffle phase and sometimes causes a major bottleneck by saturating the network bandwidth. Value-based partitioning has the potential to eliminate this bottleneck by locating related data in the same node. This idea, known as co-partitioning and co-location, is explored in [ETÖ+11, FPST11]. However, each query might require a very different partitioning. We propose a scheme in which each replica can be partitioned differently in order to minimize the need for network communication.

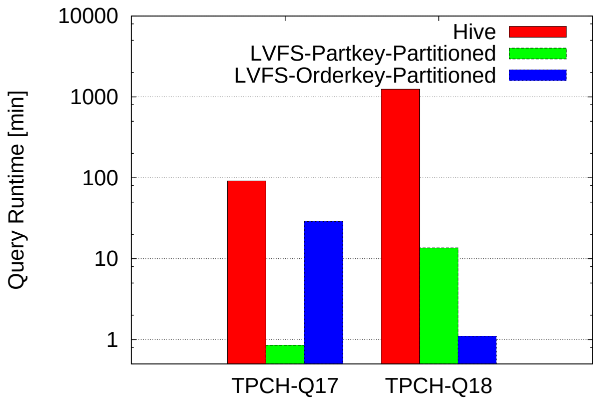

Figure 5.1: Motivating Experiments: Hive compared to our LVFS. LVFS achieves significantly faster query performance by efficient storage and execution layer. Further speed-up for an order of magnitude is available with beneficial partitioning, which suppresses the repartitioning phase. However, the beneficial partitioning varies among queries.

5.1.1 Preliminary Experiments and Motivation

As am example, Figure 5.1 shows a performance comparison between Hive and the new distributed file system, Las Vegas Filesystem (LVFS), described here [1]. LVFS employs distributed columnar storage and an optimized query-execution layer as in prior work [FPST11]. As observed in prior work, such a native columnar storage achieves a significant query speed-up compared to Hive. Furthermore, as the figure shows, an even more significant speed-up is possible with beneficial partitioning. This observation is consistent with other prior work suggesting value-based partitioning in Hadoop [ETÖ+11]. What is missing in this prior work is that ***each query requires a different partitioning***. TPC-H Q17 benefits from partkey-partitioning which suppresses repartitioning and redistribution in the query, while TPC-H Q18 benefits from orderkey-partitioning.

A homogeneous partitioning in Hadoop or other distributed systems does not satisfy this need. Instead, we need replicas partitioned on different keys each of which speeds up a different set of queries. Replicas that are simply mirrors are of no advantage in query processing. The goal here is to create such redundant and optimized data-access methods in MapReduce systems.

[1]Details of this preliminary experiment are given i Section 5.6.

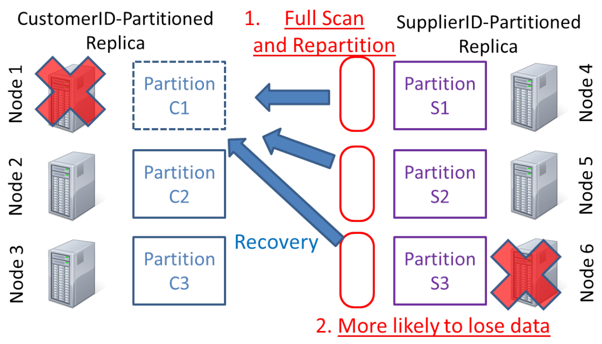

Figure 5.2: Key Challenge: Partitioning and Recovery.

5.1.2 Key Challenges

Simply partitioning each replica with different partitioning keys causes a few problems. First, partitioning the data based on values could require random in-place updates in the data files when new data are loaded, causing expensive random writes on disks. MapReduce systems usually allow only sequential writes for exactly this reason. Second and more important, altering the partitioning of data for each replica poses significant challenges to recovery. MapReduce systems consist of a large number of unreliable nodes some of which fail frequently. In such settings, recovery performance and no permanent data loss cannot be guaranteed without some additional algorithmic techniques to be described below.

To explore the second point further, Figure 5.2 shows a case in which two replicas are differently partitioned. Suppose Node 1 fails and its data is lost. As the figure illustrates, the corresponding data in another replica are scattered among many nodes. Hence, if another node fails, we can lose the data permanently. The risk is substantially higher than in the identical replication scheme in MapReduce where the corresponding data are always stored in one block. Furthermore, recovery time is also substantially greater because we must scan and repartition all blocks in another replica to recover the lost block, and this increases the probability of a critical node failure during recovery. Last but not least, if we place some blocks of different replicas in the same node, we lose the data stored in both of them when the node is lost.

Here, we suggest an alternative way of organizing and replicating data in MapReduce systems that addresses the aforementioned challenges. We have implemented a prototype system called Las Vegas on top of Hadoop that uses column-store techniques to vertically partition and compress data. It also makes possible an arbitrary number of differently partitioned and sorted data, similarly to materialized views in relational databases. We observe in the experimental section that these advantages together achieve orders of magnitude faster query performance for a variety of queries by exploiting replicas in Hadoop. Our data structures and data placement policy based on Fractures makes possible the efficient maintenance of each replica that entails only sequential writes and also reduces recovery time and the risk of data loss on failures. We have also devised analytic models to calculate recovery time and the probability of permanent loss of some data in the given situation.

These models have a wide range of use. The user or automated design tools can use them to choose the number of replicas and their partitioning, sorting, balancing recoverability, recovery latency, storage consumption, and query latency. Also, they can be used to issue an automated alert to the user when the risk of data loss exceeds some threshold, potentially automatically altering replicas to lower the risk. To the best of our knowledge, this work is the first to explore such an analytic model on recovery for differently partitioned replicas in distributed filesystems.

The remainder of this article is organized as follows. Section 5.2 gives a brief overview of our solution. Section 5.3 and Section 5.4 describe how we organize and place data in MapReduce systems in order to reduce the risk of data loss and speed up recovery time. Section 5.5 defines our analytic model to quantify the recovery time and probability of permanent data loss for a given data placement. Section 5.6 empirically evaluates our approaches, Section 5.7 reviews related work, and Section 5.8 discusses future work.

5.2 System Overview

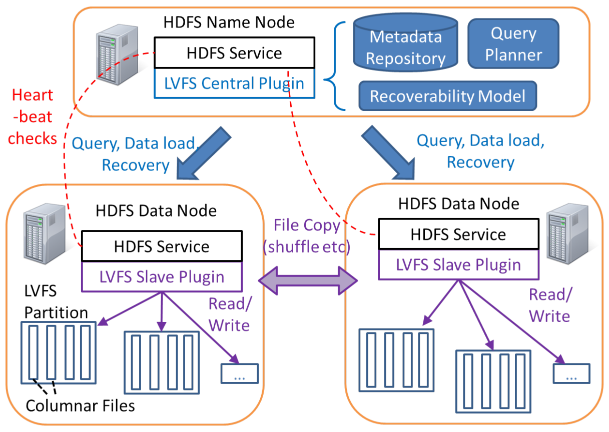

Figure 5.3: Las Vegas Filesystem (LVFS) Overview.

Figure 5.3 gives an overview of the Las Vegas Filesystem (LVFS), which is a set of plugins on top of Hadoop services. The central plugin runs as a part of the HDFS Name Node and stores metadata for LVFS, such as the partitioning and sorting keys for each replica, and the storage node of each partition for each replica. The slave plugin runs as a part of an HDFS Data Node and implements the columnar storage layer that is similar to [FPST11], that vertically splits records into columnar files and applies efficient compressions.

Every strategic decision, such as data placement and query planning, is made in the central node, which then assigns to data nodes local tasks such as reading files, executing query fragments, and copying files from other data nodes. The central node also periodically checks the availability of each data node by using HDFS's heartbeat messages. A node's failure to respond triggers a recovery job or an adjustment to a query plan if necessary.

The key issue here is how to organize partitions of the columnar files and how to determine their placement to reduce the risk of data loss; this is detailed in Sections 5.3 and 5.4. Further, there may be trade-offs between the risk of data loss and other factors, such as the replication factor (i.e., space consumption) and query performance. Thus, LVFS also provides an analytic model to predict the recoverability of various data placement plans based on the metadata stored in the central node. We detail this analytic model in Section 5.5.

Note that it is our central plugin, not HDFS, that directly governs recovery and the placement of data stored in LVFS since HDFS lacks built-in support for differently partitioned replicas.

5.3 Data Structure

This section describes how LVFS stores user data in each data node. For efficiency and flexibility, we organize each file in a columnar fashion similar to that in prior work [FPST11]. Each "table" is horizontally partitioned into multiple chunks that are then vertically split into columnar files in binary format.

There are three structural variations that differentiate the present approach from prior work. First, we do not replicate each columnar file using the underlying HDFS. Rather, we create additional set(s) of columnar files that store data logically identical to the original data table, but using different partitioning and/or sorting for data placement. How this is managed is described in Sections 5.3.1 and 5.3.2.

Second, we do not store each column in a single file. Instead, each data column is stored as a number of different *fractures* that are automatically merged in the background as necessary (each columnar file in the same table is fractured in an identical manner). The idea is similar to *differential files* or *log-structured merge tree* (LSM-tree). This makes it possible to alter data partitioning and sorting without random disk writes: All changes can be contained within the relevant subset of fractures. Furthermore, keeping a reasonable number of fractures reduces the amount of work and probability of data loss while recovering from failures. Section 5.3.3 details how it works.

Last, we apply custom placement to columnar files and the respective replicas in order to minimize the risk of data loss in the presence of different possible failures. The failure risks and the design to mitigate them is explained in Section 5.4 and then verified using both analytic models and experiments in Section 5.6.

5.3.1 Partitioning (Replica Group)

Each replica has a statically defined (i.e., assigned at its creation time) partitioning key and partition ranges. A partitioning key is typically one of the columns but can be an arbitrary expression computed from each tuple (e.g., a hash function) and the partitioning ranges are applied to the hash function result. As Figure 5.4 depicts, all tuples are horizontally partitioned according to non-overlapping key ranges. Each partition is then split into columnar files.

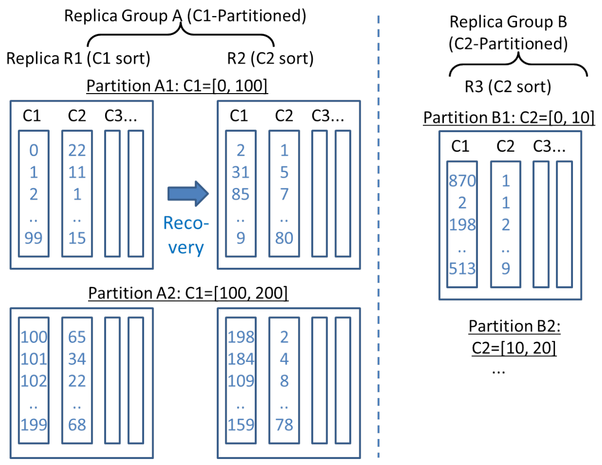

Figure 5.4: Replica Group and Replicas.

Two replicas that share the same partitioning key and partitioning ranges are called *buddy replicas* - such replicas differ only in the sorting and compression scheme, as is detailed in Section 5.3.2.

Recovery between buddy replicas is very efficient and safe: we simply read the column file for the particular partition that corresponds to the damaged column file in another partitioning scheme, then sort the data as necessary to replace the damaged file. In contrast, when two available replicas have different partitioning schemes, all the remaining partitions of the replica must be scanned to recover the damaged file.

Figure 5.4 provides an example. Let R1 and R2 be buddy replicas sorted by column C1 and C2 respectively. When R1-1 (R1's partition 1) is damaged, we read the data in R2-1 and sort it by the value of C1 in order to recover R1-1. This process is extremely fast and causes little disk and network I/O. When a partition of another replica R3 in another replica group R3-1 is damaged, however, we need to read all partitions in R1 or R2 and then repartition the data by C2 before we can recover R3-1. Although the amount of data transmitted over the network is the same (we first apply filtering based on the C2 partition), this process causes a significant amount of disk I/O and thus takes a long time.

5.3.2 Sorting, Compression and Indexing of Replicas

Each replica also has a statically defined sorting key and its own scheme to compress each of the columnar files. A sorting key is again typically one of the columns, but it can be an arbitrary expression or even not specified at all, in which case replica contents are not sorted. As Figure 5.4 shows, sorting is applied after partitioning and hence we call it as *in-block sorting.*

A columnar file may be compressed using run-length encoding, dictionary encoding, or a general compression algorithm such as gzip [2]. Columnar files support random access to arbitrary tuple positions (e.g., 123rd tuple) through sparse indexes that refer to the byte positions of individual tuples (e.g., 123rd value is placed at 3,422nd byte) with intervals punctuated by markers (e.g., every 1,000 tuples), except when the byte position is implicit based on the type of columnar file storage (e.g., when the column file stores four-byte integers without compression, the byte position can be calculated from the tuple position). The effectiveness of most compression schemes, such as run-length encoding, depends on the partitioning and sorting scheme, and we therefore allow specifying individual compression schemes for each replica.

We also maintain a sparse index file on the sorting column. Because the column is sorted, a sparse index allows an efficient value-based lookup while adding negligible space.

[2]We support Snappy compression for lightly compressed columns and gzip for heavily compressed columns.

5.3.3 Fracture

In addition to the value-based partitioning described in Section 5.3.1, we support another level of partitioning, *fracture*, that is based on the time when the data is loaded into the system. As Figure 5.5 illustrates, each fracture is an independent and self-contained (in terms of scheme) data set containing all replicas, their partitions, and columnar files in it.

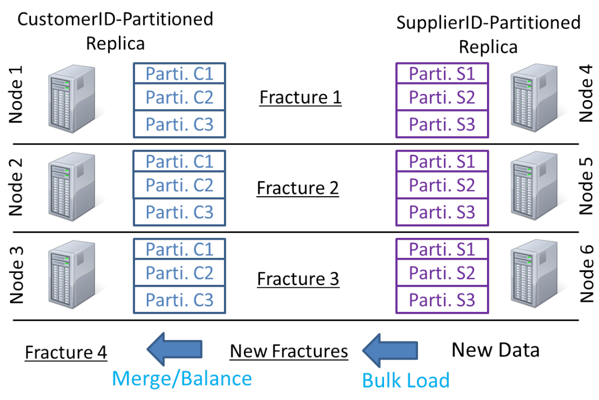

Figure 5.5: Fractures as Another Level of Data Partitioning.

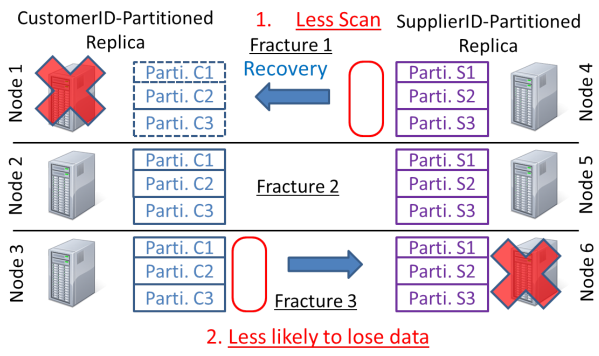

Figure 5.6: Recovery in Fractures.

Loading and merging fractures: When a new chunk [3] of data is loaded into the system, we consider it a new fracture and apply partitioning and sorting to the data as part of the loading process. This performs quickly as it involves only sequential reads and writes or potentially multiple stages of merge-sorts.

The arrival of every new chunk adds a new fracture in our data placement. However, as discussed

[3]If new data comes in a more ad-hoc manner, such as single-row inserts, we buffer insertions and dump them as a chunk. However, most use cases in MapReduce systems will have only bulk data loading.

below, having a large number of fractures can affect query performance. Therefore, we periodically merge multiple fractures into a single (larger) fracture. Just like the loading mechanism, fracture merging involves only sequential reads and writes, and is likely to be even faster than loading as all data sources are already sorted and compressed in a uniform fashion.

The ideal number of fractures and frequency of merging depend on the size of the data and the frequency of loading. For example, a small data file that is rarely changed (e.g., dimension tables in OLAP) should be composed of just one or at most two fractures. For fault tolerance, such data files can simply use multiple replicas with the same partitioning key.

The mechanism of converting batched updates into fracture in order to restrict the update to sequential writes is equivalent to log-structured merge tree (LSM-Tree) [OCGO96], which is also employed in BigTable [CDG+08]. The main difference is that we intentionally store data in a number of fractures and control their size and placement to minimize the risk of data loss.

Query over fractures: In most cases, fractures can be treated as independent subsets of data, and thus we can simply run the same query over each fracture and return a union of the results from each fracture, just as in the Reduce (or Combine) phase in MapReduce systems. In this case having many fractures does not hurt query performance.

However, when the query joins data across files (e.g., SQL JOIN), the number of fractures could affect query execution performance because all joins must be repeated for every fracture. In the worst case of a self-join, the execution cost could be $O(n^2)$ where n is the number of fractures in the data set. Thus, in such cases, fewer fractures will translate into significantly improved query performance.

Recovery in fractures: In contrast to query execution, increasing the number of data fractures improves the latency of recovery and the risk of data loss during recovery. As explained earlier, each fracture is an independent piece of data. As Figure 5.6 shows, damage to a file or a partition in one fracture does not affect other fractures in any way. Nor need we read any data from other fractures for recovery. Thus, fractures reduce the latency of recovery as well as the risk of data loss because each fracture is smaller and thus stored on fewer nodes. In short, with a larger number of fractures, we expect recovery to be both faster and safer.

However, there is a trade-off between query performance and recovery that will be determined by the number of fractures. The ideal number of fractures depends on the type of query workload, the size of entire data, redundancy of the schema, and data placement as detailed in the following section.

5.4 Data Placement

The placement of columnar files affects both query performance and data recoverability. For example, if all of the table files are placed on a single node, we cannot parallelize query execution over that table nor can we recover the table when the node fails. However, in that particular case, the data will be lost only if the one node holding this data fails.

In order to achieve the best balance between the query performance and recoverability, our file placement policy considers both the number of fractures and data partitioning.

5.4.1 Fracture Placement

As described in the previous section, each fracture can be treated as an independent unit for the purposes of recovery. Therefore we can design the best data placement strategy independently for each fracture.

Suppose we are designing a data placement scheme for a particular fracture. Let us first consider the groups of replicas in which buddy replicas are grouped together. Storing any two files from different replica groups on the same node will cause permanent data loss when the node is lost. Intuitively, as we are using a different portioning scheme across replica groups, any two files are likely to contain at least some matching rows. A similar problem occurs when such two files are stored in the same rack and the entire rack is lost (e.g., the switch is broken). Therefore, it is imperative that we separate files from different replica groups.

Dedicated racks: For this reason, we exclusively designate one or more racks to every replica group that shares a partitioning scheme. Such dedicated racks store only the files in the corresponding replica group, also restricting the amount of inter-rack communications even if a query requires a join of the data in the fracture. The number of racks assigned to the replica group depends on the size of the fracture, and is designed to distribute the load evenly.

We apply this policy to each fracture when it is created. Rack assignments are independent for each fracture, but we usually choose dedicated racks in a round-robin fashion for maximum parallelization of query execution.

5.4.2 Partition Placement

Figure 5.7, 5.8, and 5.9 show our techniques for placing data from each replica group; these strategies are evaluated in Section 5.6.

Buddy exclusion: Buddy replicas in a replica group share the partitioning key and its value ranges. Our data placement in the dedicated racks is based on the partition in the group. For example, suppose buddy replicas A and B both have partitioning ranges 1 to 8. Corresponding partitions of the buddy replicas, such as A1 and B1, are called buddy partitions and can be used to efficiently restore each other. So as always to exploit this potential, we place buddy partitions in different nodes or at least in different disk drives so that the probability of losing both simultaneously is low. Figure 5.7 shows a buddy replica placement that will result in data loss with a single node failure and a buddy exclusion placement that can withstand any single node failure without data loss.

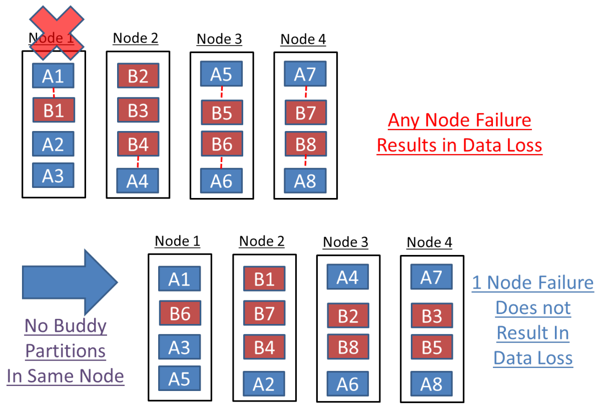
Node 1
Node 2
Node 3
Node 4
A1
B1
A2
A3
B2
B3
B4
A4
A5
B5
B6
A6
A7
B7
B8
A8
Any Node Failure Results in Data Loss
No Buddy Partitions In Same Node
Node 1
Node 2
Node 3
Node 4
A1
B6
A3
A5
B1
B7
B4
A2
A4
B2
B8
A6
A7
B3
B5
A8
1 Node Failure Does not Result In Data Loss

Figure 5.7: Buddy Exclusion.

Node coupling: We can further reduce the risk of data loss by coupling nodes to store the same set of partitions. In Figure 5.8, after buddy exclusion placement is implemented, data is permanently lost when any pair of nodes fail together. For example, when nodes 1 and 2 are lost together, we lose partition 1. The safest way to align the partitions is to couple nodes and store the same set of partitions in them. For instance, nodes 1 and 3 are coupled and both store partitions 1, 2, 3, and 4. Node 2 and 4 are coupled as well and store partitions 5, 6, 7, and 8. Figure 5.8 shows that the placement after applying node coupling permanently loses data only when nodes 1-3 and nodes 2-4 are lost together. Any other combinations of node failures does not result in data loss. We choose to couple nodes from different racks so that any single rack failure does not result in data loss.

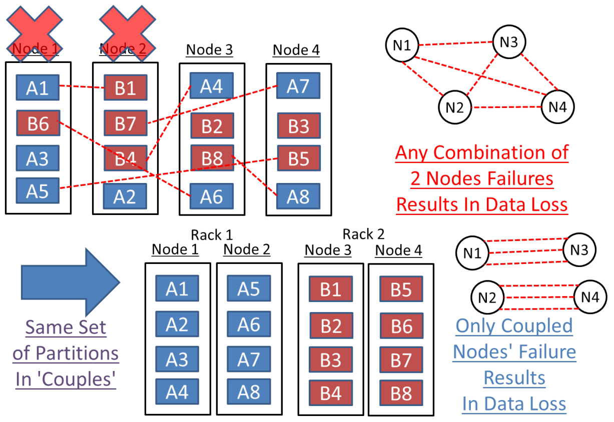

Figure 5.8: Node Coupling.

Buddy swapping: After applying the placement techniques described, the resulting data placement may limit the the performance of queries. Because only nodes 1 and 2 store the data of replica A, we can utilize only two nodes out of four to run queries over A. In order to achieve better parallelization, we *swap* buddy partitions so that the number of partitions for each replica is balanced for each node. For example, we swap A2 in node 1 and B2 in node 3. After the swapping, all four nodes store some partition of A; hence a query over A can be distributed to execute over all nodes in the cluster, as illustrated in Figure 5.9.

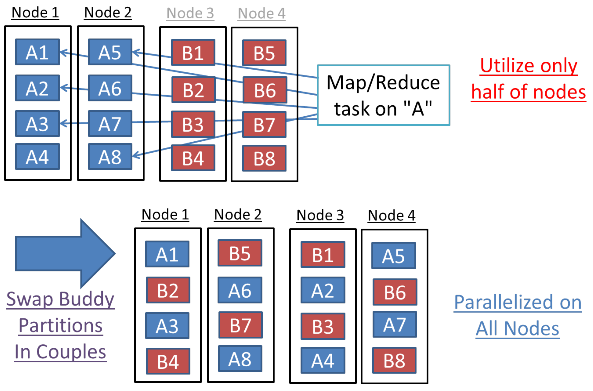

Figure 5.9: Buddy Swapping.

5.4.3 Copartitioning

Our system also allows specifying a copartitioning of a replica of a table with replica of another table. For example, let L1 be a replica of the LINEITEM table where L_PARTKEY is the partitioning key and P1 be a replica of the PART table where P_PARTKEY is the partitioning key. To run queries that join LINEITEM and PART tables, L1 and P1 are declared to be *linked*, which means our system synchronizes the partition ranges between the two replicas, placing the corresponding partitions (with same key values) in the same node.

Global tables: For small tables that are frequently used in JOIN queries, a table can be specified to be a *global* table. In that case, we copy all replicas of the table to all racks. This improves the flexibility of data placement in linked tables.

5.4.4 Columnar File Placement

Finally, following the design approach proposed in [FPST11], we place each family of columnar files of the same partition and the same replica in the same node. This allows a faster tuple-reconstruction without network I/O.

5.5 Analytic Recoverability Model

Here, we discuss our analytic cost model designed to estimate the probability of an un-recoverable data loss for a given design (a set of tables with specified partition and replication) and hardware configuration (racks, nodes, and their individual probability of failure).

Our goal is to predict with high accuracy the probability of data loss while keeping the cost of the estimation low. As in a query-cost model in databases, we need to evaluate this probability many times when selecting good data placement, partitioning and sorting.

Section 5.6 evaluates the accuracy of these analytic models compared to a simulator that actually simulates failures and recoveries over the lifetime of the system. Although such simulation is the most accurate way to estimate the recoverability and thus works well for our purposes, it requires huge computational resources and an exponentially large number of (or exponentially long) iterations to evaluate a highly recoverable design. Thus, using a simulation to estimate the probability of data loss is not a viable approach.

Let $P(f, t)$ be the probability of permanent loss of some data in the fracture $f \in F$ within a time period t. Let $P(t)$ be the probability of permanent loss of some data in some fracture within time

Table 5.1: Symbols.

$r \in R$	A rack ($\lvert R \rvert$ is the number of racks).
$n \in N$	A node ($\lvert N \rvert$ is the number of nodes).
$f \in F$	A fracture ($\lvert F \rvert$ is the number of fractures).
$g \in G$	A replica group.
$s \in S$	A replica.
$N(r)$	A set of nodes in the rack r.
$R(f,g)$	A set of racks assigned to replica group g for fracture f.
t	The period of time to consider (e.g., 10 years).

t. Assuming independence between data-loss events in each fracture yields

$$P(t) \;=\; \sum_{f}^{F} P(f,t)$$

Now consider how fracture f could permanently lose some data. Such data loss happens when all replica groups $g \in G$ concurrently lose some partition in a way that the partition cannot be recovered *within* the replica group. Let $P(g,f,t)$ be the probability of this event within a time period t. Now, assume such an event has happened (that is, the replica group g has lost fracture f within time period t). Once a fracture is lost, we need to recover the lost partition using other replica groups, which requires data repartitioning. Let $Q_{rep}(f)$ be the expected time to repartition the contents of the fracture from another replica group and store the resulting files in stable storage. A concurrent failure in all replica groups occurs when one of the replica groups fails and then other groups fail before the group can be repartitioned for recovery. Therefore,

$$P(f,t) \;=\; \sum_{g}^{G}(P(g,f,t) \times \prod_{g' \neq g}^{G} (P(g,f,Q_{rep}(f))))$$

$Q_{rep}(f)$ can be calculated using the size of the fracture, the rate to repartition, and the number of nodes assigned to each replica group. Then, in order to calculate $P(g,f,t)$, let us consider how replica group g can lose some partition in the group. The replica group can recover a partition between replicas in the group. Hence, such an event happens only when all buddy replicas $s \in S \in g$ lose the same partition. We can easily calculate the probability of such event assuming the use of the buddy exclusion and node coupling policies described in Section 5.4.

As described earlier, nodes are coupled in different racks in order to endure rack failure. To simplify, suppose $|S|$ racks $(r_1, r_2, \ldots, r_{|S|})$ have the same number of nodes and we couple the nodes in the racks. No node in the same rack has an overlapping partition with respect to the fracture f and the replica group g. Therefore, the only case in which a partition can be lost is when the coupled nodes fail at the same time.

Let $n \in N(r)$ denote the node in the rack r and $n', n'', \ldots$ the nodes coupled with n in other racks (there are $\frac{R(f,g)}{|S|}$ such coupled racks). Let $D_R(r,t)$ be the probability that the rack r fails (e.g., switch failure) within a time period t. Let $D_N(n,t)$ be the probability that the node n fails (e.g., HDD dies) within a time period t. Let $Q_{copy}(n)$ be the expected time to recover the node n from a crash, assuming that replacement hardware is immediately available when node n crashes. $Q_{copy}(n)$ can be calculated based on the size of data in the node, its network bandwidth, disk bandwidth, and the expected number of recoveries concurrently happening in the system (which might affect backbone network availability). Similarly to the prior $P(f,t)$ discussion, the probability of the concurrent replica failures is calculated as:

$$\begin{aligned} P(g,f,t) &= \frac{R(f,g)}{|S|} \sum_{r}^{r_1,r_2,\ldots,r_{|S|}} \left(\sum_{n}^{N(r)} D_N(n,t) \times \prod_{r' \neq r}^{r_1,r_2,\ldots,r_{|S|}} \right. \\ &\quad \left. (D_N(n', Q_{copy}(n)) + D_R(r', Q_{copy}(n))) \right) \end{aligned}$$

We then calculate $D_R(r,t)$ and $D_N(n,t)$ from the empirically calculated mean time to failure (MTTF) of a rack and a node, assuming exponential distribution of rack and node failures. Following the discussion in [FLP+10], we assume exponential distribution of failures because our focus is on node and rack failures; although expected disk failures might fit a Weibull distribution better, this issue does not significantly affect the distribution of node and rack failures because current hard disks are highly reliable and the disk ages in a large data center vary among machines.

Suppose $MTTF_N$ is the node MTTF. With the above assumption, $D_N(n,t)$ is simply:

$$D_N(n,t) = \int_0^t \frac{exp(\frac{-t}{MTTF_N})}{MTTF_N} dt = 1 - exp(\frac{-t}{MTTF_N})$$

Finally, we get the probability of losing some data in some object (table) by summing the $P(t)$ for each object. The data loss events of each table are assumed to be independent, in addition to the independence assumption between fractures. This is a reasonable assumption because each fracture in each object is randomly assigned to racks and they are thus unlikely to fail together. For example, a fracture that is assigned racks r_1, r_5, r_9 is very unlikely to lose data together with another fracture that is assigned racks r_2, r_7, r_8 or even r_2, r_7, r_9. The probability that two fractures are assigned to the same set of racks is quite low assuming a reasonably large number of racks. However, as observed in the experimental section, this independence assumption may not hold for designs that are extremely unreliable and our analytic model may yield an overestimation of the risk, although such designs will be out of the user's choice even without the overestimation.

5.6 Experiments

5.6.1 Implementation

We have implemented the Las Vegas filesystem (LVFS) described in earlier sections on top of Hadoop Distributed File System (HDFS). All source codes and experimental results are open-sourced at github.com/hkimura/las-vegas.

5.6.2 Query Runtime Experiments

Before the experiments about recoverability, we ran preliminary experiments about query runtime shown in Figure 5.1 above.

For this experiment, we used a cluster of 60 machines running Debian Squeeze with four cores and four GB of RAM. These machines are installed in three racks and have gigabit Ethernet interfaces. Between each query execution, we flush the disk cache in all nodes.

We loaded TPC-H Scale-1020 (1 TB without replication) onto Hadoop 0.20.2-dev, Hive 0.8.1 and our Las Vegas filesystem. For LVFS, we deployed two replicas, one partitioned by Partkey and another partitioned by Orderkey. Table 5.2 shows a drill-down analysis of the query runtime for TPC-H Q17 and Q18 using these replicas.

As can be seen in the simplified SQL for the two queries, Q17 needs to aggregate over Partkey while Q18 needs to aggregate over Orderkey. Unless the underlying data files are pre-partitioned and sorted by each aggregate key, we need to repartition the lineitem table, redistribute them over the network, and sort them before processing the query. On the other hand, a replica with the beneficial partitioning and sorting completely skips the steps and performs substantially faster, achieving speedups of two to three orders of magnitude compared to Hive.

Table 5.2: LVFS Query Runtime Drill-Down. Hive took 91 minutes for Q17 and 1245 minutes for Q18.

	TPC-H Q17 [min]		TPC-H Q18 [min]	
	Partkey	Orderkey	Partkey	Orderkey
Repartitioning	0	$5 \sim 6$	$3 \sim 4$	0
Redistribution and Sorting	0	$7 \sim 9$	$5 \sim 8$	0
Other	0.85	$1 \sim 12$	$0.5 \sim 1$	1.1
Total	0.85	28.8	13.6	1.1

TPC-H Q17: SELECT ... FROM lineitem JOIN part WHERE ... AND **L_QUANTITY<(SELECT 0.2*AVG(L_QUANTITY) FROM lineitem WHERE L_PARTKEY=P_PARTKEY)**

TPC-H Q18: SELECT ... FROM lineitem, orders, customer WHERE **O_ORDERKEY IN (SELECT L_ORDERKEY FROM lineitem GROUP BY L_ORDERKEY HAVING ...)** GROUP BY ... ORDER BY ...

We have not made comparisons with prior work on top of Hadoop or HDFS [ABPA+09, FPST11, ETÖ+11] because the source code is unavailable. However, the relative speedups from Hadoop/Hive achieved either by advanced storage layer techniques, such as native columnar storage, or by beneficial partitionings are quite consistent with observations in [ABPA+09, FPST11, ETÖ+11]. As remarked earlier, it was this result that motivated our exploration of multiple partitionings in distributed file systems.

5.6.3 Recoverability Experiments Setup

In order to analyze the recoverability and the accuracy of our analytic model, we have also implemented a simulator similar to that in [LCZ05], but with a notion of correlated failures (e.g., rack failures). In the simulator and our analytic model, we assumed the parameters in Table 5.3, which are partially derived from related work [FLP+10, LCZ05].

Using these parameters, the simulator randomly generates a failure event, either a node failure or a rack failure. Upon each failure event, the simulator simulates recovery tasks between each node and calculates their progress based on local disk rate, network rate, and backbone network rate

Table 5.3: Parameters for the Recoverability Simulator and Model.

Parameters	Values
Number of racks	60
Number of nodes per rack	40
Number of objects (tables)	100
Size of each object without replication	10 TB
Local disk rate	200 MB/s/node
Local network rate	100 MB/s/node
Backbone network rate	3 GB/s
Repartitioning rate*	20 MB/s/node
Node MTTF	4.3 month
Rack MTTF	10.2 years
Simulation period	10 years
Simulation iterations	100 times

* Repartitioning rate reflects all required steps of repartitioning: reading the replica, partitioning, transferring to other nodes, and writing to stable storage. Hence, it is much fewer than the local disk and network rate.

(which limits the total amount of data concurrently exchanged over the network). One iteration of the simulator terminates as soon as it detects some permanent data loss. We make 100 iterations and calculate the geo-mean of the time to lose some data.

Running one iteration of a 10-year simulation takes about half a minute on the simulator, so that 100 iterations take up to one hour. The analytic model, on the other hand, takes negligible time for prediction.

5.6.4 Recoverability Experiments

Fractures: We now evaluate recoverability with our data structure and data placement policy. We ran both the recoverability simulator as well as the analytic model described in Section 5.5. Figures 5.10 and 5.11 show the mean time to data loss (MTTDL), the time to observe a permanent data loss in the system with varying the number of fractures in each object (table). [2] is a design having only one replica group with two replicas, [1,1] has two replica groups with only one replica. In this experiments, we turned on all data-placement techniques for buddy replicas described in Section 5.4.

In the designs with heterogeneous partitioning ([1,1] and [1,1,1]), we observed higher MTTDL with more fractures because the fractures localize the risk of data loss and reduce the amount of data to be recovered from another replica. Without fractures, any single damaged partition requires repartitioning on another replica group that must be intact at the time.

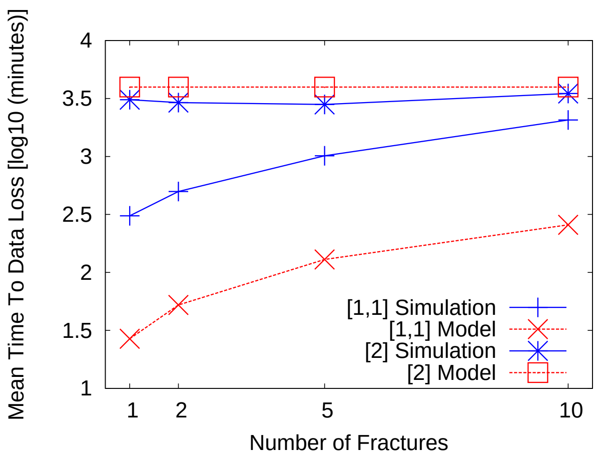

Figure 5.10: Recoverability and Fractures: LVFS Replication Factor = 2.

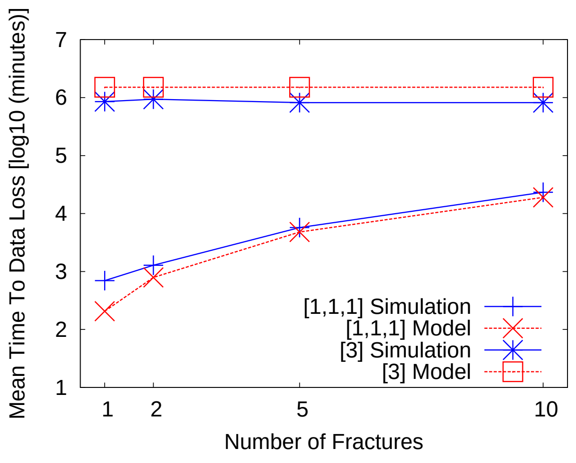

Figure 5.11: Recoverability and Fractures: LVFS Replication Factor = 3.

On the other hand, for homogeneous partitioning ([2] and [3]), fractures do not change the recoverability. This is because such these designs have only one partitioning key, just like HDFS, and so the damaged partition can be recovered from a buddy partition without repartitioning anyway. Assuming buddy exclusion and node coupling (as verified in the next experiment), the risk of data loss is already localized.

In all cases, our analytic model accurately predicated the MTTDL, capturing how the partitioning and fracturing affects the recoverability. It had a large error only when the expected MTTDL was extremely low (data loss is very frequent), such as [1,1]. In such cases, the independence assumption between the data losses does not hold: data losses of multiple tables and fractures happen at the same time. However, such a non-reliable design will be not a viable option to the user, so the overestimation of the risk in this case is irrelevant.

Data Placement Techniques: In this experiment we evaluate the data placement techniques for buddy replicas described in Section 5.4. Table 5.4 shows the results of simulations with and without each technique. The buddy-exclusion and node-coupling techniques significantly improved recoverability when there are buddy replicas. The buddy-swapping technique did not affect recoverability while it maximizes the query parallelism on the replicas.

Since these placement techniques have no drawbacks, we assume all of them are applied in subsequent experiments.

Table 5.4: Recoverability and Data Placement Policies: LVFS. $\phi(x\%)$ means that MTTDL was too high to estimate because $x\%$ of iterations observed no data loss during 10-year simulation.

Placement Policy	log10(MTTDL minutes)		
	[2]	[3]	[4]
None	1.54	3.10	4.60
BuddyExclusion	2.54	4.42	4.80
BuddyExclusion, NodeCoupling	3.54	5.92	ϕ(99%)
BuddyExclusion, NodeCoupling, BuddySwapping	3.54	5.91	ϕ(99%)

HDFS Analysis: As in the previous experiment, we also simulated the recoverability of HDFS to compare with LVFS. In this experiments, we turned on and off two data-placement techniques.

The first technique is Stripe-Chunk [LCZ05], which groups small file blocks to a large chunk that is used as the unit of replication. The goal is the same as in the present node coupling: to reduce the number of combinations in which the partition (block) are replicated to each node. We make the chunk size as recommended in [LCZ05] based on the ratio of the backbone network rate to the local bandwidth.

The second technique is trivial: the default HDFS replica placement policy places the second replica in the same rack to the first replica, then places other replicas in other racks. However, because of correlated failures in a rack, it is always safer to place even the second replica in other racks.

Table 5.5 shows the MTTDL calculated from the simulator with and without the two techniques. The result confirms that these techniques improve the recoverability of HDFS. The second-rack technique did not improve in the two-replica design because the MTTDL is so low in these cases anyhow that correlated failures (rack failures) do not happen during the simulation. Thus, since all node failures are uniformly random it does not matter whether or not we place the second replica in the same rack.

In all subsequent experiments, we assume these techniques are applied to HDFS for fair comparison.

Table 5.5: Recoverability and Data Placement Policies: HDFS.

Placement Policy	log10(MTTDL minutes)		
	[2]	[3]	[4]
Default	3.21	3.41	4.82
Stripe-Chunk	3.48	4.85	ϕ(71%)
Stripe-Chunk, 2nd-Rack	3.32	5.82	ϕ(98%)

Design Spaces: Finally, Table 5.6 compares the recoverability of various designs with all data placement techniques enabled and with ten fractures. The results are categorized by the redundancy (the total number of replicas). As discussed in prior work [FLP$^+$10, LCZ05], increasing the redundancy significantly improves the recoverability in both LVFS and HDFS at the cost of more space consumption and maintenance overheads. This is the reason Hadoop provides the replication factor (redundancy) as an important tuning parameter for users.

We extend this design flexibility to another dimension, the number of different partitionings. While HDFS allows only homogeneous partitioning, LVFS enables each replica to be partitioned by different keys, thereby speeding up different sets of queries. The number of different partitionings has an interesting tradeoffs between this opportunity for query optimization and the recoverability. The more replica groups, having fewer replicas in each replica group causes more frequent repartitioning and sometimes data loss on node failures.

Our analytic model accurately captures the tradeoffs among the redundancy, number of partitionings, and recoverability. This result shows that our analytic model can help the user choose the best design conforming to requirements.

5.7 Related Work

Efforts have been made for decades to store and analyze large amounts of data. Parallel and distributed databases [DGS$^+$90] extend the scalability of datawarehouses to shared-nothing clusters.

Table 5.6: Design Spaces: Tradeoffs among Redundancy, Number of Different Partitionings, and Recoverability.

Design Choice	Replica Factor	#Partitionings	log10(MTTDL minutes) Simulation	Model
HDFS [2]	2	1	3.32	N/A
LVFS [2]	2	1	3.54	3.60
LVFS [1,1]	2	2	3.31	2.41
HDFS [3]	3	1	5.82	N/A
LVFS [3]	3	1	5.91	6.18
LVFS [1,2]	3	2	4.89	5.17
LVFS [1,1,1]	3	3	4.37	4.28
HDFS [4]	4	1	ϕ(98%)	N/A
LVFS [4]	4	1	ϕ(99%)	8.56
LVFS [2,2]	4	2	ϕ(93%)	7.97
LVFS [1,3]	4	2	ϕ(70%)	7.67
LVFS [1,1,2]	4	3	ϕ(3%)	6.91
LVFS [1,1,1,1]	4	4	5.47	6.20

MapReduce [DG08] systems, such as Hadoop, employ the simpler architecture for ease of use and higher scalability.

The MapReduce are systems backed by the distributed storage system, such as HDFS, which splits the object (file or table) into small pieces and replicates them onto a number of nodes for fault tolerance. Dean et al. [Dea09] reported a various hardware and software failures in MapReduce systems due to the large number of nodes and commodity hardware. Lian et al. [LCZ05] built a Markov model to evaluate the risk of data loss in MapReduce considering the saturation of network bandwidth by concurrent recovery tasks. Ford et al. [FLP$^+$10] observed failure bursts in which many nodes in a rack fail concurrently, a major risk of data loss.

MapReduce typically stores row-oriented text files, parsing the files during query execution. It does not have special data structures to speed up queries, such as indexes and materialized views in database systems. Pavlo et al. [PPR$^+$09] pointed out that the lack of advanced data structures and query optimizations sometimes causes query performance orders of magnitude slower than in database systems.

Extensive research efforts have been made to address this drawback of MapReduce systems, especially Hadoop. Floratou et al. [FPST11] studied native columnar storage formats in MapReduce, observing orders of magnitude faster performance than for text files and PAX-formats such as RC-File [HLH$^+$11]. CoHadoop [ETÖ$^+$11] was the first to explore value-based partitioning, as opposed to file offsets, in Hadoop. CoHadoop achieves significant query speed-up by copartitioning joined tables and co-locating them in same nodes. Hadoop++ [DQRJ$^+$10] and Trojan Data Layout [JQRD11] alter the data structure and query execution in Hadoop to inject index data in an nonintrusive way (without affecting Hadoop's replication and recovery features) and utilize it in query processing. On the other hand, HadoopDB [ABPA$^+$09] stores structured data in relational databases to exploit advanced indexing and query optimizations in DBMS.

Despite this plethora of prior work, none has considered heterogeneously partitioned replicas, the key issue here. Replicas in HDFS are simply identical replicas, so that recovery is much simpler. The only exception is the Trojan Data Layout [JQRD11], which alters the data layout in each replica to be in row, columnar, or PAX formats. However, it still assumes an identical partitioning so that the data in corresponding replicas are logically the same. To allow multiple partitioning, one must address the issue of recovery between replicas, the primary focus here.

5.8 Conclusions

We have proposed a new approach to data redundancy in distributed data analytic platforms such as MapReduce systems. We demonstrated the potential of different partitioning for each replica in distributed filesystems to speed up various queries. We then identified the key challenge that this approach introduces to *recovery*.

We developed a distributed filesystem called Las Vegas filesystem (LVFS). LVFS allows flexible partitioning in addition to sorting of the data. The data structure and data placement policies reduce the risk of data loss and speed up recovery between replicas even if they are differently partitioned.

We also developed an analytic model to evaluate recoverability of the given design quickly and accurately. The analytic model can be employed by users or automatic design tools to determine a physical design that balances recoverability and query performance.

Our experiments verify that a beneficial partitioning along with an efficient storage layer in LVFS realizes orders of magnitude faster query performance compared to Hive. We verified through simulation that our data structure and data placement policies improve recoverability.

Our most notable observation from the simulation results is the tradeoff between recoverability and the number of different partitionings, or the opportunity to speed up different sets of queries. Unlike uniformly partitioned filesystems, LVFS gives users another possible design dimension to better meet their query requirements.

5.8.1 Future Work

The fact that LVFS allows more flexibility in the physical data layout complicates the process of performance tuning. The user must now choose partitioning schemes and alternate query implementation for the replica with beneficial partitioning for the query. The main reason Hadoop is now a common data analytic platform is that it is extremely easy to set up, use, and manage.

We thus plan to work on automating the design process. The key requirements are a declarative (or semi-declarative, such as Pig-Latin) query language, a query execution layer equipped with a cost-based query optimizer, and an automated physical design framework that uses the analytic model developed here to satisfy the user's recoverability requirements.

Finally, we are also interested in figuring out which of the findings here can be integrated into the mainstream HDFS and Hive projects in a simple way. For example, unlike differently partitioned replicas, replicas that are differently sorted in each block can be implemented on HDFS without substantial changes in a similar way to the Trojan Data Layout [JQRD11], because in-block sorting does not change the logical information stored in the block.

Chapter 6

Uncertain Primary Index: *An Application to Uncertain Databases*

Uncertain data management has received growing attention from industry and academia. Many efforts have been made to optimize uncertain databases, including the development of special index data structures. However, none of these efforts have explored primary (clustered) indexes for uncertain databases, despite the fact that clustering has the potential to offer substantial speedups for non-selective analytic queries on large uncertain databases. In this chapter, we propose a new index called a UPI (*Uncertain Primary Index*) that clusters heap files according to uncertain attributes with both discrete and continuous uncertainty distributions.

Because uncertain attributes may have several possible values, a UPI on an uncertain attribute duplicates tuple data once for each possible value. To prevent the size of the UPI from becoming unmanageable, its size is kept small by placing low-probability tuples in a special Cutoff Index that is consulted only when queries for low-probability values are run. We also propose several other optimizations, including techniques to improve secondary index performance and techniques to reduce maintenance costs and fragmentation by buffering changes to the table and writing updates in sequential batches. Finally, we develop cost models for UPIs to estimate query performance in various settings to help automatically select tuning parameters of a UPI.

Table 6.1: Running Example: Uncertain *Author* table

Name	*Institution*^{p}	Existence	...
Alice	Brown: 80%, MIT: 20%	90%	...
Bob	MIT: 95%, UCB: 5%	100%	...
Carol	Brown: 60%, U. Tokyo: 40%	80%	...

Query 1: Example Uncertain Query.
SELECT * FROM Author WHERE Institution=MIT
Threshold: confidence $\geq QT$ (QT is given at runtime)

We have implemented a prototype UPI and experimented on two real datasets. Our results show that UPIs can significantly (up to two orders of magnitude) improve the performance of uncertain queries both over clustered and unclustered attributes. We also show that our buffering techniques mitigate table fragmentation and keep the maintenance cost as low as or even lower than using an unclustered heap file.

6.1 Introduction

A wide range of applications need to handle uncertainty. Uncertainty comes from sources such as errors in measuring devices (e.g., sensors), probabilistic analysis, and data integration (e.g., integration of multiple semantic databases that are potentially inconsistent). As shown by the large body of recent research in this area [DRS09, BSHW06, Suc08, CXP$^+$04, SIC07], there is a high demand to process such uncertain data in an efficient and scalable manner.

The database community has made great progress in the area of uncertain databases by establishing new data models, query semantics and optimization techniques. Several models for uncertainty in databases have been proposed. In the most general model, both tuple existence and the value of attributes can be uncertain. For example, Table 6.1 shows 3 uncertain tuples in the *Author* table of a publications database modeled after the DBLP computer science bibliography (see Section 6.7.1 for how we derived the uncertainty). Each tuple has an existence probability that indicates the likelihood it is in the table and an uncertain attribute (denoted as ^{p}) *Institution* that the author works for. In the example, Alice exists with probability 90% and, if she exists, works for Brown with probability 80% and MIT with probability 20%.

Possible World Semantics [DRS09] is a widely used model for uncertainty in databases. It conceptually defines an uncertain database as a probability distribution over a collection of possible database instances (*possible worlds*). Each possible world is a complete, consistent and deterministic database instance as in traditional DBMS. For example, there is a possible world where Alice exists and works for Brown, Bob works for MIT and Carol does not exist. The probability of such a world is $90\% \times 80\% \times 95\% \times 20\% \approx 13.7\%$. Based on possible world semantics, a probabilistic query over an uncertain database can output tuples along with a *confidence* indicating the probability that the tuple exists in some possible world where it satisfies the query predicates. For example, Query 1 would answer {(Alice, confidence=$90\% \times 20\% = 18\%$), (Bob, 95%)}. Thus, confidence represents how probable each answer is. Users can also specify thresholds on the minimum confidence they require from query results (the QT in Query 1.)

Though possible world semantics is a widely used data model, achieving an efficient implementation is difficult. In particular, it requires a new approach to data storage, access methods and query execution [DRS09]. One active area of research has been in building index data structures to efficiently answer queries over such probabilistic tables [CXP$^+$04, TCX$^+$05, ACTY09]; the primary addition that these data structures provide over traditional B+Trees and R-Trees is the ability to find tuples with confidence above some specified threshold.

These proposed indexes, however, are *secondary* indexes. To the best of our knowledge, no work has been done to cluster a heap file containing uncertain attributes as a *primary* index. To address this limitation, the key contribution of this work is to propose techniques to build primary indexes over probabilistic databases. Just as in a conventional (non-probabilistic) database, a primary index can be orders of magnitude faster than a secondary index for queries that scan large portions of tables, for example in OLAP workloads. Because a secondary index stores only index keys with pointers to corresponding tuples in the heap file, the query executor has to access the heap file by following the pointers to retrieve non-indexed attributes. This can cause an enormous number of random disk seeks for an analytical query that accesses millions of tuples, even if the query executor sorts the pointers before accessing the heap file (e.g., *bitmap index scan*). Furthermore, recent work has shown that building primary indexes on appropriate attributes can also boost the performance of secondary indexes that are correlated with the primary index [KHR$^+$09]. In this chapter, we demonstrate that a new primary index structures on uncertain attributes can be up to two orders of magnitude faster than a secondary index and can boost the performance of secondary indexes by up to two orders of magnitude when an appropriate correlated primary index is available.

However, building a primary index on uncertain attributes poses several challenges. If we simply cluster a tuple on one of its possible values, a query that is looking for other possible values needs additional disk accesses. For example, if we store Carol in a *Brown* disk block, a query that inquires about *U. Tokyo* authors must access the Brown block in addition to the *U. Tokyo* block. One solution is to replicate tuples for every possible value, but this makes the heap file very large and increases maintenance especially for *long tail* distributions with many low probability values. Furthermore, building a primary index on attributes other than auto-numbered sequences imposes a significant maintenance cost (to keep the heap clustered) and leads to fragmentation of the heap file over time, which also slows down the query performance.

In this chapter, we develop a novel data structure we call the UPI (*Uncertain Primary Index*), which is a primary index on uncertain attributes with either discrete or continuous distributions. UPI replicates tuples for all possible values but limits the penalty by storing tuples with a probability less than some threshold in a *Cutoff Index*. We propose a novel data structure for secondary indexes built over UPIs that stores multiple pointers for each entry to take advantage of the replicated tuples. We also describe the *Fractured UPI* which buffers data updates and occasionally flushes them to a new partition, or a *fracture* to reduce maintenance costs and fragmentation. Our experimental results on two real uncertain datasets show that UPI has substantial performance gains and similar maintenance costs to (unclustered) heap files.

In summary, our contributions include:

- The UPI data structure and corresponding secondary indexes
- Algorithms to answer queries using UPIs
- Methods to reduce update cost and fragmentation of UPIs
- Cost models to help select cutoff values and guide the formation of cutoff indexes
- Experimental results on real datasets that verify our approach and demonstrate order-of-magnitude performance gains over existing secondary indexses

In the next section, we describe a naive implementation of UPI and discuss its limitations. Sections 6.3 through 6.6 extend UPIs to address these limitations. Section 6.7 validates our approach with intensive experiments on two real datasets. Finally, Section 6.8 summarizes related work and Section 6.9 concludes this chapter.

6.2 A Simple UPI

We begin by describing a naive implementation of UPIs, followed by a discussion of their shortcomings that are addressed in later sections.

To answer Query 1, an uncertain secondary index on *Institution* would be inefficient because there are thousands of researchers who work for MIT, and each would require a random disk seek to fetch. Instead, if we build a UPI on *Institution*, it will duplicate each tuple once for each possible value of *Institution*, as shown in Table 6.2.

Also, we do not need tuples that have less than QT probability to satisfy the query. Therefore, we order the tuples by decreasing probability of institution, which allows the query executor to terminate scanning as soon as it finds a tuple that has a lower probability than the query threshold. Physically, the heap file is organized as a B+Tree indexed by {*Institution* (ASC) and *probability* (DESC)}. This is similar to the inverted index in [SMP+07] except that we duplicate the entire tuple, rather than just a pointer to the heap file.

This scheme achieves significantly faster performance than a secondary index for Query 1 because it requires only one index seek followed by a sequential scan of matching records. However, this naive UPI has several limitations.

First, since it duplicates the whole tuple for every possible value of *Institution*, the size of the heap file can be significantly larger than a heap file without the primary index. This is especially true when the probabilistic distribution has a long tail (i.e., many possible values with low probabilities).

Second, now that a single tuple exists in multiple places on disk, it is not clear how we should organize secondary indexes. Specifically, if we could use the duplicated tuples, a query could use the secondary index to access fewer heap blocks (fewer seeks) and run substantially faster.

Third, maintaining UPIs causes two problems. As newly inserted or deleted tuples will have different values of *Institution*, we need to update the B+Tree nodes in a variety of locations leading to many disk seeks. Also, splits and merges of B+Tree nodes will fragment the disk layout of the UPI and degenerate query performance.

Lastly, the naive approach applies only to tuples with discrete probability distributions. For continuous distributions like Gaussians, we need index schemes other than B+Trees.

We address these problems in turn. Section 6.3 describes the design *Cutoff Indexes* to address long-tail distributions and proposes a new index data structure for a secondary index that exploits

Table 6.2: A Naive UPI. Sorted by institution and probability.

<u>*Institution*</u>^{p}↓ (<u>*Probability*</u>↑)	<u>*TupleID*</u>	Tuple Data
Brown (80%*90%=72%)	Alice	...
Brown (60%*80%=48%)	Carol	...
MIT (95%)	Bob	...
MIT (18%)	Alice	...
UCB (5%)	Bob	...
U. Tokyo (32%)	Carol	...

duplicated tuples in the UPI. Section 6.4 explains the design of *Fractured UPIs* that minimize UPI maintenance cost and fragmentation. Section 6.5 extends UPIs for continuous distributions. Finally, Section 6.6 defines cost models which are useful to design and maintain UPIs.

6.3 Improved UPI

In this section, we improve our basic UPI design by addressing issues with the database size and improving the performance of secondary indexes on the same table as the UPI.

6.3.1 Cutoff Index

One problem with our naive UPI is that the database size can grow significantly when a tuple has many possible values of the indexed attribute. This increased size will not only affect the storage cost but also increase maintenance costs.

We observe, however, that for long-tailed distributions, with many duplicated values, the user may not care about very low confidence tuples, since those are unlikely to be correct answers. For example, Query 1 includes the threshold *QT* that filters out low-confidence tuples. Such queries are called Probabilistic Threshold Queries, or PTQs, and are very common in the literature [CXP$^+$04, TCX$^+$05, ACTY09]. For PTQ's, low probability tuples can typically be ignored.

We anticipate that most queries over long-tailed distributions will be PTQs. To handle such queries, we attach a *Cutoff Index* to each UPI heap file. The idea is that the query executor does not need to read the low probability entries when a relatively high probability threshold is specified in a PTQ. Therefore, we can remove such entries from the UPI heap file and store them in another index, which we call the *cutoff index.* The cutoff index is organized in the same way as the UPI heap file, ordered by the primary attribute and then probability. It does not, however, store the entire

```
Input: $t$: Inserted tuple, $C$: Cutoff threshold.
$Alternatives$ = sort_by_probability ($t$.primary_attribute);
foreach $a \in Alternatives$ do
    if $a = Alternatives.first$ OR $a.probability \geq C$ then
        Add (key: $a$, tuple: $t$) to Heap File;
    else
        Add (key: $a$, pointer: $Alternatives.first$, TupleID: $t.TupleID$) to Cutoff Index;
    end
end
```

Algorithm 2: Insertion into a UPI

```
Input: $key$: Queried value, $QT$: Probability threshold, $C$
Output: $S$: Set of tuples to return.
$S = \emptyset$;
$Cur$ = UPI.$seekTo$ ($key$);
while $Cur.key = key$ AND $Cur.probability \geq QT$ do
    $S = S \bigcup Cur.tuple$;
    $Cur.advance()$;
end
if $QT < C$ then
    $Cur$ = CutoffIndex.$seekTo$ ($key$);
    while $Cur.key = key$ AND $Cur.probability \geq QT$ do
        $CurIn$ = UPI.$seekTo$ ($Cur.pointer$);
        $CurIn$.moveTo ($Cur.TupleID$);
        $S = S \bigcup CurIn.tuple$;
        $Cur.advance()$;
    end
end
```

Algorithm 3: Answering a PTQ using a UPI

tuple but only the uncertain attribute value, a (*pointer*) to the heap file to locate the corresponding tuple, and a tuple identifier (*TupleID*). For example, in Table 6.3, the Bob tuple with institution value *UCB*, which has only 5% probability, is moved to the cutoff index with a pointer to another possible value of Bob (MIT).

Top-k queries and nearest neighbor (*NN*) queries [SIC07] benefit from the cutoff index as well. A top-k query can terminate scanning the index when the top-k results are identified. Thus, a cutoff index is particularly useful when a majority of the queries on the database are PTQs or Top-k.

Algorithm 2 shows how we build and maintain UPIs and cutoff indexes. Given a *Cutoff Threshold* C for the UPI, we duplicate a tuple in the UPI for every possible value that has probability equal to or greater than C. For every possible value with probability less than C, we insert a pointer to the first possible value of that tuple (a value that has highest probability) into the cutoff index. If

Table 6.3: Cutoff Index to compress UPI (C=10%)

UPI Heap File

Brown (72%)	Alice	...
Brown (48%)	Carol	...
MIT (95%)	Bob	...
MIT (18%)	Alice	...
U. Tokyo (32%)	Carol	...

Cutoff Index

<u>*Key*</u>↓	<u>*TupleID*</u>	Pointer
UCB (5%)	Bob	MIT

Stores pointers for possible values with probability $< C$

a value has probability lower than C, but is the first possible value, we leave the tuple in the UPI instead of moving it, to not lose tuples that do not have any possible value with probability larger than C.Deletion from the UPI is handled similarly, deleting entries from the heap file or cutoff index depends on the probability. Updates are processed as a deletion followed by an insertion.

Algorithm 3 shows how we use the UPI to answer PTQs. When C is less than QT, we simply retrieve the answer from the UPI heap file, which requires only one index seek. When C is larger than QT, we additionally need to look in the cutoff index to retrieve cutoff pointers and perform an index seek for each pointer.

The value of C is an important parameter of a UPI that the database administrator needs to decide. Larger C values could reduce the size of the UPI by orders of magnitude when the probability distribution is long tailed. But, they substantially slow the performance of PTQs with query threshold less than C, since such queries require pointer-following (and many random I/Os.) Smaller values of C work well for a large mix of queries with varying QT, at the cost of a larger UPI. To help determine a good value of C taking into account both the workload and limits on storage consumption and maintenance cost, we developed an analytic model for cutoff index performance; we present this model in Section 6.6.

6.3.2 Secondary Indexes on UPIs

Another challenge is exploiting the structure of UPIs to improve secondary index performance. A secondary index in conventional databases points to a single tuple in the heap file by storing either a *RowID* consisting of physical page location and page offset (e.g., PostgreSQL) or the value of the primary index key (e.g., MySQL InnoDB). Unlike such traditional secondary indexes, in UPIs, we employ a different secondary index data structure that stores multiple pointers in one index entry, since there are multiple copies of a given tuple in the UPI heap.

For example, suppose $Country^p$ is another uncertain attribute of the relation Author shown in

```
Input: key: Queried value, QT: Probability threshold.
Output: P: Set of pointers to heap file.
$P = \emptyset$;
$Entries = SecondaryIndex.select(key, QT)$;
foreach $e \in Entries$ do
    if $e.pointers.length = 1$ then
        $P = P \bigcup e.pointers[0]$;
    end
end
foreach $e$ in $Entries$ do
    if $\forall p \in e.pointers : p \notin P$ then
        $P = P \bigcup e.pointers[0]$;
    end
end
```

Algorithm 4: Tailored Secondary Index Access

Table 6.4: *Country*^{p} in Author table

Name	*Institution*^{p}	*Country*^{p}	Existence
Alice	Brown: 80%, MIT: 20%	US: 100%	90%
Bob	MIT: 95%, UCB: 5%	US: 100%	100%
Carol	Brown: 60%, U. Tokyo: 40%	US: 60%, Japan: 40%	80%

Table 6.5: Secondary Index on *Country*^{p}

Country^p↓	*TupleID*	*Pointers*	
Japan (32%)	Carol	Brown	U. Tokyo
US (100%)	Bob	MIT	<*cutoff*>
US (90%)	Alice	Brown	MIT
US (48%)	Carol	MIT	U. Tokyo

Table 6.4 with a secondary index on it as shown in Table 6.5. Each row in the secondary index stores all possible values of the primary attribute (*Institution*^{p}), except cutoff values. Algorithm 4 shows our algorithm for answering PTQs using these multiple pointers. For example, suppose the following PTQ is issued on *Country*^{p} with $QT = 80\%$:

SELECT * FROM Author WHERE Country=US

We first retrieve matching entries from the secondary index (Bob and Alice) and then find entries that have only one pointer (Bob). We record the institution for these pointers (MIT) and then check other secondary index entries, preferentially choosing pointers to institutions we have already seen. In the above case, Alice contains a pointer to MIT, so we retrieve tuple data for Alice from the MIT record about her. The advantage of this is that because Bob's data is also stored in the MIT portion of the heap, we can retrieve data about both authors from a small, sequential region of the heap corresponding to MIT. If there is no pointer to an institution we have already seen, we simply pick the first (highest probability) pointer. Note that in this case we would have accessed two disk blocks (MIT and Brown) if the secondary index stored only the first pointers.

We call this algorithm as *Tailored Secondary Index Access* and demonstrate in Section 6.7 that it can speed up secondary indexes substantially for analytical queries . One tuning option for this algorithm is to limit the number of pointers stored in each secondary index entry. Though the query performance gradually degenerates to the normal secondary index access with a tighter limit, such a limit can lower storage consumption.

6.4 Fractured UPI

In this section, we describe a second extension to UPIs called *Fractured UPIs.* The idea of fracturing is to reduce UPI maintenance cost and fragmentation. The approach we take is similar to that taken in log structured merge trees (*LSM-Trees*) [OCGO96] and partitioned exponential files [JOY07] for deterministic databases, which try to eliminate random disk I/O by converting all updates into appends to a log similarly to *deferred updates* of transaction processing.

6.4.1 The Maintenance Problem

The problem of maintaining a UPI is that insertion or deletion may perform random I/O to the UPI to retrieve pages. This makes the maintenance cost of UPIs much higher than for an append-only table without primary indexes.

Another problem is that insertions cause splits of B+Tree nodes when nodes become full, and deletions cause merges of nodes. Thus, over time, these operations result in fragmentation of the primary index, leading to random disk seeks even when a query requests a contiguous range of the primary index attribute.

For these two reasons, primary B+Tree indexes sometimes have adverse effects on performance over time [JOY07], canceling out the initial benefits obtained by clustering a table on some key.

6.4.2 Fractured UPI Structure

To overcome these problems, we store UPIs as *Fractured* indexes [Ikh10]. Figure 6.1 shows the structure of a Fractured UPI. The *insert buffer* maintains changes to the UPI in main memory.

When the buffer becomes full, we sequentially output the changes (insertions and deletions) to a set of files, called a *Fracture*. A fracture contains the same UPI, cutoff index and secondary indexes as the main UPI except that it contains only the data inserted or deleted since the previous flush. Deletion is handled like insertion by storing a *delete set* which holds IDs of deleted tuples. We keep adding such fractures as more changes are made on the UPI, and do not immediately update the main UPI files.

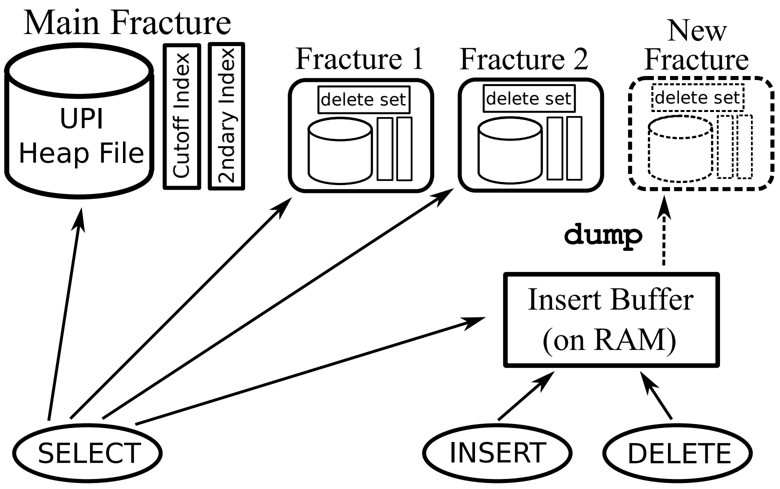

Figure 6.1: Fractured UPI Structure

To answer a SELECT query, the query executor scans the insert buffer and each fracture in addition to the main UPI, returning the union of results from each file and ignores tuples that were contained in any delete set. In this scheme, all files are read-only and are written out sequentially by the clustering key as a part of a single write. Therefore, the maintenance cost is significantly lower and there is essentially no fragmentation.

One difference from prior work (e.g., [OCGO96]) is that a fracture contains a *set* of indexes that constitute an independent UPI. A secondary index or a cutoff index in a fracture always points to the heap file in the same fracture. This architecture makes query execution in each fracture simpler and easier to parallelize. The only exception is the *delete set*, which is collected from all fractures and checked at the end of a lookup.

Another benefit of independent fractures is that each fracture can have different tuning parameters as long as the UPI files in the fracture share the same parameters. For example, the cutoff threshold C, the maximum number of pointers to store in a secondary index entry and even the size of one fracture can vary. We propose to dynamically tune these parameters by analyzing recent query workloads based on our cost models whenever the insert buffer is flushed to disk. This kind of adaptive database design is especially useful when the database application is just deployed and we have little idea about the query workload and data growth.

6.4.3 Merging Fractured UPI

Although fracturing UPIs avoids slowdown due to fragmentation, query performance still deteriorates over time as more and more fractures accumulate. The additional overhead to access the in-memory insert buffer is negligible, but accessing each fracture causes additional disk seeks. This overhead linearly increases for the number of fractures and can become significant over time.

Thus, we need to occasionally reorganize the Fractured UPI to remove fractures and merge them into the main UPI (this is similar to the way in which conventional indexes need reorganization or defragmentation to maintain their performance.) The merging process is essentially a parallel sort-merge operation. Each file is already sorted internally, so we open cursors on all fractures in parallel and keep picking the smallest key from amongst all cursors.

The cost of merging is about the same as the cost of sequentially reading all files and sequentially writing them back out, as we show in Section 6.7. As the size of the database grows, this merging process could take quite a long time, since it involves rewriting the entire database. One option is to only merge a few fractures at a time. Still, the DBA has to carefully decide how often to merge, trading off the merging cost with the expected query speedup. In Section 6.6, we show how our cost model can help estimate the overhead of fractures guide the decision as to when to merge.

6.5 Continuous UPI

In this section, we extend UPIs to handle attributes with continuous distributions (e.g., spatial attributes). For example, we might have imprecise GPS data for a position that is within a circle of 100m radius centered at (42°, 72°) with a uniform distribution. As the number of possible values in such distributions is infinite, we cannot apply the basic UPI presented above to such attributes.

Our solution is to build a primary index on top of R-Tree variants like PTIs [CXP$^+$04] and U-Trees [TCX$^+$05]. These indexes themselves are secondary indexes, and as such require additional seeks to retrieve tuples. We cannot make them primary indexes by simply storing tuples in the leaf nodes. As tuples are orders of magnitude larger than pointers, it would significantly reduce the maximum number of entries in a node, resulting in a deep and badly clustered R-Tree with high maintenance costs. Instead, we build a separate heap file structure that is synchronized with the underlying R-Tree nodes to minimize disk access. We cluster this separate heap file by the hierarchical location of corresponding nodes in the R-Tree.

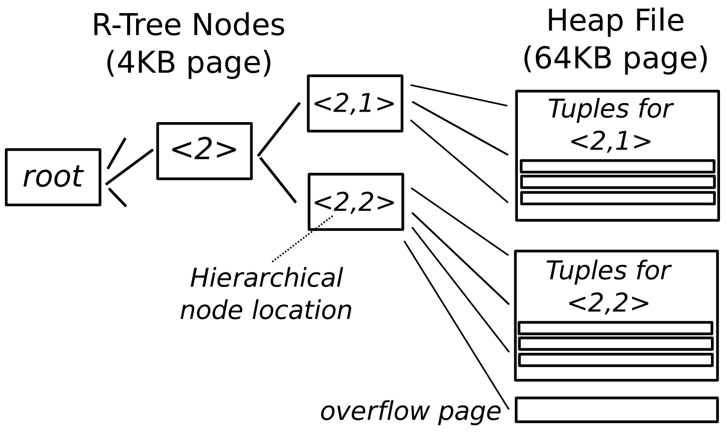

Figure 6.2: A Continuous UPI on top of R-Tree

Figure 6.2 shows a continuous UPI on top of an R-Tree. It consists of R-Tree nodes with small page sizes (e.g., 4KB) and heap pages with larger page size (e.g., 64KB). Each leaf node of the R-Tree is mapped to one heap page (or more than one when tuples for the leaf node do not fit into one heap page). Consider the 3rd entry in the R-Tree leaf node that is the 1st child of the 2nd child of the root node. We give this tuple the key <2, 1, 3> store it in the third position of heap page <2, 1>. When R-Tree nodes are merged or split, we merge and split heap pages accordingly. In this scheme, tuples in the same R-Tree leaf node reside in a single heap page and also neighboring R-Tree leaf nodes are mapped to neighboring heap pages, which achieves sequential access similar to a primary index as long as the R-Tree nodes are clustered well.

One interesting difference from prior work is that UPIs can exploit duplicated entries in the underlying R-Tree to speed up secondary index accesses as described in Section 6.3.2. Duplicating entries in an R-Tree (R+Tree) is also useful to reduce overlap of minimum bounding rectangles (MBRs) and improve clustering, which will lead to better query performance. PTIs and U-Trees are based on the R*Tree which does not duplicate entries although it tries to improve clustering by re-inserting entries. Developing an R+Tree analogue might further improve the performance of UPIs especially when wider and less skewed (e.g., Uniform) distributions cause too much MBR overlap. We leave this as future work.

6.6 Cost Models

In this section, we develop two cost models that capture the effects of the number of fractures and query thresholds on the query runtime respectively. When we need to account for both effects in one query, both estimates are added to estimate the total query runtime. The cost models are useful for the query optimizer to pick a query plan and for the database administrator to select tuning parameters such as the merging frequency and the cutoff threshold. We verify the accuracy of our cost models in Section 6.7 and observe that the cost models match the observed runtime quite well.

6.6.1 Parameters and Histograms

Table 6.6 shows the list of parameters used in our cost model as well as their values in our experimental environment. We get these parameters by running experiments (e.g., measure the elapsed time to open/close a table in Berkeley DB) and by collecting statistics (using, e.g., DB::stat()) for the particular configuration of interest.

Another input to our cost model is the *selectivity* of the query. Unlike deterministic databases, selectivity in our cost model means the fraction of a table that satisfies not only the given query predicates but also the probability threshold (QT). We estimate the selectivity by maintaining a probability histogram in addition to an attribute-value-based histogram. For example, a probability histogram might indicate that 5% of the possible values of attribute X have a probability of 20% or more. We estimate both the number of tuples satisfying the query that reside in the heap file and that reside in the cutoff index using the histograms. We also use the histogram to estimate the size of the table for a given cutoff threshold.

6.6.2 Cost Model for Fractured UPIs

We estimate the cost of a query on a Fractured UPI with the following equation. In addition to the sequential read cost, it counts the cost of table initialization and an index lookup for each fracture.

Table 6.6: Parameters for cost models

Parameter	Description	Typical Value
T_{seek}	Cost of one random disk seek	10 [ms]
T_{read}	Cost of sequential read	20 [ms/MB]
T_{write}	Cost of sequential write	50 [ms/MB]
H	Height of B+Tree	4
S_{table}	Size of *table*	10 [GB]
N_{leaf}	Count of leaf pages	S_{table} / 8KB
N_{frac}	Count of UPI fractures	10
$Cost_{init}$	Cost to open a DB file	100 [ms]
$Cost_{scan}$	Cost to full scan the table	$T_{read} \cdot S_{table}$

$$Cost_{frac} = Cost_{scan} \cdot Selectivity + N_{frac}(Cost_{init} + HT_{seek})$$

Based on this estimate and the speed of database size growth, a database administrator can schedule merging of UPIs to keep the required query performance. To estimate how long the merging will take, she can simply refer the cost to fully read and write all fractures; $Cost_{merge} = S_{table}(T_{read} + T_{write})$.

6.6.3 Cost Model for Cutoff Indexes

For a query whose probability threshold QT is less than the cutoff threshold C, we need to access the cutoff index, causing random seeks that are much more expensive than the sequential reads required to access the UPI itself. To confirm this, we ran Query 1 with various values for QT and C.

Figure 6.3 compares the runtime of a *non-selective* query over the *Author* table that could return as many as 37,000 authors and a *selective* query which returns as many as 300 authors. In both cases, the query performs slower for lower QT especially when $QT < C$ because the query has to access the cutoff index as expected. When $QT \geq C$, the query is very fast because it is answered purely through sequential I/O.

However, the runtime of the non-selective query is the same for all QT when $C > 0.4$. This result is not intuitive because the number of pointers read from the cutoff index should be larger for smaller values QT. In fact, $QT = 0.05$ retrieves 22,000 pointers from the cutoff index while $QT = 0.25$ retrieves 3,000, but the query runtime is the same.

This happens because in both cases we access nearly every page in the table. We call this

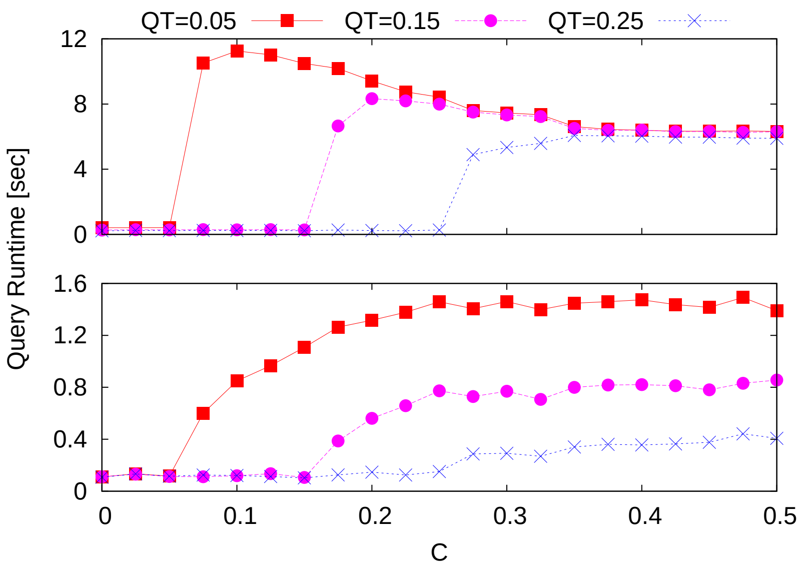

Figure 6.3: Cutoff Index Real Runtime. Non-selective (top) and Selective (bottom) queries.

case *saturation*. As the query needs to retrieve thousands of pointers from the cutoff index, these pointers already cover almost all of the heap file, and the disk access pattern degenerates to a full table scan (assuming the database performs a heap file lookup by ordering the pointers relatively to the positions in the heap file). At this point, further increasing the number of pointers (smaller QT) does not make the query slower. Another interesting observation is that, as demonstrated in the $QT = 0.05$ curve, a query might perform faster with larger C when pointers are saturated because the full table scan cost is smaller.

These observations suggest that query runtime is not simply the number of retrieved pointers multiplied by the disk seek cost, especially when the number is large. Instead, the growth of the number of real disk seeks gradually decreases for more pointers because more and more pointers will land on the same blocks and eventually get saturated. Our main target is non-selective analytical queries, so ignoring this effect can cause a huge error in query cost estimation.

In order to model this saturation behavior, we use a generalized logistic function $f(x)$, which is a type of *sigmoid* function. A sigmoid function is often used to model phenomena like population growth where the rate of reproduction is proportional to the amount of available resource which decreases as population increases. This is consistent with our notion of saturation.

$$\begin{aligned} Cost_{cut} &= Cost_{scan} \cdot Selectivity + 2(Cost_{init} + HT_{seek}) \\ &\quad + f(\#Pointers) \\ f(x) &= Cost_{scan}\left(\frac{1-e^{-kx}}{1+e^{-kx}}\right) \end{aligned}$$

The first line is basically the same as $Cost_{frac}$ except that we access two tables (the UPI Heap File and the Cutoff Index). $f(x)$ is the cost to retrieve tuples from the heap file which satisfies $f(0) = 0$ and $f(\infty) = Cost_{scan}$. k is a parameter that represents how quickly we reach saturation. We determine this value by applying a heuristic $f(0.05 \cdot N_{leaf}) = 0.99 \cdot Cost_{scan}$, which is based on experimental evidence gathered through our experience with UPIs.

We propose to use the cost models for selecting the cutoff threshold as follows: First, an administrator collects query workloads of the database to analyze the frequency of queries to have low QTs. Second, she figures out the acceptable size of her database given available disk capacity and expected maintenance time. Finally, she picks a value of C that yields acceptable database size and also achieves a tolerable average (or nth percentile) query runtime.

6.7 Experimental Results

In this section, we evaluate the query and maintenance performance of UPIs as well as the accuracy of our cost models. We implemented a prototype UPI for both discrete and continuous distributions and compared the performance with prior uncertain indexes on two real datasets.

6.7.1 Setup

All of our UPI implementations are in C++ on top of BDB (BerkeleyDB) 4.7 except the continuous UPI because BDB does not support R-Tree indexes. Instead, we implemented a custom heap file layer (See Section 6.5) on top of the U-Tree provided by Tao et al [TAO] which pre-computes integrals of probability distributions as MBRs. For other experiments, we used BDB's B+Trees. We always sort pointers in heap order before accessing heap files similarly to PostgreSQL's bitmap index scan to reduce disk seek costs caused by secondary index accesses. Our machine for all experiments runs Fedora Core 11 and is equipped with a quad core CPU, 4GB RAM and 10k RPM hard drive. All results are the average of 3 runs, and were performed with a cold database and buffer cache.

DBLP Dataset and Query: Our first dataset is derived from DBLP [Ley09], the free bibliographic database of computer science publications. DBLP records more than 1.3 million publications and 700k authors. This dataset exemplifies uncertainty as a result of data integration. DBLP itself has no uncertainty but by integrating DBLP with other data sources, we can produce uncertain data. For instance, the affiliated institution of each author is useful information for analysis, but is not included in DBLP. SwetoDblp [AMHBAS07] supplies it by integrating DBLP with ontology databases. Nagy et al [NFJ09] applied machine learning techniques to automatically derive affiliation information by querying a web search engine and then analyzing the homepages returned by that search engine.

Such analysis is useful but inherently imprecise, so the resulting affiliation information is uncertain. We generated such uncertain affiliations by querying all author names in DBLP via Google API and assigning probabilities to the returned institutions (determined by domain names) up to ten per author. We used a zipfian distribution to weigh the search ranking and sum the probabilities if an institution appears at more than one ranks for the author.

The resulting data is the *Author* table exemplified in Table 6.4 which has uncertain attributes like institution and country for all 700k authors. We also added the same uncertain attributes into the list of publications (assuming the last author represents the paper's affiliation) and stored it as the *Publication* table which contains information about 1.3M publications.

We loaded the uncertain data into BDB and built a UPI on the *Institution* attribute with various cutoff thresholds. For the Publication table, we also built a secondary index on Country, which is correlated with Institution. We then experimented with the following queries on the two tables.

<u>Query 1: Author Extraction</u>

SELECT * FROM Author WHERE Institution=MIT

<u>Query 2: Publication Aggregate on Institution</u>

SELECT Journal, COUNT(*) FROM Publication
WHERE Institution=MIT GROUP BY Journal

<u>Query 3: Publication Aggregate on Country</u>

SELECT Journal, COUNT(*) FROM Publication

WHERE Country=Japan GROUP BY Journal

Cartel Dataset and Query: Our second dataset is derived from Cartel (`http://cartel.csail.mit.edu`) data. Cartel is a mobile sensor network system which collects and analyzes GPS data sent from cars to visualize traffic. During the analysis, the raw GPS data is converted into car observations which contain the location, estimated speed, road segment and the direction of cars. Because of the imperfect accuracy of GPS and probabilistic analysis, the resulting car observations are uncertain.

We generated uncertain Cartel data based on one year of GPS data (15M readings) collected around Boston. We assigned a constrained Gaussian distribution to location with a boundary to limit the distribution as done in [TCX+05] and added an uncertain road segment attribute based on the location. We built our 2-D continuous UPI on the uncertain location attribute (i.e., longitude/latitude) and also built a secondary index on the road segment attribute. We then experimented with the following queries.

Query 4: Cartel Location

SELECT * FROM CarObservation

WHERE Distance(location, 41.2°, 70.1°) $\leq$ Radius

Query 5: Cartel Road Segment

SELECT * FROM CarObservation WHERE Segment=123

6.7.2 Results

UPI on Discrete Distributions: We now present our experimental results, starting with DBLP. The DBLP dataset has discrete distributions on several attributes, therefore, we compare our UPI with our implementation of PII [SMP+07] on an unclustered heap file. PII is an uncertain index based on an inverted index which orders inverted entries by their probability. We compared UPI with PII because PII has been shown to perform fast for discrete distributions [SMP+07].

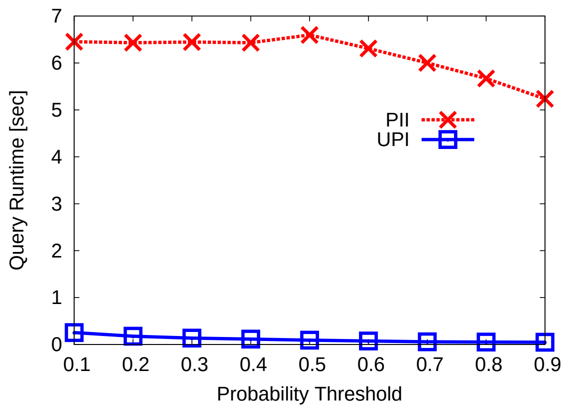

Figure 6.4: Query 1 Runtime

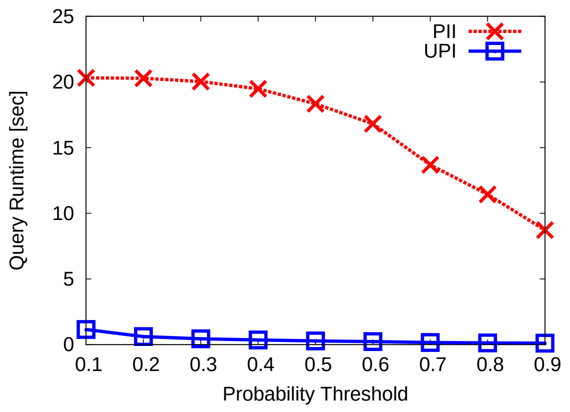

Figure 6.5: Query 2 Runtime

Figure 6.4 and Figure 6.5 show the runtimes of Query 1 and Query 2, comparing UPIs ($C = 10\%$) and PIIs on Institution. Both indexes perform faster with higher thresholds as they retrieve less data, but the UPI performs 20 to 100 times faster because the UPI sequentially retrieves tuples from the heap file while PII needs to do random disk seeks for each entry.

Figure 6.6 shows the runtime of Query 3 which uses a secondary index on Country. This time, we also test the UPI with and without tailored secondary index access as described in Section 6.3.2. Although both use secondary indexes in this case, our index performs faster because of correlation between the attributes of the primary and secondary indexes. However, the UPI without tailored

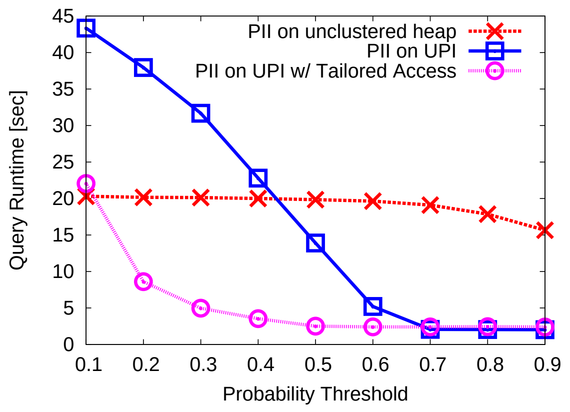

Figure 6.6: Query 3 Runtime

index access is not very beneficial, and sometimes is even slower than the unclustered case because it cannot capture the possible overlap of pointers from the secondary index. Our tailored index access performs up to a factor of 7 faster than the UPI without tailored access, and up to a factor of 8 faster than PII.

UPI on Continuous Distributions:

Next, we compare a continuous UPI with a secondary U-Tree on the Cartel dataset. Figure 6.7 shows the performance comparison between a 2-D continuous UPI and a U-Tree on Query 4. We fixed $QT = 50\%$ and varied the radius. The continuous UPI performs faster by a factor of 50 to 60 because the tuples in the UPI heap file are well clustered with the intermediate nodes. Figure 6.8 shows the runtime of Query 5, varying QT ($QT = 90\%$ returns no result). Both techniques use secondary indexes for this query. However, as in the discrete case, the secondary index performs much faster with a continuous UPI because of correlation between lat/long (primary index) and segment ID (secondary index) which reduces the number of disk seeks by orders of magnitude. The speed up is a factor of up to 180 when $QT < 50\%$. For queries $QT > 50\%$ (more selective queries) which have many fewer pointers to follow, heap access on both indexes are much faster so the performance gap is less because secondary index access cost is the same. However, the gap is still more than a factor of 50.

Fractured UPIs: We now evaluate maintenance of UPIs. To measure the maintenance cost, we randomly delete 1% of the tuples from the DBLP Author table and randomly insert new tuples equal

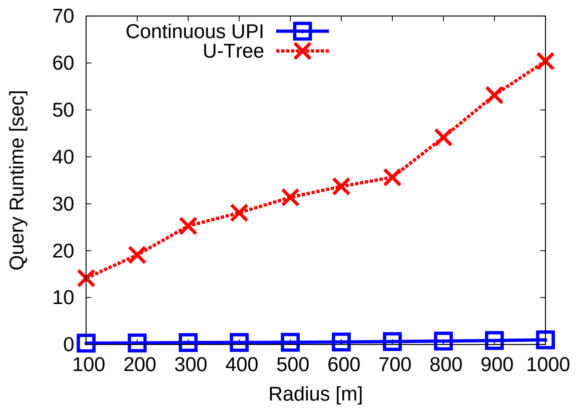

Figure 6.7: Query 4 Runtime

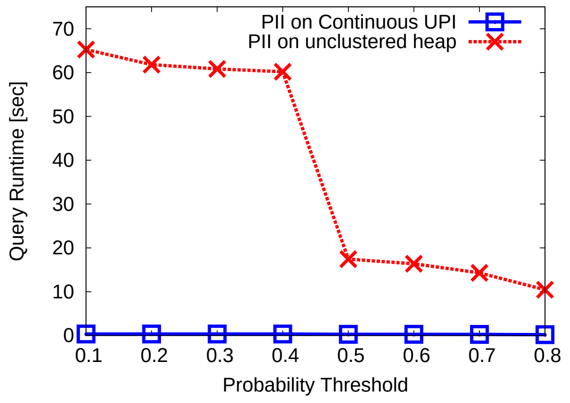

Figure 6.8: Query 5 Runtime

to 10% of the existing tuples. We compare an unclustered table (clustered by an auto-increment sequence), a UPI and a Fractured UPI. For the Fractured UPI, we drop the insert buffer after all insertions and deletions.

As shown in Table 6.7, the non-fractured UPI performs quite poorly for both insertions and deletions because random locations in the B+Tree are read, written, split and merged. Unclustered and Fractured UPIs perform well because they sequentially write the inserted data to disk. Note that for deletions, even an unclustered table performs poorly because tuples are deleted from random places. The Fractured UPI performs much faster because it simply buffers TupleIDs of deleted tuples

Table 6.7: Maintenance Cost

	Insert	Delete
Unclustered	7.8 sec	75 sec
UPI	650 sec	212 sec
Fractured UPI	4.0 sec	0.03 sec

and sequentially writes them to disk as a batch.

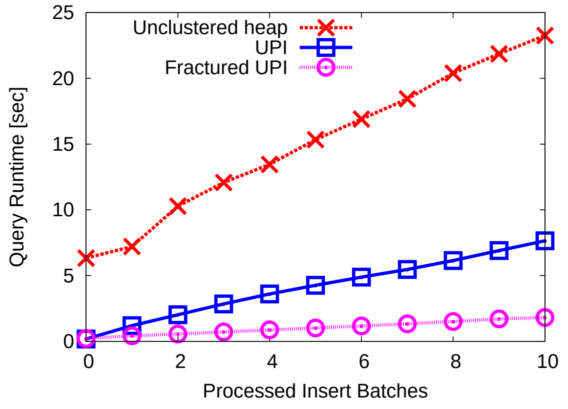

Figure 6.9: Q1 (C=QT=0.1) Deterioration

We also tested the query performance deterioration after a number of insert batches, each of which consists of the 10% insertions and 1% deletions (as before). For the Fractured UPI, we made one fracture after each insert batch. Figure 6.9 shows the query runtime deterioration. After 10 insert batches, the table size is increased by only 90% (=10*(10%-1%)), but all three approaches show much more than 90% deterioration. The unclustered table becomes 4 times slower compared with the initial state, the non-fractured UPI is 40 times slower and the Fractured UPI is 9 times slower. For the unclustered table and the UPI, the slowdown is because of fragmentation caused by deletion and (for UPI) insertion.

This result illustrates that the Fractured UPI improves not only the maintenance cost but the query performance by eliminating fragmentation. Still, the Fractured UPI does gradually slow down because of the overhead of querying each fracture.

Cost Models: To restore the query performance of the Fractured UPI, we implemented merging of fractures and compared that with our cost model for fractures described in Section 6.6.2.

Figure 6.10 shows the real and estimated query runtime during 30 insert batches.

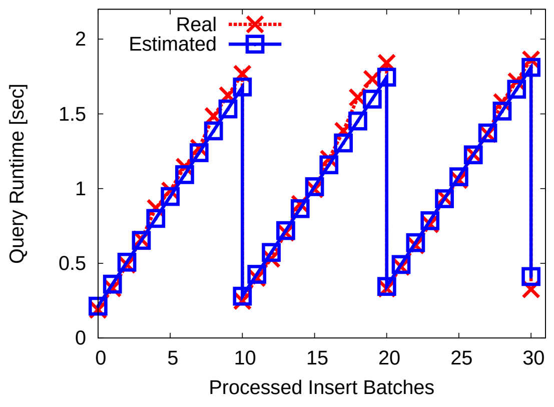

Figure 6.10: Fractured UPI Runtime

Table 6.8: Merging Cost

#	Time	DB size
1	150 sec	2.5 GB
2	247 sec	3.6 GB
3	275 sec	4.8 GB

We merged fractures after every 10 insert batches. The query performance is restored after each merging, and the estimated runtime matches the real runtimes quite well. Table 6.8 shows the cost of three merges. As the result shows, the merge cost is almost the same as reading and writing the entire table in BDB (20+50 [ms/MB]) and conforms to our cost model.

Finally, we test the query runtime when a UPI has to access a cutoff index, and we verify that our cost model can predict the behavior. We again used Query 1 and varied both QT and C. First, we checked the accuracy of selectivity estimation described in Section 6.6.1 because our cost model relies on accurate estimates of the number of pointers returned from the cutoff index. Figure 6.11 compares the true number and the estimated number of cutoff pointers for various QT and C settings (except $QT > C$). The result shows that our selectivity estimation is accurate.

Figure 6.12 shows the runtimes estimated by our cost model with the exact same setting as Figure 6.3 in Section 6.6.3. As the two figures show, our cost model (which estimates disk seek costs and saturation of cutoff pointers using a sigmoid function) matches the real runtime very well for both selective and non-selective queries.

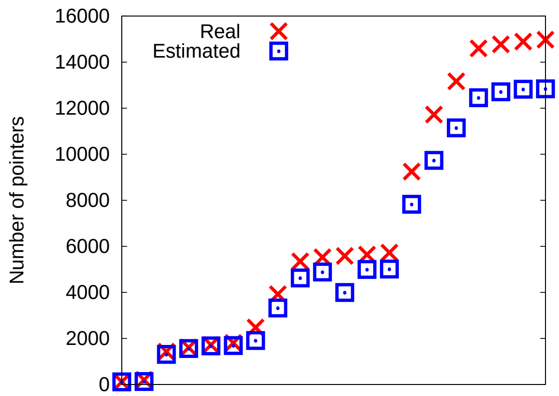

Figure 6.11: #Cutoff-Pointers Estimation

These results above confirm that our cost models can accurately estimate the query costs in various settings. These cost models will be useful for the query optimizer to choose execution plans and for a database administrator or auto tuning program to choose tuning parameters for UPIs.

6.8 Related Work

The most closely related work to UPIs relates has to do with the use of indices for uncertain data. Some work [BSHW06] uses traditional B+Trees to index uncertain data. Other work has shown that a special index can substantially speed up queries over uncertain data. For example, Cheng et al [CXP+04] developed the PTI (*Probabilistic Threshold Indexing*) based on R-Trees to speed up PTQs on uncertain attributes with one dimensional continuous distributions. Other research has extended these ideas to higher dimensions (*U-Trees* [TCX+05]) and more variable queries (*UI-Trees* [ZLZ+55]). Similarly, Singh et al [SMP+07] proposed the PII (*Probabilistic Inverted Index*) for PTQs on uncertain attributes with discrete distributions based on inverted indexes as well as the PDR-tree (*Probabilistic Distribution R-tree*) based on R-Trees.

Although these indexes successfully speed up query execution in some cases, they are essentially secondary indexes and can lead to many random disk seeks when the query needs to retrieve other attributes from the heap file. This problem arises especially when the query is not selective as shown in Section 6.7. Hence, UPIs complement this prior work by adding support for primary indexes on

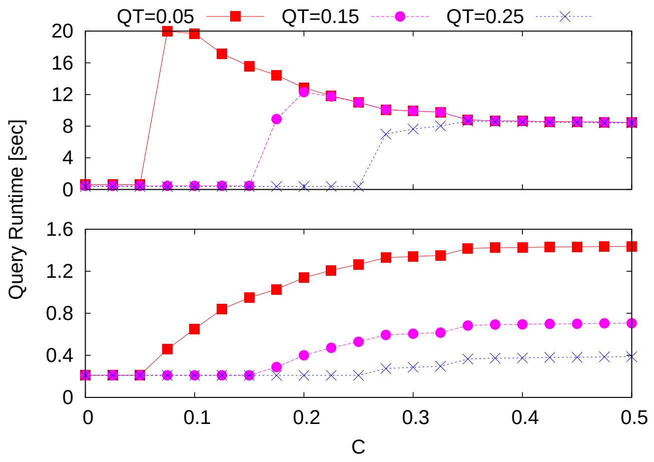

Figure 6.12: Cutoff Index Cost Model

uncertain attributes, which are particularly useful for analytical PTQs which process thousands or millions of tuples.

6.9 Conclusion

In this chapter, we developed a new primary index for uncertain databases called a UPI. Our empirical results on both discrete and continuous uncertain datasets show that UPIs can perform orders of magnitude faster than prior (secondary) indexing techniques for analytic queries on large, uncertain databases. We proposed several techniques to improve the performance of UPIs, including cutoff indexes to reduce their size, and tailored indexes to improve the performance of secondary indexes built on top of UPIs. We also discussed Fractured UPIs that help handle data updates and eliminate fragmentation, further improving query performance. Finally, we provide accurate cost models to help the query optimizer to choose execution plans and the DBA to select tuning parameters.

6.9.1 Future Work

As future work, we plan to apply UPIs for queries other than PTQs, especially *Top-k*. Top-k, or k-NN queries, in uncertain databases need a careful handling in its semantics and optimization.

Because simply applying the existing top-k query semantics would ignore one of the two orthogonal metrics, values and probabilities. Ilyas et al suggested a few query semantics and a query processing engine to determine probabilistic top-k answers. They devise optimization techniques to minimize the number of tuples extracted from a Tuple Access Layer (TAL) which provides tuples in probability order [SIC07]. However, they assumed the cost of one TAL access is the same regardless of the underlying clustered and secondary indexes, which we observed not true throughout this thesis. Evaluation and optimization of the underlying indexes for uncertain top-k queries is yet to be studied.

We expect that a UPI can work as an efficient TAL if it is well correlated with the attribute ranked by top-k. One approach is to estimate the minimum probability of tuples required to answer the top-k query and use this probability as a threshold for the UPI. Another approach is to access UPI a few times with decreasing probability thresholds until the answer is produced. Both approaches are promising future work.

Chapter 7

Conclusions

We have studied techniques to improve the performance and maintainability of large analytic databases, issues being increasingly significant due to the rapidly growing amount of data in individual repositories and on the internet. Our key discovery is that exploiting correlation in the databases significantly contributes to these goals.

7.1 Potentials of Correlation-Awareness

In Chapter 2, we found that database indexes can be made orders of magnitude faster and also smaller by exploiting their correlation with clustering of the table. Using this finding, we developed a new data structure, Correlation Map (CM), which stores correlations as compact secondary indexes.

In Chapter 3, we designed and implemented a physical database design tool, CORADD, that enhances correlations between indexes to design a faster and more maintainable database. We demonstrated that CORADD speeds up the entire query workload by up to a factor of 6 compared to a state-of-the-art design tool. Our experimental results confirmed that correlation-awareness in indexing, query optimization, and physical database designs has significant impacts on analytic databases. We then extended the potential of exploiting correlations in various settings of current and increasing importance.

7.2 Optimizing Index Deployment Order

To exploit the potential of correlations fully, we often need to deploy many clustered and secondary indexes. In Chapter 4, we classified the challenges in optimizing deployment of a large number of indexes. The primary factors about index deployment in such an environment are how long deployment will take and how promptly users can observe the query speed-ups.

We observed that traditional optimization methods such as mixed integer programming (MIP) cannot efficiently solve the problem because it is highly non-linear. We thus formulated and solved the problem as a constraint programming instance with our pruning techniques; this dramatically shrinks the search space by exploiting the combinatorial properties of the problem.

7.3 Extensions to Distributed Systems

In Chapter 5, we found that flexible partitioning is essential in applying the idea of correlations to shared-nothing distributed file systems such as HDFS. Without partitioning that is well correlated with the aggregating key of the query, such a distributed system must repartition and transmit a large amount of data during query execution.

Our approach is to utilize the redundancy in the distributed file systems to deploy multiple partitionings of the data. The key ideas in this work were to reduce the risk of permanent data loss due to heterogeneous partitioning and to find the best balance between query performance, space consumption, and the recoverability by an analytic model on recoverability.

7.4 Extensions to Uncertain Databases

In Chapter 6, we demonstrated a new primary index data structure, UPI, for uncertain databases. The key challenge in this work was that uncertain attributes may have several possible values. If we cluster the heap file based on only one of the possible values (e.g., the value with the largest probability), the clustered index cannot answer the probabilistic queries that are necessary to probe all possible values.

UPI addresses the problem by duplicating tuples for possible values beyond a certain probabilistic threshold, balancing the query performance and the index size. It supports clustering on attributes with both discrete and continuous uncertainty distributions. Our empirical results confirmed that

UPI dramatically speeds up query execution that predicates or aggregates on the primary attribute and also enables secondary indexes to exploit correlations with the clustering.

7.5 Future Work

Here, we discuss a few open problems that were not addressed in individual chapters.

First, in order to take the effects of correlations into account, query cost models in a DBMS require reasonably accurate correlation statistics. Dynamically maintaining such statistics at runtime might incur too much overheads for data update. In fact, this concern was raised by the developers of a popular commercial DBMS when we discussed with them incorporating our work in Chapters 2 and 3 into their DBMS. An efficient yet accurate method to maintain correlation statistics is a part of future work.

Second, as mentioned at the end of Chapter 4, a method to jointly consider the set *and* order of indexes to deploy needs more research. Such an integrated approach has more applicability in the distributed analytics setting discussed in Chapter 5. In so-called big-data analytics, the data are often unstructured or semi-structured and even the user has no idea of its data scheme. Hence, the query workloads and even the logical scheme can change quickly and significantly over time. Furthermore, the *order* of changing the physical scheme (e.g., partitioning and sorting of each replica) becomes even more important and more challenging to make sure the recoverability of the entire data store satisfies requirements. Developing an automated design tool to take these issues into account is a promising future step in realizing an efficient, scalable, and reliable distributed analytics platform.

Appendix A

Correlation Aware Database Design

A.1 Database Size and Maintenance Cost

In this section, we demonstrate how the size of the database is directly linked to its maintenance costs. To illustrate this, we ran an experiment where we inserted 500k tuples into the SSB *lineorder* table while varying the total size of additional database objects (e.g., MVs) in the system (see Section 3.7 for our experimental setup). The results are shown in Figure A.1; as the size of the

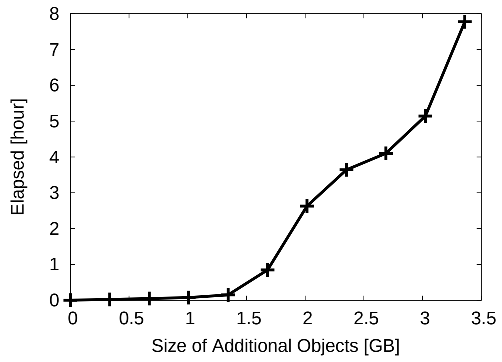

Figure A.1: Cost of 500k insertions.

materialized MVs grows, the cost of 500k insertions grows rapidly. With 3 GB worth of additional

MVs, the time to perform the insertions is 67 times slower than with 1 GB of additional MVs. The reason why maintenance performance deteriorates given more objects is that additional objects cause more dirty pages to enter the buffer pool for the same number of INSERTs, leading to more evictions and subsequent page writes to disk. The *lineorder* table has 2 GB of data while the machine has 4 GB RAM; creating 3 GB of MVs leads to significantly more page writes than 1 GB of MVs.

In Chapter 2, we observed similar deterioration in update performance as we added more B+Tree indexes while adding more CMs had almost no effects because of their small sizes.

Therefore, despite the decreasing price per gigabyte of storage, space budgets remain an important parameter of database design tools so that data warehousing workloads can be completed within available time limits (e.g., 100k insertions between 12 am and 3 am).

A.2 Proof Sketch of Termination of Selectivity Propagation

In this section, we give a sketch of proof that *Selectivity Propagation* described in Section 3.4.1 always terminates in finite time.

Let a *step* be the process of calculating propagated selectivities for all attributes and updating the selectivity vector. At each step, each attribute can be updated by a single *parent* which gives the minimum selectivity after propagation. The parent could have been updated by its parent (grand parent), but there can not be any cycle in the update path because the strength of functional dependency is always less than one. Thus, the maximum length of an update path is $|A|$ where A is the set of all attributes. Therefore, selectivity propagation terminates at most after $|A|$ steps and each step takes $O(|A|^2)$, resulting in $O(|A|^3)$ computation cost.

The proof becomes more complex and runtime becomes larger when considering composite functional dependencies (i.e., $AB \rightarrow C$) but the main concept above stays the same.

Appendix B

Index Deployment Order

B.1 Source Code, Datasets

All of our source code and experimental data can be accessed on our web site (http://*now preparing*). This includes Java projects, CPlex/COMET models, and problem data files.

Our purpose is two-fold. First, we would like to ensure the reproducibility of our experiments. Second, we expect this problem will be useful for testing various solver technologies and we want to make it available to the operation research community.

B.2 Full MIP Model

This section provides the detailed MIP model for the ordering problem. The model uses the input data described in Table 4.2 and defines additional constants and variables in Table B.1. Variables annotated with a hat have a slightly different semantics than those in the CP model, but their meaning is roughly the same. The biggest decision variable change is that the B variables are used to determine the orders of indexes.

The index order problem can be formulated as a MIP as follows,

Table B.1: Additional Symbols & Variables

$d \in D$	A discretized timestep. $D = \{1, 2, \ldots, \lvert D\rvert\}$
$A_i \in D$	Timestep to start building index i.
$B_{i,j} \in \{0,1\}$	Whether index i precedes index j.
$\hat{C}_i$	Cost to create index $\boldsymbol{i}$.
$\hat{X}_{q,d}$	q's **runtime** (*not speed-up*) at time d.
$\hat{Y}_{q,p,d} \in \{0,1\}$	Whether p is **used** (*not only available*) for q at d.
$\hat{Z}_{i,d} \in \{0,1\}$	Whether i available at time d.
$CY_{i,j} \in \{0,1\}$	Whether j is utilized to create i.

Table B.2: Greedy Solutions vs. 100 Random Permutations. (TPC-DS is 400 times larger in scale.)

Dataset	Greedy	Random (AVG)	Random (MIN)
TPC-H	47.9	65.5	51.5
TPC-DS	65.9	74.1	69.6

$$\text{Objective:} \quad min \sum_d \left(\sum_q \hat{X}_{q,d} \right) \tag{B.1}$$

$$\text{Subject to:} \quad B_{i,j} + B_{j,i} = 1 : \forall i \neq j \tag{B.2}$$

$$B_{i,k} \leq B_{i,j} + B_{j,k} : \forall i \neq j \neq k \tag{B.3}$$

$$B_{i,j} \leq 1 - \frac{A_i + \hat{C}_i - A_j}{|D|} : \forall i \neq j \tag{B.4}$$

$$\sum_p \hat{Y}_{q,p,d} = 1 : \forall q, d \tag{B.5}$$

$$\hat{Y}_{q,p,d} \leq Z_{i,d} : \forall q, p \in plans(q), d, i \in p \tag{B.6}$$

$$\hat{X}_{q,d} = \sum_{p \in plans(q)} (qtime(q) - qspdup(p,q))\hat{Y}_{q,p,d} : \forall q, d \tag{B.7}$$

$$Z_{i,d} \leq 1 - \frac{A_i + \hat{C}_i - d}{|D|} : \forall i, d \tag{B.8}$$

$$\sum_j CY_{i,j} \leq 1 : \forall i \tag{B.9}$$

$$CY_{i,j} \leq B_{j,i} : \forall i, j \tag{B.10}$$

$$\hat{C}_i = ctime(i) - \sum_{j \in I} (cspdup(i,j) CY_{i,j}) : \forall i \tag{B.11}$$

(B.2) assures either i precedes j or j precedes i. (B.3) assures the index order preserves transitivity; i cannot precede k if j precedes i and k precedes j. The A variables determine when each index is made. (B.4) means that, when i precedes j, A_i has to be C_i (cost to create i) smaller than

```
Inputs  : Index set $I$. Query set $Q$.
Outputs: Ordered list of indexes $N$.
$N = []$;
while $I$ is not empty do
    bestDensity = 0;
    bestIndex = null;
    foreach $i \in I$ do
        benefit = 0;
        foreach $q \in Q$ do
            previous = $q$.getRuntime($N$);
            next = $q$.getRuntime($N \cup i$);
            benefit += previous - next;
            // Add remaining interactions to benefit
            foreach $p \in plans(q) : i \in p$ do
                interaction = next - $q$.getRuntime($p$);
                if interaction $> 0$ and $p \setminus N \neq \phi$ then
                    benefit += interaction / $|p \setminus N|$;
                end
            end
        end
        density = benefit / $i$.getBuildCost($N$);
        if bestIndex = null or density > bestDensity then
            bestDensity = density;
            bestIndex = $i$;
        end
    end
    $N$.append(bestIndex);
    $I = I \setminus$ bestIndex;
end
return $N$;
```

Algorithm 5: Interaction Guided Greedy Algorithm

A_j. $A_i + \hat{C}_i - A_j$ is divided by $|D|$ to normalizes the expression to a range between 0 and 1.

The Y variables determine whether the plan is **used** for each query at d. Therefore, the sum of Y is always 1 (B.5). There is always an empty-plan $\{\emptyset\}$ which gives no speed-up to ensure feasibility of (B.5). (B.6) assures the plan is available only when all indexes in the plan are available. Then, (B.7) calculates the runtime of each query from Y.

As constraints (B.4-B.6) calculate the query performance at a given time, constraints (B.8-B.11) calculate the query build cost at a given time. B.8 determines whether each index is available at each time step by checking A and C. (B.9) and (B.10) are equivalent to the constraints on Y except the interaction to build index is always pair-wise. (B.11) calculates the time to create each index from them.

We also add the additional constraints developed in Section 4.4 by posting constraints on A and B (e.g., $i_3 < i_5$ yields, $B_{3,5} = 1$).

The objective is simply the sum of X for all time steps, because we discretized the time steps uniformly. We also add an imaginary query plan which requires all the indexes and makes the runtimes of all queries zero. This ensures the objective value is 0 for time steps that remain after all the queries are built.

This MIP model correctly solves the ordering problem but introduces many constraints and variables (it requires more than 1 million variables for large problems) due to non-linear properties of the problem. Because of this, MIP solvers cannot find a feasible solution after several hours when solving large problems.

B.3 Greedy Algorithm

Algorithm 5 provides the full greedy algorithm described in Section 4.6.4. We developed this algorithm to provide good initial solutions to our local search methods. Table B.2 shows that the objective value of the greedy solutions are always better than average and minimum values of 100 random permutations of indexes.

B.4 Full Problem Properties

This section provides formal proofs and detection algorithms for the problem properties discussed in Section 4.4 as well as a detailed analysis of the pruning-power.

B.4.1 Proof Preparation

Notations: Let N denote a complete sequence of indexes $I = \{i_1, i_2, \ldots, i_n\}$, e.g., $N = i_1 \to i_2 \to i_3$. Let L denote a *subsequence*, which is an order of a subset of the indexes, e.g., $L_1 = i_1 \to i_2$, or $L_2 = i_3$. Let M denote an unordered set of indexes, e.g., $M_1 = \{i_1, i_2\}$ and let $\{L\}$ denote the unordered set of indexes in L.

Let $C(i, M)$ be the build cost of index i when indexes in M are already built. Let $C(L, M)$ be the total cost of building the indexes of L in the order L specifies. As an abbreviation, we will use $C(i) \equiv C(i, \emptyset)$, e.g., $L_1 = i_1 \to i_2$ and $C(L_1) = C(i_1) + C(i_2, \{i_1\})$. Let $S(i, M)$ be the query speed-up of building i assuming the indexes of M are already built. We will also use the $S(i) \equiv S(i, \emptyset)$ abbreviation. Because the eventual speed-up achieved by the indexes does not depend on the order of indexes, the first parameter of S can be a *set* of indexes unlike C.

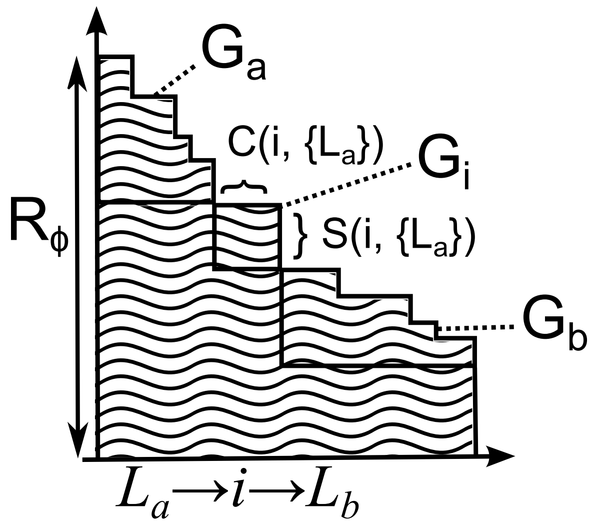

Figure B.1: Notations

Let G_i be the basic area of index i. Trivially, $G_i = S(i, \ldots)C(i, \ldots)$. To simplify the notation, let us extend G to subsequences as illustrated in Figure B.1. Note that the second parameter of both C and S is the *set* of indexes built before. All indexes built after have no effect on the value of C and S.

Finally, let R_M be the total query runtime when indexes in M exist, namely $R_M = R_\emptyset - S(M)$. For example, the total objective area (shaded area) in Figure B.1 is

$$\begin{aligned} Obj(L_a \to i \to L_b) &= G_a + R_{L_a} C(L_a) \\ &\quad + G_i + R_{L_a+i} C(i, \{L_a\}) \\ &\quad + G_b + R_{L_a+i+L_b} C(L_b, \{L_a + i\}) \end{aligned}$$

<u>The Swap Property:</u> Here we discuss a useful building block for the other proofs in this section. Consider the objective values of solutions $N = L_a \to L_i \to L_j \to L_b$ and $N' = L_a \to L_j \to L_i \to L_b$ which are identical except for the swap of L_i and L_j.

$$\begin{aligned} Obj(N) &= G_a + R_{L_a} C(L_a) \\ &\quad + G_i + R_{L_a+L_i} C(L_i, \{L_a\}) \\ &\quad + G_j + R_{L_a+L_i+L_j} C(L_j, \{L_a + L_i\}) \\ &\quad + G_b + R_{L_a+L_i+L_j+L_b} C(L_b, \{L_a + L_i + L_j\}) \\ Obj(N') &= G'_a + R_{L_a} C(L_a) \\ &\quad + G'_j + R_{L_a+L_j} C(L_j, \{L_a\}) \\ &\quad + G'_i + R_{L_a+L_j+L_i} C(L_i, \{L_a + L_j\}) \\ &\quad + G'_b + R_{L_a+L_j+L_i+L_b} C(L_b, \{L_a + L_j + L_i\}) \end{aligned}$$

Because L_a precedes both L_i and L_j, $G_a = G'_a$. Additionally, because both query and build time interactions depend only on the *set* of indexes built before, L_b receives exactly the same interaction from indexes in L_i and L_j. Therefore, $G_b = G'_b$. Hence, we see

$$\begin{aligned} & Obj(N) - Obj(N') = (G_i - G'_i) + (G_j - G'_j) \\ & + R_{L_a+L_i} C(L_i, \{L_a\}) - R_{L_a+L_j} C(L_j, \{L_a\}) \\ & + R_{L_a+L_i+L_j} (C(L_j, \{L_a + L_i\}) - C(L_i, \{L_a + L_j\})) \end{aligned} \tag{B.12}$$

Also, consider the case when a swap occurs around an interior order, e.g. $N = L_a \to L_i \to L_j \to L_k \to L_b$ to $N' = L_a \to L_k \to L_j \to L_i \to L_b$, where L_i and L_k are swapped and L_j remains in the middle. By the same argument we can deduce,

$$\begin{aligned} Obj(N) - Obj(N') = (G_i - G'_i) + (G_j - G'_j) + (G_k - G'_k) \\ +R_{L_a+L_i}C(L_i, \{L_a\}) - R_{L_a+L_k}C(L_k, \{L_a\}) \\ +R_{L_a+L_i+L_j}C(L_j, \{L_a + L_i\}) \\ -R_{L_a+L_k+L_j}C(L_j, \{L_a + L_k\}) \\ +R_{L_a+L_i+L_j+L_k}(C(L_k, \{L_a + L_i + L_j\} \\ -C(L_i, \{L_a + L_k + L_j\}) \end{aligned} \tag{B.13}$$

B.4.2 Alliances

Definition: Allied indexes are a set of indexes that only appear in query plans as a complete group and have no external interactions for building cost improvements.

Theorem 1: Every problem has at least one optimal solution [1] in which allied indexes are built consecutively.

Proof. Let i be the first created index among some allied indexes. Suppose a solution N in which there is a non-empty sub sequence L_b between i and its allied indexes, namely $N = L_a \to i \to L_j \to L_b$ where L_b contains the allied index of i. Now, consider an altered solution $N' = L_a \to L_j \to i \to L_b$. We will prove the objective of N' is always smaller or the same as that of N.

Because i requires the allied indexes contained in L_b to speed up any query, $G_i = G'_i = 0$ and $R_{L_a+i} = R_{L_a}, R_{L_a+i+L_j} = R_{L_a+L_j}$. By definition i has no interactions that speed up building any index in L_j, therefore $G_j = G'_j$, and $C(L_j, \{L_a + L_i\}) = C(L_j, \{L_a\})$. Because $R_{L_a+L_j} \leq R_{L_a}, C(L_i, \{L_a\}) \geq C(L_i, \{L_a + L_j\})$, from (B.12),

$$\begin{aligned} & Obj(N) - Obj(N') \\ = \quad & R_{L_a}C(L_i, \{L_a\}) - R_{L_a+L_j}C(L_i, \{L_a + L_j\}) \\ \geq \quad & R_{L_a+L_j}(C(L_i, \{L_a\}) - C(L_i, \{L_a + L_j\})) \geq 0 \end{aligned}$$

[1] If there are not multiple optimal solutions (*tie*), each theorem simply means "every optimal solution should ...".

Thus, a solution that does not create allied indexes consecutively can be improved by swapping so that the allied indexes come closer. By induction on the swapping of indexes an optimal solution can always contain a consecutive order of allied indexes. □

Detection: We detect alliances in problem instances as follows. First, we list all interactions as candidate alliances. Second, for each alliance, we look for overlaps with the other candidates. In the example in Figure 4.3, i_5 overlaps between $\{i_1, i_3, i_5\}$ and $\{i_2, i_5\}$. If there is any overlap, we break the alliances into non-overlapping subsets. In the above case $\{i_1, i_3\}$, $\{i_2\}$ and $\{i_5\}$. We remove alliances with only one index, obtaining $\{i_1, i_3\}$ in the example. The detection overhead is $O(|P|^2)$.

B.4.3 Colonized Indexes

Definition: An index i is called colonized by a colonizer index, j, *iff* all query plans using index i also use the colonizer, j, and the index has no interaction to speed up building other indexes.

Theorem 2: Every problem has at least one optimal solution where every colonized index is built after its colonizer.

Proof. Let i be a colonized index. Suppose a solution N in which there is a subsequence L_b between i and its colonizer, j, namely $N = L_a \to i \to L_j \to L_b$ where L_b contains j. Now, consider an altered solution $N' = L_a \to L_j \to i \to L_b$. With the same proof as alliance, the objective of N' is always smaller or same as that of N. Repeating this yields $N'' = L_a \to i \to j \to L_b$ which is no worse than all the other solutions that create indexes between i and j.

Consider $N''' = L_a \to j \to i \to L_b$. By the same discussion, we show that N''' is no worse than N'' and may even be better.

Once again by induction on the swapping operation, any solution that builds a colonized index before its colonizer can be improved by moving the colonized index after its colonizer. □

Detection: The detection algorithm for colonized indexes and its computational cost is quite similar to that of alliances. For each index, we consider all the query plans it appears in and take the intersection (overlap) of them, which is the colonizer(s). The detection overhead is $O(|I||P|)$.

B.4.4 Dominated Indexes

In Section 4.4.3, we explained a simplified case of dominated indexes. Here we discuss dominated indexes in detail.

Definition: Index i is dominated by index k *iff* all of the following conditions hold. $\forall L_a, L_j, j \in L_j$ in (B.13),

1. $S(k, \{L_a + L_j\}) \geq S(i, \{L_a + L_j\})$
2. $C(i, \{L_a + L_j + k\}) \geq C(k, \{L_a\})$
3. $C(j, \{L_a + i\}) \geq C(j, \{L_a + k\})$
4. $S(j, \{L_a + M + i\}) \leq S(j, \{L_a + M + k\}) : \forall M \in L_j, j \notin M$
5. $C(k, \{L_a + L_j\}) = C(k, \{L_a\})$

In short, k is *always* more beneficial and cheaper to build than i. Note that these conditions are re-evaluated when some index is determined to be before or after i or k because indexes after both i and k are irrelevant to these conditions. At each iteration we re-evaluate these conditions to ensure maximum dominance detection.

Theorem 3: An optimal solution does not build i before k.

Proof. Consider two solutions $N = L_a \rightarrow i \rightarrow L_j \rightarrow k \rightarrow L_b$ and $N' = L_a \rightarrow k \rightarrow L_j \rightarrow i \rightarrow L_b$. In this setting, i and k are single indexes. Therefore, in (B.13),

$$\begin{aligned}
G_i &= C(i, \{L_a\})S(i, \{L_a\}) \\
G'_i &= C(i, \{L_a + k + L_j\})S(i, \{L_a + k + L_j\}) \\
G_k &= C(k, \{L_a + i + L_j\})S(k, \{L_a + i + L_j\}) \\
G'_k &= C(k, \{L_a\})S(k, \{L_a\})
\end{aligned}$$

Also by the definition of S and R,

$$\begin{aligned}
S(i,\{L_a\}) + R_{L_a+i} &= R_{L_a} \\
S(i,\{L_a+k+L_j\}) + R_{L_a+i+L_j+k} &= R_{L_a+L_j+k} \\
S(k,\{L_a+i+L_j\} + R_{L_a+i+L_j+k} &= R_{L_a+i+L_j} \\
S(k,\{L_a\}) + R_{L_a+L_k} &= R_{L_a}
\end{aligned}$$

Applying these to (B.13), we get

$$\begin{aligned}
Obj(N) - Obj(N') &= (G_j - G'_j) \\
&\quad + R_{L_a}(C(i,\{L_a\}) - C(k,\{L_a\})) \\
&\quad + R_{L_a+i+L_j} C(L_j,\{L_a+i\}) \\
&\quad - R_{L_a+k+L_j} C(L_j,\{L_a+k\}) \\
&\quad + R_{L_a+i+L_j} C(k,\{L_a+i+L_j\}) \\
&\quad - R_{L_a+L_j+k} C(i,\{L_a+k+L_j\})
\end{aligned}$$

Because of the condition (3) and (4),

$$\begin{aligned}
\ldots &\geq R_{L_a}(C(i,\{L_a\}) - C(k,\{L_a\})) \\
&\quad + R_{L_a+i+L_j} C(L_j,\{L_a\}) \\
&\quad - R_{L_a+k+L_j} C(L_j,\{L_a+k\}) \\
&\quad + R_{L_a+i+L_j} C(k,\{L_a+i+L_j\}) \\
&\quad - R_{L_a+L_j+k} C(i,\{L_a+k+L_j\})
\end{aligned}$$

Because $C(L_j, \{L_a + k\}) \leq C(L_j, \{L_a\})$,

$$\begin{aligned} \ldots \geq\ & C(L_j, \{L_a\})(R_{L_a+i+L_j} - R_{L_a+k+L_j}) \\ & +R_{L_a}(C(i, \{L_a\}) - C(k, \{L_a\})) \\ & +R_{L_a+i+L_j}C(k, \{L_a + i + L_j\}) \\ & -R_{L_a+L_j+k}C(i, \{L_a + k + L_j\}) \end{aligned}$$

Because $C(i, \{L_a + k + L_j\}) \leq C(i, \{L_a\})$ and the condition (5) $(C(k, \{L_a + i + L_j\}) = C(k, \{L_a\}))$,

$$\begin{aligned} \ldots \geq\ & C(L_j, \{L_a\})(R_{L_a+i+L_j} - R_{L_a+k+L_j}) \\ & +C(i, \{L_a + k + L_j\})(R_{L_a} - R_{L_a+L_j+k}) \\ & -C(k, \{L_a\})(R_{L_a} - R_{L_a+i+L_j}) \\ =\ & C(L_j, \{L_a\})(S(k, \{L_a + L_j\}) - S(i, \{L_a + L_j\})) \\ & +C(i, \{L_a + k + L_j\})S(L_j + k, \{L_a\}) \\ & -C(k, \{L_a\})S(L_j + i, \{L_a\}) \end{aligned}$$

Now, by the definition of S,

$$\begin{aligned} S(L_j + k, \{L_a\}) &= S(L_j, \{L_a\}) + S(k, \{L_a + L_j\}) \\ S(L_j + i, \{L_a\}) &= S(L_j, \{L_a\}) + S(i, \{L_a + L_j\}) \end{aligned}$$

From the condition (1), $S(k, \{L_a + L_j\}) \geq S(i, \{L_a + L_j\}))$ and $S(L_j + k, \{L_a\}) \geq S(L_j + i, \{L_a\})$. Thus,

$$\ldots \geq= S(L_j + i, \{L_a\})(C(i, \{L_a + k + L_j\}) - C(k, \{L_a\}))$$

From the condition (2), $\ldots \geq= 0$ $\square$

Detection: We find dominated indexes in the following way. For each index, we calculate the minimum benefit and the maximum creation cost to make the indexes in each query plan. Then, we compare its ratio of minimum benefit to maximum cost with every other index's ratio of maximum benefit to minimum cost. During this procedure, we consider the additional constraints to tighten the minimum/maximum. The detection overhead is $O(|I||P|)$.

B.4.5 Disjoint Indexes and Clusters

Definition: A disjoint index is an index that has no interactions with other indexes.

Let $den_i(M) \equiv \dfrac{S(i,M)}{C(i,M)}$ denote the density of i. Let, $L_a \to i \to L_b$ be an optimal solution. Suppose a suffix L_j of L_a such that $L_a = L'_a \to L_j$.

Theorem 4: Every suffix is more dense than i if i is disjoint.

Proof. We compare two solutions $N = L'_a \to i \to L_j \to L_b$ and $N' = L'_a \to L_j \to i \to L_b$. Because i is a disjoint index, $G_i = G'_i, G_j = G'_j$, thus from (B.12),

$$\begin{aligned} Obj(N) - Obj(N') &= C(i)(R_{L'_a+i} - R_{L_a+i}) \\ &\quad -C(L_j,\{L'_a\})(R_{L_a} - R_{L_a+i}) \\ &= C(i)S(L_j,\{L'_a\}) - C(L_j,\{L'_a\})S(i) \end{aligned}$$

The density of i and L_j is $den_i = \dfrac{S(i)}{C(i)}, den_j = \dfrac{S(\{L_j\},\{L'_a\})}{C(L_j,\{L'_a\})}$.

Hence, $Obj(N) - Obj(N') = S(i)S(L_j,\{L'_a\})(den_i^{-1} - den_j^{-1})$. Therefore, if i has a larger density than any suffix, we can improve the solution by placing i before the suffix which contradicts the optimality assumption of N'. □

Likewise, the following theorem regarding a prefix of L_b holds. The proof is omitted as it is symmetric.

Theorem 5: Every prefix is less dense than i if i is disjoint.

Let a dip be the place where we can place a disjoint index i without violating the two theorems above. Now we prove that there is only one dip (except when there are ties).

Theorem 6: Every sequence has only one dip to insert a disjoint index i.

Proof. Suppose there are two or more dips. Let $d_1 < d_2$ be the dips. Consider the sub-sequence L_j between the places d_1 to d_2. From Theorem 4, L_j has a larger density than i, but from Theorem 5, L_j has a smaller density than i. By contradiction, there cannot be two or more dips. □

Now, we consider the more general cases of backward and forward disjoint. Their formal definition is as follows,

Definition: i is backward-disjoint to k *iff* all interacting indexes of i and k succeed i or precede k.

Definition: i is forward-disjoint to k *iff* all interacting indexes of i and k precede i or succeed k.

Theorem 7: An optimal solution does not build k before i if i is backward-disjoint to k and $den_i > den_k$.

Proof. Suppose $N = L_a \rightarrow k \rightarrow L_j \rightarrow i \rightarrow L_b$ is an optimal solution.

Consider the interactions i and k could have with L_j. Because i is backward-disjoint, none of its interacting indexes are in L_j. Also, none of k's interacting indexes are in L_j either. In other words, i and k are disjoint indexes regarding the subsequence L_j.

Therefore, from Theorem 4 and Theorem 5, k must be denser than L_j and L_j must be denser than i. However, by definition $den_i > den_k$ and we have a contradiction. Therefore, N cannot be an optimal solution. As L_a, L_j, L_b are arbitrary, and include empty sets, this means an optimal solution does not build k before i. □

Theorem 8: An optimal solution does not build i before k if i is forward-disjoint to k and $den_i < den_k$.

This proof is omitted as it is symmetric to the previous one.

Detection: We detect such cases as follows. For each pair of indexes, we check whether they are forward or backward disjoint to each other. If either of them is forward or backward disjoint, we can determine the interactions which i and k receive and calculate den_i, den_k. If the situation defined above occurs, we introduce the appropriate additional constraints. The overhead of this procedure is $O(|I|^2|P|)$.

B.4.6 Tail Indexes

Definition: Tail indexes are the last indexes to be built in a given build order L. Given some subset of indexes $M \in I$, we defined M's *tail group* as all solutions where tail indexes are permutations of

M. A *tail champion* of M is the solution in M's tail group that minimizes the tail's objective.

Theorem 9: A tail champion of M is better or same as all the other solutions in M's tail group.

Proof. Consider the set of preceding indexes $A \equiv I \setminus M$ and its order L_A. Let us compare the objective of $N = L_A \to L_M$ and $N' = L_A \to L'_M$. Suppose N' is a tail champion of M's tail group but N is not.

$$\begin{aligned} Obj(N) &= G_A + R_A C(L_A) + G_M + R_{A+M} C(L_M, A) \\ Obj(N') &= G_A + R_A C(L_A) + G'_M + R_{A+M} C(L'_M, A) \end{aligned}$$

Now, because N' and N are in the same tail group and N' is the tail champion, $G_M + R_{A+M} C(L_M, A) > G'_M + R_{A+M} C(L'_M, A)$ Therefore, $Obj(N) - Obj(N') \geq 0$ for every possible L_A. $\square$

Let $F = \{M_1, M_2, \ldots\}$ be the set of all possible tail groups in the problem. Let $Const$ be a rule that holds in all tail champions of $M \in F$.

Theorem 10: $Const$ holds in the optimal solution.

Proof. From Theorem 9, the only possible optimal solution from M's tail group is the tail champion. Because F is a comprehensive set of all possible tail groups, the optimal solution is one of the tail champions.

Thus, regardless which tail group the optimal solution appears in, $Const$ holds in the optimal solution. $\square$

This theorem proves the property used in Section 4.4.5. We note that $Const$ can be any kind of rule. For example, "i_1 appears as the last index", "i_2 is built after i_1", "i_3 never appears in the last 3 indexes".

Detection: At the end of each problem analysis iteration, we apply the tail analysis. We start from the tail length of 3 and increase the tail length until the number of tail candidates exceeds the threshold k. For each tail candidate, we calculate the tail objective and group them by the set of tail indexes as explained in Section 4.4.5. The detection overhead is obviously $O(k)$, thus k is a tuning parameter balancing on the pruning power and the overhead of pre-analysis. In our experiments, we used $k = 50000$.

B.4.7 Additional Experiments

Table B.3 shows how the additional constraints from each problem property affects the performance of the complete search experiment described in Section 4.7.3. We start with no additional constraint and add each problem property one at a time in the following order, **A**lliances, **C**olonized-indexes, **M**in/max-domination, **D**isjoint-clusters, and **T**ail-indexes. We only used additional constraints we could deduce within one minute, so the overhead of pre-analysis is negligible.

Table B.3: Exact Search (Reduced TPC-H). Time [min]. (DF) Did not Finish in 12 hours.

$\|I\|$	6	11	13	18	22	25	31	16	21
Density	low	low	low	low	low	low	low	mid	mid
CP	<1	7	214	DF	DF	DF	DF	DF	DF
+A	<1			DF	DF	DF	DF	DF	DF
+AC	<1			69	DF	DF	DF	DF	DF
+ACM	<1				249	DF	DF	DF	DF
+ACMD	<1					24	DF	DF	DF
+ACMDT	<1							1	DF

The results demonstrate that each of the five techniques improves the performance of the CP search by several orders of magnitude. The runtime of CP without pruning is roughly proportional to $|I|!$. Hence, the total speed-up of the additional constraints is at least $\frac{31!}{13!}214 = 2.7 \times 10^{26}$.

Bibliography

[Aba08] Daniel Abadi. *Query execution in column-oriented database systems*. PhD thesis, Massachusetts Institute of Technology, 2008.

[ABPA$^+$09] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *Proceedings of the VLDB Endowment*, 2(1):922–933, 2009.

[ACFT06] T. Apaydin, G. Canahuate, H. Ferhatosmanoglu, and A.S. Tosun. Approximate encoding for direct access and query processing over compressed bitmaps. In *Proceedings of the 32nd international conference on Very large data bases*, pages 846–857. VLDB Endowment, 2006.

[ACN00] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, pages 496–505, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[ACN06] S. Agrawal, E. Chu, and V. Narasayya. Automatic physical design tuning: workload as a sequence. In *SIGMOD*, 2006.

[ACTY09] Pankaj K. Agarwal, Siu-Wing Cheng, Yufei Tao, and Ke Yi. Indexing Uncertain Data. In *PODS*, 2009.

[ADJ$^+$10] S. Arumugam, A. Dobra, C.M. Jermaine, N. Pansare, and L. Perez. The DataPath system: a data-centric analytic processing engine for large data warehouses. In *Proceedings of the 2010 international conference on Management of data*, pages 519–530. ACM, 2010.

[AMF06] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 671–682, New York, NY, USA, 2006. ACM.

[AMHBAS07] B. Aleman-Meza, F. Hakimpour, I. B Arpinar, and A.P. Sheth. SwetoDblp ontology of Computer Science publications. *Web Semantics: Science, Services and Agents on the WWW*, 2007.

[AV07] David Arthur and Sergei Vassilvitskii. k-means++: the advantages of careful seeding. In *SODA '07: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[BC07] N. Bruno and S. Chaudhuri. Physical design refinement: The merge-reduce approach. *TODS*, 2007.

[BF93] J. Bunge and M. Fitzpatrick. Estimating the number of species: a review. *Journal of the American Statistical Association*, 88(421):364–373, 1993.

[BH03] Paul Brown and Peter Haas. BHUNT: Automatic Discovery of Fuzzy Algebraic Constraints in Relational Data. In *VLDB*, 2003.

[BPN01] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-Based Scheduling - Applying Constraint Programming to Scheduling Problems.* Springer, 2001.

[BSHW06] Omar Benjelloun, Anish D. Sarma, Alon Halevy, and Jennifer Widom. ULDBs: databases with uncertainty and lineage. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 953–964. VLDB Endowment, 2006.

[BU77] R. Bayer and K. Unterauer. Prefix B-trees. *ACM Transactions on Database Systems (TODS)*, 2(1):11–26, 1977.

[CCMN00] Moses Charikar, Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. Towards Estimation Error Guarantees for Distinct Values. In *PODS*, 2000.

[CDG+08] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[CFPT94] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Automatic generation of production rules for integrity maintenance. *ACM Transactions on Database Systems (TODS)*, 19(3):367–422, 1994.

[CGK+99] Q. Cheng, J. Gryz, F. Koo, C. Leung, L. Liu, X. Qian, and B. Schiefer. Implementation of two semantic query optimization techniques in DB2 universal database. In *PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES*, pages 687–698. Citeseer, 1999.

[CN97] Surajit Chaudhuri and Vivek R. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 146–155, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[CN98] S. Chaudhuri and V. Narasayya. AutoAdmin what-if index analysis utility. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, page 378. ACM, 1998.

[CN99] S. Chaudhuri and V. Narasayya. Index merging. In *Data Engineering, 1999. Proceedings., 15th International Conference on*, pages 296–303, 1999.

[COO08] X. Chen, P. O'Neil, and E. O'Neil. Adjoined dimension column clustering to improve data warehouse query performance. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 1409–1411. IEEE, 2008.

[CXP+04] R. Cheng, Y. Xia, S. Prabhakar, R. Shah, and J. S. Vitter. Efficient indexing methods for probabilistic threshold queries over uncertain data. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 876–887. VLDB Endowment, 2004.

[Dea09] Jeff Dean. Designs, lessons and advice from building large distributed systems. Keynote at LADIS, 2009. `http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf`.

[DG08] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[DGS$^+$90] D.J. DeWitt, S. Ghandeharizadeh, D.A. Schneider, A. Bricker, H.I. Hsiao, and R. Rasmussen. The Gamma database machine project. *Knowledge and Data Engineering, IEEE Transactions on*, 2(1):44–62, 1990.

[DQRJ$^+$10] J. Dittrich, J.A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *Proceedings of the VLDB Endowment*, 3(1-2):515–529, 2010.

[DRS09] Nilesh Dalvi, Christopher Ré, and Dan Suciu. Probabilistic databases: diamonds in the dirt. *Commun. ACM*, 52(7):86–94, 2009.

[EBa08] eBay Developer API. http://developer.ebay.com/products/trading, 2008.

[ETÖ$^+$11] M.Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson. CoHadoop: flexible data placement and its exploitation in Hadoop. *Proceedings of the VLDB Endowment*, 4(9):575–585, 2011.

[FLP$^+$10] D. Ford, F. Labelle, F.I. Popovici, M. Stokely, V.A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[FPST11] A. Floratou, J.M. Patel, E.J. Shekita, and S. Tata. Column-oriented storage techniques for MapReduce. *Proceedings of the VLDB Endowment*, 4(7):419–429, 2011.

[GGZ01] P. Godfrey, J. Gryz, and C. Zuzarte. Exploiting constraint-like data characterizations in query optimization. *ACM SIGMOD Record*, 30(2):592, 2001.

[Gib01] P.B. Gibbons. Distinct Sampling for Highly-Accurate Answers to Distinct Values Queries and Event Reports. In *VLDB*, pages 541–550, 2001.

[GK03] F. Glover and G.A. Kochenberger. *Handbook of metaheuristics.* Springer, 2003.

[GL97] F. Glover and M. Laguna. *Tabu Search.* Kluwer, 1997.

[GM99] H. Gupta and I. Mumick. Selection of views to materialize under a maintenance cost constraint. *ICDT*, 1999.

[GST$^+$02] J. Gray, A.S. Szalay, A.R. Thakar, P.Z. Kunszt, C. Stoughton, D. Slutz, et al. Data mining the SDSS SkyServer database. *Arxiv preprint cs/0202014*, 2002.

[GSZZ02] J. Gryz, B. Schiefer, J. Zheng, and C. Zuzarte. Discovery and application of check constraints in DB2. In *Data Engineering, 2001. Proceedings. 17th International Conference on*, pages 551–556. IEEE, 2002.

[HLH$^+$11] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu. RCFile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 1199–1208. IEEE, 2011.

[Huo07] George Huo. Correlation indices: a new access method to exploit correlated attributes. Master's thesis, Massachusetts Institute of Technology, 2007.

[HZ80] M. Hammer and S.B. Zdonik. Knowledge-based query processing. In *Proceedings of the sixth international conference on Very Large Data Bases-Volume 6*, pages 137–147. VLDB Endowment, 1980.

[Ikh10] Newton Ikhariale. Fractured Indexes: Improved B-trees To Reduce Maintenance Cost And Fragmentation. Master's thesis, Brown University, 2010.

[IMH$^+$04] Ihab F. Ilyas, Volker Markl, Peter Haas, Paul Brown, and Ashraf Aboulnaga. CORDS: automatic discovery of correlations and soft functional dependencies. In *SIGMOD*, pages 647–658, 2004.

[JOY07] Christopher Jermaine, Edward Omiecinski, and Wai Yee. The partitioned exponential file for database storage management. *The VLDB Journal*, 16(4):417–437, October 2007.

[JQRD11] A. Jindal, J.A. Quiané-Ruiz, and J. Dittrich. Trojan data layouts: Right shoes for a running elephant. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 21. ACM, 2011.

[KCRZ12] H. Kimura, C. Coffrin, A. Rasin, and S.B. Zdonik. Optimizing index deployment order for evolving olap. *EDBT*, 2012.

[KHR+09] Hideaki Kimura, George Huo, Alexander Rasin, Samuel Madden, and Stanley B. Zdonik. Correlation Maps: A Compressed Access Method for Exploiting Soft Functional Dependencies. In *VLDB'09: Proceedings of the 2009 VLDB Endowment*. VLDB Endowment, August 2009.

[KHR+10] Hideaki Kimura, George Huo, Alexander Rasin, Samuel Madden, and Stanley B. Zdonik. CORADD: Correlation Aware Database Designer for Materialized Views and Indexes. *Proc. VLDB Endow.*, 3:1103–1113, September 2010.

[Kin81] J.J. King. Quist: A system for semantic query optimization in relational databases. In *Proceedings of the seventh international conference on Very Large Data Bases-Volume 7*, pages 510–517. VLDB Endowment, 1981.

[KMZ10] Hideaki Kimura, Samuel Madden, and Stanley B. Zdonik. UPI: A Primary Index for Uncertain Databases. In *Proceedings of the 36th International Conference on Very Large Data Bases*. VLDB Endowment, September 2010.

[KTS+02] N. Karayannidis, A. Tsois, T. Sellis, R. Pieringer, V. Markl, F. Ramsak, R. Fenk, K. Elhardt, and R. Bayer. Processing star queries on hierarchically-clustered fact tables. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 730–741. VLDB Endowment, 2002.

[LCZ05] Q. Lian, W. Chen, and Z. Zhang. On the impact of replica placement to the reliability of distributed brick storage systems. In *Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference on*, pages 187–196. IEEE, 2005.

[LD05] Marco E. Lubbecke and Jacques Desrosiers. Selected Topics in Column Generation. *Oper. Res.*, 53(6):1007–1023, 2005.

[Ley09] Michael Ley. DBLP - Some Lessons Learned. *PVLDB*, 2009.

[Llo82] S. Lloyd. Least squares quantization in PCM. *Information Theory, IEEE Transactions on*, 28(2):129–137, 1982.

[Mon] MonetDB. `http://monetdb.cwi.nl/`.

[MSD] Optimizing Queries That Access Correlated datetime Columns. `http://msdn.microsoft.com/en-us/library/ms177416(SQL.90).aspx`.

[NFJ09] I. Nagy, R. Farkas, and M. Jelasity. Researcher affiliation extraction from homepages. *ACL-IJCNLP 2009*, page 1, 2009.

[OCGO96] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[Ola98] Olap. APB-1 OLAP Benchmark Release II, 1998.

[OOC07] P. E. O'Neil, E. J. O'Neil, and X. Chen. The Star Schema Benchmark (SSB). Technical report, University of Massachusetts, Boston, 2007.

[OR95] F. Olken and D. Rotem. Random Sampling from Databases - A Survey, 1995.

[ORA] Using Extended Statistics to Optimize Multi-Column Relationships and Function-Based Statistics. `http://www.oracle.com/technology/obe/11gr1_db/perform/multistats/multicolstats.htm`.

[PA07] Stratos Papadomanolakis and Anastassia Ailamaki. An Integer Linear Programming Approach to Database Design. In *ICDE Workshops*, pages 442–449, 2007.

[PBM+03] S. Padmanabhan, B. Bhattacharjee, T. Malkemus, L. Cranston, and M. Huras. Multi-dimensional clustering: a new data layout scheme in DB2. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 637–641. ACM, 2003.

[PPR+09] A. Pavlo, E. Paulson, A. Rasin, D.J. Abadi, D.J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 35th SIGMOD international conference on Management of data*, pages 165–178. ACM, 2009.

[PSQa] PostgreSQL home page. `http://www.postgresql.org/`.

[PSQb] PostgreSQL 8.1.17 Documentation: pg_stats. `http://www.postgresql.org/docs/8.1/static/view-pg-stats.html`.

[S$^+$09] K. Schnaitter et al. Index interactions in physical design tuning: Modeling, analysis, and applications. *VLDB*, 2009.

[SAB$^+$05] M. Stonebraker, Daniel Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-store: a column-oriented DBMS. In *Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.

[SIC07] M.A. Soliman, I.F. Ilyas, and K.C.C. Chang. Top-k query processing in uncertain databases. In *ICDE*, 2007.

[SMP$^+$07] S. Singh, C. Mayfield, S. Prabhakar, R. Shah, and S. Hambrusch. Indexing uncertain categorical data. In *IEEE ICDE Conference*, 2007.

[SO87] Sreekumar T. Shenoy and Meral Z. Ozsoyoglu. A system for semantic query optimization. In *SIGMOD*, 1987.

[Suc08] Dan Suciu. Database Theory Column: Probabilistic databases. *SIGACT News*, 39(2):111–124, 2008.

[SW92] Jorge P. Sousa and Laurence A. Wolsey. A time indexed formulation of non-preemptive single machine scheduling problems. *Mathematical Programming*, 54:353–367, 1992. 10.1007/BF01586059.

[TAO] www.cse.cuhk.edu.hk/˜taoyf/paper/tods07-utree.html.

[TCX$^+$05] Yufei Tao, Reynold Cheng, Xiaokui Xiao, Wang K. Ngai, Ben Kao, and Sunil Prabhakar. Indexing multi-dimensional uncertain data with arbitrary probability density functions. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 922–933. VLDB Endowment, 2005.

[Vec] VectorWise. `http://www.vectorwise.com/`.

[Ver] Vertica — Data at the Speed of Life. `http://www.vertica.com/`.

[VHM09] P. Van Hentenryck and L. Michel. *Constraint-based local search.* The MIT Press, 2009.

[WM11] E. Wu and S. Madden. Partitioning Techniques for Fine-grained Indexing. In *Proceedings of ICDE 2011*, 2011.

[ZLZ$^+$55] Ying Zhang, Xuemin Lin, Wenjie Zhang, Jianmin Wang, and Qianlu Lin. Effectively Indexing the Uncertain Space. *IEEE Transactions on Knowledge and Data Engineering*, 99(2), 5555.

[ZRL$^+$04] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy Lohman, Adam Storm, Christian G. Arellano, and Scott Fadden. DB2 design advisor: integrated automatic physical database design. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 1087–1097. VLDB Endowment, 2004.