Multi-column Bitmap Indexing

by

Eduardo Gutarra Velez

Bachelor of Computer Science, EAFIT, 2009

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Computer Science (MCS)

in the Graduate Academic Unit of Computer Science

Supervisor(s):	Owen Kaser, Ph.D. Computer Science
	Daniel Lemire, Ph.D. Engineering Mathematics
Examining Board:	Josée Tassé, Ph.D. Computer Science, Chair
	Huajie Zhang, Ph.D. Computer Science
	Dongmin Kim, Ph.D. MIS (Faculty of Business)

This thesis is accepted

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

May, 2012

Abstract

Bitmap indices are used to accelerate queries against large databases. We often expect bitmap indices over high-cardinality columns to be too large to be practical. Thus, we might expect *multi-column* bitmap indices to be disastrous because they are similar to indices over high-cardinality columns. Our findings, however, contradict this intuition. We find that multi-column bitmap indices can be smaller than conventional bitmap indices over individual columns. Yet multi-column bitmap indices can also be alarmingly large in some instances. Hence, our discovery motivates an engineering question: how do we efficiently determine whether a multi-column bitmap index is relatively small without first constructing it? We tried modeling the problem in terms of statistical characteristics such as correlation. However, we found simple heuristics based on indexing samples most effective. Our heuristics could be integrated in database systems to help database administrators pick a better indexing strategy.

Table of Contents

List of Tables

List of Figures

List of Symbols

Notation	Definition	Page
b_i	base for a component	14
c	number of columns	58
k	# of bitmaps encoded to an attribute value	14
L	attribute cardinality	11
L_i	attribute cardinality of column	43
$L_{\max}$	column with highest attribute cardinality	35
$L_{\min}$	column with lowest attribute cardinality	32
m_α	# of bitmaps in a single-column index	44
m_β	# of bitmaps in a multi-column index	44

m_λ	number of bitmaps in bitmap index λ	44
n	number of records	11
p	probability of duplicating an element	68
w	processor word size	11
α	a single-column bitmap index	44
β	a multi-column bitmap index	44
λ	a bitmap index	44
ϕ_m	maximum cardinality-based correlation	35
ϕ_p	product cardinality-based correlation	35
ϱ	probability a possible tuple is selected	70
$\textsc{RunCount}_{32}(e)$	$\textsc{RunCount}_{32}$ of an element e	47

Chapter 1

Introduction

In a data warehouse, there is often a need to scan datasets to answer several queries for different analyses. In these datasets the number of records may well be in the order of billions and above. Answering queries in such large databases is often a slow process, but the speed can be improved by using an index. There are many types of indices, but one index that is commonly used is the bitmap index. A bitmap index is a collection of bit arrays (also known as bitmaps) where the attribute values are taken as attributes, and their presence in a given row is denoted with "1" and absence with a "0" (see Section 2.3 and Figure 1.1). The bitmap index is known to be efficient, especially when the data is mostly read-only or append-only. Major DBMS vendors such as Oracle, Sybase, and IBM support bitmap indices in their products.

The size of a bitmap index can grow considerably as attribute cardinality

Area Code	City
506	Fredericton
514	Montreal
506	Saint John
506	Saint John
506	Fredericton
506	Fredericton
514	Montreal

(a)

506	514	Fredericton	Montreal	Saint John
1	0	1	0	0
0	1	0	1	0
1	0	0	0	1
1	0	0	0	1
1	0	1	0	0
1	0	1	0	0
0	1	0	1	0

(b)

Figure 1.1: Illustration of a table of cities and area codes (a), with its corresponding bitmap index (b). Each column in Figure (b) is a bitmap or bit array of the index, with "1s" in the positions where the attribute value is present, and "0s" where it is absent.

is increased and thus conventional wisdom tells us that bitmap indices are only appropriate for low cardinality columns, such as gender or month [3, 4]. However, there are techniques to reduce the size of a bitmap index even when cardinality is high. These techniques can be classified into two general approaches [17]:

- Reducing the number of bitmaps in the index by encoding the attribute values.
- Reducing the size of the bitmaps themselves by compressing runs of 0s and 1s with Run Length Encoding (RLE) compression schemes. For example, the bitmap '0000001111' can be represented as 6 $\times$ 0's and 4 $\times$ 1's.

Because bitmap indices can be reduced in size, researchers show that it is not always the case that they are only suitable for low cardinality columns [17,

26, 32, 33, 35, 36]. In fact, according to Sharma, the effectiveness of bitmap indices may depend more on the type of application than on cardinality [26]. Coping with cardinality is not the only reason for reducing the size of bitmap indices, other good reasons include:

- Making the process of evaluating a query with the index less I/O bound.
- Making the process of scanning the index less computationally bound, if the indices are processed compressed.
- Decreasing the cost on memory thus making better use of disk space. This decrease allows for a greater number of columns to be indexed.

In the following section, we discuss the major focus of our thesis which is that there are two types of bitmap indices that provide different advantages. We focus on the advantages in size that one of the two bitmap indices can have for certain datasets. Our goal is to be able to describe for which datasets it is better to index using a multi-column bitmap index than a single-column bitmap index. Then Section 1.2 presents our contributions and an overview of how this thesis is organized.

1.1 Single-Column and Multi-Column

Bitmap indices have a bitmap for each distinct element where an element can be an attribute value of a column, or a conjunction of columns. A *single-column bitmap index* can be built considering the attribute values of the

columns of a table separately (see Figure 1.1b). We may also build bitmap indices considering each distinct tuple of a table as an attribute value, namely a *multi-column bitmap index* (also described as a concatenated index [9]) (see Figure 1.2). As an alternative to encoding attribute values for reducing the size of a bitmap index, we consider building multi-column bitmap indices. Although a multi-column bitmap index may have more bitmaps than a single-column bitmap index, the bitmaps themselves may be more compressible. Thus, the multi-column bitmap index can be smaller than the single-column bitmap index when compressed. Moreover, multi-column bitmap indices have other advantages over single-column bitmap indices on conjunctive queries, as briefly discussed in Section 4.1. In this thesis, however, we concern ourselves with the advantages multi-column bitmap indices can have in size.

506 ∧ Fredericton	506 ∧ Saint John	514 ∧ Montreal
1	0	0
0	0	1
0	1	0
0	1	0
1	0	0
1	0	0
0	0	1

Figure 1.2: Multi-column bitmap index for the table illustrated in Figure 1.1a. Notice that the single-column index in Figure 1.1b has two more bitmaps than the multi-column index.

To the best of our knowledge no previous work has been done on evaluating the applicability and effectiveness of multi-column bitmap indices. In this thesis, we compare the size of multi-column bitmap indices to that of single-

column bitmap indices. Deciding when a multi-column bitmap index is considered small may vary depending on the context. Thus, we focus on finding characteristics of datasets, or techniques for quickly recognizing datasets for which multi-column bitmap indices are smaller than single-column bitmap indices.

We examine the relevant characteristics of those datasets as statistical parameters and demonstrate that they cannot be used reliably to make predictions on realistic datasets. Alternatively, we contribute sampling heuristics that help determine when a multi-column bitmap index is smaller than a single-column bitmap index.

1.2 Organization and Contributions

In the next chapter, we discuss important concepts necessary for understanding the storage requirements of multi-column bitmap indices. We present characteristics of bitmap indices as well as techniques used to reduce their disk space usage. We also present related work. Chapter 3 describes the statistics we use to characterize the datasets. These statistical characteristics help us categorize datasets, and from these statistical characteristics we hope to determine which datasets lead to small multi-column bitmap indices. Of the statistical characteristics, we cover in greater depth correlation and introduce two correlation measures based on cardinality. We also describe the two data distributions we used to generate synthetic datasets in

Chapter 5, namely, the Zipfian and uniform distributions. These synthetic datasets allow understanding the behavior of disk space occupied as the statistical parameters with which we generated them are varied.

In Chapter 4, we explore multi-column bitmap indices in greater detail. We provide bounds for important characteristics that determine a multi-column bitmap index's size. For instance, we establish an upper and lower bound for the number of bitmaps in a multi-column bitmap index in terms of number of records and attribute cardinalities. We also demonstrate that it is possible to determine when a multi-column bitmap index will be smaller than a single-column bitmap index under certain extreme conditions. Yet, most conditions are not extreme and it is overall not possible to determine when a multi-column bitmap index will be smaller than a single-column bitmap index without information on the row order of the table.

In Chapter 5, we describe experiments on synthetic datasets that validate some of our theoretical results. We also contribute theoretical results found from experiments on these synthetic datasets. For instance, we found that increasing the number of columns or the cardinality or the correlation often leads to smaller multi-column bitmap indices than single-column bitmap indices. We also explore correlation more in depth on various models where the correlation factor with which we generate the dataset is varied. In our synthetic datasets, correlation always favored smaller multi-column bitmap indices as the correlation factor increased. The last section of this chapter explores the behavior of Zipfian synthetic datasets by varying a skew parameter

that converts an initially uniform distribution to a Zipfian distribution. We use Zipfian distributions because many real datasets can be approximated to a Zipfian distribution. There are various researchers that use Zipfian distributions to emulate real data and we follow their lead [12, 34]. Overall, we noticed that increasing the skew, making a data distribution "more Zipfian", made multi-column indices larger when compared to single-column indices. With synthetic data experiments we see the effect that varying statistical parameter has in the size of multi-column and single-column bitmap indices.

In Chapter 6, we continue our work with experiments based on real data. In these experiments, we compare statistical metrics according to their ability to predict size difference between multi-column and single-column bitmap indices. We do so through a notion of profit where we pick the top-n predictions of each metric. As the competing statistical metrics we use correlation measures and indexing samples. We index samples as another metric because, unlike our correlation techniques, this technique can take into account the order of the table. Our largest dataset has hundreds of millions of records. Our datasets have up to 42 different columns.

In Chapter 7 we present sampling heuristics derived from the results of our realistic experiments. All of our heuristics can usually help us reduce the amount of disk space occupied when using a sample size of 10%. In our datasets we obtained a reduction of 2% to 17% in the disk space occupied by the bitmap indices. We compare three different heuristics on five realistic datasets.

Last, Chapter 8 reviews our main conclusions and discusses possible future work. This thesis focuses on the storage requirements, which plays a part in the effectivess of bitmap indices, but it is not sufficient for a decision on when to use multi-column or single-column bitmap indices. An important aspect that is left as future work for such a decision is the workload performance of these indices. In other words, we could explore the performance of our bitmap indices under specific collections of queries.

Chapter 2

Background

In this chapter, we start with Section 2.1 where we briefly review common concepts of relational databases. We also explain in further detail compression techniques that may be used to reduce the size of bitmap indices. In Section 2.2, we discuss existing techniques for reducing the size of bitmap indices that we divide in two categories, encoding and compression. Because we only use a trivial encoding technique, we focus on compression schemes that are based on Run Length Encoding (RLE). The compressibility of RLE depends on the order of the data, thus we discuss in Section 2.3 how sorting improves the compressibility of bitmaps. We also introduce the lexicographic sort order that is used throughout our work to improve the compressibility of our bitmap indices.

2.1 Relational Databases

Throughout, we use relational database terminology as we build bitmap indices over tables. These tables consist of *rows* or *records* and have one or more columns. Each *row* of a *table* has particular *attribute value* for each *column*. We often keep count of the number of records in the table and refer to this count as n. Also, a count for the number of distinct attribute values found in all records for a column is also kept; it is referred to as the *attribute cardinality* and denoted as L.

2.2 RLE Compression

Run Length Encoding (RLE) is a simple data compression technique that consists in replacing redundant sequences of a same element with more compact sequences [24]. RLE has been the basis for various types of compression techniques such as Golomb coding [11]. The compact sequence consists of the repeated element along with a count of the number of element repetitions. For example, the sequence aaaaaabbccc becomes 6 × a, 2 × b, 3 × c.

In RLE, a run is a sequence of repeating elements. Runs can have one or more elements; we call a run with one element a single element run, and runs with two or more elements, multi-element runs. For example, in the sequence aaabba; aaa and bb are multi-element runs and 'a' is a single-element run.

RLE often differ in what is considered the element to be counted. For example, suppose we consider in an RLE variation that an element is a triplet

of 0s, thus the sequence 000000 would constitute a run of two elements, or a run of length two.

RLE is not only used to reduce disk space size, but it may also be used to reduce processing time [16]. For instance, when doing operations that involve counting or summing the number of elements, the counts in the RLE compressed sequence can be used.

2.3 Bitmap Indices

Bitmap indices were first introduced by O'Neil, Spiegler and Maayan [21, 28]. They are a way of representing a table (or certain relevant columns of a table) such that a processor with a w-bit architecture can process the data within w rows in parallel. Certain types of queries may benefit from this vectorization, thus reducing query response times.

In a bitmap index, the attribute values of a column become attributes themselves, and each of those attributes has only two possible attribute values: present "1" or absent "0" (see Figure 1.1). The size of this representation is larger than a representation where we number each distinct attribute value with a unique binary consecutive integer starting at zero. If we let n be the number of rows and L be the number of distinct attribute values in a given column, we have a total of nL bits in the index. In contrast, if we assign a binary value to each distinct attribute value, we obtain a smaller representation of the data with size $n \log_2 L$. Figure 2.1 shows the size comparison

0	1	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0
3	0	0	0	1	0	0	0	0
4	0	0	0	0	1	0	0	0
5	0	0	0	0	0	1	0	0
6	0	0	0	0	0	0	1	0
7	0	0	0	0	0	0	0	1

(a) nL bits in size

0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

(b) $n \log_2 L$ bits in size

Figure 2.1: Comparison between binary representation and bitmap representation. The first column in each table represents an attribute value, the remaining cells in each row are the bits necessary to represent each attribute value.

between these two representations, and it can be noted that $n \log_2 L < nL$ is always true.

As indicated before, we find the following compelling reasons for making a bitmap index small:

- It makes the process of evaluating a query with the index less I/O bound.
- When bitmap indices are processed compressed the process of scanning the index becomes less computationally bound.
- The cost on memory decreases, thus more effective use of disk space can be made since a greater number of columns can be indexed.

To reduce the space taken by a bitmap index, there are two general approaches that may be used [17]:

- **Encoding Techniques**: This approach reduces the number of bit arrays in the index by encoding attribute values.
- **RLE compression schemes**: This approach reduces the size of the bit arrays themselves by compressing runs of 0s and 1s using specialized variants of the technique discussed in Section 2.2.

These two general approaches will be discussed in through the remainder of this section. Often these two general approaches—encoding attribute values and compressing the bit arrays through RLE schemes—work in opposite ways, and thus changing the encoding of the attribute values often has the effect that while reducing the number of bit arrays, it makes them "dirty" and not very compressible.

Encoding

The bitmap indices we construct in this thesis use the most basic encoding known as trivial encoding where we assign each distinct attribute value its own bitmap (see Figure 2.2a). However, it is possible to use non-trivial encodings to build either multi-column or single-column bitmap indices. For completeness, we discuss some non-trivial encoding strategies that could be used for multi-column bitmap indices in future work.

Non-trivial encoding strategies can be lossy or lossless. Lossless strategies consist in creating attribute value signatures that correspond to a unique subset of bitmaps. Therefore all queries on any attribute value can be fully

resolved by the index. Lossy strategies render indices that are not able to fully resolve all queries accurately as will be illustrated later.

One strategy to decrease the number of bitmaps is using bit-sliced indices [20]. It involves using the binary representation of the attribute value. Therefore one can potentially represent 2^b attribute values with b bitmaps. This strategy decreases the number of bitmap indices the most, but also has the disadvantage that when processing a query, all or most of the bit arrays associated to an attribute must be scanned to answer the query. Figure 2.2b illustrates how the number of bitmaps is reduced when using binary representation encoding.

A second strategy consists in creating unique signatures for each attribute value with k bitmaps. This way we would only need to look up k bitmaps to identify the attribute value correctly. The number of attribute values that can be represented with n bitmaps with combinations of k elements is given by $\binom{n}{k}$ [31]. This strategy also decreases the number of bitmaps to a lesser extent than the first strategy, but it only needs to scan k bitmaps to answer a query (see Figure 2.2c).

A third encoding strategy we could use involves splitting the attribute values into k components by doing division or modulus operations with a set of bases $b_1, b_2, \ldots, b_i, b_k$ [5, 6, 27, 34]. Under this encoding strategy the maximum number of attribute values we can represent is given by $\prod_{i=1}^{k} b_i$. For each component i , the number of bitmaps used is equal to the value of its base b_i . Therefore, we have $\sum_{i=1}^{k} b_i$ bitmaps in total. As an example we will

use 2 components, both with the same base of 4. With this encoding, we can represent a maximum of 16 attribute values with a total of 8 bitmaps. Here, the first component is represented with the first 4 bitmaps, and the second is represented with the other 4 (see Figure 2.2d). The first encoding strategy discussed, the bit-sliced encoding, can be viewed as an instance of this encoding strategy where the base of the component is equal to 2.

Finally, the last encoding strategy we discuss is a lossy encoding strategy. It consists in identifying a group of attribute values that fall in a certain interval with a single value. This encoding strategy is often viewed as a separate category known as binning. Unlike the regular encoding strategies, a bitmap or set of bitmaps may not uniquely map to a single attribute value, but a set of them. All queries may not be fully resolved by the index alone and the base data must be examined to determine which of the values in the interval truly satisfy the query's condition. For instance, suppose the values Jack and Jill are in the same bin. If we query for tuples containing the value Jack, our index also returns rows containing Jill. Thus, we must retrieve the rows themselves and determine if they indeed contain the value Jack or if they are false positives containing the value Jill. This process is known as a *candidate check*, and will often dominate the total query processing time. This setback can be improved by organizing the base data such that it can be accessed quickly from disk. To do this, Wu et al. use a data structure named Order-preserving Bin-based Clustering (OrBiC) table [35].

	b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}	b_{11}	b_{12}	b_{13}	b_{14}	b_{15}
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

(a) Trivial Encoding Strategy

	b_0	b_1	b_2	b_3
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1

(b) Binary Encoding

	b_0	b_1	b_2	b_3	b_4	b_5	b_6
0	1	1	0	0	0	0	0
1	1	0	1	0	0	0	0
2	1	0	0	1	0	0	0
3	1	0	0	0	1	0	0
4	1	0	0	0	0	1	0
5	1	0	0	0	0	0	1
6	0	1	1	0	0	0	0
7	0	1	0	1	0	0	0
8	0	1	0	0	1	0	0
9	0	1	0	0	0	1	0
10	0	1	0	0	0	0	1
11	0	0	1	1	0	0	0
12	0	0	1	0	1	0	0
13	0	0	1	0	0	1	0
14	0	0	1	0	0	0	1
15	0	0	0	1	1	0	0

(c) Encoding $\binom{n}{k}$ for $k = 2$

	b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7
0	1	0	0	0	1	0	0	0
1	1	0	0	0	0	1	0	0
2	1	0	0	0	0	0	1	0
3	1	0	0	0	0	0	0	1
4	0	1	0	0	1	0	0	0
5	0	1	0	0	0	1	0	0
6	0	1	0	0	0	0	1	0
7	0	1	0	0	0	0	0	1
8	0	0	1	0	1	0	0	0
9	0	0	1	0	0	1	0	0
10	0	0	1	0	0	0	1	0
11	0	0	1	0	0	0	0	1
12	0	0	0	1	1	0	0	0
13	0	0	0	1	0	1	0	0
14	0	0	0	1	0	0	1	0
15	0	0	0	1	0	0	0	1

(d) Multi-Component Encoding with 2 components and base 4

Figure 2.2: Different encoding strategies for bitmap indices. Notice the number of bitmaps necessary to represent 16 attribute values under the different encoding strategies.

Compression Schemes

Making the bit arrays smaller can also reduce the size of the bitmap index. To do so, compression schemes such as LZ77 [37], Byte-Aligned Bitmap Code (BBC) [2], or Word Alignment Hybrid (WAH) [33] are used to compress the bitmaps. LZ77 is a general purpose deflation algorithm, while both BBC and WAH are variations of RLE schemes. Among these compression schemes there is a space-time trade off. WAH compressed indices are larger but faster than BBC indices, and BBC are also larger and faster than LZ77 compressed indices [33]. We discuss variations of the RLE compression schemes for bitmaps and do not go into LZ77 as it is too slow for further consideration.

The BBC compression scheme divides a sequence of bits into bytes. Each BBC element consists of 8 bits with the same bit value. As an example, if we have 3 elements of value 0, we actually have $3 \times 8 = 24$ bits with a bit value of 0. Thus, elements are counted in multiples of 8 bits. In contrast, a WAH element is a sequence of 31 bits with the same bit value. WAH encoding consists of two types of words: literals and fills. One bit is reserved to identify them. A literal word is a mixed sequence of 0s and 1s. A fill is a run comprised of clean words only, where clean words are 31-bit sequences containing only 0s or only 1s. Fill words reserve 1 more bit to determine whether their runs are of 0s or 1s. Thus, 30 bits are left to store the total number of WAH runs in the sequence. WAH uses a word instead of a byte as the element to form runs because a general-purpose CPU can operate on

words more efficiently than on bytes.

We compress our bitmaps using a compression scheme similar to WAH called Enhanced Word Alignment Hybrid (EWAH). This compression scheme encodes 32-bit fills or runs of clean words. Two types of words are used: marker words and verbatim words. Verbatim words are 32-bit sequences of mixed 0s and 1s. Marker words use one bit to indicate the type of clean word (0 or 1), 16 bits to store the total number of clean words that follow, and 15 bits to store the number of dirty words that follow after the run of clean words.

A recent paper by Fusco et al. [10] introduces the COMPressed Adaptive Index (COMPAX), which consists in initially compressing with a small variation of WAH that encodes only 0-fill words, and dirty words. These WAH compressed bitmaps are then further compressed by encoding sequences of words that form the patterns LFL or FLF. In these patterns, L words are dirty words that contain a single dirty byte and F words are 0-fill words with a count between 1 and 255. Thus, they effectively reduce these sequences of three words into a single word. In Figure 2.3 we illustrate a WAH compressed sequence forming an LFL pattern, which is further encoded in an LFL word.

2.4 Sorting to Improve RLE Compression

The compression achieved by RLE schemes in a bitmap index may be improved by sorting. Because the compressibility of a bitmap index depends on the RLE scheme used, we opt for a more generic model for its representation.

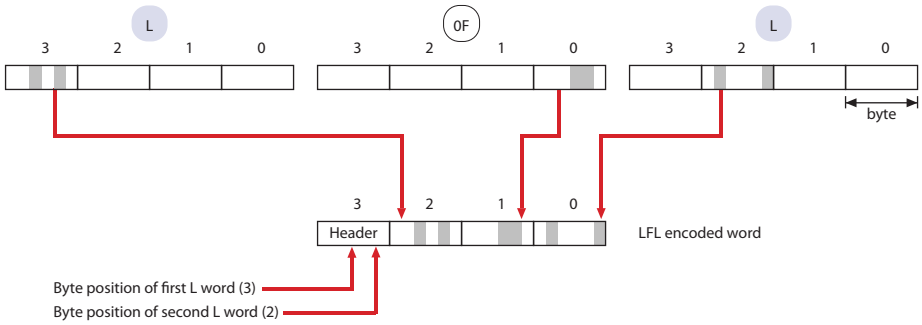


Figure 2.3: LFL sequence of 32-bit words after compressing with WAH. This three word sequence is further encoded storing dirty bytes in a single word.

Lemire et al. [16] model RLE compression by counting the runs, and consider that tables minimizing the total number of runs are more compressible. Even though counting runs is a convenient and sensible model for the compressibility of a bitmap index, it is not always a good approximation. In fact, compression techniques may be more efficient on datasets with row orderings that do not necessarily minimize the number of runs, but have many "long runs". Nonetheless, we use the number of runs, also referred to as the RunCount, to determine compressibility as it is usually a good approximation, and it simplifies the problem.

For a single column, minimizing RunCount is trivial—putting all instances of an attribute value adjacent to one another attains the minimum RunCount. However, determining which row ordering attains the minimum RunCount for a table with several columns is difficult. A row arrangement that minimizes the RunCount of every column in a table is often not possible as columns need different row rearrangements to minimize their own

RunCounts. Thus, a compromise among the different column's row rearrangements must be met where the table's RunCount, that is the sum of the column's RunCounts, should be minimized.

To find the best row arrangement for a table, one could try all possible combinations of row orders until the minimum RunCount is found, but doing so would be combinatorially explosive. A table with 100 records has more than a googol (10^{100}) possible row rearrangements, and big tables often have billions of records. This problem is in fact NP-Hard and reduces to the Traveling salesman problem under the Hamming distance [16]. Thus, we opt for sorting heuristics that can take us to good approximate solutions without considering all row rearrangements.

Row reordering may be done offline or online. Offline row reordering schemes work on the entire dataset, while online row reordering schemes work on pieces of the dataset without considering the entire dataset. Fusco et al. [10] uses online Locality Sensitive Hashing (oLSH), an online row reordering scheme. With oLSH, they reorder records as they arrive in streams, improving compression but reducing the flow processing rate. For our work, we use offline row reordering since the entire dataset is used.

Lexicographic Order

The sorting heuristic we use for this thesis is the lexicographic order. Previous studies have suggested that it is a good heuristic for the RunCount minimization problem [1, 29]. In the lexicographic order, the first component

where two tuples differ indicates tuple order. For example, consider the tuples (a,b,d,f,x) and (a,b,c,a,t); we find the first two components are identical, but the third differs, and so, tuple (a,b,c,a,t) precedes tuple (a,b,d,f,x).

Chapter 3

Statistics

Our work is founded both on bitmap indexes, and also on statistical measures and models. Some datasets are more suitable for building a multi-column bitmap index than others. Intuitively, datasets with columns that are closely related statistically could be good candidates for multi-column indexing. We assess this in Chapter 6. We can get an overall description of a dataset through statistical measurements. These measurements may allow us to classify datasets that may be good candidates for building small multi-column bitmap indices.

In Section 3.1, we review basic statistical measurements from which other important metrics can be derived. In Section 3.2 , we discuss the different data distributions that we use in our synthetic data experiments (see Chapter 5). In Section 3.3 and Section 3.4 we present more complex statistical measurements. We start with the statistical dispersion of histograms that

helps us describe how far a column's data distribution is from a uniform distribution. We also discuss various correlation measurements that can help us determine when a column pair is a possible good match considering space for a multi-column bitmap index.

3.1 Basic Statistical Measurements

Many of our statistical measurements are based on basic statistical measures. Among them is the arithmetic mean, which is a central tendency measure. The mean is a number that represents the tendency of some quantitative data to cluster around a particular value. Let i range over the rows of the column x, and x_i be the attribute value at the given row i. For the given column x, we can take the mean of its attribute values as follows:

$$\sum_{i=1}^{n} \frac{x_i}{n}.$$

Because most of our data is not quantitative, the arithmetic mean of the attribute values is of little use to us. In fact, most of our data is categorical—meaning that it has no numerical significance. As an example, suppose we tried to take the mean of customers' phone numbers in a table. The resulting mean would not give us significant information. Thus, instead of taking the mean of the attribute values, we use the mean frequency of the attribute values. We let f_i be the frequency for the distinct attribute value i. To

calculate the mean frequency, we add all f_i and divide by their number. By adding all f_i we obtain the number of records n and the number of distinct attribute values is the column cardinality L. Therefore we can calculate the mean frequency as:

$$\overline{f} = \frac{n}{L}. \tag{3.1}$$

This mean gives us no new statistical information from the dataset, as we can easily calculate it from n and L.

Another important basic statistical measurement is the variance. The variance is a quantitative measure that describes how far a set of numbers are spread out from one another. This description is based on how far the numbers lie from the mean (expected value) of the dataset. The following formula calculates the variance σ_x^2 for the column x as follows:

$$\sigma_x^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2.$$

We are mostly interested in the frequencies of the column values rather than on the column values themselves. Thus, we can calculate the variance of the frequencies as follows:

$$\sigma_f^2 = \frac{1}{L}\sum_{i=1}^{L}(f_i - \overline{f})^2.$$

The standard deviation of the column x, denoted as σ_x, also tells us how dispersed the column values are from the mean. By taking the square root of the variance, we express this measure in terms of the original units of x's

attribute values, instead of units squared.

The variance can be seen as a particular instance of *covariance* where the two columns in question are identical (see Equation 3.3). The covariance is a measure of how the values of two columns change together. It may also be used as a measure of correlation (see Section 3.4). The covariance can be calculated as follows:

$$\text{cov}(x, y) = \sum_{i=1}^{n} \frac{(x_i - \overline{x})(y_i - \overline{y})}{n}. \tag{3.2}$$

$$\sigma_x^2 = \text{cov}(x, x) = \sum_{i=1}^{n} \frac{(x_i - \overline{x})(x_i - \overline{x})}{n}. \tag{3.3}$$

Here, recall that n is the number of records and x_i is the attribute value for the i^{th} row in the attribute x. The mean of the attribute x is $\overline{x}$. The covariance may not be very intuitive for some, and therefore other measures may be derived from it.

3.2 Data Distributions

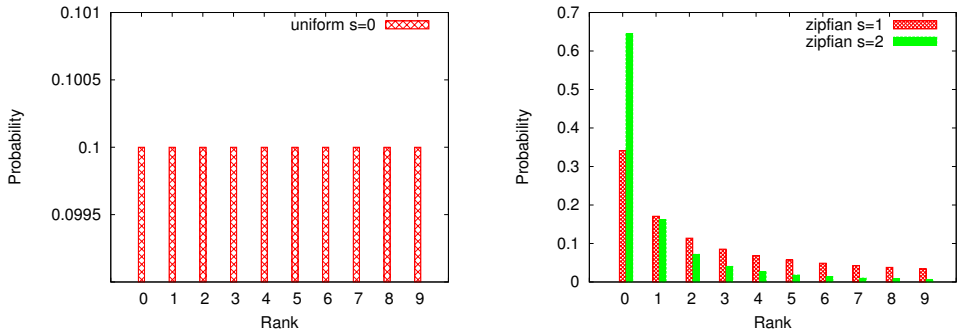
In this section, we review the distributions used in our synthetic datasets. A data distribution can be described through a probability function which describes the probability a random variable has on taking certain values. The data distribution can be observed from a table or column perspective. In a

table perspective, we look at the probability functions for different possible tuples, while in a column perspective we look at the probability function for the attribute values of the columns separately. In this section, we focus on the data distributions of columns and not of tables.

We first look at the uniform distribution as it is simple and widely used by other researchers [32]. In this distribution, the probability function assigns equal probability to all values a given column can take. For example, a uniform distribution function of a column with 10 distinct attribute values, gives each attribute value a probability of 1/10 (see Figure 3.1a).

We look at the Zipfian distribution as it is also used to approximate some realistic data [8, 14]. This distribution is characterized through a skew parameter referred as s. For $s > 0$, the probability function of this distribution is based on rank where the probability of an attribute value is inversely related to its rank. In a Zipfian distribution, the highest-ranking value has the highest probability of being repeated, the second highest-ranking value's probability is a fraction of the first's, $(1/2)^k$, the third's a fraction of the second's, $(2/3)^k$, and so on. We let L be the number of values in a column, k be the rank of an attribute value, i an index that ranges over the different ranks and s the skew. In a column, the probability of an attribute value with rank k, given an attribute cardinality of L, can be calculated with the following probability function:

$$\frac{k^{-s}}{\sum_{i=1}^{L} \frac{1}{i^s}}. \tag{3.4}$$



(a) (b)

Figure 3.1: Comparison between an uniform and Zipfian distributions for a column with 10 different attribute values. The uniform distribution can be seen as an extreme case of a Zipfian distribution where the skew is zero

A uniform distribution can be seen as an extreme case of a Zipfian distribution where the skew is zero. As the skew (s) is increased, the probability of the first ranking value increases, causing the probabilities of the remaining values to decrease accordingly. In Figure 3.1b we show a comparison between the Zipfian distribution for a column with 10 distinct attribute values with $s = 1$ and $s = 2$.

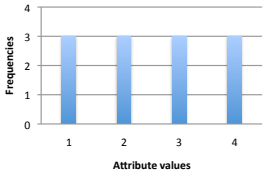
3.3 The Statistical Dispersion of Histograms

A histogram is a graphical representation of the distribution of attribute values in a given column. It consists of bars with fixed widths and with heights proportional to the frequency of each distinct attribute value.

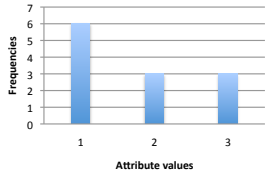
We use histograms to visualize the distribution of a column's attribute values. They can also help us determine when columns are correlated. The his-

c_0	c_1
1	1
2	2
3	3
4	1
1	1
2	2
3	3
4	1
1	1
2	2
3	3
4	1

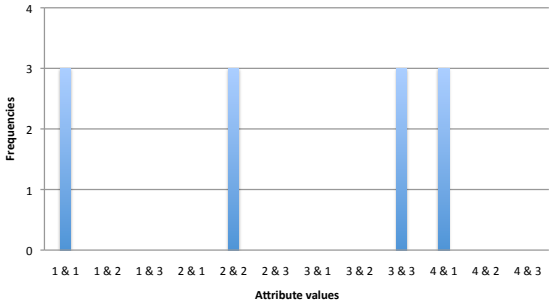
(a) Two columns



(b) Column c_0's histogram



(c) Column c_1's histogram



(d) Histogram of conjuncted attribute values for columns c_0 and c_1

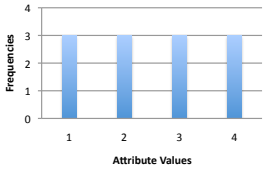
Figure 3.2: Histograms for two perfectly correlated columns

tograms for columns taken independently (as in Figure 3.2b and 3.2c), and taken together (as in Figure 3.2d), generally differ. The resulting histograms for columns taken together give us insight into what may be a highly correlated pair of columns. When columns are perfectly statistically correlated the joint-column's histogram has many absent attribute-value conjunctions, as shown in Figure 3.2d. In contrast, a histogram of perfectly statistically independent columns is shown in Figure 3.3d.

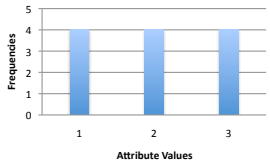
Columns with attribute values of equal frequencies have a uniform distribution and also a perfectly flat histogram. Yet, in general, many columns have attribute values that are more frequent than others. We use the coefficient

c_0	c_1
1	1
2	1
3	1
4	1
1	2
2	2
3	2
4	2
1	3
2	3
3	3
4	3

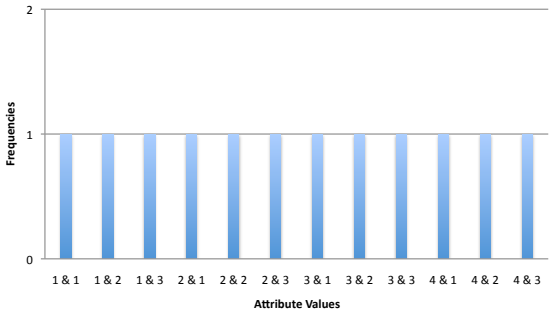
(a) Two columns



(b) Column c_0's histogram



(c) Column c_1's histogram



(d) Histogram of conjuncted attribute values for columns c_0 and c_1

Figure 3.3: Histograms for two perfectly independent columns

of variation, a measure of statistical dispersion, to determine how far a column's histogram is from a uniform histogram. The measure of statistical dispersion is related to, but different from, the skew parameter of a Zipfian distribution. For a given column x the coefficient of variation is $\sigma_f/\overline{f}$, where σ_f is the standard deviation of the frequencies and $\overline{f}$ is the average frequency. The coefficient of variation is a dimensionless measure. Thus, if we multiply all frequencies by a constant it remains the same. In other words, only the shape of the histogram matters, not its amplitude. However, the maximum coefficients of variation do vary with the cardinality of the columns. As an example, consider a histogram f having a large average $\bar{f}$ ($\bar{f} \gg 1$) and many

distinct values L ($L \gg 1$). The coefficient of variation is $\approx \frac{\sqrt{L}\bar{f}}{\bar{f}} = \sqrt{L}$. We expect to handle columns with widely varying cardinalities and thus look for a measure that treats high-cardinality and low-cardinality columns similarly. To overcome the setback of different coefficient bounds for different cardinalities, we introduce the normalized coefficient of variation:

$$\frac{\sigma_f}{\sqrt{L}} = \frac{\sqrt{\sum_{i=1}^{L}(f_i - \overline{f})^2}}{L\overline{f}}.$$

3.4 Correlation

Correlation is a probabilistic dependence between two attributes or columns. It is often used to indicate the possibility of underlying causal or mechanistic relationships, but it alone cannot determine their existence. Pearson's product-moment coefficient (ρ_{xy}) is often used to measure the degree of correlation between 2 columns. Given the columns x and y, we can calculate it as:

$$\rho_{xy} = \frac{\mathrm{cov}(x, y)}{\sigma_x \sigma_y}.$$

Dividing by the standard deviations σ_x and σ_y has a normalizing effect on the covariance.

Pearson's correlation measures the degree of linear dependence between columns and varies from -1 to 1. A positive coefficient indicates a direct relationship between the columns while a negative coefficient indicates an inverse rela-

tionship between the columns. Coefficients closer to -1 or 1 indicate strong linear relationships while coefficients closer to 0 indicate weak, or no linear relationship. While Pearson's correlation can indicate the strength of a possible linear dependence between columns, it can fail to indicate the existence of non-linear relationships. It is also limited to numerical data and cannot indicate correlation when there is categorical data in the columns.

Other correlation measures exist that are not limited to linear relationships or numerical data. One such measure is Pearson's chi-square test. In this test, a null hypothesis of independence between the columns is asserted with the expected theoretical frequencies one would see if the columns were independent. An alternative hypothesis corresponds to the probability that the columns are indeed correlated. The chi-square statistic tells us how far the observed frequency is from the theoretical frequency and can help us determine the likelihood that the columns are dependent. Before giving the chi-square formula, the following notation and definitions must be established:

1. Recall that the total number of records is n.
2. We denote the cardinalities of the two participating columns as L_1 and L_2.
3. We denote the two columns as x and y.
4. We let i and j range over the distinct attribute values of columns x and y, respectively.

5. We denote the frequency of the i^{th} attribute value of the first column as $n_i^{(c_0)}$ and the frequency of the j^{th} attribute value of the second column as $n_j^{(c_1)}$.

6. We denote the observed frequency of the record containing both attribute values i and j as n_{ij}.

7. We consider that the expected (theoretical) frequency of the record containing both attribute values i and j is $n_i^{(c_0)} n_j^{(c_1)}/n$.

With the notation defined we can calculate the chi-square statistic, denoted as χ^2, as follows:

$$\chi^2 = \sum_{i=1}^{L_1} \sum_{j=1}^{L_2} \frac{(n_{ij} - n_i^{(c_0)} n_j^{(c_1)}/n)^2}{n_i^{(c_0)} n_j^{(c_1)}/n}. \tag{3.5}$$

The Mean square contingency is related to the χ^2 test, and was used when Ilyas et al. [15] implemented the tool CORDS (CORrelation Detection via Sampling). CORDS uses the mean square contingency to determine the degree of correlation between two columns of a table (see Equation (3.5)). The mean square contingency is a normalized version of the χ^2 test that is defined as:

$$\phi^2 = \frac{\chi^2}{n(L_{\min} - 1)}. \tag{3.6}$$

Here, $n(L_{\min} - 1)$ is the normalizing factor, where n is the number of records in the sample and $L_{\min}$ is the minimum cardinality of the two variables or

attributes compared. Moreover, after substituting and simplifying we can express the mean square contingency as follows:

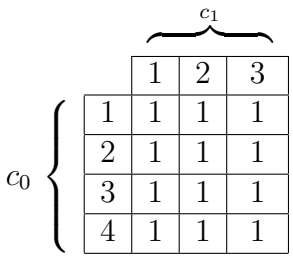
$$\phi^2 = \frac{1}{d-1} \sum_{i=1}^{L_1} \sum_{j=1}^{L_2} \frac{(\pi_{ij} - \pi_i^{(c_0)} \pi_j^{(c_1)})^2}{\pi_i^{(c_0)} \pi_j^{(c_1)}}.$$

Here, the fraction of records where the attribute values for columns i and j are the same is denoted as π_{ij}. This fraction, π_{ij}, is also related to the frequency n_{ij} as $\pi_{ij} = n_{ij}/n$. The fractions of records where the attribute values for columns i and j alone are the same are $\pi_i^{(c_0)}$ and $\pi_j^{(c_1)}$ respectively. These fractions are also related to the frequencies $n_i^{(c_0)}$ and $n_j^{(c_1)}$. Their relationship can be expressed as $\pi_i^{(c_0)} = n_i^{c_0}/n$ and $\pi_j^{(c_1)} = n_j^{c_1}/n$.

In Figure 3.4 we illustrate a sample dataset with perfectly uncorrelated data along with its respective contingency table. Each cell in the contingency table represents the frequency n_{ij} of one of all possible attribute values combinations between columns c_0 and c_1. The frequency of column c_0's attribute value i, $n_i^{(c_0)}$, can be calculated by adding all frequencies where the attribute i is present. Likewise, the frequency of column c_1's attribute value j, $n_j^{(c_1)}$, is obtained by adding all frequencies where j is present. The number of records, n, in this sample dataset is 12. After computing with the corresponding frequencies, we get $\chi^2 = 0$, further confirming that the dataset is independent. For the mean square contingency, following computation with Equation 3.6, we obtain $\phi^2 = 0$. In Figure 3.5, we illustrate a contrasting dataset with perfectly correlated data. For this dataset, we obtain $\chi^2 = 24$, then after

c_0	c_1
1	1
2	1
3	1
4	1
1	2
2	2
3	2
4	2
1	3
2	3
3	3
4	3

(a) sample dataset



	1	2	3
1	1	1	1
2	1	1	1
3	1	1	1
4	1	1	1

$\chi^2 = 0$
$\phi^2 = 0$

(b) contingency table

Figure 3.4: Perfectly uncorrelated sample dataset with contingency table of possible attribute value combinations for columns c_0 and c_1.

substituting in Equation 3.6, the mean square contingency $\phi^2 = \frac{\chi^2}{12(3-1)} = 1$. This example further confirms that if the sample dataset is perfectly correlated $\phi^2 = 1$.

3.4.1 Cardinality-Based Measures of Correlation

Consider two columns in a table. The product of both columns' cardinalities represents the maximum number of combinations of attribute values that can be generated. If the cocardinality (the cardinality of the columns taken jointly) is significantly smaller than the product of both columns' cardinalities, there is a strong indication that both columns are correlated. Considering the cardinalities of the participating columns we provide our own

c_0	c_1
1	1
1	1
1	1
2	3
2	3
2	3
3	1
3	1
3	1
4	2
4	2
4	2

(a) sample dataset

		c_1		
		1	2	3
c_0	1	3	0	0
	2	0	0	3
	3	3	0	0
	4	0	3	0

$\chi^2 = 24$
$\phi^2 = 1$

(b) contingency table

Figure 3.5: Perfectly correlated sample dataset with contingency table of possible attribute value combinations for columns c_0 and c_1

.

correlation measures.

One measure, ϕ_p uses a ratio of products to normalize the correlation between 0 and 1. Let L_i be the cardinality of the i^{th} column and $L_{1,2,\ldots,c}$ be the joint cardinality of columns 1 through c. With the aforementioned, we can calculate a cardinality-based correlation, ϕ_p, as follows:

$$\phi_p = 1 - \frac{\prod_{i=1}^{c}(L_{1,2,\ldots,c} - L_i)}{\prod_{i=1}^{c}(L_1 L_2 L_3 \cdots L_c - L_i)}.$$

Our other correlation measure, ϕ_m, uses the same variables used in ϕ_p's formula with addition to the maximum cardinality denoted as $L_{\max}$. We calculate this cardinality-based correlation as follows:

$$\phi_m = 1 - \frac{L_{1,2,\ldots,c} - L_{\max}}{L_1 L_2 L_3 \cdots L_c - L_{\max}}.$$

Both cardinality-based correlation measures range from 0 to 1 because

$$\begin{aligned} L_{\max} &\leq L_{1,2,\ldots,c} \\ &\leq L_1 L_2 L_3 \cdots L_c. \end{aligned}$$

Moreover, these measures are 1 when we have that $L_{\max} = L_{1,2,\ldots,c}$ and they are 0 when $L_{1,2,\ldots,c} = L_1 L_2 L_3 \cdots L_c$.

3.4.2 Coentropy

Paradies et al. [22] propose yet another measure to determine correlation between columns. They use an entropy-based measure to determine the degree of correlation. The entropy is a measure used in statistics that was introduced by Shannon in information theory [25].

For a column, the frequency with which each attribute value occurs determines the information about its distribution. The degree of dispersion of a column's distribution is characterized in the entropy of the column. For a column x, let f_i be the frequency of the distinct attribute value i. We can

calculate the entropy, denoted H_x, of the column x as follows:

$$H_x = -\sum_{i=1}^{L} \frac{f_i}{n} \log_2 \left(\frac{f_i}{n} \right).$$

The entropy measure can be used for the conjunctions of attribute values of two columns. The combined entropy, the coentropy, of the two columns also gives us information of how correlated the two columns are. For columns x and y, let i and j range over their respective distinct attribute values. Let f_{ij} be the frequency of the attribute values i and j considered together. We can calculate the coentropy, denoted H_{xy}, as follows:

$$H_{xy} = -\sum_{i=1}^{L_1} \sum_{j=1}^{L_2} \frac{f_{ij}}{n} \log_2 \left(\frac{f_{ij}}{n} \right).$$

The maximum coentropy between two columns x and y is equal to the sum of the entropies of x and y; in other words, $H_{xy} \leq H_x + H_y$. When the columns are perfectly independent, $H_{xy} = H_x + H_y$.

The minimum coentropy of two columns x and y is the maximum of their individual entropies, $\max(H_x, H_y)$. The minimum occurs when these columns are perfectly correlated. In such a case, their coentropy equals the entropy of one of the columns, as the other column's information can be directly determined from the first.

The maximum coentropy and minimum coentropy vary with the attribute cardinality of the columns. To compare results of different pairs of columns

c_0	c_1
1	1
2	1
3	1
4	1
1	2
2	2
3	2
4	2
1	3
2	3
3	3
4	3

(a) sample dataset

$$\begin{aligned} H_{c_0} &= 2.0 \\ H_{c_1} &= 1.58 \\ H_{c_0 c_1} &= 3.58 \end{aligned}$$

Thus the normalized coentropy is 1.0

(b) coentropy example

Figure 3.6: Perfectly uncorrelated sample dataset with calculated entropies for columns c_0, c_1 and joint c_0 and c_1.

with various cardinalities, we propose to normalize the coentropy between 0 and 1 as follows:

$$\frac{H_{xy} - \max(H_x, H_y)}{(H_x + H_y) - \max(H_x, H_y)} = \frac{H_{xy} - \max(H_x, H_y)}{\min(H_x, H_y)}.$$

In Figure 3.6 we illustrate a sample dataset with perfectly uncorrelated data along with its respective normalized coentropy. Notice that unlike the mean square contingency where perfectly correlated data lead to $\phi_{msc} = 0$, the normalized coentropy is 1. In Figure 3.7, we show the other extreme case of a sample dataset with perfectly correlated data. In this case, the normalized coentropy is 0.

c_0	c_1
1	1
1	1
1	1
2	3
2	3
2	3
3	1
3	1
3	1
4	2
4	2
4	2

(a) sample dataset

$$H_{c_0} = 2.0$$
$$H_{c_1} = 1.5$$
$$H_{c_0 c_1} = 2.0$$

Thus the normalized coentropy is 0.0

(b) coentropy example

Figure 3.7: perfectly correlated sample dataset with calculated entropies for columns c_0, c_1 and joint c_0 and c_1.

3.5 Summary

We presented basic statistical measurements that can be used to describe column pairs. We also defined the following measurements of correlation: coentropy, mean square contingency and our novel cardinality-based measures which we later use in Chapter 6 to try to predict whether a multi-column or single-column bitmap index is smaller for a particular dataset.

These measurements may help us determine when a column pair is a good candidate for a multi-column bitmap index in terms of size. We presented both Zipfian and uniform distributions that we use in Chapter 5 to analyze the effect that varying statistical characteristics of datasets has on size of bitmap indices.

Chapter 4

Multi-column Bitmap Indices

Our primary goal is to analyze the applicability and effectiveness of multi-column bitmap indices when compared to single-column bitmap indices. Part of the effectiveness of a multi-column bitmap index lies in the disk space it occupies. We focus only on the storage requirements for multi-column and single-column bitmap indices. By the end of this chapter we determine a theoretical case when a multi-column index is smaller than a single-column index, and demonstrate that statistical measurements do not hold enough information.

In Section 4.1, we introduce multi-column bitmap indices and give a brief overview of their advantages and disadvantages in query performance. In the following sections we examine two important factors that determine the size of multi-column bitmap indices, namely, the number of bitmaps and RunCount. Furthermore, in Section 4.2 we describe how the number of

bitmaps of the resulting index differs in a multi-column and single-column bitmap index. We also give bounds for the number of bitmaps of multi-column and single-column bitmap indices. Section 4.3 covers RunCount and provides an analysis of different RunCounts for multi-column bitmap indices. In Section 4.4 we give bounds for the RunCount of multi-column bitmap indices. The last section combines formulas for the RunCount and number of bitmaps of a multi-column bitmap index with contingency tables introduced in Section 3.4. Finally, we determine theoretically when we can know that a multi-column bitmap index is smaller than a single-column bitmap index in terms of RunCount given a contingency table.

4.1 Multi-column Bitmap Indices

In a multi-column bitmap index, the attribute values of a pair of columns are considered as a single attribute value (see Figure 4.1). A multi-column index is especially effective for queries that involve all columns that participate in it. Queries on a single column or subset of the columns contained in the index must be translated to a series of queries that can be answered by the multi-column bitmap index. For these queries, all possible attribute value combinations for the missing components are considered.

Out of all the combinations of attribute values for missing components, there may be some combinations that do not have an index associated to them. Therefore, the additional overhead of using a multi-column index for alter-

Student Id	Gender
1001	M
1002	F
1003	F
1004	M
1005	M
1006	F

(a)

1001M	1002F	1003F	1004M	1005M	1006F
1	0	0	0	0	0
0	1	0	0	0	0
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	0	1	0
0	0	0	0	0	1

(b)

1001	1002	1003	1004	1005	1006	M	F
1	0	0	0	0	0	1	0
0	1	0	0	0	0	0	1
0	0	1	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	0	0	1	0	1	0
0	0	0	0	0	1	0	1

(c)

Figure 4.1: Multi-column (b) and single-column (c) bitmap indices for a table with a high cardinality attribute (Student Id) and a low cardinality attribute (Gender). Notice that the single-column index has two more bitmaps than the multi-column index.

nate queries depends on the cardinalities of the missing components and the bitmaps associated to their conjunctions. For each column i, let L_i be the attribute cardinality, and let S be a subset of the c columns. This subset contains columns that participate in the multi-column index but do not have a value specified within a query. The original query may be translated to $\prod_{i \in S} L_i$ queries that our multi-column bitmap index can answer. The number of bitmaps that need to be scanned is at most $\prod_{i \in S} L_i$ since some combinations of attribute values are not present.

As an example, consider the table in Figure 4.1a with a corresponding multi-column bitmap index in Figure 4.1b. Suppose that we want to find student id 1001. To do so, we would need to query bitmaps for values 1001M and 1001F. In this case, the overhead is only of one more query when compared to a single-column index. Because bitmap 1001F does not exist, only the bitmap for 1001M would be scanned. In this case, the overhead is minimal as the number of bitmaps that are scanned by both indices is the same. However, if we queried for all students with female gender, we would potentially scan bitmaps associated to the following values: 1001F, 1002F, 1003F, 1004F, 1005F and 1006F. In our case, only three bitmaps exist for the following values: 1002F, 1003F and 1006F. Unlike a single bitmap scan in a single-column bitmap index, three bitmaps would need to be scanned with a multi-column bitmap index.

4.2 Number of Bitmaps

When using a bitmap compression technique such as EWAH, there is a per-bitmap space overhead. Thus, we examine the number of bitmaps of bitmap indices in isolation.

In some scenarios, the number of bitmaps is greater in a multi-column index and in others it is greater in a single-column index. For a bitmap index λ, let m_λ be the number of bitmaps. When we have a single-column bitmap index (α), we have that $m_\alpha = \sum_{i=1}^{c} L_i$. In contrast, for a multi-column index (β), m_β is in the interval $[L_{\max}, \min(\prod_{i=1}^{c} L_i, n)]$, where $L_{\max} = \max_i L_i$. The lower bound $L_{\max}$ represents the case where the number of bitmaps can be determined solely by the cardinality of a single column. Such is the case when the columns are duplicates. The upper bound $\min(\prod_{i=1}^{c} L_i, n)$ represents the case where a multi-column index produces all possible conjunctions of attribute values among the participating columns or the number of bitmaps has reached the number of records, n. To illustrate with an example, consider a 2-column table with 50 records, and an attribute cardinality of 10 per column. Using the formula, we obtain $m_\alpha = 20$, and m_β bounded within $[10, 50]$. If the table's columns when considered together form less than 20 distinct conjunctions, then $m_\alpha > m_\beta$; otherwise, $m_\alpha \leq m_\beta$.

4.3 RunCount

The total number of runs in a multi-column and single-column bitmap index may also differ. As mentioned previously, a run is a sequence of repeating values. For example, in the sequence *aaabba*; *aaa*, *bb*, and *a* form 3 separate runs. The total number of runs or RunCount of the *aaabba* sequence is 3. We determine the total number of runs for bitmaps, bitmap indices, columns and tables using a RunCount function.

To further elucidate, in Figure 4.2 we illustrate the RunCount function for a column, table, bitmap, column's bitmaps and bitmap index. Notice that the RunCount of the table, RunCount(T), is equal to sum of the RunCounts of its columns, $\sum_{i=0}^{c}$ RunCount(c_i). A bitmap index built considering both columns separately leads to a single-column bitmap index. The RunCount of the single-column bitmap index, denoted as RunCount(α), is equal to the sum of the RunCounts of its individual bitmaps, $\sum_{i \in m_\alpha}$ RunCount(b_i). Likewise, for a multi-column bitmap index we calculate RunCount(u), where u is a column of unique identifiers. Each identifier maps to a distinct attribute value conjunction for table T. Building a bitmap index of column u produces a multi-column bitmap index (see Figure 4.2b). Later on, (Section 4.4) we shall demonstrate that the RunCount of the multi-column bitmap index, denoted as RunCount(β), can be calculated directly from RunCount(u).

Multi-column and single-column indices generally benefit from sorted data,

Table T | Single-column Bitmap Index α

c_0	c_1	b_0	b_1	b_2	b_3	b_4	b_5
0	3	1	0	0	1	0	0
0	4	1	0	0	0	1	0
0	5	1	0	0	0	0	1
1	3	0	1	0	1	0	0
1	3	0	1	0	1	0	0
1	3	0	1	0	1	0	0
2	3	0	0	1	1	0	0
2	5	0	0	1	0	0	1
2	5	0	0	1	0	0	1

RunCount(c_0)	3
RunCount(c_1)	5
RunCount(b_0)	2
RunCount(b_1)	3
RunCount(T)	8
RunCount(α)	18

(a) Calculated RunCount for our table T, its columns c_0 and c_1, and its single-column bitmap index α

Column u | Multi-column Bitmap Index β

$c_0 \wedge c_1 \rightarrow u$	$b_{0\wedge3}$	$b_{0\wedge4}$	$b_{0\wedge5}$	$b_{1\wedge3}$	$b_{2\wedge3}$	$b_{2\wedge5}$
$0 \wedge 3 \rightarrow 0$	1	0	0	0	0	0
$0 \wedge 4 \rightarrow 1$	0	1	0	0	0	0
$0 \wedge 5 \rightarrow 2$	0	0	1	0	0	0
$1 \wedge 3 \rightarrow 3$	0	0	0	1	0	0
$1 \wedge 3 \rightarrow 3$	0	0	0	1	0	0
$1 \wedge 3 \rightarrow 3$	0	0	0	1	0	0
$2 \wedge 3 \rightarrow 4$	0	0	0	0	1	0
$2 \wedge 5 \rightarrow 5$	0	0	0	0	0	1
$2 \wedge 5 \rightarrow 5$	0	0	0	0	0	1

RunCount(u)	6
RunCount(β)	16

(b) Calculated RunCounts for our column of unique conjunctions u, and a multi-column bitmap index β for table T

Figure 4.2: Illustration of function RunCount for different elements such as table, column, bitmap, and bitmap index.

so we examine their total number of runs when they are built on sorted tables. There are different sorting heuristics but for our purposes we focus only on the lexicographic order.

RunCount$_{32}$

Runs may also be counted as groups of elements with a number of them equal to a multiple. For instance, if we calculate RunCount$_4$, each run constitutes a quadruplet of the same element. Thus, given a sequence, such as 0000 0000 0011 0011 1111 1111, we have one run at the beginning (two groups of four zeroes) and one run at the end (two group of four ones). The two subsequences of 0011 do not count as a single run because they are not a quadruples of the same element. Consequently, RunCount$_4$ for the given sequence is 4. Following this pattern, we define RunCount$_{32}$ as the total number of runs, where each element is a 32-plet of the same element. We are specifically interested in RunCount$_{32}$ for bitmap indices, as it more accurately portrays the type of runs the EWAH compression scheme uses.

4.4 RunCount Bounds After Sorting

In this section, we compute simple a priori bounds for the RunCount of multiple-column and single-column bitmap indices. For a multi-column bitmap index β, calculating RunCount(β) is similar to calculating the RunCount of a bitmap index built on a sorted isolated column. Suppose we take

all the attribute-value conjunctions and assign each of them a unique identifier. After doing so, we have a single column of unique conjunction identifiers; we denote this column as u. When u is sorted, RunCount(u) is equal to its attribute cardinality L_u. We often refer to L_u as the cocardinality–that is the cardinality when considering the columns of a table in conjunction. A multi-column bitmap index built on such a column has a total of $3(L_u-2)+4$ runs: bitmaps corresponding to attribute values that are neither first nor last have 3 runs each. Consider the case where $L_u > 1$. There are L_u-2 such bitmaps, and thus in all they contribute $3(L_u-2)$ runs to RunCount(β). The term of 4 accounts for the runs in the first and last bitmaps, which always have 2 runs. Hence, a bitmap index built on u has RunCount(β) $= 3(L_u-2)+4$. Moreover, when building a basic multi-column bitmap index β, L_u and m_β are the same making RunCount(β) $= 3(m_\beta-2)+4$. In Figure 4.2, we illustrate an example of a sorted 2-column table where $L_u = m_\beta = 6$ and RunCount(β) $= 3(6-2)+4 = 16$.

The RunCount of a single-column bitmap index, RunCount(α), is not as easily determined as that of a multi-column bitmap index. Some of the bitmaps comprising it may have more than 3 runs. Unlike in a single column, when sorting multiple columns, all columns may not be optimally sorted. In fact, the first column considered is the only one that can be guaranteed to be optimally sorted.

Lemma 4.1. *For α built on a sorted 2-column table, we bound* RunCount*(α) as being within* $[3L_1+3L_2-4\,,\ 3L_1+L_2+2n-4]$ *where L_1 is the cardinality*

of the first column, L_2 is the attribute cardinality of the second column and n is the number of records.

Proof. The lower bound represents a best case scenario when α is built on two columns c_1 and c_2 that are optimally sorted. Because optimally sorted columns have a RunCount equal to their cardinality, we have that $\text{RunCount}(c_1) = L_1$ and $\text{RunCount}(c_2) = L_2$. Let α_i be the set of bitmaps associated to the optimally sorted column c_i. We can calculate the RunCount for c_i's bitmaps as $\text{RunCount}(\alpha_i) = 3(L_i - 2) + 4$. Therefore, after simplifying, we have the lower bound:

$$\text{RunCount}(\alpha) = \text{RunCount}(\alpha_1) + \text{RunCount}(\alpha_2) = 3L_1 + 3L_2 - 4.$$

We now prove the upper bound, which represents the worst case scenario where c_1 is optimally sorted but c_2 has the maximum RunCount possible. For c_1's bitmaps, α_1, we have $\text{RunCount}(\alpha_1) = 3(L_1 - 2) + 4 = 3L_1 - 2$. Because c_2 is not optimally sorted, we present a more generic formula that allows calculating the total RunCount of a column's associated bitmaps. We can calculate the total RunCount for the associated bitmaps of a given column c with the following monotonically increasing function:

$$\text{RunCount}(\alpha_c) = L_c + 2(\text{RunCount}(c) - 1). \tag{4.1}$$

There are $\text{RunCount}(c) - 1$ transitions between values in column c, and

each transition in c creates a transition from 0 to 1 in one bitmap, and a transition from 1 to 0 in another bitmap. Thus, we can calculate the total RunCount of the associated bitmaps of c_2 as $\text{RunCount}(\alpha_2) = L_2 + 2(\text{RunCount}(c_2) - 1)$. We maximize $\text{RunCount}(\alpha_2)$, by maximizing $L_2 + 2(\text{RunCount}(c_2) - 1)$. The maximum is reached when C_2 is as disordered as possible—that is when $\text{RunCount}(c_2) = n$—which consequently maximizes α_2 as $L_2 + 2(n-1)$.

An example of a case that maximizes $\text{RunCount}(c_2)$, and therefore also maximizes $\text{RunCount}(\alpha_2)$, is illustrated in Figure 4.3a. The RunCount of the bitmaps associated to the second column of the bitmap index can be calculated as $L_2 + 2n - 2$. We can think of this as having L_2 runs for the first row and 2 runs for each row we add to the table thereafter. More specifically, a run of 1s is started and a run of 0s is also started where a run of 1s ends (see Figure 4.3b). By adding up the minimum $\text{RunCount}(\alpha_1)$ and the maximum $\text{RunCount}(\alpha_2)$, we obtain the upper bound for $\text{RunCount}(\alpha)$:

$$(3L_1 - 2) + L_2 + 2(n-1) = 3L_1 + L_2 + 2n - 4.$$

□

1	1
1	2
1	3
1	4
2	1
2	2
2	3
2	4
3	1
3	2
3	3
3	4

(a) n runs for column c_2

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

(b) $L_2 + 2(n-1)$ runs in α_2

Figure 4.3: Column RunCount for the second column, c_2, of table (a) with its associated bitmaps, α_2, and its respective RunCount.

4.5 Contingency Tables and RunCount

For a given table, a contingency table tells us how correlated two columns are and how many bitmaps a multi-column or single-column bitmap index will have (see Section 3.4). If a contingency table has a mean square contingency of 1, the RunCount of the multi-column bitmap index will be smaller than that of the single-column bitmap index. That is because the two columns joined would be duplicates in terms of RunCount, and when joined the resulting column is an identical copy of either of the individual columns. Consequently, while α stores the two duplicate columns, β stores just one. Yet, for contingency tables having a mean square contingency different from

1, the information from a contingency table is not sufficient to determine when a table is or is not a good candidate for a multi-column bitmap index because it lacks information on the row ordering of the table. Consider the example in Figure 4.4 where two datasets with different row orderings but with the same contingency table are illustrated. One of the datasets leads to RunCount(α) > RunCount(β) while the other leads to RunCount(β) > RunCount(α).

c_0	c_1
1	2
1	3
1	1
2	1
2	2
2	3
3	3
3	2

$\text{RC}(\alpha) = 20$
$\text{RC}(\beta) = 22$

(a) dataset a

c_0	c_1
1	1
1	2
1	3
2	1
2	2
2	3
3	2
3	3

$\text{RC}(\alpha) = 24$
$\text{RC}(\beta) = 22$

(b) dataset b

c_0 \ c_1	1	2	3
1	1	1	1
2	1	1	1
3	0	1	1

$\chi^2 \approx 0.889$
$\phi^2 \approx 0.0556$

(c) contingency table

Figure 4.4: Two datasets (a) and (b) with the same contingency table shown in (c). Both datasets lead to different RunCounts for a single-column bitmap index (α) and multi-column bitmap index (β). The respective RunCounts are abbreviated as RC in the illustrations.

The information of a contingency table is sufficient to determine when a multi-column bitmap index will be smaller than a single-column bitmap index in terms of RunCount given two assumptions. The first is that we only consider 2-column tables, and the second is that we assume that the tables are

sorted lexicographically. With these two assumptions, the following lemma formulates how to calculate RunCount(α) and RunCount(β):

Lemma 4.2. *Given a 2-column table that has been sorted lexicographically, we can calculate* RunCount*(α) and* RunCount*(β) using the cocardinality L_u, and the cardinalities of the first and second columns L_1 and L_2 respectively, as follows:*

$$\text{RunCount}(\beta) = L_u + 2(L_u - 1) = 3L_u - 2,$$

$$\text{RunCount}(\alpha) = 3L_1 - 2 + L_2 + 2(L_u - 1).$$

Proof. We first demonstrate the equation for RunCount(β). As previously shown, the RunCount of the generated bitmaps for a column c can be calculated as $L_c + 2(\text{RunCount}(c) - 1)$ (see Equation 4.1). The number of bitmaps of a multi-column bitmap index is equal to the total number of distinct rows L_u. When sorted lexicographically, all distinct rows of a multi-column bitmap index are consecutive, effectively making $\text{RunCount}(u) = L_u$. Thus, considering that a multi-column bitmap index is essentially the set of bitmaps for a column of distinct tuples u, we have that $\text{RunCount}(\beta) = L_u + 2(L_u - 1) = 3L_u - 2$.

We calculate RunCount(α) by summing up the RunCounts of the bitmaps of the first and second columns of the table. In a lexicographically sorted table, the RunCount of the first column, RunCount(c_1), is equal to

L_1. As a result, the total RunCount of the bitmaps of the first column, RunCount(α_1), is equal to $3L_1 - 2$. The second column of a lexicographically sorted table has L_u runs since distinct tuples are also kept adjacent. Therefore, RunCount(c_2) = L_u and RunCount(α_2) = L_2+2(L_u−1). Having RunCount(α_1) and RunCount(α_2), we have that RunCount(α) = $3L_1 - 2 + L_2 + 2(L_u - 1)$. □

4.6 Summary

To summarize this chapter, we have explored theoretically the advantages and disadvantages multi-column bitmap indices can give us in query performance over single-column bitmap indices. We also explored two important factors that determine the size of multi-column and single-column bitmap indices, namely, number of bitmaps, and RunCount. We gave bounds for these two factors for both types of indices. Last, in Lemma 4.2, we combined formulas for the RunCount and number of bitmaps of a multi-column bitmap index with contingency tables. We were able to determine when a multi-column bitmap index would be smaller than a single-column bitmap index in terms of RunCount given the following two assumptions:

- The table had only two columns.
- The table was sorted using the lexicographic order.

We demonstrated that, in most cases, the information of the contingency table is insufficient for determining size for multi-column or single-column bitmap indices. We realized that row-order matters greatly and, because our statistical measures of correlation are reflections of a contingency table, they can merely help in predicting the number of bitmaps. Knowing the number of bitmaps for a single-column or multi-column bitmap index is not enough for determining size.

Chapter 5

Synthetic Datasets

A starting point to investigate the storage requirements of multi-column versus single-column bitmap indices is to analyze the behavior of their size as we vary the statistical parameters of datasets from which the indices are built. We do so through experiments performed on synthetic data where we vary each statistical parameter.

To perform the experiments, we first generate tables with certain characteristics defined by statistical parameters. We sort the table lexicographically or shuffle it to compare the improvement sorting can have on both types of indices. We build multi-column bitmap indices by merging the columns of the table into one column and single-column bitmap indices by indexing both columns of the table separately.

In the results of the experiments, we present cases where multi-column bitmap indices are smaller than single column bitmap indices and vice versa. We also

show graphically how a particular characteristic either reduces or increases the size of a multi-column bitmap index with respect to a single-column bitmap index.

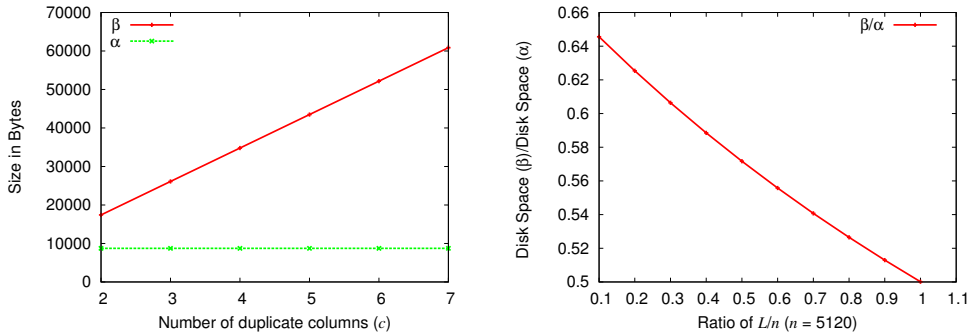
This chapter is organized as follows: To begin exploring cases where a multi-column index, β, is smaller than a single-column index, α, we introduce extreme cases where multi-column bitmap indices are always smaller than single-column bitmap indices in Section 5.1. We then continue our search for scenarios where multi-column indices are smaller than single-column indices by analyzing how different aspects of datasets affect multi-column and single-column bitmap index size. In Section 5.2, we vary the number of columns of synthetically generated datasets. We build multi-column and single-column bitmap indices on those synthetic datasets and explore the relationship between size of the bitmap index, and the number of columns of the dataset. We also vary the statistical parameters used to generate uniform distributions and compare the resulting bitmap indices. In Section 5.3, we explore the behavior of sizes of bitmap indices as we gradually turn an originally uniform distribution into a Zipfian distribution.

5.1 Extreme Cases

We first try experiments with extreme cases, and one such case is when the relevant indexed columns are duplicates of one another. In this case, the size of the multi-column index remains constant as we add more columns. In

contrast, the size of the single-column bitmap index would be approximately c times greater, where c is the number of indexed columns (See Figure 5.1a) A second extreme case is when we have a column with no repeating elements, merged to another column. An example of this could be a first column representing a unique student identifier, and an arbitrary second column that may be as varied as another identifier, or of few distinct values such as gender. By creating a 2-column bitmap index, we omit the cost of creating bit arrays for the second column (See Figure 4.1). With all elements being different on the first column, there are no new elements formed when the two columns are considered together into one. Therefore, the size of the 2-column index is always the same, no matter what the characteristics of the second column are. In contrast, a single column index on the same columns is always bigger. The amount of space we save by creating a multi-column index depends on the cardinality of the second column. The greater the cardinality of the second column, the more space that is saved by creating a multi-column index. We consider the cardinality of a column relative to the total number of records, and we refer to this ratio, L/n, as the *cardinality ratio* of a column.

In Figure 5.1b we plot disk space used against the cardinality ratio of the second column. The table consists of 5 120 rows and 2-columns, where the first column has a cardinality equal to the number of records. As we increase the cardinality ratio of the second column we get a multi-column index that is a smaller fraction of an equivalent single-column bitmap index. When the



(a) Duplicate Columns

(b) Unique first column with a second column of varied cardinality

Figure 5.1: Comparison between Multi-column bitmap indices and single-column bitmap indices in extreme cases.

second column is also unique, the single-column bitmap index, α, is twice the size of an equivalent multi-column bitmap index, β. We can generalize this for c columns as α is c times greater in size than β.

To conclude, in the extreme cases of having duplicate columns or a unique column, building a multi-column bitmap index saves disk space. These two particular cases are part of the more general case of perfectly correlated columns where mean square contingency is one (see Section 4.5).

5.2 Uniform Cases

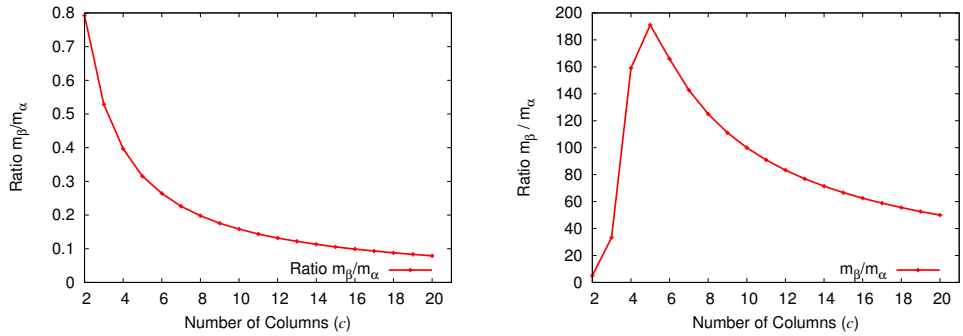
To determine the behavior of the bitmap index size in simpler scenarios, we build our indices on data that follows a uniform distribution (see Section 3.2). In this distribution each attribute value of a column has equal probability of being part of a tuple in a table. Uniform distributions are commonly used

to model database tables. For instance, in the table of the Star Schema Benchmark [19] seven out of 17 columns have uniform distributions.

The graphs generated from our results did not seem to show the effect of much noise, and therefore we generated the data only once for each of our datasets. To build a multi-column index we merged the columns into one column, which may or may not be sorted afterwards. In the graphs, we illustrate the behavior of different characteristics of multi-column bitmap indices over those of single-column bitmap indices. We also consider the two cases of unsorted and lexicographically sorted tables for building multi-column and single-column bitmap indices.

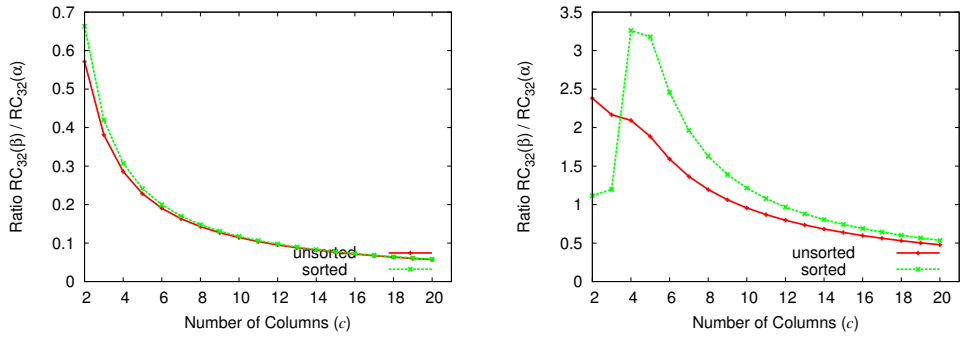
5.2.1 Varying Columns

We first look at the behavior as we increase the number of columns used to build a multi-column index and a single-column index. The table in this synthetic experiment has 10 000 rows, and is varied in the number of columns from 2 to 20. All columns have nearly the same cardinality. We explore the behavior of the ratio m_β/m_α for a high cardinality of 6 320 and a low cardinality of 10. As shown in Figure 5.2a, when the columns have a high cardinality of 6 320 each, m_β is always smaller than m_α and m_β/m_α further decreases as more columns are added to the table. While there are $6\,320^2$ possible conjunctions, m_β is also bounded by the number of records, thus m_β remains at 10 000 as the number of columns is increased. In contrast, m_α is not limited by the number of records alone, and increases by 6 320 bitmaps



(a) cardinality per column ≈ 6320 (b) cardinality per column ≈ 10

Figure 5.2: Ratio of number of bitmaps of a multi-column index with respect to a single-column index. A general decrease in this ratio can be observed as we increase the number of columns asymptotically. Sorting does not affect the number of bitmaps making the ratios the same for both sorted and unsorted case, thus we display only one curve.



(a) cardinality per column at 6 320 (b) cardinality at 10

Figure 5.3: Ratio of a multi-column bitmap index's $\textsc{RunCount}_{32}$ over a single-column bitmap index's $\textsc{RunCount}_{32}$ built on an unsorted table. Overall, the ratio $\textsc{RunCount}_{32}(\beta)/\textsc{RunCount}_{32}(\alpha)$ (abbrev. $\mathrm{RC}_{32}(\beta)/\mathrm{RC}_{32}(\alpha)$) for a sorted table decreases asymptotically as we increase the number of columns.

each time a new column is added to the table. Therefore, initially when using two columns the ratio $m_\beta/m_\alpha = 10\,000/(6\,320 \times 2) \approx 0.8$, adding a column adds 6 320 to the denominator decreasing the ratio to $10\,000/(6\,320 \times 3) \approx 0.53$ and so on.

We also look at the behavior of m_β/m_α for a case where the attribute cardinality is significantly small, 10 distinct attribute values per column (See Figure 5.2b). Initially, as columns are added, m_β/m_α increases rapidly as it gains more attribute value conjunctions to draw from with each column added. Each new column, adds 10 new distinct attribute values. For m_β, with c columns there are up to $\min(10^c, 10\,000)$ bitmaps while for m_α, with c columns there are $10c$ bitmaps. Starting with 2 columns, we have $m_\beta/m_\alpha = 10^2/(10 \times 2) \approx 5$, with 3 columns $m_\beta/m_\alpha = 10^3/(10 \times 3) \approx 33$. The ratio m_β/m_α continues to increase until a maximum is reached when five columns are used. There could be up to $10^5 = 100\,000$ bitmaps, but this number exceeds the number of records, thus $m_b \approx 10\,000$. Past five columns, m_β/m_α starts to decrease as m_β remains at 10 000, while m_α continues to increase by 10 bitmaps with each new column.

Next, we look at RunCount_{32} and the size on disk for bitmap indices with cardinalities $\approx 6\,320$ and 10. We consider two cases: an unsorted case where the table is shuffled and a sorted case where the table is sorted lexicographically. Starting with the unsorted case, when column cardinality is $\approx 6\,320$

per column, the ratio of

$$\textsc{RunCount}_{32}(\beta)/\textsc{RunCount}_{32}(\alpha)$$

behaves in a very similar way to m_β/m_α as shown in the unsorted curve of Figure 5.3a. Because cardinality is high, finding clean runs in the unsorted table is unlikely. Consequently, the relationship of $m_\alpha > m_\beta$ directly translates to $\textsc{RunCount}_{32}(\alpha) > \textsc{RunCount}_{32}(\beta)$. In contrast, for a low cardinality of 10, because $m_\alpha < m_\beta$ and multi-element runs are more likely to form for single-column bitmap indices, we have that $\textsc{RunCount}_{32}(\alpha) < \textsc{RunCount}_{32}(\beta)$ for 2–10 columns (see unsorted curve in Figure 5.3b). Because $\textsc{RunCount}_{32}$ improves when there are long clean runs of '0's or '1's, it does not map directly to the number of bitmaps. As a result, while m_β is ≈ 160 times greater than m_α, $\textsc{RunCount}_{32}(\beta)$ is only ~ 2.5 times greater than $\textsc{RunCount}_{32}(\alpha)$. As m_β increases with each new column, the bitmaps in β tend to become cleaner when compared to those of α. When $m_\beta = 10\,000$ bitmaps, $\textsc{RunCount}_{32}(\beta)$ becomes constant, while $\textsc{RunCount}_{32}(\alpha)$ continues to increase with each new column. Unlike m_β, which is limited by the number of records $n = 10\,000$, m_α is not limited by the number of records and can grow beyond this bound.

Lexicographic sorting improves the RunCount and size of a single-column index, but it becomes less effective as the number of columns increases. This can be noted by the decreasing distance between the sorted and unsorted

curves as number of columns is increased for both high and low cardinality in Figures 5.3a and 5.3b respectively.

In a single-column index, when the number of columns is increased, the improvement gained through reordering the table lexicographically decreases. In Figure 5.3b, this effect can be best appreciated on low cardinalities, when 2–4 columns are used. The maximum size of a multi-column bitmap index occurs with 5 columns; when the number of bitmaps in a multi-column index is very close or at the total number of records. From that point on, one can notice that the curves for the sorted multi-column index and the unsorted multi-column index are almost the same. Only columns with higher priority of being sorted gain benefit, while later columns gain little if any benefit from sorting. In contrast, a multi-column index can be optimally sorted, as all the attribute values in each record are treated as a single attribute value of a single column. However, the improvement due to sorting becomes less significant as the number of columns increases. This is due to the effect an increase on the number of columns has on the number of bitmaps. The number of bitmaps increases to a value equal to the number of records. Once it reaches this maximum, sorting has no effect, because the elements are all repeated only once.

5.2.2 Cardinality

Attribute cardinality also determines when a multi-column index or single-column index is greater in size than the other. In our experiment, we used

2-column tables with 10 000 records. These tables had participating columns of approximately equal cardinality. Because high or low cardinality is a term relative to the number of records, we use the cardinality ratio introduced in Section 5.1.

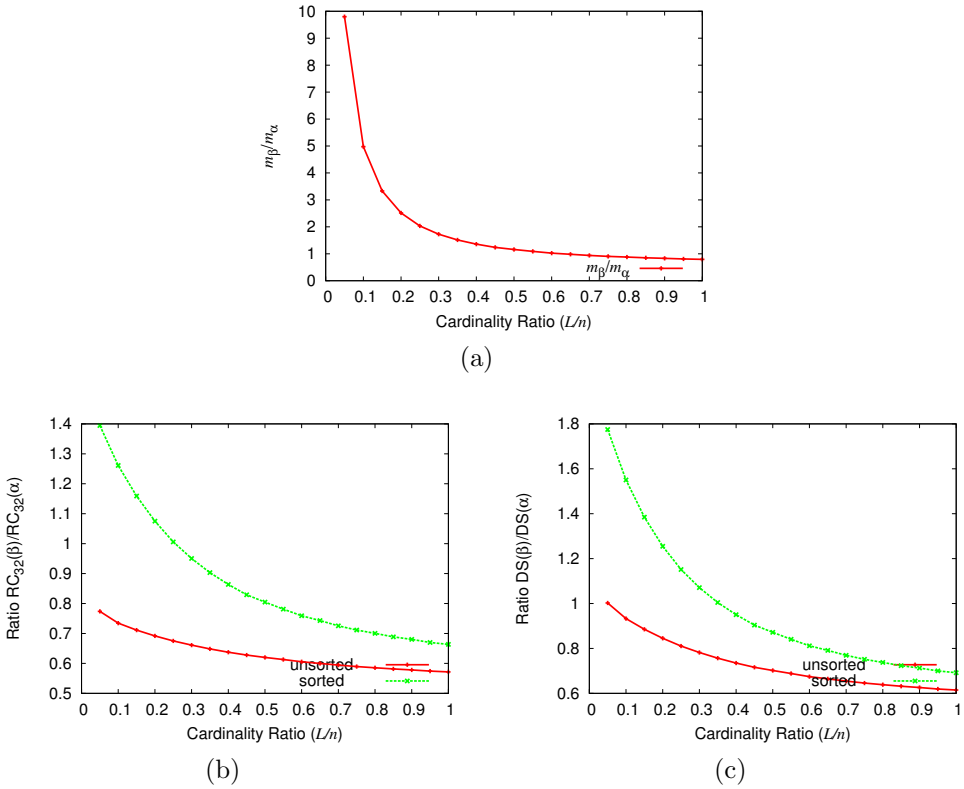


Figure 5.4: We examine the behaviors of different aspects of bitmap indices as we vary the cardinality ratio per column. Columns have the same cardinality ratio.

Let us look at the case when there is no correlation between columns. In our model, we represent this as columns in a table where attribute values from a first column have equal probabilities of pairing up with attribute values

of a second column in each row. When there is no correlation, knowing the value of the first component in a given tuple cannot help us predict the value of the second component. The cardinality per column L_i was varied from 500 to 10 000. Recall from Section 4.1 that in a single-column bitmap index α, the expected number of bitmaps $m_\alpha = \sum_{i=1}^{c} L_i$. When the cardinality ratio is $500/10\,000 = 0.05$, $m_\alpha = 500 + 500 = 1000$. However, for a multi-column index β, m_β is bounded by $[500, 10\,000]$. Because there is no correlation between the columns we expect approximately 10 000 bitmaps in β. Therefore, our multi-column index is 10 times greater than our single-column index for a cardinality ratio of 0.05 per column (see Figure 5.4a).

As we start increasing the cardinality of each column, the number of bitmaps we expect in a multi-column index does not change because it is already at its highest bound of 10 000 bitmaps. In contrast, in our single-column bitmap index the expected number of bitmaps continues to increase, reaching 10 000 when the cardinality ratio is approximately 0.5 per column, and going beyond it for $L_i/n \gtrapprox 0.5$. Thus, when the cardinality ratio is around 0.5, the ratio $\frac{m_\beta}{m_\alpha} \approx 1$. Increasing the cardinality ratio further makes $\frac{m_\beta}{m_\alpha} < 1$, as shown in Figure 5.4a.

We also look at $\textsc{RunCount}_{32}$ and the size on disk for multi-column and single-column bitmap indices as the cardinality ratio per column is increased. As usual, we look at an unsorted and sorted case.

We start with the unsorted case, illustrated in the curve labeled as unsorted in Figure 5.4b. As previously explained, the number of bitmaps $m_b \approx 10000$ for

all cardinality ratios. Because there are only 10000 records, most bitmaps in the multi-column bitmap index contain one or two bit values of 1 and the remaining bit values are 0. Thus, $\text{RunCount}_{32}(\beta)$ remains mostly unchanged as cardinality continues to be increased. In contrast, the single-column bitmap index has fewer but much dirtier bitmaps. Initially, $m_\alpha \approx 1000$, and most if not all bitmaps in α are dirty and possibly not compressible. As m_α increases with the cardinality ratio towards 10000 and more bitmaps, the bitmaps tend to become slightly cleaner explaining the slower than initial decline in $\text{RunCount}_{32}(\alpha)$.

For the same reasons explained in the unsorted case, the sorted case follows the same decline in the ratio $\text{RunCount}_{32}(\beta)/\text{RunCount}_{32}(\alpha)$. Yet, as illustrated in the curve labeled as unsorted in Figure 5.4b, we notice that unlike in the sorted case, $\text{RunCount}_{32}(\beta)/\text{RunCount}_{32}(\alpha)$ is greater than one. Sorting improves α making its average bitmap cleaner and more compressible, consequently allowing the fewer bitmaps in α to become a meaningful advantage over the more abundant bitmaps in β.

The behavior of the disk space ratio, $\text{DS}(\alpha)/\text{DS}(\beta)$, closely resembles that of $\text{RunCount}_{32}(\beta)/\text{RunCount}_{32}(\alpha)$, making the conclusions derived from the RunCount_{32} model directly applicable to the size of single-column and multi-column bitmap indices on disk (see Figure 5.4c).

5.2.3 Varying Correlation Between Columns

We look at the behavior of multi-column bitmap indices by varying the correlation between two columns. Correlation is the probabilistic dependence between two variables or observed data values (see Section 3.4). If two variables are correlated then it is possible to estimate or predict one from the other. Correlation is a factor in the size of multi-column bitmap indices, because bitmaps are made on conjunctions of attribute values instead of separate attribute values. The higher the correlation between two columns, the more likely it is that the same conjunctions of attribute values are present when building the index. This tends to decrease the total number of bitmaps. The converse is also true: if columns are not correlated then different combinations of attribute values are more likely to be present, thus increasing the total number of bitmaps.

We model correlation in our uniform datasets by duplicating each value of the first column onto the second with a given probability p (see Algorithm 1). In our experiments, we continued using datasets with 10 000 records, and 2 columns. In these datasets we vary the correlation factor p between the 2 columns. Varying the correlation has an impact in the number of bitmaps, the RunCounts, and ultimately the size on disk of multi-column bitmap indices. We keep the cardinality per column roughly constant when varying p.

Algorithm 1 Pseudocode for simulating correlated columns

```
for r from 1 to n do
  Let v_{r,1} ← a random value between 0 and L
  if rand() ≤ p then
    Let v_{r,2} ← v_{r,1}
  else
    Let v_{r,2} ← a random value between 0 and L
  end if
  Put tuple (v_{r,1}, v_{r,2}) in table T
end for
```

Number of Bitmaps for Correlated Data

Correlation affects the number of bitmaps m_β in a multi-column bitmap index β. Indeed, m_β depends on the number of conjunctions that are created when the multi-column bitmap index is built. In our experiments, m_α is not affected because m_α depends on the individual attribute cardinalities, and we hold these nearly constant.

For a multi-column bitmap index on a table with 2 columns, we estimate m_β of our synthetic datasets for a given probability p, through the following lemma.

Lemma 5.1. *Consider a uniformly distributed* 2*-column table containing* n *records. The cardinalities for the first and second columns are* L_1 *and* L_2 *respectively. Assuming that* $n \ll L_1 L_2$*, we have that* m_β*, the expected number of bitmaps for a multi-column bitmap index, can be calculated through the*

following equation:

$$m_\beta \approx L_1(\varrho + (1-\varrho)\pi) + (L_2L_1 - L_1)(1-p)\varrho.$$

Here, $\varrho = \frac{n}{L_1L_2}$, $\pi = 1 - ((1-\varrho) + \varrho(1-p))^{L_2-1}$, *and* p *is the probability of overwriting an element in the first column onto the second.*

Proof. For illustration purposes, imagine a dataset with 2 columns, each with 4 distinct attribute values ranging from 1 to 4. The product of these 2 cardinalities L_1L_2 gives us a total of 16 possible rows to choose from (see Figure 5.5a). Out of those 16 rows, n are picked for the dataset. In our mathematical analysis we approximate through a model without replacement. In other words, we assume that a row cannot be re-picked. This is likely to be true when $n \ll L_1L_2$, which is often the case for many datasets. Our model's accuracy is determined by n, the number of records we pick out of all the possible records that can be made with the columns. The bigger n is in relation to the product of our cardinalities L_1L_2, the less valid our prediction.
Initially, before applying the correlation factor p, all possible rows have equal probability of being selected when picking n rows. This probability we denote as ϱ and it is equal to $\frac{n}{L_1L_2}$. Elements in the diagonal of the table of possible tuples represent where the attribute values are duplicates. Elements outside of the diagonal represent cases where the attribute values are not duplicates (See Figure 5.5a). When we add the correlation factor p, the probability that an element outside of the diagonal is selected decreases from ϱ to $\varrho(1-p)$

	1	2	3	4
1	$\varrho_{1,1}$	$\varrho_{1,2}$	$\varrho_{1,3}$	$\varrho_{1,4}$
2	$\varrho_{2,1}$	$\varrho_{2,2}$	$\varrho_{2,3}$	$\varrho_{2,4}$
3	$\varrho_{3,1}$	$\varrho_{3,2}$	$\varrho_{3,3}$	$\varrho_{3,4}$
4	$\varrho_{4,1}$	$\varrho_{4,2}$	$\varrho_{4,3}$	$\varrho_{4,4}$

(a) Matrix representing all possible rows that can be formed with the attribute values of the first (first values) and second (second values) column

ϱ'	$\varrho(1-p)$	$\varrho(1-p)$	$\varrho(1-p)$
$\varrho(1-p)$	ϱ'	$\varrho(1-p)$	$\varrho(1-p)$
$\varrho(1-p)$	$\varrho(1-p)$	ϱ'	$\varrho(1-p)$
$\varrho(1-p)$	$\varrho(1-p)$	$\varrho(1-p)$	ϱ'

(b) Matrix representing probabilities all rows, considering the probability of overwriting the second attribute value with a duplicate of the second

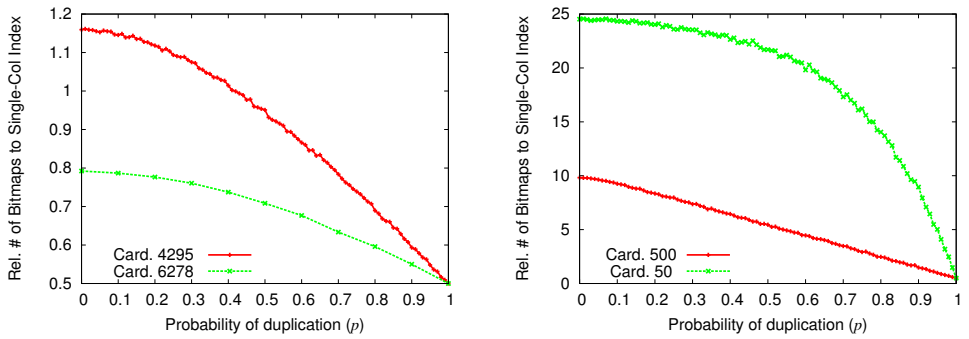
Figure 5.5: Matrices representing all possible records, each record has a probability $\varrho = \frac{n}{L_1 L_2}$.

(see Figure 5.5b). In contrast, the probability that a row in the diagonal is selected increases. We denote this new probability as ϱ'. This probability ϱ' is constituted by 2 possible events. The first event considers that there is a tuple of duplicate values already on the diagonal (before applying correlation). The probability of this event is ϱ. For this first event the outcome can only be duplicate elements whether the first attribute value is duplicated onto the second or not. The second event is that the picked tuple is not initially in the diagonal. In other words, the tuple's attribute values are not duplicates. For this second event we must then compute the probability that the original value for the second column is overwritten by the first column's

value. We denote this probability as π. What is the value of the probability π? To answer this question, we consider the probability that our tuple of different attribute values is not mapped to the diagonal. The tuple may have different attribute values in its two columns either because it was one of the n selected tuples in the first place, or because when correlation was applied, it did not overwrite the original value with the duplicate. This probability is $(1-\varrho)+\varrho(1-p)$. Therefore, we have that $\pi = 1-((1-\varrho)+\varrho(1-p))^{L_2-1}$. Here, the exponent L_2-1 is the number of tuples that are off the diagonal. □

In Figure 5.6 we confirm our theoretical results with our synthetic experiments. We plot the results after running experiments on synthetic uniform 2-column tables with $n = 10\,000$. To compare, we also plot a predictor curve derived from Lemma 5.1. On the Y-axis we have the ratio m_β/m_α, and on the X-axis we have our measure of correlation for these datasets, the probability p.

The curves produced by Lemma 5.1 estimate the number of bitmaps for different probabilities. The prediction of m_β is usually most accurate at the endpoints when p is either 0 or 1. Our predicted m_β/m_α ratio becomes progressively closer to the actual m_β/m_α as the product of the cardinalities becomes even greater than the number of records in the dataset. Table 5.1 summarizes the maximum relative error percentage between the predicted and actual ratios at a given cardinality per column. Thus, Lemma 5.1 approximates m_β within 3% for any value of p.



(a) Average Cardinality per Column at 4295 and 6278

(b) Average Cardinality per Column at 50 and 500

Figure 5.6: Relative number of bitmaps of a multi-column index to that of a single-column bitmap index. The ratio m_β/m_α decreases as p increases. Some of the curves produced by our lemma are not visible because they are very near to the curves produced experimentally.

Table 5.1: Maximum relative error between predicted m_β/m_α and actual m_β/m_α.

Cardinality	Max. Rel. Error (%)
500	3.4
5 000	1.0
10 000	0.7

Like our experimental results, the prediction derived from Lemma 5.1 also falls within the bounds given in Section 4.1. When p is 0, m_β can be equal to the product of the cardinalities, $L_1 L_2$, but it may not exceed the number of records n. Therefore, for cases where $L_1 L_2 > n$, $m_\beta \approx n$. Lemma 5.1 does not cover cases where $L_1 L_2 < n$. At the other endpoint, when $p = 1$, the first and second columns are perfect replicas. The number of attribute values does not change when the columns are taken together, and therefore the number of bitmaps is equal to the cardinality of the first column, L_1. This last conclusion is shown in all plots of Figure 5.6 when $p = 1$, $m_\beta / m_\alpha = 0.5$. Also, notice that for all plots in Figure 5.6 that m_β / m_α is minimal when $p = 1$ and maximal when $p = 0$. For probabilities that are neither 0 or 1, as we increase factor p, m_β decreases towards its lower bound $L_{\max}$ (see Section 4.1).

Unlike m_β, the number of bitmaps in a single-column index, m_α, is not bounded by the number of records alone, but by the product of the number of records and columns, nc. The value m_α is the sum of the cardinalities of the indexed columns. Theoretically, m_α does not change because the cardinalities per column are the same for any value of p. In practice, m_α does, but not significantly in our experiment.

We illustrate the curves of our experiment when using high cardinality in Figure 5.6a. For these curves, the experiment was performed holding maximum column cardinalities at 5 000 and 10 000. Notwithstanding, the actual column cardinalities do not reach these maxima because some of the column

values are repeated in different tuples, thus decreasing the actual or observed cardinality. The actual column cardinalities were $\approx 4\,300$ and $\approx 6\,300$ for $5\,000$ and $10\,000$ respectively. The expected number of observed distinct values in each column or actual L_1 and L_2 are, respectively, $L_1(1-(1-\varrho)^{L_2})$ and $L_2(1-(1-\varrho)^{L_1})$ when $p \geq 0$ and $L_1 = L_2$. For the curve labeled as 'Card 4295', the maximum $m_\beta/m_\alpha \approx 1.2$, and the curve labeled as 'Card 6278', the maximum $m_\beta/m_\alpha \approx 0.8$; thus we notice that ratios are smaller for all probabilities when cardinalities are increased (see Figure 5.6)

For the curves illustrated in Figure 5.6b, the experiment was performed holding maximum column cardinalities at 50 and 500. We consider these as low cardinality cases. We did not make a curve for Lemma 5.1 when the cardinality is 50 because Lemma 5.1 only makes sound predictions when $L_1 L_2 \geq n$. In the curve labeled as 'Card 50', L_1 and L_2 are both 50; thus, $m_\beta \approx 50 \times 50 = 2\,500$ and $m_\alpha \approx 50 + 50 = 100$. As a result, when $p = 0$, the ratio $m_\beta/m_\alpha \approx 25$ as seen in Figure 5.6b. In the curve labeled 'Card. 500', L_1 and L_2 are both 500, m_β could potentially be $500 \times 500 = 250\,000$, yet $n = 10\,000$ and thus $m_\beta \approx 10\,000$. In contrast $m_\alpha \approx 500 + 500 = 1\,000$; thus $m_\beta/m_\alpha \approx 10$ when $p = 0$ (also shown in Figure 5.6b).

RunCounts for u and T

In this section, we look at RunCount behaviors related to the RunCounts of multi-column and single-column bitmap indices as we vary the correlation factor p. The RunCount of the tuples taken as attribute values, Run-

$\textsc{Count}(u)$, is always smaller than the sum of the $\textsc{RunCount}$s of the individual columns, $\textsc{RunCount}(T)$.

Lemma 5.2. *Consider a c-column table T and a column u of unique identifiers for each distinct tuple. We can bound* $\textsc{RunCount}(T)$ *through the following inequality:*

$$\textsc{RunCount}(u) + c - 1 \leq \textsc{RunCount}(T) \leq c \cdot \textsc{RunCount}(u)$$

Proof. When we build multi-column bitmap indices, we consider each distinct tuple as a distinct attribute value. Thus, when we add a tuple that is not identical to the previous one, we count it as a new run. We denote this $\textsc{RunCount}$ as $\textsc{RunCount}(u)$, where u is a column that assigns a unique identifier to each distinct tuple (also explained in Section 4.4). In contrast, when we build a single-column bitmap index, we consider the attribute values for the columns of the table separately, and thus we look at the table $\textsc{RunCount}$, $\textsc{RunCount}(T)$.

Consider calculating $\textsc{RunCount}(u)$ and $\textsc{RunCount}(T)$ as we build the table T tuple by tuple. Initially, when we add the first tuple to the table, $\textsc{RunCount}(u)$=1, and $\textsc{RunCount}(T)$=c. For each distinct new tuple that we add to the table the number of runs accrued in $\textsc{RunCount}(T)$ can be between 1 and c. In contrast, for $\textsc{RunCount}(u)$ only one run is accrued when a new distinct tuple is added to the table T. When the new tuple added to the table is identical to our previous tuple, the number of new

runs is 0. Otherwise, if the new tuple added to the table is distinct in every attribute value from the previous tuple, the number of new runs is c. Consequently, for every new run in $\textsc{RunCount}(u)$, there must be at least one new run in RunCount(T). Therefore, $\textsc{RunCount}(u) + (c - 1) \leq \textsc{RunCount}(T)$. Conversely, there can only be at most c runs for every new run in $\textsc{RunCount}(u)$; thus if we start with c and add c runs with each new distinct tuple added, we obtain an upper bound for $\textsc{RunCount}(T)$ as follows: $\textsc{RunCount}(T) \leq c + c \cdot (\textsc{RunCount}(u) - 1)$. This bound can be further simplified as $\textsc{RunCount}(T) \leq c \cdot \textsc{RunCount}(u)$. □

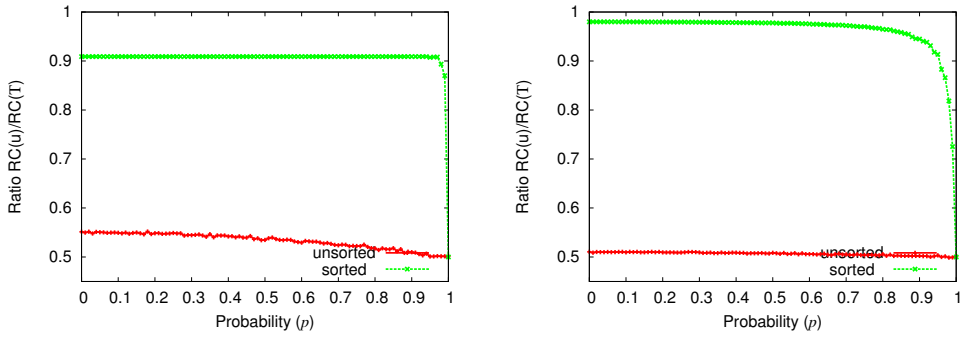
As a corollary, we can bound $\textsc{RunCount}(u)$ with respect to $\textsc{RunCount}(T)$ as:

$$\frac{\textsc{RunCount}(T)}{c} \leq \textsc{RunCount}(u) \leq \textsc{RunCount}(T) + 1 - c.$$

Furthermore, we can also bound our RunCount ratio as:

$$\frac{1}{c} \leq \frac{\textsc{RunCount}(u)}{\textsc{RunCount}(T)} \leq 1 + \frac{1 - c}{\textsc{RunCount}(T)}.$$

In Figure 5.7, we plot the results for $\textsc{RunCount}(T)$ and $\textsc{RunCount}(u)$ for 2-column tables of 10 000 records. We look at two cases: unsorted and sorted. In the unsorted case, the table is shuffled, while in the sorted case the table is sorted using the lexicographic order. Lexicographic sorting takes place before building the indices.



(a) cardinality per column at 10

(b) cardinality per column at 50

(c) cardinality per column at 1000

(d) cardinality per column at 4300

Figure 5.7: Plots of the RunCount(u)/RunCount(T) ratio with varying correlation factor p. Cardinality is varied in all four figures.

We start with the unsorted case. The curve labeled as unsorted in the legend of the figures is the ratio of $\textsc{RunCount}(u)/\textsc{RunCount}(T)$. In all four graphs, notice that for all cardinalities and all values of p, $\textsc{RunCount}(u)$ is always smaller than $\textsc{RunCount}(T)$, as predicted by Lemma 5.2.

As we increase the cardinality, the curvature of the unsorted curve becomes flatter because a greater number of multi-element runs is more likely to form for columns with low cardinalities. This effect is apparent in Figure 5.7a, where the cardinality is ten.

As we increase the cardinality of the columns, the likelihood of getting multi-element runs is low, therefore RunCount(T) and $\textsc{RunCount}(u)$ both reach their maximum, and the maximum of $\textsc{RunCount}(T)$ is always twice that of $\textsc{RunCount}(u)$. As a result, the ratio of $\textsc{RunCount}(T)/\textsc{RunCount}(u)$ is close to 1/2 for all values of p (see Figures 5.7b,5.7c,5.7d).

We also look at the sorted case. When a table is sorted, $\textsc{RunCount}(u) = m_\beta$. Thus $\textsc{RunCount}(u)$ also follows the bounds given in Section 4.4. We can also express $\textsc{RunCount}(T)$ as $L_1 + \textsc{RunCount}(u)$. As a result, we have that the ratio

$$\textsc{RunCount}(u)/\textsc{RunCount}(T) = \textsc{RunCount}(u)/(\textsc{RunCount}(u) + L_1).$$

When $p = 0$, this ratio increases as we increase the cardinalities per column, as long as $L_1 L_2 < n$ (see Figures 5.7a and 5.7b). The maximum is obtained when $L_1 L_2 = n$. Because $\textsc{RunCount}(u) = m_\beta$, it cannot exceed

n, thus for $L_1L_2 > n$, RunCount$(u) = n$. Yet if we increase the cardinalities per column, L_1 continues to increase, and so we see a decrease in RunCount(u)/RunCount(T) when $L_1L_2 > n$ (see Figures 5.7b and 5.7c). The curvature of the curve increases with the cardinality per column. This is because as L_1L_2 increases, more possible attribute values can be formed with the multi-column bitmap index. The probability that a given value will repeat itself decreases as the cardinality is increased, and thus more attribute values get eliminated when p is increased.

Finally, notice that RunCount(u)/RunCount(T) always converges to 0.5 when the correlation factor $p = 1$. Indeed, the two columns of the table become duplicates, and therefore sorting one optimally also sorts the other optimally. Therefore, the RunCounts of both columns are identical, and because no new attribute values are generated, RunCount$(u) =$ RunCount$(T)/2$.

RunCounts for α and β

In this section, we look at the unit RunCount for a single-column bitmap index α and a multi-column bitmap index β as we vary the correlation factor p. RunCount(α) depends on the individual RunCounts of the participating columns c_0 and c_1. By contrast, RunCount(β) depends on the RunCount of distinct tuples, RunCount(u). Recall from Equation 4.1 that we can calculate the total RunCount for the associated bitmaps of a given column

c as

$$L_c + 2(\text{RunCount}(c) - 1).$$

Thus, for the multi-column bitmap index, we calculate $\text{RunCount}(\beta) = L_u + 2(\text{RunCount}(u) - 1)$, while for the single-column bitmap index we calculate it as $L_1 + L_2 + 2(\text{RunCount}(c_1) - 1) + 2(\text{RunCount}(c_2) - 1)$. We can calculate our $\text{RunCount}(\beta)/\text{RunCount}(\alpha)$ ratio as

$$\frac{L_u + 2(\text{RunCount}(u) - 1)}{L_1 + L_2 + 2(\text{RunCount}(c_1) - 1) + 2(\text{RunCount}(c_2) - 1)}.$$

We plot the ratio of $\text{RunCount}(\beta)/\text{RunCount}(\alpha)$ for both unsorted and sorted cases in Figure 5.8. We see that the overall behavior of ratio $\text{RunCount}(\beta)/\text{RunCount}(\alpha)$ is very similar to that of the ratio $\text{RunCount}(u)/\text{RunCount}(T)$.

Like $\text{RunCount}(u)/\text{RunCount}(T)$, $\text{RunCount}(\beta)/\text{RunCount}(\alpha)$ also converges to $1/2$ when $p = 1$. To demonstrate this, we have that

$$\begin{aligned} \text{RunCount}(c_0) &= \text{RunCount}(c_1) \\ &= \text{RunCount}(u) \end{aligned}$$

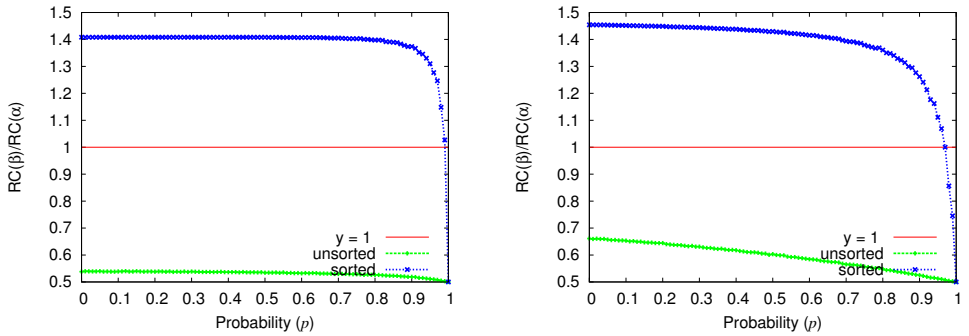
and $L_1 = L_2 = L_u$, thus expressing the ratio in terms of a single variable we obtain

$$\frac{L_1 + 2(\text{RunCount}(c_1) - 1)}{2L_1 + 4(\text{RunCount}(c_1) - 1)} = \frac{1}{2}. \tag{5.1}$$

This result is also seen experimentally: notice that all curves in Figure 5.8

converge to 1/2 when $p = 1$. Moreover, the curvature of the curves is small when the cardinality is low and becomes greater as the cardinality increases. Next, we look at the sorted case. In all plots of Figure 5.8 we see that, unlike in the plots for RunCount(u) and RunCount(T) (in Figure 5.7), RunCount(β) is not always smaller than RunCount(α). Indeed, it becomes smaller than RunCount(α) at a break-even point that depends on the cardinality per column. We look at the break-even points for low cardinalities, namely 30 and 100. In Figure 5.8a, where the cardinality per column is 30, the break even point is past $p = 0.99$. A small decrease is seen in Figure 5.8b where the break-even point is now near $p = 0.97$. We also look at RunCount(β)/RunCount(α) when varying p for higher cardinalities. In Figure 5.8c, where the cardinality per column is 1000, the break-even point occurs near $p = 0.7$. Finally, in Figure 5.8d, the cardinality per column is at 4300; there is not a break-even point at this cardinality and RunCount(β) is always smaller than RunCount(α).

We look at the unsorted cases shown in the curves labeled as unsorted in Figure 5.8. For all cardinalities and all values of p, RunCount(β) is always smaller than RunCount(α) (see Figure 5.8). As we increase the cardinality, the curvature of the unsorted curve decreases. This is due to the fact that runs are more likely to form for lower cardinalities than they are likely to form for high cardinalities. Thus, when the correlation is 0 it is unlikely to find duplicates and the number of bitmaps for the multi-column bitmap index is not reduced. As we increase the cardinality of the columns the



(a) cardinality per column at 30

(b) cardinality per column at 100

(c) cardinality per column at 1000

(d) cardinality per column at ≈ 4300

Figure 5.8: Ratio RunCount(β)/RunCount(α) for different cardinalities per column.

likelihood of multi-element runs are almost none; therefore RunCount(β) and RunCount(α) both reach their maximum, and the maximum of RunCount(α) is always twice that of RunCount(β). Therefore the ratio of RunCount(β)/RunCount(α) = 1/2 for all values of p see Figure 5.8a.

RunCount$_{32}(\beta)$ and RunCount$_{32}(\alpha)$

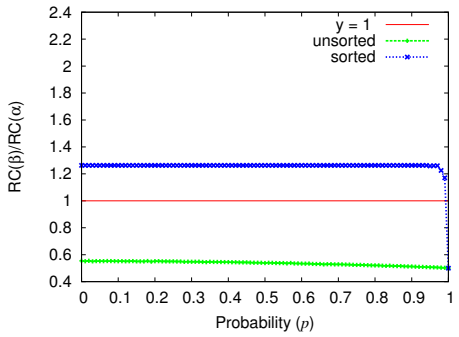
Recall from Section 4.3 that dirty words are 32-bit sequences of mixed 1s and 0s and clean runs are sequences of only 0s or only 1s whose lengths are a

multiple of 32. We obtain RunCount_{32} for our single-column bitmap index, α, and multi-column bitmap index, β by adding the number of dirty words and clean runs.

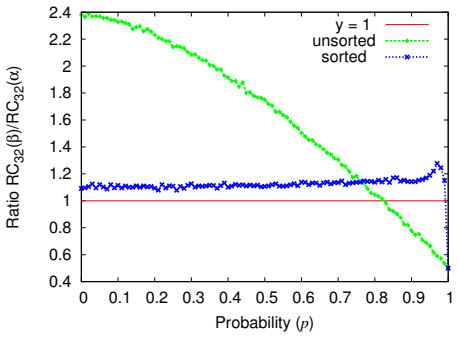
As previously mentioned in Section 2.4, RunCount is an approximate model to the size of bitmap indices. Thus, its behavior may not correctly represent the size of a bitmap index compressed using EWAH which compresses runs of 32-plets of 0s or 1s. Constituting a multi-element run when using RunCount requires two or more instances of a particular element while constituting a multi-element run when using RunCount_{32} would require two or more 32-plets of an element (see Section 4.3). Thus, a run that may be compressed using a compression scheme for unit runs may not necessarily be compressed using a compression scheme for 32-plet runs.

Often, the behavior of the RunCount and RunCount_{32} ratios with respect to p is similar, but in certain cases they can differ significantly. One such case is when the cardinality per column is 10, as illustrated in Figures 5.9a and 5.9b. We can clearly see that when the table is unsorted, $\frac{\text{RunCount}(\beta)}{\text{RunCount}(\alpha)} < 1$ but $\frac{\text{RunCount}_{32}(\beta)}{\text{RunCount}_{32}(\alpha)} > 1$ for most values of p.

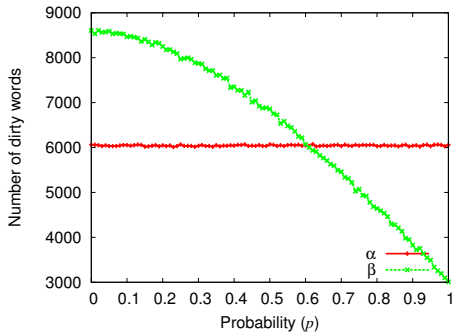
We look at the number of dirty words and clean words for α and β, when the table is unsorted. In such case, the number of dirty and clean words in α remains constant as p is increased, as illustrated in Figures 5.9c and 5.9d. Clean words are very unlikely to form in α, which explains the small number of clean runs and the relatively large number of dirty words counted for α. In contrast, the number of dirty words and clean runs for the multi-



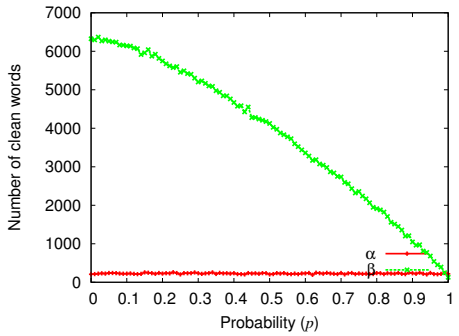
(a) cardinality per column at 10



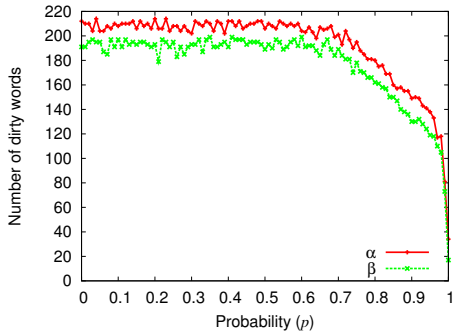
(b) cardinality per column at 10



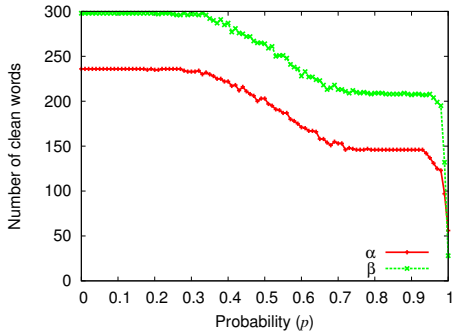
(c) dirty words in unsorted case



(d) clean words in unsorted case



(e) dirty words in sorted case



(f) clean words in sorted case

Figure 5.9: Graphs a and b illustrate a comparison between RunCount$_{32}$ and RunCount when they are dissimilar. Graphs c through f illustrate the number of dirty words and runs of clean words for α and β when when the table has been sorted and unsorted.

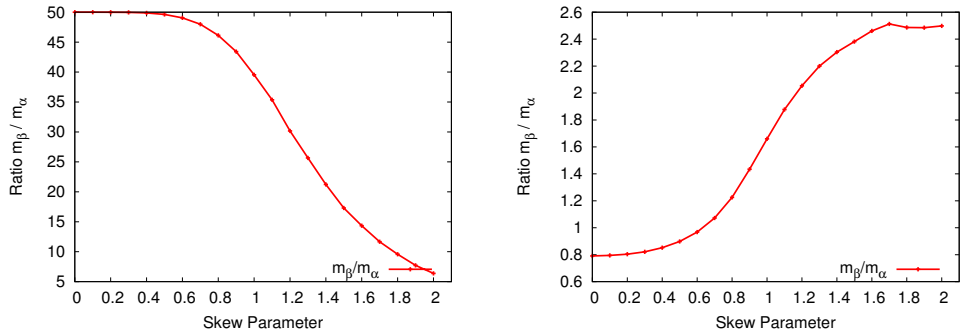
column bitmap index both decrease as p increases, effectively decreasing the ratio $\textsc{RunCount}_{32}(\beta)/\textsc{RunCount}_{32}(\alpha)$. This effect can be attributed to the fact that as p is increased, certain tuples become more frequent than others. Ultimately, this creates longer runs of 0s for bitmaps of infrequent conjunctions by moving more 1s into the already dirty bitmaps of frequent conjunctions. As illustrated in Figures 5.9c and 5.9d, when $p = 0$, β starts with many dirty words and many single-element 32-plets with possible few multi-element 32-plets of 0s. As p is increased, 1s tend to appear mostly in 10 of the 100 different bitmaps. These 10 bitmaps correspond to the 10 duplicated value tuples in the diagonals of the matrices show in Figure 5.5. The other 90 bitmaps are left with longer runs of 0s, possibly forming more multi-element 32-plets; ultimately this decreases both dirty words and clean runs.

We also look at the number of dirty words and clean words after sorting the table lexicographically, as shown in Figures 5.9e and 5.9f. We can see that the number of dirty words and clean words in α and β both decrease in a similar fashion as p increases. Because the table is ordered, it is now possible for α to form clean runs of 32-plets. Even though α has more dirty words than β, most of its dirty words have been transformed to clean runs through sorting. On the other hand, β has slightly fewer dirty words than α and also benefits from sorting as it also has more clean runs than dirty words. However, β has 80 more bitmaps than α does. This effectively makes β larger even though it has relatively more clean runs than dirty words as

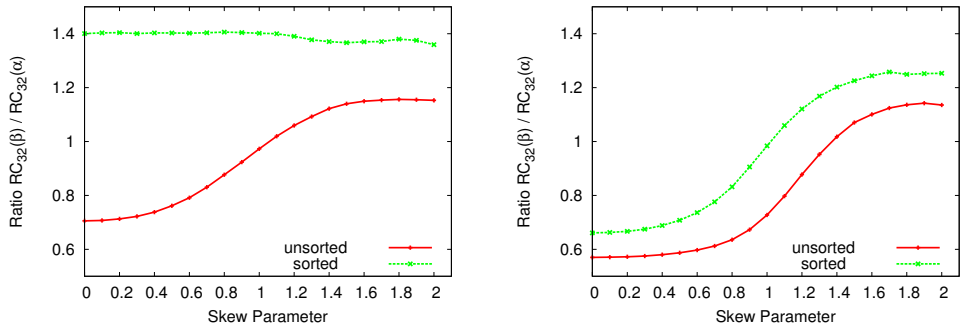
opposed to α.

5.3 The Effects of Varying Skew

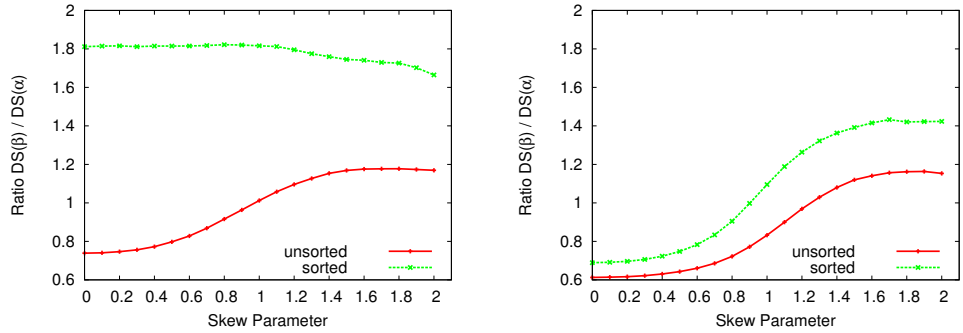
Database researchers use the Zipfian distribution to model the content of tables for various datasets that have real data [12, 34]. Hence, to model the behavior of bitmap index size of datasets with real data, we build our indices on data that follows Zipfian distributions with various skews (see Section 3.2). The skew parameter allows us to control the frequency of values, making some values more frequent than others. We use this parameter to study how statistical dispersion may affect the behavior in factors determining size of multi-column and single-column bitmap indices. The following plots in Figure 5.10 were produced with 2-column tables with 100 000 records with both a low and high cardinality ratio, L/n, of 0.001 and 0.632 respectively. We look at the ratio of the number of bitmaps of a multi-column bitmap index to that of a single-column bitmap index as we increase the skew parameter. The skew parameter can affect the number of bitmaps differently depending on the maximum cardinality of the synthetic dataset. In Figure 5.10a, the maximum cardinality is of 100 per column with a corresponding cardinality ratio of 0.001. Notice that when there is no skew the multi-column bitmap index has 50 times more bitmaps than the single-column bitmap index. Because there is no correlation between the columns, m_β hits its upper bound which is $100 \times 100 = 10000$, while m_α maintains a total of 200 bitmaps;



(a) $L/n = 0.001$ (b) $L/n \approx 0.632$



(c) $L/n = 0.001$ (d) $L/n \approx 0.632$



(e) $L/n = 0.001$ (f) $L/n \approx 0.632$

Figure 5.10: These graphs were created using a 2-column table with 100 000 records in total. The skew can affect the number of bitmaps differently depending on the cardinality ratio. Overall, increasing skew favors single-column bitmap indices even more when the table is sorted lexicographically.

making $m_\beta/m_\alpha = 50$. As the skew parameter is increased, m_β decreases while m_α remains the same effectively reducing m_β/m_α. As the skew parameter increases, certain attribute values become far more frequent than others. Consequently, certain attribute value conjunctions become very frequent leaving many others so infrequent that they may not even occur once; thus, leading to a decreasing m_β.

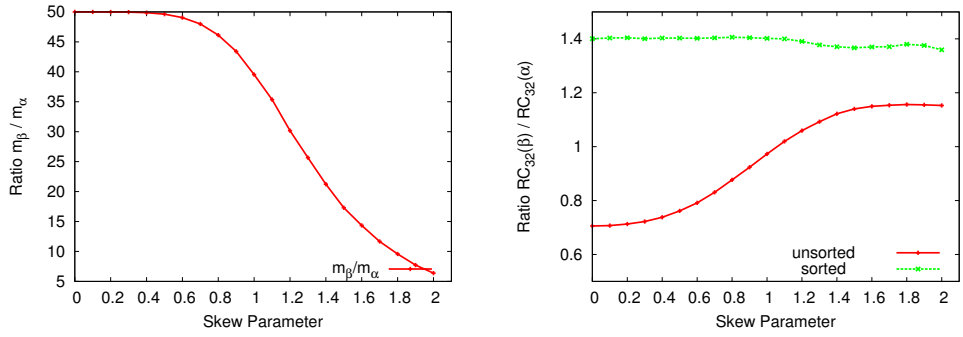
In Figure 5.10b, the maximum cardinality is approximately of 63 200 per column with a corresponding cardinality ratio of 0.632. When the skew parameter is 0, the multi-column bitmap index has approximately 20% fewer bitmaps than the single-column bitmap index. When the number of records is much smaller than the product of the cardinalities per column, m_β reaches its upper bound at 100 000. In contrast, m_α is $63\,200 + 63\,200 = 126\,400$ bitmaps making the ratio $m_\beta/m_\alpha \approx 0.8$. Because the maximum cardinality per column is relatively high, the observed cardinality decreases as the skew parameter is increased. Consequently, when the cardinality decreases, both m_β and m_α decrease. The number of joint attribute values that can be created with both columns makes m_β decrease more slowly than m_α, effectively increasing m_β/m_α as the skew parameter increases.

Next, we look at RunCount_{32} and disk space size for a low cardinality of 0.001 and high cardinality 0.632. Notice that RunCount_{32} closely resembles the behavior of disk space size of both types of indices as shown between Figures 5.10c, 5.10e and Figures 5.10d, 5.10f. We look at an unsorted case where the table is shuffled and a sorted case where the table is sorted lex-

icographically. We start with the unsorted case. Let the function DS give the size of the bitmap index in bytes. When the cardinality ratio is 0.001 we can see that the ratio of DS(β)/DS(α) increases as the skew parameter increases. Even though m_β is around 10000 when skew is 0, the bitmaps become sparse and thus also more compressible. On the other hand, m_α is always 200 and thus it has fewer bitmaps but they are not very compressible when they remain unsorted. As a result, the multi-column bitmap index takes less disk space than the single-column bitmap index. When the skew parameter is increased, m_β starts to decrease and some bitmaps become less compressible, thus multi-column bitmap indices start to lose their advantage over single-column bitmap indices in having highly compressible bitmaps. The ratios RunCount$_{32}(\beta)$/RunCount$_{32}(\alpha)$ and DS(β)/DS(α) become greater than one when the skew parameter is 1.1 and 1 respectively. When the table is sorted lexicographically, single-column bitmap indices are always smaller than multi-column bitmap indices (see Figures 5.10c,5.10e).

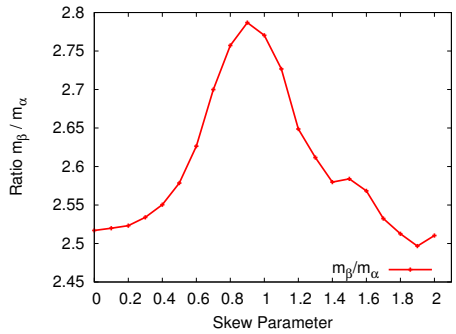
Overall, increasing the skew parameter improves the size for both single-column and multi-column bitmap indices. In single-column bitmap indices the number of bitmaps remains the same, the runs of certain bitmaps become longer and more compressible using RunCount$_{32}$. In multi-column bitmap indices, m_β decreases causing a decrease in RunCount$_{32}(\beta)$. A decrease in m_β also improves the RunCount of the second column of the table effectively improving RunCount$_{32}(\alpha)$.

In Figures 5.10f and 5.10d, we repeat the analysis using a cardinality ratio

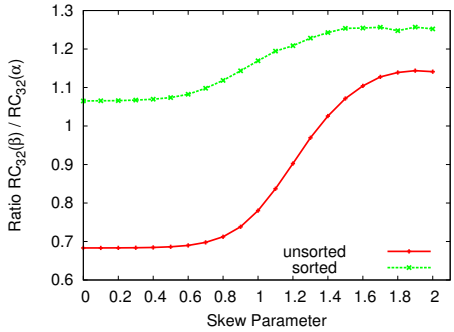


(a) $L/n = 0.001$

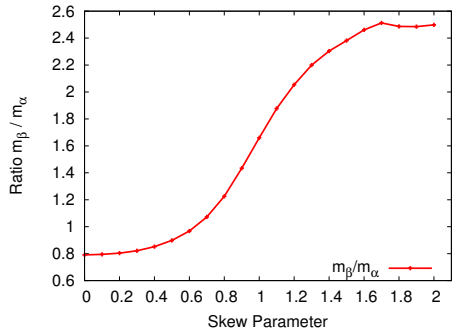
(b) $L/n = 0.001$



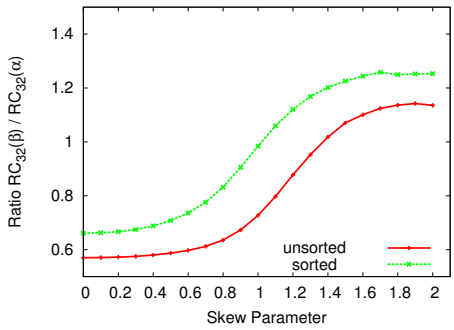
(c) $L/n = 0.2$



(d) $L/n = 0.2$



(e) $L/n \approx 0.632$



(f) $L/n \approx 0.632$

Figure 5.11: These graphs illustrate the behavior in number of bitmaps and RunCount_{32} of multi-column and single-column bitmap indices when the cardinality per column is moderate. We compare moderate cardinality with high and low cardinalities. The bitmap indices were built on a 2-column table with 100 000 records in total.

of 0.632. In this case, the multi-column bitmap index can be smaller than a single-column bitmap index. Initially, $m_\beta < m_\alpha$, but as the skew parameter increases, the cardinality per column decreases making m_α and m_β both decrease. The ratio of RunCount_{32} and disk space behaves similarly to the ratio m_β/m_α. As previously explained m_α decreases more quickly than m_β from 0 to 1.2 and gradually slows down from 1.2 to 2. As a result, an increase in skew eliminates more bitmaps in m_α and improves RunCount_{32}. The improvement slows down as higher skew parameters from 1.2 to 2.0 are reached. In the unsorted case, the single-column bitmap index can only take advantage of a reduced number of bitmaps. Yet, if the table is sorted lexicographically, $\text{RunCount}_{32}(\alpha)$ can be further improved as cardinality per column is also decreased. Consequently, the sorted curve $\text{RunCount}_{32}(\beta)/\text{RunCount}_{32}(\alpha)$ is greater than one for a wider range of skew parameters than for the unsorted curve. The single-column bitmap index becomes more effective as increasing skew decreases column cardinality and row reordering improves RunCount_{32} and ultimately disk space usage. The behavior of multi-column and single-column bitmap indices as the skew parameter is increased differs greatly when using a high or low cardinality. For moderate cardinalities between 0.001 and 0.632, the behavior combines aspects of low cardinality and high cardinality behavior. In Figure 5.11 we illustrate the ratios m_β/m_α and $\text{RunCount}_{32}(\beta)/\text{RunCount}_{32}(\alpha)$ as the skew parameter increases for a columns with a cardinality ratio at 0.2. As the cardinality moves further away from 0.001 towards the moderate range car-

dinalities, m_β approaches 10 000 while m_α continues to grow decreasing the ratio when $s = 0$. However, as s increases, column cardinality decreases causing m_α and m_β to decrease. The decrease in m_α is initially more rapid than m_β's, causing the peak of the curve in Figure 5.11c when $s = 1$. Similarly, at a cardinality ratio of 0.2 per column, RunCount_{32} displays a behavior halfway between 0.001 and 0.632 (see Figure 5.11d).

5.4 Summary

We have presented the following extreme cases where a multi-column bitmap index is always smaller than a single column bitmap index:

- When building a multi-column bitmap index from a table with duplicate columns.
- When building a multi-column bitmap index with a unique column. The higher the cardinality of the second column, the smaller the multi-column index will be in comparison to a single-column index.

We continued our research using uniform distributions varying the number of columns. Overall, **increasing the number of columns decreases the size of the multi-column index with respect to the size of a single-column index.** However, when the cardinality is low, the ratio can initially increase until a maximum is reached. The maximum is reached when the product of the columns' cardinalities is approximately equal to the number

of records. Thereafter, the ratio continuously decreases as more columns are appended to the table.

The rest of this chapter focused on using two-column tables to cut down on the complexity of the problem. We began by varying the cardinality per column of the 2-column tables and built multi-column and single-column bitmap indices. We found that **increasing the cardinality ratio decreases the size of the multi-column index relative to that of a single-column index.** Past a cardinality ratio of 0.5 the multi-column bitmap index was always smaller than the single-column bitmap index with our uniform datasets. We varied the correlation between columns using a correlation factor. The correlation factor, called p, is the probability of duplicating a value of a given tuple's first column onto the second. Overall, **increasing the correlation factor p decreased the size ratio of the multi-column bitmap index** to the single-column bitmap index. Both the ratio of number of bitmaps (m_β/m_α) and the ratio of RunCounts (RunCount(β)/RunCount(α)) also decreased when increasing the probability p. We looked at the following RunCount ratios:

- RunCount(u)/RunCount(T)
- RunCount(β)/RunCount(α)
- RunCount$_{32}$(β)/RunCount$_{32}$(α).

All three ratios showed similar behavior as we increased the correlation factor.

Last, we examined the effect of gradually changing an originally uniform distribution to a Zipfian distribution by varying the skew parameter. We found that **increasing the skew caused multi-column bitmap indices to grow larger than single-column bitmap indices.** This effect was more pronounced with higher cardinality ratios.

In this chapter, we analyzed the behavior of disk space occupied by multi-column and single-column bitmap indices as the statistical parameters of cardinality, correlation and skew were varied. Experiments in the next chapter assess correlation metrics in their effectiveness in choosing the appropriate index for minimizing disk-space usage.

Chapter 6

Datasets with Real Data

After examining synthetic data experiments in Chapter 5, we conduct experiments using datasets that have realistic data. We use realistic data to determine the applicability of our conclusions in datasets with real data. When working with synthetic data, we input certain parameters to generate data with a degree of correlation, skew or cardinality. We use these parameters as our statistical measures to compare the different datasets we build indices for. These measures cannot be applied in datasets with real data because we do not generate them. Thus instead, we measure statistical parameters from the datasets themselves. The statistical measures we used include the coentropy, mean square contingency (discussed in Chapter 3) and a sample-based metric later discussed in this chapter.

We begin with Section 6.1 by discussing the characteristics of the datasets that we used to perform our experiments. In Section 6.2, we review the sta-

tistical and sampling metrics, and explain how we compared them. Finally, Section 6.3 provides a brief discussion on how to find the correct sampling size for the sample-based metric.

6.1 The Datasets

We used several datasets that use real data for our experiments: Tweed [30], Census-Income [13], Census1881 [16, 23] and Netflix [18]. These datasets are in the format of comma-separated value files. The datasets' details are summarized in Table 6.1.

Our smallest dataset is Tweed, which is a projection of 16 columns from the original Tweed dataset. The Tweed dataset contains information on events related to terrorism in 18 West European countries for the period 1950 through 2004. Census-Income is larger than Tweed, and while its number of records is in the order of 100 000s, it has 42 columns making it the dataset with the largest number of columns. Because the number of projections of column pairs we can create is large ($\binom{42}{2} = 861$ different column pairs), we ignore some column pairs in order to reduce the workload. Thus we split Census-Income into two datasets: Census-Income part 1 with 20 different columns and Census-Income part 2 with 22 different columns. Census1881 is our intermediate-size dataset which comes from the Canadian census of 1881. Finally, our largest dataset in number of records, Netflix, contains over 100 million ratings from 480 thousand randomly-chosen, anonymous

	rows	columns	disk size
Tweed	4 963	16	240 KB
Census-Income	199 523	42	99.1 MB
Census-Income part 1	199 523	20	47 MB
Census-Income part 2	199 523	22	53 MB
Census1881	4 277 807	7	114.2 MB
NetFlix	100 480 507	4	2.5 GB

Table 6.1: Characteristics of realistic datasets we use to test our model.

Netflix customers over 17 thousand movie titles [18].

We selected these datasets because we have access to them and they are chosen to be of different sizes, to determine if the applicability of our research may vary with size. The following section uses these datasets to compare our correlation metrics against sample-based metrics.

6.2 Sampling vs Statistical Metrics

In this section, we compare the statistical metrics discussed in Section 3.4, coentropy and mean square contingency. Both of these metrics give some indication of when a multi-column bitmap index occupies less disk space than a single column bitmap index.

In our experiment, we take pairs of columns from a dataset and calculate our statistical metrics. To determine when a column pair is a good candidate for a multi-column bitmap index, we use a measure of profit where we take the difference between the disk space occupied by a multi-column bitmap

index and a single-column bitmap index. If the profit is positive, the multi-column bitmap index occupies less disk space than the single-column bitmap index and the column pair is a good candidate. If the profit is negative, the multi-column bitmap index occupies more disk space than the single-column bitmap index and the column pair is a bad candidate. Our cost model assumes that a new single-column bitmap index is built for each column pair, even if a bitmap already exists for that column in another column pair.

Calculating the statistical metrics involves building the joint histogram of the tables, which requires scanning the entire table. This may be nearly as expensive as building the bitmap indices on the entire dataset; thus, to improve the practicality of using statistical metrics, we look at calculating them on randomized samples from the dataset. A profit can also be calculated from these randomized samples. Thus, as yet another metric, we also build the bitmap indices on the samples and calculate the profit of the sample. We refer to this metric as our sample-based metric. We choose to have a sample-based metric because unlike the coentropy and mean square contingency it is not oblivious to the order of the table (see Section 4.5).

For our samples, we only considered sampling shuffled tables as sampling sorted tables raises some difficulties. We identified the following three strategies for sampling sorted tables, but we did not find them entirely satisfying:

1. A simple random sample: Even assuming that we sort the sample, it may not reflect the ordering of the original table, except maybe in the first column which would possibly have short runs.

2. Taking the first n records: This sampling would lead to an extremely compressible first column, but this may also not be very representative of the original table.

3. Hybrids between strategies (1) and (2): While it is sensible to sample blocks of consecutive rows (as in Chaudhuri et al. [7]), we may still end up with short runs that may poorly represent the original table.

Thus, we decide to work with shuffled tables because when we have many columns, and pick two at random, they are likely to "appear" shuffled even when the table was initially sorted. That is because sorting tends to mostly benefit the first few columns [17].

We also considered the following sensible transformations that may be performed on the dataset to improve the best profit. These transformations show a flaw/weakness in statistical metrics. We briefly explored the following transformations:

- Row reordering: The statistical metrics we use do not change with row reordering because they only take into account the frequencies of attribute values and not their particular order. The profit, however, can change when the table is reordered because sorting improves the compressibility of bitmap indices that have low attribute cardinalities and poor RunCounts.

- Frequency Multiplication: Our statistical metrics are also invariant under the multiplication of the frequency. If we take every distinct

row and multiply its frequency by ten or 100, the correlation would remain the same. Yet, the profit may change, as runs that were not long enough to be compressible, become compressible.

From this section, we conclude that mean-square contingency and coentropy are invariant to dataset transformations, such as row reordering and frequency multiplication. As a result, they may not be effective as dataset transformations are a determining factor in the size of multi-column and single-column bitmap indices. We also decide to experiment only with unsorted datasets as simple sampling techniques on sorted tables often fail to portray the row order of the dataset accurately.

In the following sections, we compare experimentally our two measures of correlation (mean square contingency and coentropy) against sampling. In Section 6.2.1 we provide the procedure and in Section 6.2.2 we provide the results and confirm that sampling techniques are more effective in predicting when to use multi-column or single-column bitmap indices.

6.2.1 The Procedure

To compare the effectiveness of our statistical metrics against our sample-based metric in determining good column-pairs, we performed our experiment with the following procedure:

- To begin, we calculate the profit for each column pair on the entire dataset, and keep a table mapping each column pair to its correspond-

ing profit.

- We take samples from each column-pair and with each statistical metric we calculate a value for each column-pair. For our sample-based metric the value we calculate is the profit of the column-pair sample.

- Having measured our column-pairs with our statistical metrics and sample-based metric, we sort them from highest to lowest predicted profit in accordance to each metric. Our statistical metrics do not directly predict profit amounts. They produce a ranking of column pairs that we assume matches a ranking of highest to lowest profit. We are left with lists of column pairs for each metric, ordered from most likely to produce the best profit to least likely.

- We pick the top $m = 10$ column-pair candidates that are most likely to produce the best profit. If m is chosen to be too small, convergence may occur too soon, giving inaccurate results. If m is too large then convergence may not occur until a sample size close to the total number of records is reached. Hence, there would be little or no reduction in computation time when using samples.

- Finally we use the table of actual profits to look up the profits for the top ten candidates proposed for each metric. We presume that the best metric is the one that has the highest sum of ten profits.

This procedure is repeated for different samples taken from the dataset's column-pairs. We vary the number of records in the samples and calculate their *sampling fraction*. The sampling fraction is the ratio of number of records in the sample over the number of records in the entire dataset. For instance, in a dataset with 10 000 records, a sampling fraction of 0.1 has 1 000 records.

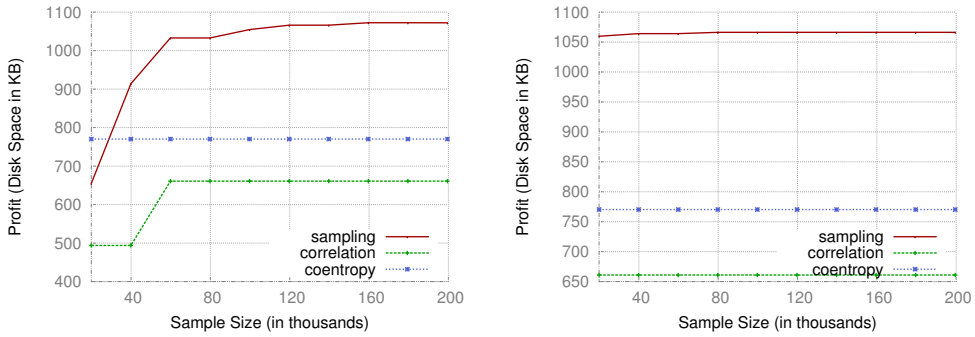
6.2.2 The Results

Having explained our experimental procedure we present our results in this section. In Figure 6.1, we show the sum of the top ten profits of the best column pairs with each of our metrics, namely, coentropy, mean square contingency referred to as "correlation", and our sample-based metric referred to as "sampling" in the legend of the figure. The column pairs were obtained from the Census Income dataset, where all columns had a cardinality ratio less than 0.0005, except for one column with a cardinality ratio above 0.5.
Neither of the statistical metrics presented was as effective in predicting what column pairs would produce the best profit as our sample-based metric. This can be observed in Figure 6.1 where the curve labeled as "sampling" has the highest profit for most sample sizes. The sample-based metric was also more consistent than the other metrics.
Yet, the prediction made by sampling was not as reliable for a small sample where the cardinality of one of the columns was greater than the number of records in the sample. This effect can be observed in Figure 6.1a for sample

sizes with less than 20 000 records.

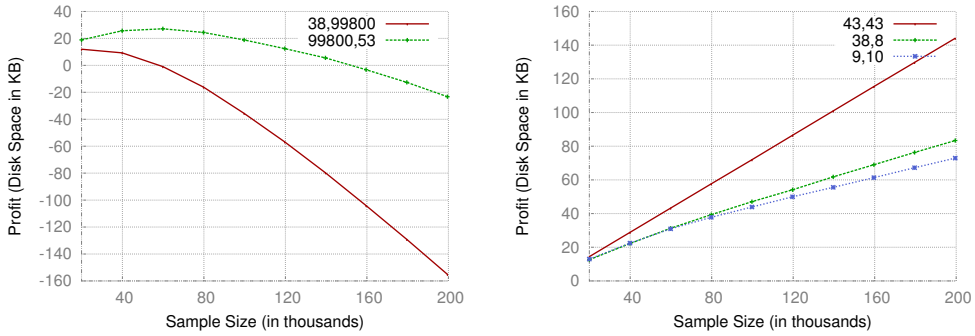
In fact, for a sampling fraction of 0.1 (labeled in the x axis as $x = \lfloor 0.1 \times 199523 \rfloor = 19952$) when a column with a cardinality ratio above 0.5 is involved, coentropy is a better metric (see Figure 6.1a). In Figure 6.1b, we repeat the plot omitting all pairs involving the column with a cardinality ratio above 0.5 and noticed that when the sampling fraction was 0.1, the profit obtained through sampling improved and overcame that of the coentropy measure. In Figure 6.2, we graphed the profits for selected individual column pairs as the size of the sample varied. Notice that when the high cardinality column (above 0.5) is present the profit may initially slowly increase only to decrease rapidly as the size of the sample is increased (see Figure 6.2a). When the high cardinality column is absent, the profit appears to increase linearly and consistently as the size of the sample is increased (see Figure 6.2b). Notice that the sampling behavior of low cardinality columns, below 0.0005, appears to be more predictable than that of high cardinality columns, above 0.5.

We conclude that overall, **the best approach for determining the best candidate pairs for building a multicolumn index is using profit based sampling.** While there may be cheaper methods to compute the correlation between pairs of columns, we did not find any. Finding such metrics could be left as future work.



(a) High cardinality column included (b) High cardinality column absent

Figure 6.1: Sum of top ten profits attained with column pairs predicted with our metrics.



(a) High cardinality column included (b) High cardinality column absent

Figure 6.2: Profits of individual column pairs as sample size is increased. Each key in the legend illustrates the cardinalities of the column pairs as L_1, L_2. For instance, one column-pair had a column with 38 values and a column with 99 800 values.

6.3 Finding the Right Sample Size

We want the sample size to be as small as possible since it saves computation time. However, the sample size should also reliably portray the entire dataset accurately. In this section, we concern ourselves with finding a sample size as small as possible that can give us accurate predictions.

Using the profit of the sample as a metric tends to lead to inaccurate results when either the individual or cocardinality of the columns is greater than the number of records in the sample. The higher the cardinality of the column, the larger the sample needed to make a sound prediction. When taking a small sample from a column with many possible attribute values, the sample will tend to overestimate the cardinality of the column. To illustrate this point, consider taking a sample of 10 records from a 10 000 record column with a uniform distribution. With such a sample, it is not possible to differentiate between an attribute cardinality of 100 and 1 000. We expect that the cardinality measured in the sample would usually be 10 for both cardinalities as all values sampled would be different. The probabilities for repeating a value in the column with cardinality of 100 and 1 000 are $1 - (99/100)^{10} \approx 0.10$ and $1 - (999/1\,000)^{10} \approx 0.01$ respectively. The prediction could be relatively accurate if the ratio of cardinality to the number of records is one, meaning that the column has unique attribute values. Yet, there is no way of knowing from the sample alone if that is the case; thus, the sample-based prediction made would normally be inaccurate.

Considering the columns' cardinalities individually can give us an idea of what the sample size should be to portray a column accurately. However, because for a multi-column bitmap index we consider the two columns together, it is not sufficient to have a sample size as large as the largest column cardinality. The column's individual cardinalities may be small but when considered together the cocardinality may be considerably larger. A larger sample may be required in order to portray the dataset accurately, as the sample size based on the column with the largest cardinality may overestimate the cocardinality of the sample.

Having concluded that column cardinality and cocardinality affect the reliability of a sample size raises the question of how many observations are necessary for a sample to be reliable. We attempted to answer this question analytically but it proved to be difficult. We close this section by concluding that the cocardinality of the columns plays an important role in choosing the right sample size for a particular dataset.

6.4 Summary

We concluded that order-oblivious statistics, such as mean-square contingency and coentropy, are invariant to dataset transformations, such as row reordering and frequency multiplication. Because dataset transformations are a determining factor in the size of multi-column and single-column bitmap indices, order-oblivious statistics may not be effective. We also realized that

making predictions on sorted tables with sample-based metrics is considerably more difficult. More sophisticated sampling techniques are required for ordered tables than for unordered tables.

We presented an experimental analysis using datasets that have real data. Through this experimental analysis we compared mean-square contingency, coentropy and a sample-based metric by their effectiveness in determining what column pairs should be indexed as a single-column bitmap index and which as a multi-column bitmap index. Calculating the statistical metrics on the entire dataset is nearly as expensive as building the bitmap indices on the dataset. Therefore, we calculated them on randomized samples from our datasets. Our sample-based metric also uses a randomized sample and calculates the size difference between a multi-column and single-column bitmap index built on that sample. The size difference is then used to determine whether a column pair is a good candidate for a multi-column bitmap index. In our results, we concluded that overall the best approach for determining the best candidate pairs for building a multi-column index is through our sample-based metric. We also concluded that the column's cardinalities affect the reliability of the sample size used for our sample-based metric. The higher the cocardinality, the larger the sample must be in order to obtain a reliable sample size.

Chapter 7

Sampling Heuristics

In Chapter 6, we realized that our sample-based metric works better than our correlation-based metrics in determining how a column-pair should be indexed. In this chapter, we put our efforts in proposing heuristics that can tell us whether, as far as size is concerned, when it is best to use a multi-column or single-column index.

In this chapter, we present some novel heuristics. The single-sample heuristic consists in taking a fraction of the data as a sample and making a prediction for the entire dataset based on it. Our correlation-based heuristic uses Pearson's correlation on a number of samples and their respective profits.

In our heuristics, we are allowed to use up to a certain fraction of the entire dataset. This fraction, denoted as f, depends on the user's context which involves the size of the entire dataset and the time allowed for the computation. This parameter can be chosen by a user who can then better impose

the restrictions, but we use as a default a fraction of 0.1 for datasets with more than 100 000 records. While sampling half the dataset is not of practical sense for large datasets, we used half of the Tweed dataset ($f = 0.5$), because for a fraction of 0.1 there were not enough records to make sufficiently large samples for EWAH compression to be effective.

In Section 7.1 we introduce our single-sample heuristic that uses a single sample to predict when a column pair should be indexed as a multi-column bitmap index and vice versa. With that same purpose, in Section 7.2 we introduce our correlation-based heuristic that uses Pearson's correlation to make the prediction. In Section 7.3, we compare our single-sample heuristic against two variations of the correlation-based heuristic through two computational experiments and conclude that our correlation-based heuristic with two iterations was the best heuristic for predicting how a column pair should be indexed. Lastly, in Section 7.4 we summarize our conclusions.

7.1 Single-Sample Heuristic

We first present the single-sample heuristic which can be summarized by the following steps:

- Generate a randomized sample from the dataset with nf records
- Calculate the sample's profit
- Choose a multi-column or single-column bitmap index for the column-

pair based on the sample's profit.

This heuristic tends to work well in situations where a sample correctly represents the entire dataset. However, this heuristic ignores the direction in which the profit and sampling relationship is headed when the sample size increases. Instances where samples' profits are initially negative, but are headed for a positive outcome, could make this heuristic fail. We assess experimentally how likely such instances are in Section 7.3.

Algorithm 2 Correlation-based heuristic

1: $\{s$ is the sample size$\}$
2: $\{\iota$ is the number of iterations$\}$
3: $\{n$ is the total number of records in the dataset$\}$
4: $\{f$ is the fraction of records taken from the dataset$\}$
5: $f \leftarrow 0.1$
6: $s \leftarrow \lfloor nf \rfloor$
7: Shuffle the order of the records in the dataset
8: **for each** iteration i from 0 to $\iota - 1$ iterations **do**
9: Build single-column bitmap index α using first s records
10: Build muti-column bitmap index β using first s records
11: Add s to Vector S
12: Add s's profit $\mathrm{DS}(\alpha) - \mathrm{DS}(\beta)$ to Vector P
13: Let $s \leftarrow s/2$
14: **end for**
15: Calculate Pearson's correlation ϕ using S and P
16: **if** $\phi > 0$ **then**
17: Choose β for the column pair
18: **else**
19: Choose α for the column pair
20: **end if**

7.2 Correlation-Based Heuristic

In this section, we present a heuristic that is based on Pearson's correlation. We use Pearson's correlation to predict the trend that profit is taking in anticipation of predicting the full dataset's profit. Possible future work could involve using the linear regression equation to estimate the full dataset's profit. This heuristic uses a number of samples taken from the dataset. As with the single-sample heuristic, an initial sample is built by taking the first nf records from the shuffled dataset. Smaller samples embedded in the original sample are taken by extracting the first $nf/2^i$ rows, where i is an iteration. More specifically, each iteration of our heuristic uses a sample that has half the number of records of the sample used in the previous iteration. With each iteration, a single-column and multi-column bitmap index are built for each sample of size s. The sample sizes used are pushed to a vector of sizes denoted S. This vector S is also referred to as the sampling schedule. We then measure the multi-column and single-column bitmap indices' sizes in bytes as DS(β)and DS(α), respectively. We obtain a positive or negative profit depending on whether β is smaller than α, and add it to a vector of profits denoted P. Finally, having vectors S and P we calculate Pearson's correlation ϕ. If ϕ is greater than 0 then we choose β as the appropriate bitmap index. Otherwise, we choose α as the appropriate bitmap index for the given column pair. The pseudocode for this heuristic is presented in Algorithm 2.

The number of records used in our sample schedule is $\sum_{i=0}^{\iota-1} \frac{nf}{2^i}$ which is greater than the number of records used in our single sample heuristic. Alternatively, the sampling schedule of this heuristic can be modified to be based on the sum of the samples' lengths. This change will cause both the single-sample heuristic and correlation-based heuristic to use the same total number of records.

With a limit on the number of records that we may use for sampling, the number of iterations and initial sample size in the sampling schedule can be determined. These two parameters offer an important trade-off. Increasing the number of iterations involves decreasing the average sample size and decreasing the number of iterations increases the average sample size. Because we prefer samples to be large as they tend to be more representative of the original dataset, we keep the number of iterations at two or three. Three iterations give us a sense of the potential accuracy of the prediction while two iterations give us larger samples.

By fixing the number of iterations, we can calculate the initial sample size and complete the sampling schedule in ι iterations. Given a sample fraction f, we can calculate the initial sample size s as $\frac{nf}{2^\iota-1}$. To switch to the sampling schedule based on the sum of the sample's lengths, the following two changes are need in Algorithm 2:

- Change line 6 to initialize sample size s as $\frac{nf}{2^\iota-1}$
- Change line 13 to double on each iteration ($2s$).

We also tested experimentally this alternative sampling schedule as with it our heuristic would be closer in computational time to the single-sample heuristic. Our results did not differ much from the original sampling schedule of our heuristic.

7.3 Experiments

We test our heuristics with all the datasets presented in Table 6.1. To determine when a heuristic is successful we compare the choices made by the heuristic with the optimal choices. The optimal choices are made based on the profit of the entire dataset. As with our heuristics, the optimal choice uses a single-column bitmap index if the entire dataset's profit is negative and a multi-column bitmap index if it is positive. We count as correct predictions the number of times where the optimal choice and the choice made by the heuristic are the same. Conversely, we count the number of times when they differ as incorrect predictions. We then calculate a score to determine the efficiency of the heuristic as follows:

$$\text{SCORE} = \frac{\text{CORRECT PREDICTIONS}}{\text{INCORRECT PREDICTIONS} + \text{CORRECT PREDICTIONS}} \times 100.$$

We calculated the scores for all our heuristics on all datasets and summarize them in Table 7.1. Each cell of this table is the average score attained by a heuristic on a particular dataset in three different trials. We processed

the dataset using Pearson's correlation-based heuristic with two and three iterations. Each trial involved shuffling the dataset once and allowing our different heuristics to work on that particular dataset. As shown in Table 7.1, for most datasets, Pearson's correlation-based heuristic with two iterations had the highest score making it the best heuristic under this scoring system. With the exception of Netflix, the difference among the heuristic's scores was within 5%. The single-sample heuristic was often slightly less effective than the other heuristics. Because the single-sample heuristic scored relatively high in accuracy for most datasets, we conjecture that it is rare to find an instance where negative sample profits are headed for a positive outcome or vice versa.

We also calculate the percentage by which a heuristic, $\mathcal{H}$, saves disk space as opposed to always choosing a single-column bitmap index. We start by totaling the disk space occupied when choosing a single-column bitmap index for every column pair, and the disk spaced occupied when either choosing a multi-column or single-column bitmap index, as recommended by $\mathcal{H}$. Let the total disk space occupied when choosing all α be $\sum \mathrm{DS}(\alpha)$ and the total disk space occupied when choosing α or β based on the $\mathcal{H}$ heuristic be $\sum \mathrm{DS}(\mathcal{H})$. The percentage of disk space saved when using a particular heuristic can be calculated as follows:

$$\text{PERCENTAGE SAVED} = \frac{\sum \mathrm{DS}(\alpha) - \sum \mathrm{DS}(\mathcal{H})}{\sum \mathrm{DS}(\alpha)} \times 100.$$

Table 7.1: Average scores on three trials when using 10% sample size (50% for Tweed)

Heuristic	CIP1	CIP2	Census1881	Netflix	Tweed
Single Sample	94	92	97	66	**98**
Correlation-based ($\iota = 2$)	**98**	**93**	**100**	**89**	**98**
Correlation-based ($\iota = 3$)	96	93	**100**	**89**	**98**

Table 7.2: Average percentage of space saved on three trials when using 10% size (50% for Tweed)

Heuristic	CIP1	CIP2	Census1881	Netflix	Tweed
Single Sample	3.65	3.29	16.58	-15.96	**2.20**
Correlation-based ($\iota = 2$)	**3.98**	3.33	**16.62**	**1.25**	2.19
Correlation-based ($\iota = 3$)	3.05	**3.80**	16.58	**1.25**	2.08

The percentage saved is positive when there is profit—that is our heuristic's bitmap indices occupy less disk space than simply always choosing a single-column bitmap index. By contrast, a negative percentage indicates that instead of taking less disk space, our heuristic takes more disk space, and instead of a profit in disk space we have a loss.

We averaged the percentages for three trials using our heuristics with all datasets and summarized the results in Table 7.2. The correlation-based heuristic with two iterations shows to be the most effective on average by a small fraction when compared to the other heuristics.

In most datasets our heuristics helped, reducing the amount of disk space occupied by 2% to 17% when building indices for all column pairs. How-

ever, the single-sample heuristic had a negative percentage on the Netflix dataset. This percentage means that instead of taking less disk space, it took nearly 16% more disk space. The single-sample heuristic made only one more incorrect choice than the other heuristics but this choice turned out to be an expensive mistake. For a column pair in Netflix, the single-sample heuristic chose a multi-column bitmap index. The correct choice would have been a single-column bitmap index, which is around 43% smaller than the multi-column bitmap index. Thus, while the heuristic had other gains from other pairs, it could not recover the amount of disk space lost because of this incorrect choice.

7.4 Summary

Having concluded from Chapter 6 that sample-based heuristics are more efficient than heuristics based on mean square contingency and coentropy, we presented novel sample-based heuristics that determine whether a column pair is better indexed by a multi-column or single-column bitmap index in terms of size.

The single-sample heuristic consists in taking a fraction of the data as a sample and making a prediction for the entire dataset based on it. We also presented a correlation-based heuristic that uses Pearson's coefficient of correlation to make the prediction. Unlike the single-column bitmap index, this heuristic has a sense of direction of where the profit might be headed. We

compared our heuristics through two experiments that measured effectiveness differently. Overall, our best heuristic for determining how a column pair should be indexed was the Pearson's correlation-based heuristic.

Chapter 8

Conclusions and Future Work

We conclude by summarizing our work as follows:

In Chapters 1 and 2, we introduced bitmap indices for improving the speed of answering queries in large databases or data warehouses. The size of a bitmap index can grow considerably as attribute cardinality increases, thus encoding and compression techniques are often used to reduce their size. In many cases, researchers have shown that compressed bitmap indices can even be suitable for high cardinality columns. We focused our research on bitmap indices with trivial encoding and compressed them using EWAH.

Two types of bitmap indices were explored, namely, single-column bitmap indices and multi-column bitmap indices. Single-column bitmap indices are built by considering the attribute values of a table's columns separate.

Multi-column bitmap indices are built by considering each distinct tuple of a table as an attribute value. While a multi-column bitmap index may have

more bitmaps than a single-column bitmap index, the bitmaps themselves may be more compressible. Consequently, multi-column bitmap indices can be smaller than single-column bitmap indices when compressed. We attempted using statistical characteristics for determining which datasets are better suited for a multi-column or single-column bitmap index.

We conjectured that datasets with highly correlated columns are good candidates for multi-column bitmap indices. Following our conjecture, in Chapter 3, we introduced the mean square contingency and coentropy correlation measures, which are used to determine the likelihood that two columns are statistically independent. We also introduced two novel correlation measures based on cardinality; one based on the maximum of the cardinalities, and the other based on the product. We later used our correlation measures as statistical metrics in experiments with realistic data.

In Chapter 4, we concluded that row-order matters greatly in determining size for multi-column or single-column bitmap indices and thus order-oblivious statistics can merely help in predicting the number of bitmaps. Knowing the number of bitmaps for a single-column or multi-column bitmap index is not enough for determining size.

Later, in Chapter 5, we presented extreme cases where multi-column bitmap indices are always smaller than single column bitmap indices. We analyzed how the number of columns affected the size of multi-column and single-column bitmap indices. We also analyzed how statistical parameters such as cardinality, skew and different forms of correlation affected the sizes of

the two different types of indices. We found from experiments on synthetic datasets that increasing the number of columns, the cardinality or correlation often favors multi-column bitmap indices. We also found that increasing the correlation factor in our synthetic experiments decreases the size of multi-column bitmap indices, and that when the mean square contingency is 1.0, the multi-column bitmap index is always smaller than the single-column bitmap index. From experiments where we varied the skew parameter to convert an initially uniform distribution to a Zipfian distribution, we found that Zipfian distributions tend to favor single-column bitmap indices as the skew parameter is increased.

In Chapter 6 we introduced a notion of profit that allows us to compare decisions on indexing column pairs with a multi-column versus a single-column bitmap index. We compared the coentropy and mean-square contingency statistical metrics explained in Chapter 3 by looking at their total profit as we increased the sample fraction and found coentropy to be better. However, while statistical parameters like correlation show an effect on the behavior of multi-column and single-column bitmap indices, they were not as useful in helping us predict when a column pair was more suitable for a single or multi-column bitmap index on realistic data. We noticed that indexing samples was a better alternative.

Consequently, in Chapter 7 we presented our novel sampling heuristics to make a decision on indexing a column pair as a multi-column or single-column bitmap index. We compared our heuristics through two experiments

that measured effectiveness differently. Overall, our Pearson's correlation-based heuristic was the best in determining how a column pair should be indexed.

We do not usually think of bitmap indices as well suited to index high cardinality columns. Hence, it follows that multi-column bitmap indexing may appear to be a poor idea because it effectively considers several columns as one high cardinality column. Yet our work shows that, at least as far as size is concerned, these concerns are often unwarranted. In many cases, multi-column indices are smaller than corresponding single-column indices. It remains to expand our work from merely comparing the size of the indices, to actual performance under specific workloads.

Bibliography

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 671–682, New York, NY, USA, 2006. ACM.

[2] G. Antoshenkov. Byte-aligned bitmap compression. In *Data Compression Conference (DCC'95)*, page 476, Washington, DC, USA, 1995. IEEE Computer Society.

[3] L. Bellatreche, K. Boukhalfa, and M. Mohania. Pruning search space of physical database design. In *18th International Conference on Database and Expert System Applications (DEXA'07)*, pages 479–488, 2007.

[4] L. Bellatreche, R. Missaoui, H. Necir, and H. Drias. A data mining approach for selecting bitmap join indices. *Journal of Computing Science and Engineering*, 2(1):206–223, 2008.

[5] C. Y. Chan and Y. E. Ioannidis. Bitmap index design and evaluation.

In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 355–366, 1998.

[6] C. Y. Chan and Y. E. Ioannidis. An efficient bitmap encoding scheme for selection queries. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 215–226, 1999.

[7] S. Chaudhuri, R. Motwani, and V. Narasayya. Random sampling for histogram construction: how much is enough? *SIGMOD Record*, 27:436–447, June 1998.

[8] T. Eavis and D. Cueva. A Hilbert space compression architecture for data warehouse environments. In *Data Warehousing and Knowledge Discovery (DaWaK'07) (LNCS 4654)*, pages 1–12, Berlin, Heidelberg, 2007. Springer.

[9] R. Foote. Oracle blog. `http://richardfoote.wordpress.com/2010/05/12/concatenated-bitmap-indexes-part-ii-everybodys-got-to-learn-sometime`, 2010. (Last checked 03-04-2012).

[10] F. Fusco, M. P. Stoecklin, and M. Vlachos. NET-FLi: On-the-fly compression, archiving and indexing of streaming network traffic. In *VLDB'10, Proceedings of the 36th International Conference on Very Large Data Bases*, San Jose, CA, USA, 2010. VLDB Endowment.

[11] S. W. Golomb. Run-length encodings. *IEEE Transactions on Information Theory*, 12:399–401, 1966.

[12] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. *SIGMOD Record*, 23:243–252, May 1994.

[13] S. Hettich and S. D. Bay. The UCI KDD archive. `http://kdd.ics.uci.edu` (checked 01-02-2012), 2000.

[14] K. Houkjær, K. Torp, and R. Wind. Simple and realistic data generation. In *VLDB'06, Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 1243–1246, San Jose, CA, USA, 2006. VLDB Endowment.

[15] I. F. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulnaga. CORDS: automatic discovery of correlations and soft functional dependencies. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 647–658, 2004.

[16] D. Lemire and O. Kaser. Reordering columns for smaller indexes. *Information Sciences*, 181:2550–2570, June 2011.

[17] D. Lemire, O. Kaser, and K. Aouiche. Sorting improves word-aligned bitmap indexes. *Data & Knowledge Engineering*, 69(1):3–28, 2010.

[18] Netflix, Inc. Netflix prize README. online: `http://www.netflixprize.com/community/viewtopic.php?id=68` (checked 05-23-2011), 2007.

[19] P. O'Neil, E. O'Neil, X. Chen, and S. Revilak. The star schema benchmark and augmented fact table indexing. In *Performance Evaluation and Benchmarking (LNCS 5895)*, pages 237–252, Berlin, Heidelberg, 2009. Springer.

[20] P. O'Neil and D. Quass. Improved query performance with variant indexes. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 38–49, 1997.

[21] P. E. O'Neil. Model 204 architecture and performance. In *2nd International Workshop on High Performance Transaction Systems*, pages 40–59, London, UK, 1989. Springer-Verlag.

[22] M. Paradies, C. Lemke, H. Plattner, W. Lehner, K.-U. Sattler, A. Zeier, and J. Krueger. How to juggle columns: an entropy-based approach for table compression. In *International Database Engineering and Applications Symposium (IDEAS'10)*, pages 205–215, 2010.

[23] Programme de recherche en démographie historique. PRDH 1881. `http://www.prdh.umontreal.ca/census/en/main.aspx`, 2009. last checked 2011-12-19.

[24] D. Salomon. *Data Compression: The Complete Reference*. Springer, 4th edition, December 2006.

[25] C. E. Shannon. A mathematical theory of communications. *Bell System Technical Journal*, 1948.

[26] V. Sharma. Bitmap index vs. B-tree index: Which and when? online: `http://www.oracle.com/technology/pub/articles/sharma_indexes.html`, March 2005.

[27] Y. Sharma and N. Goyal. An efficient multi-component indexing embedded bitmap compression for data reorganization. *Information Technology Journal*, 7(1):160–164, 2008.

[28] I. Spiegler and R. Maayan. Storage and retrieval considerations of binary data bases. *Information Processing & Management*, 21(3):233–254, 1985.

[29] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-Store: a column-oriented DBMS. In *VLDB'05, Proceedings of the 31st International Conference on Very Large Data Bases*, pages 553–564, San Jose, CA, USA, 2005. VLDB Endowment.

[30] H. Webb, O. Kaser, and D. Lemire. Evaluating multidimensional queries by diamond dicing. *Computing Research Repository*, abs/1006.3726, 2010.

[31] H. K. T. Wong, H. F. Liu, F. Olken, D. Rotem, and L. Wong. Bit transposed files. In *VLDB'85, Proceedings of the 11th International Conference on Very Large Data Bases*, pages 448–457, 1985.

[32] K. Wu, E. Otoo, and A. Shoshani. On the performance of bitmap indices for high cardinality attributes. In *VLDB'04, Proceedings of the 30th International Conference on Very Large Data Bases*, pages 24–35, 2004.

[33] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems*, 31(1):1–38, 2006.

[34] K. Wu, K. Stockinger, and A. Shoshani. Performance of multi-level and multi-component compressed bitmap indexes. available from `http://www.osti.gov/bridge/servlets/purl/920347-LbONsx/`, 2007.

[35] K. Wu, K. Stockinger, and A. Shoshani. Breaking the curse of cardinality on bitmap indexes. In *SSDBM'08: Proceedings of the 20th International Conference on Scientific and Statistical Database Management*, pages 348–365. Springer, 2008.

[36] M. Zaker, S. Phon-Amnuaisuk, and S. Haw. An adequate design for large data warehouse systems: Bitmap index versus B-tree index. *International Journal of Computers and Communications*, 2(2):39–46, 2008.

[37] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

Curriculum Vitae

Candidate's full name: Eduardo Gutarra Velez

University attended:

Bachelor of Computer Science
EAFIT University Jun 2009
Medellín, Antioquia, Colombia
G.P.A. 4.3/5.0

Publications:
Daniel Lemire, Owen Kaser and Eduardo Gutarra, "Reordering Rows for Smaller Indexes: Beyond the Lexicographic Order", manuscript under review, submitted 11 November 2010.