ANALYTIC QUERY PROCESSING AT BARE METAL SPEEDS

by

Yinan Li

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2015

Date of final oral examination: 08/17/15

The dissertation is approved by the following members of the Final Oral Committee:
Jignesh M. Patel, Professor, Computer Sciences
David J. DeWitt, Emeritus Professor, Computer Sciences
Mark D. Hill, Professor, Computer Sciences
Jeffrey F. Naughton, Professor, Computer Sciences
Robert D. Nowak, Professor, Electrical and Computer Engineering

© Copyright by Yinan Li 2015
All Rights Reserved

To Di, for all she went through.

Acknowledgments

Foremost, I would like to take this opportunity to express my appreciation and gratitude to my advisor, Jignesh Patel. During my Ph.D. study, I benefited tremendously from his energy, his industrial insight, his communication skills, his patience, and his generosity. He has given me a lot of freedom in choosing research topics, and has always encouraged me to think big and take risks. Without him, none of the work in this dissertation would even exist. I could not have asked for a better advisor than Jignesh, and I am forever indebted to him for his role in my life. Thank you, Jignesh.

I feel extremely fortunate to have David DeWitt, Mark Hill, Jeff Naughton, and Robert Nowak on my committee. Jeff gave me numerous constructive comments on this dissertation. Without his key suggestions during my preliminary exam, some most interesting sections in this dissertation may not exist. I also appreciate all the stories and the jokes (about many legendary researchers) he told in discussions, seminars, and classes over the years. I met David relatively late in my Ph.D. study. In spite of that, I benefited tremendously from his sharp and insightful comments on my presentations and on this dissertation. David also influenced me by his passion and enthusiasm, as well as his adventurous spirit. Mark provided invaluable comments from a computer architect's perspective. I was also influenced by his curiosity, his attitude towards work, and his dedication to his research. Finally, thanks to Robert for his time and effort to help me improve this dissertation.

Thanks to all the members of the Quickstep team at Wisconsin: Shoban Chandrabose, Craig Chasseur, Harshad Deshmukh, James Paton, Adalbert G. Soosai Raj, Qiang Zeng, and Zuyu Zhang. Without their hard-working and commitment to this project, some key parts of this dissertation could not be completed. Thanks especially to Craig Chasseur, who coded the initial version of the Quickstep engine, and worked closely with me on some components of the engine.

The database group at University of Wisconsin-Madison has largely influenced my research philosophy and style. I was extremely fortunate to have two Wisconsin alumni serving as my primary advisors for both Ph.D. and master's degree. During my Ph.D. study at Wisconsin, David DeWitt, AnHai Doan, Jeffrey Naughton, Jignesh Patel, and Christopher

Ré helped me grow both professionally and personally in many different ways. I also want to thank other current or former members of the Wisconsin database group: Adel Ardalan, Spyros Blanas, Xiaoyong Chai, Craig Chasseur, Fei Chen, Ting Chen, Jaeyoung Do, Xixuan Feng, Avrilia Floratou, Chaitanya Gokhale, Yeye He, Arun Kumar, Willis Lang, Jiexing Li, Feng Niu, Erik Paulson, Ian Rae, Chong Sun, Khai Tran, Ba-Quy Vyong, Wentao Wu, Chen Zeng, Qiang Zeng, Ce Zhang, and Ning Zhang. They provided amazing work and life environment for my Ph.D. study.

During my study, I have been lucky enough to intern at Facebook and two research labs: Microsoft Research and IBM Almaden Research Center. I thank Badrish Chandramouli, Jonathan Goldstein, Paul Larson, and David Lomet at Microsoft Research; Ronald Barber, Guy Lohman, René Müller, Ippokratis Pandis, Hamid Pirahesh, Vijayshankar Raman, and Richard Sidle at IBM Almaden Research Center; Sam Rash, and Zheng Shao at Facebook, for their vast knowledge and industrial perspective. Working closely with these talented researchers and engineers, I experienced many real examples of turning research prototype systems into production tools and software that makes contributions beyond the academic community.

I would like to especially thank Qiong Luo, who served as my advisor for master's degree and guided me to the field of database systems research. Without her help, support, patience, and encouragement, I may have chosen another research area, or even have given up my application for Ph.D. degree. I am very grateful that I have been granted the opportunity to work with her.

In addition, I would like to thank National Science Foundation for its support of our research. I also feel very grateful to have my research supported by a Facebook Fellowship and the Anthony Klug NCR Fellowship in Database Systems.

Finally but most importantly, I would like to express my deepest thank you to my family for their unconditional support and their endless faith in me. I am forever indebted to my parents for supporting me in all my pursuits and raising me with a love of science and an interest in the way things work. Then there is my wife, Di Wang, whom I met near the fantastic Lake Mendota in Madison. From then on, she gave me her support without reservation, through the good times and the bad. Di, thank you for all the sacrifices that you have made for me through this long journey.

Contents

List of Tables

List of Figures

Abstract

Modern large-scale data processing platforms require techniques to process data at high speeds, as there is an arms race towards building and deploying near real-time analytical systems. The primary goal of this dissertation is to design fast analytical data processing techniques that aim to process data at or near the speed of the underlying hardware.

In this dissertation, we describe several techniques that can improve the speed and scalability of interactive analytic data processing systems. First, we present a fast scan method, which is a core operation in interactive analytic data engines. The proposed scan method, called BitWeaving, exploits the parallelism available at the bit level in modern processors, and operates on multiple compressed data items in each processor instruction. Second, we describe an encoding scheme that turns skew in both the data and predicate distributions into a benefit for scan performance. This scheme creates encodings that map frequent data items to (compressed) codes that can easily be distinguished from other codes by only examining a few bits in the full code. Consequently, the encoding scheme reduces the memory bandwidth and CPU costs that are consumed when evaluating scans. Third, given these fast scan algorithms and other technical trends, we believe that an old idea, namely denormalization, now becomes more practical in modern analytical systems. The last part of the thesis presents a technique called WideTable, which uses aggressive denormalization to flatten a database schema into one or more "wide tables", and converts complex join queries to simple and efficient (BitWeaved) scans on the denormalized table. To avoid the pitfalls associated with denormalization, e.g. large space overhead, WideTable uses a combination of techniques including dictionary encoding and columnar storage.

Finally, we benchmark the performance of the proposed techniques in the Quickstep database system. We experimentally evaluate our methods, and compare them to the leading open-source main memory DBMS in a main memory setting using the TPC-H and the star schema benchmarks. We find that our methods outperform the traditional method for nearly all queries, with over an order of magnitude in performance improvement on a majority of queries. In addition, our methods also show better scalability on modern multi-core CPUs.

Chapter 1

Introduction

Analytical data query processing systems today face a host of challenges and opportunities that were not as prominent just a few years ago.

First, the hardware landscape has changed dramatically in recent years. Modern server systems adopt multi-socket multi-core CPU architecture and NUMA-based memory architecture. Today's database server systems often contain 4 to 8 sockets, each of which has up to 20 CPU cores. Consequently, it is not uncommon to execute tens to hundreds of concurrent threads in a single database server system. In addition, large main memory configurations are now very common and affordable. Commodity servers with hundreds of gigabytes to terabytes of DRAM are increasingly economical. Memory densities are predicted to continue to grow over the upcoming decade, and fully or mostly in-memory processing is increasingly common for many analytical applications. This technological trend has led to the resurgence of main memory databases and has pushed in-memory computing to the forefront of data analytics products that are offered today. Many systems have been developed to meet this growing requirement in both the research community, e.g. MonetDB [16, 15, 41], Blink [80], Hyper [48], Spark [103], Shark [99], and in the industry, e.g. Vectorwise [106], SAP HANA [27], Oracle Exalytics [72] and IBM DB2 BLU [79]. These analytical data processing systems have developed a couple of key techniques over the last two decades, which are different from the traditional DBMS model that traces its roots back to System R [9] days. These key techniques include using columnar storage organization, optimizing relational operations for CPU cache/TLB performance, and employing a vectorized query execution model. However, the fundaments of how query plans are assembled and executed has largely remain unchanged. In other words, there hasn't been quite a "re-thinking from basics" for query processing in these main-memory analytical data processing environments.

Second, there is increasing need for high performance on analytic workloads. The

focus on performance is driven by a number factors, including closer integration of data analytics with decision-making in enterprises, which naturally leads to a demand for interactive analytics. Interactive response times often make a huge difference in data science, monitoring, advertisement, rapid prototyping, debugging of data pipelines, and other key data processing applications. This increasing demand can be seen by the recent interest in building large-scale interactive analytics systems like Google Dremel [69] and Facebook Scuba [5]. In addition, performance is also critical as the move to cloud settings (both public and private clouds) has made "accountability" a bigger emphasis for users of the analytics systems. For example, when running analytics in rented public clouds, higher performance translates to lower monetary cost for running the data analytics service. Even when running in private clouds, there is an increasing trend to monitor the usage of the cloud resources for groups within the enterprise and to "bill" each group to create increased awareness about the costs that each group is incurring.

Finally, there is a shift in many data analytics environments to read-mostly settings. Data that is loaded into such warehouses is often not updated in-place, and new data is often appended to the existing database. This shift can be seen by the huge interest in moving SQL-based workloads to Hadoop-based systems like Impala [52], Hive on Tez [38], Apache Drill [68], and HAWQ [17]. Since these systems are built on top of the Hadoop Distributed File System (HDFS), and since HDFS is append-only, these systems naturally have to live with this read-mostly constraint. But, it turns out that many warehousing scenarios are amenable to this paradigm of read-mostly settings.

A natural question is whether the technological factors listed above require us to rethink how we build modern analytical systems. The main focus of this dissertation is to design, implement, and experiment with analytical data processing techniques that meet these requirements. More specifically, we aim to build systems to leverage these opportunities and run analytical queries at "bare metal" speeds. Essentially, this means that the system must aim to process data at or near the speed of the underlying hardware. There is a huge gap, often more than a order of magnitude, between the highest performance that analytical database systems deliver today and the raw speed of the underlying hardware. The primary goal of this dissertation is to close this gap, producing fast analytical data processing techniques.

In order to exploit the full potential of the hardware, this dissertation takes the approach of thinking bottom-up from the hardware to the software. This hardware-software co-design approach naturally forces us to compare the raw performance of the hardware and the end performance delivered by data analytics systems running on this hardware to ask if there are technical mechanisms that could be developed to allow for the software to

deliver bare-metal performance.

Following this philosophy, this dissertation first develops key data processing kernels (e.g. implementation of selection operator algorithms) that can run at the speed of the underlying hardware, and then finds ways to compose them efficiently to answer more complex queries in a data processing system.

As the first aspect in this dissertation, we have designed a fast scan method, called BitWeaving [64], which is a core data processing kernel in large-scale analytic data processing systems. As background for BitWeaving, we note that a lot of attention has been paid to running scans efficiently, including using column stores and using SIMD-based parallelism. With the state-of-the-art techniques, it still takes many cycles per input value to apply simple predicates on a single column of a table. The BitWeaving technique, however, exploits the parallelism available at the bit level in modern processors. BitWeaving operates on multiple bits of data in a single cycle, processing bits from different column values in each cycle. Thus, bits from a batch of tuples are processed in each cycle, allowing BitWeaving to drop the cycles per column to below one in some case. BitWeaving comes in two flavors: BitWeaving/V which looks like a columnar organization but at the bit level, and BitWeaving/H which packs bits horizontally. We also developed an arithmetic framework that is needed to evaluate predicates using these BitWeaved organizations. Our experimental results show that both these methods produce significant performance benefits over the existing state-of-the-art methods, and in some cases produce over an order of magnitude in performance improvement.

In the next step, we shifted our focus to encoding techniques that have been commonly used with the efficient scan algorithms in in-memory data analytic systems. Data in modern data processing systems is often stored in compressed form using dictionary encoding or other order-preserving encoding schemes. We observe that in such systems, there is an opportunity to turn data skew, a common and well-known phenomenon in real workloads, into a benefit that can be leveraged to further accelerate the scans in a data processing system. In the second part of this dissertation, we propose a dictionary-based encoding scheme, called padded encoding scheme [63], to address this opportunity. The proposed scheme creates encodings that map common attribute values to codes that can easily be distinguished from other codes by only examining a few leading bits in the full code. Consequently, scans on columns stored using the padded encoding scheme can safely prune the computation without examining all the bits in the code, thereby reducing the memory bandwidth and CPU cycles that are consumed when evaluating scan queries. Our padded encoding method results in a fixed-length encoding, as fixed-length encodings are easier to manage. With the design of the padded encoding scheme in place, we develop

an algorithm that can construct an optimal padded encoding. To reduce the algorithmic time complexity, we also propose a more practical padded encoding algorithm that uses heuristics and provides a near-optimal solution.

The main focus of the first two aspects of this dissertations is on the scan primitive, which is the core step in the execution of selection queries. Scans are far simpler to execute, making them crucial to high-performance analytical systems. Nevertheless, analytical applications require the technique to efficiently answer more complex queries, e.g. the queries that contain joins or nested subqueries. Consequently, in the third part of this dissertation, we considered how to expand the "sweet spot" of the bare metal performance of BitWeaving and the padded encoding techniques to a wider variety of analytical query processing, i.e. how to use scan operations to efficiently answer more complex queries.

The answer that we have come up with is called WideTable [65], which uses aggressive denormalization to flatten a database schema into one or more wide tables. Queries on the original database, even complex join queries, now become simple and efficient (BitWeaving) scans on the wide (denormalized) tables. When denormalizing the data, WideTable uses outer joins to ensure that queries on tables in the schema graph, which are now nested as embedded tables in the WideTable, are processed correctly. Although the idea of denormalization is quite old, we observe that it is becoming practical in modern analytical database systems due to several technical trends, as described next. First, in modern analytic environments, it is very common to see read-mostly append-only workloads. For example, nearly all analytical tools in the Hadoop ecosystem belong to this category. Thus, given the read-mostly append-only environments, maintaining a denormalized table is no longer a significant challenge (compared to an environment that allows in-place updates). Second, modern analytical databases tend to store database tables as columns of data, rather than as rows of data. A column-oriented database serializes all of the values of a column (attribute) together. As a result, to evaluate a given query, column-oriented databases only access the column values that are required for the query, regardless of how many columns there are in the underlying table. Consequently, adding more columns when using denormalization is nearly "cost-free" in terms of the query processing cost. Third, modern database systems use encoding techniques to compress data, and thus control the space overhead that is associated with denormalization. Finally, the emergence of efficient scan methods, like BitWeaving, improves the speed of the key access method that is used to answer queries on the denormalized tables.

WideTable and BitWeaving have been implemented in Quickstep – a larger umbrella project that is building a distributed data processing engine that runs at bare metal speeds. Using this implementation, we experimentally evaluate our methods in a main memory

setting using all the 13 queries in the Star Schema Benchmark (SSB). WideTable can run all of the 13 queries in the benchmark. We have also compared WideTable to MonetDB [16, 15, 41], a leading open-source main memory DBMS, and found that WideTable outperforms MonetDB on all queries, with over an order of magnitude in performance improvement on nearly all of the 13 SSB queries. WideTable also shows better scalability than MonetDB on modern multi-core CPUs. We also present more detailed experiments to evaluate the key BitWeaving scan methods, which clear show that the BitWeaving methods produce significant performance benefits over the existing state-of-the-art methods, and in some cases produce over an order of magnitude in performance improvement.

This dissertation is organized as follows.

First, Chapter 2 presents the BitWeaving technique that focuses on running scans in a main memory data processing system at bare metal speed. The purpose of this work is to improve the performance of the most widely used operations – scans in the WideTable approach. In additional, we present an arithmetic framework that is needed to evaluate complex selection predicates using these BitWeaving scan methods. The BitWeaving technique was initially presented in [64].

Second, Chapter 3 introduces the padded encoding scheme that turns skew in both the data and query predicate distribution into a benefit that can be leveraged to further accelerate the scan primitives in a data processing system. The details about how we encode (compress) a database are present in this chapter. The padded encoding scheme was originally described in [63].

Next, Chapter 4 presents the denormlization technique that aims to answer more complex queries with efficient scan primitives in analytical data processing systems. This chapter also contains an empirical evaluation of the Quickstep engine with and without the WideTable and BitWeaving techniques, using the Star-Schema Benchmark. The initial results of this work were described in [65]. In Chapter 4, we extend this work by adding the space and time analysis of the denormalization method, an evaluation of the performance impact of the techniques used in the WideTable design, and an evaluation of the query processing performance by varying the selectivity of selection predicates.

Finally, Chapter 5 outlines the contributions of the dissertation and discusses future work.

Chapter 2

BitWeaving: fast scans for main memory data processing

A key operation in a main memory DBMS is the full table scan primitive, since ad hoc business intelligence queries frequently use scans over tabular data as base operations. An important goal for a main memory data processing system is to run scans at the speed of the processing units, and exploit all the functionality that is available inside modern processors. For example, a recent proposal for a fast scan [96] packs (dictionary) compressed column values into four 32-bit slots in a 128-bit SIMD word. Unfortunately, this method has two main limitations. First, it does not fully utilize the width of a word. For example, if the compressed value of a particular attribute is encoded by 9 bits, then we must pad each 9-bit value to a 32-bit boundary (or what ever is the boundary for the SIMD instruction), wasting $32 - 9 = 23$ bits every 32 bits. The second limitation is that it imposes extra processing to align tightly packed values to the four 32-bit slots in a 128-bit SIMD word.

In this chapter, we propose a set of techniques, which are collectively called BitWeaving, to aggressively exploit "intra-cycle" parallelism. The insight behind our intra-cycle parallelism paradigm is recognizing that in a single processor clock cycle there is "abundant parallelism" as the circuits in the processor core are simultaneously computing on multiple bits of information, even when working on simple ALU operations. We believe that thinking of how to fully exploit such intra-cycle parallelism is critical in making data processing software run at the speed of the "bare metal", which in this study means the speed of the processor core.

The BitWeaving methods that are proposed in this chapter target intra-cycle parallelism for higher performance. BitWeaving does not rely on the hardware-implemented SIMD capability, and can be implemented with full-word instructions. (Though, it can also leverage SIMD capabilities if they are available.) BitWeaving comes in two flavors:

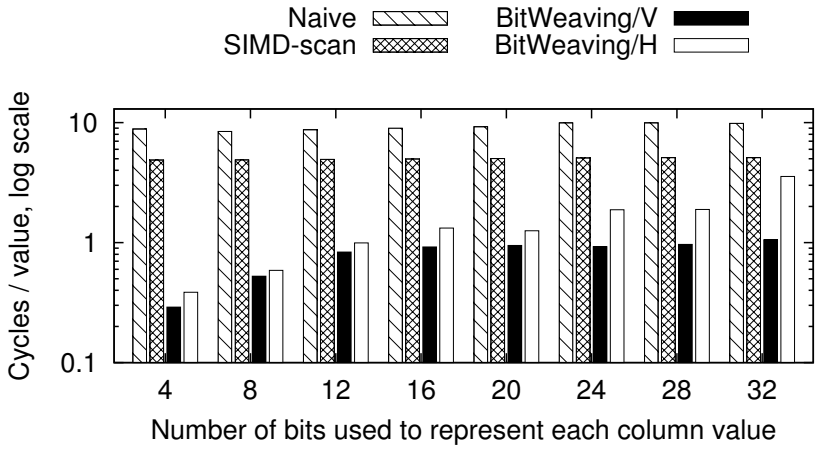


Figure 2.1: Performance Comparison between BitWeaving and other scan methods.

BitWeaving/V and BitWeaving/H, corresponding to two underlying storage formats. Both methods produce as output a result bit vector, with one bit per input tuple that indicates if the input tuple matches the predicate on the column.

The first method, BitWeaving/V, uses a *bit-level* columnar data organization, packed into processor words. It then organizes the words into a layout that results in largely sequential memory address lookups when performing a scan. Predicate evaluation in the scan operation is converted to logical computation on these "words of bits" using the arithmetic framework proposed in this chapter. In this organization, storage space is not wasted padding bits to fit boundaries that are set by the hardware. More importantly, in many cases, an early pruning technique allows the scan computation to be safely terminated, even before all the bits in the underlying data are examined. Thus, predicates can often be computed by only looking at some of most significant bits in each column. This scheme also naturally produces compact result bit vectors that can be used to evaluate the next stage of a complex predicate efficiently.

The second method, BitWeaving/H, uses a bit organization that is a dual of BitWeaving/V. Unlike the BitWeaving/V format, all the bits of a column value are stored together in BitWeaving/H, providing high performance when fetching the entire column value. Unlike previous horizontal bit packing methods, BitWeaving/H staggers the compressed values across processor words in a way that produces compact result bit vectors that are easily reusable when evaluating the next stage of a complex predicate.

Both BitWeaving methods can be used as a native storage organization technique in a column store database, or as an indexing method to index specific column(s) in row stores or column stores.

Figure 2.1 illustrates the performance of a scan operation on a single column, when

varying the width of the column from 1 bit to 32 bits (Section 3.6 has more details about this experiment). This figure shows the SIMD-scan method proposed in [96], and a simple method (labeled Naive) that scans each column in a traditional scan loop and interprets each column value one by one. As can be seen in the figure, both BitWeaving/V and BitWeaving/H outperform the other methods across all the column widths. Both BitWeaving methods achieve higher speedups over other methods when the column representation has fewer number of bits, because this allows more column predicates to be computed in parallel (i.e. the intra-cycle parallelism per input column value is higher). For example, when each column is encoded using 4 bits, the BitWeaving methods are **20X** faster than the SIMD-scan method. Even for columns that are wider than 12 bits, both BitWeaving methods are often more than **4X** faster than the SIMD-scan method. Note that as described in [96], real world data tends to use 8 to 16 bits to encode a column; BitWeaving is one order of magnitude faster than the SIMD-scan method within this range of code widths.

The contribution of this chapter is the presentation of the BitWeaving methods that push our intra-cycle parallelism paradigm to its natural limit – i.e. to the bit level for each column. We also develop an arithmetic framework for predicate evaluation on BitWeaved data, and present results from an actual implementation.

The remainder of this chapter is organized as follows: Section 2.1 contains background information. The BitWeaving methods and the related arithmetic framework is described in Sections 2.2 through 2.4. Section 2.5 contains our experimental results. Related work is covered in Section 2.6, and Section 2.7 contains our concluding remarks.

2.1 Overview

Main memory analytic DBMSs often store data in a compressed form [27, 10, 54, 26]. The techniques presented in this chapter apply to commonly used column compression methods, including null suppression, prefix suppression, frame of reference, and order-preserving dictionary encoding [27, 10, 54, 26]. Such a scheme compresses columns using a fixed-length order-preserving scheme, and converts the native column value to a *code*. In this chapter, we use the term "code" to mean an encoded column value. The data for a column is represented using these codes, and these codes only use as many bits as are needed for the fixed-length encoding.

In these compression methods, all value types, including numeric and string types, are encoded as an unsigned integer code. For example, an order-preserving dictionary can map strings to unsigned integers [54, 12]. A scale scheme can convert floating point numbers to unsigned integers by multiplying by a certain factor [26]. These compression

methods maintain an order-preserving one-to-one mapping between the column values to the codes. As a result, column scans can usually be directly evaluated on the codes.

For predicates involving arithmetic or similarity predicates (e.g. the `LIKE` predicates on strings), scans cannot be performed directly on the encoded codes. These codes have to be decoded, and then are evaluated in a conventional way.

2.1.1 Problem statement

A *column-scalar scan* takes as input a list of n k-bit codes and a predicate with a basic comparison, e.g. $=, \neq, <, >, \leqslant, \geqslant$, `BETWEEN`, on a single column. Constants in the predicate are also in the domain of the compressed codes. The column-scalar scan finds all matching codes that satisfy the predicate, and outputs an n-bit vector, called the *result bit vector*, to indicate the matching codes.

A *processor word* is a data block that can be processed as a unit by the processor. For ease of explanation, we initially assume that a processor word is an Arithmetic Logic Unit (ALU) word, i.e. a 64-bit word for modern CPUs, and in Section 2.2.4 we generalize our method for wider words (e.g. SIMD). The instructions that process the processor word as a unit of computation are called *full-word instructions*. Next, we define when a scan is a *bit-parallel method*.

Definition 2.1. *If w is the width of a processor word, and k is the number of bits that are needed to encode a code in the column C, then a column-scalar scan on column C is a bit-parallel method if it runs in $O(\frac{nk}{w})$ full-word instructions to scan over n codes.*

A bit-parallel method needs to run in $O(\frac{nk}{w})$ instructions to make full use of the "parallelism" that is offered by the bits in the entire width of a processor word. Since processing nk bits with w-bit processor words requires at least $O(\frac{nk}{w})$ instructions, intuitively a method that matches the $O(\frac{nk}{w})$ bound has the potential to run at the speed of the underlying processor hardware.

2.1.2 Framework

The focus of this chapter is on speeding up scan queries on columnar data in main memory data processing engines. Our framework targets the single-table predicates in the WHERE clause of SQL. More specifically, the framework allows conjunctions, disjunctions, or arbitrary boolean combinations of the following basic comparison operators: $=, \neq, <, >, \leqslant, \geqslant$, `BETWEEN`.

For the methods proposed in this chapter, we evaluate the complex predicate by first evaluating basic comparisons on each column, using a *column-scalar scan*. Each column-scalar scan produces a *result bit vector*, with one bit for each input column value that indicates if the corresponding column value was selected to be in the result. Conjunctions and disjunctions are implemented as logical AND and OR operations on these result bit vectors. Once the column-scalar scans are complete, the result bit vector is converted to a list of record numbers, which is then used to retrieve other columns of interest for this query. (See Appendix A.1 for more details.) NULL values and three-valued boolean logic can be implemented in our framework using the techniques proposed in [71], and, in the interest of space, this discussion is omitted here.

We represent the predicates in the SQL WHERE clause as a binary predicate tree. A leaf node encapsulates a basic comparison operation on a single column. The internal nodes represent logical operation, e.g. `AND`, `OR`, `NOT`, on one or two nodes. To evaluate a predicate consisting of arbitrary boolean combinations of basic comparisons, we traverse the predicate tree in depth-first order, performing the column-scalar comparison on each leaf node, and merging result bit vectors at each internal node based on the logical operator that is represented by the internal node. In Section 2.2, we focus on single-column scans, and we discuss complex predicates in Section 2.3.3.

Figure 2.2 illustrates the workflow of evaluating an example query (the query Q6 from the TPC-H benchmark). The query is shown below:

```
SELECT sum(l_extendedprice * l_discount)
FROM lineitem
WHERE l_shipdate BETWEEN Date and Date + 1 year
 and l_discount BETWEEN Discount - 0.01
 and Discount + 0.01 and l_quantity < Quantity
```

As shown in Figure 2.2, the SQL query is converted to an execution structure. Column-scalar scans are first performed on the columns `l_shipdate`, `l_discount`, and `l_quantity`, producing result bit vectors. Then, we perform logical and operations on these result bit vectors. At the end of this operation, the final result bit vector is converted to a list of record IDs of matching tuples, which is then used to retrieve values from the columns `l_extendedprice` and `l_discount` for the matching tuples. If the column values are stored in a encoded (compressed) form, we need to decode these values before calculating the aggregation functions.

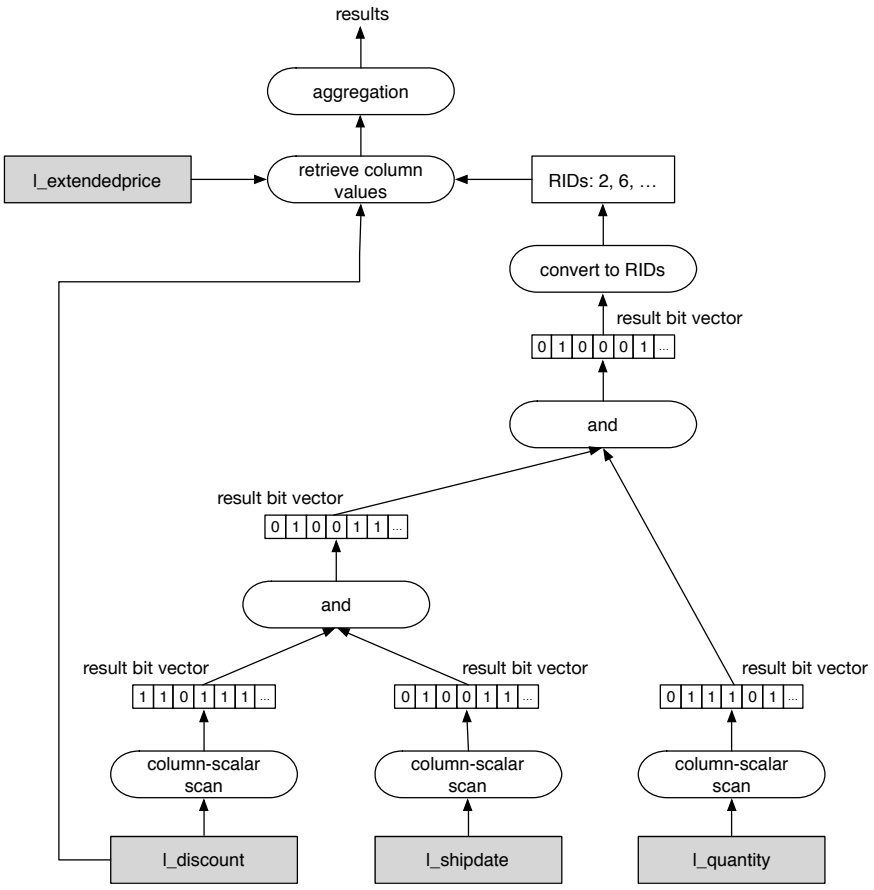


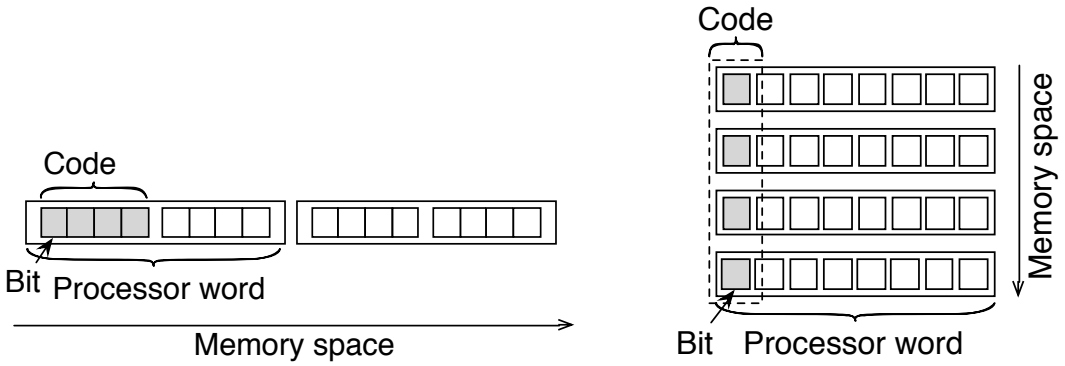
Figure 2.2: Workflow of evaluating the query Q6 in the TPC-H benchmark.

2.2 Bit-parallel methods

In this section, we propose two bit-parallel methods that are designed to fully utilize the entire width of the processor words to reduce the number of instructions that are needed to process data. These two bit-parallel methods are called Horizontal Bit-Parallel (HBP) and Vertical Bit-Parallel (VBP) methods. Each method has a storage format and an associated method to perform a column-scalar scan on that storage method. In Section 2.3, we describe an early pruning technique to improve on the column-scalar scan for both HBP and VBP. Then, in Section 2.4 we describe the BitWeaving method, which combines the bit-parallel methods that are described below with the early pruning technique. BitWeaving comes in two flavors: BitWeaving/H and BitWeaving/V corresponding to the underlying bit-parallel method (i.e. HBP or VBP) that it builds on.

2.2.1 Overview of the two bit-parallel methods

As their names indicate, the two bit-parallel methods, HBP and VBP, organize the column codes horizontally and vertically, respectively. If we think of a code as a tuple consisting of multiple fields (bits), HBP and VBP can be viewed as row-oriented storage and column-oriented storage *at the bit level*, respectively. Figure 2.3 demonstrates the basic idea behind HBP and VBP storage layouts.



(a) Horizontal Bit-Parallel (b) Vertical Bit-Parallel

Figure 2.3: HBP and VBP layouts for a column with 4-bit codes. The shaded boxes represent the bits for the first column value.

Both HBP and VBP only require the following full-word operations, which are common in all modern CPU architectures (including at the SIMD register level in most architectures): logical and ($\wedge$), logical or ($\vee$), exclusive or ($\oplus$), binary addition ($+$), negation ($\neg$), and k-bit left or right shift ($\leftarrow_k$ or $\rightarrow_k$, respectively).

Since the primary access pattern for scan operations is the sequential access pattern, both the CPU cost and the memory access cost are significant components that contribute to the overall execution time for that operation. Consequently, our methods are optimized for both the number of CPU instructions that are needed to process the data, as well as the number of CPU cache lines that are occupied by the underlying (HBP or VBP) data representations.

2.2.1.1 Running example

To illustrate the techniques, we use the following example throughout this section. The data set has 10 tuples, and the column of interest contains the following codes: $\{1 = (001)_2, 5 = (101)_2, 6 = (110)_2, 1 = (001)_2, 6 = (110)_2, 4 = (100)_2, 0 = (000)_2, 7 = (111)_2, 4 = (100)_2, 3 = (011)_2\}$, denoted as c1 – c10 respectively. Each value can be encoded by 3 bits ($k = 3$). For ease of illustration, we assume 8-bit processor words (i.e. $w = 8$).

2.2.2 The horizontal bit-parallel (HBP) method

The HBP method compactly packs codes into processor words, and implements the functionality of hardware-implemented SIMD instructions based on ordinary full-word instructions. The HBP method solves a general problem for hardware-implemented SIMD that the natural bit width of a column often does not match any of the bank widths of the SIMD processor, which leads to an underutilization of the available bit-level parallelism.

We first present the storage layout of HBP in Section 2.2.2.1, and then describe the algorithm to perform a basic scan on a single column over the proposed storage layout in Section 2.2.2.2.

2.2.2.1 Storage layout

In the HBP method, each code is stored in a $(k+1)$-bit section whose leftmost bit is used as a delimiter between adjacent codes (k denote the number of bits needed to encode a code). A method that does not require the extra delimiter bit is feasible, but is much more complicated than the method with delimiter bits, and also requires executing more instructions per code [58]. Thus, our HBP method uses this extra bit for storage.

HBP tightly packs and pads a group of $(k+1)$-bit sections into a processor word. Let w denote the width of a processor word. Then, inside the processor word, $\lfloor \frac{w}{k+1} \rfloor$ sections are concatenated together and padded to the right with 0s up to the word boundary.

In the HBP method, the codes are organized in a storage layout that simplifies the process of producing a result bit vector with one bit per input code (described below in Section 2.2.2.2). The column is divided into fixed-length *segments*, each of which contains $(k+1) \cdot \lfloor \frac{w}{k+1} \rfloor$ codes. Each code represents $k+1$ bits values, with k bits for the actual code and the leading bit set to the delimiter value of 0. Since a processor word fits $\lfloor \frac{w}{k+1} \rfloor$ codes, a segment occupies $k+1$ contiguous processor words in memory space. Inside a segment, the layout of the $(k+1) \cdot \lfloor \frac{w}{k+1} \rfloor$ codes, denoted as $c_1 \sim c_{(k+1)\cdot\lfloor \frac{w}{k+1} \rfloor}$, is shown below. We use v_i to denote the i^{th} processor word in the segment.

$$
\begin{array}{lccccc}
v_1: & c_1 & c_{k+2} & c_{2k+3} & \cdots & c_{k\cdot\lfloor \frac{w}{k+1} \rfloor+1} \\
v_2: & c_2 & c_{k+3} & c_{2k+4} & \cdots & c_{k\cdot\lfloor \frac{w}{k+1} \rfloor+2} \\
\vdots & \vdots & \vdots & \vdots & & \vdots \\
v_k: & c_k & c_{2k+1} & c_{3k+2} & \cdots & c_{(k+1)\cdot\lfloor \frac{w}{k+1} \rfloor-1} \\
v_{k+1}: & c_{k+1} & c_{2k+2} & c_{3k+3} & \cdots & c_{(k+1)\cdot\lfloor \frac{w}{k+1} \rfloor}
\end{array}
$$

Figure 2.4 demonstrates the storage layout for the example column. Since each code in the example column is encoded by 3 bits ($k = 3$), we use $4 = 3 + 1$ bits to store each

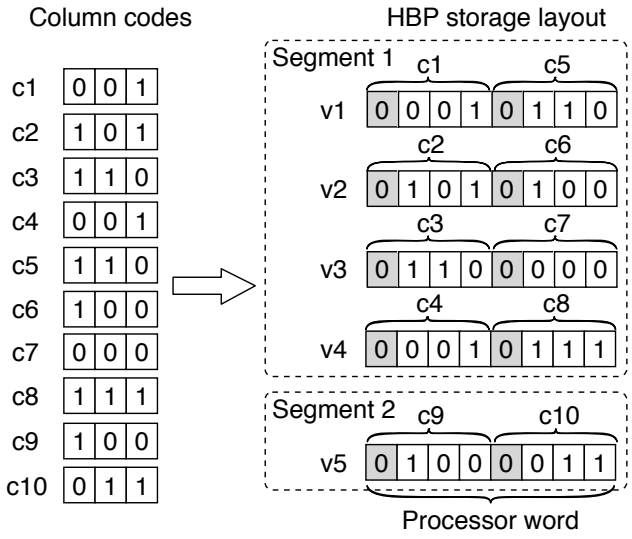


Figure 2.4: Example of the HBP storage layout ($k = 3, w = 8$). Delimiter bits are marked in gray.

code and fit two codes into a 8-bit word ($w = 8$). As shown in the figure, the 10 values are divided into 2 segments. In segment 1, eight codes are packed into four 8-bit words. More specifically, word 1 contains code 1 and 5. Word 2 contains code 2 and 6. Word 3 contains code 3 and 7. Word 4 contains code 4 and 8. Segment 2 is only partially filled, and contains code 9 and code 10 that are packed into word 5.

2.2.2.2 Column-scalar scans

The HBP column-scalar scan compares each code with a constant C, and outputs a bit vector to indicate whether or not the corresponding code satisfies the comparison condition.

In HBP, $\lfloor \frac{w}{k+1} \rfloor$ codes are packed into a processor word. Thus, we first introduce a function $f_\circ(X, C)$ that performs simultaneous comparisons on $\lfloor \frac{w}{k+1} \rfloor$ packed codes in a processor word. The outcome of the function is a vector of $\lfloor \frac{w}{k+1} \rfloor$ results, each of which occupies a $(k + 1)$-bit section. The delimiter (leftmost) bit of each section indicates the comparison results.

Formally, a function $f_\circ(X, C)$ takes as input a comparison operator $\circ$, a comparison constant C, and a processor word X that contains a vector of $\lfloor \frac{w}{k+1} \rfloor$ codes in the form $X = (x_1, x_2, \cdots, x_{\lfloor \frac{w}{k+1} \rfloor})$, and outputs a vector $Z = (z_1, z_2, \cdots, z_{\lfloor \frac{w}{k+1} \rfloor})$, where $z_i = 10^k$ if $x_i \circ C = \texttt{true}$, or $z_i = 0^{k+1}$ if $x_i \circ C = \texttt{false}$. Note that in the notation above for z_i, we use exponentiation to denote bit repetition, e.g. $1^4 0^2 = 111100$, $10^k = 1\underbrace{00\cdots00}_{k}$.

Since the codes are packed into processor words, the ALU instruction set can not be directly used to process these packed codes. In HBP, the functionality of vector processing

is implemented using full-word instructions. Let Y denote a vector of $\lfloor \frac{w}{k+1} \rfloor$ instances of constant C, i.e. $Y = (y_1, y_2, \cdots, y_{\lfloor \frac{w}{k+1} \rfloor})$, where $y_i = C$. Then, the task is to calculate the vector Z in parallel, where each (k+1)-bit section in this vector, $z_i = x_i \circ y_i$; here, $\circ$ is one of comparison operators described as follows. Note that most of these functions are adapted from [58].

INEQUALITY ($\neq$). For the INEQUALITY, observe that $x_i \neq y_i$ iff $x_i \oplus y_i \neq 0^{k+1}$. Thus, we know that $x_i \neq y_i$ iff $(x_i \oplus y_i) + 01^k = 1*^k$ (we use $*$ to represent an arbitrary bit), which is true iff $((x_i \oplus y_i) + 01^k) \wedge 10^k = 10^k$. We know that $(x_i \oplus y_i) + 01^k$ is always less than 2^{k+1}, so overflow is impossible for each $(k+1)$-bit section. As a result, these computation can be done simultaneously on all x_i and y_i within a processor word. It is straightforward to see that $Z = ((X \oplus Y) + 01^k 01^k \cdots 01^k) \wedge 10^k 10^k \cdots 10^k$.

EQUALITY (=). EQUALITY operator is implemented by the complement of the INEQUALITY operator, i.e. $Z = \neg((X \oplus Y) + 01^k 01^k \cdots 01^k) \wedge 10^k 10^k \cdots 10^k$.

LESS THAN ($<$). Since both x_i and y_i are integers, we know that $x_i < y_i$ iff $x_i \leqslant y_i - 1$, which is true iff $2^k \leqslant y_i + 2^k - x_i - 1$. Observe that $2^k - x_i - 1$ is just the k-bit logical complement of x_i, which can be calculated as $x_i \oplus 01^k$. It is then easy to show that $(y_i + (x_i \oplus 01^k)) \wedge 10^k = 10^k$ iff $x_i < y_i$. We also know that $y_i + (x_i \oplus 01^k)$ is always less than 2^{k+1}, so overflow is impossible for each $(k+1)$-bit section. Thus, we have $Z = (Y + (X \oplus 01^k 01^k \cdots 01^k)) \wedge 10^k 10^k \cdots 10^k$ for the comparison operator $<$.

LESS THAN OR EQUAL TO ($\leqslant$). Since $x_i \leqslant y_i$ iff $x_i < y_i + 1$, we have $Z = (Y + (X \oplus 01^k) + 0^k 1) \wedge 10^k 10^k \cdots 10^k$ for the comparison operator $\leqslant$.

Then, GREATER THAN ($>$) and GREATER THAN OR EQUAL TO ($\geqslant$) can be implemented by swapping X and Y for LESS THAN ($<$) and LESS THAN OR EQUAL TO ($\leqslant$) operators, respectively.

Thus, the function $f_\circ(X, C)$ computes the predicates listed above on $\lfloor \frac{w}{k+1} \rfloor$ codes using 3–4 instructions.

Figure 2.5 illustrates an example when applying $f_<(v_i, 5)$ on the words $v_1 \sim v_5$ shown in Figure 2.4. The i^{th} column in the figure demonstrates the steps when calculating $Z = f_<(v_i, 5)$ on the word v_i. The last row represents the results of the function. Each result word contains two comparison results. The value $(1000)_2$ indicates that the corresponding code is less than the constant 5, whereas the value $(0000)_2$ indicates that the corresponding code does not satisfy the comparison condition.

Next, we present the HBP column-scalar scan algorithm based on the function $f_\circ(X, C)$. Algorithm 1 shows the pseudocode for the scan method. The basic idea behind this algorithm is to reorganize the comparison results in an appropriate order, matching the

	$v_1(c_1, c_5)$	$v_2(c_2, c_6)$	$v_3(c_3, c_7)$	$v_4(c_4, c_8)$	$v_5(c_9, c_{10})$
$X =$	$(0001\ \ 0110)_2$	$(0101\ \ 0100)_2$	$(0110\ \ 0000)_2$	$(0001\ \ 0111)_2$	$(0100\ \ 0011)_2$
$Y =$	$(0101\ \ 0101)_2$	$(0101\ \ 0101)_2$	$(0101\ \ 0101)_2$	$(0101\ \ 0101)_2$	$(0101\ \ 0101)_2$
$mask =$	$(0111\ \ 0111)_2$	$(0111\ \ 0111)_2$	$(0111\ \ 0111)_2$	$(0111\ \ 0111)_2$	$(0111\ \ 0111)_2$
$X \oplus mask =$	$(0110\ \ 0001)_2$	$(0010\ \ 0011)_2$	$(0001\ \ 0111)_2$	$(0110\ \ 0000)_2$	$(0011\ \ 0100)_2$
$Y + (X \oplus mask) =$	$(1011\ \ 0110)_2$	$(0111\ \ 1000)_2$	$(0110\ \ 1100)_2$	$(1011\ \ 0101)_2$	$(1001\ \ 1001)_2$
$Z = (Y + (X \oplus mask)) \wedge \neg mask =$	$(1000\ \ 0000)_2$	$(0000\ \ 1000)_2$	$(0000\ \ 1000)_2$	$(1000\ \ 0000)_2$	$(1000\ \ 1000)_2$

Figure 2.5: Evaluating a predicate $c < 5$ on the example column c.

Algorithm 1 HBP column-scalar scan

Input: a comparison operator $\circ$
a comparison constant C
Output: BV_{out}: result bit vector

```
1: for each segment s in column c do
2:   m_s := 0
3:   for i := 1 ... k + 1 do
4:     m_w := f_∘(s.v_i, C)
5:     m_s := m_s ∨ →_{i−1} (m_w)
6:   append m_s to BV_out
7: return BV_out;
```

order of the original codes. As shown in the algorithm, for each segment in the column, we iterate over the $k+1$ words. In the inner loop over the $k+1$ words, we combine the results of $f_\circ(v_1, C) \sim f_\circ(v_{k+1}, C)$ together to obtain the result bit vector on segment s. This procedure is illustrated below:

$$
\begin{array}{rccccccc}
f_\circ(v_1, C): & R(c_1) & 0 & \cdots & 0 & 0 & R(c_{k+2}) & \cdots \\
\rightarrow_1 (f_\circ(v_2, C)): & 0 & R(c_2) & \cdots & 0 & 0 & 0 & \cdots \\
\vdots & \vdots & \vdots & & \vdots & & & \\
\rightarrow_{k-1} (f_\circ(v_k, C)): & 0 & 0 & \cdots & R(c_k) & 0 & 0 & \cdots \\
\rightarrow_k (f_\circ(v_{k+1}, C)): & 0 & 0 & \cdots & 0 & R(c_{k+1}) & 0 & \cdots \\
\sum_\vee: & R(c_1) & R(c_2) & \cdots & R(c_k) & R(c_{k+1}) & R(c_{k+2}) & \cdots
\end{array}
$$

In the tabular representation above, each column represents one bit in the outcome of $f_\circ(v_i, C)$. Let $R(c_i)$ denote the binary result of the comparison on $c_i \circ C$. Since $R(c_i)$ is always placed in the delimiter (leftmost) bit in a $(k+1)$-bit section, the output of $f_\circ(v_i, C)$ is in the form: $R(c_i)0^k R(c_{k+1+i})0^k \cdots$. By right shifting the output of $f_\circ(v_i, C)$, we move the result bits $R(c_i)$ to the appropriate bit positions. The OR ($\vee$) summation over the $k+1$ result words is then in the form of $R(c_1)R(c_2)R(c_3)\cdots$, representing the comparison results on the $\lfloor \frac{w}{k+1} \rfloor$ codes of segment s, in the desired result bit vector format.

For instance, to compute the result bit vector on segment 1 (v_1, v_2, v_3, and v_4) shown in Figure 2.4 and Figure 2.5, we perform $(1000\ 0000)_2 \vee \rightarrow_1 (0000\ 1000)_2 \vee \rightarrow_2 (0000\ 1000)_2 \vee \rightarrow_3 (1000\ 0000)_2 = (1001\ 0110)_2$. The result bit vector $(1001\ 0110)_2$ means that c_1, c_4, c_6, and c_7 satisfy the comparison condition.

Note that the steps above, which are carried out to produce a result bit vector with one bit per input code, are essential when using the result bit vector in a subsequent operation (e.g. the next step of a complex predicate evaluation in which the other attributes in the predicate have different code widths).

The HBP storage layout is designed to make it easy to assemble the result bit vector with one bit per input code. Taking Figure 2.4 as an example again, imagine that we lay out all the codes in sequence, i.e. put c1 and c2 in $v1$, put c3 and c4 in $v2$, and so forth. Now, the result words from the predicate evaluation function $f_\circ(v_i, C)$ on $v1, v2, \cdots$ are $f_\circ(v_1, C) = R(c_1)000R(c_2)000$, $f_\circ(v_2, C) = R(c_3)000R(c_4)000$, $\cdots$. Then, these result words must be converted to a bit vector of the form $R(c_1)R(c_2)R(c_3)R(c_4)\cdots$, by extracting all the delimiter bits $R(c_i)$ and omitting all other bits. Unfortunately, this conversion is relatively expensive compared to the computation of the function $f_\circ(v_i, C)$ (See Appendix A.2 for more details). In contrast, the storage layout used by the HBP method does not need to execute this conversion to produce the result bit vector. In Section 2.5.2, we empirically compare the HBP method with a method that needs this conversion.

2.2.3 The vertical bit-parallel (VBP) method

The Vertical Bit-Parallel (VBP) method is like a bit-level column store, with data being packed at word boundaries. VBP is inspired by the bit-sliced method [71], but as described below, is different in the way it organizes data around word boundaries.

2.2.3.1 Storage layout

In VBP, the column of codes is broken down to fixed-length segments, each of which contains w codes (w is the width of a processor word). The w k-bit codes in a segment are then transposed into k w-bit words, denoted as v_1, v_2, $\cdots$, v_k, such that the j-th bit in v_i equals to the i-th bit in the original code c_j.

Inside a segment, the k words, i.e. v_1, v_2, $\cdots$, v_k, are physically stored in a continuous memory space. The layout of the k words exactly matches the access pattern of column-scalar scans (presented below in Section 2.2.3.2), which leads to a sequential access pattern on these words, making it amenable for hardware prefetching.

Figure 2.6 illustrates the VBP storage layout for the running example shown in Section 2.2.1.1. The ten codes are broken into two segments with eight and two codes, respectively. The two segments are separately transposed into three 8-bit words. The word v_1 in segment 1 holds the most significant (leftmost) bits of the codes c1 $\sim$ c8, the word v_2 holds the middle bits of the codes c1 $\sim$ c8, and the word v_3 holds the least significant (rightmost) bits of the codes c1 $\sim$ c8. In segment 2, only the leftmost two bits of the three words are used, and the remaining bits are filled with zeros.

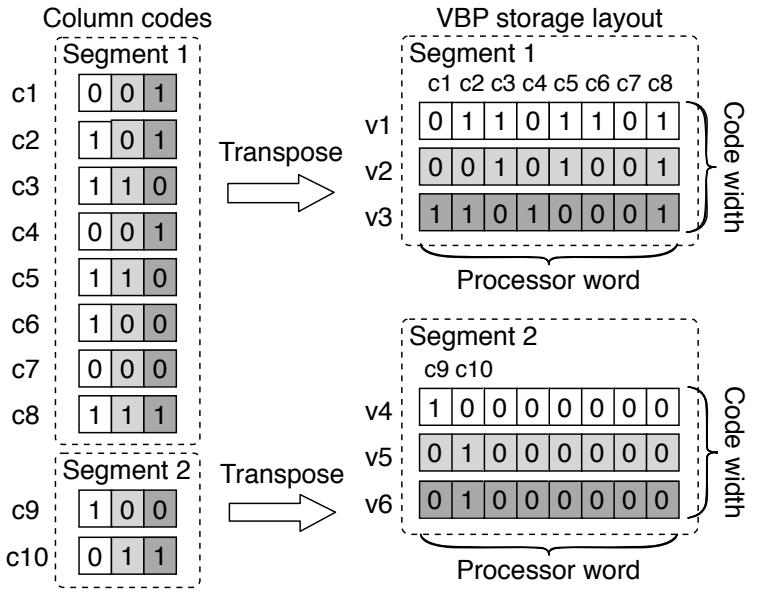


Figure 2.6: Example of the VBP storage layout. The middle bits of codes are marked in light gray, whereas the least significant bits are marked in dark gray.

2.2.3.2 Column-scalar scans

The VBP column-scalar scan evaluates a comparison condition over all the codes in a single column and outputs a bit vector, where each bit indicates whether or not the corresponding code satisfies the comparison condition.

The VBP column-scalar scan follows the natural way to compare two integers in the form of bit strings: we compare each pair of bits at the same position of the two bit strings, starting from the most significant bits to the least significant bits. The VBP method essentially performs this process on a vector of w codes in parallel, inside each segment.

Algorithm 2 shows the pseudocode to evaluate the comparison predicate `BETWEEN` C1 `AND` C2.

At the beginning of the algorithm (Lines 1–5), we create a list of words $C1_1 \sim C1_k$ to represent w instances of C1 in the VBP storage format. If the i-th bit of C1 is 1, $C1_i$ is set to be all 1s, as all the i-th bits of the w instances of C1 are all 1s. Otherwise, $C1_i$ is set to be all 0s. Similarly, we create $C2_1 \sim C2_k$ to represent C2 in the VBP storage format (in Line 6–10).

In the next step, we iterate over all segments of the column, and simultaneously evaluate the range on all the w codes in each segment. The bit vector m_{gt} is used to indicate the codes such that they are greater than the constant C1, i.e. if the i-th bit of m_{gt} is on, then the i-th code in the segment is greater than the constant C1. Likewise, m_{lt} is used to indicate the codes that are less than the constant C2. m_{eq1} and m_{eq2} are used to indicate the codes that are equivalent to the constant C1 and C2, respectively.

Algorithm 2 VBP column-scalar comparison

Input: a predicate $C1 < c < C2$ on column c
Output: BV_{out}: result bit vector

```
 1: for i := 1 ... k do
 2:   if i-th bit in C1 is on then
 3:     C1_i := 1^w
 4:   else
 5:     C1_i := 0^w
 6: for i := 1 ... k do
 7:   if i-th bit in C2 is on then
 8:     C2_i := 1^w
 9:   else
10:     C2_i := 0^w
11: for each segment s in column c do
12:   m_lt, m_gt := 0
13:   m_eq1, m_eq2 := 1^w
14:   for i := 1 ... k do
15:     m_gt := m_gt ∨ (m_eq1 ∧ ¬C1_i ∧ s.v_i)
16:     m_lt := m_lt ∨ (m_eq2 ∧ C2_i ∧ ¬s.v_i)
17:     m_eq1 := m_eq1 ∧ ¬(s.v_i ⊕ C1_i)
18:     m_eq2 := m_eq2 ∧ ¬(s.v_i ⊕ C2_i)
19:   append m_gt ∧ m_lt to BV_out
20: return BV_out;
```

In the inner loop (Line 14–18), we compare the codes with the constants C1 and C2 from the most significant bits to the least significant bits, and update the bit vector m_{gt}, m_{lt}, m_{eq1}, and m_{eq2}, correspondingly. The k words in the segment s are denoted as $s.v_1 \sim s.v_k$. At the i-th bit position, for a code with the i-th bit on, the code must be greater than the constant C1 iff the i-th bit of C1 is off and all bits to the left of this position between the code and C1 are all equal ($m_{eq1} \wedge \neg C1_i \wedge s.v_i$). The corresponding bits in m_{gt} are then updated to be 1s (Line 15). Similarly, m_{lt} is updated if the i-th bit of a code is 0, the i-th bit of C2 is 1, and all the bits to the left of this position are all equal (Line 16). We also update m_{eq1} (m_{eq2}) for the codes that are different from the constant C1(C2) at the i-th bit position (Line 17 & 18).

After the inner loop, we perform a logical AND between the bit vector m_{gt} and m_{lt} to obtain the result bit vector on the segment (Line 19). This bit vector is then appended to the result bit vector.

Algorithm 2 can be easily extended for other comparison conditions. For example, we can modify Line 19 to "append $m_{gt} \wedge m_{lt} \vee m_{eq1} \vee m_{eq2}$ to BV_{out}" to evaluate the condition $C1 \leqslant c \leqslant C2$. For certain comparison conditions, some steps can be eliminated. For instance, Line 15 and 17 can be skipped for a LESS THAN ($<$) comparison, as we do not

need to evaluate m_{gt} and m_{eq1}.

2.2.4 SIMD-based bit-parallel methods

In this section, we describe how our methods can be extended to work with modern vector processors that support larger SIMD word. Today 256-bit SIMD words are common, and Intel's latest Haswell processors now have 512 bit long SIMD words [44].

As per our definition of bit-parallel methods (Definition 2.1), bit-parallel methods can potentially achieve linear speedup with increasing word size w. With wider SIMD registers/word size, our bit-parallel methods can operate simultaneously on more codes. Unlike the previous work using SIMD instructions to implement scan operations [96], bit-parallel methods do not rely on the capability of vector processing of SIMD instructions to achieve parallelism. Instead, we treat SIMD instructions as ordinary non-vector instructions that are simply operating on wider words. Parallelism inside SIMD words is achieved by the horizontal and vertical bit-parallel techniques introduced in Section 2.2.2 and Section 2.2.3.

SIMD-based VBP. The VBP method only uses logical operations over processor words, including logical and ($\wedge$), or ($\vee$), an exclusive or ($\oplus$), and negation ($\neg$). For these logical operations, each bit is calculated independently inside a processor word. Although SIMD logical instructions are designed to work on a vector of independent banks, e.g. 8 bits, 16 bits, 32 bits, or 64 bits, inside a SIMD word, the results of logical operations works even when the codes cross the boundaries of SIMD banks. As a result, it is straightforward to implement VBP using SIMD instructions.

SIMD-based HBP. The HBP method relies on arithmetic and shift operations in addition to logical operations. Arithmetic and shift operations on a vector of independent banks inside a SIMD word are not equivalent to that across the whole SIMD word. For instance, the sum of two 256-bit unsigned integers is different from the concatenation of four sums of two 64-bit unsigned integers, because carry overs don't propagate across the boundary of SIMD banks. As a result, in the storage layout of SIMD-based HBP, the codes are padded to the largest bank unit supported by the SIMD instruction set. Figure 2.7 demonstrates an example layout that pads twelve 18-bit codes to four 64-bit banks inside a 256-bit SIMD word. It is known that this solution underutilizes the full-width of a SIMD word, e.g. a 256-bit SIMD word should contain fourteen, rather than twelve, 18-bit codes. If the future SIMD instruction set supports the new instructions across the whole SIMD word, more codes can be made to fit into a SIMD word, leading to speedups for the SIMD-based HBP.

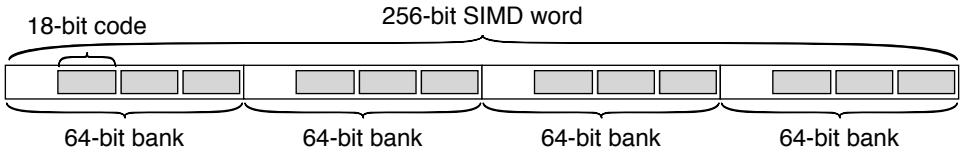


Figure 2.7: Storage layout of SIMD-based HBP.

2.3 Early pruning

The early pruning technique aims to avoid accesses on unnecessary data at the bit level. This technique is orthogonal to the two bit-parallel methods described in Section 2.2, and hence can be applied to both the HBP and the VBP methods. However, as the early pruning technique is more naturally described within the context of VBP, we first describe this technique as applied to VBP. Then, in Section 2.4.2 we discuss how to apply this technique to HBP.

2.3.1 Basic idea behind early pruning

It is often not necessary to access all the bits of a code to compute the final result. For instance, to compare the code $(110\underline{1}0101)_2$ to a constant $(110\underline{0}1010)_2$, we compare the pair of bits at the same position, starting from the most significant bit to the least significant bit, until we find two bits that are different. At the 4th position (underlined above), the two bits are different, and thus we know that the code is greater than the constant. We can now ignore the remaining bits.

	Constant	VBP words	m_{lt}
1st bit	0	01101101	00000000
2nd bit	1	00101001	10010010
3rd bit	1	11010001	-

Figure 2.8: Evaluating a predicate $c < 3$ with the early pruning technique.

It is easy to apply the early pruning technique on VBP, which performs comparisons on a vector of w codes in parallel. Figure 2.8 illustrates the process of evaluating the eight codes in segment 1 of the example column c with a comparison condition $c < 3$. The constant 3 is represented in the binary form $(011)_2$ as shown in the second column in the figure. The first eight codes ($1 = (001)_2$, $5 = (101)_2$, $6 = (110)_2$, $1 = (001)_2$, $6 = (110)_2$, $4 = (100)_2$, $0 = (000)_2$, $7 = (111)_2$) of column c are stored in three 8-bit VBP words, as shown in the third column in the figure.

By comparing the first bit of the constant (0) with the first bits of the eight codes (01101101), we notice that no code is guaranteed to be less than the constant at this point. Thus, the bit vector m_{lt} is all 0s to reflect this situation. Next, we expand the comparison to the second bit between the constant and the codes. Now, we know that the 1st, 4th, and 7th codes are smaller than the constant because their first two bits are less than the first two bits of the constant (01). We also know that the 2nd, 3rd, 5th, 6th, and 8th codes are greater than the constant, as their first two bits are greater than the first two bits of the constant (01). At this point, all the codes have a definite answer w.r.t. this predicate, and we can terminate the VBP column-scalar scan on this segment. The bit vector m_{lt} is updated to be 10010010, and it is also the final result bit vector.

2.3.2 Estimating the early pruning probability

We first introduce the *fill factor* f of a segment, defined as the number of codes that are present over the maximum number of codes in the segment, i.e. the width of processor word w. For instance, the fill factor of the segment 1 in Figure 2.6 is $8/8 = 100\%$, whereas the fill factor of the segment 2 is $2/8 = 25\%$. According to this definition, a segment contains wf codes.

The early pruning probability $P(b)$ is defined as the probability that the wf codes in a segment are all different from the constant in the most significant b bits, i.e. it is the probability that we can terminate the computation at the bit position b.

We analyze the early pruning probability $P(b)$ on a segment containing wf k-bit codes. We assume that a code and the constant have the same value at a certain bit position with a probability of $1/2$. Thus, the probability that all of the leading b bits between a code and the constant are identical is given by $(\frac{1}{2})^b$. Since a segment contains wf codes, the probability that these codes are all different from the constant in the leading b bits, i.e. the early pruning probability $P(b)$, is:

$$P(b) = (1 - (\frac{1}{2})^b)^{w \cdot f}$$

Figure 2.9 plots the early pruning probability $P(b)$ with a 64-bit processor word ($w = 64$) by varying the bit position b. We first look at the curve with a 100% fill factor. The early pruning probability increases as the bit position number increases. At the bit position 12, the early pruning probability is already very close to 100%, which indicates that in many cases we can terminate the scan after looking at the first 12 bits. If a code is a 32-bit integer, VBP with early pruning potentially only uses 12/32 of the memory bandwidth and the processing time that is needed by the base VBP method (without early pruning).

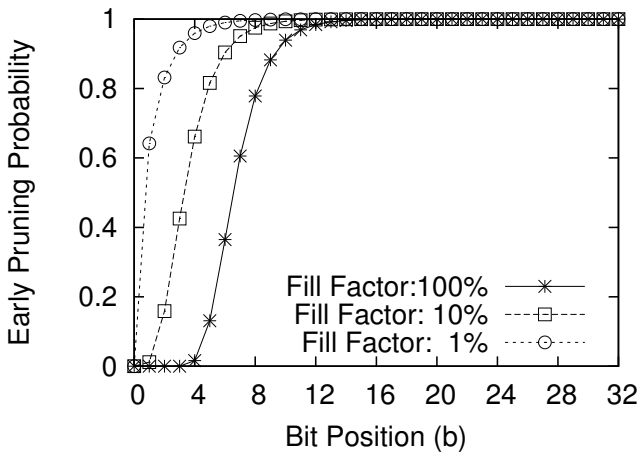


Figure 2.9: Early Pruning Probability P(b).

In Figure 2.9, at the lower fill factors, segments are often "cut-off" early. For example, for segments with fill factor 10%, we can prune the computation at bit position 8 in most (i.e. 97.5%) cases. This cut-off mechanism allows for efficient evaluation of conjunction/disjunction predicates in BitWeaving, as we will see next in Section 2.3.3.

2.3.3 Filter bit vectors on complex predicates

The early pruning technique can also be used when evaluating predicate clauses on multiple columns. Predicate evaluation on a single column can be pruned as outlined above in Section 2.3.1. But, early pruning can also be used when evaluating a series of predicate clauses with the result vector from the first clause being used to "initialize" the pruning bit vector for the second clause.

The result bit vector that is produced from a previous step is called the *filter bit vector* of the current column-scalar scan. This filter bit vector is used to filter out the tuples that do not match the predicate clauses that were examined in the previous steps, leading to a lower fill factor on the current column. Thus, the filter bit vector further reduces the computation on the current column-scalar scan (note that at the lower fill factors, predicate evaluation are often "cut-off" early, as shown in Figure 2.9).

As an example, consider the complex predicate: `R.a < 10 AND R.b > 5 AND R.c < 20 OR R.d = 3`. Figure 2.10 illustrates the predicate tree for this expression. First, we evaluate the predicate clause on column `R.a`, using early pruning. This evaluation produces a result bit vector. Next, we start evaluating the predicate clause on the column `R.b`, using early pruning. However, in this step we use the result bit vector produced from the previous step to seed the early pruning. Thus, tuples that did not match the predicate clause

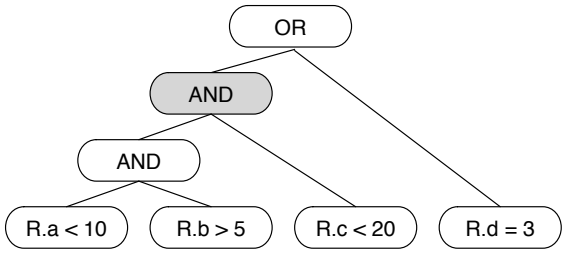


Figure 2.10: An example predicate tree for the expression `R.a < 10 AND R.b > 5 AND R.c < 20 OR R.d = 3`.

`R.a` < 10 become candidates for early pruning when evaluating the predicate clause on `R.b`, regardless of the value of their `b` column. As a result, the predicate evaluation on column `b` is often "cut-off" even earlier. Similarly, the result bit vector produced at the end of evaluating the AND node (the white AND node in the figure) is fed into the scan on column `R.c`. Finally, since the root node is an OR node, the complement of the result bit vector on the AND node (the gray one) is fed into the final scan on column `R.d`.

2.4 BitWeaving

In this section, we combine the techniques proposed above, and extend them, into the overall method called BitWeaving. BitWeaving comes in two flavors: BitWeaving/H and BitWeaving/V corresponding to the underlying bit-parallel storage format (i.e. HBP or VBP described in Section 2.2) that it builds on. As described below, BitWeaving/V also employs an adapted form of the early pruning technique described in Section 2.3.

We note that the BitWeaving methods can be used as a base storage organization format in column-oriented data stores, and/or as indices to speedup the scans over some attributes. For ease of presentation, below we assume that BitWeaving is used as a storage format. It is straightforward to employ the BitWeaving method as indices.

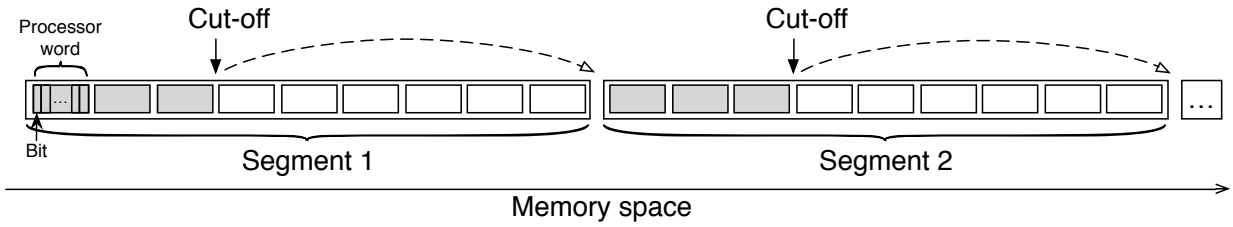
2.4.1 BitWeaving/V

BitWeaving/V is a method that applies the early pruning technique on VBP. BitWeaving/V has three key features: 1) The early pruning technique skips over pruned column data, thereby reducing the total number of bits that are accessed in a column-scalar scan operation; 2) The storage format is not a pure VBP format, but a *weaving* of the VBP format with horizontal packing into *bit groups* to further exploit the benefits of early pruning, by making access to the underlying bits more sequential (and hence more amenable for hardware

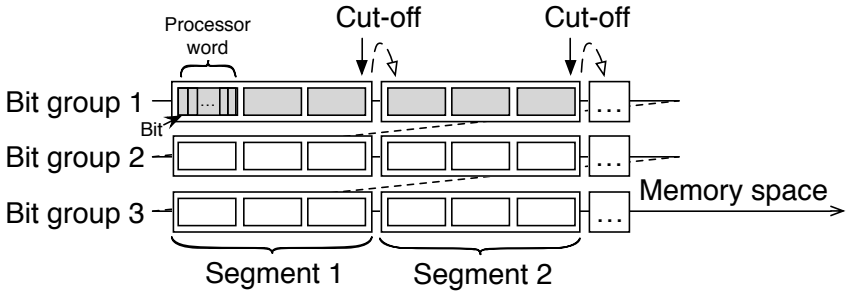
prefetching); 3) It can be implemented with SIMD instructions allowing it to make full use of the entire width of the (wider) SIMD words in modern processors.

2.4.1.1 Storage layout

In this section, we describe how the VBP storage layout is adapted in BitWeaving/V to further exploit the benefits of the early pruning technique. In addition to the core VBP technique of vertical partitioning the codes at the bit level, in BitWeaving/V the codes are also partitioned in a horizontal fashion to provide better CPU cache performance when using the early pruning technique. This combination of vertical and horizontal partitioning is the reason why the proposed solution is called BitWeaving.



(a) VBP



(b) BitWeaving/V

Figure 2.11: Early pruning on VBP and BitWeaving/V.

The VBP storage format potentially wastes memory bandwidth if we apply the early pruning technique on the base VBP storage format. (See Section 2.2.3 for details.) Figure 2.11a illustrates a scan on a column stored in the VBP format. Suppose that with the early pruning technique (described in Section 2.3.1), the outcome of comparisons on all the codes in a segment is determined after accessing the first three words. Thus, the 4th to the 9th words in segment 1 can be skipped, and the processing can move to the next segment (as shown by the dashed arrow). Suppose that a CPU cache line contains 8 words. Thus, the six words that are skipped occupy the same CPU cache line as the first three words. Skipping over the content that has already been loaded into the CPU cache results

in wasted memory bandwidth, which is often a critical resource in main memory data processing environments.

We solve this problem by dividing the k words in a segment into fixed sized *bit groups*. Let B denote the size of each bit group. The words in the same bit group are physically stored in continuous space. Figure 2.11b illustrates how the storage layout with the bit grouping reduces the amount of data that is loaded into the CPU cache. As shown in this figure, the nine words in each segment are divided into three bit groups, each containing three words per segment. Suppose that with the early pruning technique, the outcome of the comparisons on all the codes in a segment is determined after accessing the first three words. In this case, we only need to access the first three words of each segment, which are all laid out continuously and compactly in the first bit group. Consequently, the bit grouping technique uses memory bandwidth more judiciously, and results in a more sequential access pattern.

In the example above, if the early pruning triggers at two bits (instead of three bits), then we still save on memory accesses over a method that does not use bit groups. However, we will likely waste memory bandwidth bringing in data for the third bit. Picking an optimal bit group size is an interesting direction for future work. In Section 2.5.5, we empirically demonstrate the impact of the bit group size.

2.4.1.2 Column-scalar scans

We apply the early pruning technique on the VBP column-scalar scan, in two ways. First, when evaluating a comparison condition on a vector of codes, we skip over the least significant bits as soon as the outcome of the scan is fully determined. Second, a filter bit vector is fed into the scan to further speedup comparisons. This bit vector is used to filter unmatched tuples even before the scan starts. This technique reduces the number of available codes in each segment, and thus speedups the scan (recall that early pruning technique often runs faster on segments with a lower fill factor as shown in Section 2.3.2).

The pseudocode for a VBP column-scalar scan with early pruning technique is shown in Algorithm 3. The algorithm is based on the VBP column-scalar scan shown in Algorithm 2. The modified lines are marked with $\lhd$ and $\square$ at the end of lines.

The first modification over the VBP scan method is to skip over the least significant bits once the outcome of the scan is fully determined (marked with $\square$ at the end of lines). In the BitWeaving/V storage layout, k words representing a segment are divided into fixed-size bit groups. Each bit group contains B words in the segment. Predicate evaluation is also broken into a group of small loops to adapt to the design of bit groups. Before working on each bit group, we check the values of the bit masks m_{eq1} and m_{eq2}. If both bit masks

Algorithm 3 BitWeaving/V column-scalar scan

Input: a predicate $C1 < c < C2$ on column c
BV_{in}: filter bit vector
Output: BV_{out}: result bit vector

```
 1: initialize C1 and C2 (same as Lines 1-10 in Algorithm 2)
 2: for each segment s in column c do
 3:   m_lt, m_gt := 0
 4:   m_eq1, m_eq2 := BV_in.s                                   ◁
 5:   for g := 1 ... ⌊k/B⌋ do                                   □
 6:     if m_eq1 == 0 and m_eq2 == 0 then                       □
 7:       break                                                 □
 8:     for i := gB + 1 ... min(gB + B, k) do                   □
 9:       m_gt := m_gt ∨ (m_eq1 ∧ ¬C1_i ∧ s.w_i)
10:       m_lt := m_lt ∨ (m_eq2 ∧ C2_i ∧ ¬s.w_i)
11:       m_eq1 := m_eq1 ∧ ¬(s.w_i ⊕ C1_i)
12:       m_eq2 := m_eq2 ∧ ¬(s.w_i ⊕ C2_i)
13:   append m_gt ∧ m_lt to BV_out
14: return BV_out;
```

are all 0s, then the leading bits between the codes and the constant are all different. Thus, the outcome of the scan on the segment is fully determined. As a result, we terminate the evaluation on this segment, and move to the next one.

We check the cut-off condition (in Line 6) in one of every B iterations of processing the k words of a segment. The purpose of this design is to reduce the cost of checking the condition as well as the cost of CPU branch mispredictions that this step triggers. If the cut-off probability at a bit position is neither close to 0% nor 100%, it is difficult for the CPU to predict the branch. Such branch misprediction can significantly slows down the overall execution. With the early pruning technique, checking the cut-off condition in one of every B iterations reduces the number of checks at the positions where the cut-off probability is in the middle range. We have observed that without this attention to branch prediction in the algorithm, the scans generally run slower by up to 40%.

The second modification is to feed a filter bit vector into the column-scalar comparisons. In a filter bit vector, the bits associated with the filtered codes are turned off. Filter bit vectors are typically the result bit vectors on other predicates in a complex WHERE clause (see Section 2.3.3 for more details).

To implement this feature, the bit masks m_{eq1} and m_{eq2} are initialized to the corresponding segment in the filter bit vector (marked with $\triangleleft$ at the end of the line). During the evaluation on a segment, the bit masks m_{eq1} and m_{eq2} are updated by $m_{eq1} := m_{eq1} \wedge \neg(s.w_i \oplus C1_i)$ and $m_{eq2} := m_{eq2} \wedge \neg(s.w_i \oplus C2_i)$, respectively. Thus, the filtered codes remain 0s in m_{eq1} and m_{eq2} during the evaluation. Once the bits associated with

the unfiltered codes are all updated to 0s, we terminate the comparisons on this segment following the early pruning technique. The filter bit vector potentially speedups the cut-off process.

2.4.2 BitWeaving/H

It is also feasible to apply early pruning technique on data stored in the HBP format. The key difference is that we store each bit group in the HBP storage layout (described in Section 2.2.2). For a column-scalar scan, we evaluate the comparison condition on bit groups starting from the one containing the most significant bits. In addition to the result bit vector on the input comparison condition, we also need to compute a bit vector for the inequality condition in order to detect if the outcome of the scan is fully determined. Once the outcome is fully determined, we skip the remaining bit groups (using early pruning).

However, the effect of early pruning technique on HBP is offset by the high overhead of computing the additional bit vector, and has an overall negative impact on performance (see Section 2.5.3). Therefore, the BitWeaving/H method is simply the HBP method.

2.4.3 Comparison between BitWeaving/H and BitWeaving/V

In this section, we compare the two BitWeaving methods, in terms of performance, applicability, as well as ease of implementation. The summary of this comparison is shown in Table 2.1.

	BitWeaving/H	BitWeaving/V
Scan Complexity	$O(\lfloor \frac{n(k+1)}{w} \rfloor)$	$O(\frac{nk}{w})$
SIMD Implementation	Limited	Good
Early Pruning	No	Yes
Lookup Performance	Good	Poor

Table 2.1: Comparing BitWeaving/H and BitWeaving/V.

Scan Complexity. BitWeaving/H uses $k+1$ bits of processor word to store a k-bit code, while BitWeaving/V requires only k bits. As both methods simultaneously process multiple codes, the CPU cost of BitWeaving/H and BitWeaving/V are $O(\lfloor \frac{n(k+1)}{w} \rfloor)$ and $O(\frac{nk}{w})$, respectively; i.e., both are bit-parallel methods as per Definition 2.1. Both BitWeaving methods are generally competitive to other methods. However, in the extreme cases, BitWeaving/V could be close to 2X faster than BitWeaving/H due to the overhead of the delimiter bits (in BitWeaving/H). For instance, BitWeaving/H fits only one 32-bit code

(with an addition delimiter bit) in a 64-bit process word, whereas BitWeaving/V fits two codes.

SIMD Implementation. The implementation of BitWeaving/H method relies on arithmetic and shift operations, which is generally not supported on an entire SIMD word today. Thus, BitWeaving/H has to pad codes to the width of banks in the SIMD registers, rather than the SIMD word width. This leads to underutilization of the full width of the SIMD registers. In contrast, BitWeaving/V method achieves the full parallelism that is offered by SIMD instructions.

Early Pruning. Applying early pruning technique on HBP requires extra processing that hurts the performance of HBP. As a result, BitWeaving/H does not employ the early pruning technique. In contrast, in BitWeaving/V, the early pruning technique works naturally with the underlying VBP-like format with no extra cost, and usually improves the scan performance.

Lookup Performance. With the BitWeaving/H layout, it is easy to fetch a code as all the bits of the code are stored contiguously. In contrast, for BitWeaving/V, all the bits of a code are spread across various bit groups, distributed over different words. Consequently, looking up a code potentially incurs many CPU cache misses, and can thus hurts performance.

To summarize, in general, both BitWeaving/H and BitWeaving/V are competitive methods. BitWeaving/V outperforms BitWeaving/H for scan performance whereas BitWeaving/H achieves better lookup performance. Empirical evaluation comparing these two methods is presented in the next section.

2.5 Evaluation

2.5.1 Experimental setups

We ran our experiments on a machine with dual 2.67GHz Intel Xeon 6-core CPUs, and 24GB of DDR3 main memory. Each processor has 12MB of L3 cache shared by all the cores on that processor. The processors support a 64-bit ALU instruction set as well as a 128-bit Intel SIMD instruction set. The operating system is Linux 2.6.9.

In the evaluation below, we compare BitWeaving to the SIMD-scan method proposed in [96], the Bit-sliced method [71], and a method based on Blink [47]. Collectively, these three methods represent the current state-of-the-art main memory scan methods.

To serve as a yardstick, we also include comparison with a naive method that simply extracts, loads, and then evaluates each code with the comparison condition in series,

without exploiting any word-level parallelism. In the graphs below the tag *Naive* refers to this simple scan method.

Below, the tag *SIMD-scan* refers to the technique in [96] that uses SIMD instructions to align multiple tightly packed codes to SIMD banks in parallel, and then simultaneously processes multiple codes using SIMD instructions.

Below, the tag *Bit-sliced* refers to the traditional bit-sliced method proposed in [71]. This method was originally proposed to index tables with low number of distinct values; it shares similarities to the VBP method, but does not explore the storage layout and early pruning technique. Surprisingly, previous recent work on main memory scans have largely ignored the bit-sliced method.

In the graphs below, we use the tag *BL* (Blink-Like) to represent the method that adapts the Blink method [47] for column stores (since we focus on column stores for this evaluation). Thus, the tag *BL* refers to tightly (horizontally) packed columns with a simple linear layout, and without the extra bit that is used by HBP (see Section 2.2.2). The *BL* method differs from the BitWeaving/H method as it does not have the extra bit, and it lays out the codes in order (w.r.t. the discussion in the last paragraph in Section 2.2.2, the layout of the codes in *BL* is c1 and c2 in $v1$, c3 and c4 in $v2$, and so on in Figure 2.4).

Below, the tags BitWeaving/H (or BW/H) and BitWeaving/V (or BW/V) refer to the methods proposed in this chapter. The size of the bit group is 4 for all experiments. The effect of the other bit group sizes on scan performance is shown in Section 2.5.5.

We implemented each method in C++, and compiled the code using g++ 3.4.6 with optimization flags (O3).

In all the results below, we ran experiments using a single process with a single thread. We have also experimented using multiple threads working on independent data partitions. Since the results are similar to that of a single thread (all the methods parallelize well assuming that each thread works on a separate partition), in the interest of space, we omit these results.

In the experiment below, we created a table R with a single column and one billion uniformly distributed integer values in this column. The domain of the values are $[0, 2^d)$, where d is the width of the column that is varied in the experiments. The query (Q1), shown below, is used to evaluate a column-scalar scan with a simple LESS THAN predicate. The performance on other predicates is similar to that on the LESS THAN predicate. (See Section 2.5.4 for more details.) The constants in the WHERE clause are used to control the selectivity. By default, the selectivity on each predicate is set to 10%, i.e. 10% of the input tuples match the predicate. Note, we also evaluate the impact of different selectivity, but by default use a value of 10%.

```
Q1:  SELECT COUNT(*) FROM R WHERE R.a < C1
```

2.5.2 BitWeaving v/s the other methods

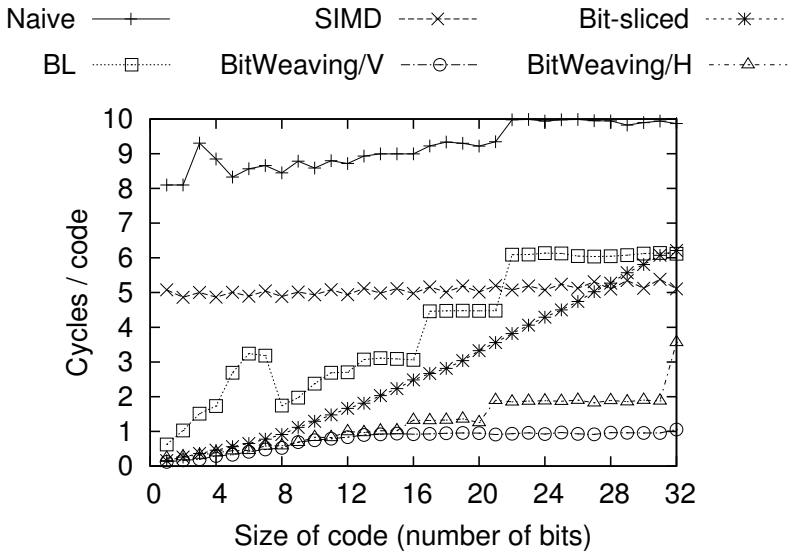
In the evaluation below, we first compare BitWeaving to the Naive, the SIMD-scan [96], the Bit-sliced [71], and the BL methods. Figure 2.12a, Figure 2.12b, and Figure 2.12c illustrate the number of cycles, cache misses, and CPU instructions for the six methods for Q1 respectively, when varying the width of the column code from 1 bit to 32 bits. The total number of cycles for the query is measured by using the RDTSC instruction. We divide this total number of cycles by the number of codes to compute the cycles per code, which is shown in Figure 2.12a.

As can be observed in Figure 2.12a, not surprisingly, the Naive method is the slowest. The Naive method shifts and applies a mask to extract and align each packed code to the processor word. Since each code is much smaller than a processor word (64-bit), it burns many more instructions than the other methods (see Figure 2.12c) on every word of data that is fetched from the underlying memory subsystem (with L1/L2/L3 caches buffering data fetched from main memory). Even when most of the data is served from the L1 cache, its CPU cost dominates the overall query execution time.

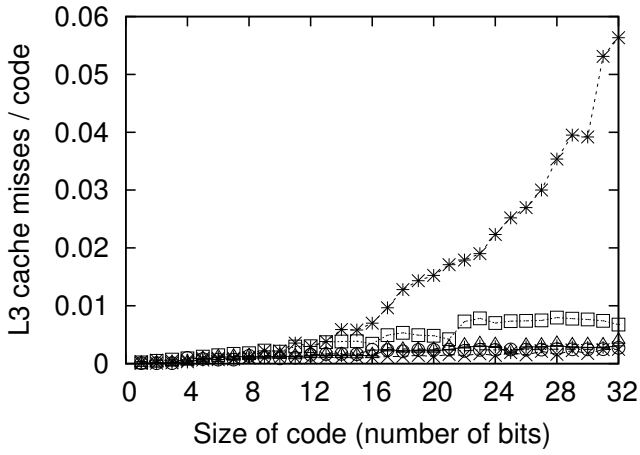
The SIMD-scan achieves 50%–75% performance improvement over the Naive method (see Figure 2.12a), but it is still worse compared to the other methods. Even though a SIMD instruction can process four 32-bit banks in parallel, the number of instructions drops by only 2.2-2.6X (over the Naive method), because it imposes extra instructions to align packed codes into the four banks before any computation can be run on that data. Furthermore, we observe that with SIMD instructions, the CPI (Cycles Per Instructions) increases from 0.37 to 0.56 (see Figure 2.12c), which means that a single SIMD instructions takes more cycles to executed than a ordinary ALU instruction. This effect further dampens the benefit of this SIMD implementation.

As can be seen in Figure 2.12a, the Bit-sliced and the BL methods shows a near linear increase in run time as the code width increases. Surprisingly, both these methods are almost uniformly faster than the SIMD-scan method. However, the storage layout of the Bit-sliced method occupies many CPU cache lines for wider codes. As a result, as can be seen in Figure 2.12b, the number of L3 cache misses quickly increases and hinders overall performance.

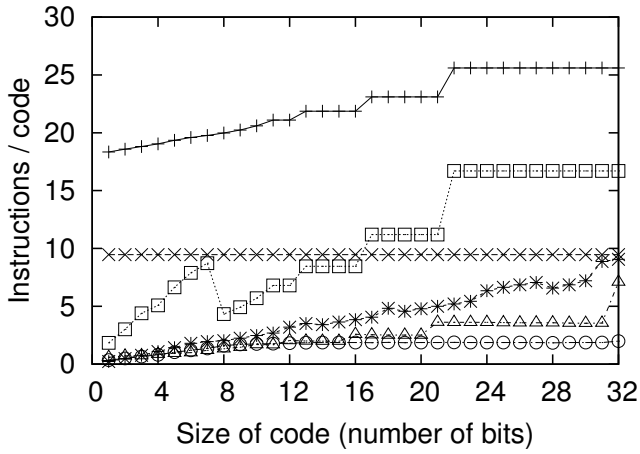
In this experiment, the BitWeaving methods *outperform all the other methods across all the code widths* (see Figure 2.12a). Unlike the Naive and the SIMD-scan methods, they do not need to move data to appropriate positions before the predicate evaluation computation.



(a) Query Execution Time



(b) Memory Performance



(c) CPU Performance

Figure 2.12: Performance on query Q1.

In addition, as shown in Figure 2.12b and 2.12c, the BitWeaving methods are optimized for both cache misses and instructions due to their storage layouts and scan algorithms. Finally, with the early pruning technique, the execution time of BitWeaving/V (see Figure 2.12a) does not increase for codes that are wider than 12 bits. As can be seen in Figure 2.12a, for codes wider than 12 bits, both BitWeaving methods are often more than 3X faster than the SIMD-scan, the Bit-sliced and the BL methods.

2.5.3 Individual BitWeaving components

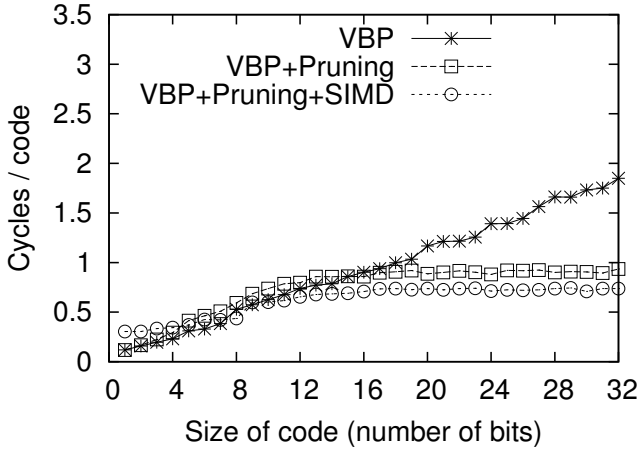
In this experiment, we compare the effect of the various techniques (VBP v/s HBP, early pruning, and SIMD optimizations) that have been proposed in this chapter. Figure 2.13a and 2.13b plot the performance of these techniques for VBP and HBP for query Q1, respectively.

First, we compare the scan performance of the HBP and the VBP methods for query Q1. From the results shown in Figure 2.13a and 2.13b, we observe that at certain points, VBP is up to 2X faster than HBP. For example, VBP is 2X faster than HBP for 32-bit codes, because HBP has to pad 32-bit code to a entire 64-bit word to fit both the code and the delimiter bit. In spite of this, HBP and VBP generally show a similar performance trend as the code width increases. This empirical results matches our analysis that both methods satisfy the cost-bound for bit-parallel methods.

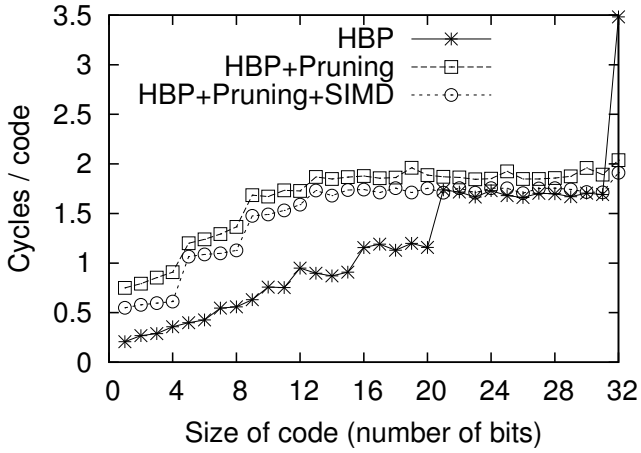
Next, we examine the effects of the early pruning technique on both the VBP and the HBP methods. As can be seen in Figure 2.13a, for wider codes, the early pruning technique quickly reduces the query execution time for VBP, and beyond 12 bits, the query execution time with early pruning is nearly constant. Essentially, as described in Section 2.3.2, for wider codes early pruning has a high chance of terminating after examining the first few bits.

In contrast, as can be seen in Figure 2.13b, the effect of the early pruning technique on the HBP method is offset by the high overhead of computing the additional masks (see Section 2.4.2). Consequently, the HBP method (which is the same as BitWeaving/H, as discussed in Section 2.4.2) is uniformly faster than "HBP + Pruning".

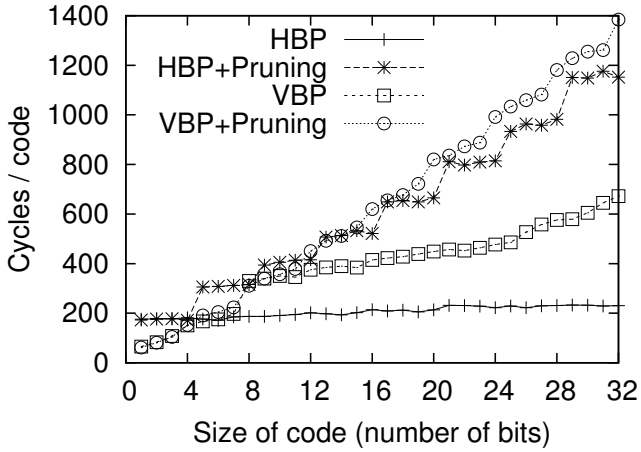
Applying SIMD parallelism achieves marginal speedups for both the HBP and the VBP methods (see Figures 2.13a and 2.13b). Ideally, the implementation with a 128-bit SIMD word should be 2X faster than that with 64-bit ALU word. However, by measuring the number of instructions, we observed that the SIMD implementation reduces the number of instructions by 40%, but also increase the CPI by 1.5X. Consequently, the net effect is that the SIMD implementation is only 20% and 10% faster than the ALU implementation, for VBP and HBP respectively.



(a) VBP-related methods: Execution Time



(b) HBP-related methods: Execution Time



(c) Lookup Performance

Figure 2.13: Performance comparison between the HBP and the VBP related methods on query Q1.

Next, we evaluate the performance of a lookup operation. A lookup operation is important to produce the attributes in the projection list after the predicates in the WHERE clause have been applied. In this experiment, we randomly pick 10 million positions in the column, and measure the average number of cycles that are needed to fetch (and assemble) a code at each position. The results for this experiment are shown in Figure 2.13c.

As can be seen in Figure 2.13c, amongst the four methods, the lookup performance of the HBP method is the best, and its performance is stable across the code widths. The reason for this behavior is because all the bits of a code in the HBP method are located together. For the VBP method, all the bits of a code are stored in continuous space, and thus it is relatively fast to access all the bits and assemble the code. For the methods with the early pruning technique, the bits of a code are distributed into various bit groups. Assembling a code requires access to data across multiple bit groups at different locations, which incurs many CPU cache misses, and thus significantly hurts the lookup performance.

2.5.4 Varying Predicate Types

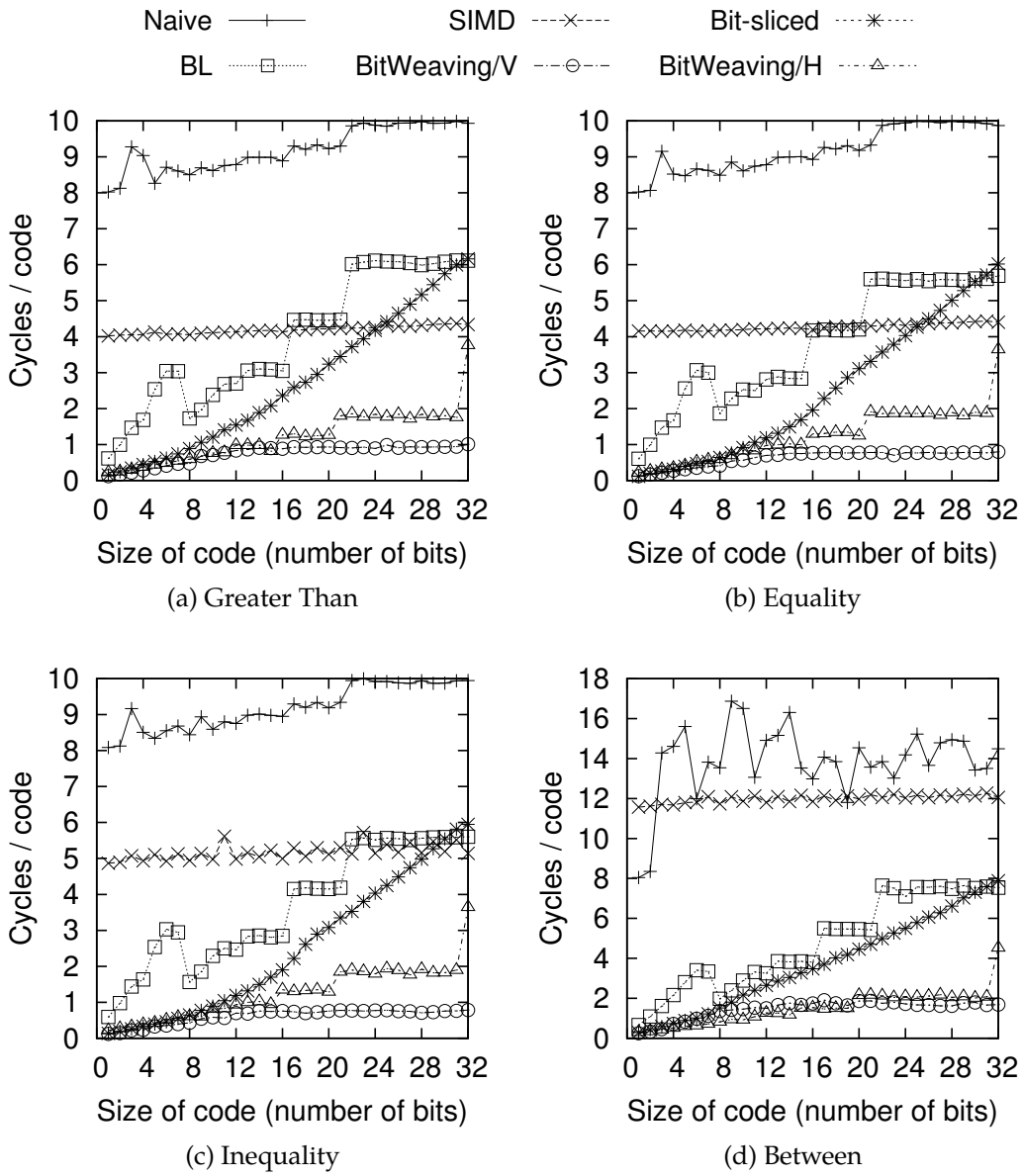
We reran the experiments in Section 2.5.2 with different predicates in the WHERE clause of Q1, including GREATER THAN, EQUALITY, INEQUALITY, and BEWTEEN. Figure 2.14 illustrates the number of cycles for the six methods for Q1 with various predicates, when varying the width of the column code from 1 bit to 32 bits.

As can be seen in Figure 2.14a, Figure 2.14b, and Figure 2.14c, the performance of the six methods with the Greater Than, Equality, and Inequality predicates are nearly identical to those with Less Than predicate shown in Section 2.5.2.

For the BETWEEN predicate (Figure 2.14d), most methods slow down by 50% to 100% over other predicates (i.e. LESS THAN, GREATER THAN, EQUALITY, and INEQUALITY) because they need to simultaneously compare the column values to both the upper and the lower bounds in the predicate. However, BitWeaving/H is only 20% slower when evaluating the BETWEEN predicate compared to the other (simpler) predicates. Compared with BitWeaving/V, BitWeaving/H with the BETWEEN predicate is even slightly faster than BitWeaving/V when the width of code is less than 20 bits, and is only 15% slower than BitWeaving/V when the code is wider than 20 bits. Thus BitWeaving/H is more competitive with the BETWEEN predicate.

2.5.5 Effect of the bit group size

In the next experiment, we examine the effects of bit group size on scan performance. Figure 2.15 shows the scan performance of BitWeaving/V and BitWeaving/H for Q1, when



(a) Greater Than

(b) Equality

(c) Inequality

(d) Between

Figure 2.14: Performance on query Q1 with Various Predicates.

varying the bit group size from 1 to 32. Note that when the bit group size is 32, the BitWeaving/V (HBP+Prune) is identical to VBP (BitWeaving/H) as all the bits fit into the same bit group. In addition, when the bit group size is 1, BitWeaving/V is identical to the Bit-sliced method, with early pruning technique, because each bit is stored in a separate space.

We observe that the scan performance first increases and then decreases as the size of the bit group is increased. Both HBP+Prune and BitWeaving/V achieve the best scan performance when the size of bit group is 4. In some sense, BitWeaving/V is the hybrid solution between VBP and the Bit-sliced method (with early pruning technique), and

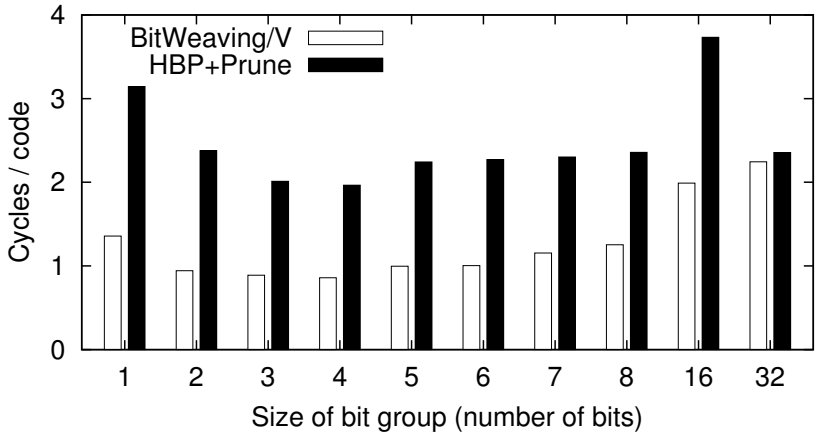


Figure 2.15: Performance when varying the size of the bit group.

achieves better performance than the other two methods.

2.5.6 TPC-H Evaluation

In this experiment, we use a scan query from the TPC-H benchmark [89] (The results for other queries in the TPC-H benchmark can be found in Chapter 4). This experiment was run against a TPC-H dataset at scale factor 10. The total size of the database is approximately 10GB. We compare the performance of the various methods on the query Q6. This query is shown below:

```
SELECT sum(l_extendedprice * l_discount)
FROM lineitem
WHERE l_shipdate BETWEEN Date and Date + 1 year
 and l_discount BETWEEN Discount - 0.01
 and Discount + 0.01 and l_quantity < Quantity
```

As per the TPC-H specifications for the domain size for each of the columns/attributes in this query, the column `l_shipdate, l_discount, l_quantity, l_extendedprice` are encoded with 12 bits, 4 bits, 6 bits, and 24 bits, respectively. The selectivity of this query is approximately 2%.

Figure 2.16 shows the time breakdown for the scan and the aggregation operations for the BitWeaving and the other methods. Not surprisingly, the Naive method is the slowest. The SIMD-scan method only achieves about 20% performance improvement over the Naive method, mainly because the SIMD-scan method performs relatively poorly when evaluating the BETWEEN predicates (see Section 2.5.4). Evaluating a BETWEEN predicate is complicated/expensive with the SIMD-scan method since the results of the SIMD compu-

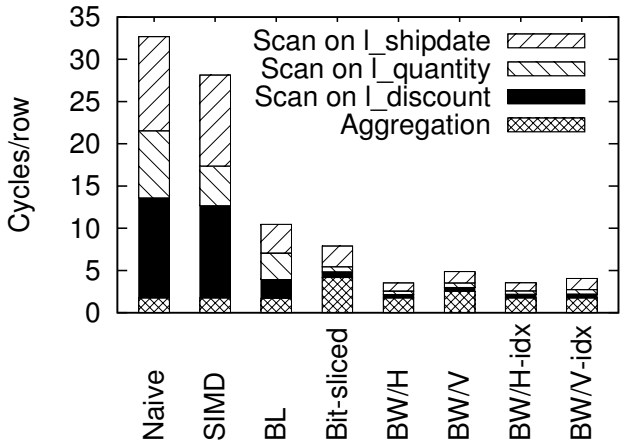


Figure 2.16: Execution time breakdown with the TPC-H benchmark (BW=BitWeaving).

tations are always stored in the original input registers. Consequently, we have to make two copies for each attribute value, and compare each copy with the lower and upper bound constants in the BETWEEN predicate, respectively.

The BL method runs at a much higher speed compared to the Naive and the SIMD methods. However, compared to BitWeaving/H, the BL method uses more instructions to implement its functionality of parallel processing on packed data and the conversion process to produce the result bit vector, which hinders its scan performance.

Note that both the BitWeaving methods (BW/H and BW/V) outperform all existing methods. As the column `l_extendedprice` is fairly wide (24 bits), BitWeaving/V spends more cycles extracting the matching values from the aggregation columns. As a result, for this particular query, BitWeaving/H is faster than BitWeaving/V.

We also evaluated the effects of using the BitWeaving methods as indices. In this method, the entire WHERE clause is evaluated using the corresponding BitWeaving methods on the columns of interest for this WHERE clause. Then, using the method described in Appendix A.1, the columns involved in the aggregation (in the SELECT clause of the query) are fetched from the associated column store(s) for these attributes. These column stores use a Naive storage organization.

In Figure 2.16, these BitWeaving index-based methods are denoted as BW/H-idx and BW/V-idx. As can be seen in Figure 2.16, BW/H-idx and BW/H have similar performance. The key difference between these methods is whether the aggregation columns are accessed from either the BW/H format or from the Naive column store. However, since using BW/H always results in accessing one cache line per lookup, its performance is similar to the lookup with the Naive column store organization (i.e the BW/H-idx case). On the other

hand, the BW/V-idx method is about 30% faster than the BW/V method. The reason for this behavior is that the vertical bit layout in BW/V results in looking up data across multiple cache lines for each aggregate column value, whereas the BW/V-idx method fetches these attribute values from the Naive column store, which requires accessing only one cache line for each aggregate column value.

We note that there are interesting issues here in terms of how to pick between BitWeaving/H vs. BitWeaving/V, and whether to use the BitWeaving methods as an index of for base storage. Building an accurate cost model that can guide these choices based on workload characteristics is an interesting direction for future work.

2.6 Related work

The techniques present in this chapter are applicable to main-memory analytics DBMSs. In such DBMSs, data is often stored in compressed form. SAP HANA [27], IBM Blink [10, 80], and HYRISE [54] use sorted dictionaries to encode values. Dynamic order-preserving dictionary was proposed to encode strings [12]. Other light-weight compression schemes can also be used for main-memory column-wise databases, such as [26, 2, 107].

SIMD instructions can be used to speed up database operations [104], and the SIMD-scan [96] method is the state-of-the-art scan method that uses SIMD. In this chapter we compare BitWeaving with this scan method. We note that BitWeaving can also be used in processing environments that do not support SIMD instructions.

The BitWeaving/V methods shares similarity to the bit-sliced index [71], but the storage layout of a bit-sliced index is not optimized for memory access, as well as our proposed early pruning technique. A follow-up work presented the algorithms that perform arithmetic on bit-sliced indices [81]. Some techniques described in that chapter are also applicable to our BitWeaving/V method.

The BitWeaving/H method relies on the capability to process packed data in parallel. This technique was first proposed by Lamport [58]. Recently, a similar technique [47] was used to evaluate complex predicates in IBM's Blink System [10].

Since the original BitWeaving paper [64] was published, there has been increasing interests in the BitWeaving technique. Feng and Lo applied the idea of intra-cycle parallelism to aggregation operations [28]. The ByteSlice storage layout [29], a variant of the BitWeaving/V method, was proposed to optimize two common access patterns in column-oriented database systems. Polychroniou and Ross used SIMD instructions to speed up the packing and unpacking operations for both horizontal and vertical bit packing methods

[76]. The BitWeaving method is also extended and evaluated in other emerging computer architectures, e.g. tightly-integrated GPUs and 3D die-stacking [77, 78].

2.7 Concluding remarks

With the increasing demand for main memory analytics data processing, there is an critical need for fast scan primitives. This chapter proposes a method called BitWeaving that addresses this need by exploiting the parallelism available at the bit level in modern processors. The two flavors of BitWeaving are optimized for two common access patterns, and both methods match the complexity bound for bit-parallel scans. Our experimental studies show that the BitWeaving techniques are faster than the state-of-the-art scan methods, and in some cases by over an order of magnitude.

Chapter 3

A padded encoding scheme to accelerate scans by leveraging skew

There has been a recent flurry of interest in efficient scan methods for main memory analytic data processing systems (e.g. the BitWeaving technique presented in Chapter 2, and [104, 80, 47, 96, 95, 29]). The key motivation for this interest is the frequent use of scans in these systems. There is also a recent trend towards building scan-based data processing engines, (e.g. [5, 80, 65, 69, 92, 33]). Such systems leverage the simplicity of scan operations to improve the effectiveness and predictability of the system, both in terms of query performance and scalability, when running in a distributed environment or on many-core machines.

To achieve maximum efficiency for the scan operation, data is often stored in compressed form using dictionary encoding or other fixed-length encoding schemes, e.g. null suppression and prefix suppression. Such schemes compress columns (attributes) and convert the native column (attribute) value to a code. The data for a column is represented using these codes, and these codes only use as many bits as are needed for the fixed-length encoding. The small code size presents opportunities to optimize the scan by performing predicate evaluation on multiple codes in parallel. However, existing encoding schemes do not exploit skew in the data distribution.

In this chapter, we propose an encoding scheme called *padded encoding* to leverage skew to enhance scan performance. Similar to the well-known Huffman encoding [40], padded encoding assigns smaller code lengths to frequent values. However, in contrast to the Huffman encoding, padded encoding preserves the order of codes to match the order of the native column domain values, thereby allowing for predicate evaluation on the (compressed) codes directly for higher performance. Our encoding method pads these variable-length codes to a fixed-length code block, producing a fixed-length encoding

scheme. Such fixed-length encoding are desirable, as they make storage management and query processing easier.

Scans on columns stored using our padded encoding scheme can *safely prune* the computation early without examining all the bits in each code when the code is stored in a vertical bit-packed representation. Such representations – e.g. [64, 71, 81, 29] – store each bit position in a column separately (think of these as column-stores but at the bit level). Safe *early pruning* in such methods reduces the memory bandwidth and CPU cycles that are consumed when evaluate scan queries, thus improving performance. Our padded encoding scheme can enhance this early pruning effect.

We note that our proposed padded encoding may produce longer codes than those produced by simpler popular methods such as simple dictionary encoding. However, the average number of bits that we need to access when evaluating a scan predicate is actually reduced when there is skew in the data and/or predicate literals, resulting in improved scan performance. A subtle but crucial point to note is that the encoding that we propose takes into account *both* the distribution of the column values and the distribution of the values that are used in query predicates to improve the scan performance. Accounting for only one of these aspects potentially misses opportunities for holistic optimization.

With the design of the padded encoding scheme in place, we next develop a cost function to capture the effects of both data and predicate skew for improved scan performance. We present an algorithm that can construct an optimal padded encoding in $O(n^3)$ time, where n is the number of distinct codes/values in an encoding. However, this algorithm may be prohibitively expensive in practice when applied to a large domain of values. The next key contribution of this chapter is to propose a more practical padded encoding algorithm that uses heuristics to provide a near-optimal solution while significantly reducing the algorithmic time complexity.

We have conducted an evaluation of the padded encoding using both micro-benchmarks and queries from the TPC-H benchmark. Our evaluation shows that the use of the padded encoding method improves scan performance by up to 3.8X for a highly skewed distribution. The additional overhead associated with the longer codes produced by the padded encoding is generally small, even in the worst-case where the padded encoding is constructed based on an incorrect distribution. We also evaluate our methods on the scan component of TPC-H queries, and show that the scan component can be accelerated as much as 3.4X. An astute reader may have noticed that the TPC-H dataset has no skew in the distribution of the column values, so any gains (even for scans) may seem surprising. We note that TPC-H dataset is in fact not an ideal test dataset for our method for precisely this reason, but our methods still improve the scan performance for some queries in the benchmark

as these queries have predicate literal skew – i.e. certain constants in predicates appear more frequently, and our methods can leverage that skew. If *both* the data and the predicate literals are uniformly distributed, our methods do not improve scan performance, and we acknowledge this fact.

This chapter makes four contributions. First, to the best of our knowledge, this is the first work to draw insight from the observation that skew in the distribution of the constants in query predicates can be leveraged for improved scan performance. Second, we design a padded encoding scheme that leverages data and predicate literal skew to improve scan performance. Third, we map the padded encoding problem to existing work, and then design a new practical (heuristic) algorithm to compute a padded encoding. Finally, we empirically demonstrate the effectiveness of our method.

The remainder of this chapter is organized as follows: Section 3.1 contains some preliminaries. Section 3.2 describes data and predicate literal skew. Section 3.3 describes the new padded encoding scheme, and Section 3.4 presents the algorithms to construct such encodings. Section 3.5 introduces further optimizations. Section 3.6 contains our experimental results. Related work is covered in Section 3.7, and Section 3.8 contains our concluding remarks.

3.1 Background

3.1.1 Encoding techniques for database systems

Encoding techniques have been extensively used in main memory analytical data processing settings in both the research community (e.g. [54, 26, 12, 2, 83, 64, 65, 8, 20]), and in industry (e.g. [27, 96, 79, 61, 56]). An encoding scheme compresses columns, or attributes, using a fixed-length order-preserving scheme, and converts a native column value to a *code*. In this chapter, we use the term "code" to refer to an encoded column value. The data for a column is represented using these codes, and these codes only use as many bits as are needed for the fixed-length encoding. Note that the size of a code can be an arbitrary number of bits, and the code need not be aligned to the boundaries of a byte or a processor word.

In these compression methods, all value types, including numeric and string types, are encoded as an unsigned integer code, resulting in a discrete domain for column values. For example, an order-preserving dictionary can map strings to unsigned integers [54, 12]. Floating point numbers can be converted using a scale scheme that multiples by a certain factor [26, 88]. These compression methods maintain an order-preserving one-to-one mapping between the column values and the codes. As a result, column scans can often

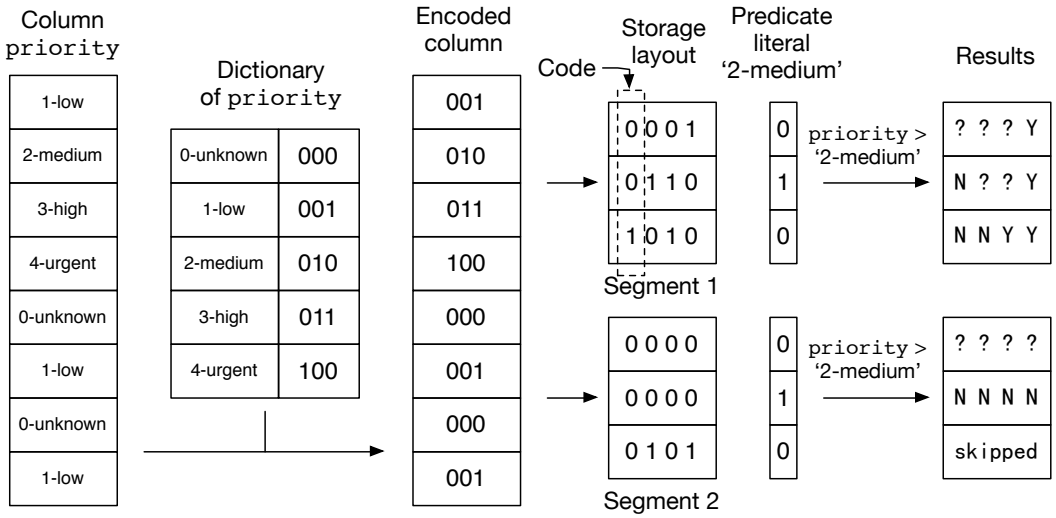
be directly evaluated on the codes. Codes may need to be recomputed if the underlying values change (i.e. new values are inserted/updated), but in many analytic settings the data is read-mostly. Nevertheless, methods to re-encode the database efficiently may be needed, and it is an interesting direction for future work.

3.1.2 Vertical bit-packing

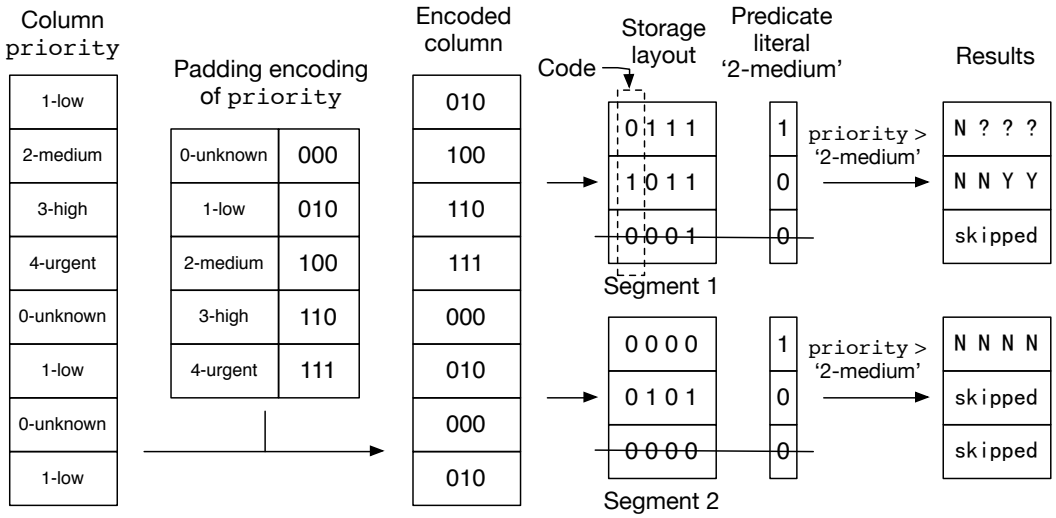
This chapter focuses on data stored in columnar vertical bit-packing format, such as the BitWeaving/V method presented in Section 2.4.1 and [71, 81, 29], as these have been shown to be very efficient (see Section 2.4.1). In this chapter, we call these methods *vertical bit-packing*, as they use a bit-level columnar data organization; i.e., the highest-order bits of all column codes are arranged sequentially, followed by the second highest-order bit, and so on.

The example shown in Figure 3.1a illustrates how a scan operation works on a vertical bit-packed column. Here, the column `priority` is encoded using an order-preserving dictionary, converting the native column values to 3-bit codes. With vertical bit-packing, the eight codes in the column are broken into two fixed-length segments. In each segment, we store the most significant bits of the four codes in the first processor word (this example assumes 4-bit processor words), followed by the second most significant bits, and finally the least significant bits. Thus, the vertical dashed box in the figure represents the code for the first column value.

A scan operation on a vertical bit-packed column leverages a key insight that within each processor clock cycle there is "abundant parallelism" as the processor's ALU operates on multiple bits in parallel. To find the tuples with a priority value higher than "2-medium", we evaluate the predicate on each segment independently. For each segment, a vertical bit-packing scan algorithm compares each pair of bits at the same bit position of the column code and a code that is obtained by encoding the predicate literal using the same dictionary, starting from the most significant bits to the least significant bits. In addition, the scan performs this bitwise comparison operation on a vector of codes in parallel. Consider Segment 1 in Figure 3.1a. By comparing the first bit of the predicate constant code (0) with the first bits of the four codes (0001), we notice that only the 4th code is guaranteed to be greater than the constant at this point (we use the symbol "?" to denote an unknown result in the figure). Next, we expand the comparison to the second bit between the constant and the column codes. Now, we also know that the 1st code is less than the constant because their first two bits (00) are less than the first two bits of the constant (01). Finally, by looking at the least significant bits, we obtain the comparison results for all the four column codes.



(a) Fixed-length encoding E_0



(b) Padded encoding E_2 (with base encoding E_1 from Figure 3.3)

Figure 3.1: Workflow of evaluating the predicate "`priority` > 2-medium" on the column `priority` with the example values.

The predicate evaluation is converted to a bitwise logical computation on processor words that are fully filled with bits from different column codes.

In many cases, we can further improve the efficiency of a vertical bit-packed scan operation by safely pruning the computation without looking at all the bits in the column. The intuition behind this *early pruning* technique is as follows. Consider Segment 2 in Figure 3.1a. By comparing the leading two bits of the predicate literal code with the leading two bits of the four column codes in this segment, we know that all the four codes (and hence the underlying column values) are less than "2-medium", and do not match the

predicate. At this point, all the codes have a *definite* answer with respect to this predicate, and we can terminate the computation on this segment. Organizing the column codes at the bit level allows us to skip over less significant bit positions once a predicate has been conclusively evaluated for all the codes in a segment.

In general, each segment contains a fixed number of codes, denoted as b. The segment size b equals to the number of bits in a processor word ($b = 64$ for modern processor ALU words and 128 - 512 bits for SIMD vectors). The probability of early pruning increases at successively larger bit position numbers (counting from the most significant bit at position 0). More specifically, let $f(i)$ denote the early pruning probability at bit position i. According to the definition, $f(i)$ equals to the probability that the b column codes are all different from the predicate literal in the most significant i bits. Assume that a column code and the predicate literal have the same value at a certain bit position with a probability of $1/2$ (which is true when the column values and the predicate literals are uniformly distributed, and the column values span the entire range of values that can be encoded by b bits.). Then, we have:

$$f(i) = (1 - (\frac{1}{2})^i)^b. \tag{3.1}$$

When $b = 64$, the early pruning probability at the bit position 12, $f(12)$, is above 99%, which indicates that in many cases we can *safely* terminate the scan after looking at the first 12 bits.

In this chapter, we focus on designing encodings that go beyond simple dictionary encoding to further enhance scan performance by exploring two opportunities: 1) assigning short codes to frequent predicate literals such that scans only need to access the prefixes of column value codes to compute a definite result for the predicate; 2) assigning short codes to frequent column values such that we relax the assumption that column codes and predicate literal codes have the same value at a bit position with a probability value of $1/2$ as discussed above, and make the early pruning occur at an earlier bit position. We will continue the example in Section 3.3 to show the basic idea behind the proposed encoding (Figure 3.1b).

3.2 Workload analysis

Data skew is a well-known phenomenon that has been extensively studied in the context of database systems. The term "data skew" loosely refers to several related but distinct types of skew. The key focus of this chapter is on two types of data skew: column value skew and predicate literal skew. Both column values and predicate literals need to be converted

to codes using the encoding algorithms that are proposed in this chapter.

Column Value Skew (CVS). Column value skew, or attribute value skew, is the best-known type of data skew. It broadly denotes a non-uniform distribution of column (attribute) values in databases. Column value skew is a property of the real-world data, and is also called *intrinsic skew* in the literature [93]. In order to compute a column value distribution, we simply count the number of occurrences of each distinct column values in a certain column.

Predicate Literal Skew (PLS). Predicate literal skew denotes an uneven distribution in the literals that appear in query predicates, which generally tend to choose certain values more frequently than others. Like column value skew, predicate literal skew derives from real-world data, independent of the storage or query processing algorithms. In order to calculate the distribution of predicate literals, for each predicate in the form of $\mathtt{R.a} \circ A$, where the operator $\circ \in \{<, >, =, \neq, \leqslant, \geqslant\}$, we increment the weight of the literal A; for each range predicate in the form of $\mathtt{R.a}$ `between` A_0 `and` A_1, we update the weights of both A_0 and A_1 (a BETWEEN predicate is equivalent to a conjunction of two inequalities); for each IN predicate $\mathtt{R.a}$ `in` $A_0, A_1, \cdots, A_n$, we increment the weights of all literals in the list, i.e. $A_0 \sim A_n$ (an IN predicate is equivalent to a disjunction of several equality predicates). We note that the distribution of predicate literals is independent of the operations that the predicate literal is involved in (e.g $<, >, =,$ `BETWEEN`, `IN`).

Predicate literal skew is related to but distinct from predicate skew, also known as filter skew [88, 7]. Predicate skew refers to a non-uniform distribution of the range of values that a predicate matches, whereas predicate literal skew refers to uneven distribution of the start and end values of the range, without regard to values between those bounds. For instance, the predicates "R.a between A_0 and A_1" and "R.a between A_0 and A_2" are two distinct predicates, but they share a common predicate literal A_0. Thus, the distribution on the predicate literals A_0, A_1, and A_2 is more skewed than the distribution on the predicates denoted by their range $[A_0, A_1]$ and $[A_0, A_2]$. In general, predicate literal skew has a higher degree of skew than predicate skew on a given database workload.

As a concrete example of predicate literal skew, consider the column `l_shipdate` in the TPC-H benchmark [89], which records the shipping dates of items. For each predicate literal on this column, e.g. the literal "1998-12-01" in the predicate "`l_shipdate` < 1998-12-01", we calculate its weight, i.e. how many instances of this literal occur in a predicate in one of the 22 TPC-H queries. Figure 3.2 plots the cumulative percentage of TPC-H queries that use a certain number of predicate literals from all those that appear in the benchmark. It is clear that the distribution of predicate literals on the column `l_shipdate` is highly skewed. (See [88] for other examples of skews in distribution of predicates.)

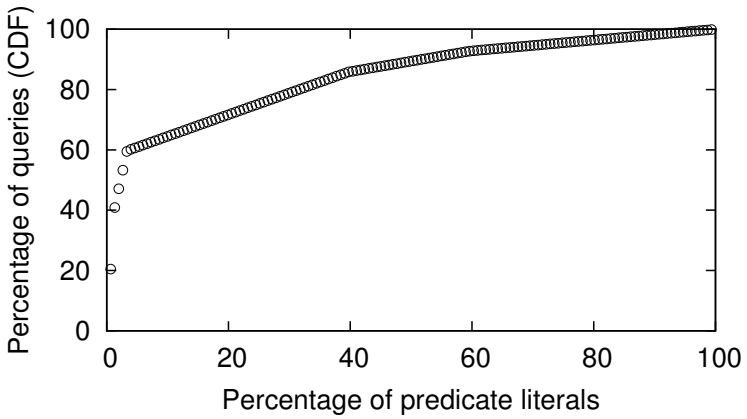


Figure 3.2: The distribution of predicate literals for the column `l_shipdate` in the TPC-H benchmark.

We note that the distribution of column values and the distribution of predicate literals can be unrelated. For example, one can imagine that shipping-date columns in a database may have frequent column values for dates that are close to holiday seasons, such as Thanksgiving and Christmas seasons, as there are likely more purchases made during these periods. In contrast, the frequent predicate literals that are used to query this attribute often include the first and the last day of each year, each quarter, each month, etc. These dates are frequently selected as predicate literals to run analysis queries on an annual, quarterly, and monthly basis.

The encoding algorithms proposed in this chapter focus on scenarios with both skewed column values and skewed predicate literals.

3.3 Padded encoding

There are two broad categories for data encoding schemes: *fixed-length encoding*, where each code consists of the same number of bits, and *variable-length encoding*, where the codes consist of a variable number of bits. Figure 3.3 shows a fixed-length encoding E_0 and a variable-length encoding E_1 constructed for the example column `priority`. While a variable-length encoding can leverage data skew to speed up scans by assigning smaller codes to frequent values, it is inefficient to locate and decode a code in an encoded column, and it is problematic to perform predicate evaluation on a set of codes in parallel (as we discussed in Section 3.1.2). Fixed-length encoding, on the other hand, misses the opportunity to leverage skew in the distribution of the column codes or the predicate literals to enhance scan performance for frequent values. Therefore, in this section, we propose a new encoding scheme, called *padded encoding*, to capture the best of both worlds.

Values	Fixed-length encoding E_0	Variable-length encoding E_1	Padded encoding E_2
0-unknown	000	00	000
1-low	001	01	010
2-medium	010	10	100
3-high	011	110	110
4-urgent	100	111	111

Figure 3.3: Example encodings E_0, E_1, and E_2 constructed for the example column `priority`.

Terminology. In this chapter, we use the word *code* to refer to a finite sequence of bits (binary digits). An *encoding* is a mapping that associates a code c_i with each column value a_i in a database column. A code c is said to be in an encoding E if there exists one value which could be converted to c using E. We use $l(c)$ to denote the length of a code c in terms of the number of bits. In addition, we use $l(E)$ to denote the maximum length of all codes in an encoding E. We say that we *encode* a value a to mean that we convert the value a to a code, and *decode* a code c to mean that we convert the code c to its corresponding (native) value.

3.3.1 Basic idea behind padded encoding

The basic idea behind the padded encoding method is to pad the codes produced by a variable-length encoding scheme into a fixed-length code block, making all codes in the padded encoding identical in code length. Formally, given a variable-length encoding E', we construct a padded encoding E on E' as follows: for each code c' in E', we add a sequence of $l(E') - l(c')$ zeros at the end of the binary representation of the code c', producing a new code c, i.e. $c = c' \underbrace{00\cdots00}_{l(E')-l(c')}$; then, c is the code associated with the value of c' in the padded encoding E. We use the term *base encoding* to mean the variable-length encoding that the padded encoding is constructed upon. We define the *effective length* of a code c, denoted as $l^*(c)$, in a padded encoding E as the length of the corresponding code c' in the base encoding E', i.e. $l^*(c) = l(c')$.

Example: *Figure 3.3 shows the padded encoding E_2 that is constructed on the variable-length encoding E_1 for the example column values. Thus, E_1 is the base encoding of E_2. To produce E_2, zeros are appended to the codes 00, 01 and 10 associated with the values "0-unknown", "1-low", and "2-medium", respectively. Thus, all codes in E_2 have a 3-bit representation.*

The padded encoding method provides an opportunity for performing efficient predi-

cate evaluation on an encoded column. On one hand, as all codes in a padded encoding have an identical code length, it is nearly as efficient as a fixed-length encoding to locate a code or perform predicate evaluation on an encoded column in parallel. On the other hand, it leverages the variable-length codes embedded in the padded encoded codes to speed up the predicate evaluation on frequent values (we will discuss two key properties that are required to achieve this goal in the section below). In short, the padded encoding has the potential to capture the best of both worlds in terms of scan performance on an encoded column.

3.3.2 Key properties

Two key properties are required of a base encoding in order to leverage padded encodings for efficient predicate evaluation on skewed column codes, or with skewed predicate literals:

- *Order-preserving property*. An encoding $\mathtt{E}$ has the order-preserving property if the numerical binary order of the codes (comparing bits lexicographically first-to-last) in $\mathtt{E}$ corresponds to the order of the associated values in $\mathtt{E}$.
- *Prefix property*. A base encoding $\mathtt{E}'$ has the prefix property if no code in $\mathtt{E}'$ is a prefix of any other code in $\mathtt{E}'$.

The order-preserving property of the base encoding enables predicate evaluation directly on the code domain when performing a scan. Lemma 3.1 shows that the order-preserving property is retained when constructing a padded encoding on a base encoding.

Lemma 3.1. *A padded encoding has the order-preserving property if its base encoding has the order-preserving property.*

Proof. We use a_1 and a_2 to denote two values. Let c_1 and c_2 denote the codes of a_1 and a_2 in the padded encoding, and c_1' and c_2' to denote their corresponding codes in the base encoding. If the base encoding has the order-preserving property, then we have $c_1' > c_2' \Leftrightarrow a_1 > a_2$. It is clear that adding zeros to the end of codes does not change the numerical binary order of c_1' and c_2', i.e. $c_1 > c_2 \Leftrightarrow c_1' > c_2'$. Thus, we obtain $c_1 > c_2 \Leftrightarrow a_1 > a_2$; that is, the padded encoding also has the order-preserving property. □

To perform a scan on an order-preserving padded encoded column, it is usually unnecessary to convert all codes in the column to the native column values and compare them with the predicate literal. Instead, since the order-preserving property guarantees

that the order of codes is identical to the order of native column values, we can encode the predicate literal and compare all codes with the code of the predicate literal directly without decoding. For example, say that we wished to find all tuples whose priority is higher than "2-medium" using the order-preserving example padded encoding E_2 in Figure 3.3. To accomplish this task, we simply convert the predicate literal to a code "100" using E_2, and then scan the column to find all tuples whose code is greater than "100".

The base (variable-length) encoding should also have the prefix property to speed up scans with frequent predicate constants. The lemma below shows the basic principle behind this observation.

Lemma 3.2. *The answer to a comparison between two codes* c_1 *and* c_2 *in a padded encoding* E *can be obtained by comparing up to* $\min(l^*(c_1), l^*(c_2))$ *leading bits so long as* E*'s base encoding has the prefix property.*

Proof. Let $P_l(c)$ denote the prefix of a code c of length l bits. We use c_1' and c_2' to denote the corresponding codes of c_1 and c_2 in the base encoding of E, respectively. According to the definitions, we have $c_1' = P_{l^*(c_1)}(c_1)$ and $c_2' = P_{l^*(c_2)}(c_2)$. Without loss of generality, we assume $l^*(c_1) < l^*(c_2)$. If we have $P_{l^*(c_1)}(c_1) = P_{l^*(c_1)}(c_2)$, then c_2 must be identical to c_1, because there does not exist another code in the base encoding of E whose prefix is the same as $c_1' = P_{l^*(c_1)}(c_1)$, according to the prefix property of the base encoding. If $P_{l^*(c_1)}(c_1) < P_{l^*(c_1)}(c_2)$, we have $c_1 < P_{l^*(c_1)}(c_1)+1 \leqslant P_{l^*(c_1)}(c_2) \leqslant c_2$, which implies $c_1 < c_2$. Similarly, if $P_{l^*(c_1)}(c_1) > P_{l^*(c_1)}(c_2)$, we have $c_2 < P_{l^*(c_1)}(c_2) + 1 \leqslant P_{l^*(c_1)}(c_1) \leqslant c_1$, which implies $c_1 > c_2$. In summary, the comparison result between $P_{l^*(c_1)}(c_1)$ and $P_{l^*(c_1)}(c_2)$ implies the comparison result between c_1 and c_2. Thus, we can compare the prefixes of length $l^*(c_1) = \min(l^*(c_1), l^*(c_2))$ to obtain the comparison result between c_1 and c_2. This completes the proof. □

Using Lemma 3.1 and Lemma 3.2, we can devise an efficient scan method for skewed column values and skewed predicate literals using padded encoding.

With predicate literal skew it is only necessary to access enough leading bits of the codes, i.e. the effective length of the literal code, until we get a definite result on the predicate. Given a vertical bit-packed column where all bit positions are stored separately (see Section 3.1.2), this method could enhance the scan performance by reducing both the number of instructions and the memory bandwidth that is needed to compute the scan operation.

Example: *Figure 3.1b shows the process of evaluating the example predicate on a column encoded with the padded encoding* E_2*. The effective length of the code associated with the predicate*

literal "2-medium" is two. Thus, we only need to access the leading two bits of each code to obtain the comparison results. Let us first focus on Segment 1 in Figure 3.1b. Even though the 2nd code has the same value as the predicate literal code in the leading two bits, we can still safely terminate the comparison at this bit position, as the 2nd code is guaranteed to be equal to the predicate literal code according to Lemma 3.2. Consequently, the scan with E_2 *has the potential to achieve a 1.5X speedup over the scan with* E_0 *that accesses all three bits of all codes in Segment 1 (shown in Figure 3.1a).*

For column value skew, we could quickly get definite results when comparing a set of column codes to a literal code, increasing the probability that the comparison between the whole set of column codes in a segment and the literal code safely terminates at an early bit position. To achieve this goal, we assign short codes for frequent column values, which in turn increases the likelihood that codes have distinct bit values in the leading bits.

Example: *Reconsider Segment 2 in Figure 3.1. Recall that with the encoding* E_0*, the column values "0-unknown" and "1-low" have the same bit value as the predicate literal "2-medium" at the 1st bit position (as shown in Figure 3.1a). The early pruning then occurs at the 2nd bit position. In contrast, the padded encoding* E_2 *assigns codes with short effective lengths to the frequent values "0-unknown", "1-low", and "2-medium" (as shown in Figure 3.1b), which generally increases the likelihood that the codes have different values in the leading bits. In the case of Segment 2, the column codes (000, 010) and the predicate literal code (100) have a distinct bit value at the most significant bit position. Thus, we can prune the predicate evaluation on Segment 2 earlier at the first bit position. As compared to* E_0*,* E_2 *has the potential to be 2X faster when evaluating the scan on Segment 2.*

3.4 Encoding algorithms

In this section, we present a series of encoding algorithms to construct optimal and near-optimal variable-length encodings. Then, a padded encoding can be constructed by appropriately padding these variable-length encodings.

3.4.1 Problem definition

Encoding algorithms for data skew take as input a sequence of n values, $a_0, a_1, \cdots, a_{n-1}$, that have been sorted in a domain-specific order, e.g. an alphabetical order for string values. We define the distribution of predicate literals as a sequence of weights, $p_0, p_1, \cdots, p_{n-1}$, where p_i represents the frequency that the value a_i is selected as a predicate literal in

queries. Similarly, we define the distribution of column values as a sequence of weights, $q_0, q_1, \cdots, q_{n-1}$, where q_i represents the frequency that the value a_i occurs in a column. The sum of all column value weights and predicate literal weights equals 1, i.e. $p_0 + p_1 + \cdots + p_{n-1} = 1$, and $q_0 + q_1 + \cdots + q_{n-1} = 1$.

The encoding problem is to find a variable-length encoding E with minimum value for the *cost of encoding* $C(E)$ amongst all variable-length encodings that have both the order-preserving property and the prefix property. The cost of encoding $C(E)$ is derived in more detail below in Section 3.4.2.

3.4.2 Cost model

In this section, we present a framework to capture data and predicate skews, and then use that framework to produce a mathematical model to create an optimal encoding. Given a variable-length encoding E, we quantify the cost of encoding $C(E)$ in terms of the expected cost of running a scan operation with this encoding. We validate this cost function in Section 3.6.1.6. The notations used in this section are summarized in Table 4.1.

Symbol	Definition
n	The number of distinct values in the column
b	The number of codes in a segment
a_i	The i-th value
c_i	The code of a_i
$l(c_i)$	The code length of c_i
p_i (q_i)	The predicate literal (column value) frequency of a_i
u_i (v_i)	The i-th internal (leaf) node in the binary tree
$w_p(u)$ ($w_q(u)$)	The predicate literal (column value) weight of a subtree rooted at an internal node u

Table 3.1: Summary of notations used in Chapter 3.

In the interest of simplicity of presentation, we use a binary tree representation to show a variable-length encoding. Each leaf node of the binary tree corresponds to a value in the encoding whose binary code defines a path from the root of the binary tree to the corresponding leaf node. In this path traversal, choosing the left branch at the level i corresponds to setting the i-th bit in the code to 0, and choosing the right branch maps to a bit value of 1. Figure 3.4 shows the binary tree representation of the example variable-length encoding E_1 from Figure 3.3.

The values associated with the leaf nodes are sorted in the domain specific order of the values, ensuring the encoding's order-preserving property. Let v_i denote the $(i+1)$-th

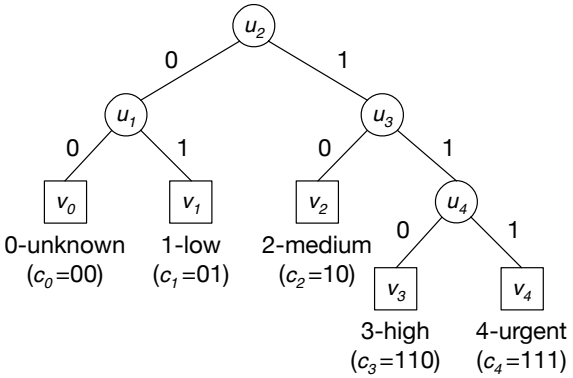


Figure 3.4: The binary tree representation of the example variable-length encoding E_1 from Figure 3.3.

leaf node in the binary tree. Thus, the leaf node v_i corresponds to the value a_i with the predicate literal weight p_i and the column code weight q_i. Also, we use $u_1 \sim u_{n-1}$ to denote the $n-1$ internal nodes. Here, u_i is the i-th internal node in symmetric order (i.e. in-order). We also use $w_p(u)$ to denote the predicate literal weight of a subtree rooted at an internal node u, i.e. the sum of the predicate literal weights p_i of all leaf nodes in this subtree. Similarly, $w_q(u)$ denotes the column code weight of a subtree rooted at the internal node u.

With the binary tree representation, the cost of encoding $C(E)$ can be calculated as the sum of the cost for each internal node, denoted as $C(u_i)$, as shown in Formula 3.2. Formula 3.3 shows the cost function for each internal node $C(u_i)$. Collectively, Formula 3.2 and 3.3 provide the cost function that is needed by the encoding algorithms. The unit of cost in these formula is the cost of accessing and comparing at one bit position.

$$C(E) = \sum_{i=1}^{n-1} C(u_i). \tag{3.2}$$

$$C(u_i) = (1 - (1 - w_q(u_i))^b) \cdot w_p(u_i). \tag{3.3}$$

Note that the cost of each internal node $C(u_i)$ only relies on the predicate literal weights and the column value weights of nodes that are in the subtree rooted at the internal node u_i, as $w_p(u_i)$ (or $w_q(u_i)$) represents the sum of the predicate literal (or the column value) weights of all leaf nodes in the subtree under u_i. This key property implies that we can design more efficient recursive encoding algorithms compared to an algorithm that exhaustively searches over all possible encodings.

Example: *Consider computational cost of the encoding* E_1 *shown in Figure 3.4. Assume that the column value weight of a value is the frequency that the value occurs in the example column*

shown in Figure 3.1. Thus we have $q_0 = 2/8$, $q_1 = 3/8$, $q_2 = q_3 = q_4 = 1/8$. *We also suppose that the only value that can be used as a predicate literal is "2-medium", thus we have* $p_2 = 1.0$, $p_0 = p_1 = p_3 = p_4 = 0.0$. *Suppose that the segment size* b *is 4 in this example. Then, we can calculate the cost associated with the internal node* u_3 *using Formula 3.3 as:* $C(u_3) = (1 - (1 - (1/8+1/8+1/8))^4) \cdot 1 = 0.847$. *Similarly, the cost of the root node is* $C(u_2) = (1-(1-1)^4) \cdot 1 = 1$. *The costs of other internal nodes are all zeros. Finally, we obtain the cost of the encoding* E_1 *as:* $C(E_1) = C(u_2) + C(u_3) = 1.847$. *In this case,* E_1 *actually has the minimum cost amongst all encodings that have both the prefix and the order-preserving properties.*

In the remainder of this section, we illustrate how we derive the cost function shown in Formula 3.2 and 3.3. The cost of the encoding $C(E)$ is the weighted sum of the cost of each literal code c_k, denoted as $C(c_k)$, over all possible literal codes in E, i.e. $c_0 \sim c_n$. Note that if a predicate literal is not present in the encoding, then it cannot be converted to a code. In this case, we transform it to a value in the encoding, without changing the semantic meaning of the predicates, using the method presented in Section 3.5.3. Thus, we can express $C(E)$ in terms of the number of values (i.e. n as defined in the beginning of Section 3.4.1) as follows:

$$C(E) = \sum_{k=0}^{n-1} C(c_k) \cdot p_k \tag{3.4}$$

Next, we analyze the cost of a literal code c_k, $C(c_k)$, in terms of the cost of a scan operation on a segment containing b codes, i.e. the cost of comparing a vector of b column codes to the literal code c_k. According to Lemma 3.2, the number of bits that must be accessed in order to obtain a definite answer for a comparison of a column code with the literal code c_k, is bounded by the length of the literal code: $l(c_k)$. Thus, even in the worst-case, we can always complete the comparison between the entire segment and the literal code c_k by comparing each bit of the literal code c_k. In certain cases, the comparison terminates before accessing all the $l(c_k)$ bits due to early pruning (see Section 3.1.2). We use $f(i)$ to denote the early pruning probability at the i-th bit position, i.e. the probability that we can terminate the computation at the i-th bit position (note, $f(0) = 0$ as the early pruning cannot occur before the first bit position). Let us assume that the cost of accessing a bit position is 1. Then, the cost associated with the i-th bit position can be represented by the probability that early pruning does not occur in its immediately previous bit position, i.e. $1 - f(i-1)$. Thus, the total cost for the predicate literal c_k is the sum of the costs at each bit position of c_k, i.e.:

$$C(c_k) = \sum_{i=1}^{l(c_k)} (1 - f(i-1)). \tag{3.5}$$

Next, we derive the early pruning probability $f(i)$, i.e. the probability that the b column codes in a segment are all different from the predicate literal c_k in the most significant i bits, by using the binary tree representation of the encoding. With the binary tree representation, the literal code c_k corresponds to a path from the root to the leaf node v_k. Let $U_i(c_k)$ denote the node at level i on the path associated with c_k (thus, the edge between nodes $U_{i-1}(c_k)$ and $U_i(c_k)$ corresponds to the i-th bit of c_k). The codes that share the common prefix of length i with the literal code c_k are exactly those in the subtree rooted at $U_i(c_k)$. Consequently, the probability that a code is identical to the literal code c_k in the leading i bits is the probability that the column code is in the subtree rooted at $U_i(c_k)$, i.e. the column value weight of the subtree $w_q(U_i(c_k))$. Hence, the early pruning probability on a segment of b column codes is:

$$f(i) = (1 - w_q(U_i(c_k)))^b \tag{3.6}$$

By substituting Equation 3.5 and Equation 3.6 into Equation 3.4, the cost of the encoding E can be rewritten as:

$$C(E) = \sum_{k=0}^{n-1} (\sum_{i=0}^{l(c_k)-1} 1 - (1 - w_q(U_i(c_k)))^b) \cdot p_k \tag{3.7}$$

Now, we reformulate formula 3.7 so that the cost of the encoding E is the sum of the cost on each internal node u_i, denoted as $C(u_i)$, of the binary tree associated with E. For a given literal code c_k, Formula 3.7 essentially adds the cost $(1 - (1 - w_q(U_i(c_k)))^b) \cdot p_k$ to each internal node $U_i(c_k)$ that we encounter in the path corresponding to c_k (note that the leaf nodes, which can be represented by $U_{l(c_k)}(c_k)$ in a path for c_k, are not involved in Equation 3.7). Thus, the cost associated with each internal node $C(u_i)$ is the sum of the cost associated with u_i in $C(c_k)$ for all possible literal codes c_k that are in the subtree rooted at u_i, as shown in Formula 3.3. Finally, we complete our derivation by rewriting Equation 3.7 as the sum of cost of all the internal nodes, obtaining Formula 3.2.

3.4.3 Optimal algorithm

We first present the optimal algorithm to find the minimum cost binary tree. The binary tree with the minimum cost, $C(E)$, is called an *optimal tree*. A key observation for an optimal tree is stated in the Lemma below.

Lemma 3.3. *All subtrees of an optimal tree are optimal.*

Proof. Suppose, to the contrary, that there exists a subtree T' in an optimal tree T that is not optimal. Then, there exists another subtree T'' that is built on all leaf nodes of T' and has a lower cost than T', i.e. $C(T'') < C(T')$. According to Equation 3.2, the cost associated with the tree T is the sum of the cost associated with T' and the cost associated with all other internal nodes that are not in T'. According to Equation 3.3, the cost associated with an internal node is only related to the predicate literal weights and the column code weights of codes that belongs to the subtree rooted at the node. Then, we can reduce the cost associated with T', without changing the cost of all other nodes in T, by substituting the subtree T' with T'', which results in reducing the cost associated with the tree T. This contradicts the supposition that T is optimal. Thus, Lemma 3.3 follows by contradiction. □

According to the lemma above, the problem at hand exhibits an optimal substructure: that is, the solution to the entire problem relies on solutions to subproblems. Naturally then, we can use a dynamic programing-based solution for this problem. Let us define a function $C(i, j)$ as the minimum cost of a binary tree built on a sequence of leaf nodes starting from the i-th leaf node v_i to the j-th leaf node v_j, where $(0 \leqslant i \leqslant j < n)$. In addition, we use $W_p(i, j)$ to denote the predicate literal weight of a subtree built on $v_i \sim v_j$, i.e. $W_p(i, j) = \sum_{k=i}^{j} p_k$. Similarly, let $W_q(i, j)$ denote the column code weight of a subtree built on $v_i \sim v_j$, i.e. $W_q(i, j) = \sum_{k=i}^{j} q_k$.

We can define $C(i, j)$ recursively as follows. If $i = j$, the problem is trivial; the tree consists of just one node with the cost equal to 0. To compute $C(i, j)$ when $i < j$, we take advantage of the optimal substructure shown in Lemma 3.3. We enumerate the value of k between i and j, splitting the tree between the $(k-1)$-th and k-th leaf nodes. Then $C(i, j)$ equals the minimum cost to compute the left subtree and the right subtree, plus the cost associated with the root node. Note that according to Equation 3.3, the cost of the root node is independent of the structure of the tree, and can be calculated as: $(1 - (1 - W_q(i, j))^b) \cdot W_p(i, j)$. Thus, we have

$$C(i, j) = \begin{cases} 0 & \text{, if } i = j \\ \min\limits_{i<k\leqslant j} (C(i, k-1) + C(k, j)) + & \\ \qquad (1 - (1 - W_q(i, j))^b) \cdot W_p(i, j) & \text{, if } i < j. \end{cases}$$

Algorithm 4 shows the pseudocode for an iterative implementation of this dynamic-programming solution (we omit the pseudocode for constructing the optimal tree). This solution requires a running time of $O(n^3)$ to find the optimal tree, because there are $O(n^2)$ cells in the matrix of $C(i, j)$, and it takes $O(n)$ instructions to compute each cell. The

algorithm requires $O(n^2)$ space to the store the $C(i,j)$ matrix.

Algorithm 4 Computing the optimal encoding

Input: The predicate literal weights $p_0, p_1, \cdots, p_{n-1}$
The column code weights $q_0, q_1, \cdots, q_{n-1}$
Output: The minimum cost of the tree

```
1: W_p(i,j) := Σ_{k=i}^{j} p_k, for 0 ⩽ i ⩽ j < n
2: W_q(i,j) := Σ_{k=i}^{j} q_k, for 0 ⩽ i ⩽ j < n
3: C_q(i,i) := 0, for 0 ⩽ i < n
4: for d := 2...n do
5:   for i := 0...n − d do
6:     j := i + d − 1
7:     C(i,j) := ∞
8:     for k := i + 1...j do
9:       if C(i,k−1) + C(k,j) < C(i,j) then
10:        C(i,j) := C(i,k−1) + C(k,j)
11:    C(i,j) := C(i,j) + (1 − (1 − W_q(i,j))^b) · W_p(i,j)
12: return C(0,n−1);
```

3.4.4 Near-optimal algorithm: AMPE

Since the optimal algorithm has an $O(n^3)$ time complexity, it is impractical for large values of n. In this section, we present a heuristic-based approach that provides a faster, but near-optimal solution. We call the algorithm AMPE for **A**pproximate **M**ethod to compute **P**added **E**ncoding. The AMPE algorithm uses two primary heuristics, namely the monotonicity heuristic (Section 3.4.4.1) and the early pruning heuristic (Section 3.4.4.2), and the code size reduction technique (Section 3.4.4.3).

3.4.4.1 Monotonicity heuristic

The first heuristic is from Knuth's solution to the problem of the optimal binary search tree [49, 50]. Knuth observed that there is a *monotonicity property* in the optimal binary search tree problem: that is, the root of the optimal tree constructed on a sequence of continuous leaf nodes $v_i \sim v_j$, where $(0 \leqslant i \leqslant j < n)$, is never on the left of the root of the optimal tree constructed on the leaf nodes $v_i \sim v_{j-1}$. Formally, let $R(i,j)$ denote the root position for the subtree for the leaf nodes $v_i \sim v_j$. Let r denote the node that is the root of this subtree. Then, the $R(i,j)$ value is computed as the index of the leaf node that is the leftmost leaf node in the right subtree of the node r. For example, in the encoding tree shown in Figure 3.4, $R(0,4) = 2$ and $R(2,4) = 3$. The monotonicity property states

that $R(i, j-1) \leqslant R(i, j)$. We can also obtain $R(i, j) \leqslant R(i+1, j)$ according to the symmetric property.

The encoding problem does not have a monotonicity property, as it uses a different cost function from the optimal binary search tree problem (see Appendix A.3 for a counterexample). Nevertheless, we observe that it is fairly rare that the root position of the optimal tree breaks the monotonicity property. Thus, we use the monotonicity property as a heuristic to quickly find a near-optimal solution.

The monotonicity heuristic suggests an improved algorithm. When computing the minimum value of $C(i, j)$, it is not necessary to enumerate all values in the entire range ($i \leqslant k < j$) to find the root position of the optimal tree, but we can focus on the limited range between $R(i, j-1)$ and $R(i+1, j)$.

We show the pseudocode for the modified dynamic programming approach with the monotonicity heuristic in Algorithm 5. The modified lines are marked with the symbol $\triangleleft$.

With the monotonicity heuristic, the running time of the algorithm is reduced to $O(n^2)$ [49]. The cost of the inner loop (Line 6-13) in Algorithm 5 is given by: $\sum_{0 \leqslant i \leqslant n-d, j=i+d-1} (R(i+1, j) - R(i, j-1) + 1) = R(n-d+1, n) - R(n-d, n-1) + R(n-d, n-1) - R(n-d-1, n-2) + \cdots - R(0, d-1) + n - d + 1 = R(n-d+1, n) - R(0, d-1) + n - d + 1 < 2n$. Then, as the outer loop takes on $n-1$ values, the algorithm requires a running time of $O(n^2)$.

Algorithm 5 Computing the near-optimal tree for the general case

Input: The predicate literal weights $p_0, p_1, \cdots, p_{n-1}$
The column code weights $q_0, q_1, \cdots, q_{n-1}$
Output: The minimum cost of the tree

1: $W_p(i, j) := \sum_{k=i}^{j} p_k$, for $0 \leqslant i \leqslant j < n$
2: $W_q(i, j) := \sum_{k=i}^{j} q_k$, for $0 \leqslant i \leqslant j < n$
3: $C_q(i, i) := 0$, for $0 \leqslant i < n$
4: $R(i, i) := i$, for $0 \leqslant i < n$ $\triangleleft$
5: **for** $d := 2 \ldots n$ **do**
6: **for** $i := 0 \ldots n - d$ **do**
7: $j := i + d - 1$
8: $C(i, j) := \infty$
9: **for** $k := R(i, j-1) \ldots R(i+1, j)$ **do** $\triangleleft$
10: **if** $C(i, k-1) + C(k, j) < C(i, j)$ **then**
11: $C(i, j) := C(i, k-1) + C(k, j)$
12: $R(i, j) := k$ $\triangleleft$
13: $C(i, j) := C(i, j) + (1 - (1 - W_q(i, j))^b)) \cdot W_p(i, j)$
14: **return** $C(0, n-1)$;

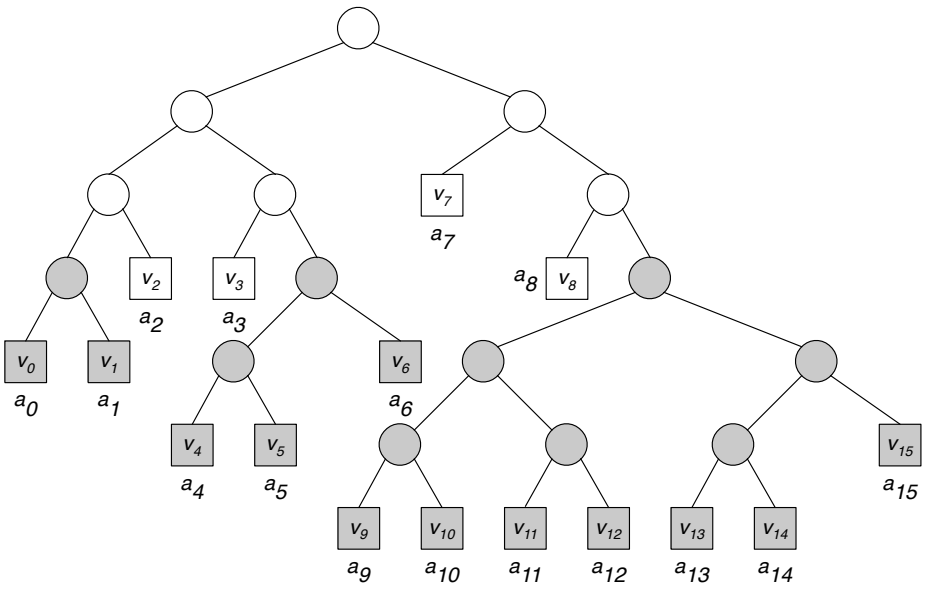


Figure 3.5: The tree constructed with the early pruning heuristic.

3.4.4.2 Early pruning heuristic

This second heuristic is based on an observation of the early pruning technique (see Section 3.1.2). When comparing a segment of b column codes to a literal code starting from the most significant bit, the probability for early pruning increases as the bit position number increases. At a certain position, the early pruning probability becomes very close to 100%, which indicates that in many cases we can safely terminate the scan after looking at a limited number of leading bits. In other words, the effects of both the predicate literal weights and the column code weights decline as we go down the binary tree from the root to the bottom levels, leading to a relatively balanced tree structure near the bottom of the binary tree. It is also the case that the leaf nodes that are close to the root in an optimal tree are usually associated with the most frequent values.

With this early pruning heuristic, we now present a preprocessing algorithm. Given a sequence of n values, we first find up to $2k$ values, called *frequent values*, which includes values with the top k predicate literal weights or with the top k column code weights. The other values remaining in the sequence are then split into a set of segments by the frequent values. For each segment, we construct a balanced binary tree. The root of each balanced binary tree is treated as a "virtual" value with the predicate literal (column code) weight equal to be the sum of the predicate literal (column code) weights of all the values in this subtree. A follow-up encoding algorithm is then performed on up to $2k$ frequent values and the up to $2k + 1$ "virtual" values.

Figure 3.5 shows an example binary tree constructed on 16 values using the early

pruning heuristic. In this example, we construct the binary tree with the 4 most frequent values (a_2, a_3, a_7, and a_8) ($k = 4$). Other values are split into three segments ($a_0 \sim a_1$, $a_4 \sim a_6$, and $a_9 \sim a_{15}$). For each segment, we build a balanced binary tree (marked grey in the figure), forming the subtrees with 2, 3, and 7 leaf nodes, respectively. Thus, a follow-up encoding algorithm needs to process only 7 nodes (4 frequent nodes plus the 3 roots of these balanced subtrees), instead of all the 16 nodes.

We select the value of k according to the bit position number where early pruning occurs with a fairly high probability. We denoted this bit position as e. Then, there are up to 2^e leaf nodes in the top part of the optimal tree that has a relatively unbalanced structure. Hence, we set $k = 2^e$. For example, when the segment size $b = 64$, $e \approx 12$, then we set $k = 2^{12} = 4196$.

This algorithm takes $O(n \log k)$ time to find the frequent values, and then $O(n)$ time to construct balanced binary subtrees. Together with the monotonicity heuristic, the follow-up encoding algorithm requires a running time of $O(k^2)$, as there are up to $2k$ frequent values and up to $2k + 1$ "virtual" values after the preprocessing algorithm. Thus, the time complexity of the near-optimal algorithm is $O(n \log k + k^2)$. The algorithm's space complexity is $O(n + k^2)$. With this near-optimal encoding algorithm, it now becomes practical to tackle the encoding problem for large values of n.

3.4.4.3 Reducing the code size

Our padded encoding method results in a fixed-length encoding, as fixed-length encodings are easier to manage for storage and query processing. However, the padded encoding method may produce longer (fixed-length) codes than those produced by popular order-preserving encoding methods, such as simple dictionary encoding. This additional space overhead has the potential to negate the performance gains of the padded encoding as: a) it increases the space that is needed to store the encoded column(s); b) it could hinder the scan performance if the distribution of the column values or the predicate literals changes; and c) it increases the execution time to fetch a code from the encoded column(s). In this section, we present a method to reduce the maximum code length of a padded encoding.

The basic idea behind this algorithm follows the early-pruning heuristic (see Section 3.4.4.2): the effects of both the predicate literal weights and the column code weights decline as we go down the binary tree from the root to the bottom levels, leading to a relatively balanced tree structure at the bottom of the binary tree. In the remainder of this section, we use the binary tree representation of the encoding. Thus, given a binary encoding tree, we replace the subtree that contains the deepest leaf node of the encoding tree with a balanced binary tree, reducing the maximum level of the encoding tree.

Algorithm 6 Reducing the maximum level of an encoding tree

Input: The binary encoding tree T
The level l
Output: A binary encoding tree T'
1: $T' := T$
2: $l_{max} := 0$
3: **for all** internal node u at level l of T **do**
4: $n :=$ the number of nodes in the subtree rooted at u
5: $l_{max} := \max(l_{max}, \lceil \log_2 n \rceil + l)$
6: **for all** internal node u at level l of T **do**
7: $l_u :=$ the maximum level of nodes in the subtree rooted at u
8: **if** $l_u > l_{max}$ **then**
9: Replace the subtree rooted at u in T' by a balanced tree
10: **return** T'

Algorithm 6 shows the pseudocode for the algorithm that reduces the maximum level of a binary encoding tree, starting from an input level l. In the first loop (Line 3 to Line 5), we compute the maximum level (l_{max}) of the encoding tree if we replace the subtree rooted at each internal node at the level l by a balanced binary subtree. Then, in the second loop (Line 6 to Line 9), we perform the substitution process. However, if the maximum level in a subtree is lower than l_{max}, replacing this subtree cannot reduce the maximum level of the encoding tree, but could degrade the performance of scans with this encoding. As a result, we only substitute a subtree if the maximum level of the subtree exceeds l_{max}. We note that all nodes above the level l remain unchanged using this algorithm. This solution requires a running time of $O(n)$, because we need to access and reconstruct all subtrees below the level l in the worst-case.

3.5 Optimizations

In this section, we discuss several techniques that further improve the encoding algorithms proposed in this chapter in terms of performance and applicability. More specifically, we present a method to support scans that compare two distinct columns in Section 3.5.1. Then, we discuss why and how we construct a padded encoding for clustered columns in Section 3.5.2. In Section 3.5.3, we describe a method that transforms a predicate to a semantically identical predicate, but with a more frequent predicate literal. Finally, we discuss how we preserve data locality in Section 3.5.4.

3.5.1 Handling comparisons between columns

So far the main focus of the type of predicates is a comparison between a column and a predicate literal. There is another common type of predicate that compares two distinct columns, which can be represented canonically as $R.a1 - R.a2 \circ A$, where the operator $-$ is the minus operator, the operator $\circ \in \{<, >, =, \neq, \leqslant, \geqslant\}$, and A is a predicate literal. For example, we might want to find the sale records whose shipping date is within one week of the order date by using the predicate `shipdate` $\leqslant$ `orderdate + 7`. However, scans with this type of predicate do not directly benefit from the encoding techniques proposed in this chapter so far.

To handle predicates that compares two columns $R.a1$ and $R.a2$, we propose adding a *delta column*, which is a materialized column representing the difference between the values from the two columns, i.e. $R.a1 - R.a2$. Thus, predicates that compare these two columns, i.e. $R.a1 - R.a2 \circ A$, now becomes a "regular" predicate that compares a delta column to a predicate literal, i.e. $\texttt{delta}(R.a1, R.a2) \circ A$. An encoding can now be constructed on the delta column based on the column value weights and predicate literal weights of the values appearing in the delta column, making it feasible to leverage our padded encoding technique to speed up scans with this type of predicate.

Considering the above example again, assume that the predicate literal 7 is a frequent predicate literal on the delta column that is built on `shipmate - orderdate`. Then, the scan with the example predicate runs at a higher speed as we can assign a short code to the value 7 in the encoding constructed for the delta column.

In practice, we only construct delta columns for comparisons that actually show up frequently in the query workload. These delta columns consume extra space, and present a space vs. time tradeoff for speeding up scan operations on these columns.

3.5.2 Encoding for clustered columns

The assumption that an encoding is constructed on a single column can be relaxed. Thus, we can produce an encoding for a cluster of columns that share the same domain, and have a similar distribution for their column values. To construct an encoding for a set of columns, the predicate literal weights and the column value weights have to be computed across all columns that share the encoding. Then, we encode the values in these columns using the constructed encoding. As all codes in these columns are produced by using the same encoding, it is efficient to compare the column values from two different columns.

For instance, we can directly compare the codes for the columns `shipdate` and `orderdate` without decoding, if the two columns use the same encoding. The shared encoding also

reduces the space consumption that is used to store multiple individual dictionaries.

3.5.3 Transforming predicates

A predicate on a scan operation can often be transformed into a semantically identical predicate but with a distinct predicate literal. This is because the domain of an encoded column is discrete, consisting of the values in the encoding. Thus, we can replace the literal in a predicate with its immediate previous or next value in the domain and correspondingly change the comparator. For example, a predicate `shipdate <= '2014-10-31'` can be rewritten as `shipdate < '2014-11-01'` without changing the semantic meaning of the predicate.

There are three primary reasons to consider transforming predicates. The first reason is that we could switch to a more frequent predicate literal by transforming the predicate. In the example above, it is worthwhile to transform the predicate to `shipdate < '2014-11-01'` if the predicate literal weight for the date "2014-11-01" is higher than that of "2014-10-31", leading to a scan with a shorter predicate literal code.

Second, the predicate transformation may produce a predicate literal distribution with a higher degree of skew, which often results in a more efficient encoding. For example, if we transform the predicate literal "2014-10-31" in all queries of a database to the literal "2014-11-01", then the predicate literal weight of the date "2014-10-31" is shifted to that of the date "2014-11-01". Thus, we now have a more skewed distribution for the predicate literals, which improves the efficiency of the padded encoding scheme.

Finally, we have to transform the predicate if the predicate literal is not in the encoding. For instance, if the date column does not contain the date value "2014-10-31," but a query is issued with the predicate `shipdate < '2014-10-31'`, then there is no assigned code for the date "2014-10-31" in the encoding. In this case, we have to convert the predicate literal with the immediate previous or next value in the encoding. Note that the decision between choosing the immediate previous value or next value can be made based on the weights of these two values.

3.5.4 Preserving data locality

Although the padded encoding method may increase the code lengths and may allow the scan operation to skip over some bit positions, the storage layout of the BitWeaving(/V) method naturally preserves the data locality for the padded encoding. With the BitWeaving storage layout, a column of codes is broken down into fixed-length segments. The codes in a segment are then transposed into the vertical bit-packing representation, and are further

divided into fixed sized bit groups. This storage layout is designed to improve the data locality and fully utilize the memory bandwidth. We omit a detailed discussion of this aspect in this chapter, and refer interested readers to Section 2.4.1.1.

3.6 Evaluation

Our experimental server has dual 2.0 GHz Intel Xeon E5-2620 6-core processors, and 32GB of 1600 MHz DDR3 main memory. The server runs 64-bit Linux 2.6.32. All experiments are run in an in-memory setting as the database fits entirely in memory.

We have implemented the encoding techniques proposed in this chapter in C++. We used an implementation of the BitWeaving/V [64] technique, as the vertical bit-packing method to store and scan the encoded columns. We compiled the code using g++ 4.4.7 with optimization flags (`-O3 -march=native`).

3.6.1 Micro-benchmark evaluation

For the first experiment, we create a table `R` with a single column `a`. The column `R.a` contains one billion values. (We have also experimented with different cardinalities for this table, and found that the results are similar to the case with one billion values; thus, we omit these additional results in the interest of space.) The column values are integer numbers in the range of $[0, 2^d)$, where `d` is a parameter to adjust the domain size. By default, the parameter `d` is set to 12, but we evaluate the impact of this parameter (see Section 3.6.1.3). The performance of an encoding is evaluated by the time it takes to execute a scan query on the column with a simple predicate `R.a < A`. The value of `A` is selected based on the distribution of the predicate literals. The performance for other predicates is similar to that with the less than ($<$) predicate, and is omitted in interest of space.

In the evaluation below, we generate three set of workloads with: a) only skewed predicate literals, called *UD-SP* (Uniform-Data-Skewed-Predicates), b) only skewed column values, called *SD-UP* (Skewed-Data-Uniform-Predicates), and c) with both skewed predicate literals and skewed column values, called *SD-SP* (Skewed-Data-Skewed-Predicates). The skewed column values/predicate literals follow a Zipfian distribution, with the skew factor varying from 1.0 to 2.0. For SD-SP, the distribution of column values are different from the distribution of predicate literals.

We construct padded encodings on the three sets of workloads. The encodings are constructed on all integer values in the domain, no matter whether or not a value appears in the column. For the workload UD-SP (or SD-UP), we set equal values for the column value

(or predicate literal) weights, i.e. $q_0 = q_1 = \cdots = q_{n-1} = \frac{1}{n}$, (or $p_0 = p_1 = \cdots = p_{n-1} = \frac{1}{n}$). We use the AMPE algorithm (see Section 3.4.4) to construct the padded encodings.

To serve as a yardstick, we also include the performance on an uniformly distributed workload, called *UD-UP* (Uniform-Data-Uniform-Predicates). To encode this workload, we use a simple null suppression encoding that removes leading zeros and converts the column values in the range $[0, 2^d)$ to d-bit codes. This simple encoding does not exploit the skewed distribution of column values and predicate literals.

Each experiment was run 10 times with different datasets that are produced by our Zipfian data generator. For each run, we generate 100 queries based on the predicate literal distribution. We report the average execution time for the 10×100 queries.

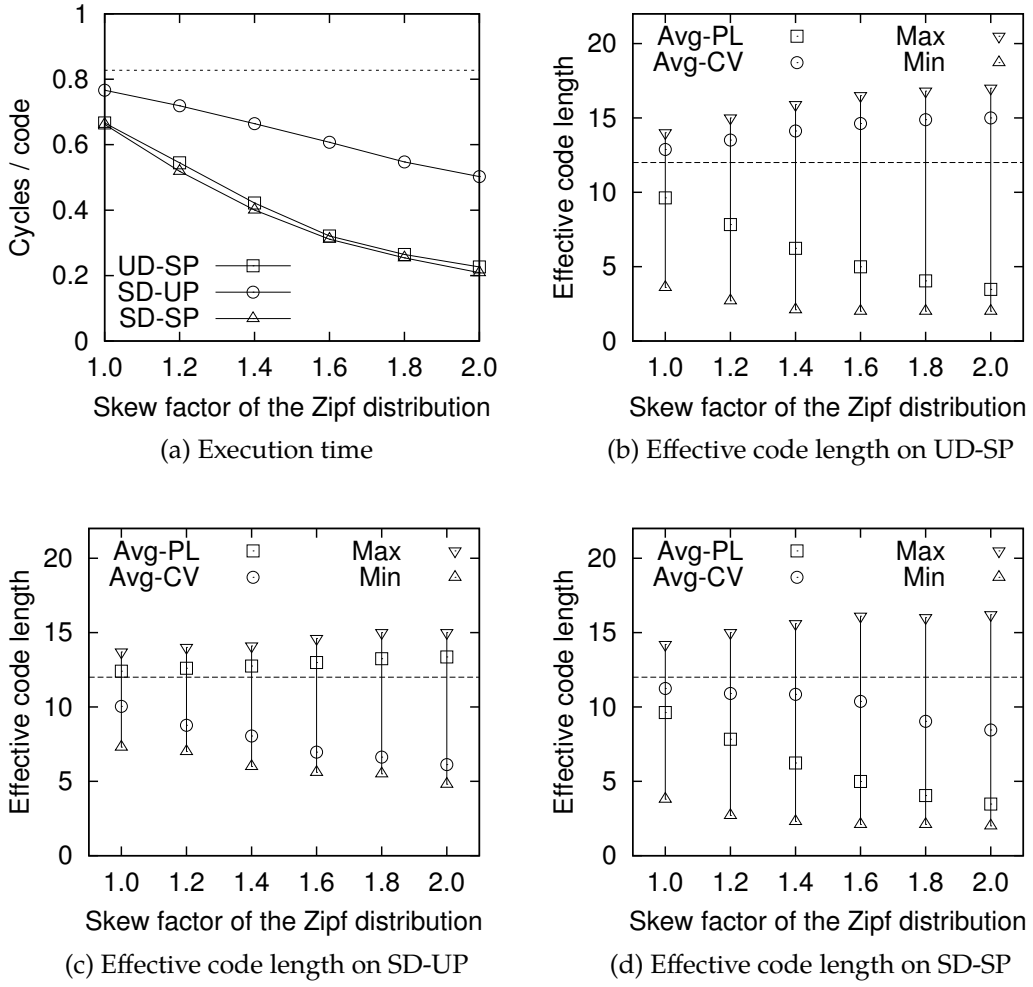
3.6.1.1 Varying skew factor

In this evaluation below, we evaluate the scan performance (the focus of this chapter) with the padded encoding on the three sets of workloads (UD-SP, SD-UP, and SD-SP).

Figure 3.6a shows the scan performance with padded encoding when varying the skew factor of the Zipfian distribution. We also mark a horizontal line in the figure to indicate the scan performance of the simple encoding on UD-UP. As can be seen in the figure, not surprisingly, the scan execution time reduces as the skew increases on all the three workloads. On UD-SP, the padded encoding improves scan performance by up to 3.6X over the simple encoding. On SD-UP, the padded encoding is relatively less effective than that on UD-SP, achieving an up to 65% speedup over the simple encoding. The reason for this behavior is because the scan performance is determined by: 1) the effective code lengths of all column codes in a segment, and 2) the effective code length of the only predicate literal code. Thus, a short code may enhance the scan performance if it is used as a predicate literal, but might not necessarily improve the scan performance if it is contained in a segment as a column code.

As can be seen in Figure 3.6a, the padded encoding results in the best scan performance on SD-SP, achieving up to 4.0X speedup over the simple encoding. We see that the performance improvement on SD-SP is mainly due to predicate literal skew, rather than column value skew, as the SD-SP curve is much closer to the UD-SP curve. The small gap between the SD-SP and the UD-SP curves is due to the fact that the encoding on SD-SP is also optimized for column value skew, in addition to predicate literal skew.

To better understand the effects of the padded encoding on the three workloads, we plot the range of *effective code length*, i.e. the code length without the padded zeros, on UD-SP, SD-UP, and SD-SP in Figure 3.6b, 3.6c, and 3.6d, respectively. In these figures, the bar at each skew factor value represents the range of effective code length of the padded



(a) Execution time

(b) Effective code length on UD-SP

(c) Effective code length on SD-UP

(d) Effective code length on SD-SP

Figure 3.6: Execution time and effective code length of the padded encoding scheme when varying the Zipfian skew factor.

encoding (thus, the low-end and high-end of the bar represent the minimum and maximum effective code length). We also plot two points in each bar corresponding to the weighted average effective code length based on the predicate literal weights (with the tag "Avg-PL"), and column value weights (with the tag "Avg-CV"), respectively. Essentially, the "Avg-PL" point represents the average effective code length of the values that are used in predicates, whereas the "Avg-CV" point shows the average effective code length of the codes in the column. In these figures, we also draw a horizontal line at 12 to represent the code length of the simple encoding scheme.

As can be seen from Figure 3.6b ~ 3.6d, the maximum code length of the encodings on the UD-SP, the SD-UP, and the SD-SP datasets are 17.0, 15.0, and 16.2, respectively. All these padded encodings amplify the size of codes, as compared to the code length 12 of

the simple encoding. In spite of this, the average effective code length generally decreases as the skew increases. On the UD-SP dataset, the average effective code length with the predicate literal weights (Avg-PL) drops quickly. However, since the encoding on the UD-SP dataset only optimizes for predicate literal skew, the weighted average effective code length with the column value weights (Avg-CV) increases as the maximum effective code length increases. Similarly, the encoding on the SD-UP dataset only reduces the average effective code length with the column value weights. On the SD-SP dataset, nevertheless, the encoding makes a trade-off between the predicate literal skew and column value skew, and therefore reduces the average effective code length with the column value weights and the predicate literal weights. We also see that the encoding on the SD-SP dataset reduces the code length by around 1 bit while improving the scan performance relative to that on the UD-SP dataset.

The space that is needed to store an encoded column is proportional to the code length of the encoding, i.e. the maximum effective code length of the padded encoding (the "Max" points in Figure 3.6 (b-d)), or the code length of the simple encoding (the dashed lines in Figure 3.6 (b-d)). Thus, as can be seen in Figure 3.6 (b-d), the column encoded with the padded encoding is up to 42%, 25%, or 35% larger than that with the simple encoding on the UD-SP, the SD-UP, and the SD-SP datasets, respectively.

3.6.1.2 Varying locality

Next, we evaluate the effects of the locality of frequent values on the scan performance with padded encoding. Consider the shipping date example that we discussed previously in Section 3.2. The frequent values are the dates that are close to holiday seasons. Thus, the frequent values tend to be located closer to each other sequentially. We call this phenomenon the *locality of frequent values*. In this experiment, in order to study the effect of the locality, we restrict the top 1% frequent values in the range $[0, 2^{12} \cdot \alpha)$, where α is a locality factor and is varied from 1% to 100%. For example, when $\alpha = 10\%$, the top 1% frequent values are within a small key range from 0 to $2^{12} \times 10\%$.

Figure 3.7 plots the execution time of the scans on the UD-SP and the SD-UP datasets, and with the locality factor set to 1%, 10%, and 100%. On the SD-UP dataset, the scan speed is further improved by up to 30% when there exists strong locality of frequent values. In this case, most column codes in a segment tend to have similar prefixes. As a result, it is more likely that the early pruning occurs at an earlier bit position, which in turn enhances the scan performance. Also, when values exhibit strong locality, the encoding algorithm can often construct a more effective binary tree as frequent values are closer to each other. In contrast, the encoding on the UD-SP dataset only achieves marginal additional speedups

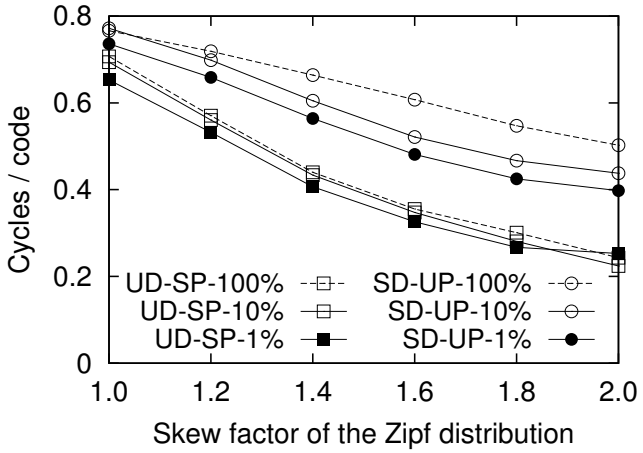


Figure 3.7: Execution time of scans varying the locality of frequent values.

with locality of frequent values. The effect of locality of frequent values on the SD-SP dataset is similar to that on the UD-SP dataset, and is omitted in the interest of space.

3.6.1.3 Varying domain sizes

Next, we evaluate the performance when varying the domain size from $[0, 2^4)$ to $[0, 2^{28})$. Note that the code size of an encoding is determined by the size of the domain that the encoding is constructed on. For example, the code size of the simple encoding is varied from 4 to 28 for the domains evaluated in this experiment.

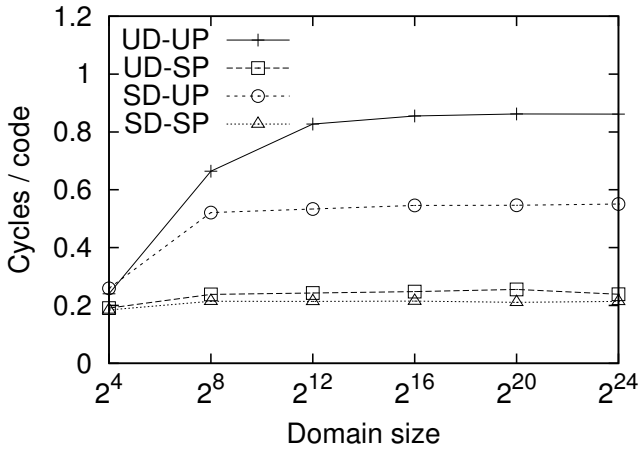


Figure 3.8: Execution time of scans varying the domain sizes.

Figure 3.8 compares the scan speeds with the simple encoding and the padded encoding with varying domain sizes. When the domain size is 2^4, scans on the UD-SP and the SD-SP datasets are only 50% faster than those on the UD-UP and the SD-UP datasets. As the

domain size increases to 2^8, the execution time of the scan on the UD-UP dataset quickly rises, while those on the UD-SP and the SD-SP datasets change very little. When the domain size exceeds 2^{12}, the execution time of scans on the four workloads no longer increases, as at this point the early pruning occurs in nearly all cases. In summary, the performance improvement of the padded encoding scheme over the simple encoding scheme increases when the domain size is smaller than 2^{12}, and does not change once the domain size exceeds 2^{12}. Note that the execution time of scans on the UD-SP and the SD-SP datasets remains nearly constant across all domain sizes.

3.6.1.4 Robustness evaluation

In this experiment, we evaluate the robustness of the padded encoding scheme, i.e. the performance of padded encoding if the distribution that the encoding is constructed on does not match the real distribution. Note that the distribution of column values changes slowly as new values are inserted into the column, especially when the number of new values is a small fraction of the original dataset. As a result, the old distribution is often still fairly accurate. Thus, we focus on the scenario where the distribution of predicate literals changes, i.e. we construct padded encodings on the UD-SP and the SD-SP datasets based on a Zipfian distribution with the scale factor set to 2.0, but generated queries with uniformly distributed predicate literals (thus, SD-SP and UD-SP temporarily become SD-UP and UD-UP, respectively).

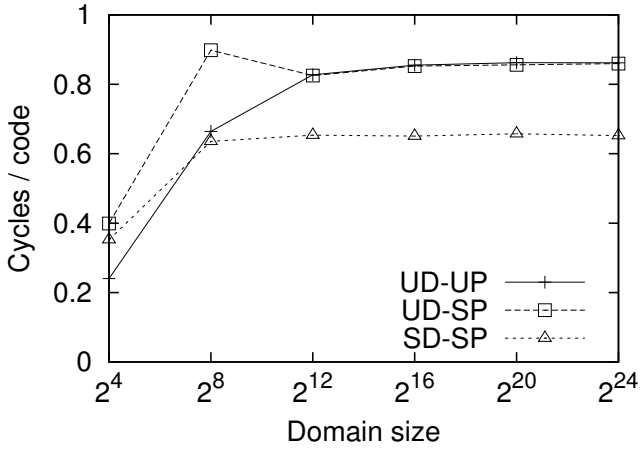


Figure 3.9: Robustness evaluation.

Figure 3.9 plots the execution time of scans with the encodings that are constructed on an incorrect distribution of predicate literals. We include the performance of the simple encoding on UD-UP as a yardstick of performance. The results on SD-UP are omitted here,

as its performance is irrelevant to the distribution of predicate literals. When the domain size is smaller than 2^{12}, the incorrect distribution imposes an up to 66% performance penalty (relative to UD-UP) on the UD-SP and the SD-SP datasets. This is because the padded encodings often have wider codes than the simple encoding. If the distribution of predicate literals is incorrect, then the scan does not benefit from the small codes associated with the frequent values in the padded encoding, but suffers from the increased code size of the entire encoding. However, when the domain size exceeds 2^{12}, there is no performance penalty on the UD-SP and the SD-SP datasets, because nearly all comparisons safely terminate around bit position 12 due to early pruning, even for the codes whose length is extended by the padded encodings.

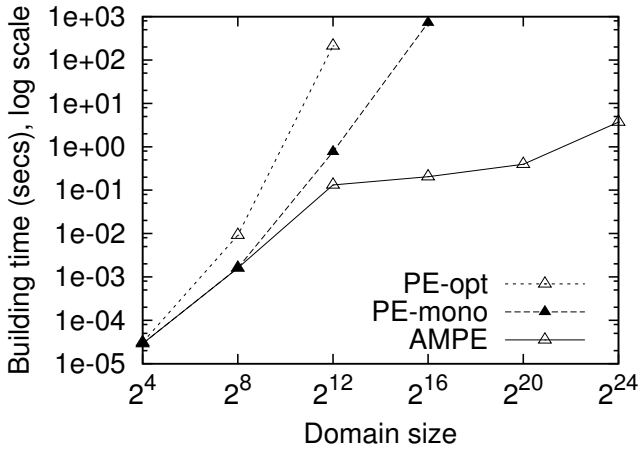
From the results shown in Figure 3.9, we see that the scan on the SD-SP dataset is still 25% faster than the scan on the UD-UP dataset, even though the encoding is constructed on an incorrect distribution on predicate literals. This performance improvement results from the effect of column value skew. In this case, the encoding on the SD-SP dataset has a similar effect to the padded encoding on the SD-UP dataset. Although the padded encoding with both predicate literal skew and column value skew is only slightly faster than that with predicate literal skew (see Figure 3.6a), it is more resilient to changes in the distribution of predicate literals.

3.6.1.5 Optimal vs. near-optimal encodings

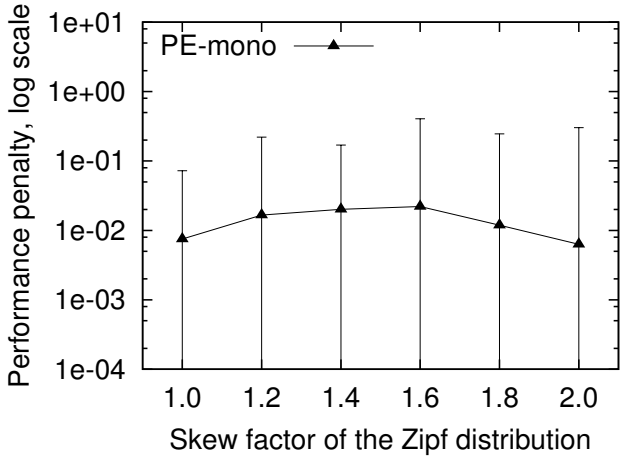
Finally, we compare the optimal and near-optimal encoding algorithms in terms of the encoding building time and the scan performance. The tag "PE-opt" and "PE-mono" refer to the optimal encoding algorithm (c.f. Section 3.4.3), and the near-optimal algorithm with the monotonicity heuristic (c.f. Section 3.4.4.1), whereas "AMPE" refers to the near-optimal algorithm with both monotonicity and early pruning heuristics (c.f. Section 3.4.4).

Figure 3.10a plots the building time of the optimal and near-optimal encoding algorithms when varying the domain size. Not surprisingly, the building time of PE-opt and PE-mono quickly increases as the domain size increases, due to their high time complexity. The AMPE method is much faster than both PE-opt and PE-mono, taking less than four seconds to construct a padded encoding with $2^{24} = 16$ million values. Thus AMPE enables practical applications of our proposed padded encoding scheme.

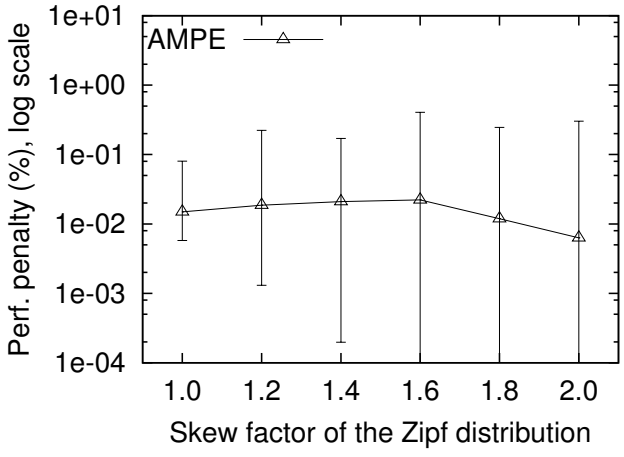
Figure 3.10b and 3.10c show the average performance penalty of the two near-optimal encodings, PE-mono and AMPE, over the optimal encoding PE-opt, respectively. The error bars represent the minimum and maximum performance penalty in 100 runs for each experiment. The performance penalty is shown as a percentage of the increase in the execution time of scans. Since the measured execution time of scans is not sufficiently



(a) Building time



(b) Performance penalty of PE-mono



(c) Performance penalty of AMPE

Figure 3.10: Comparisons between the optimal and near-optimal encoding algorithms on the SD-SP dataset.

accurate to reflect the performance penalty, we measured the performance penalty based on the cost function we developed in Section 3.4.2. As can be seen from the figure, the average performance penalty of PE-mono is between 0.1% and 0.0001%. AMPE adds extra penalty when the skew factor is low, and has similar penalty with PE-mono when the skew factor increases. The maximum performance penalty for both near-optimal algorithms across all skew factors is less than 1%.

3.6.1.6 Validating the cost function

Figure 3.11 compares the measured time and the theoretical cost of scan operations with the padding encodings on the UD-SP, the SD-UP, and the SD-SP datasets. The solid curves represent the measured execution time, whereas the dashed curves show the theoretical cost according to the cost model (present in Section 3.4.2). With the UD-SP dataset, the theoretical cost matches the measured time across all the skew factors. With the SD-UP and the SD-SP datasets, the lines associated with the measured time have a similar trend to that associated with the theoretical cost, but are generally below the the cost curves. The reason for this behavior is because the implementation of the early pruning mechanism imposes an extra performance overhead that is not accounted for in the theoretical model. Improving the implementation of the early pruning mechanism is an interesting direction for future work.

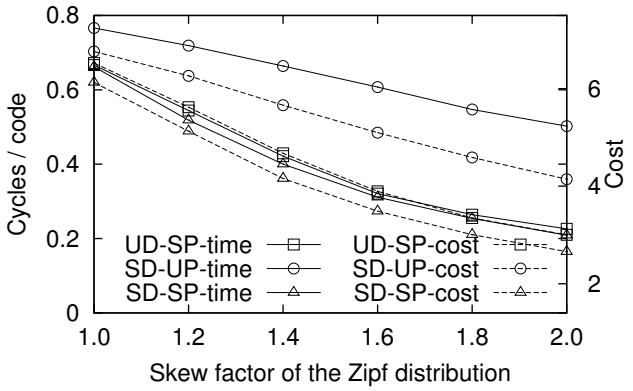


Figure 3.11: Comparison between the execution time and the theoretical cost of a scan operation with the padded encodings.

3.6.2 TPC-H evaluation

In the evaluation below, we use the TPC-H benchmark [89] to evaluate the encoding methods. These experiments were run against a TPC-H dataset at scale factor 10. The total

size of the database is approximately 10GB.

Although the TPC-H data generator assumes a uniform distribution for data in the database, some of the predicate constants in the TPC-H queries are not uniformly distributed, e.g. the first of January of each year is frequently used on predicates. As a result, we produced padded encodings for the TPC-H benchmark based on the distribution of predicate literals, as per the TPC-H specification. To capture the uniform data distribution aspect of the benchmark dataset, we simply set the column value weights to $1/n$.

Encoding names	Columns	Code sizes
skew_date	o_orderdate, l_shipdate, l_commitdate, l_receiptdate	17 (12)
skew_partname	p_name	24 (21)
skew_container	p_container	8 (6)
skew_size	p_size	7 (6)
skew_shipinstruction	l_shipinstruction	2 (2)
skew_shipmode	l_shipmode	4 (3)
skew_delta_date	l_commitdate - l_receiptdate, l_shipdate - l_commitdate	9 (8)

Table 3.2: Padded encodings used in the TPC-H benchmark. The numbers in parentheses are the code sizes with the simple encoding. With the padded encodings, codes in these columns need 0 – 5 extra bits to represent the column values, as compared to using the simple encoding.

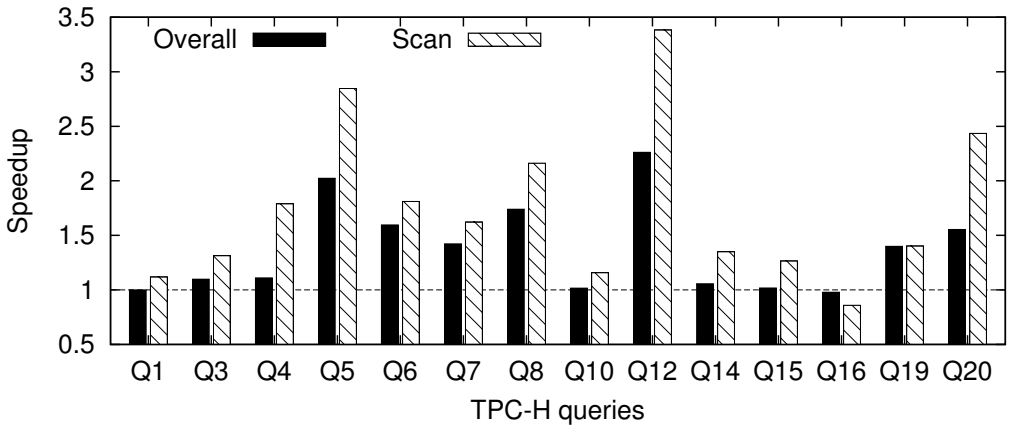
Table 3.2 summarizes the padded encodings that we constructed for the TPC-H dataset. For each encoding, we calculated the distribution of predicate literals across the 22 TPC-H queries. The columns in the same domain use identical padded encoding, thus codes in these columns can be compared directly without additional decoding operations. For example, the "skew_date" encoding shown in Table 3.2 is used to encode all `DATE` columns. The predicate literal distribution for this padded encoding is calculated by merging the distribution of predicate literals on each `DATE` column. In addition, using the method presented in Section 3.5.1, we created two "delta columns." on the expressions `l_commitdate - l_receiptdate` and `l_shipdate - l_commitdate`, as well as the associated padded encodings on these "delta columns". As a baseline method, we use a simple encoding method to encode column values. More specifically, we use null suppression encoding for integer and date columns, and the simple dictionary encoding for string columns. The encoding and loading time for the simple encoding scheme is around 90 minutes. With the AMPE algorithm, this time increases by only 2 minutes to construct the padded encodings. This time includes 5 seconds to construct all the encodings that are listed in Table 3.2. Thus, the extra time in loading the data with the padded encoding scheme is small.

To mitigate the performance overhead associated with fetching codes from padded encoded columns in projection and aggregation operations, we use the padded encoded column as a secondary index only for evaluating scans. In this method, the columns involved in projections or aggregations are fetched from the associated base column representations for these attributes. With this method, 0.9GB of extra space is needed to store all the padded encoding column indices listed in Table 3.2, increasing the size of the underlying database by 10%. We discuss the alternative method, i.e. using padded encoded columns as base column representations, at the end of this section.

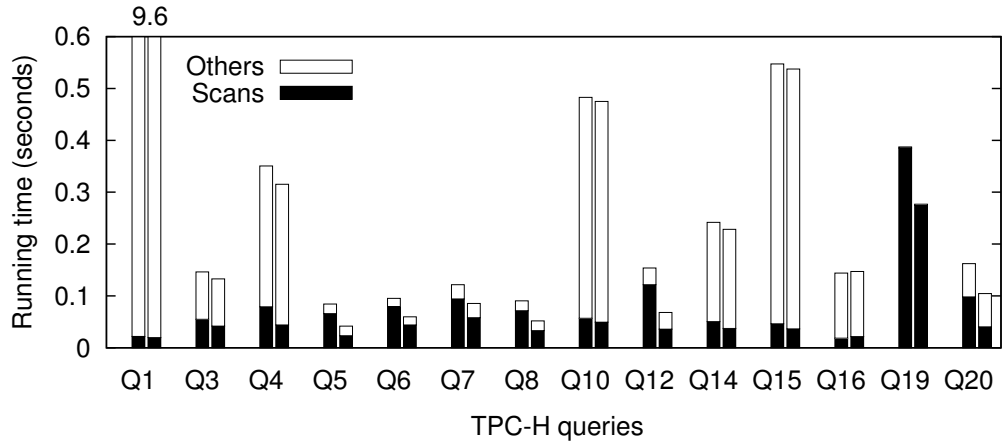
In the evaluation below, we focus on 14 of the 22 TPC-H queries (Q1, Q3 - Q8, Q10, Q12, Q14 - Q16, Q19-Q20), as the other 8 queries in the TPC-H benchmark do not involve the columns listed in Table 3.2. (In other words, the padded encodings have neither a positive nor a negative effect on the performance of these remaining 8 queries.) Following previous work [64, 88], we materialize the join component in these queries, and then ran scan operations on the pre-joined materialized tables so as to stress the scan performance (the focus of this study). All the 14 queries that we focus on, except for Q19, contain a predicate clause that is a conjunction of one to five predicates. Q19 has a more complex predicate clause, which includes a disjunction of three predicate clauses, each of which is a conjunction of six predicates, some of which are performed on the padded encoded columns. These queries contain a variety of predicates, including $<$, $>$, $=$, $<>$, `BETWEEN`, and `IN`. Some queries also involve predicates that perform comparisons between two columns. For most queries, there are complex aggregation operations following the scan operations.

Figure 3.12a plots the speedup of the padded encoding over the simple encoding with the 14 queries in the TPC-H benchmark. (The corresponding graph showing the raw execution time is in Figure 3.12b.) The total execution time of a query includes the time for the scan phase, and the time on other operations, e.g. aggregations and grouping. As the encoding methods have an immediate effect on scans but an insignificant effect on other operations, we show the speedup for scans and other operations in this figure separately.

For scan operations, the speedup of the padded encoding scheme over the simple encoding scheme ranges from 0.9X to 3.4X, with a geometric mean (GM) of 1.6X. The speedup that the padded encodings achieve depends on many query characteristics, including the number of predicates that involve the padded encoded columns, and the frequency of the predicate constants used in these queries. Q12, for instance, contains five scan predicates, four of which are performed on padded encoded columns or padded encoded delta columns. The two predicates on the column `l_shipdate` always choose the most frequent literals, i.e. the first of January of each year, as the predicate constants to filter the table. Consequently, the computation comparing the column codes and the predicate literals can



(a) Speedup of the padded encoding over the simple encoding



(b) Execution time breakdown of the simple encoding scheme (the left bars) and the padded encoding scheme (the right bars)

Figure 3.12: Performance comparisons between the padded encoding and the simple encoding with the TPC-H queries.

be safely pruned by only looking at the first 3-4 of the 17 bits. In addition, the use of padded encoded delta columns speeds up the evaluation of the two predicates that compare two date columns. Consequently, the padded encoding scheme achieves a 3.4X speedup over the simple encoding on Q12. In contrast, Q1 randomly selects a day between 30 and 60 days prior to a certain day as the literal for the predicate on the column `l_shipdate`. As those dates are rarely chosen by other queries, the padded encodings achieve only 1.1X speedup over the simple encoding for Q1. Also notice that Q16 shows a 10% degradation in the scan performance, as the predicate literals in this query are randomly selected from all possible values in the column `p_size`.

Taking other operations into consideration, the speedup of the padded encodings over the simple encoding ranges from 0.98X to 2.3X, with a GM of 1.3X. The speedup reduces

significantly for many queries, e.g. Q3, Q4, Q10, Q14, and Q15, as the scan phase contributes a smaller portion to the total execution time for these queries. More specifically, the scan times for Q3, Q4, Q10, Q14, and Q15 account for 37%, 22%, 12%, 21%, and 8% of the total execution time, respectively. Figure 3.12b shows the execution time breakdown of the 14 queries in the TPC-H benchmark. The left bar for each query in the figure represents the execution time of the scans and other operations with the simple encoding scheme, while the right bar shows the execution time with the padded encoding scheme.

We also evaluated the performance on these queries when the base columns shown in Table 3.2 are stored using padded encoding. With this method, the execution time of the scan phase is generally the same as that when using the padded encoded columns as indices. However, the performance of some aggregation operations that need to access these (padded encoded) columns slows down. For instance, Q7 uses the column `l_shipdate` as a part of the grouping key. With the padded encoding scheme, the size of the codes that are associated with the column `l_shipdate` grows from 12 bits to 17 bits, which increases the time it takes to fetch these column values for the aggregation operation. As a result, the positive effect of padded encoding on the scan operation (62% faster) is offset by the negative effect on the aggregation operations (36% slower), resulting in a reduced net improvement (20%) in the total running time for Q7. Similarly, the performance degradation on Q16 increases from 10% to 25% when using the padded encoded column as the base column representation. The performance for other queries remains largely unchanged from that presented above.

3.7 Related work

3.7.1 Encoding techniques in database systems

Encoding techniques have been extensively used for main memory analytical databases in both the research community [54, 26, 12, 2, 83, 64, 65, 8], and in the industry, e.g. SAP HANA [27, 96], IBM DB2 BLU [79, 61], Vertica [56]. Most systems listed here use the simple encoding techniques, e.g. dictionary encoding and null suppression encoding, and do not take the distribution of data into consideration.

IBM Blink [80] and its commercial successor IBM DB2 BLU [79] employ an encoding technique called frequency compression or frequency partitioning to exploit column value skew. Based on the frequency of data, a column is split into multiple partitions, each of which uses an independent fixed-length encoding. Thus, the partitions containing frequent column values use shorter codes. Similar to padded encoding, this technique can

be viewed as a hybrid between the fixed-length encoding and variable-length encoding. Unlike frequency compression/partitioning, our work is based on a distinct storage format (vertical bit-packing storage), and exploits skew in the distribution of predicate literals in addition to the distribution of column values. Furthermore our methods can be used in addition to their methods to encode each partition (with padding).

Recently, there has been increasing interest in using encoding techniques to speed up query execution, instead of reducing data space usage, i.e. compression. With the use of encoding techniques, the reduced code size introduces an opportunity to operate on multiple small codes in parallel. Many efficient scan methods have been proposed recently (e.g. [64, 105, 80, 47, 96, 95, 20, 29, 74]) to exploit the parallelism available for small codes. A recent work also studied joins over encoded data [61]. We leverage these methods in our work, and note that our work is complementary to this body of work, as we use these methods to scan the encoded columns.

3.7.2 Skew handling in database systems

Data skew is a well-known phenomenon that has been extensively studied in nearly all aspects of data processing systems, e.g. query processing [62, 24, 100, 97, 101, 75], indexing [98, 25], query optimization [93, 66], partitioning [73], data skipping [88, 7], and MapReduce processing [55, 14]. The focus of this work is on exploiting skew in data distribution to construct efficient skew-aware encodings. Unlike most previous work that attempted to reduce the performance degradation caused by data skew, we leverage data skew to speed up query performance.

The padded encoding scheme is also related to the database cracking technique [42, 43, 37], which automatically reorganize data based on incoming queries in column-oriented database, to speed up query execution.

3.7.3 Optimum binary search tree

Several algorithms have been devised to compute an optimum Binary Search Tree (BST) [34, 49, 39, 32]; this is, to find a binary search tree with the minimum expected search cost with given frequencies. A key application for this problem is alphabetic encoding [34]: by assigning 0 to the left branch in the binary tree and 1 for a right branch, we obtain an alphabetic encoding which has the minimum expected code length amongst all variable-length encodings that have both the prefix property and the order-preserving property.

Our padded encoding is inspired by alphabetic encoding, as both encodings require the order-preserving property and the prefix property. However, in contrast to alphabetic

encoding, the padded encoding needs a more complicated cost function (as shown in Section 3.4.2). Thus, existing algorithms for the alphabetic encoding (and optimum BST) problem can not be used to solve the padded encoding problem.

The first solution to the optimum BST problem was given by Gilbert and Moore [34]. This solution is based on the dynamic-programming technique and has a time complexity of $O(n^3)$. We adapted this algorithm to the padded encoding problem and obtained the optimal algorithm (see Section 3.4.3). Knuth improved this solution and reduced the time complexity to $O(n^2)$ with the use of the monotonicity property [49]. This property is not guaranteed to hold for the padded encoding problem, and is used as the first heuristic in our near-optimal solution. Hu and Tucker [39] found a solution in $O(n \log n)$ steps. This solution was further simplified as the Garsia-Wachs algorithm [32]. However, these solutions cannot be adapted to the padded encoding problem directly.

3.8 Concluding remarks

Scans are a crucial primitive in real-time analytic data processing systems, and in this chapter we propose a scheme called padded encoding that leverages skew in the data and predicate literals to improve the performance of scans. To construct a padded encoding, an algorithm (called AMPE) was proposed to efficiently build a near-optimal encoding while reducing the overhead associated with the code size. We have empirically demonstrated the effectiveness of the padded encoding scheme to speed up scan performance.

Chapter 4

WideTable: an accelerator for analytical data processing

There is a unique confluence of technologies with the trend towards read-mostly and append-only databases (popularized by the MapReduce style of processing), the move towards main-memory databases (for speed), and the use of column stores (for speed and flexibility in schema evolution). In this chapter, we design, develop and evaluate a technique called WideTable that leverages these forces to produce a high-performance analytical data processing system.

WideTable uses aggressive denormalization to flatten a database schema into one or more big (wide) tables. Queries on the original database, even complex join queries, now become simple scans on the WideTables. The use of outer joins during the denormalization process is critical to ensure that the WideTable technique produces correct answers. WideTable uses a columnar storage representation with dictionary encoding to control the space overhead that is associated with denormalization. It also uses recently-proposed fast columnar scan techniques that pack multiple codes into a processor word, and evaluate scans on these "packed codes."

While the WideTable technique can be used in various settings, in this chapter we focus on main memory analytical data processing systems. Such main memory settings are an important part of the analytical space, and there is considerable interest in this area (e.g. [16, 15, 41, 80, 48, 99, 106, 27, 79]). This interest has been kindled by the observation that with large main memory configurations, it is often practical to stage at least the most crucial part of the database in memory. The high performance associated with such main memory settings is specially appealing as there is an arms race towards near real-time analytical systems.

We note that WideTable can be used in many different settings, but in this chapter we

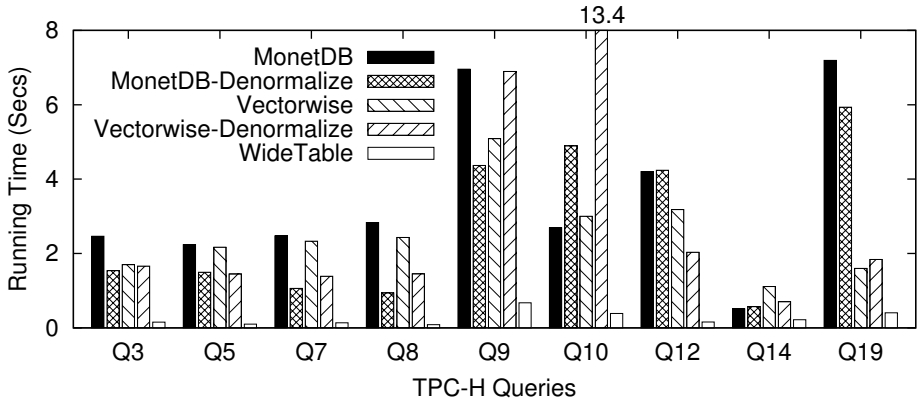


Figure 4.1: Performance comparison with a subset of TPC-H queries.

focus on using it as an front-end accelerator for a traditional data processing platform. Thus, our goal is to focus on performance when evaluating an important and common class of queries in WideTable, and queries that cannot be answered by WideTable are processed in a traditional way.

We have conducted an evaluation of WideTable on the TPC-H benchmark. Figure 4.1 compares the performance of MonetDB, Vectorwise, and our implementation of WideTable. (Section 4.4 presents additional results.) This figure shows two results for both MonetDB and Vectorwise. The first uses MonetDB and Vectorwise as is, and the second denormalizes the TPC-H schema by pre-joining all the tables to form a join table (akin to part of the WideTable design, but using MonetDB and Vectorwise as is). This latter method aims to show how a simple materialization approach works compared to the WideTable technique. As can be seen in this figure, the use of denormalization improves the performance of queries with MonetDB and Vectorwise by up to 3X in some cases. But, there is a far larger improvement when using WideTable, as it uses techniques that go beyond simply pre-joining the underlying tables.

We also note that in Figure 4.1, we only show the results for a small number of TPC-H queries. The queries that are missing in this figure require techniques to rewrite and optimize nested queries against a database of materialized pre-joined tables. Part of our contribution in this chapter is developing these techniques for WideTable.

With the WideTable technique that is proposed in this chapter, we can run 21 of the 22 TPC-H queries. Our evaluation shows that our WideTable implementation outperforms MonetDB (with or without denormalization) for nearly all of these queries. WideTable results in over 10X speedup for about half of the 21 queries. Furthermore, the WideTable technique also shows better scalability when running on a many-core machine.

The remainder of this chapter is organized as follows: Section 4.1 discusses denor-

malization, and Section 4.2 shows our analysis on space and time costs of the WideTable technique. Section 4.3 describes the WideTable design. Section 4.4 presents experimental results. Related work is covered in Section 4.5, and our concluding remarks are in Section 4.6.

4.1 Revisiting denormalization

In this section, we present the basic idea behind the denormalization method, as well as the three key techniques that make the denormalization method used in WideTable practical and efficient, namely: column-stores, dictionary encoding, and packed code scans.

Running Example. Throughout this chapter, we use a running example based on the sample data warehousing schema shown below. In this example, the primary key fields are underlined, and the foreign keys are shown in bold. Figure 4.2 shows the example instances for these tables. In this example, the `Buy` relation is a "fact" table, and the other tables are "dimension" tables.

```
Region(rid, rname)
Nation(nid, nname, rid)
Customer(cid, cname, gender, address, nid)
Product(pid, pname, price, nid)
Buy(cid, pid, amount, status)
```

4.1.1 Denormalization

Relational databases are often normalized to eliminate various types of anomalies associated with duplicating information. The basic idea behind the denormalization method is straightforward: we pre-join all the tables to produce a flat table that retains tuples from the original tables, as well as the relationships between these tuples. Consequently, join queries on the original normalized tables now become simple scans on the denormalized table.

Figure 4.3 shows the denormalized table for the example database shown in Figure 4.2. In this example, we use outer joins on each pair of primary key and foreign key to create the denormalized table. Thus, the denormalization tuple retains one copy of each tuple. For instance, the product "Milk" is not purchased by any customer, but it is still included in the denormalized table, and the corresponding `Customer`, and `Buy` attributes are padded with NULLs.

As illustrated by the example shown in Figure 4.3, the number of attributes in the denormalized table is nearly the sum of the number of attributes in each individual normalized

Customer

cid	cname	gender	address	nid
1	Andy	M	100 Main St.	1
2	Kate	F	20 10th blvd.	2
3	Bob	M	300 5th Ave.	1

Product

pid	pname	quantity	nid
1	Milk	10.00	1
2	Coffee	1.00	1
3	Tea	5.00	3

Nation

nid	nname	rid
1	United States	1
2	Canada	1
3	China	2

Region

rid	rname
1	America
2	Asia

Buy

cid	pid	amount	status
1	2	1	S
2	2	3	F
3	3	2	S
1	2	1	S
2	3	1	S

Figure 4.2: Normalized tables.

cid	cname	gender	address	cnid	cnname	cnrid	cnrname	pid	pname	quantity	pnid	pnname	pnrid	pnrname	amount	status
1	Andy	M	100 Main St.	1	United States	1	America	2	Coffee	1.00	1	United States	1	America	1	S
2	Kate	F	20 10th blvd.	2	Canada	1	America	2	Coffee	1.00	1	United States	1	America	3	F
3	Bob	M	300 5th Ave.	1	United States	1	America	3	Tea	5.00	3	China	2	Asia	2	S
1	Andy	M	100 Main St.	1	United States	1	America	2	Coffee	1.00	1	United States	1	America	1	S
2	Kate	F	20 10th blvd.	2	Canada	1	America	3	Tea	5.00	3	China	2	Asia	1	S
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	1	Milk	10.00	1	United Status	1	America	NULL	NULL

Figure 4.3: Denormalized table.

table (we drop all foreign key fields to avoid duplicating these keys), whereas the number of rows in the denormalized table is nearly equal to the number of rows in the largest original table. In a way, we have produced a wider (fact) table with the denormalization technique. In this chapter, we use the term "WideTable" to metaphorically describe such a denormalized table. As we will see below, the use of outer joins is a critical aspect of the WideTable design.

Queries on the original tables, even complex join queries, now can be executed as simple scan queries on the WideTable. As an example, consider the following query Q1 that finds the names of customers who have purchased products from their own nation:

```
Q1: SELECT cname
    FROM Customer, Buy, Product
    WHERE Customer.cid = Buy.cid
     AND Buy.pid = Product.pid
     AND Customer.nid = Product.nid
```

The execution of this query on the original tables requires two joins across three tables, but only requires a single scan with a single predicate `cnid = pnid` on the corresponding WideTable.

We note that there is a rich legacy of work on denormalization in the area of database research, including work done in the early days of this field (e.g. [53, 67, 91]), and also more recent work (e.g. [22, 82, 86]). Several drawbacks of the denormalization method have been identified and discussed, such as how the duplication of information in the denormalized table takes extra space, how it makes updates more challenging, and how the query performance could potentially suffer since the denormalized data is much larger in size. In this chapter, we argue that many recent technical trends now make it practical to reconsider the idea of denormalization. We present these (three) key techniques below, and empirically evaluate these techniques in Section 4.4.1.3.

4.1.2 Columnar storage

Denormalizing a database might slow down query processing in traditional row-oriented database systems, even for the simplest queries. Although denormalization might simplify query processing by converting join operations into (potentially faster) scan operations, it can slow down the access to each individual tuple. This is because a tuple in the denormalized table is generally much larger than the tuple(s) in the original normalized tables, as there are more attributes in the denormalized table, and large fields/attributes (e.g. strings) might be added to the denormalized table. Consequently, when tuples in the

denormalized format are brought into the processor during query processing, considerable (memory and/or IO) bus bandwidth is wasted in fetching attributes that are not needed for query processing. In other words, in a row-oriented storage format, adding more columns into the table (i.e. denormalizing tables) generally hurts the performance of queries that only access a few columns in the database (which is the common case).

There has been significant interest in column-oriented databases in both the research community (e.g. [2, 87, 16]) and in the commercial products (e.g. [30, 57, 106]). We observe that storing data in columns is well-suited to the WideTable design, because column-oriented databases only access the values of the columns that are required for processing a given query, (largely) regardless of how many columns there are in the underlying table. Consequently, adding more attributes or columns (when using denormalization) is nearly "cost-free" in terms of the query processing cost.

4.1.3 Dictionary encoding

Denormalization methods generally store additional redundant information that consumes extra space, and slows down query processing as more data has to be moved through the memory and/or IO buses. In contrast, in a normalized database, each data item is stored in only one location.

For example, in Figure 4.2, each address for each customer is stored only once in the normalized representation, whereas in the denormalized representation (Figure 4.3), this address information is often duplicated. Imagine that each customer purchases a large number of products, and the address field is a wide string, e.g. a `CHAR(100)` data type, then the space associated with storing this attribute can cause a large increase in the space that is needed to store the denormalized table.

WideTable uses dictionary encoding to address this limitation. Dictionary encoding [20, 94] is a popular method to compress databases, even in main memory settings [80, 27, 54, 18]. The padded encoding techqniue presented in Chapter 3 is also a dictionary-based encoding scheme. Dictionary encoding builds a dictionary on all the distinct values in the column, and maps each native column value to a *code*. In this chapter, we use the term "code" to mean an encoded column value. The data for a column is represented using these codes, and these codes only use as many bits as are needed for the fixed-length encoding.

Figure 4.4 demonstrates the dictionary-encoded denormalized table with the dictionaries on each string or numeric attribute (we omit showing the dictionaries for integer attributes). Each dictionary that is built on the dimension attributes contains up to three values, which is equal to the number of tuples in the dimension tables `Customer`, `Buy`, and

cname dict

value	code
Andy	0
Bob	1
Kate	2

gender dict

value	code
F	0
M	1

address dict

value	code
100 Main St.	0
20 5th Ave.	1
300 10th blvd.	2

nname dict

value	code
Canada	0
China	1
United States	2

rname dict

value	code
America	0
Asia	1

status dict

value	code
F	0
S	1

pname dict

value	code
Coffee	0
Milk	1
Tea	2

quantity dict

value	code
1.00	0
5.00	1
10.00	2

cid	cname	gender	address	cnid	cnname	cnrid	cnrname	pid	pname	quantity	pnid	pnname	pnrid	pnrname	amount	status
1	0	1	0	1	2	1	0	2	0	0	1	2	1	0	1	1
2	2	0	1	2	0	1	0	2	0	0	1	2	1	0	3	0
3	1	1	2	1	2	1	0	3	2	1	3	1	2	1	2	1
1	0	1	0	1	2	1	0	2	0	0	1	2	1	0	1	1
2	2	0	1	2	0	1	0	3	2	1	3	1	2	1	1	1
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	1	1	2	1	2	1	0	NULL	NULL

Figure 4.4: Encoded denormalized table with dictionaries.

`Nation`.

We observe that the dictionary encoding technique is well-suited to the idea of denormalization. Essentially, the dictionaries play a similar role as the dimension table in a normalized database – i.e. it reduces the amount of redundant data. More interestingly, the number of entries in each dictionary is bounded by the cardinality of the corresponding (dimension) table, because the number of distinct values for an attribute is less than or equal to the number of tuples in the (dimension) table.

4.1.4 Packed code scan

There has been a recent flurry of activity/interest in efficient scan primitives for main memory analytic database systems, where the data is often stored in compressed form using dictionary encoding or other encoding schemes. Many recent scan methods exploit the parallelism that is available at the processor word level, such as the BitWeaving technique presented in Chapter 2 and [80, 105, 47, 96, 95, 29]. In this chapter, we call these methods *packed code scans*, as they pack multiple codes into a processor word, and evaluate scans on these "packed codes". In these methods, the scan evaluation is carried out by computing the scan predicates on all the codes in the packed processor word in parallel.

For example, the query Q1 (cf. Section 4.1.1) can be executed with a single predicate `cnid` = `pnid` on the denormalized table. If, as in this example, the attributes `cnid` and `pnid` have only three distinct values (see Figure 4.4), then we can encode each attribute using only 2 bits of space. If the processor word is 64-bits, then there is a 32-way parallelism at the processor word level when performing packed code scans on these columns' attributes.

WideTable is particularly well suited to leverage packed code scans, as WideTable converts complex queries into scan queries. Since denormalization may introduce redundancy, WideTable may require scanning columns over underlying tables that contain more tuples (especially the columns in the dimension table). Thus, efficient packed scan methods are critical for WideTable.

4.2 Cost analysis

In this section, we analytically evaluate the normalized method and the denormalized method, in terms of the space costs (Section 4.2.1) and the time costs (Section 4.2.2). In this section, we use the term "denormalized method" to mean a method that we pre-join all normalized tables to produce a denormalized table that is stored in a column-oriented

format, and use dictionary-based encoding methods to compress the denormalized table (see Section 4.1 for more details).

In the interest of simplicity, and without loss of generality, we assume that the database contains two tables, a fact table R and a dimension table S. Our analysis can be generalized to a complex schema graph where more joins are needed to denormalize the database.

We use $R.r_i$ ($S.s_i$) to denote the i-th attribute in table the R (S). The notation $\bar{r}$ and $\bar{s}$ is used to represent the number of attributes in R and S, respectively.

The denormalized table T contains columns from the fact table, which we call *fact columns* (denoted as $T.r_i$), as well as the columns from the dimension table, which we call *dimension columns* (and denote them as $T.s_i$).

We use the notation $\|X\|$, and $|X|$ to represent the cardinality, and the size of X in terms of the number of bits, respectively.

We assume that all tables are stored in a column-oriented storage format. We also assume that the denormalized table T is produced by performing a primary-key foreign-key join between the tables R and S. Thus, the cardinality of the denormalized table T is equal to that of the fact table R, i.e. $\|T\| = \|R\|$. (This equi-join assumption can be relaxed using the considerations discussed at the end of Section 4.2.1.) Finally, without loss of generality, we assume that only one denormalized table is produced with this schema graph (which only has two tables R and S).

The notations used in this section are summarized in Table 4.1.

Notations	Description
$\|R\|$	The number of rows in the fact table.
$\|S\|$	The number of rows in the dimension table.
$\|R \bowtie S\|$	The number of rows in the join results on R and S.
$\bar{r}$	The number of attributes in the fact table.
$\bar{s}$	The number of attributes in the dimension table.
A_R	The number of fact columns of interest in the selection clause.
A_S	The number of dimension columns of interest in the selection clause.
F_R	selectivity on fact columns.
F_S	selectivity on dimension columns.
C_{scan}	the cost of scanning a single column value.
C_{join}	the cost of joining a row.

Table 4.1: Summary of notations used in Chapter 4.

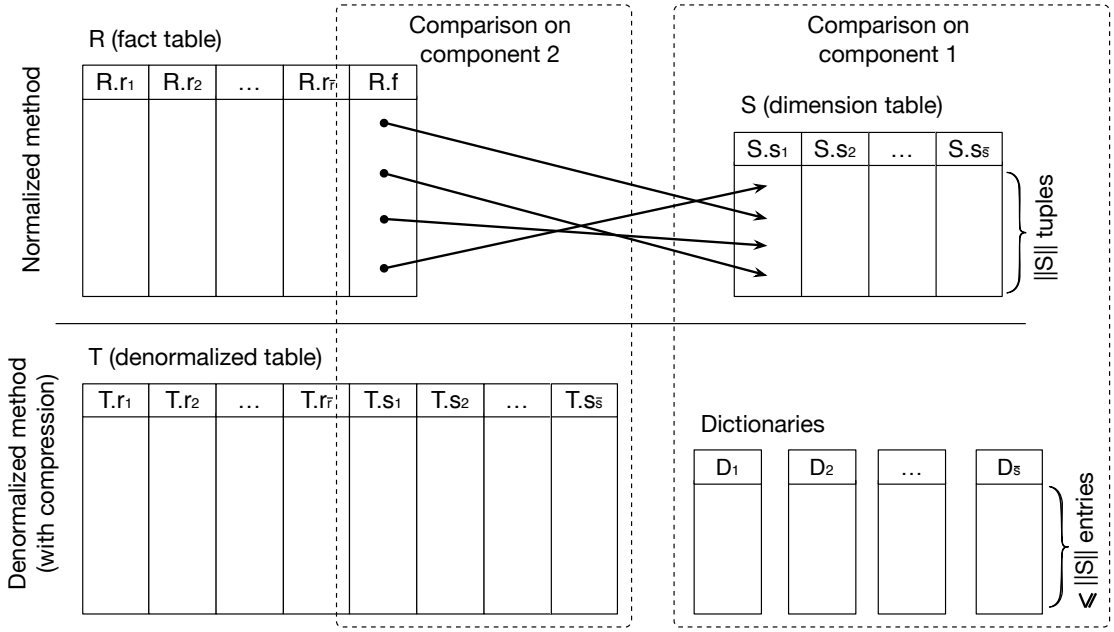


Figure 4.5: Comparisons between the normalized and denormalized methods.

4.2.1 Space cost analysis

In this section, we analyze the additional space that is needed by the denormalized method. Recall that the denormalized table T can be thought of as a "wider" R table, with additional attributes for S that are encoded using the dictionaries, D_i.

Figure 4.5 demonstrates the two methods. We analyze the size of the components that differ in the normalized and the denormalized schemes. These components are related to the representation of the data in the dimension table, and the relationships between the dimension table and the fact table (the dashed boxes with the labels "Comparison on component 1" and "Comparison on component 2" in Figure 4.5). As can be seen in Figure 4.5, the normalized method stores a separate dimension table S and a foreign key column in the fact table, denoted as R.f, to refer to the primary key in the dimension table S. In contrast, the denormalized method adds dimension columns $T.s_i$ to the denormalized table T, along with a set of dictionaries D_i to encode the values in these columns. A column in the fact table, $R.r_i$, except for the foreign key column R.f, is identical to the fact column $T.r_i$ in the denormalized table T, and therefore is not taken into consideration in our space cost analysis. In the analysis below, we separately compare the size of the dimension table S to the total size of dictionaries (the dashed box with the label "Comparison on component 1" in Figure 4.5), and then compare the size of the foreign key column R.f to the total size of the dimension columns in the denormalized table (the dashed box with the label "Comparison on component 2" in Figure 4.5).

First, we show that the total size of the dictionaries, $\sum_{i=1}^{\bar{s}} |D_i|$, is never more than 2X larger than the dimension table size, $|S|$, i.e.

$$\sum_{i=1}^{\bar{s}} |D_i| \leqslant 2 \cdot |S|. \tag{4.1}$$

Proof. Suppose that the size of each value in attribute $S.s_i$ is w_i (in terms of the number of bits), then the size of the dimension table can be represented by $|S| = \|S\| \cdot \sum_{i=1}^{\bar{s}} w_i$. Each dictionary D_i contains two fields, an attribute value and a code. Since these codes only use as many bits as are needed for the fixed-length encoding, the size of a code is $\log \|D_i\|$ bits. As a result, the size of the dictionary D_i can be represented as $|D_i| = \|D_i\| \cdot (w_i + \log \|D_i\|)$. Then, we show that $w_i \geqslant \log \|D_i\|$, because each value in attribute $S.s_i$ has $\|D_i\|$ distinct values, and needs at least $\log \|D_i\|$ bits to represent each distinct value. Thus, we have $|D_i| \leqslant 2 \cdot \|D_i\| \cdot w_i$. We also know that $\|D_i\| \leqslant \|S\|$, because there are up to $\|S\|$ distinct values in any attribute in the dimension table S. Finally, we have: $\sum_{i=1}^{\bar{s}} |D_i| \leqslant \sum_{i=1}^{\bar{s}} (2 \cdot \|D_i\| \cdot w_i) \leqslant \sum_{i=1}^{\bar{s}} (2 \cdot \|S\| \cdot w_i) = 2 \cdot \|S\| \cdot \sum_{i=1}^{\bar{s}} w_i = 2 \cdot |S|$. □

Next, we show that the total size of the dimension columns in the denormalized table, $\sum |T.s_i|$, is never more than $\bar{s}$ times larger than the size of the foreign key column in the fact table, $|R.f|$, where $\bar{s}$ denotes the number of attributes in the dimension table S.

$$\sum_{i=1}^{\bar{s}} |T.s_i| \leqslant \bar{s} \cdot |R.f|. \tag{4.2}$$

Proof. Given the dictionary D_i on the dimension column $T.s_i$, we know that the size of a code in $T.s_i$ is $\log \|D_i\|$ bits. Thus, the total size of the dimension columns, $\sum_{i=1}^{\bar{s}} |T.s_i| = \|T\| \cdot \sum_{i=1}^{\bar{s}} \log \|D_i\|$. We see that the size of a foreign key is certainly larger than $\log \|S\|$ bits, because there are $\|S\|$ tuples in the dimension table. Therefore, we have a bound on the size of the foreign key column in the fact table: $|R.f| \geqslant \|R\| \cdot \log \|S\|$. We also know that $\|D_i\| \leqslant \|S\|$, because there are up to $\|S\|$ distinct values in the attribute $S.s_i$. Finally, we have $\sum_{i=1}^{\bar{s}} |T.s_i| = \|T\| \cdot \sum_{i=1}^{\bar{s}} \log \|D_i\| = \|R\| \cdot \sum_{i=1}^{\bar{s}} \log \|D_i\| \leqslant \|R\| \cdot \bar{s} \cdot \log \|S\| \leqslant \bar{s} \cdot |R.f|$. □

Combining Equation 4.1 and Equation 4.2, we have:

$$\sum_{i=1}^{\bar{s}} (|T.s_i| + |D_i|) \leqslant \max(2, \bar{s}) \cdot (|S| + |R.f|). \tag{4.3}$$

Equation 4.3 shows a theoretical upper bound on how the denormalized method might

take extra space to store the information in the dimension table. This is a key improvement over the standard denormalized method without dictionary encoding, as otherwise the space overhead associated with denormalization could be overwhelming (cf. Section 4.1.3). We also note that in practice, the extra space that is needed by the denormalized method is often only a small portion of the total size of a database, far smaller than the upper bound shown in the equation above. We show these empirical results in Section 4.4.

4.2.2 Time cost analysis for a Select-Project-Join query

In this section we develop a simple model to compare the cost of a simple Select-Project-Join (SPJ) query, as that template is fairly common in analytical settings. (We note that the model that is presented here can be generalized to other types of queries and more complex schema graphs, and that analysis is dependent on the properties of the underlying schema graph and the join graph. The analysis here is meant to simply develop an intuition for the simple schema graph that we consider, and bring out the factors that affect the query performance.)

To evaluate an SPJ query on the normalized database, we first perform scans on both the fact table and the dimension table, applying the selection on columns of interest in the selection clause. Then, assuming that a hash join algorithm is used, a hash table is built on the small table (dimension table), and this table is probed using the scanned fact table tuples.

We use the terms F_R and F_S to denote the selectivity of the predicates on the tables R and S, respectively. We use A_R and A_S to represent the number of columns of interest in this query in the tables R and S, respectively. Let C_{join} and C_{scan} denote the cost of processing a single tuple with a join, and a scan, respectively. In the interest of simplicity, we ignore the cost of materializing result tuples, as the selectivity of selection predicates is often low and the cost of materializing result tuples is often a small portion of the total cost.

Equation 4.4 shows the cost of evaluating an SPJ query on a normalized database. There are four terms in the equation. The first and second terms refer to the cost of performing scans on the columns of the fact table and the dimension table, respectively. The third term represents the cost of building a hash table on the selected dimension tuples. (In the interest of simplicity, we also use the notation C_{join} to represent the cost of inserting a key into the hash table when building the hash table.) Finally, the last term shows the cost of probing the hash table.

$$\|R\| \cdot A_R \cdot C_{scan} + \|S\| \cdot A_S \cdot C_{scan} + \|S\| \cdot F_S \cdot C_{join} + \|R\| \cdot F_R \cdot C_{join} \tag{4.4}$$

Equation 4.5 shows the cost of evaluating an SPJ query on a denormalized database. To evaluate an SPJ query on the denormalized database, we simply scan all the columns of interest. Thus, the cost is the sum of the scanning costs for both the fact and the dimension tables.

$$\|T\| \cdot A_R \cdot C_{scan} + \|T\| \cdot A_S \cdot C_{scan} \tag{4.5}$$

Note, here we assume that the number of rows in the join results is exactly the same as the number of rows in the larger table, i.e. $\|T\| = \|R\|$. (This assumption can be relaxed using the same considerations discussed at the end of Section 4.2.1.) Thus, we can rewrite Equation 4.5 as:

$$\|R\| \cdot A_R \cdot C_{scan} + \|R\| \cdot A_S \cdot C_{scan} \tag{4.6}$$

Now, let us compare the cost of both methods. We first assume that the dimension table is much smaller than the fact table, i.e. $\|R\| \gg \|S\|$. Then, we can remove the terms in Equation 4.4 that are associated with the dimension table. Now, let us examine when the denormalized method is faster than the normalized method:

$$\begin{aligned} &\text{Equation 4.4} > \text{Equation 4.6} \\ \Rightarrow &\|R\| \cdot A_R \cdot C_{scan} + \|R\| \cdot F_R \cdot C_{join} \\ &> \|R\| \cdot A_R \cdot C_{scan} + \|R\| \cdot A_S \cdot C_{scan} \\ \Rightarrow &F_R > A_S \cdot \frac{C_{scan}}{C_{join}} \end{aligned} \tag{4.7}$$

The equation above shows that the denormalized method is faster than the normalized method when the selectivity on the fact table side is greater than $A_S \cdot \frac{C_{scan}}{C_{join}}$. Interestingly, the ratio between the scan cost and the join cost, i.e. $\frac{C_{scan}}{C_{join}}$, is critical for the relative performance between these two methods, and thus the trend towards faster packed code scan methods works in favor of the denormalized design.

Previous studies [13] show that the value of C_{join} for an efficient join implementation is about 50 cycles/tuple. This number is obtained on a join with a dimension table that contains 16 million tuples, and could even increase when the dimension table is larger or is highly skewed. On the other hand, as we shown in Chapter 2, the BitWeaving scan method reduces the value of C_{scan} from 10 cycles/tuple to about 1 cycle/tuple for large codes or even below 0.5 cycles/tuple for small codes. Furthermore, with the use of padded encoding scheme presented in Chapter 3, the values of C_{scan} can be further reduced to 0.2 cycles/tuple for highly skewed datasets. In sum, the use of the BitWeaving method and the padded encoding scheme reduces the threshold on F_R from $0.2 \cdot A_S$ to even below

$0.01 \cdot A_S$, and significantly expands the "sweet spot" of the denormalized method.

4.3 WideTable

WideTable uses the four techniques described above in Section 4.1 using the relationships shown in Figure 4.6. Each arrow in the figure represents a dependency relationship between the techniques. As shown in the figure, to avoid the pitfalls associated with denormalization, we exploit columnar storage to only fetch the columns that are of interest to the query. In addition, the use of dictionary encoding bounds the additional space overhead that is associated with the WideTable design. Finally, using efficient packed code scans improves the speed of the key access method that is used to answer queries on WideTables.

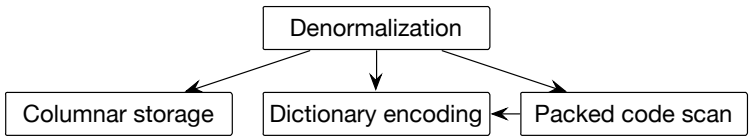


Figure 4.6: Relationships between the techniques used in WideTable.

4.3.1 System architecture

Figure 4.7 shows a representative architecture of how the WideTable technique can be used in a data processing ecosystem. (Note there are other ways to embed WideTable in data processing pipelines, and here we show the approach that we evaluate in this chapter.)

The WideTable component is a module in a larger computational pipeline for the data warehouse. Data from external sources are extracted, transformed, and loaded (using *ETL* gateways) into the *archival* data store, which can use a standard scale-out data platform (e.g. Hadoop or sharded MySQL) to store the data. The data is cooked (prepared) in the *WideTable Baking* module to feed into the main WideTable module.

Inside the WideTable component, the *denormalizer* is responsible for issuing commands to the WideTable Baking component to produce a set of WideTables based on the database schema (cf. Section 4.3.2). The user can also provide hints to denormalize additional WideTables to enhance performance for certain classes of queries (we hope to automate this part in the future). The denormalized table generated by the WideTable Baking component is fed into the *loader*, which parses the input data, and invokes the *encoder* to convert all native values into codes. These codes are stored in a columnar WideTable format in a main memory storage manager.

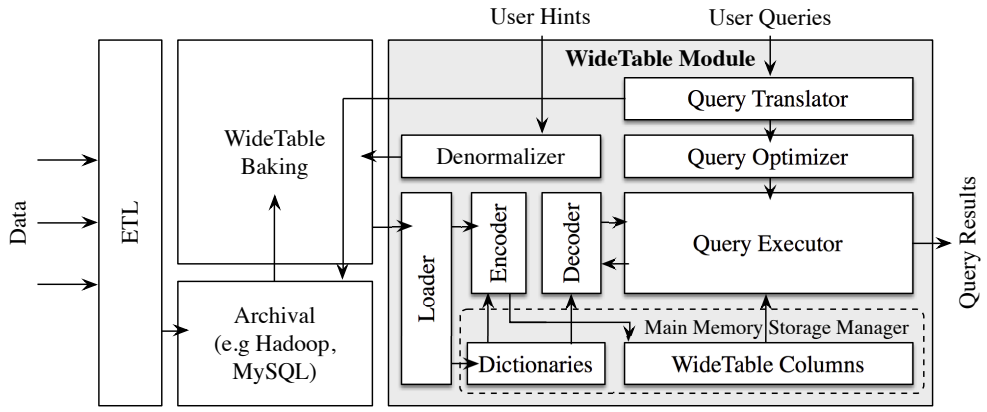


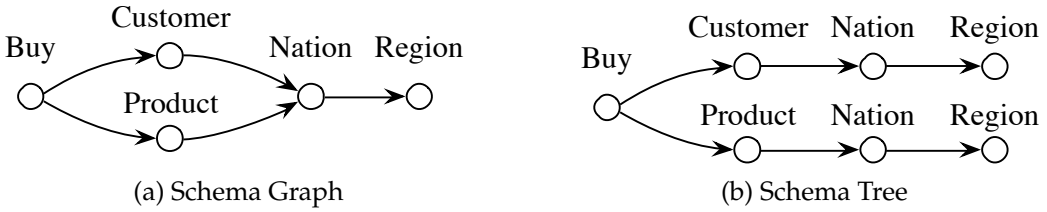
Figure 4.7: WideTable in a Data Processing Framework.

On the query processing side, the *query translator* takes as input user queries, and first checks whether the query can be evaluated with one of the materialized WideTable(s) (cf. Section 4.3.3). If so, then the query translator generates an actual scan-based query plan to execute this query against the selected WideTable. If not, then the query is sent to the archival system, which runs that query and sends the results to the user. Note in this architecture, WideTable is simply used as a potential accelerator to the archival system. Thus, a primary goal of this chapter is to increase the functionality of WideTable to answer as large a class of queries as it can. (As part of future work, we plan to "expand" the footprint of WideTable.)

Notice that the *query optimizer* here is simpler than an optimizer in a standard DBMS, because no join is performed inside the WideTable system. The query optimizer selects an order of scans on a set of columns based on many factors, e.g. the widths of these columns (in terms of the number of bits), the types of predicates, the estimated selectivities, etc. Our current optimizer is fairly simple and simply selects the order based on the widths of the columns. Then, the *query executor* (cf. Section 4.3.4) evaluates the query using a sequence of WideTable operations, that include scans, group-by operations and aggregations. The *decoder* is invoked to convert the code back to native column values when an arithmetic operation is needed, or when the results are returned to the users.

4.3.2 Denormalizing databases

The denormalizer component generates a set of WideTables, called the Set of Materialized WideTables (**SMW**), which includes the WideTables that are automatically produced by the denormalizer based on the database schema, as well as WideTables that are explicitly requested by the users to enhance the performance of certain classes of queries.



(a) Schema Graph (b) Schema Tree

Figure 4.8: Schema graph and schema tree for the example database.

The WideTable denormalizer converts the input database schema into a directed acyclic graph (DAG), called the *schema graph*, with a vertex for each table and a directed edge from the vertex u to a vertex v for each foreign key in table u that points to a primary key in table v[1]. As an example, Figure 4.8a shows the schema graph for the example database (Figure 4.2).

We note that although circular reference of foreign keys is possible in a database schema, it is not common, and in this section we assume that the schema graph is a DAG. Appendix A.4 covers how we handle schema graphs with cycles.

Next, the WideTable denormalizer transforms each component of the schema graph into a hierarchical tree representation, which we call the *schema tree*. For each component in the schema graph, we continue to split vertices of indegree more than one, until all the vertices have at most one incoming edge. To split a vertex v of indegree k ($k > 1$), we replace v by k vertices $v_1 \sim v_k$. The i-th ($1 \leqslant i \leqslant k$) incoming edge (u_i, v) of v is replaced by an edge (u_i, v_i). Finally, each outgoing edge (v, w) of v is replaced by k edges $(v_1, w) \sim (v_k, w)$. Figure 4.8b shows the schema tree for the example schema graph. The `Nation` and `Region` vertices in the schema graph are split in turn to produce the schema tree.

In the description below, we use $T(u)$ to denote the associated table of vertex u in a schema tree. For instance, if s denotes the source vertex of the example schema tree (Figure 4.8b), then $T(s)$ represents the `Buy` table. In addition, for a table R, we use $R.p$ and $R.f(S)$ to denote the primary key and the foreign key referencing the table S, respectively.

The SMW contains the WideTables that are automatically constructed by the denormalizer based on the schema tree(s). To automatically materialize a WideTable, the system performs joins on all nodes/tables in the associated schema tree, using a post-order depth-first traversal algorithm. For each node v that we traverse, if there is an incoming edge (u, v) in the schema tree, we perform a join between $T(u)$ and $T(v)$.

Rather than regular joins, WideTable actually uses outer joins on each pair of primary key and foreign key to produce the denormalized tables. Formally, for each directed edge

[1]For the sake of simplicity, in this discussion, we assume that a primary key or a foreign key is a single attribute.

(u, v) in the schema tree, we perform $T(v) \ltimes T(u)$, where the operator $\ltimes$ represents a full outer join[2]. With these outer joins, a WideTable retains each tuple in the original normalized tables, even if there are no matching tuple on the "other side" of the join. For instance, even though the product "Milk" is not purchased by any customer, it is included in the `BuyWT` WideTable, and missing attributes are marked as NULLs. These NULL-padded tuples are required to answer queries that contain nested subquery blocks, or to answer queries on the table that has been embedded into a WideTable (a more detailed discussion follows in Sections 4.3.3.2 and 4.3.3.3).

For the example database, there is only one connected component in the schema graph, and thus the SMW is the singleton set {`BuyWT`}, where `BuyWT` refers to the WideTable built on the original `Buy` table and is the result of the expression (`Region` $\ltimes$ `Nation` $\ltimes$ `Customer`) $\ltimes$ (`Region` $\ltimes$ `Nation` $\ltimes$ `Product` $\ltimes$ `Buy`). Note that the `Nation` and the `Region` tables are joined twice to produce this WideTable.

Once the denormalization on all the sources is complete, each table in the original database has been denormalized into at least one WideTable, since each vertex in the schema graph is either reachable from a source, or is a source itself.

Some queries may run faster on smaller WideTables (as discussed below in Section 4.3.3.3), and users can make explicit requests to create additional WideTables (we plan to automate this part in the future). Note that dictionaries can be shared by all the WideTables on the same columns, thus adding a new WideTable does not necessarily result in creating new dictionaries.

For example, a user can explicitly request creating a WideTable on the `Customer` table in the example database. Figure 4.9 shows the corresponding `CustomerWT` WideTable that is constructed by the expression `Region` $\ltimes$ `Nation` $\ltimes$ `Customer`. The dictionaries for this WideTable are shared with the `BuyWT` WideTable, and is not shown in this figure.

With this new WideTable, the SMW for the example database is {`BuyWT`, `CustomerWT`}, and we use this SMW instance in the examples below.

cid	cname	gender	address	cnid	cnname	cnrid	cnrname
1	0	1	0	1	2	1	0
2	2	0	1	2	0	1	0
3	1	1	2	1	2	1	0
1	0	1	0	1	2	1	0
2	2	0	1	2	0	1	0

Figure 4.9: WideTable `CustomerWT` (Dictionaries are in Figure 4.4).

[2]In practice, we can use a left/right outer join for certain vertices/nodes, due to the foreign key constraint.

Algorithm 7 Translating an RA expression for single block queries

Input: q': a relational algebra (RA) expression
Output: q: a equivalent RA expression on a WideTable
1: $q := q'$
2: Construct query graph G for q
3: Remove certain FKJ conditions in G and q
4: **if** G is not connected **then**
5: **return** NULL
6: Replace all semi joins by inner joins in q
7: Push inner joins ahead of selections and projections in q
8: Select the smallest WideTable W that covers G
9: Replace the permutation of all inner joins by W in q
10: Remove unnecessary selection conditions in q
11: **return** q.

4.3.3 Query translation

WideTable evaluates queries posed by the users on the denormalized tables by translating the queries into a relational algebraic (RA) expression on an appropriate WideTable in the SMW. The query translator module is responsible for parsing the input SQL query and generating an equivalent RA expression.

The operators of the relational algebra include selection (σ), projection (π), set union ($\cup$), set difference ($-$), inner join ($\bowtie$), semi join ($\ltimes$), (full) outer join ($⟗$), and aggregation (γ).

4.3.3.1 Single block queries

A single block (i.e. with no nested subquery blocks) query is often expressed as a Select-Project-Join (SPJ) query. A representative SPJ query performs scans, followed by joins on multiple tables, followed by a group operation, and finally some aggregate operations.

WideTable supports two variants of joins: regular (inner) joins ($\bowtie$) and semi joins ($\ltimes$). Semi joins are not common in single block queries, but may arise when WideTable flattens nested queries as we will present in Section 4.3.3.2.

WideTable first converts the input SQL query to an RA expression on the original tables using traditional methods [84]. This RA expression is then transformed into an equivalent RA expression on a WideTable using Algorithm 7, as discussed below.

We define a condition of the form $R.f(S) = S.p$ in the RA expression of a query to be a *Foreign Key Join (FKJ) condition* if the attribute $R.f(S)$ is a foreign key that references the primary key attribute $S.p$ in table S. Note that FKJ conditions can explicitly appear in

selection or join conditions, or be implicitly derived by common attribute names in natural joins.

Given an RA expression q, we convert q into a graph, called the *query graph*, with a vertex for each table variable that appears in the query, and an edge from vertex u to vertex v if there is a FKJ condition between the foreign key table $T(u)$ and the primary key table $T(v)$. If a table is involved with k variables in the query, we add k vertices associated with this table in the query graph.

If two directed edges (u_1, v) and (u_2, v) enter the same vertex v in the query graph, then there must exist two FKJ conditions $T(u_1).f(T(v)) = T(v).p$ and $T(u_2).f(T(v)) = T(v).p$. In this case, we arbitrarily replace one of the two FKJ conditions by a non-FKJ condition $T(u_1).f(T(v)) = T(u_2).f(T(v))$, and remove the corresponding edge in the query graph. This step does not change the semantics of the query, but reduces the indegree of the vertex that has more than one incoming edge. By successively applying this step, we guarantee that each vertex in the query graph has at most one incoming edge.

The query can be evaluated with the WideTables in the SMW iff the query graph is connected. If the query graph is disconnected, then certain join components have not been materialized in any WideTables in the SMW, and that query must be sent to the archival data processing system for evaluation. In this chapter, we focus on evaluating queries that are fully covered by WideTable. Part of future work is to develop techniques that allow processing a part of a query using WideTable, and then evaluating the rest of the query in the underlying archival system.

Now, the query graph is connected and has no vertex of indegree more than one, and therefore can be represented as a tree. The remaining steps for transforming q into an equivalent RA expression on a WideTable are as follows.

First, we replace semi join operations ($\ltimes$) in q by inner join operations ($\bowtie$) with a sequence of equivalent transformations. For each semi join $R \ltimes S$ in q, we replace it by $\pi_{R.*}(R \bowtie S)$, applying the relational equivalence E1 shown below. (Note $R.*$ denotes all the attributes in R). This step essentially adds a projection operation for each semi join operation to eliminate duplicate values.

$$R \ltimes S \equiv \pi_{R.*}(R \bowtie S) \tag{E1}$$

Second, we push all the inner join operations ahead of all the selection and the projection operations, as we can commute an inner join operation with selection and projection operations.

Next, we pick the smallest (in terms of cardinality) WideTable in the SMW that "covers"

the query. A query is "covered" by a WideTable iff its query graph is a subgraph of the associated schema tree of this WideTable. Then, we replace the inner join operations in q by outer join operations ($\rtimes$). For the permutation of all the inner joins $R \bowtie \ldots \bowtie S$ in q, we replace it by $\sigma_{R.p \neq NULL \wedge \ldots \wedge S.p \neq NULL}(R \rtimes \ldots \rtimes S)$, applying the relational equivalence E2. The selection operation that is introduced filters out tuples that occur because of the outer joins, e.g. the "milk" row in the denormalized table shown in Figure 4.3. Then, we replace the permutation of all the outer join operations in q by the selected WideTable. (For simplicity, we assume that the source of the selected schema tree is included in the query graph. We relax this assumption in Section 4.3.3.3).

$$R \bowtie \ldots \bowtie S \equiv \sigma_{R.p \neq NULL \wedge \ldots \wedge S.p \neq NULL}(R \rtimes \ldots \rtimes S) \tag{E2}$$

Finally, we optimize the transformed RA expression q by removing unnecessary selection conditions. A selection condition in the form of $R.p \neq NULL$ (introduced by E2) can be removed from q, if the R attributes in the WideTable does not contain NULLs that are introduced by outer joins in the results of q. The algorithm to remove unnecessary selection conditions is as follows: for the source s in the query graph, if there exists no non-FKJ predicate on $T(s)$, and s is not the lowest common ancestor of all vertices whose associated tables are involved in non-FKJ predicates, then we keep a condition of the form $T(s).p \neq NULL$; otherwise, all conditions of the form $R.p \neq NULL$ are removed from q.

Example. Consider the query Q2 below. The RA expression for this query is shown as q_2'.

Q2:
```
SELECT DISTINCT cname
FROM Customer C, Buy B, Product P
   Nation N1, Nation N2, Region R
WHERE C.cid = B.cid AND B.pid = P.pid
 AND C.nid = N1.nid AND N1.rid = R.rid
 AND P.nid = N2.nid AND N2.rid = R.rid
 AND N1.nname ≠ N2.nname
 AND C.gender = 'M'
 AND R.rname = 'America'
```

$$q_2' : \pi_{cname}(\sigma_{N1.nname \neq N2.nname}((\sigma_{rname='America'}(R) \ltimes N1 \ltimes \sigma_{gender='M'}(C) \ltimes B) \ltimes (P \ltimes N2)))$$

$$q_2'' : \pi_{cname}(\sigma_{N1.nname \neq N2.nname \wedge rname='America' \wedge gender='M' \wedge N1.rid=N2.rid}(R \ltimes N1 \ltimes C \ltimes B \ltimes P \ltimes N2))$$

The query graph for Q2 is shown in Figure 4.10. The edge from the `N1` vertex to the `Region` vertex (marked as a dashed arc) is removed as there is another edge entering the `Region` vertex, and a new condition `N1.rid = N2.rid` is therefore added. The selection conditions `Customer. gender = 'M'` and `cnrname = 'America'`, as well as the join conditions `N1. nname` $\neq$ `N2.nname` and `N1.rid = N2.rid` are non-FKJ conditions. We push the selection with these conditions after all the join operations, as shown in q_2''. To run Q2 on the example database, we select the `BuyWT` WideTable, since the schema tree of `CustomerWT` WideTable does not cover the query graph of Q2. Then, we replace the permutation of joins in q_2'' by the `BuyWT` WideTable (by applying E2). Since all vertices, whose associated tables are involved in these non-FKJ conditions (shaded in gray in Figure 4.10), are only reachable from the source, we remove all selection conditions in the form of `R.p` $\neq$ `NULL` that are introduced when converting the inner joins to outer joins (E2). The final transformed RA expression is shown as q_2.

$$q_2 : \pi_{cname}(\sigma_{cnrid=pnrid \wedge cnname \neq pnname \wedge gender='M' \wedge cnrname='America'}(\texttt{BuyWT}))$$

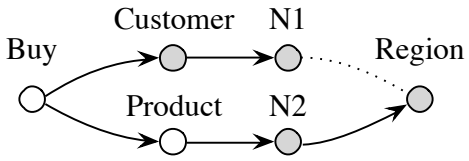


Figure 4.10: Query graph for the example query Q2. The table variables involved in non-FKJ predicates are shaded in gray.

4.3.3.2 Nested queries

Nested queries are fairly common in analytical data warehousing environments, and often contain keywords such as `IN`, `EXISTS`, or set-comparison operators. In this section, we focus

on correlated nested queries, i.e. queries whose inner blocks involve variables that are defined in the outer block (otherwise, a nested query can be simply split into multiple single-block queries, each of which can be separately evaluated using the method present in Section 4.3.3.1).

Nested queries are generally flattened using techniques such as [23, 31, 85]. Amongst these, Dayal's methods [23] are well-suited for WideTable as they use generalized outer joins.

In order to rewrite a nested query for WideTables, we first flatten the query with Dayal's methods, producing a set of equivalent RA expressions for the nested query. In the interest of space, we omit restating Dayal's methods here, as we use it exactly as was proposed in [23], and it behaves like a black box to transform/flatten queries. Then, we enumerate all the produced (flattened) RA expressions, and find the one that can be transformed into an equivalent RA expression on an appropriate WideTable. If such an (RA) expression does not exist, then the query must be sent to the archival data processing system (see Figure 4.7).

With Dayal's method, the nested predicates and correlated predicates in the original query are removed and translated into three variants of joins: inner joins ($\bowtie$), semi joins ($\ltimes$), and asymmetric outer joins ($⟕$). In this chapter, we focus on nested queries that can be converted into a RA expression with inner and semi joins after applying Dayal's methods. To translate such a RA expression, we use the method for single block queries (cf. Section 4.3.3.1). Outer joins are necessary for certain nested queries (with the `COUNT` aggregation function in inner blocks or with the `NOT EXISTS` quantifier). Our method can also be extended to support outer joins, as the denormalized WideTables are produced by outer joins.

However, before we feed each of the produced RA expression q into the method for single block queries, we need to push all join operations ahead of all the aggregation operations. To delay processing an aggregation operation (γ) to after an inner join operation ($\bowtie$), we add to the grouping attributes all the attributes of the table being joined to, as shown by the relational equivalence E3. In practice, only the primary key and the attributes that are referenced in the subsequent operators need to be added into the grouping attributes of the aggregation operation. Semi joins are first converted to inner joins by using relational equivalence E1.

$$\gamma_{\ldots}(R) \bowtie S \equiv \gamma_{\ldots,S.*}(R \bowtie S) \tag{E3}$$

Example. As an example of this approach, consider query Q3. This query finds the

names of the nations that produce products, except for coffee, that are available in quantities that are greater than the amount of this product that has been successfully purchased by male customers. This query has two levels of nested subqueries, the first subquery is nested under the IN keyword, and the second subquery is nested under the arithmetic operator $>$.

Q3:
```
SELECT DISTINCT Nation.nname
FROM Nation N
WHERE N.nid IN (
  SELECT P.nid FROM Product P
  WHERE P.name ≠ 'Coffee'
  AND P.quantity > (
    SELECT SUM(amount)
    FROM Buy B, Customer C
    WHERE B.pid = P.pid AND B.cid = C.cid
      AND B.status = 'S' AND C.gender = 'M'))
```

Applying Dayal's methods on Q3 produces q_3' and a set of other RA expressions. We first push the semi join in q_3' ahead of the aggregation, transforming q_3' to q_3''. Then, we use the method for single block queries to translate q_3''. The query graph of q_3'' (shown in Figure 4.11) is a subgraph of the associated schema tree of BuyWT. Thus, we select BuyWT as the base table to transform q_3''. To convert the two inner joins in q_3'' to outer joins, we add two new selection conditions pid $\neq$ NULL and cid $\neq$ NULL. Both conditions are then removed because there exists a non-FKJ condition (status='S') on the Buy table. After walking through all the steps to rewrite q_3'', the transformed RA expression is shown as q_3.

q_3': $\pi_{nname}(N \ltimes \sigma_{quantity>Sum}(\gamma_{pid,quantity,Sum(amount)}($
$\sigma_{pname\neq'Coffee'}(P) \bowtie \sigma_{status='S'}(B) \bowtie \sigma_{gender='M'}(C))))$

q_3'': $\pi_{nname}(\sigma_{quantity>Sum}(\gamma_{pid,quantity,nname,Sum(amount)}($
$N \bowtie (\sigma_{pname\neq'Coffee'}(P) \bowtie \sigma_{status='S'}(B) \bowtie \sigma_{gender='M'}(C)))))$

q_3''': $\pi_{pnname}(\sigma_{quantity>Sum}(\gamma_{pid,quantity,pnname,Sum(amount)}($
$\sigma_{pname\neq'Coffee'\wedge status='S'\wedge gender='M'}(N \bowtie P \bowtie B \bowtie C))))$

q_3: $\pi_{pnname}(\sigma_{quantity>Sum}(\gamma_{pid,quantity,pnname,Sum(amount)}($
$\sigma_{pname\neq'Coffee'\wedge status='S'\wedge gender='M'}(BuyWT))))$

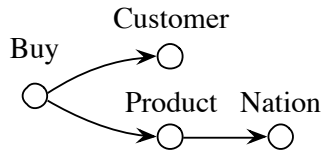


Figure 4.11: Query graph for the example query Q3.

4.3.3.3 Child table queries

If the query graph of a query does not include the root node of the schema tree of the selected WideTable, then this query is called a *child table query*. The key problem with evaluating a child table query on a WideTable is that each tuple of the child table may appear more than once in the WideTable, which may produce incorrect query results. We fix this problem by adding a projection operator in the RA expression to eliminate duplicate tuples.

Translating a child table query follows the method for single table queries with additional handling for the "child table". Given a child table query, we use the method for single block queries (Section 4.3.3.1) to generate an RA expression q. Let q' be a subexpression in q that corresponds to the selection operators on the chosen WideTable. Then, we replace q' by $\pi_{T(u).p,\ldots}(q')$ in q, where u denote the source of the query graph of q, and the triple-dot punctuation represents a list of attributes that are referenced in the scope of q, but outside the scope of q'.

The correctness of this algorithm can be demonstrated with the relational equivalences E4 and E5. Let s be the source of the selected schema tree, and u be the vertex in the schema tree that corresponds to the source of the query graph. Then, we can replace $T(u)$ by $\pi_{T(u).*}(T(u) \ltimes \ldots \ltimes T(s))$ in q, by continuously applying E4 and E5 to include all ancestors of u into q. Thus, we transform the child table query into a regular single block query.

$$R \equiv \pi_{R.*}(R \ltimes S) \tag{E4}$$

$$\pi_{a_1}(R) \equiv \pi_{a_1}(\ldots(\pi_{a_n}(R))) \text{ , where } a_1 \subseteq \ldots \subseteq a_n \tag{E5}$$

Example. The example query Q4 computes the count of products that are available in a quantity greater than 3, for each nation. Notice that of the two WideTables in the SMW (i.e. `BuyWT` and `CustomerWT`), only the `BuyWT` WideTable "covers" the query graph of this query, and thus can be used to answer this query. Since the table `Buy` is not involved in this query, Q4 is treated as a child table query. By applying the method for translating single block queries, Q4 is rewritten as an intermediate RA expression q_4' (note that q_4' is not equivalent

to the original query Q4), which is then further transformed to an RA expression q_4.

Q4:
```
SELECT N.nname, COUNT(*)
FROM Product P, Nation N
WHERE P.nid = N.nid AND P.quantity > 3.00
GROUP BY N.nname
```

$$q_4' : \gamma_{pnname,Count(*)}(\sigma_{quantity>3.00}(\texttt{BuyWT}))$$
$$q_4 : \gamma_{pnname,Count(*)}(\pi_{pid,pnname}(\sigma_{quantity>3.00}(\texttt{BuyWT})))$$

The use of outer joins is critical to obtain correct query results for child table queries. The relational equivalence E4 does not hold if we replace $\rtimes$ by $\bowtie$. In the case of query Q4, the tuple "Milk" in the original `Product` table has a value of 10.00 for the attribute `quantity`, which is greater than the literal (3.00) that is specified in the query. Hence, this tuple should be counted in the result. However, this value would be missed if the underlying WideTable was generated with inner join operations.

We note that if the system notices that many child table queries are being issued, then a new WideTable corresponding to these child tables can be materialized and added to the SMW. This automatic materialization is part of future work.

4.3.4 Query evaluation

With the query translation techniques presented in Section 4.3.3, an original query is translated into a logical query plan (RA expression) on a single WideTable. WideTable then evaluates this query in an efficient scan-based column-wise fashion as follows.

In WideTable, the method for mapping a logical query plan (the tree representation of an RA expression) to a physical execution plan is fairly simple. Given a logical query plan (RA expression), the selection operator (σ) is converted to a subtree in the execution plan tree: a leaf node encapsulates a packed code scan on a single column (attribute); the internal nodes represent logical operation, e.g. AND, OR, NOT, on one or two nodes. Other operators are also mapped to corresponding execution operations in the physical execution tree.

Given a physical execution plan, WideTable performs scans on the selection conditions in the RA expression, using a packed code scan method (See Section 4.1.4). The scan operation iteratively evaluates a predicate on a set of codes that fit into a processor word, resulting in a much higher speed compared to the standard scan method. To perform scans in WideTable, the scan primitive in WideTable first evaluates basic comparisons on each

column, using a packed code scan method. Each packed code scan produces a *result bit vector*, with one bit for each input column value that indicates if the corresponding column value is selected as part of the result set. Conjunctions and disjunctions in the selection condition are implemented as logical AND and OR operations on these result bit vectors. Note that the columns of interest in the query in the WideTable may come from different original tables in the normalized database. However, since we have denormalized them into the WideTable, all the columns have the same cardinality and the same order of tuples, which makes it efficient to merge the bit vectors that are produced on these columns. The output of the scan primitive on WideTable is a single result bit vector.

Once the scans are complete, the result bit vector is converted to a list of record numbers/ids, which is then used to retrieve other columns of interest for this query. Group-by operations and aggregations are performed using standard hash-based aggregation algorithms [35], which build a hash table with entries on the group-by columns, and then use the the hash table entries to store scratch pad variables to incrementally compute the aggregate.

As an example, Figure 4.12 shows the WideTable's physical execution plan for the translated RA expression q_3. To evaluate this query, we continue to perform logical ANDs between the result bit vectors that are produced by packed code scans on each involved column. The results of the scans are then converted to a list of record numbers/ids, and fed into a standard group-by and aggregation pipeline.

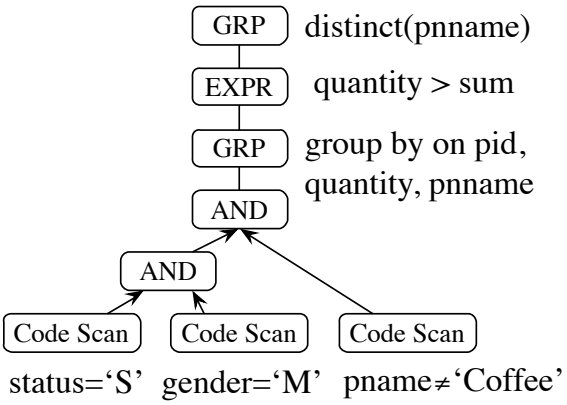


Figure 4.12: Execution plan of the example query Q3.

We note that besides the materialized foreign key joins, certain types of complex components in the original queries are also converted into simple scans on WideTables. Such components include unmaterialized join predicates (e.g. the join predicate `N1.nname` $\neq$ `N2.nname` in Q2) and predicates with nested subblocks (such as `Product. quantity > (...)`

in Q3). In summary, WideTable can process a rich set of queries beyond primary-key foreign-key join queries.

4.3.5 Maintenance of WideTables

In this section, we focus on the maintenance of the WideTables when tuples are added to or deleted from an original table. Our algorithm for maintaining the WideTables targets the scenarios when the original tables are being updated in a batch, which is fairly common in analytical data warehousing environments.

When inserting a set of tuples, I, into an original table R, we do the following for each WideTable in the SMW. Let U be the set of vertices corresponding to R in the schema tree of the selected WideTable. For each vertex $u \in U$, WideTable performs inner joins between I and all the tables whose associated vertices are in the subtree rooted at u. These joins are computed in the WideTable baking component to produce the denormalized tuples of I (missing attributes are padded with NULLs). These denormalized tuples of I are then loaded into the selected WideTable and are encoded using dictionaries or other encoding schemes in the WideTable. If new values are added to a sorted dictionary, then we update the dictionary and may repopulate the corresponding column(s) with the updated dictionary.

A certain class of tuples must be removed from the WideTable due to the insertions of new tuples. Suppose that the table R has a foreign key that references the table S. If there exists a tuple s in S that is not referenced by any tuple in R, then s has been added into the WideTable as an "unjoined" tuple with NULL-padded attributes using outer joins. Nevertheless, if there is a tuple in I that references s, then s should be removed from the WideTable. WideTable runs the following query in the WideTable baking component (see Figure 4.7) to find such tuples, and deletes these tuples from the WideTable.

```
SELECT I.p
FROM I
WHERE NOT EXISTS (
  SELECT * FROM R WHERE I.f(S) = R.f(S) )
```

When deleting a set of tuples, D, from an original table R, we first create an index, K, on the primary keys of all tuples in D. Then, we scan the WideTable as follows. For each tuple in the WideTable, we lookup the values of the attributes, that correspond to the primary key of R, against K. If such attribute values were found in K, then this tuple is either a tuple to be deleted, or a tuple that references a tuple that must be deleted. In either case, this

tuple should be removed from the WideTable.

To deal with "unjoined" tuples in the WideTable, WideTable runs the following query in the WideTable Baking component to find the new "unjoined" tuples, and inserts these new "unjoined" tuples into the WideTable. An efficient implementation of this step is to modify the tuples to be deleted in the WideTable to these "unjoined" tuples directly, by setting the attributes in R to NULLs. More specifically, suppose that s is a tuple in the table S that is not referenced by any tuples in $R - D$, then WideTable rewrites the tuple in the WideTable that references s (i.e. containing an embedded s) into a "unjoined" tuple s by setting the attributes in R to NULLs.

```
SELECT D.p FROM D WHERE NOT EXISTS (
 SELECT * FROM R WHERE D.f(S) = R.f(S)
   AND D.p ≠ R.p )
```

We note that many techniques could be used to improve the maintenance performance of WideTables, e.g. employing indices to quickly find the tuples to be deleted, using "sparse" dictionaries to avoid frequent repopulation of column values, and storing the updated data into a separate "delta" WideTable that is used to evaluate incoming queries in combination with the "primary" WideTable. The investigation of these techniques is part of future work.

4.4 Evaluation

In this section, we present results from an empirical evaluation of the WideTable technique. In the first part of this section, we evaluate the performance of the WideTable technique in the Quickstep database engine with the Star Schema Benchmark. Next, in the second part, we conduct experiments using a standard-alone implementation of the WideTable technique to evaluate the more complex queries from the TPC-H benchmark.

4.4.1 Star Schema Benchmark

In this section, we present results using the Star Schema Benchmark (SSB) [70]. The SSB is a simplified version of the TPC-H benchmark, but is designed to measure performance of database systems in support of classical data warehousing applications. Here we wanted to present end-to-end results that run through the system. The front-end parser and optimizer of current Quickstep engine are not yet ready to run complex workloads like TPC-DS and TPC-H (see Section 4.4.2 for evaluation using the TPC-H benchmark with a standard-alone

implementation of the WideTable technique). We note that the SSB has been used in many previous studies, e.g. [3, 1, 90], and the goal of this experiment is to determine the base performance of our methods on these admittedly simple class of analytic queries.

4.4.1.1 Experimental setups

System Configuration. For our experiments, we use a server with four Intel Xeon E7-4850 CPUs clocked at 2.0 GHz. Each CPU has 10 cores and 20 hyperthreading hardware threads. The machine runs CentOS Linux 7.0 with Linux Kernel 3.10.0. The server has four NUMA nodes, one for each socket, and 64 GB of directly-attached memory per NUMA node. The total memory size is 256 GB for the machine.

Implementation. We implemented and evaluated the WideTable technique in the Quickstep database engine, developed at University of Wisconsin-Madison. The Quickstep engine aims to run data processing at the speed of the underlying hardware.

The Quickstep storage manager is based on a novel block-based architecture [18] that allows a large variety of different physical data organizations to coexist within the same database. Quickstep currently implements both row-store and column-store layouts, which can optionally be used with compression. In addition, the BitWeaving technique (see Chapter 2) has been implemented in Quickstep, and can be used as indices to speed up scans. Consequently, Quickstep offers the flexibility to evaluate the performance of various variants that uses a different set of techniques (that we discussed in Section 4.1). The results of this evaluation are presented in Section 4.4.1.3.

Dataset. For this evaluation, we used SSB datasets at scale factor 10 and 100. The total sizes of the SSB datasets are approximately 6GB and 64GB, respectively.

4.4.1.2 Comparison to MonetDB

In this section, we present results comparing Quickstep (with and without the WideTable technique) to a leading open-source in-memory analytical DBMS – MonetDB (version 11.19, released in October 2014). MonetDB is a full-fledged column-oriented DBMS developed at CWI. It is designed to provide high performance on complex analytics queries, and is optimized for modern multi-core CPUs.

In the evaluation below, we fed each query into MonetDB or Quickstep from the command line, and configured both MonetDB and Quickstep to report execution time, including the time of printing results on screen. Each query in the SSB was run 10 times. We report the average execution time for the 10 runs for each query. Both MonetDB and Quickstep were warmed up before each experiment. To warm up each of the two systems, we ran all

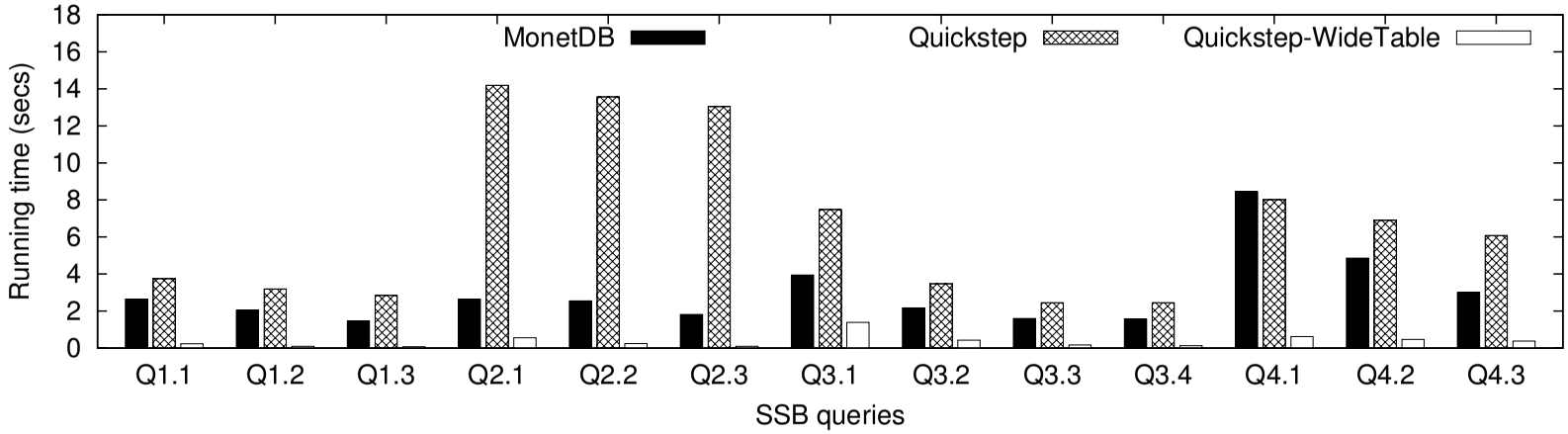
the 13 SSB queries before measuring execution time to make sure all data are loaded into memory. We also pinned the server thread(s) on particular CPU core(s), so that no thread migration occurs during this experiment.

Figure 4.13 and Figure 4.14 show the run times of MonetDB and two Quickstep variants (with and without the WideTable technique) with the 13 queries in the SSB benchmark. In these figures, the tag *Quickstep* refers to the basic settings: we used the columnar storage format for all tables. Below, the tag *Quickstep-WideTable* refers to the method with the WideTable technique: we flattened out the SSB schema, and pre-joined all tables together to form a single denormalized table; we compressed all attributes in the wide table and created BitWeaving indices on all filter attributes. We also conducted experiments on other variants of Quickstep. The experimental results for other Quickstep variants are shown in Section 4.4.1.3.

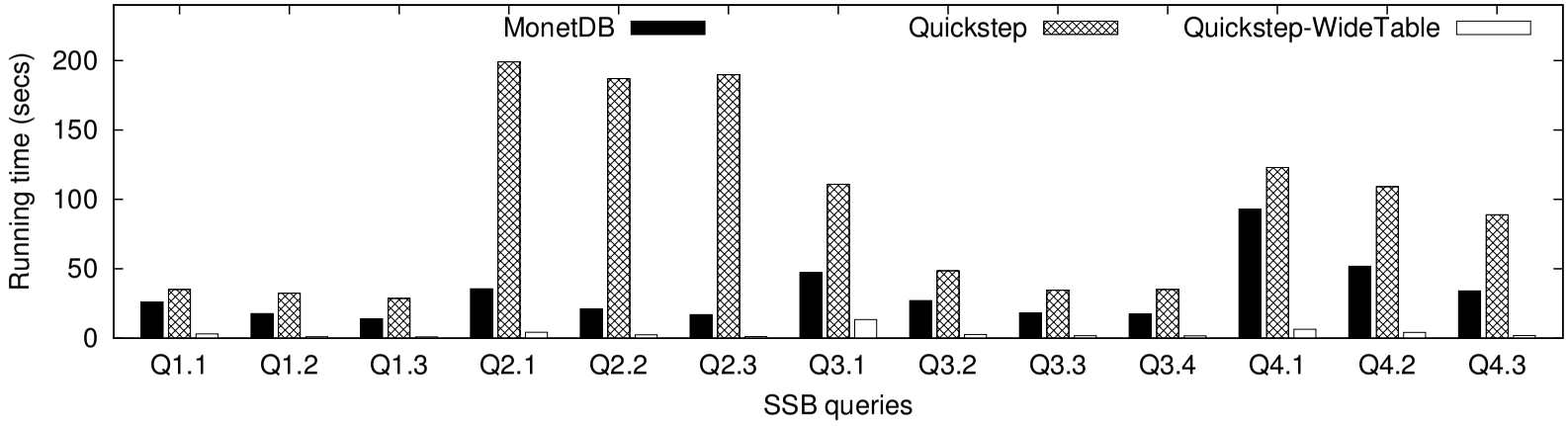
Single thread performance comparison. Figure 4.13 compares the performance of MonetDB and Quickstep with and without the WideTable technique using a single-thread execution, at scale factors 10 and 100. As can be seen from these figures, Quickstep is generally slower than MonetDB on these SSB queries. The performance gap is further increased for the queries Q2.1, Q2.2, and Q2.3, which contain joins with two large dimension tables. This is mainly because the join operator in Quickstep is not yet optimized for the CPU cache performance. As a result, the number of cache misses quickly increases and hinders overall performance.

Nevertheless, with the use of the WideTable technique, Quickstep-WideTable is much faster than Quickstep, and also outperforms MonetDB on all queries, with over 10X in speedup for a majority of the SSB queries. Quickstep-WideTable takes advantage of the denormalization method, which evaluates complex join queries using sequential scans over compressed values with the BitWeaving technique. For a small subset of SSB queries, the speedup of Quickstep-WideTable over MonetDB is relatively reduced. For example, Quickstep-WideTable is only 2.8X faster than MonetDB on the query Q3.1. This is mainly because the group-by operator makes up a large portion of the total run time for these queries. The group-by operator uses a hash table on the group-by attributes, which does not fit in CPU cache and hinders the overall performance for these queries.

Multithreading performance comparison. In the next experiment, we set the number of threads to 40, which is equal to the number of processors (cores). We have also experimented using 80 threads (which is equal to the number of hardware contexts in the system), but we did not see significant performance gain over using 40 threads. Since the execution engine of MonetDB is not NUMA-aware, we did not force Quickstep to allocate memory space on all NUMA nodes. Instead, in order to make a fair comparison with MonetDB, we

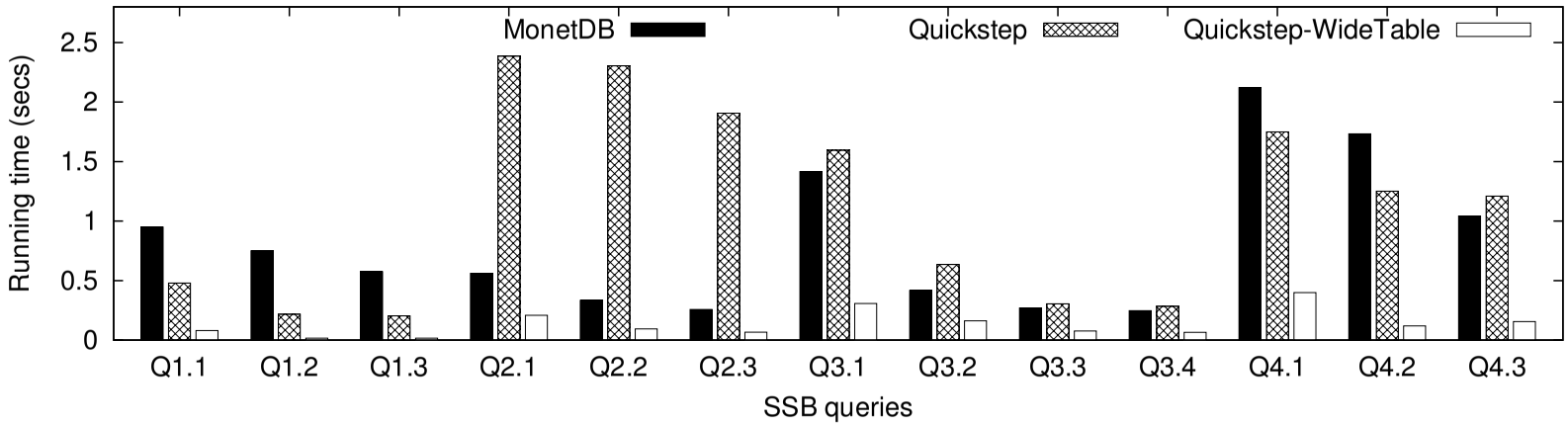


(a) Scale factor = 10

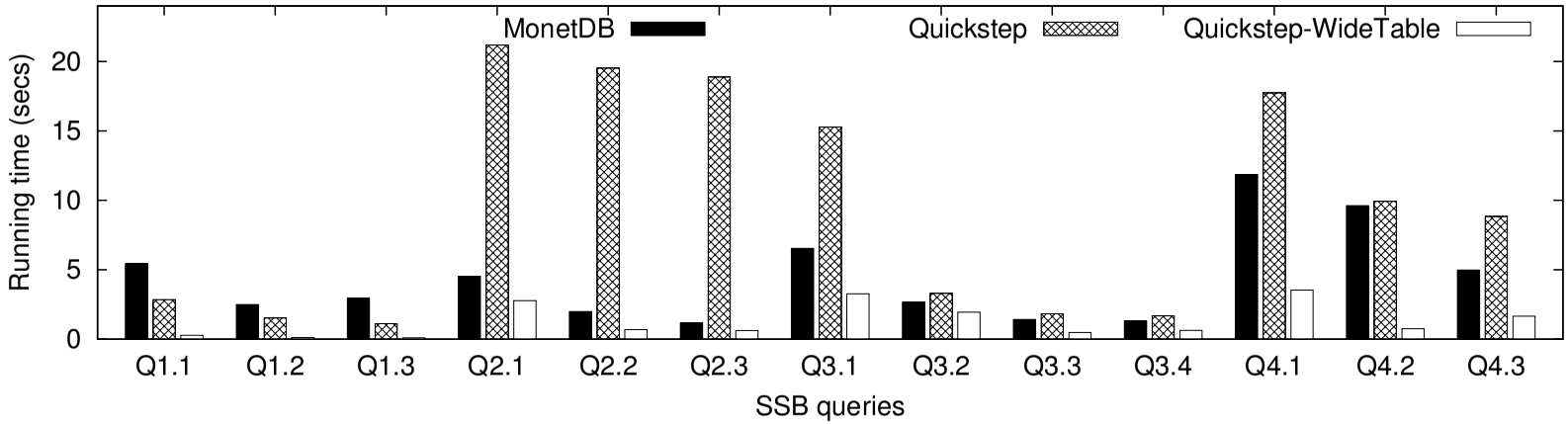


(b) Scale factor = 100

Figure 4.13: Single-thread performance comparison between MonetDB and Quickstep with the SSB benchmark.



(a) Scale factor = 10



(b) Scale factor = 100

Figure 4.14: Multithreading performance comparison between MonetDB and Quickstep with the SSB benchmark.

relied on OS to manage the memory allocation and decide the physical memory location.

Figure 4.14 shows the performance comparison across Quickstep, Quickstep-WideTable, and MonetDB with multithreading, at scale factors 10 and 100. Unlike the single thread execution, Quickstep outperforms MonetDB or shows comparable performance on most all queries, except for Q2.1, Q2.2, and Q2.3. This is mainly because the query execution in Quickstep takes advantage of the block-based design in the storage manager and uses a workflow-based query execution paradigm. The execution model results in a more scalable query processing engine.

Not surprisingly, Quickstep-WideTable also outperforms MonetDB when running with 40 threads, with an average 6.7X and 4.5X speedup across all queries in the SSB benchmark at scale factors 10 and 100, respectively. However, the relative speedup of Quickstep-WideTable over MonetDB generally decreases with multithreading, compared to a single-thread execution. This is because the worker threads in Quickstep-WideTable sequentially access memory at a high speed, and quickly reach the maximum memory bandwidth in the system. Since the OS allocates memory from a single NUMA node, the memory channel between the NUMA socket and the associated memory bank becomes the bottleneck to transfer all data into all cores. The bandwidth of the memory channel in this system is only about 10GB/s in the system, which significantly limits the scalability (to multi-cores) of Quickstep-WideTable. It is expected that Quickstep-WideTable will likely see better scalability by either running on a latest architecture (e.g. the maximum memory bandwidth of the Intel Xeon E7 v3 families is 102GB/s [45]) or allocating memory across all NUMA nodes and fully optimizing even complex queries for NUMA.

4.4.1.3 Effects of various techniques

In this section, we take advantage of the flexible block-based storage manager in Quickstep to produce seven Quickstep variants and empirically analyze the space requirements and the query processing performance of these variants. Based on these results, we empirically demonstrate the effects of a set of techniques we discussed in Section 4.1. In the evaluation below, we focus on four aspects of the Quickstep engine: storage format, compression, scans, and denormalization. To allow for comparison, we use the same setting as that used in the previous experiment (cf. Section 4.4.1.2). In this experiment, we use the SSB benchmark at scale factor 10, because at scale factor 100, some Quickstep variants need more than 300GB data that cannot fit in memory of the machine.

By combining the four aspects (storage format, compression, indexing, and denormalization) together, we configured Quickstep to produce seven Quickstep variants, namely *col*, *col-zip*, *col-bw*, *wide-row*, *wide-col*, *wide-col-zip*, and *wide-col-bw*. Here, the tag *col* and

row refer to column-oriented, and row-oriented storage formats, respectively. The tag *wide* refers to a denormalized method, where we flatten out the SSB schema and pre-join all tables to form an single WideTable, and translate the original queries on normalized tables into equivalent queries on the WideTable. The tag *zip* indicates that we compress all attributes in the storage layout (we only compress all variable-length attributes and all filter attributes for normalized tables). Quickstep automatically chooses the compression method between ordered dictionary compression and truncation based compression (for integers), based on the compression ratio and the query performance. Finally, the tag *bw* means that we compress all filter attributes, and then create BitWeaving indices on all filter attributes. Note that the tag *bw* implicitly indicates the *zip* tag.

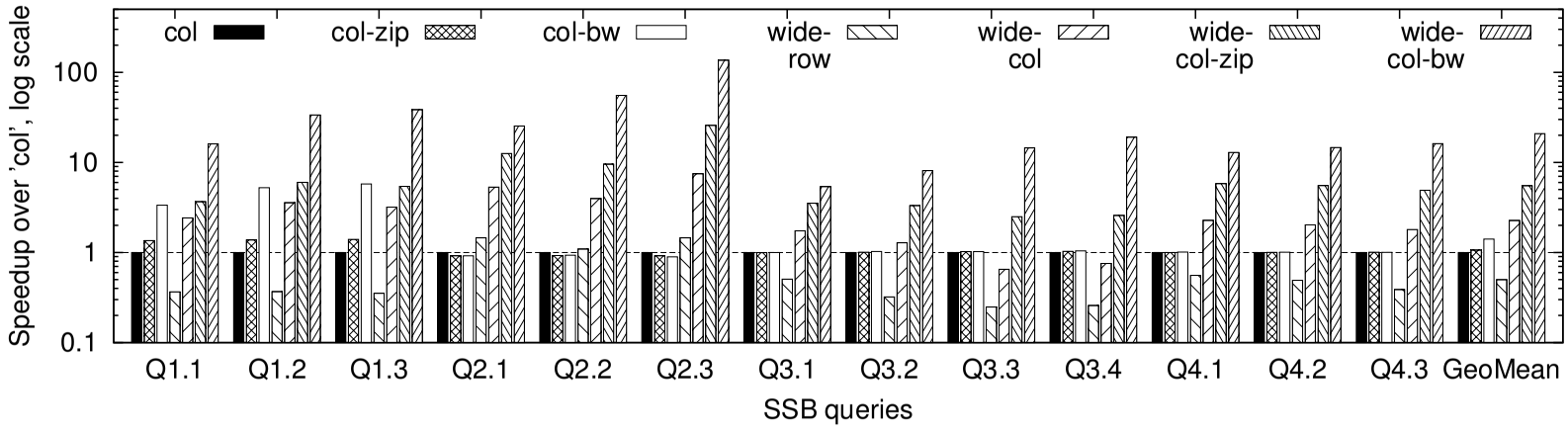
Table 4.2 summarizes the characteristics and the space requirements of the seven Quickstep variants with the SSB benchmark. For scale factor 10, the total size of the raw data set is approximately 6GB. The denormalization technique extensively increases the database size from 4.9GB to 30.2GB (or 29.6GB for the columnar storage format). Compression is very effective in mitigating this, however, reducing the size of the denormalized table to 6.0GB, which takes 40% - 55% more space than the normalized variants. The BitWeaving indices need an additional 0.6GB and 0.1GB of space for denormalized and normalized variants, respectively.

Quickstep variants	storage format	compression	scans	denormalization	space (SF10)
col	column	No	Naive	No	4.9 GB
col-zip	column	Yes	Naive	No	4.5 GB
col-bw	column	Yes	BitWeaving	No	4.6 GB
wide-row	row	No	Naive	Yes	30.2 GB
wide-col	column	No	Naive	Yes	29.6 GB
wide-col-zip	column	Yes	Naive	Yes	7.0 GB
wide-col-bw	column	Yes	BitWeaving	Yes	7.6 GB

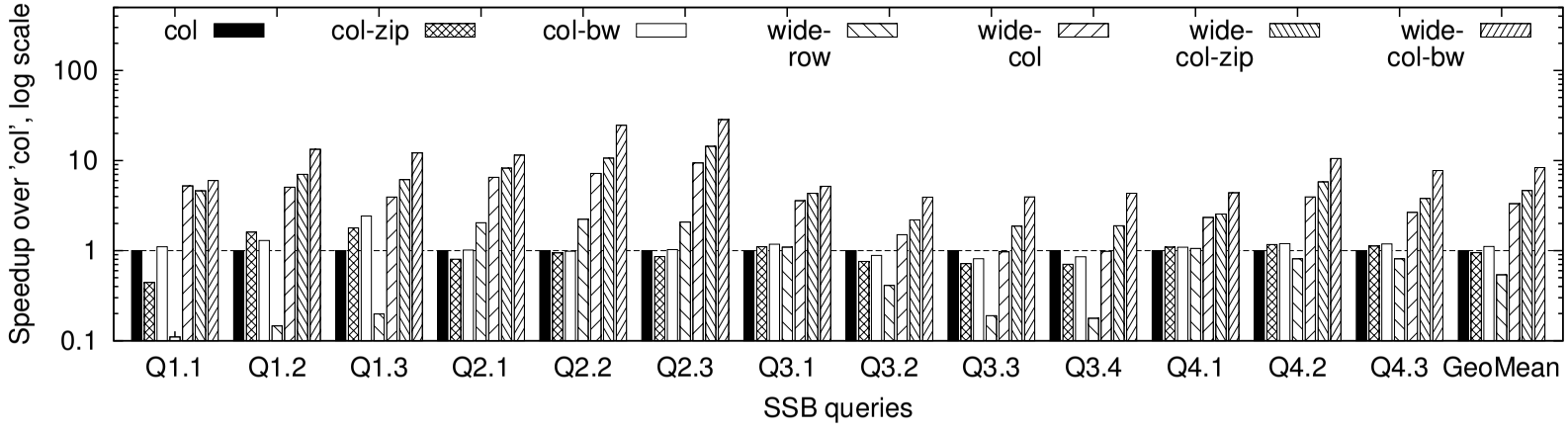
Table 4.2: Characteristics of the seven Quickstep variants.

Figure 4.15a shows the performance of the seven Quickstep variants with the single-thread execution of the 13 queries in the SSB (at scale factor 10). The performance is normalized with respect to that of the *col* variant. (Note that the y-axis has a log scale.)

As can be seen in Figure 4.15a, the compression technique and the BitWeaving technique improve the performance of Quickstep for a subset of queries, i.e. Q1.1, Q1.2, and Q1.3. These queries include joins with the `Date` table that contains only about 2500 tuples. Since the hash table of the `Date` table can entirely fit in CPU cache, the join operator takes only a small portion of the overall execution time, while the scan operators are the bottleneck for



(a) Single-thread execution



(b) Multi-thread execution

Figure 4.15: Performance comparison across various Quickstep valueriants with the SSB benchmark (scale factor = 10) (*col*: column store, *row*: row store, *zip*: compression, *bw*: compression and BitWeaving indices, *wide*: full schema-based denormalization, GeoMean: geometric mean).

these queries. As a result, with the use of the compression and BitWeaving techniques, the *col-zip* and *col-bw* variants significantly speed up the scan operators, and thus outperform the *col* variant on these queries. However, these techniques have negligible performance impact on the other queries, as the join operators are the bottleneck in these other queries.

Surprisingly, we also observe that, with the use of the simple denormalization technique, the *wide-row* variant is actually slower than the *col* variant for nearly all the SSB queries. The reasons is threefold. First, although the *wide-row* variant might simplify query processing by converting join operators into (potentially faster) scan operators, it can slow down the access to each individual tuple. This is because a tuple in the denormalized table is generally much larger than the tuple(s) in the original normalized tables, as there are more attributes in the denormalized table. Consequently, memory bandwidth is wasted in fetching attributes that are not needed for query processing. Second, as the denormalized table is typically much larger than the original tables, the *wide-row* variant requires a larger memory footprint and may reduce the locality and efficiency of the CPU cache. Third, with the use of the denormalization technique, the scan operator may need to access more tuples to evaluate the selection predicates on the original dimension table.

Nevertheless, we show that we overcome these limitations by combining the columnar storage layout (*wide-col*), the compression technique (*wide-col-zip*), and the BitWeaving scans (*wide-col-bw*) with the simple denormalization technique. As shown in Figure 4.15a, by storing all blocks of the demoralized table in a columnar storage format, the *wide-col* variant outperforms the *col* variant for all the SSB queries. These results are consistent with previous studies [3]. The *wide-col-zip* variant compresses all attributes in the denormalized table, achieving an additional 2X speedup over the *wide-col* variant. Finally, with the use of the BitWeaving scans, the *wide-col-bw* variant is up to 10X faster than the *wide-col-zip* variant. Collectively, the fastest variant, *wide-col-bw*, is on average 20.9X and 41.8X faster than the *col* and *wide-row* variants, respectively, across all the 13 SSB queries.

Figure 4.15b illustrates the performance of the seven Quickstep variants when running with 40 threads (scale factor 10). Compared to the single-thread execution (see Figure 4.15a), the speedups of the *wide-col-zip* and *wide-col-bw* are relatively reduced. This is because the scan operators in these two variants access data at a higher speed than other variants, and thus reaches the maximum memory bandwidth of the system. Note that for this experiment, we allocated memory from a single NUMA node. The memory accesses are bottlenecked on the the memory channel between the CPU and the memory bank in the NUMA node.

4.4.1.4 Varying selectivity

In this experiment, we use the tables in the Star Schema Benchmark (at scale factor 10) and two synthetic queries Q1 and Q2 to evaluate the impact of changing the selectivity parameter. The first query (Q1) is based on the query Q1 in the SSB. It accesses two integer attributes in the projection list. In the second query (Q2), we add three more fixed-size string attributes to the projection list. As it is rare to see that string attributes are used as aggregate attributes (only MAX and MIN can be applied to string attributes), we use these attributes as grouping keys.

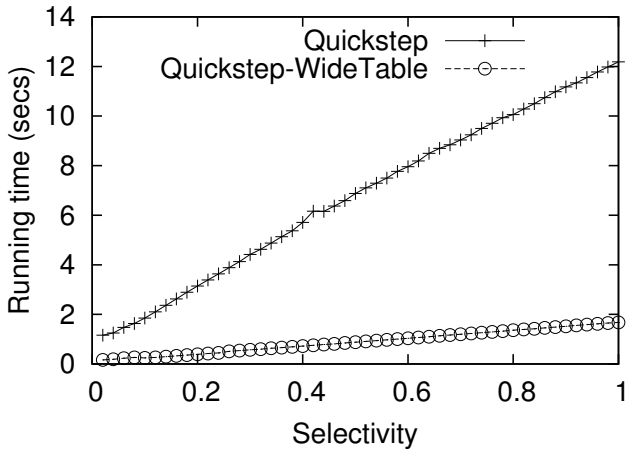
```
Q1: SELECT SUM(lo_extendedprice * lo_discount) AS revenue
    FROM lineorder, date
    WHERE lo_orderdate = d_datekey
     AND d_year between [parameter, parameter]
     AND lo_quantity < [parameter]

Q2: SELECT d_month, lo_orderpriority, lo_shipmode,
     SUM(lo_extendedprice * lo_discount) AS revenue
    FROM lineorder, date
    WHERE lo_orderdate = d_datekey
     AND d_year between [parameter, parameter]
     AND lo_quantity < [parameter]
    GROUP BY d_month, lo_orderpriority, lo_shipmode
```

The parameters in Q1 and Q2 are used to control the selectivity. For each query Qi, we generate three variants, namely Qi-1, Qi-2, and Qi-3, with varying selectivity of the predicates on the fact table (`lineorder`) and the dimension table (`date`). The selectivity of these variants is summarized in Table 4.3.

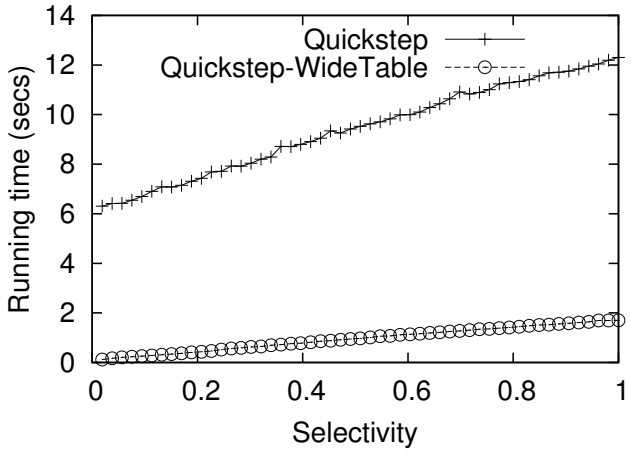
	Q1-1	Q1-2	Q1-3	Q2-1	Q2-2	Q2-3
Fact-table selectivity	2%-100%	100%	10%	2%-100%	100%	10%
Dimension-table selectivity	100%	2%-100%	2%-100%	100%	2%-100%	2%-100%
Overall selectivity	2%-100%	2%-100%	0.2%-10%	2%-100%	2%-100%	0.2%-10%

Table 4.3: Selectivity of different variants of the synthetic queries Q1 and Q2.

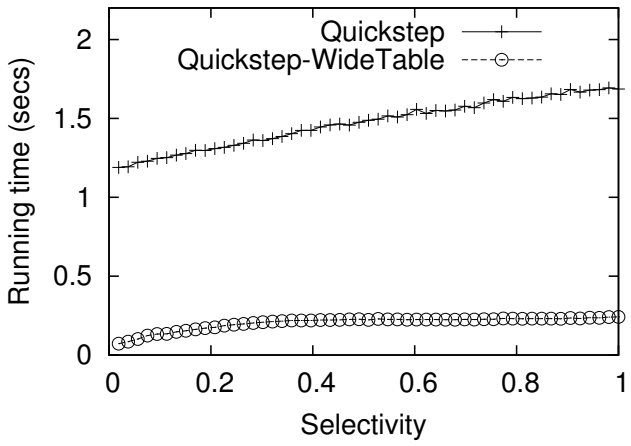
Figure 4.16 shows the performance of Quickstep and Quickstep-WideTable with the query Q1. This query accesses two integer attributes in the aggregation phase. Quickstep



(a) Varying the selectivity on the fact table (Q1-1).



(b) Varying the selectivity on the dimension table (Q1-2).



(c) Varying the selectivity on the dimension table (Q1-3).

Figure 4.16: Execution time of Q1 varying the selectivity.

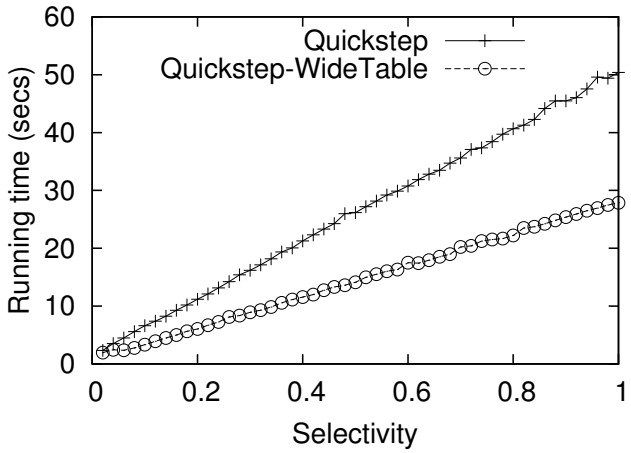
chooses a truncation-based encoding scheme (remove leading zeros) for these integer attributes. Thus, for this query, decoding these attributes in the aggregation phase does not need to lookup dictionaries, and is therefore very efficient. As a result, Quickstep-WideTable significantly outperforms Quickstep for Q1.

In Figure 4.16a, both methods slow down linearly as the selectivity of the predicate on the fact table increases from 2% to 100%. However, the reasons are different for the two methods. For Quickstep, when the predicate selectivity increases from 2% to 100%, it has to lookup more tuples from the fact table against the hash table built on the dimension table. For Quickstep-WideTable, the execution time of the scan phase is nearly constant, but it has to decode more matching tuples and apply the aggregate function on more values.

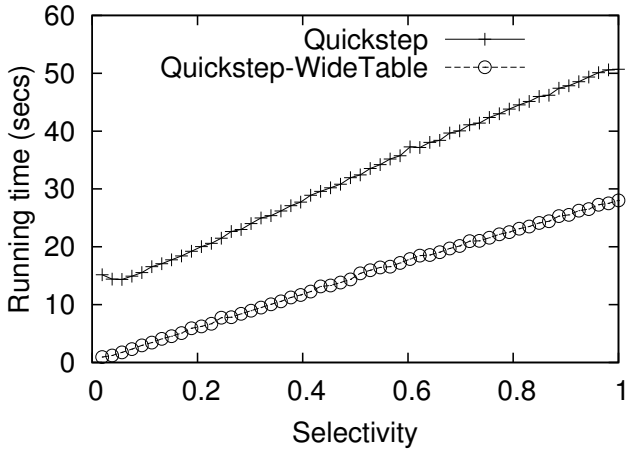
In Figure 4.16b and Figure 4.16c, the performance of both methods also reduces when we increase the selectivity of the predicate on the dimension table. For Quickstep, this is mainly because it requires more time to lookup a larger hash table that is built on the matching tuples from the dimension table. Hence, the execution time increases slowly, compared to Figure 4.16a. For Quickstep-WideTable, the effect of varying the selectivity on the dimension table is similar to that of varying the selectivity on the fact table (as shown in Figure 4.16a). As can be seen from Figure 4.16b, the speedup of Quickstep-WideTable over Quickstep decreases from 50X to 7.2X as the dimension-table selectivity increases. Similarly, as shown in Figure 4.16c, the speedup decreases from 16X to 7.0X as the selectivity increases.

Figure 4.17 show the performance of Quickstep and Quickstep-WideTable with the query Q2. This query requires decoding two integer attributes and three string attributes in the aggregation phase. Decoding string attributes has to lookup dictionaries and is relatively slow. As a result, the speedup of Quickstep-WideTable over Quickstep is generally reduced. However, Quickstep-WideTable is still faster than Quickstep.

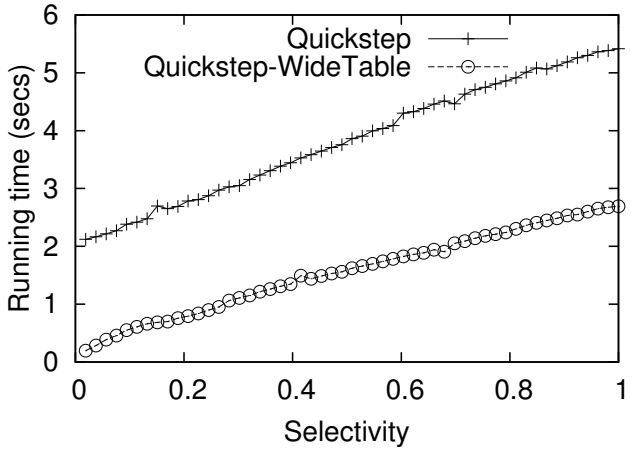
We note that it is possible to speed up the decoding operations on the string attributes for Q2. In Q2, these string attributes are used as grouping keys. In this case, we do not need to decode all these string values for all matching tuples, but can directly use the encoded values as the grouping keys when computing the aggregate value for each group. After computing all aggregate values for all groups, we then decode each distinct value in grouping attributes only once (when we print out the result table). Although it has not been implemented in the Quickstep engine, this method likely performs well when the number of distinct values is low. Note that this method cannot be used on aggregate attributes (e.g. `lo_extendedprice` and `lo_discount` in Q2). However, it is rare to see that string attributes are used as aggregate attributes.



(a) Varying the selectivity on the fact table (Q2-1).



(b) Varying the selectivity on the dimension table (Q2-2).



(c) Varying the selectivity on the dimension table (Q2-3).

Figure 4.17: Execution time of Q2 varying the selectivity.

4.4.2 TPC-H benchmark

In this section, we evaluate the WideTable technique with a standard-alone implementation, as the current Quickstep engine is not yet ready to run the more complex queries in the TPC-H benchmark. Instead of end-to-end comparison results, the comparison with other systems here is to establish a yardstick to better understand the benefits of WideTable on the class of queries that WideTable can handle.

4.4.2.1 Experimental setups

System Configuration. For this evaluation, our server runs 64-bit Linux, and has dual 2.0 GHz Intel Xeon E5-2620 6-core processors, and 32GB of 1600 MHz DDR3 main memory. Each processor has 15MB of L3 cache, which is shared by all the cores on that processor. In addition, each core has 32KB of private L1 instruction cache, 32KB of L1 data cache, and 256KB of L2 cache.

Implementation. We have implemented the WideTable techniques in C++. We choose the BitWeaving technique presented in Chapter 2 as the packed code scan method used in WideTable. The implementation uses regular CPU instructions, and does not use any SIMD optimizations. We have also fairly-standard implementations of the aggregate (with group-by), sorting, top-k, and string matching operations in the query processor component (see Figure 4.7), and we compose the code for each query based on the query plan generated by the methods presented in Section 4.3.3. We note that our implementation supports a limited class of queries, and is less general than MonetDB. The comparison with MonetDB is to establish a yardstick to better understand the benefits of WideTable on the class of queries that WideTable can handle.

Dataset. In the evaluation below, we use queries from the TPC-H benchmark [89]. These experiments were run against a TPC-H dataset at scale factor 10. The total size of the database is approximately 10GB.

For the TPC-H schema, the WideTable technique produces a single WideTable, called `lineitemWT` (`lineitem` ⟗ (`partsupp` ⟗ `part` ⟗ `supplier` ⟗ `nation` ⟗ `region`) ⟗ (`orders` ⟗ `customer` ⟗ `nation` ⟗ `region`)).

In addition, we explicitly created the following three additional WideTables: `partsuppWT` (`partsupp` ⟗ `part` ⟗ `supplier` ⟗ `nation` ⟗ `region`), `ordersWT` (`orders` ⟗ `customer` ⟗ `nation` ⟗ `region`), and `customerWT` (`customer` ⟗ `nation` ⟗ `region`). Without these three additional WideTables, queries on "child" tables (cf. Section 4.3.3.3) are slower by up to 6X (these queries are shown in the last three rows of Table 4.5).

In our experiments, we used MonetDB as the WideTable Baking component to produce

these WideTables, and then requested MonetDB to dump the WideTables as raw text files. This step took 75 minutes. We then loaded these raw text files into the WideTable module (see Figure 4.7). The loading and encoding time was about 90 minutes with one thread. In the future, we plan to speed up this step with multithreading.

Table 4.4 shows the characteristics of these four WideTables[3]. With the encoding techniques in WideTable, the size of the database is reduced from 45.7GB to 8.5GB. Note that the database size in the WideTable format (8.5GB) is even smaller than the size of the raw text files of the original (normalized) tables (~10GB).

Components	WideTable sizes	WideTable cardinality	Raw text sizes	Compression ratios
lineitemWT	5.4 GB	60.5 M	38.8 GB	7.2X
ordersWT	0.7 GB	15.5M	4.4 GB	6.3X
partsuppWT	0.2 GB	8 M	2.2 GB	11.0X
customerWT	0.05 GB	1.5 M	0.3 GB	5.0X
dictionaries	0.8 GB	N/A	N/A	N/A
filter columns	1.3 GB	N/A	N/A	N/A
Total	8.5 GB	N/A	45.7 GB	5.4X

Table 4.4: Sizes of the TPC-H WideTable components. A "filter column" is a special column that is used to evaluate string predicates.

With our WideTable implementation we can run 21 of the 22 queries in the TPC-H benchmark. The characteristics of the 22 queries are summarized in Table 4.5. Two of the 22 queries are simple scan queries. Nested queries are processed using the technique presented in Section 4.3.3.2. There is only one query (Q21) that contains a non-FKJ (foreign key join). The WideTable that only materializes the FKJs does not support Q21. However, there are some extensions of our method to support non-FKJs, which we defer to future work.

4.4.2.2 Single thread performance comparison

In this experiment, we measure the single thread performance by running the TPC-H queries using a single process with a single thread. Each query in the TPC-H benchmark was run 10 times with different query parameters. We report the average execution time for the 10 runs for each query. Both the MonetDB and the WideTable systems were warmed

[3]The compression ratio for each WideTable does not count the dictionary size, since the dictionaries are shared by all the WideTables.

TPC-H queries	Joins	Nested queries	Non-FK joins	WideTable
Q1, Q6				`lineitemWT`
Q3, Q5, Q7-Q10, Q12, Q14, Q19	×			`lineitemWT`
Q4, Q15, Q17, Q18, Q20	×	×		`lineitemWT`
Q21	×	×	×	`lineitemWT`
Q2, Q11, Q16	×	×		`partsuppWT`
Q13	×			`ordersWT`
Q22	×	×		`ordersWT`, `customerWT`

Table 4.5: Characteristics of the queries in the TPC-H benchmark.

up before each experiment. We also pinned the server thread on a particular CPU core, so that no thread migration occurs during this experiment.

Figure 4.18 shows the run times of MonetDB and WideTable with the 21 queries in the TPC-H benchmark. We also label the speedup of WideTable over MonetDB on the top of the bar for each query. As can be seen in the figure, *WideTable outperforms MonetDB on all queries, except for Q18, and with over 10X in speedup for about half of the 21 queries*.

Q18 is the only query for which MonetDB is (slightly) faster than the WideTable approach. This query performs a full table scan on the `Lineitem` table with no filter predicate, followed by a group-by operation on keys in the customer table. As there is no filter predicate (the selectivity is 100%), Q18 suffers from fetching the values in the involved columns in the group-by operation for every tuple in the table. Note that fetching column values from the underlying packed code storage format is often relatively slow [64], which makes the WideTable approach underperform on this query.

Q1 and Q6 are the two simple scan queries in the benchmark. WideTable outperforms MonetDB by 2.0X and 2.3X respectively. Q1 is a simple query that selects 2.4% – 3.6% of the rows in the `Lineitem` table, and calculates aggregate values on nine columns. As MonetDB and WideTable have similar query plans for this query (no join operations), and the aggregate operations accounts for a large portion of the total run time for this query, the performance gap here can be largely attributed to the implementation of the group-by and the aggregate operations. For Q6, WideTable shows a slightly higher speedup over MonetDB than for Q1, mainly because the scan phase contributes a larger portion to the total run time (the packed code scan primitive in WideTable is faster than the scan method used in MonetDB).

For other (join) queries in TPC-H, WideTable takes advantage of the denormalization

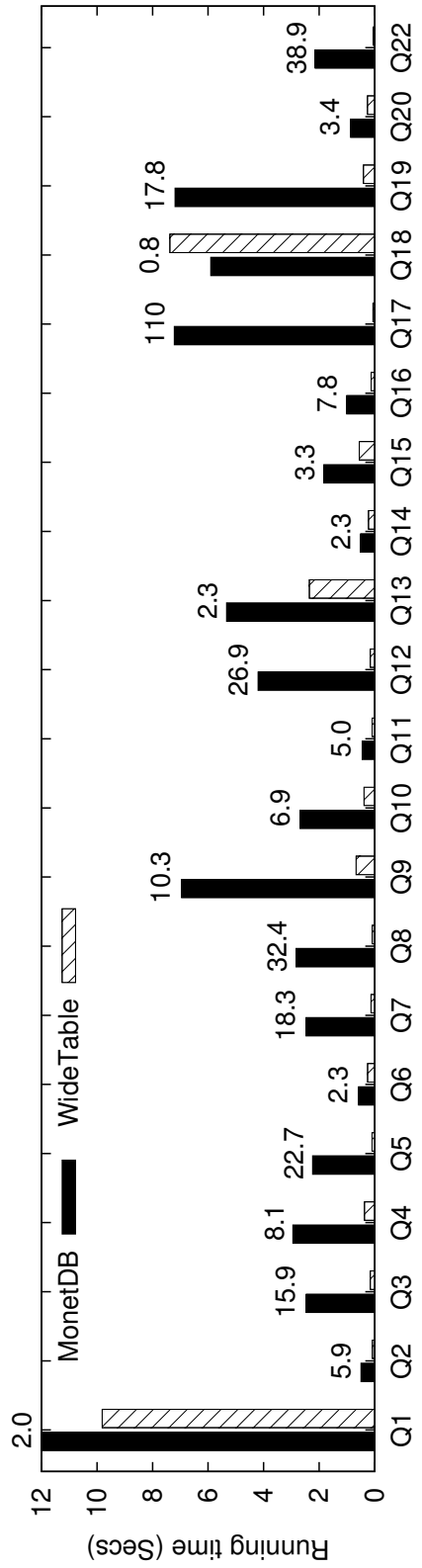


Figure 4.18: Single-thread performance comparison with the TPC-H benchmark. The numbers at the top of each bar are the speedups of WideTable over MonetDB.

method, which evaluates complex queries using sequential scans over packed codes. For most of these join queries, WideTable achieves over 5X speedup over MonetDB.

For the join queries Q11, Q13, Q14, Q15, Q20, the speedups of WideTable over MonetDB are 5.0X, 2.3X, 2.3X, 3.3X, and 3.4X, respectively. This is mainly because the group-by operation makes up a large portion of the total run time for these queries. (See the discussion below for the time breakdown which is shown in Figure 4.22). Some group-by operations in these queries are specified in the original query, whereas some group-by operations are introduced into the query plan when dealing with nested queries (cf. Section 4.3.3.2). These group-by operations use a hash table on the group-by attributes. The group-by attributes often include the primary keys of the original (normalized) tables, e.g. order keys, customer keys, supplier keys, and contain from hundreds of thousands of group entries to tens of millions of group entries. As a result, the group-by hash tables are often larger than the L3 CPU cache. When accessing the group-by hash table, the number of L3 cache misses quickly increases, and hinders the overall performance. In our experiments, we used a simple open addressing-based hash table (with linear probing) as the underlying data structure for the group-by operations. This implementation runs well for small (number of groups) hash tables, but incurs many cache misses for larger hash tables. We plan to investigate a more cache-friendly method to perform group-by operations on larger hash tables as part of future work (building on ideas presented in [102]).

WideTable achieves exceptional speedups (>100X) over MonetDB on Q17. Q17 is a good example that demonstrates the effectiveness of WideTable when evaluating a complex nested query (cf. Section 4.3.3.2). The group-by table created for this query is relatively small (~60K group entries), and fits in the L3 CPU cache (15MB), which makes accesses to this group-by table efficient.

4.4.2.3 Multithreading performance comparison

In this experiment, we use multiple threads. We set the number of threads to 12, which is equal to the number of processors (cores) in the system. We have also experimented using 24 threads (which is equal to the number of hardware contexts in the system), but we did not see significant performance gain over using 12 threads. In the interest of space, we omit these results.

In this experiment, for each of the 21 queries we create 120 different instances of each query. Each query instance uses a different set of (randomly chosen) query parameters; thus, these queries generally access different portions of the database. Now, each thread runs 10 of these queries sequentially, and since there are 12 threads, collectively the system is working on 12 streams of 120 queries for each original query. We do not mix the queries

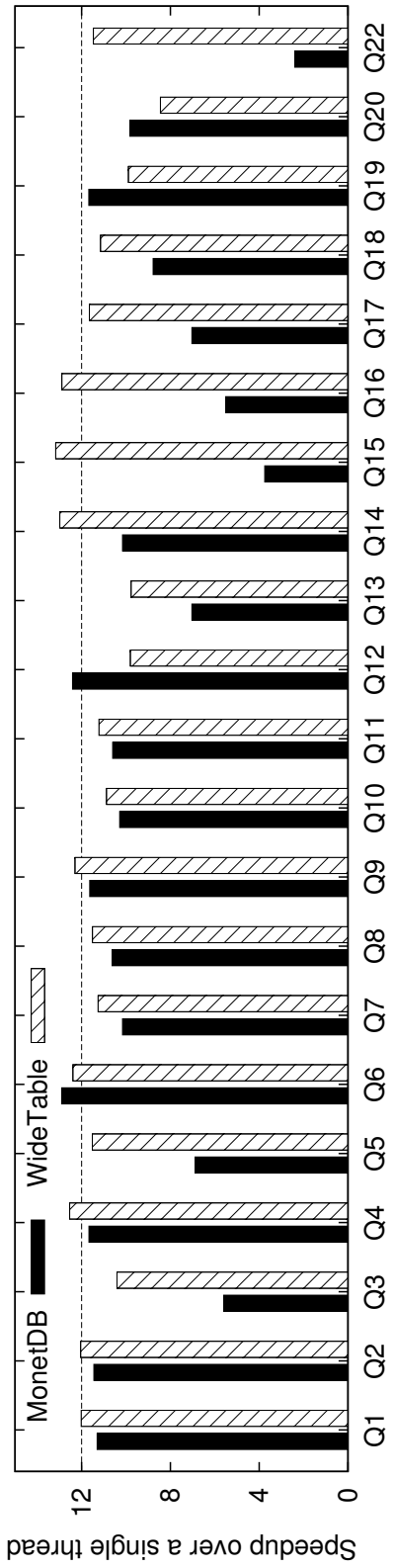


Figure 4.19: Multithreading performance comparison with the TPC-H benchmark (12 threads).

– i.e. this is not the TPC-H throughput test. So, when we show results for a query below, it is the average response time for that query across the 120 different instances. Note both WideTable and MonetDB use the same queries (i.e. they use the same random number seed to generate the queries). The goal of this experiment is to study a specific case of multithreaded performance when the system is running a "homogeneous" workload.

Figure 4.19 shows the speedups of MonetDB and WideTable over the single thread case (discussed in Section 4.4.2.2) for each of the 21 TPC-H queries. We also mark a horizontal line in the figure to indicate the ideal speedup with 12 threads.

The speedup of MonetDB over the single thread case ranges from 2.4X to 12.9X[4], with an average value of 9.3X. We see that for the two scan queries (Q1 and Q6), MonetDB nearly achieves the ideal speedup of 12X. However, for most of the other (join) queries, the speedups are not close to the ideal speedup, mainly because the join implementation needs a relatively large amount of space to store its working set. When there are 12 concurrent threads, the average size of the L3 cache per processor is reduced from 15MB to 2.5MB (six processors share a 15MB L3 cache). The reduced effective cache size per thread quickly increases the number of L3 cache misses, and hinders the overall performance of MonetDB.

The speedup of WideTable over the single thread case ranges from 8.4X to 13.2X, with an average value of 11.3X. For most queries in the TPC-H benchmark, the speedups are close to the ideal speedup of 12X, because the size of the working set for the scan operations and the aggregate operations is small. Consequently, the interference amongst the concurrent threads is insignificant for the scan and the aggregate operations. As a result, WideTable leverages scan operations on the denormalized tables to achieve near linear scalability. However, the speedups of WideTable drops for queries that involve large group-by hash tables. This degradation is more acute when the group-by hash table for one thread fits into the L3 cache, but the total size of all the group-by tables across the 12 concurrent threads exceed the L3 cache size.

In Figure 4.20, we plot the speedups of MonetDB and WideTable over the single threaded case, by varying the number of concurrent threads from 1 to 12. Each thread issues all 22 queries in the TPC-H benchmark, but in a random order (so this test is closer to the TPC-H throughput case). MonetDB is free to use the threads for either intra-query parallelism or inter-query parallelism, but our WideTable implementation is simpler, and only uses one thread per query. The speedup over the single thread case is calculated in terms of the total run time for all the 22 queries. As a result, the reported scalability is largely affected and dominated by the scalability of the long-running queries.

[4]The speedup exceeds the ideal speedup in some cases because of data sharing in the CPU caches.

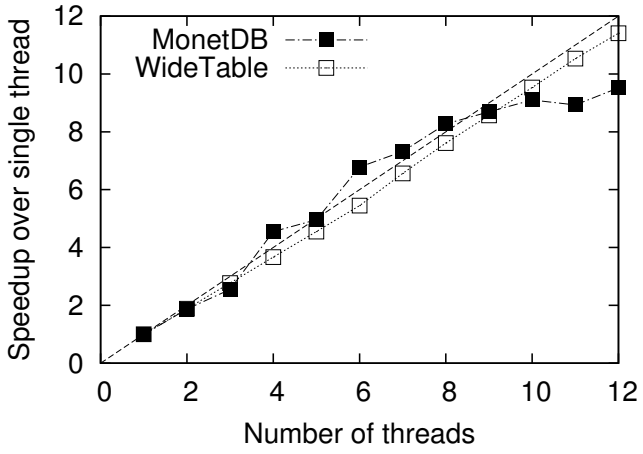


Figure 4.20: Scalability of multithreading with all the 22 TPC-H queries.

As shown in Figure 4.20, WideTable demonstrates a linear speedup as the number of threads increases. When running with 12 threads (there are 12 cores in this machine), WideTable achieves 11.4X speedup over the single thread case. MonetDB shows linear speedup as long as the number of threads does not exceed 8. The gap between the measured speedup and the ideal linear speedup increases when the number of threads is more than 8. In our experiments, the maximum speedup achieved by MonetDB is 9.5X.

4.4.2.4 Update performance

Figure 4.21 shows the performance comparison of MonetDB and WideTable with two refresh functions (update queries) in the TPC-H benchmark. As per the TPC-H specification, the first refresh function, RF1, inserts 15,000 tuples into the `Orders` table, and around 60,000 tuples into the `Lineitem` table. The other refresh function, RF2, deletes the same number of tuples from the `Orders` and the `Lineitem` tables.

Not surprisingly, WideTable is 2.0X slower than MonetDB for the insert query (RF1), as WideTable must denormalize the newly added tuples, and encode all the attribute values using dictionaries or other encoding schemes.

For the delete query (RF2), MonetDB does not complete this query within ten minutes, as it suffers from repeatedly scanning the table to find the tuples to be deleted. WideTable deletes tuples in a batch, creating an index on the primary keys of the tuples to be deleted, and scanning the primary key column(s) just once to delete all the tuples. If a similar technique was applied to MonetDB, we expect to see comparable delete performance for MonetDB and WideTable.

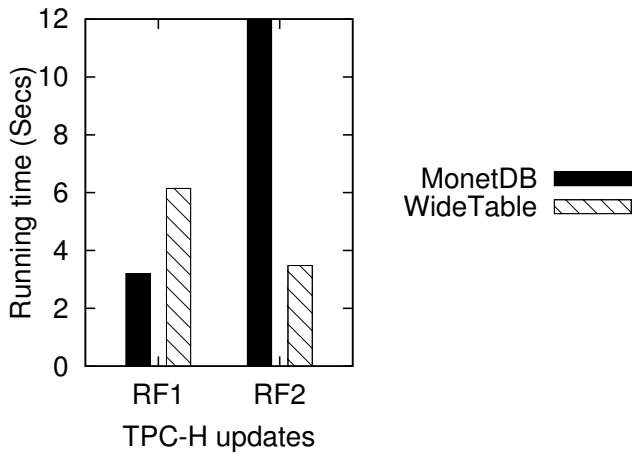


Figure 4.21: Performance comparison with TPC-H updates.

4.4.2.5 WideTable time breakdown

To better understand the performance characteristics of WideTable, in this last experiment, we examine the detailed time breakdown for the key operations when running WideTable with the TPC-H benchmark. (We also compare the time breakdown with MonetDB in Appendix A.5.)

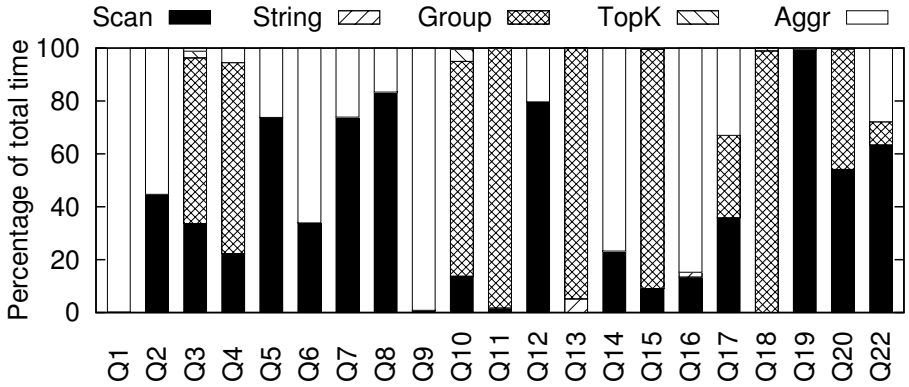


Figure 4.22: Time breakdown of WideTable with the TPC-H queries.

Figure 4.22 shows the time breakdown for the different operations. The five key operations that are shown here include: (a) *scan* operation to scan columns and generate a bit-vector to indicate matching tuples for a set of filter predicates, (b) *string matching* operation to perform a scan with complex string filter predicates, (c) *group-by* (labeled Group) to fetch column values from the matching tuples, and cluster column values based on the group keys, (d) *top-k* operation to select the top-k entries from the group-by tables or the WideTables, and (e) *aggregation* (labeled Aggr) operation to fetch column values from

the matching tuples, and calculate aggregate values (except of group-by) on these column values.

As can be seen in Figure 4.22, the string and top-k operations are the fastest operations – their cost is below 5% of the total query execution time. The other three operations, scan, aggregation, and group-by operations are all significant across all the queries in the benchmark. On average, these three operations account for 38.3%, 29.2%, and 31.7%, of the total run time, respectively.

4.5 Related work

Recently, there has been significant interest in main memory analytical databases in both the research community (e.g. MonetDB [16, 15, 41], Blink [80], Hyper [48], Shark [99]) and in the industry (e.g. Vectorwise [106], SAP HANA [27], and IBM DB2 BLU [79]). The focus of our work is to explore the denormalization technique in this popular main memory settings.

Our work is inspired by the rich body of work in the use of denormalization. In contrast to previous work on denormalization (such as [53, 67, 91, 22, 21, 82, 86]), a) we use outer joins instead of inner joins to denormalize a database schema; b) we explore the idea of denormalization at the physical schema level rather than at the logical level; and c) we focus on various techniques that can be realized in practice to avoid some of the pitfalls that are associated with denormalization.

Dictionary encoding is used to compress data in both row-oriented database systems [20, 94] and column-oriented database systems [2], and has also used in main memory analytical databases [80, 27, 54]. In our work, we directly leverage this body of work. Other compression schemes have also been well investigated in column-oriented database systems [2].

Many packed scan methods have been proposed recently (e.g. the BitWeaving technique presented in Chapter 2 and [105, 80, 47, 96, 95, 29]). We leverage these methods in our work, and note that our work here is complementary to that line of research as the WideTable design simply uses these methods to scan the underlying (wide) columns.

The idea of materialized views, caching and updating is another popular area of research with a rich history (e.g. [19, 36, 4, 6]). View matching and maintenance for outer-join views has also been well studied [59, 60]. In addition, there are more recent proposals such as the recycling technique in MonetDB [46] that is related to the WideTable design. These ideas can be used to further improve the use of WideTables and to automatically figure out the WideTables to materialize.

Blink [80] is an analytical engine that targets consistent response times to ad hoc queries. The original incarnation of Blink [80] considered the idea of denormalization with compression and a row-wise packed code scan method [47], but it evaluated data largely in a row-wise manner. The later version (called IBM DB2 BLU [79]) uses columnar storage, but abandoned the denormalization strategy due to the redundancy introduced by denormalization [11].

4.6 Concluding remarks

This chapter proposed WideTable, a new method for denormalizing data warehousing schemas that converts even complex queries on the original schema into scans on WideTables. Scans are far simpler to execute (e.g. they have predictable access patterns so that its easy for the software to explicitly issue pre-fetch calls, or for the hardware to do so implicitly), making them crucial to high-performance analytical systems. We have empirically demonstrated the performance and scalability (to multi-cores) aspects of WideTable in a main memory setting, and shown that it presents a promising approach for fast analytical query processing.

Chapter 5

Conclusions and future work

This dissertation makes three contributions to the area of analytical data query processing systems.

First, in Chapter 2, we presented the BitWeaving scan method that addresses a critical need for fast scan primitives by exploiting the parallelism available at the bit level in modern processors. The two flavors of BitWeaving make different trade-offs between two common access patterns in column-oriented database systems, and thus can be used as a base storage organization format in column-oriented data stores, or as indices to speedup scan operations. The BitWeaving method is optimized for fully utilizing both the instruction bandwidth and the memory bandwidth available on modern processors, and consequently deliver bare-metal performance for scan operations.

Second, in Chapter 3, we showed that encoding schemes, usually a problem for compression, can be used to improve the performance of scans over the encoded data by leveraging skew in the data and queries. To address this opportunity, we proposed the padded encoding, a dictionary-based encoding scheme, and corresponding algorithms to construct a theoretically optimal or near-optimal padded encoding (with respect to the performance of scans over the encoded data). When the padded encoding scheme is used with the BitWeaving scan technique, we analytically and empirically demonstrated its effectiveness to speed up scan performance.

Finally and most importantly, in Chapter 4, we demonstrated that the idea of denormalization becomes a promising approach for fast analytical data processing. The well-known pitfalls associated with denormalization become negligible or manageable due to several technical trends including read-mostly append-only workloads, columnar storage, and dictionary encoding. With the denormalization technique, most complex join queries on the original database can be converted to simple scans on the flattened table. Thus, the fast BitWeaving scan technique (introduced in Chapter 2) significantly improves the

speed of the core operation that is used to answer queries on the denormalized tables. We empirically evaluated our methods in a main memory setting using the TPC-H and the star-schema benchmarks, and demonstrated the effectiveness of our methods, both in terms of raw query performance and scalability when running on many-core machines.

5.1 Future work

The first interesting direction for future work is to extend the techniques presented in this dissertation to other emerging computer architectures, e.g. tightly-integrated accelerators (such as GPUs, and Intel Xeon Phi) and 3D die-stacking. These emerging architectures provide tremendous hardware resources for analytic database workloads. For instance, GPUs employ very wide data-parallel hardware that naturally leverages the BitWeaving scan method. The 3D die-stacking technique, on the other hand, enables higher memory bandwidth and increased compute capability. Consequently, these architectures are likely to play a big role in future analytic database systems. Using the simple scan primitive as an example, there have been some initial work [77, 78] in exploring this direction.

The second interesting direction for future work is to expand the WideTable method to distributed environments. It is well known that distributed joins are generally difficult to scale out, as it requires transferring a large amount of data across nodes in a cluster. In such an environment, the denormalization-based WideTable method likely becomes more appealing. This is primarily because that the WideTable method converts joins to scans, which are naturally parallelizable. Thus, a WideTable can be broken down into multiple partitions, each of which is separately stored in a node, and can be evaluated independently. Furthermore, the total amount of main memory in a cluster provides extra space to store the denormalized data, which often consumes more space than the original, normalized data.

Furthermore, there are many interesting open questions in expanding the BitWeaving method to evaluate queries over more complex encoded structured data. An example of such encoding method is Protocol Buffers that was originally developed at Google, and subsequently has been widely used both inside and outside Google to storing and exchange structured information. Protocol Buffers are serialized into a binary format which often uses a few bits to represent each field. The compact format of Protocol Buffers provides an opportunity to use BitWeaving-like techniques to evaluate a query over Protocol Buffers without decoding each field of the Protocol Buffers. Thus, the proposed method could significantly reduce the CPU costs associated with decoding Protocol Buffers. The new Bit Manipulation Instructions (BMI) sets in the latest Intel and AMD CPUs provide a powerful

set of primitives to develop such techniques.

Finally, there are promising areas for future research in studying how the WideTable technique could change the way in which one builds cost-effective high-performance analytical appliances. The traditional wisdom in this area is to build servers with large amounts of main memory and fit the entire database in main memory. Despite the continual drop in DRAM prices and increasing memory densities, it is still uneconomical to use this recipe in many cases. Since the WideTable method only relies on fast scans, the performance of the WideTable method is primarily determined by the sequential bandwidth of I/O devices. Thus, a high-performance analytic system might simply be a server that employs relatively cheap high-bandwidth I/O devices like flash SSDs, but has low memory. This method could dramatically change the conventional thinking today about how to build a data processing server. In fact, this line of thinking was a big motivation for the WideTable work.

Appendix A

Supplemental materials

A.1 Converting a bit vector to record numbers

The column-scalar scan methods proposed in this chapter are used to evaluate the predicates in the query (i.e. the WHERE clause), and produce a result bit vector. The next step for query processing is to convert the result bit vector to a record number that can be used to retrieve the attribute/column values that are referred to in other parts of the query (e.g. the projection attributed in the SELECT clause). Here we assume that record numbers can be converted to record ids, if the system uses record ids, using methods similar to those described in [71]. We now present the method that we use to convert the result bit vector to the record numbers of the selected tuples/records.

We first introduce two bitwise manipulations [51] that provide a fast way to find and manipulate the rightmost 1 bit in a word. The notations $\mathbf{R}(x)$, and $\mathbf{P}(x)$ are used to denote the two manipulations, respectively.

$$\begin{aligned} \mathbf{R}(x) = x\&(x-1): &\quad \text{remove the rightmost 1 in } x. \\ \mathbf{P}(x) = x \oplus -x: &\quad \text{propagate the rightmost 1 to the left in } x, \\ &\quad \text{and remove the rightmost 1 in } x. \end{aligned}$$

As an example, here is how these two operations apply on a word x.

$$\begin{aligned} x &= (0001000010011000)_2 \\ \mathbf{R}(x) = x\&(x-1) &= (0001000010010000)_2 \\ \mathbf{P}(x) = x \oplus -x &= (1111111111110000)_2 \end{aligned}$$

Algorithm 8 shows the pseudocode for the conversion algorithm. The basic idea behind

this algorithm is to extract each 1 from the bit vector and compute its offset from the word boundary of the word it occupied. As shown in the algorithm, we iterate over all the words in the input bit vector. In the loop over the words, we first propagate its rightmost 1 to the left in the word x (Line 5: $\mathbf{P}(x)$), and then use the `POPCNT` instruction to count the number of 1s in it (Line 5: $\texttt{popcnt}(\mathbf{P}(x))$), which is the offset of the rightmost 1 in x. This offset is added to the base offset p of the word x to get the corresponding row/record number. Finally, we remove the rightmost 1 from x. We continue this process on word x until there are no 1s in x.

Algorithm 8 Converting a bit vector to a list of record numbers

Input: BV: input bit vector
w: word width
Output: L: a list of record numbers
1: $p := 0$
2: **for** each word x in bit vector BV **do**
3: **while** $x \neq 0$ **do**
4: $\texttt{rid} := p + \texttt{popcnt}(\mathbf{P}(x))$
5: append rid to L
6: $x := \mathbf{R}(x)$
7: $p := p + w$
8: **return** L

A.2 Extracting delimiter bits

Bit-parallel methods rely on a function $f_\circ(X, C)$ that performs simultaneous comparisons on packed codes in a processor word. The outcome of the function is a vector of $\lfloor \frac{w}{b} \rfloor$ results, each of which occupies a b-bit section. The delimiter (leftmost) bit of each section indicates the comparison results. However, if a predicate clause contains multiple predicates on columns with different widths, conjunctions and disjunctions cannot be directly implemented as logical AND and OR operations on the result vectors.

In Section 2.2.2.1, we propose the HBP storage format that simplifies the process to produce the result bit vector with one bit per input code. In this section, we introduce the methods that produce the result bit vector without the help of the HBP storage layout. More specifically, we lays out the codes in order. With regard to the discussion in the last paragraph in Section 2.2.2, the layout of the codes is c1 and c2 in $v1$, c3 and c4 in $v2$, c5 and c6 in $v3$ and so forth in Figure 2.4. Now, the result words from the predicate evaluation function $f_\circ(v_i, C)$ on $v1, v2, \cdots$ are $f_\circ(v_1, C) = R(c_1)000R(c_2)000$, $f_\circ(v_2, C) = R(c_3)000R(c_4)000, \cdots$. Then, these result words must be converted to a bit vector of the

form $R(c_1)R(c_2)R(c_3)R(c_4)\cdots$, by extracting all the delimiter bits $R(c_i)$ and omitting all other bits.

Formally, let X be a word that contains a vector of b-bit codes, denoted as $x_1, x_2, \cdots, x_{\lfloor \frac{w}{b} \rfloor}$, each of which is either in the form of 0^b or in the form of 10^{b-1}. Let m_i be the most significant bit of x_i. Thus, X can be represented in the form of

$$X = (m_1 0^{b-1} m_2 0^{b-1}, \cdots, m_{\lfloor \frac{w}{b} \rfloor} 0^{b-1}).$$

Then, the task is to compute $Y = (m_1 m_2 \cdots m_{\lfloor \frac{w}{b} \rfloor} 0^{\lfloor \frac{w}{b} \rfloor \cdot (b-1)})$. Once the value of Y has been calculated, we trim out the 0s at the tail of Y and concatenate it to produce a result bit vector.

Divide and conquer based method. First, we introduce a method based on divide and conquer technique. In each step shown below, we move the delimiter bits of half of values in the vector.

$$\begin{aligned}
&\text{Step 1: } Y = (X \vee \leftarrow_{b-1} (X)) \wedge (0^{2b-2}1^2 \cdots 0^{2b-2}1^2)\\
&\text{Step 2: } Y = (Y \vee \leftarrow_{2\cdot(b-1)} (Y)) \wedge (0^{4b-4}1^4 \cdots 0^{4b-4}1^4)\\
&\text{Step 3: } Y = (Y \vee \leftarrow_{4\cdot(b-1)} (Y)) \wedge (0^{8b-8}1^8 \cdots 0^{8b-8}1^8)\\
&\qquad \cdots
\end{aligned}$$

It is obvious to see that after the first step, Y is in the form of $(m_1 m_2 0^{2b-2}, \cdots, m_{\lfloor \frac{w}{b} \rfloor - 1} m_{\lfloor \frac{w}{b} \rfloor} 0^{2b-2})$. After the second step, Y is in the form of

$$(m_1 m_2 m_3 m_4 0^{4b-4}, \cdots, m_{\lfloor \frac{w}{b} \rfloor - 3} m_{\lfloor \frac{w}{b} \rfloor - 2} m_{\lfloor \frac{w}{b} \rfloor - 1} m_{\lfloor \frac{w}{b} \rfloor} 0^{4b-4}).$$

We continue this process for $\log_2(\lfloor \frac{w}{b} \rfloor)$ steps. Then, Y is in the form of $Y = (m_1 m_2 \cdots m_{\lfloor \frac{w}{b} \rfloor} 0^{\lfloor \frac{w}{b} \rfloor \cdot (b-1)})$. As a result, this method requires $O(\log_2(\lfloor \frac{w}{b} \rfloor))$ logical bitwise operations to implement the process.

Multiplication based method. Another solution is based on multiplication. It is well known that shifting left by n bits on a number has the effect of multiplying it by 2^n. Essentially, the task of extracting delimiter bits is to left shift m_i by $(i-1)\cdot(b-1)$ bits, e.g. left shift m_1 by 0 bits, m_2 by $b-1$ bits, m_3 by $2\cdot(b-1)$ bits, and so forth. If all delimiter bits does not interfere with others, we can simultaneously left shift all delimiter bits to their appropriate bit positions in a single instruction, by multiplying X by $2^0 + 2^{b-1} + 2^{2b-2} + \cdots + 2^{\lfloor \frac{w}{b} \rfloor \cdot (b-1)} = (0^{b-2}10^{b-2}1 \cdots 0^{b-2}1)$. After that, we apply a mask to clear up

the rightmost $\lfloor \frac{w}{b} \rfloor \cdot (b-1)$ bits. In sum, we have

$$Y = (X \times (0^{b-2}10^{b-2}1 \cdots 0^{b-2}1)) \wedge (1^{\lfloor \frac{w}{b} \rfloor}0^{\lfloor \frac{w}{b} \rfloor \cdot (b-1)}).$$

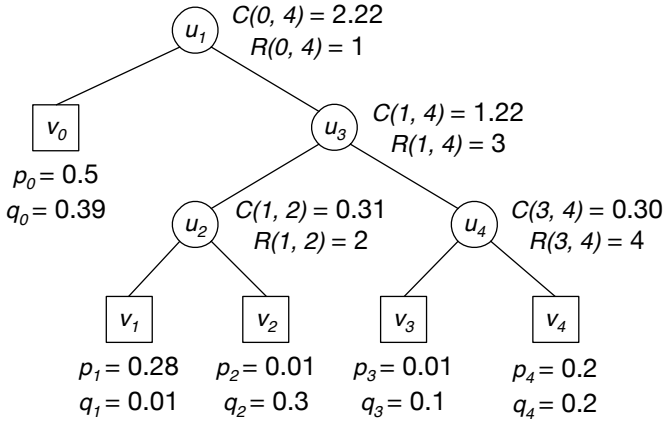
Despite its simplicity, the multiplication based method is correct only when each bit section is not narrower than $\sqrt{w}$ bits, i.e. $b \geqslant \sqrt{w}$. For the 64-bit ALU register, each bit section should be wider than 8 bits. Otherwise, these delimiter bits interfere with each other and the result is incorrect. Another drawback of the multiplication based method is that it can not be applied with the wider SIMD instructions, as the current architecture does not support a multiplication on an entire SIMD word.

Hybrid method. A hybrid method is to use the simple and efficient multiplication based method when $b \geqslant \sqrt{w}$, and the divide and conquer based method for the other scenarios.

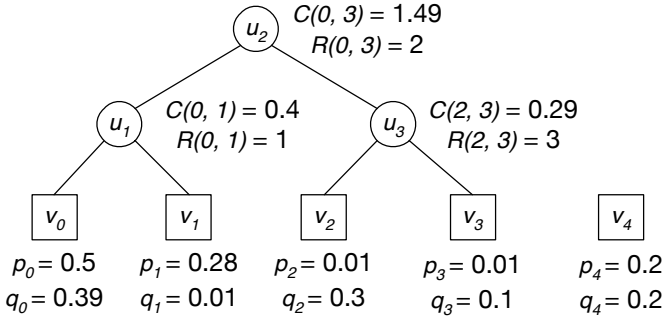
Note that the methods above are all relative expensive compared to the simple computation on the function $f_\circ(X, C)$. Even though the multiplication base method is used (note that it is not feasible for narrow columns), ALU multiplication is several times as expensive as the instructions used in the function $f_\circ(X, C)$, which hinders the overall scan performance. In Section 2.5.2, we empirically compare the HBP method with a method that needs this conversion.

A.3 Counterexample for the monotonicity heuristic

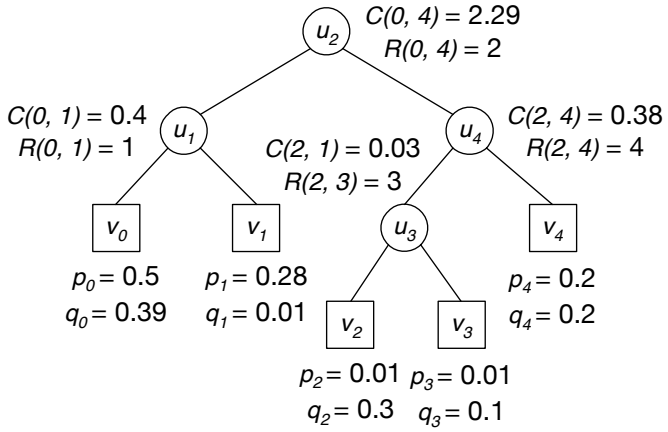
Figure A.1 shows a counterexample for the monotonicity heuristic. There are five values in the encoding, with the predicate literal weights $p_0 = 0.5$, $p_1 = 0.28$, $p_2 = 0.01$, $p_3 = 0.01$, $p_4 = 0.2$, and with the column value weights $q_0 = 0.5$, $q_1 = 0.28$, $q_2 = 0.01$, $q_3 = 0.01$, $q_4 = 0.2$. Figure A.1a illustrates the optimal encoding tree on the five leaf nodes $v_0 \sim v_4$. However, this optimal encoding tree cannot be obtained by using the monotonicity heuristic. To illustrate the reason, Figure A.1b shows the optimal subtree on the leaf nodes $v_0 \sim v_3$. The left subtree and the right subtree is split on the leaf node v_2, i.e. $R(0,3) = 2$. Thus, according to the monotonicity heuristic, $R(0,4)$ should be equal to or greater than $R(0,3)$. Nevertheless, the left subtree of the root and the right subtree in the optimal encoding tree (Figure A.1a) are split at the leaf node v_1, i.e. $R(0,4) = 1 < R(0,3)$. As a result, with the monotonicity heuristic, we fail to find the optimal encoding tree. Figure A.1c show the near-optimal encoding tree constructed with the monotonicity heuristic, whose cost (2.29) is slightly higher than that of the optimal encoding tree (2.22).



(a) Optimal tree on $\nu_0 \sim \nu_4$



(b) Optimal subtree on $\nu_0 \sim \nu_3$



(c) Near-optimal tree on $\nu_0 \sim \nu_4$

Figure A.1: The counterexample encoding tree for the monotonicity heuristic.

A.4 Processing cyclic schema graph in WideTable

In this section, we relax the assumption that the schema graph is a DAG (cf. Section 4.3.2), and extend the algorithm of producing schema trees to deal with schema graphs with cycles.

The extension is fairly simple and as follows: For a schema graph with cycles, WideTable adds a "virtual" source and connects this source to a selected vertex in the graph. To select such a vertex, WideTable finds the largest (in term of cardinality) table in the circular reference, and adds an edge from the source to the corresponding vertex of the table. With the added edge, certain vertices now have more than one incoming edge. Thus, we continue to split vertices of indegree more than one, until a user specified threshold is met on the number of iterations. Finally, we remove the "virtual" source and the latest-added edge in any cycles in the graph to produce the schema tree for the input schema graph.

Example. We illustrate the required steps that are needed to produce a schema tree for an example cyclic schema shown below. In this example, the primary key fields are underlined, and the foreign keys are shown in bold. The schema graph is shown in Figure A.2(a). There is a cyclic reference between the `Emp` and `Dept` tables.

```
Nation(nid, nname)
Emp(eid, ename, gender, did)
Dept(did, dname, nid, eid(manager))
```

The steps that needed to produce a schema tree for this schema are as follows.

1. We first select the `Emp` table as the table to start with, and add a source s and an edge from s to the vertex associated with the `Emp` table. Then, the vertex "`Emp`" becomes a vertex of indegree more than one, and must be split in the next step (Figure A.2(b)).

2. To split the vertex "`Emp`", we replace it by two vertices "`Emp1`" and "`Emp2`", and add edges from both vertices to the vertex "`Dept`". The newly added edge forms a new cycle between the "`Dept`" and "`Emp2`" vertices (Figure A.2(c)). Now, the vertex "`Dept`" has two incoming edges.

3. Next, we continue to split the vertex "`Dept`" and the vertex "`Nation`" in turn (Figure A.2(d) and Figure A.2(e)).

4. Finally, assuming that we have reached the threshold on the number of iterations, we remove the source vertex s and the latest-added edge in the cycle, and generate the schema tree of the example schema graph as shown in Figure A.2(f).

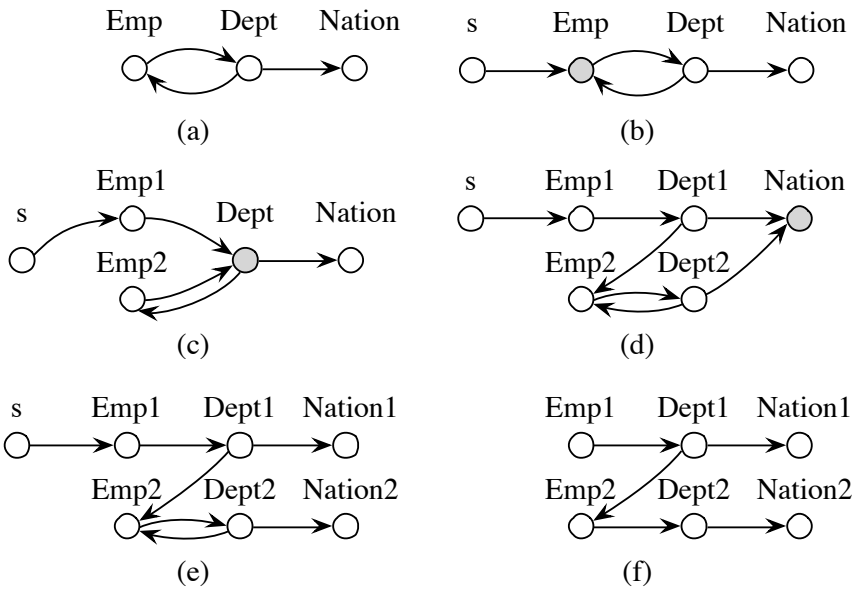


Figure A.2: Steps that are needed to produce a schema tree for a schema graph with cycles. The vertex to be split in the next step is marked in gray.

A.5 MonetDB time breakdown with the TPC-H benchmark

In order to give more insight into the time breakdown of WideTable, we also examine and compare the detailed time breakdown for the key operations when running MonetDB with the TPC-H benchmark. Figure A.3 shows the time breakdown for the different operations. In additional to the five key operations (scan, string, group, top-k, and aggregation) that are shown in Figure 4.22, we also show the time on *join* operation.

As can be seen in Figure A.3, on average, the scan, string, join, group, top-k, and aggregation operations account for 25.3%, 3.6%, 50.8%, 9.8%, 0.1% and 10.0%, of the total running time, respectively. Not surprisingly, join operation is the most expensive operation, and dominates the total running time of MonetDB.

By comparing the time breakdown of WideTable with that of MonetDB in more details, we identify three key reasons behind the high performance of WideTable as follows.

The first and the most apparent reason is that WideTable removes the join component, which accounts for, on average, 50% of the total running time of MonetDB, by flattening out the database schema and pre-joining all tables.

Second, WideTable uses packed code scan method to speedup the scan operation. Although the denormalization strategy might lead to scan more data (on the dimension table side) due to the redundancy introduced by denormalization (see Section 4.2.2 for

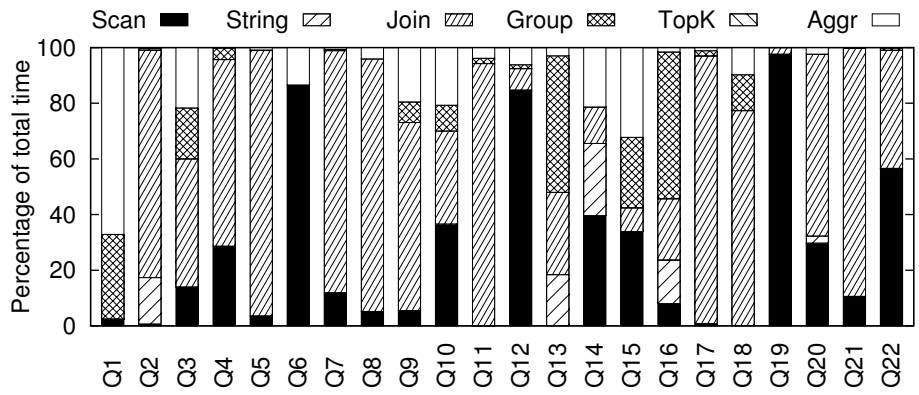


Figure A.3: Time breakdown of MonetDB with the TPC-H queries.

a more detailed analysis), WideTable spends much less time on scans compared with MonetDB. Taking the queries Q12 and Q19 in the TPC-H benchmark as an example, WideTable is 26.9X, and 17.8X faster than MonetDB (as shown in Figure 4.18), respectively. Since the scan operations of these two queries account for nearly the entire query execution time for both WideTable and MonetDB, the speedup mainly comes from the packed code scan method used in WideTable.

Finally, WideTable benefits from dictionary encoding to speedup predicate evaluation with string matching. With order-preserving dictionary encoding technique, prefix matching, e.g. `p_type LIKE 'PROMO%'`, and suffix matching, e.g. `p_type LIKE '%BRASS'`, are converted to simple range predicates in the code domain, and leverages the packed code scan method to efficiently evaluate string matching predicates.

references

[1] Abadi, Daniel J. 2008. Query execution in column-oriented database systems. MIT PhD Dissertation. PhD Thesis.

[2] Abadi, Daniel J., Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *SIGMOD conference*, 671–682.

[3] Abadi, Daniel J., Samuel Madden, and Nabil Hachem. 2008. Column-stores vs. row-stores: how different are they really? In *SIGMOD conference*, 967–980.

[4] Abiteboul, Serge, and Oliver M. Duschka. 1998. Complexity of answering queries using materialized views. In *Pods*, 254–263.

[5] Abraham, Lior, John Allen, Oleksandr Barykin, Vinayak R. Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, David Reiss, Subbu Subramanian, Janet L. Wiener, and Okay Zed. 2013. Scuba: Diving into data at facebook. *PVLDB* 6(11): 1057–1067.

[6] Agrawal, Sanjay, Surajit Chaudhuri, and Vivek R. Narasayya. 2000. Automated selection of materialized views and indexes in sql databases. In *VLDB conference*, 496–505.

[7] Alexiou, Karolina, Donald Kossmann, and Per-Åke Larson. 2013. Adaptive range filters for cold data: Avoiding trips to siberia. *PVLDB* 6(14):1714–1725.

[8] Apaydin, Tan, Guadalupe Canahuate, Hakan Ferhatosmanoglu, and Ali Saman Tosun. 2006. Approximate encoding for direct access and query processing over compressed bitmaps. In *VLDB conference*, 846–857.

[9] Astrahan, Morton M., Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim Gray, Patricia P. Griffiths, W. Frank King III, Raymond A. Lorie, Paul R. McJones, James W. Mehl, Gianfranco R. Putzolu, Irving L. Traiger, Bradford W. Wade, and

Vera Watson. 1976. System r: Relational approach to database management. *ACM Trans. Database Syst.* 1(2):97–137.

[10] Barber, Ronald, Peter Bendel, Marco Czech, Oliver Draese, Frederick Ho, Namik Hrle, Stratos Idreos, Min-Soo Kim, Oliver Koeth, Jae-Gil Lee, Tianchao Tim Li, Guy M. Lohman, Konstantinos Morfonios, René Müller, Keshava Murthy, Ippokratis Pandis, Lin Qiao, Vijayshankar Raman, Richard Sidle, Knut Stolze, and Sandor Szabo. 2012. Business analytics in (a) Blink. *IEEE Data Eng. Bull.* 35(1):9–14.

[11] ———. 2012. Business analytics in (a) blink. *IEEE Data Eng. Bull.* 35(1):9–14.

[12] Binnig, Carsten, Stefan Hildenbrand, and Franz Färber. 2009. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD conference*, 283–296.

[13] Blanas, Spyros, Yinan Li, and Jignesh M. Patel. 2011. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *SIGMOD conference*, 37–48.

[14] Blanas, Spyros, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. 2010. A comparison of join algorithms for log processing in mapreduce. In *SIGMOD conference*, 975–986.

[15] Boncz, Peter A., Martin L. Kersten, and Stefan Manegold. 2008. Breaking the memory wall in MonetDB. *Commun. ACM* 51(12):77–85.

[16] Boncz, Peter A., Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-pipelining query execution. In *CIDR conference*, 225–237.

[17] Chang, Lei, Zhanwei Wang, Tao Ma, Lirong Jian, Lili Ma, Alon Goldshuv, Luke Lonergan, Jeffrey Cohen, Caleb Welton, Gavin Sherry, and Milind Bhandarkar. 2014. HAWQ: a massively parallel processing SQL engine in hadoop. In *SIGMOD conference*, 1223–1234.

[18] Chasseur, Craig, and Jignesh M. Patel. 2013. Design and evaluation of storage organizations for read-optimized main memory databases. *PVLDB* 6(13):1474–1485.

[19] Chaudhuri, Surajit, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. 1995. Optimizing queries with materialized views. In *ICDE conference*, 190–200.

[20] Chen, Zhiyuan, Johannes Gehrke, and Flip Korn. 2001. Query optimization in compressed database systems. In *SIGMOD conference*, 271–282.

[21] Costa, João Pedro, José Cecílio, Pedro Martins, and Pedro Furtado. 2011. ONE: A predictable and scalable DW model. In *DaWaK conference*, 1–13.

[22] ———. 2012. Overcoming the scalability limitations of parallel star schema data warehouses. In *ICA3PP conference*, 473–486.

[23] Dayal, Umeshwar. 1987. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *VLDB conference*, 197–208.

[24] DeWitt, David J., Jeffrey F. Naughton, Donovan A. Schneider, and S. Seshadri. 1992. Practical skew handling in parallel joins. In *VLDB conference*, 27–40.

[25] Faloutsos, Christos, and H. V. Jagadish. 1992. On b-tree indices for skewed distributions. In *VLDB conference*, 363–374.

[26] Fang, Wenbin, Bingsheng He, and Qiong Luo. 2010. Database compression on graphics processors. *PVLDB* 3(1):670–680.

[27] Färber, Franz, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA database – an architecture overview. *IEEE Data Eng. Bull.* 35(1):28–33.

[28] Feng, Ziqiang, and Eric Lo. 2015. Accelerating aggregation using intra-cycle parallelism. In *ICDE conference*, 291–302.

[29] Feng, Ziqiang, Eric Lo, Ben Kao, and Wenjian Xu. 2015. Byteslice: Pushing the envelop of main memory data processing with a new storage layout. In *SIGMOD conference*, 31–46.

[30] French, C.D. 1997. Teaching an oltp database kernel advanced datawarehousing techniques. In *ICDE conference*, 194–198.

[31] Ganski, Richard A., and Harry K. T. Wong. 1987. Optimization of nested sql queries revisited. In *SIGMOD conference*, 23–33.

[32] Garsia, Adriano M., and Michelle L. Wachs. 1977. A new algorithm for minimum cost binary trees. *SIAM J. Comput.* 6(4):622–642.

[33] Giannikis, Georgios, Philipp Unterbrunner, Jeremy Meyer, Gustavo Alonso, Dietmar Fauser, and Donald Kossmann. 2010. Crescando. In *SIGMOD conference*, 1227–1230.

[34] Gilbert, E. N., and E. F. Moore. 1959. Variable-length binary encodings. *Bell System Technical Journal* 38(4):933–967.

[35] Graefe, Goetz. 1993. Query evaluation techniques for large databases. *ACM Comput. Surv.* 25(2):73–170.

[36] Gupta, Ashish, and Inderpal Singh Mumick. 1995. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.* 18(2):3–18.

[37] Halim, Felix, Stratos Idreos, Panagiotis Karras, and Roland H. C. Yap. 2012. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *PVLDB* 5(6):502–513.

[38] Hortonworks. 2015. Apache Tez. `http://hortonworks.com/hadoop/tez/`.

[39] Hu, T. C., and A. C. Tucker. 1971. Optimal computer search trees and variable-length alphabetical codes. *SIAM Journal on Applied Mathematics* 21(4):514–532.

[40] Huffman, David A, et al. 1952. A method for the construction of minimum redundancy codes. *Proc. IRE* 40(9):1098–1101.

[41] Idreos, Stratos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. 2012. MonetDB: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.* 35(1):40–45.

[42] Idreos, Stratos, Martin L. Kersten, and Stefan Manegold. 2007. Database cracking. In *CIDR conference*, 68–78.

[43] Idreos, Stratos, Stefan Manegold, Harumi A. Kuno, and Goetz Graefe. 2011. Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores. *PVLDB* 4(9):585–597.

[44] Intel Corporation. 2014. *Intel® Architectures Instruction Set Extensions Programming Reference*.

[45] ———. 2015. Intel xeon processor e7-4800/8800 v3 product families: Datasheet volumn1: EMTS.

[46] Ivanova, Milena, Martin L. Kersten, Niels J. Nes, and Romulo Goncalves. 2009. An architecture for recycling intermediates in a column-store. In *SIGMOD conference*, 309–320.

[47] Johnson, Ryan, Vijayshankar Raman, Richard Sidle, and Garret Swart. 2008. Row-wise parallel predicate evaluation. *PVLDB* 1(1):622–634.

[48] Kemper, Alfons, and Thomas Neumann. 2011. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *ICDE conference*, 195–206.

[49] Knuth, Donald E. 1971. Optimum binary search trees. *Acta Inf.* 1:14–25.

[50] ———. 1998. *The art of computer programming, volume iii: Sorting and searching, 2nd edition*. Addison-Wesley.

[51] ———. 2011. *The art of computer programming, volume 4a: Combinatorial algorithms, part 1*. Addison-Wesley.

[52] Kornacker, Marcel, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. 2015. Impala: A modern, open-source SQL engine for hadoop. In *CIDR conference*.

[53] Korth, Henry F., Gabriel M. Kuper, Joan Feigenbaum, Allen Van Gelder, and Jeffrey D. Ullman. 1984. System/U: A database system based on the universal relation assumption. *ACM Trans. Database Syst.* 9(3):331–347.

[54] Krüger, Jens, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. 2011. Fast updates on read-optimized databases using multi-core CPUs. *PVLDB* 5(1):61–72.

[55] Kwon, YongChul, Magdalena Balazinska, Bill Howe, and Jerome A. Rolia. 2012. Skewtune: mitigating skew in mapreduce applications. In *SIGMOD conference*, 25–36.

[56] Lamb, Andrew, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandier, Lyric Doshi, and Chuck Bear. 2012. The vertica analytic database: C-store 7 years later. *PVLDB* 5(12):1790–1801.

[57] ———. 2012. The vertica analytic database: C-store 7 years later. *PVLDB* 5(12): 1790–1801.

[58] Lamport, Leslie. 1975. Multiple byte processing with full-word instructions. *Commun. ACM* 18(8):471–475.

[59] Larson, Per-Åke, and Jingren Zhou. 2005. View matching for outer-join views. In *VLDB conference*, 445–456.

[60] ———. 2007. Efficient maintenance of materialized outer-join views. In *ICDE conference*, 56–65.

[61] Lee, Jae-Gil, Gopi K. Attaluri, Ronald Barber, Naresh Chainani, Oliver Draese, Frederick Ho, Stratos Idreos, Min-Soo Kim, Sam Lightstone, Guy M. Lohman, Konstantinos Morfonios, Keshava Murthy, Ippokratis Pandis, Lin Qiao, Vijayshankar Raman, Vincent Kulandai Samy, Richard Sidle, Knut Stolze, and Liping Zhang. 2014. Joins on encoded and partitioned data. *PVLDB* 7(13):1355–1366.

[62] Li, Wei, Dengfeng Gao, and Richard T. Snodgrass. 2002. Skew handling techniques in sort-merge join. In *SIGMOD conference*, 169–180.

[63] Li, Yinan, Craig Chasseur, and Jignesh M. Patel. 2015. A padded encoding scheme to accelerate scans by leveraging skew. In *SIGMOD conference*, 1509–1524.

[64] Li, Yinan, and Jignesh M. Patel. 2013. BitWeaving: fast scans for main memory data processing. In *SIGMOD conference*, 289–300.

[65] ———. 2014. WideTable: An accelerator for analytical data processing. *PVLDB* 7(10): 907–918.

[66] Lynch, Clifford A. 1988. Selectivity estimation and query optimization in large databases with highly skewed distribution of column values. In *VLDB conference*, 240–251.

[67] Maier, David, Jeffrey D. Ullman, and Moshe Y. Vardi. 1984. On the foundations of the universal relation model. *ACM Trans. Database Syst.* 9(2):283–308.

[68] MapR Inc. 2015. Apache Drill. `https://drill.apache.org`.

[69] Melnik, Sergey, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive analysis of web-scale datasets. *PVLDB* 3(1):330–339.

[70] O'Neil, Patrick, Elizabeth O'Neil, and Xuedong Chen. 2007. The star schema benchmark. `http://www.cs.umb.edu/~poneil/StarSchemaB.pdf`.

[71] O'Neil, Patrick E., and Dallan Quass. 1997. Improved query performance with variant indexes. In *SIGMOD conference*, 38–49.

[72] Oracle Corporation. 2014. Oracle exalytics in-memory machine: A brief introduction. *Oracle White Paper*.

[73] Pavlo, Andrew, Carlo Curino, and Stanley B. Zdonik. 2012. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD conference*, 61–72.

[74] Polychroniou, Orestis, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD vectorization for in-memory databases. In *SIGMOD conference*, 1493–1508.

[75] Polychroniou, Orestis, and Kenneth A. Ross. 2013. High throughput heavy hitter aggregation for modern SIMD processors. In *DaMoN workshop*, 6.

[76] ———. 2015. Efficient lightweight compression alongside fast scans. In *DaMoN workshop*.

[77] Power, Jason, Yinan Li, Mark D. Hill, Jignesh M. Patel, and David A. Wood. 2015. Implications of emerging 3d GPU architecture on the scan primitive. *SIGMOD Record* 44(1):18–23.

[78] ———. 2015. Toward gpus being mainstream in analytic processing: An initial argument using simple scan-aggregate queries. In *DaMoN workshop*.

[79] Raman, Vijayshankar, Gopi K. Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, René Müller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam J. Storm, and Liping Zhang. 2013. Db2 with blu acceleration: So much more than just a column store. *PVLDB* 6(11):1080–1091.

[80] Raman, Vijayshankar, Garret Swart, Lin Qiao, Frederick Reiss, Vijay Dialani, Donald Kossmann, Inderpal Narang, and Richard Sidle. 2008. Constant-time query processing. In *ICDE conference*, 60–69.

[81] Rinfret, Denis, Patrick E. O'Neil, and Elizabeth J. O'Neil. 2001. Bit-sliced index arithmetic. In *SIGMOD conference*, 47–57.

[82] Sanders, G. Lawrence, and Seungkyoon Shin. 2001. Denormalization effects on performance of rdbms. In *HICSS conference*.

[83] Schlegel, Benjamin, Rainer Gemulla, and Wolfgang Lehner. 2010. Fast integer compression using SIMD instructions. In *DaMoN workshop*, 34–40.

[84] Selinger, Patricia G., Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access path selection in a relational database management system. In *SIGMOD conference*, 23–34.

[85] Seshadri, Praveen, Hamid Pirahesh, and T. Y. Cliff Leung. 1996. Complex query decorrelation. In *ICDE conference*, 450–458.

[86] Shin, Seungkyoon, and G. Lawrence Sanders. 2006. Denormalization strategies for data retrieval from data warehouses. *Decision Support Systems* 42(1):267–282.

[87] Stonebraker, Michael, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. 2005. C-store: A column-oriented dbms. In *VLDB conference*, 553–564.

[88] Sun, Liwen, Michael J. Franklin, Sanjay Krishnan, and Reynold S. Xin. 2014. Fine-grained partitioning for aggressive data skipping. In *SIGMOD conference*, 1115–1126.

[89] Transaction Processing Performance Council. 2011. *TPC benchmark H. revision 2.14.3*.

[90] Tu, Stephen, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. 2013. Processing analytical queries over encrypted data. *PVLDB* 6(5):289–300.

[91] Ullman, Jeffrey D. 1983. On kent's "consequences of assuming a universal relation". *ACM Trans. Database Syst.* 8(4):637–643.

[92] Unterbrunner, Philipp, Georgios Giannikis, Gustavo Alonso, Dietmar Fauser, and Donald Kossmann. 2009. Predictable performance for unpredictable workloads. *PVLDB* 2(1):706–717.

[93] Walton, Christopher B., Alfred G. Dale, and Roy M. Jenevein. 1991. A taxonomy and performance model of data skew effects in parallel joins. In *VLDB conference*, 537–548.

[94] Westmann, Till, Donald Kossmann, Sven Helmer, and Guido Moerkotte. 2000. The implementation and performance of compressed databases. *SIGMOD Record* 29(3): 55–67.

[95] Willhalm, Thomas, Ismail Oukid, Ingo Müller, and Franz Faerber. 2013. Vectorizing database column scans with complex predicates. In *ADMS workshop*, 1–12.

[96] Willhalm, Thomas, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. 2009. SIMD-scan: Ultra fast in-memory table scan using on-chip vector processing units. *PVLDB* 2(1):385–394.

[97] Wolf, Joel L., Daniel M. Dias, Philip S. Yu, and John Turek. 1991. An effective algorithm for parallelizing hash joins in the presence of data skew. In *ICDE conference*, 200–209.

[98] Wu, Kun-Lung, and Philip S. Yu. 1998. Range-based bitmap indexing for high cardinality attributes with skew. In *COMPSAC conference*, 61–67.

[99] Xin, Reynold S., Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2013. Shark: Sql and rich analytics at scale. In *SIGMOD conference*, 13–24.

[100] Xu, Yu, and Pekka Kostamaa. 2009. Efficient outer join data skew handling in parallel DBMS. *PVLDB* 2(2):1390–1396.

[101] Xu, Yu, Pekka Kostamaa, Xin Zhou, and Liang Chen. 2008. Handling data skew in parallel joins in shared-nothing systems. In *SIGMOD conference*, 1043–1052.

[102] Ye, Yang, Kenneth A. Ross, and Norases Vesdapunt. 2011. Scalable aggregation on multicore processors. In *DaMoN workshop*, 1–9.

[103] Zaharia, Matei, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI conference*, 15–28.

[104] Zhou, Jingren, and Kenneth A. Ross. 2002. Implementing database operations using SIMD instructions. In *SIGMOD conference*, 145–156.

[105] ———. 2002. Implementing database operations using simd instructions. In *SIGMOD conference*, 145–156.

[106] Zukowski, Marcin, and Peter A. Boncz. 2012. Vectorwise: Beyond column stores. *IEEE Data Eng. Bull.* 35(1):21–27.

[107] Zukowski, Marcin, Sándor Héman, Niels Nes, and Peter A. Boncz. 2006. Super-scalar RAM-CPU cache compression. In *ICDE conference*, 59.