IN MEMORY DATA MANAGEMENT: FROM HARDWARE TO APPLICATION

BO TANG

Ph.D

The Hong Kong Polytechnic University

2017

The Hong Kong Polytechnic University

Department of Computing

In Memory Data Management: From Hardware To Application

Bo Tang

A thesis submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

January 2017

CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

Bo Tang

Abstract

With the availability of very large and inexpensive main memory, it is becoming practical to manage data managements in main memory and benefit from high-speed access. For instance, in-memory database management systems (e.g., SAP HANA and Oracle TimesTen) provide much higher performance over disk-oriented database management systems for relational data. In this thesis, we identify and address some unsolved issues in in-memory data management, from hardware to applications.

First, we exploit the hardware aspect (e.g., CPU and memory) to accelerate distance computations (on data points), which are core subroutines in many applications, e.g., trajectory search, motif discovery and kNN classification. This involves two research problems: (1) how to exploit every CPU cycle for computation, and (2) how to exploit every bit of main memory for caching data points. Our work is orthogonal to existing pruning techniques and index structures on data points. Regarding (1), we unlock the potentials of modern commodity CPUs (i.e., data parallelism, CPU caches, branch prediction). Regarding (2), we propose to cache compact approximate representations of data points in main memory in order to reduce the candidate refinement time in existing kNN search methods. For each research problem above, we evaluate the performance of our solutions on real datasets and show that our solutions are effective and scalable.

Next, we focus on the application aspect and consider in-memory OLAP tools, which have been extensively used by enterprises to make better and faster decisions. Specifically, we take the first attempt towards automatically extracting top-k insights from in-memory OLAP cube. It is useful not only for non-expert users, but also reduces the manual effort of data analysts. It has challenges on (i) the effectiveness of the extracted insights and (ii) the efficiency of top-k insight computation for in-memory data warehouses. We first propose a meaningful scoring function for insights to address (i). Then, we contribute a computation framework for top-k insights, together with a suite of computation optimization techniques to address (ii). Our experimental study on both real data and

synthetic data verifies the effectiveness and efficiency of our proposed solution.

Acknowledgements

I am deeply grateful to the many people who supported me during my PhD life. First and foremost, I would like to express my deepest sense of gratitude to my advisor, Dr. Man Lung Yiu, for the guidance during my study. He advised, supported and challenged me during those four years. Without him, I would not have the chance to complete this work. I would need an entire new Acknowledgment section to list all that he taught me.

Second, a number of people directly contributed to the work described in this thesis. I would also like to thank Prof. Kien A. Hua for his suggestions and comments both on research and academic career. I am grateful to Dr. Leong Hou U, and Dr. Yuhong Li, for taking me as a collaborator and for providing valuable feedback. I thank Dr. Dongmei Zhang, who gave me the chance as a Microsoft intern.

Most importantly, I would like to thank Dr. Kyriakos Mouratidis for his herculean effort to teach and train me to be a professional researcher. I would also thank Prof. Martin Kersten for told me where is the research beginning, and Prof. Stefan Manegold for his thoughts and insights about how to be a pure computer scientist. I was glad and proud to work with MonetDB team at the

Dutch National Center for Mathematics and Computer Science. In addition, I also had the pleasure of working with Dr. Chuanfei Xu, Mr. Kai Wang, Mr. Jiahao Zhang, Mr. Shi Han, and Mr. Rui Ding.

Next, my appreciation also goes to many people inside and outside database group at the Hong Kong Polytechnic University, who are: Eric Lo, Qiang Zhang, Ziqiang Feng, Jianguo Wang, Capital Li, Yu Li, Wenjian Xu, Zhian He, Duncan Yung, Petrie Wong, Jeppe Thomsen, Ran Bai, Edison Chan, Chris Liu and Henry Yang, for their kindly help and support. I am grateful to Qiang Zhang for the life in Hong Kong. He also gave me a lot of fun outside research.

Finally, I would like to acknowledge my friends and family for their support. I wish to acknowledge a few of the close friends/teachers who supported in past few years: Prof. Meng Ni, Prof. Min Zhu, Mrs. Li Ma, Mr. Ruidong Liu, Mr. Qiaomu Shen and Mr. Yuhao Su. I also thank my mother and father, who are awesome parent and I must apologize for not being able to stay with them for many years. I would also thank my sister Ya Tang, who raised me up together with my parents. She also offered me a lot of opportunities to explore the wonderful world. Last, but not least, I want to thank my wife, Na Pan, who has been together with me for many years. Her love, patience, support and endless care enable me to carry through my Ph.D. study to the end.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 In-memory Database Management System

Traditional Database Management Systems (DBMS) are designed to manage a vast amount of data stored in hard disk. Query processing and optimization techniques of traditional DBMS are tailored to exploit the characteristics of disk storage mechanisms [41].

With the availability of very large and inexpensive main memory, it is becoming practical to manage data in main memory and benefit from high-speed access. For instance, in-memory database management systems provide much higher performance over disk-oriented database management systems for relational data. Specifically, SAP HANA [5] is an in-memory data management system, which exploits the characteristics of modern hardware (e.g., massive main memory, multi/many CPU cores) to improve the performance of transaction processing and analytical processing.

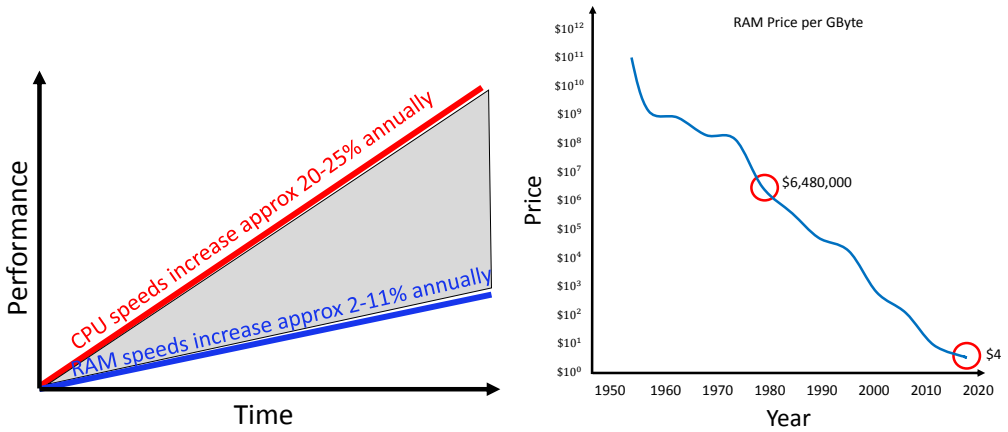
In the last two decades, we have been witnessing the advance of hardware (i.e., cheaper RAM, multi/many cores). Many techniques (e.g., indexing, data layout, parallelism, concurrency control, fault tolerance) have been proposed to redesign the in-memory database management system. These works focus on improving the efficiency of in-memory DBMS [105]. For example, in-memory DBMS has efficient indexes (e.g., cache sensitive search trees) to avoid memory-intensive scan, and columnar layout of relational table to accelerate scan / look-up operators, as B^+-tree and row-based layout are not suitable for in-memory scenario.

1.2 Research Problems on In-memory Data Management

In summary, these existing works focus on in-memory DBMS. In this thesis, we focus on three specific unsolved research problems in in-memory data management from hardware to applications.

As shown in Figure 1.1(a), in the last decades, CPU speed improved at an annual rate of 22-25% while memory (i.e., RAM) speed only improved at 2-11%. Given these trends, the speed gap between CPU and memory RAM is becoming larger and larger. Thus, CPU cache hierarchy was proposed to reduce expensive main memory data access. However, there is a tradeoff between cache size and access speed among these cache levels. For instance, L1 cache has the smallest size and the fastest access speed, and L3 cache has the largest size with the lowest access speed in modern commodity CPUs (as shown in Figure 1.2). In order to

fully exploit the computation ability of CPUs, we first investigate how to exploit every cycles of CPUs for computation intensive workloads (e.g., similarity search) in Chapter 3.



(a) Memory wall (b) Memory price

Figure 1.1. Research motivations

At the same time, with the Moore's law, the price of main memory is falling down. For example, it needs 6 millions US dollar for a 1GB RAM in 1980s, however it only needs 4 US dollar in 2015, as illustrated in Figure 1.1(b). With such large amount of main memory, how to exploit every bit for applications in massive dataset (e.g., similarity search on high dimensional dataset) is an open question, which we study in Chapter 4.

In general, there are several data storage levels in computer storage architecture (as shown in Figure 1.2), Chapter 3 and 4 focus on accelerating specific data management task (i.e., similarity search) by unlocking the potential between two or more levels in computer hardware (i.e, storage architecture).

In-memory on-line analytical processing (OLAP) is a core subroutine of in-memory data management system. OLAP tools have been extensively used by

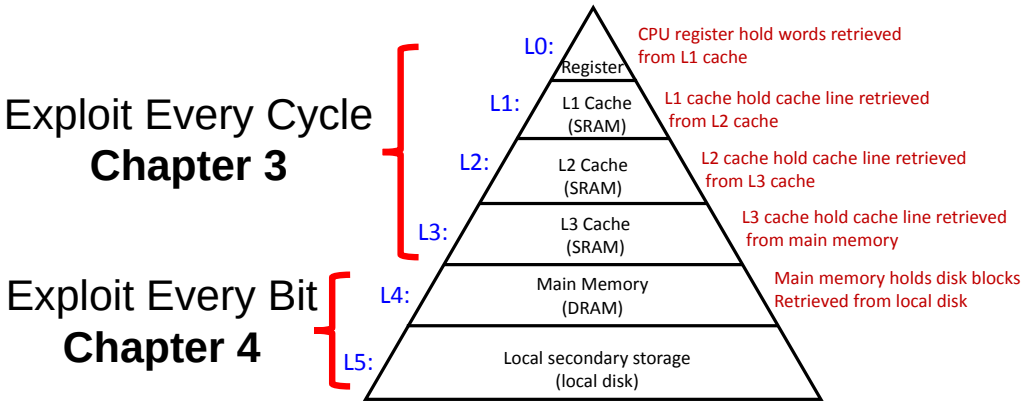


Figure 1.2. Computer storage architecture

enterprises to make better and faster decisions.

By using the results and findings in Chapter 3 and 4, we propose an in-memory computation framework to extract top-k insights from multidimensional data in Chapter 5. It is a novel extension of current OLAP systems as it can automatically extract useful insights from the dataset without any user's input. This is useful not only for non-expert users, but also reduces the manual effort of data analysts. It has many applications in business intelligence area (as shown in Figure 1.3).

1.3 Thesis Organization

The rest of this thesis is organized as follows. We summarize the literature of our research problems in Chapter 2.

Chapter 3 (based on [92]) studies distance computation acceleration for different applications on modern commodity CPUs. Many existing algorithms reduce the computation cost by pruning unpromising candidates with lower-

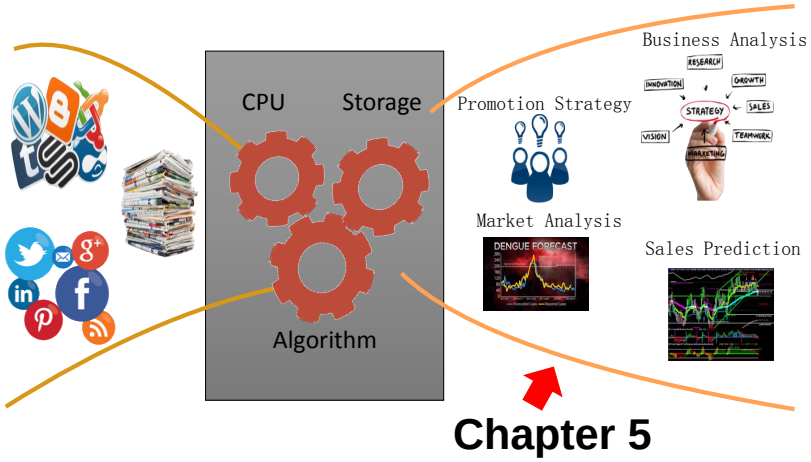


Figure 1.3. In-memory on-line analytical processing applications

bound distance functions. Even with these lower bounds, the above algorithms remain computation intensive. In this chapter, we focus on an orthogonal research direction that further boosts the performance by unlocking the potentials of modern commodity CPUs. Our experimental study on real datasets shows that our proposal can achieve up to 6 times of speedup.

Chapter 4 (based on [91]) discusses caching techniques for high dimensional nearest neighbor search problem. Existing disk-based kNN search methods incur significant I/O costs in the candidate refinement phase. We propose to cache compact approximate representations of data points in main memory in order to reduce the candidate refinement time during kNN search. First, we formulate and solve a novel histogram optimization problem that decides the most effective approximate representation scheme for data points. We then develop a cost model for automatically tuning the optimal number of bits for encoding points. In addition, our approach is generic and applicable to exact / approximate kNN search methods. Extensive experimental results on real datasets demonstrate

that our proposal can accelerate the candidate refinement time of kNN search by at least an order of magnitude.

Chapter 5 (based on [90]) proposes an automatic and effective insights extraction solution. Existing data analysis tools require tedious hit-and-trial from the user, on manually posing queries, analyzing results and deciding what is interesting. To alleviate this issue, we take the first attempt towards automatically extracting top-k *insights* from multidimensional data. This is useful not only for non-expert users, but also reduces the manual effort of data analysts. We propose a meaningful scoring function in order to determine important insights. Then, we contribute a computation framework for top-k insights, together with a suite of optimization techniques to provide an efficient solution. Our experimental study on both real data and synthetic data verifies the effectiveness and efficiency of our proposed solution.

Chapter 6 concludes the thesis and discusses the future research directions.

Chapter 2

Literature Review

In this chapter, we review the existing works related to this thesis. Section 2.1 discusses in-memory data management techniques. Section 2.2 elaborates similarity search on high dimensional dataset. Section 2.3 overviews multidimensional dataset exploration.

2.1 In-memory Techniques

In order to design an efficient in-memory data management systems, various aspects of research problems have been studied. Generally, the researches on disk-based DBMS focus on optimizing the I/O access time. However, many in-memory DBMS works exploit the following characteristics of modern CPUs to improve the performance. For instance, SIMD and multi-core CPUs have been used to speedup fundamental database operators (e.g., scalar aggregation, scan) [107], sorting [26], and joining [74, 17, 15].

1. **CPU cache hierarchy**: In order to hide memory latency from the processor, CPU cache hierarchy is used. Modern commodity CPUs include three levels cache (i.e., L1, L2, L3 in Figure 1.2). There are tradeoffs between cache size and access latency among these different cache levels. For example, L1 is 32KB with 4 cycles access latency, and L3 is 24MB with almost 100 cycles access latency in Intel Xeon E7-4850.

2. **Single instruction multiple data (SIMD)**: Modern commodity CPUs provide vector instructions (SIMD) operating on wide register (e.g., 256-bits, 512-bits) can perform the same instruction on multiple data values in parallel.

3. **Hardware prefetcher**: Modern commodity CPUs have built-in hardware prefetcher. It allows to prefetch additional lines of instruction or data into the L1 or L2 cache in CPU cores. It can reduce data and instruction access latency.

4. **Multi-core**: A chip contains multiple cores, e.g., Intel i7 has 4 physical cores in one chip. Different cores may execute different threads in parallel.

5. **Simultaneous multithreading**: This feature supports running multiple concurrent threads in the same CPU core.

The performance of disk-based DBMS is measured by the total of disk I/O times. Indexes for disk-based DBMS (e.g., B^+ tree) are not suitable for in-memory database systems. Thus, several in-memory indexes have been proposed (e.g., Cache Sensitive Search Trees [73], Adaptive Radix Tree [58]) to support effective query processing in-memory. Different data layouts (e.g., columnar

layout [71], hybird of row and column layout [5]) have been proposed to achieve good cache locality[54], better data compression [59] for efficient data scan and look-up with in-memory environment. In the literature, many other optimization aspects (e.g., concurrency control, fault tolerance) have been studied in literature for in-memory DBMS. We refer the interested reader to a recent survey [105].

2.2 Similarity Search on High Dimensional Data

Typically, given a high dimensional dataset D and a query point q, the similarity search is finding the similar objects of q in D with a specific distance function $f(\cdot)$. K nearest neighbor (kNN) search and range queries are important subclasses of similarity search. In this section, we introduce distance functions in Section 2.2.1, then review the similarity search applications in Sections 2.2.2 and 2.2.3.

2.2.1 Distance Functions

We review the distance functions from the following aspects: distance function, computation complexity and distance metric, as illustrated in Table 2.1, where m is the dimensionality of data objects in D. A distance is a metric distance when it has identity, nonnegativity, symmetry and triangle inequality properties [85].

Distance function $f(\cdot)$	Complexity	Metric	Reference
L_2 - norm (ED)	$O(m)$	Yes	[44]
Dynamic Time Warping (DTW)	$O(m^2)$	No	[44]
Longest Common SubSequence (LCSS)	$O(m^2)$	No	[44]
Discrete Fréchet Distance (DFD)	$O(m^2)$	Yes	[44]
Earth Mover's Distance (EMD)	$O(m^2)$	Yes	[30]
Kullback - Leibler Divergence (KL-D)	$O(m)$	No	[30]
Edit Distance on Real Sequence (EDR)	$O(m^2)$	No	[33]
Edit Distance with Real Penalty (ERP)	$O(m^2)$	No	[33]

Table 2.1. Distance functions

2.2.2 High Dimensional Feature Vector

In this section, we overview the kNN search on high dimensional feature vector with L_p-norm distance (e.g., L_2 norm).

Multimedia object (e.g., image, video) can be represented by a high dimensional feature vector. In high dimensional exact kNN search, tree-based indexes (e.g., R-tree, X-tree, SR-tree) [18] suffer from the *dimensionality curse* [98], so their running time for kNN search degenerates to that of linear scan. The VA-file [98] and its variants VA$^+$-file [37] proposes approximate representations of points to support efficient linear scan.

We classify existing methods on approximate kNN search into two types: (1) LSH based methods [48, 29, 64, 93, 39, 104] that provide theoretical result accuracy guarantees. Specifically, LSH methods aim to compute c-approximate kNN results in sub-linear time, i.e., the result distances are at most c times of the exact result distances, and (2) Non-LSH based methods [10, 11, 99, 40, 20] that optimize the result accuracy based on training data. Both types of methods process a query q in two phases. First, we retrieve a candidate set of object identifiers (the candidate generation phase). Then, we fetch their data points

from a disk file to finalize the kNN results (the candidate refinement phase). SK-LSH [104] rearrange the data file such that similar points are likely to be placed on the same disk page. This would reduce the I/O cost in the candidate refinement phase.

In addition, there exist some caching techniques for kNN search in the distance metric space [34, 85]. Falchi et al. [34] study caching the results of kNN queries, whereas Skopal et al. [85] propose to cache the distances obtained from kNN queries. These techniques are designed for metric space indexes (e.g., M-tree [27], iDistance [52]).

2.2.3 Time Series Data

Time series data can also be modeled as high dimensional feature vectors as discussed above. The similarity search problems on time series data: (i) the *subsequence search* problem [38, 108, 33, 69, 83, 22, 72], (ii) the *motif discovery* problem [67] and (iii) the kNN *classification* problem [33, 72]. Faloutsos et. al. used R*-tree to improve the performance of subsequence matching in time series databases in [35]. However, as stated in [72], the existing indexing techniques [35, 9] are inefficient these similarity search problems.

The typical distance functions for time series similarity search are the Euclidean distance (ED) and Dynamic Time Warping (DTW). Existing time series algorithms rely on software-level optimizations such as lower-bound functions [35, 9, 69, 108, 72], early abandon techniques [72]. However, existing solutions incur high CPU stall times and there are rooms to further improve the efficiency.

2.3 Multidimensional Data Exploration

Data exploration is about efficiently extracting knowledge from data [46]. In the following, we review the related work in each relevant area.

2.3.1 Top-k Problems

Top-k queries have been extensively studied in databases [47]. They require user to specify a ranking function (or the weighting of attributes), and then return k result objects. In contrast, our studied problem does not require any user parameter and our results are insights rather than objects.

2.3.2 OLAP Data Cube

The OLAP data cube model [43] supports efficient aggregation on a multi-dimensional dataset and allows users to navigate the aggregation result by operations (e.g., slicing, dicing, drill up/down). Efficient construction algorithms for data cubes have also been studied [36, 16]. For example, the iceberg cube model [16] avoids computing the larger group-bys that do not meet minimum support.

Advanced cubes have been proposed for other forms of analysis beyond aggregation such as dominant relationship analysis [60], statistical analysis [61], and ranking analysis [101, 102]. However, these techniques rely on monotonicity and/or convexity of an aggregate measure to speedup query processing and or reduce space size.

2.3.3 Mining and Learning-based Techniques

Recent works [63, 2] employ data mining techniques (e.g., outlier detection, cluster analysis) and machine learning techniques (e.g., inductive learning [66]) on datasets to perform pattern discovery and predictive analytics, respectively.

For example, [63] combined the best features of existing standard methodologies such as principal component and cluster analyses to provide a geometric representation of complex data sets.

Inductive learning [66] is a process of acquiring knowledge by drawing inductive inferences from teacher- or environment-provided facts. Although it is one of the most common forms of learning, it has one fundamental weakness: except for special cases, the acquired knowledge cannot be completely validated.

2.3.4 Subspace Analysis

Subspace analysis has been considered in [70, 68, 89, 102]. Pei et al. [70] examine how to identify subspaces such that an object belongs to the skyline. Hassan et al. [89] aim to discover contextual skyline objects that belong to the skyline with respect to a subspace of measures subject to a conjunctive constraint on dimensional attributes. [68] has studied various problems on subspace mining analysis (e.g., subspace clustering, outlier mining).

The most related work to ours is [102]. Given a query object specified by user, their problem is to find the top-R subspaces with the highest 'promotiveness' values. The 'promotiveness' value of an object in a subspace S is defined in terms of (i) the rank of an object in S, and (ii) the number of objects in S.

2.3.5 Exploratory Analysis

In database community, various exploratory analysis techniques have been designed to find interesting information (i.e., explain difference, finding outlier) from the data. The most relevant works in exploratory analysis area are: Sarawagi et al. [77], Wu. et. al [102], ARcube [101], IBM Cogons[1], and SEEDB [95]. We summarize these works in Table 2.2, with respect to three features, i.e., user input, top-k, and result.

Approaches	User input	Top-k	Result
Sarawagi. et. al [77]	OLAP operations	No	anomalies
Wu. et. al [102]	promotion objectives, integer k	Yes	subspaces
ARcube [101]	aggregate queries, integer k	Yes	aggregate values
IBM Cogons [1]	OLAP operations	No	aggregate values
SEEDB [95]	queries	No	visualization

Table 2.2. Comparison with related works

In addition, in the database community, several works have investigated efficient techniques for data exploration [75, 78, 32, 81]. We omit the discussion of these works and refer readers to the recent overview paper [46].

Chapter 3

Exploit Every Cycle: Accelerating Distance Computation on Modern Commodity CPUs

3.1 Introduction

Lots of similarity search algorithms in various applications (e.g., spatial, multimedia, time series) are distance computation intensive. In this chapter, we focus on boosting the performance of distance computation intensive algorithms in time series applications.

Time series data has various applications in medical diagnosis, speech pro-

cessing, climate analysis, financial analysis, etc. It has attracted extensive research in the literature [9, 38, 67, 108, 69, 83, 22, 72].

We illustrate representative problems in Figure 3.1: (a) the *subsequence search* problem, which takes a query sequence q and finds its most similar subsequence t_c of a time series t, (b) the *motif discovery* problem, which reports the most similar pair of subsequences in a time series t, and (c) the kNN *classification* problem.

These problems typically use the Euclidean Distance (ED) and Dynamic Time Warping (DTW) as the similarity measure.

These problems are computation bound rather than disk I/O bound [72]. Many time series algorithms have been evaluated on commodity CPU [9, 38, 67, 108, 69, 83, 22, 72] in single machine. These works focus on devising lower-bound distance functions to prune unpromising candidates and thus reduce calling expensive distance computations.

Even with these effective lower bounds, the above time series problems are still computation intensive, especially for increasingly long time series nowadays (e.g., medical physiological signals[1]). For example, the subsequence search on a trillion scale time series [72] would take 3.1 hours (under the Euclidean distance) and 34 hours (under Dynamic Time Warping) on a commodity PC.

Nevertheless, existing techniques overlook the characteristics of CPU and they have not studied the effect of those characteristics on the CPU time. In general, the CPU time consists of (i) busy cycles, for executing instructions, and (ii) stall cycles, for waiting for instructions or data.

[1] http://www.physionet.org/physiobank/

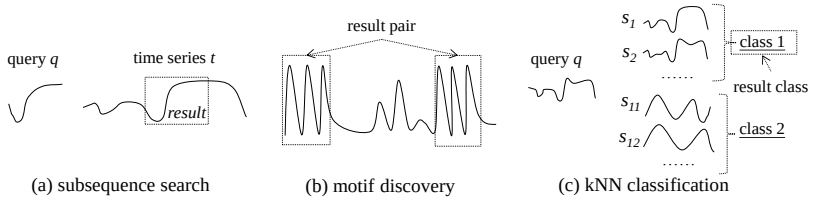


Figure 3.1. Problems on time series data

We raise the following questions:

Q1: "In these algorithms, where does time go?"

To answer this question, we profile the performance [7, 87] of existing time-series algorithms (cf. Section 3.3). Surprisingly, most of the CPU time (70%) is spent on stalling.

Q2: "What cause CPU stall cycles?"

According to our performance profiling, the CPU stall is mainly (more than 80%) caused by branch mispredictions, cache misses, and ALU stall in lower-bound and distance functions.

Q3: "How to reduce CPU stall cycles in modern CPUs?"

Modern CPUs have built-in hardware for branch prediction, caching, and processing vector data efficiently (through SIMD instructions). Recent researches have utilized these characteristics to offer speedup on different problems like join [25], sorting [26], set intersection [49]. In this chapter, we will design efficient implementations for lower-bound and distance functions by exploiting the characteristics of modern commodity CPUs.

Note that our research direction is orthogonal to the development of lower-

bound functions [9, 38, 67, 108, 69, 83, 22, 72]. Besides, our proposed techniques are also applicable to mobile time series applications (e.g., continuous heart rate monitoring on Apple watch) as Apple mobile processors (e.g., A5) have supported advanced SIMD instructions since 2011[2].

Our proposed techniques achieve performance gain through: (i) reducing branch mispredictions and cache misses, (ii) incorporating parallelism for vector processing in our computations. We then elaborate these issues in the following two paragraphs.

Conditional branches (e.g., if-then-else, case statements) are commonly used in the lower-bound and distance functions on time series. With branch prediction, a CPU can speculatively execute one path of a conditional branch. A correct prediction can improve the performance due to the CPU's instruction pipeline. However, if the prediction is wrong (i.e., *branch misprediction*), then many CPU cycles will be wasted to flush the instruction pipeline, flush and fetch the relevant data, and restart the execution for the other branch. Therefore, it is desirable to rewrite algorithms to use fewer branching statements and avoid cache pollution. Also, we need to reduce non-compulsory cache misses brought by random memory accesses in our algorithms.

Data-intensive functions, like lower-bound and distance functions on time series, execute certain arithmetic operations (e.g., multiplication, division) that incur many CPU cycles and thus cause ALU stall. To reduce ALU stall, we use SIMD instructions to process multiple data values per instruction. For example, a SIMD division instruction takes two vectors of values V_a and V_b as input, and

[2] https://en.wikipedia.org/wiki/Apple_mobile_application_processors

perform division $V_a[i]/V_b[i]$ for each position i simultaneously. In this chapter, we present vectorized implementations for lower-bound and distance functions by using SIMD. In addition, our vectorized implementations are designed to avoid using conditional branches.

Besides, our proposed techniques are generic and applicable to many time series problems (e.g., subsequence search, motif discovery, kNN classification).

3.2 Preliminaries

3.2.1 Fundamental Distance Measurement

In this work, we consider two most popular distance functions, i.e., Euclidean Distance (ED) and Dynamic Time Warping (DTW), in time series problems [35, 55, 62, 69, 72, 79]. We follow the suggestion from prior literatures [62, 72] that every subsequence should be Z-normalized in order to capture the similarity between the shapes of the sequences. Formally, the i-th value of a Z-normalized sequence $\hat{q}$ can be calculated by $\hat{q}[i] = \frac{q[i]-\mu_q}{\sigma_q}$, where μ_q and σ_q are the mean and standard deviation of q, respectively, and $q[i]$ indicates the i-th element of q. For ease of presentation, we use $dist(q,t)$ to denote the distance $dist(\hat{q},\hat{t})$ between Z-normalized subsequences in this chapter.

Euclidean Distance: This is the most common similarity metric in time series [35, 62, 72, 83, 108] due to its simplicity. We give the definition of squared

ED[3] in Equation 3.1. It takes $O(m)$ time for a query q of length m.

$$ED(q, t_c) = \sum_{i=1}^{m} (\hat{q}[i] - \hat{t}_c[i])^2 \tag{3.1}$$

Dynamic Time Warping: DTW can capture the similarity of two sequences which may vary in time or have missing values. It is shown to be effective in time series applications [12, 55, 79]. DTW aims to find the optimal alignment (i.e., minimum distance) between two sequences, according to the following recursive equation.

$$DTW(q, t_c) = (\hat{q}[1] - \hat{t}_c[1])^2 + \min \begin{cases} DTW(\hat{q}[2...last], \hat{t}_c) \\ DTW(\hat{q}[2...last], \hat{t}_c[2...last]) \\ DTW(\hat{q}, \hat{t}_c[2...last]) \end{cases} \tag{3.2}$$

where $\hat{q}[2...last]$ denotes the subsequence of $\hat{q}$ containing values from the 2^{nd} to the last offset. To avoid pathological warping (and reduce the computational cost), the literature [72] suggests to limit the warping length r such that $\hat{q}[i]$ can be matched with $\hat{t}_c[j]$ when $|i - j| \leq r$. This reduces the time complexity of DTW from $O(m^2)$ to $O(mr)$.

3.2.2 Time Series Algorithms

In Table 3.1, we summarize the computation techniques (e.g., lower-bounds functions and distance functions) that can be used in three representative time

[3]The squared distance preserves the relative ordering of distances, and it avoids expensive square root calculations.

series problems: subsequence search, motif discovery, and classification. Where LB prefixed function provides a lower bound of the exact distance.

Table 3.1. Computation techniques and distance functions used in time series problems

problem	technique(s)	distance
subsequence search	early distance stop	ED
	$LB_{Kim\mathbf{FL}}, LB^{EQ}_{Keogh}, LB^{EC}_{Keogh}$	DTW
motif discovery	LB_{ref} (uses reference indices)	ED
classification (by kNN)	early distance stop	ED
	$LB_{Kim\mathbf{FL}}, LB^{EQ}_{Keogh}, LB^{EC}_{Keogh}$	DTW

Subsequence search. Formally, given a time series t of length n, a query q of length m, and a distance function $dist(\cdot)$, the subsequence search problem returns a length-m subsequence $t_c \in t$ such that $dist(q, t_c)$ is the minimum (among all length-m subsequences in t).

To the best of our knowledge, UCR Suite [72] is the state-of-the-art solution for the subsequence search problem. It adopts the filter-and-refinement paradigm to reduce exact distance computations. Let bsf be the best-so-far distance obtained during the search process. For ED subsequence search, UCR Suite does not apply any lower-bound function. It accumulates the distance step-by-step and early stops the distance computation $dist(q, t_c)$ as soon as the accumulated value exceeds bsf. For DTW subsequence search, UCR Suite examines each candidate subsequence t_c and applies lower-bound functions on t_c in ascending order of their computation cost: first $LB_{Kim\mathbf{FL}}$, then LB^{EQ}_{Keogh} and finally LB^{EC}_{Keogh}. t_c gets pruned as soon as some $LB(q, t_c)$ exceeds bsf. If t_c survives, then UCR Suite executes the distance function on t_c. We proceed to introduce these lower-bound

functions as follows.

$\underline{LB_{Kim\mathbf{FL}}}$ is derived from the **F**irst and the **L**ast sequence values, taking only $O(1)$ time to compute. It is defined as

$$LB_{KimFL}(q, t_c) = (\hat{q}[1] - \hat{t}_c[1])^2 + (\hat{q}[m] - \hat{t}_c[m])^2 \tag{3.3}$$

$\underline{LB^{EQ}_{Keogh}}$ is derived from the distance between the candidate subsequence $\hat{t}_c$ and the envelop of $\hat{q}$. Given the warping length r, the upper and lower envelop of $\hat{q}$ are defined as $\hat{q}^u[i] = \max_{j=i-r}^{i+r} \hat{q}[j]$ and $\hat{q}^l[i] = \min_{j=i-r}^{i+r} \hat{q}[j]$, respectively, Accordingly, we have

$$LB^{EQ}_{Keogh}(q, t_c) = \sum_{i=1}^{m} \begin{cases} (\hat{t}_c[i] - \hat{q}^u[i])^2 & \text{if } \hat{t}_c[i] > \hat{q}^u[i] \\ (\hat{t}_c[i] - \hat{q}^l[i])^2 & \text{if } \hat{t}_c[i] < \hat{q}^l[i] \\ 0 & \text{otherwise} \end{cases} \tag{3.4}$$

$\underline{LB^{EC}_{Keogh}}$ is derived similarly to LB^{EQ}_{Keogh} but the lower-bound is derived from the distance between the query and the envelop of $\hat{t}_c$ (i.e., switching roles).

Motif Discovery. Formally, given a time series t of length n, and a query length m, the motif discovery problem returns a pair of length-m subsequences $t_c, t'_c \in t$ such that the Euclidean distance $ED(t_c, t'_c)$ is the minimum among all pairs.

MK [67] is a representative solution for motif discovery. It provides the first

non-trivial algorithm to discover exact motifs in time series dataset [4]. To avoid examining every subsequence pair, it proposes a reference based lower-bound. Given a set of subsequences and their distances to a set of references R, the lower-bound of two subsequences t_a and t_b can be derived as follows.

$$LB_{ref}(t_a, t_b) = \max_{r_i \in R} |distRef[r_i][t_a] - distRef[r_i][t_b]| \tag{3.5}$$

where $distRef[r_i][t] = ED(r_i, t)$.

MK first constructs a sorted list of every subsequence in terms of their distances to a reference. Intuitively, if the lower-bound of every 1^{st} neighbor pair (in terms of their positions in the sorted list) is worse than bsf, then it is not necessary to examine further neighbor pairs (e.g., 2^{nd} neighbor pairs) due to the monotonicity of the sorted list. Thereby, MK iteratively examines the subsequence pairs based on their sorted list positions. At the end of an iteration, the search terminates when no neighbor pair has lower-bound better than bsf.

Classification. ED and DTW are widely accepted for describing the similarity between time series in the classification problem [33]. We can apply the same techniques for subsequence search (i.e., early distance stop for ED and lower-bound techniques for DTW) to boost the classification process.

3.3 Profiling of Algorithms

We first describe our experimental platform and then present the profiling result on existing time series algorithms.

[4] http://www.cs.ucr.edu/~eamonn/exact_motif/

3.3.1 Experimental Setting

In all experiments, we use a machine with a 3.40GHz Intel(R) Core(TM) i7-4770 CPU based on Haswell micro-architecture, 16 GB main memory, and a SSD (solid state drive, 256GB capacity, 545 MB/s sequential read throughput). The CPU has 4 physical cores and supports simultaneous multithreading. The machine runs Ubuntu 14.04. All algorithms have been implemented in C++ and compiled by GNU C++ compiler with level 3 optimization.

We use the following real datasets and list their information in Table 4.2. All datasets are stored in the SSD.

- For the subsequence search problem, we use three datasets. Both **ECG-E**[5] and **ECG-L**[6] are electrocardiography (ECG) recordings, and we use the same query sequences (of length 421) as in [72] as the default query sequences. **EEG-C**[7] contains electroencephalography (EEG) recordings, and we randomly extract query sequences (of length 128) from the epileptic seizure recording as in [84]. For each dataset, we follow the experimental methodology in [72], and obtain a single time series by concatenating all data sequences.

- For the kNN classification problem, we use **Weather** [8] dataset, which contains the temperature data extracted from weather forecast records. It contains 11,508 sequences, each sequence in Weather corresponds to a one-year time series collected from 5,936 locations. We use the attribute "Country" as the class attribute. We randomly choose data sequences as queries and exclude

[5] http://www.physionet.org/physiobank/database/edb/
[6] http://www.physionet.org/physiobank/database/ltstdb/
[7] http://www.physionet.org/pn6/chbmit/
[8] http://data.gov.uk/metoffice-data-archive

Table 3.2. Dataset information

dataset	sequence length	data size	problem
ECG-E	$1.60 \cdot 10^8$	611 MB	subsequence search
ECG-L	$1.89 \cdot 10^9$	7.06 GB	
EEG-C	$1.01 \cdot 10^{10}$	37.5 GB	
EEG-MK	$1.80 \cdot 10^5$	704 KB	motif discovery
TAO-MK	$7.42 \cdot 10^5$	2.82 MB	
Weather	$1.81 \cdot 10^3$,	19.86 MB	kNN classification

them from the data.

- For the motif discovery problem, we use two datasets: **EEG-MK**[9] and **TAO-MK** [62].

3.3.2 Measurement Methodology

Program execution time: According to the Intel performance analysis manual [3], the program execution time (T_R) consists of: computation time (T_C), branch misprediction stall (T_{Br}), backend stall (T_{Be}), and frontend stall (T_{Fe}). The computation time (T_C) is regarded as '*CPU busy*', and the rest as '*CPU stall*'. The backend stall occurs when the requested resource is being held-up in back end. It includes ALU stall (T_{ALU}) and memory stall (T_{Cache}). T_{ALU} is the ALU execution unit stall, which is caused by the execution of arithmetic operations (e.g., divide, square root) that require many cycles. T_{Cache} is the memory-bound stall, which is caused by L1 data cache misses, L2 cache misses, L3 cache misses or TLB cache misses.

We summarize the breakdown of execution time in a CPU as follows:

[9] http://www.cs.ucr.edu/∼mueen/OnlineMotif/index.html

$$T_R = T_C + T_{stall}; \quad \text{where} \quad T_{Stall} = T_{Br} + T_{ALU} + T_{Cache} + T_{Fe}$$

Profiling experiments: To measure the above components of CPU time, we used PAPI [21] to obtain hardware performance counters from CPU, e.g., the number of stall cycles and the number of CPU cycles. In each subsequence search and classification experiment, we report the average CPU time over 10 queries. To ensure the confidence level, we repeat running each query until the maximum standard deviation of the important counters (UOPS_RETIRED:RETIRE_SLOTS, CPU_CLK_UNHALTED:THREAD_P) is less than 3%.

Experimental reproducibility: For the sake of experimental reproducibility, we have posted the datasets and source codes at [6] [10].

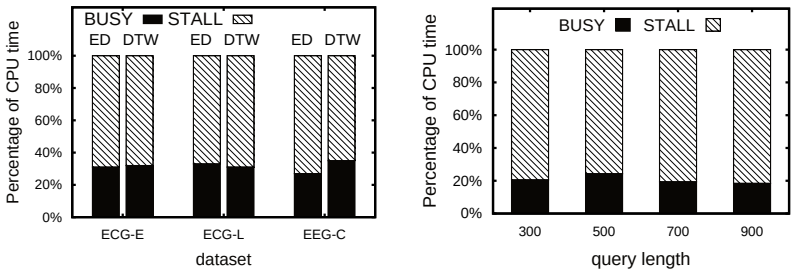
3.3.3 Identifying the Performance Bottleneck

In this section, we profile the performance of existing solutions and then identify the performance bottleneck. We conduct experiments to profile the performance of representative solutions: (i) UCR Suite [72] for the subsequence search problem, (ii) MK [67] for the motif discovery problem, and (iii) kNN classification [72] for the classification problem.

CPU stall & CPU busy: Figures 3.2(a) and (b) report the CPU time breakdown of existing solutions into *busy* time and *stall* time, for subsequence search and motif discovery, respectively.

[10] For consistency, we use the 'float' data type to represent time series values in all evaluated methods.

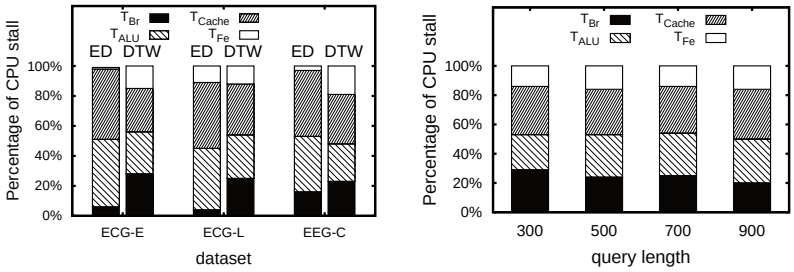
Observation: The majority (65–70%) of the CPU time is spent on stalling (i.e., wasted CPU cycles).



(a) subsequence search (UCR Suite) (b) motif discovery (MK)

Figure 3.2. Busy vs. Stall time

CPU stall breakdown: We then delve into CPU stall and plot the breakdown of CPU stall time in Figures 3.3(a) and (b).



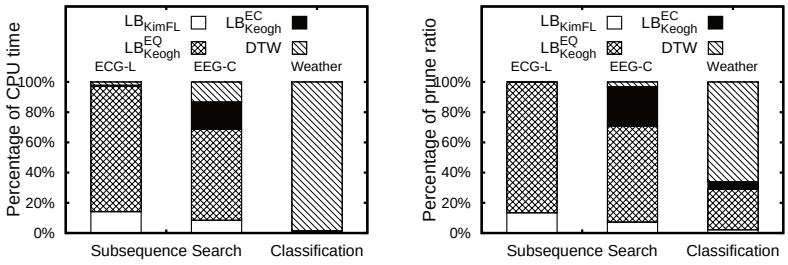
(a) subsequence search (UCR Suite) (b) motif discovery (MK)

Figure 3.3. Profiling CPU stall

Observation: The CPU stall is dominated (more than 80%) by ALU stall, cache misses, and branch mispredictions penalties.

CPU time of different functions: The DTW function and its lower-bound functions ($LB_{Kim\mathbf{FL}}, LB^{EQ}_{Keogh}, LB^{EC}_{Keogh}$) are applicable to the subsequence search problem and the classification problem [72]. We profile the performance of [72] on

these two problems in Figure 3.4(a). Different functions incur different portions of time and pruning ratio (cf. Figure 3.4(b)) in different scenarios. For example, lower-bound functions $LB^{EQ}_{Keogh}, LB^{EC}_{Keogh}$ dominate the time for subsequence search. However, the DTW computation incurs more time in kNN classification problems.



(a) CPU time (b) pruning ratio

Figure 3.4. Profiling existing solutions on DTW subsequence search and classification

Observation: Different time series problems spend very different proportions of time on different functions. Therefore, it is important to optimize the computation of both lower-bound functions $LB^{EQ}_{Keogh}, LB^{EC}_{Keogh}$ and the DTW function.

3.4 Accelerating Distance Functions with SIMD

As shown in the previous section, the majority of CPU stall is caused by ALU stall, cache misses and branch mispredictions. In this section, we will design vectorized implementations for exact distance and lower-bounds functions to reduce those stalls. We will also evaluate the efficiency of our implementations with experiments.

3.4.1 How do SIMD Instructions Reduce Stall?

3.4.1.1 SIMD vectorization: reduce ALU stall

The ALU stall is caused by the execution of arithmetic operations that require many CPU cycles. For example, the 'division' instruction for two floating-point values takes 24 CPU cycles [3].

Modern CPU provides SIMD instructions to perform the same instruction (e.g, $+, -, \times, /, \min, \max$) on multiple data values in parallel. For instance, Intel i7-4770 and AMD Phenom II support the AVX2 instruction set (SIMD instructions on 256-bit registers). The SIMD instruction `simd_div` (e.g., `_mm256_div_ps` in AVX2) performs division on 8 pairs of values in two SIMD registers R_a and R_b simultaneously. It takes only 21 CPU cycles [3], which is much cheaper than executing the 'division' instruction on 8 pairs one-by-one (using 24*8=192 cycles). Thus, SIMD instructions help reduce the ALU stall significantly.

Distance computation indeed fits well with SIMD instructions. As we illustrate in Figure 3.5, we may divide subsequences into groups of length 8, and then apply SIMD instructions on each group to compute distances for pairs.

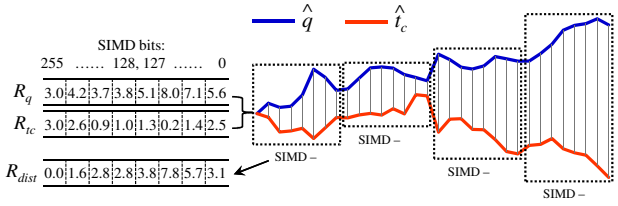


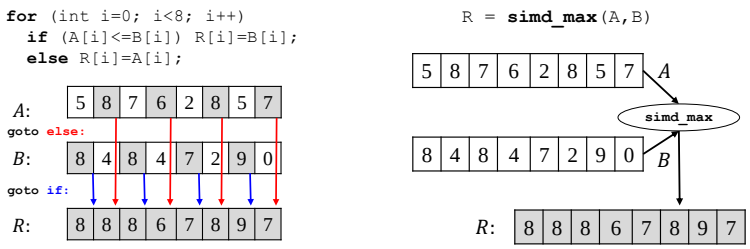
Figure 3.5. Using SIMD for distance computation

Typical SIMD width: Our CPU (Intel i7-4770) is a modern commodity CPU. It supports the following SIMD widths and instruction sets: (i) 64 bits (i.e., MMX

instruction set), (ii) 128 bits (i.e., SSE instruction set), (iii) 256 bits (i.e., AVX instruction set). Since the MMX instruction set does not support floating-point values, it cannot be used in time series problems. Thus, we report the results for 128 bits (SIMD-128) and 256 bits (SIMD-256) in following experiments. For simplicity, we set 256 bits (SIMD-256) as default SIMD register.

3.4.1.2 Hardware prefetching: reduce branch misprediction

Modern CPU is equipped with a branch prediction unit and it speculatively executes a conditional branch to maximize the utilization of CPU resources. A correct prediction can improve the performance due to the built-in instruction pipeline and hardware prefetching. However, incorrect prediction will bring cache pollution[11] and waste CPU cycles to flush instructions and restart execution.



(a) code fragment and schematic for if-else statement (b) code fragment and schematic for SIMD max instruction

Figure 3.6. Example for reducing branching statements

Some SIMD instructions help reduce branch misprediction. For example, for the code fragment in Figure 3.6(a), the CPU may incur up to 8 branch mispredictions in the worst case. In contrast, the alternative implementation in Figure 3.6(b) has no branch mispredictions because it uses a single instruction

[11] http://en.wikipedia.org/wiki/Cache_pollution

`simd_max` instead of conditional branches.

We observe that DTW and its lower-bound functions (cf. Section 3.2) have many conditional branches. Therefore, we need to design SIMD implementations for DTW and its lower-bound functions without using conditional branches.

3.4.2 Accelerating ED with SIMD

Before presenting our SIMD solutions, we first introduce the existing implementation of Euclidean distance. We call it as SISD-ED (cf. Algorithm 3.1) because it uses traditional CPU instructions, i.e., *Single Instruction, Single Data* (SISD). According to Section 3.2, we perform Z-normalization on the subsequence t_c (cf. Line 3). It early stops the computation if the accumulated distance $dist$ exceeds the best-so-far distance bsf (cf. Line 5).

Algorithm 3.1 SISD-$ED(q, t_c)$

Input: best-so-far bsf, mean μ and stdev. σ of candidate t_c,
Output: squared distance $dist$

1: $dist := 0$
2: **for** $i := 1$ to m **do**
3: $\quad c := (t_c[i] - \mu)/\sigma$ ▷ Z-normalization
4: $\quad dist := dist + (c - \hat{q}[idx])^2$ ▷ accumulation
5: $\quad$ **if** $dist \geq bsf$ **break** ▷ early stop
6: return $dist$

Next we demonstrate how we employ SIMD to accelerate $ED(\cdot)$ in different steps. The intuition is to compute 8 offsets between q and t_c by batch. In the Z-normalization step, we can normalize 8 offset values simultaneously as follows.

SIMD Z-normalization

1: R_c := `simd_load`$(\&t_c[i])$ ▷ load t_c

2: R_c := `simd_sub`(R_c, R_μ) ▷ vectorized $t_c[i] - \mu$

3: R_c := `simd_div`(R_c, R_σ) ▷ vectorized $(t_c[i] - \mu)/\sigma$

where R_c, R_μ, R_σ are the corresponding SIMD registers of variables c, μ, and σ, respectively. Note that each register stores 8 floating-point values. In the accumulation step, we can compute the distance of 8 offsets $(\hat{t}[i] - \hat{q}[i])^2$ as follows.

SIMD distance computation

1: $R_{\hat{q}}$:= `simd_load`$(\&\hat{q}[i])$ ▷ load $\hat{q}[0]...\hat{q}[7]$

2: R_d := `simd_sub`$(R_{\hat{q}}, R_c)$ ▷ vectorized $\hat{t}[i]-\hat{q}[i]$

3: R_d := `simd_mul`(R_d, R_d) ▷ vectorized $(\hat{t}[i] - \hat{q}[i])^{\mathbf{2}}$

Before examining the early stop condition (cf. Line 5 of Algorithm 3.1), we need to accumulate 8 offset distances into *dist*. Since the AVX2 instruction set has no single instruction to accumulate the values of an SIMD register, we accomplish the accumulation by the following sequence of SIMD instructions.

SIMD distance accumulation

1: R_d := `simd_hadd`(R_d, R_d) ▷ add horizontal pairs

2: R_d := `simd_hadd`(R_d, R_d) ▷ add horizontal pairs

3: S_d := `simd_extractf`$(R_d, 1)$

4: S_d := `simd_sadd`(`simd_cast`$(R_d), S_d)$

5: $dist$:= $dist$+ `simd_scvt`(S_d)

The accumulation employs instruction `simd_hadd` (e.g., `_mm256_hadd_ps`) twice that horizontally adds adjacent pairs of 32-bit floating-point elements in the input registers, and stores the results into an output register. Then decompose the vector into two parts by `simd_extractf` and `simd_cast`. Next, we sum the first value of two decomposed vectors (by `simd_sadd`), extract the lower 32-bit floating-point element from the vector (by `simd_scvt`), and accumulate it into *dist*. The accumulation process takes logarithmic cost to the SIMD register length.

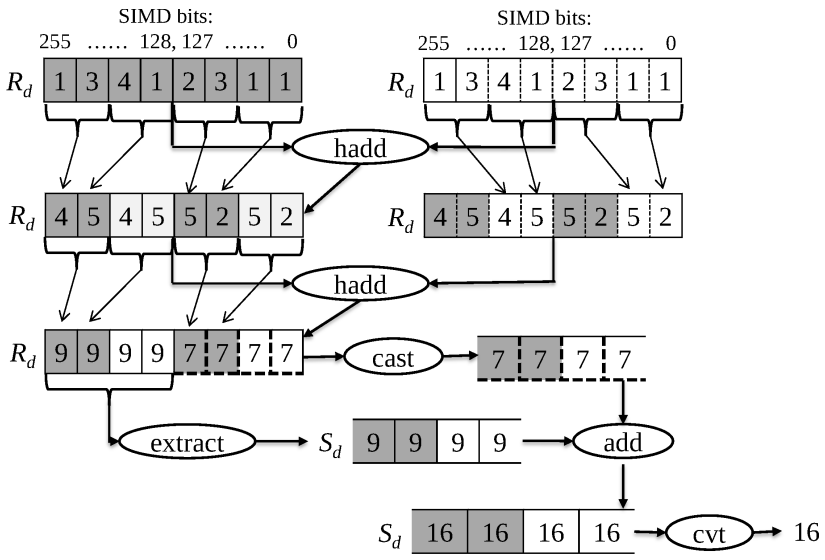
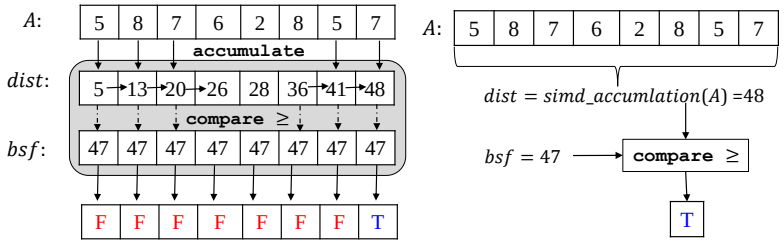


Figure 3.7. Horizontal accumulation

We illustrate the accumulation in Figure 3.7, with the initial content of $R_d : \langle 1, 3, 4, 1, 2, 3, 1, 1 \rangle$ in the example. We first execute `simd_hadd` twice to obtain $R_d : \langle 9, 9, 9, 9, 7, 7, 7, 7 \rangle$ and then decompose the vector into two parts by `simd_extractf` and `simd_cast`. Next, we sum the first value of two decomposed vectors (i.e., 9 and 7) by `simd_sadd`, extract the lower 32-bit floating-point element from the vector (i.e., 16) by `simd_scvt`, and accumulate it into *dist*. The

accumulation process takes logarithmic cost to the SIMD register length. Our



(a) SISD-ED (b) SIMD-ED

Figure 3.8. Example for early stop

vectorized implementation reduces CPU cycles by (i) incorporating parallelism for Z-normalization and distance computation, and (ii) reducing branching statements for the early stop condition. Figure 3.8(a) shows that SISD-ED requires verifying the early stop for every accumulation (i.e., 8 comparisons in total). In SIMD-ED, we only verify the early termination once per 8 accumulations as shown in Figure 3.8(b).

Cost analysis: We proceed to analyze the cost of the SISD and SIMD implementations based on the latency cycle information given in the Intel architecture optimization manual [3].

Table 3.3. Instruction latency of SISD- and SIMD-ED

Step		SISD-ED	SIMD-ED
Z-norm.	op	`load` $\hat{t}_c[i]$, $-$, $/$	`setr`, `sub`, `div`
	cost	1+3+24 = 28	(1+3+21)/8 = 25/8
Distance computation	op	`load` $\hat{q}[i]$, $-$, $\times$	`loadu`, `sub`, `mul`
	cost	1+3+5 = 9	(4+3+5)/8 = 12/8
Accumulation	op	+	2·`hadd`, 2·`add`, `extractf`, `cvtss`, `cast`
	cost	3	(2*5+2*3+1+1+0)/8 = 18/8
Early stop	cost	1	1/8
Total	cost	41	7

Our analysis covers four steps in ED: (i) Z-normalization, (ii) distance computation, (iii) distance accumulation, and (iv) early stop, as illustrated in Table 3.3. In each step, we list all used instructions and their latency cycles. For SIMD-ED, the denominator in latency is 8 as it processes 8 offset values simultaneously. In summary, SIMD-ED is $41/7 = 5.86$ times faster than SISD-ED.

3.4.3 Accelerating DTW with SIMD

For the sake of our discussion, we first present the pseudo code of DTW computation in Algorithm 3.2. It employs a matrix $C[1..m][1..m]$ whose entry $C[i][j]$ is used to store the DTW value between subsequences $\hat{q}[1..i]$ and $\hat{t_c}[1..j]$. Then, we fill the matrix C by row-by-row ordering (Lines 2–3). Observe that we cannot compute values in the same row (e.g., $C[i][j-1], C[i][j]$) in parallel because $C[i][j]$ depends on $C[i][j-1]$.

Algorithm 3.2 SISD-$DTW(q, t_c)$

Input: warping constraint length r, normalized query $\hat{q}$ and candidate $\hat{t}_c$
Output: squared distance *dist*
1: Distance array $C[1..m][1..m]$, initialized to $+\infty$
2: **for** $i := 1$ to m **do**
3: **for** $j := \max(0, i-r)$ to $\min(m, i+r)$ **do**
4: **if** $i = 1$ and $j = 1$ **then**
5: $C[1][1] := (\hat{q}[1] - \hat{t}_c[1])^2$
6: **else**
7: $C[i][j] := (\hat{q}[i] - \hat{t}_c[j])^2 + \min(C[i-1][j], C[i-1][j-1], C[i][j-1])$
8: return $C[m][m]$ as *dist*

To better utilize SIMD instructions, we rewrite the equation of $C[i][j]$ into

an alternative form as follows.

$$C[i][j] = (\hat{q}[i] - \hat{t}_c[j])^2 + \min(B_{i-1}[j], C[i][j-1]) \tag{3.6}$$

where $B_{i-1}[j] = \min(C[i-1][j-1], C[i-1][j])$. Since $B_{i-1}[j]$ depends only on values in the previous row of C (i.e., row $i-1$), we can calculate consecutive values of B_{i-1} (e.g., $B_{i-1}[j]$ to $B_{i-1}[j+7]$) in a batch.

The above discussion enables us to rewrite Line 7 in SISD-DTW as the following pseudo code using SIMD instructions.

Rewrite inner for-loop (i fixed) in Algorithm 3.2

1: $j_{min} := \max(0, i-r); j_{max} := \min(m, i+r)$
2: **for** $j := j_{min}$ to j_{max}, 8 offsets **do**
3: load R_{x1} with $C[i-1][j, \cdots, j+7]$
4: load R_{x2} with $C[i-1][j-1, \cdots, j+6]$
5: $R_B :=$ `simd_min`(R_{x1}, R_{x2})
6: $B_{i-1}[j, \cdots, j+7] =$ `simd_store`(R_B)
7: $C[i][j] := (\hat{q}[i] - \hat{t}_c[j])^2$
8: $C[i][j_{min}] := C[i][j_{min}] + B_{i-1}[j_{min}]$
9: **for** $j := j_{min} + 1$ to j_{max} **do**
10: increment $C[i][j]$ by $\min(C[i][j-1], B_{i-1}[j])$

The rewritten code has two nice properties: (i) avoid branch mispredictions by using the `simd_min` instruction (Lines 3–5), (ii) reduce cache misses by utilizing the data locality of $C[i][j-1]$ and $C[i][j]$ (Line 10). Figure 3.9 illustrates how our SIMD implementation works (when $i = 4$).

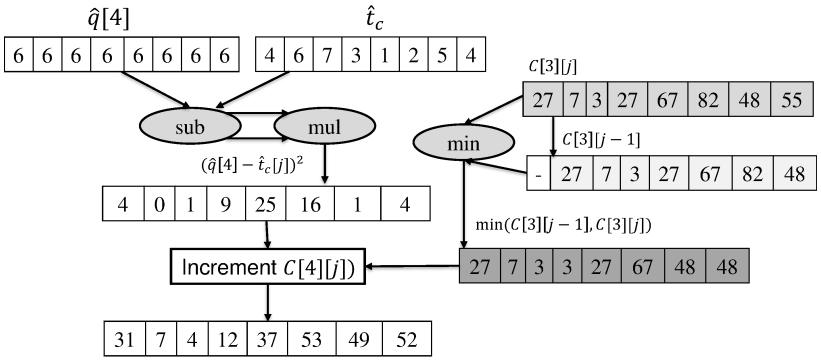


Figure 3.9. SIMD DTW illustration, at $i = 4$

Cost analysis: We analyze the latency of both SISD and SIMD implementations of DTW in Table 3.4. The speedup of the SIMD implementation over SISD one is: $\frac{48}{14.625} = 3.28$.

Table 3.4. Instruction latency of SISD- and SIMD- *DTW*

Step		SISD-*DTW*	SIMD-*DTW*
ED (c.f. Table 3)	cost	28+9+3	(25+12+18)/8=55/8
Take Minimum	op cost	3*`load`, 2*`cmp` 3*1+2=5	2*`loadu`,`min`,`storeu` (2*4+3+3)/8 = 14/8
Accumulation	op cost	+ 3	2*`load`, 1*`cmp`, + 2*1+1+3 = 6
Total		48	14.625

Optimized implementation: For ease of understanding, we employ $m \times m$ matrices in the above algorithms. An optimized implementation is to use 2 float arrays with size $2r + 1$ (i.e., store $C[i-1]$ and $C[i]$ in Line 7, Algorithm 3.2) to compute DTW (for both SISD-DTW and SIMD-DTW). Since these 2 float arrays can fit in low latency cache (e.g., L2 cache rather than L3 cache), we use this optimized implementation for both SISD-DTW and SIMD-DTW in our

code.

3.4.4 Accelerating Lower Bounds for DTW with SIMD

We proceed to present SIMD optimizations for lower-bound functions LB^{EQ}_{Keogh} and LB^{EC}_{Keogh}. Since these two functions are similar, our discussion focuses on LB^{EQ}_{Keogh}.

Algorithm 3.3 SISD-$LB^{EQ}_{Keogh}(q, t_c)$

Input: best-so-far bsf, mean μ and stdev. σ of candidate t_c, upper and lower envelops $\hat{q}^u$ and $\hat{q}^l$
Output: lower-bound distance lb

```
1: lb := 0
2: for i := 1 to m do
3:     c := (t_c[i] − μ)/σ                          ▷ Z-normalization
4:     if q̂^u[i] < c then                ▷ distance of t̂_c and the envelop of q̂
5:         lb := lb + (c − q̂^u[i])^2
6:     else if q̂^l[i] > c then
7:         lb := lb + (q̂^l[i] − c)^2
8:     if dist ≥ bsf break                          ▷ early stop
9: return lb
```

Similar to $ED(\cdot)$, we present the SISD implementation of LB^{EQ}_{Keogh} in Algorithm 3.3. It derives the lower-bound lb from the candidate subsequence t_c and the envelop of q which is handled by the **if-then-else** statement at Lines 4-7. However, the **if-then-else** statement may cause many branch mispredictions in CPU, leading to high stalling time (e.g., 10–20 clock cycles in modern CPU on average). In addition, as reported in [45], the hardware prefetching (for reducing cache misses) technique becomes less effective in the presence of multiple code paths.

To avoid branch mispredictions and better utilize hardware prefetching, we should remove branching, i.e., the **if-then-else** statement, in Algorithm 3.3. Lemma 3.1 shows the alternative form of LB^{EQ}_{Keogh} (cf. Equation 3.4 in Section 3.2).

Lemma 3.1 (Alternative form of LB^{EQ}_{Keogh})

$$LB^{EQ}_{Keogh} = \sum_{i=1}^{m}((\hat{t}_c[i] - \min\{\hat{t}_c[i], \hat{q}^u[i]\}) + (\max\{\hat{t}_c[i], \hat{q}^l[i]\} - \hat{t}_c[i]))^2$$

Proof. LB^{EQ}_{Keogh} (cf. Equation 3.4) consists of three cases.

Case 1: When $\hat{t}_c[i] < \hat{q}^u[i]$, the first part (i.e., $\hat{t}_c[i] - \min\{\hat{t}_c[i], \hat{q}^u[i]\}$) becomes zero so that the equation reduces to $(\hat{q}^l[i] - \hat{t}_c[i])^2$.

Case 2: When $\hat{t}_c[i] > \hat{q}^l[i]$, the second part (i.e., $\max\{\hat{t}_c[i], \hat{q}^l[i]\} - \hat{t}_c[i]$) becomes zero so that the equation reduces to $(\hat{t}_c[i] - \hat{q}^u[i])^2$.

Case 3: Otherwise, none of the first or the second part contributes so the equation returns 0. □

Since this form uses only $\min, \max, +, -, \times$, we can readily implement them by the corresponding SIMD instructions. Accordingly, the first part and the second part of LB^{EQ}_{Keogh} can be computed as follows. Then, we sum up both parts.

SIMD $\hat{t}_c[i] - \min\{\hat{t}_c[i], \hat{q}^u[i]\}$ computation, 8 offsets

1: $R_{\hat{q}^u} :=$ `simd_load`$(\&\hat{q}^u[i])$ ▷ vectorized load $\hat{q}^u[i]..\hat{q}^u[i+7]$

2: $R_{du} :=$ `simd_min`$(R_{\hat{q}^u}, R_c)$ ▷ vectorized $\min\{\hat{t}_c[i], \hat{q}^u[i]\}$

3: $R_{du} :=$ `simd_sub`$(R_c, R_{\hat{q}^u})$ ▷ vectorized $\hat{t}_c[i] - \min\{\hat{t}_c[i], \hat{q}^u[i]\}$

SIMD $\max\{\hat{t}_c[i], \hat{q}^l[i]\} - \hat{t}_c[i]$ computation, 8 offsets

1: $R_{\hat{q}^l} :=$ `simd_load`$(\&\hat{q}^l[i])$ ▷ vectorized load $\hat{q}^l[i]..\hat{q}^l[i{+}7]$

2: $R_{dl} :=$ `simd_max`$(R_{\hat{q}^l}, R_c)$ ▷ vectorized $\max\{\hat{t}_c[i], \hat{q}^l[i]\}$

3: $R_{dl} :=$ `simd_sub`$(R_{\hat{q}^l}, R_c)$ ▷ vectorized $\max\{\hat{t}_c[i], \hat{q}^l[i]\} - \hat{t}_c[i]$

SIMD combining the result of R_{du} and R_{dl}, 8 offsets

1: $R_d :=$ `simd_add`(R_{du}, R_{dl}) ▷ vectorized sum

2: $R_d :=$ `simd_mul`(R_d, R_d) ▷ vectorized square

We illustrate our idea by a concrete example in Figure 3.10. First we extract the *min* values between $\hat{q}^u$ and $\hat{t}_c$ of 8 offsets by `simd_min` and then store them into R_{du}. Next we subtract R_{du} from t_c to finish the first part computation. The second part is performed similarly where the *max* values are stored into R_{dl}. Next we combine the distance values from R_{du} and R_{dl} to produce R_d. Finally we multiply the values of R_d to generate the squared distance, and then execute SIMD distance accumulation as described in Section 3.4.2.

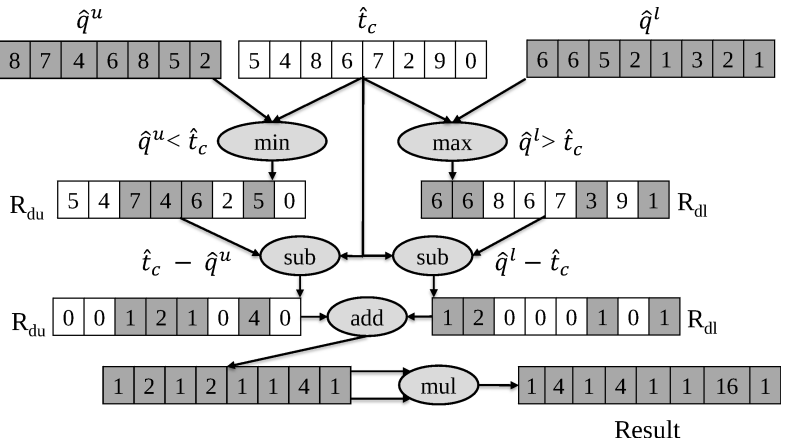


Figure 3.10. LB^{EQ}_{Keogh} **SIMD illustration**

Cost analysis: We analyze the latency for two implementations of LB^{EQ}_{Keogh}. We only list the detail cost at step (ii) distance computation in Table 3.5 as the other three steps are the same as in SIMD-ED (cf. Table 3.3). SIMD-LB^{EQ}_{Keogh} outperforms SISD-LB^{EQ}_{Keogh} by $43/9 = 4.78$ times.

Step		SISD-LB^{EQ}_{Keogh}	SIMD-LB^{EQ}_{Keogh}
Dist. comp.	op	`load` $\hat{q}^u[i]$ or $\hat{q}^l[i]$	2·`loadu`, 2·`sub`,
		2·`cmp`, $-$, $\times$	`min`, `max`, `add`, `mul`
	cost	1+2+3+5=11	(8+6+3+3+3+5)/8=28/8
Z-norm. Accumulation Early stop (cf. Table 3.3)	cost	28+3+1	(25+18+1)/8
Total		43	9

Table 3.5. Instruction latency of SISD- and SIMD- LB^{EQ}_{Keogh}

3.4.5 Accelerating Reference Index with SIMD

Before proposing our SIMD solution, we first present the existing implementation of LB_{ref} in Algorithm 3.4.

Algorithm 3.4 SISD-$LB_{ref}(t_a, t_b)$

Input: best-so-far bsf, reference distance $dist_{ref}$, # of reference R, two subsequences t_a, t_b
Output: Boolean value

```
1: for i := 1 to R do
2:     if |dist_ref[i][a] − dist_ref[i][b]| > bsf then
3:         return true                          ▷ can be pruned
4: return false                                 ▷ cannot be pruned
```

As the absolute value computation is not supported by the AVX2 instruction

set, we rewrite $|dist_{ref}[i][a] - dist_{ref}[i][b]|$ as:

$$\max(dist_{ref}[i][a], dist_{ref}[i][b]) - \min(dist_{ref}[i][a], dist_{ref}[i][b])$$

Then, we design the SIMD implementation below for LB_{ref}. It avoids using branching statements for early termination in Lines 7-8. We verify the early stop only once by executing `simd_cmp` for 8 pairs of candidates.

SIMD LB_{ref}, for 8 offsets

1: $R_a :=$ `simd_load`$(dist_{ref}[i][a], \cdots, dist_{ref}[i+7][a])$
2: $R_b :=$ `simd_load`$(dist_{ref}[i][b], \cdots, dist_{ref}[i+7][b])$
3: $R_{bsf} :=$ `simd_set1`(bsf)
4: $R_{max} :=$ `simd_max`(R_a, R_b)
5: $R_{min} :=$ `simd_min`(R_a, R_b)
6: $R_{sub} :=$ `simd_sub`(R_{max}, R_{min})
7: $R_a :=$ `simd_cmp`$(R_{sub}, R_{bsf}, >)$
8: return `simd_testz`(R_a, R_a)

We can further optimize LB_{ref} by sequentializing the memory access (and reducing CPU cache misses). This requires changing the memory layout of $dist_{ref}$ to $dist_{ref}[a][i]$ (i.e., swapping the role of rows and columns) so that Lines 1–2 have sequential main memory accesses.

Cost analysis: For each reference point i, SISD-LB_{ref} takes 6 cycles and SIMD-LB_{ref} takes 27/8 cycles. We omit the detailed analysis here.

Alternative implementation: Another implementation for $|dist_{ref}[i][a] - dist_{ref}[i][b]|$ is to use `simd_sub`, `simd_set` and `simd_andnot` instructions only. Since this im-

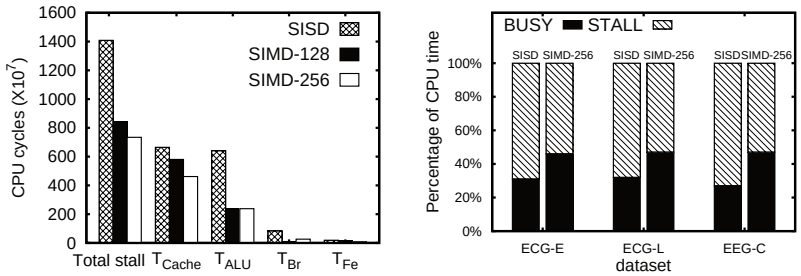
plementation spends the same number of CPU cycles as in the above algorithm, we omit its detail discussion in following experiments.

3.5 Experimental Study

In this section, we conduct extensive experiments to evaluate our proposed techniques with existing solutions. Unless otherwise stated, we use the experimental platform and measurement methodology in Section 3.3. Note that the execution time includes both disk I/O time and CPU computation time. We denote SISD as the original implementation [72, 67] (for corresponding problems), SIMD as the implementation with our proposed techniques.

3.5.1 Subsequence Search

UCR-ED: UCR-ED [72] is a representative solution for the ED-based subsequence search. It employs the *early abandoning* technique to accelerate the Euclidean distance computation. We show the performance of SISD-based and SIMD-based UCR-ED in Figure 3.11.



(a) stall breakdown on ECG-E (b) CPU time breakdown

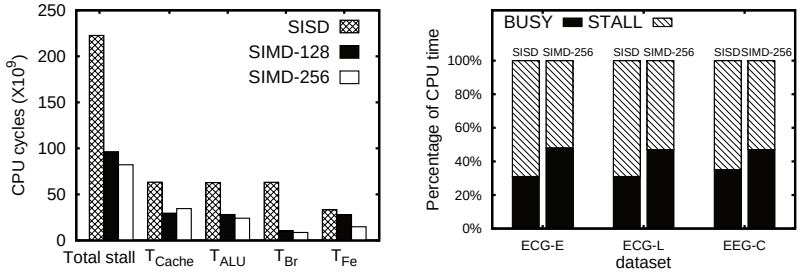
Figure 3.11. SISD-based and SIMD-based UCR-ED

First, we investigate the components of CPU stall of the methods on the dataset ECG-E in Figure 3.11(a). Since our SIMD-based solutions exploit SIMD vectorization techniques, they incur fewer instructions and ALU stall (T_{ALU}) than the SISD-based solution. The results on other datasets are similar to Figure 3.11(a), so we omit them for space reasons.

Second, we compare the CPU time of the methods in Figure 3.11(b). We omit the results of SIMD-128, as it is similar to SIMD-256. Clearly, our SIMD-based UCR-ED can reduce CPU stalls significantly (e.g., ∼20%).

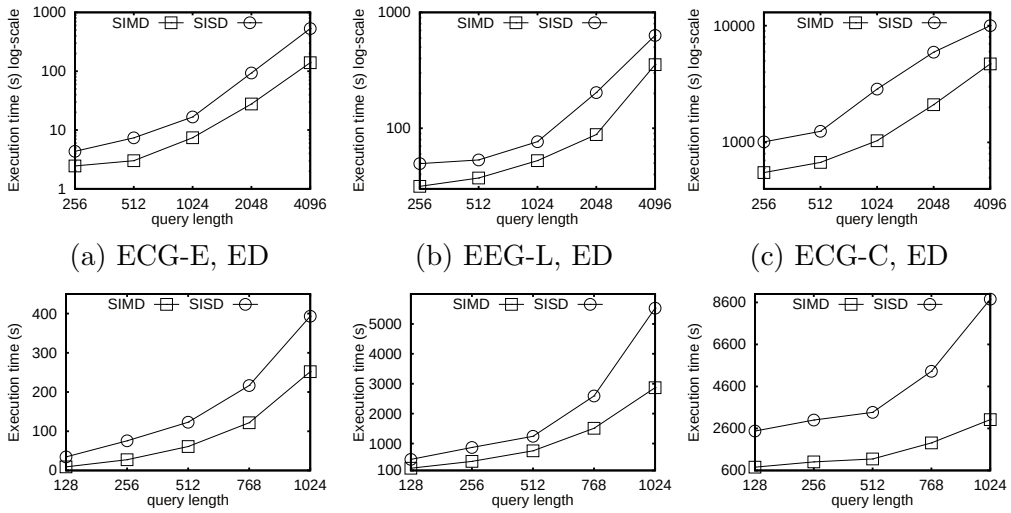
UCR-DTW: Regarding the DTW-based subsequence search, UCR-DTW [72] cascades three lower bound techniques (i.e., LB_{KimFL}, LB^{EQ}_{Keogh}, and LB^{EC}_{Keogh}) to pruning unpromising candidates without invoking expensive DTW computations. We breakdown the components of CPU stall of the methods on the dataset ECG-E in Figure 3.12(a). Since our SIMD-based UCR-DTW accelerated both the exact distance (cf. Section 3.4.3) and the lower bound computations (cf. Section 3.4.4), SIMD-based UCR-DTW introduces fewer CPU stall cycles than the SISD-based solution. Second, we compared the CPU time of the methods on three datasets in Figure 3.12(b). The CPU busy ratio of SIMD-based UCR-DTW is almost 50%, which is much higher than SISD-based UCR-DTW.

Execution Time Speedup: In this set of experiments, we report the execution time of the methods on three time series datasets (i.e., ECG-E, ECG-L, and EEG-C) varying on query lengths in Figure 3.13, where the lengths are from 256 to 4096 in UCR-ED and 128 to 1024 in UCR-DTW. Our proposed SIMD-based methods are 1.8-3.8 and 1.5-3.2 times faster than UCR-ED and UCR-DTW, respectively.



(a) stall breakdown on ECG-E (b) CPU time breakdown

Figure 3.12. SISD-based and SIMD-based UCR-DTW



(a) ECG-E, ED (b) EEG-L, ED (c) ECG-C, ED

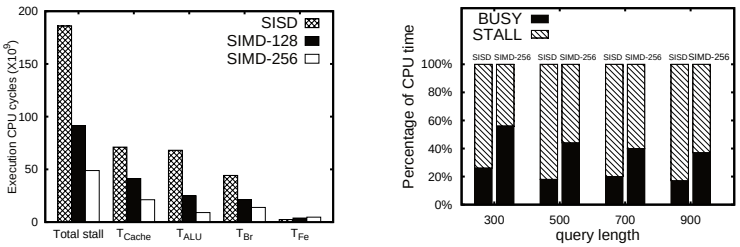
(d) ECG-E, DTW (e) ECG-L, DTW (f) EEG-C, DTW

Figure 3.13. [Subsequence search] vary query length

3.5.2 Motif Discovery

MK [67] makes use of (i) the exact distance calculation ED and (ii) the lower-bound calculation LB_{ref}. The SIMD-based implementation of these two functions has been introduced in Section 3.4.2 and 3.4.5, respectively. Figure 3.14(a) illustrates the reduced cycles of each CPU stall component in SISD-based and SIMD-based MK. Figure 3.14(b) shows the improvement of the CPU

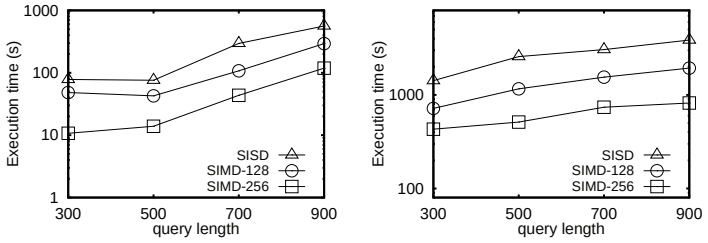
cycles with respect to different query lengths. Again, the SIMD-based solution introduces fewer stall cycles as compared with the SISD-based solution.



(a) stall breakdown on EEG-MK (b) CPU time breakdown

Figure 3.14. SISD-based and SIMD-based MK, EEG-MK

We then compare the performance of the methods on the motif discovery problem. Figure 3.15 plots the execution time (logscale) of the methods with respect to the query length. The performance gap between our methods and SISD widens as the query length increases. The speedup of SIMD over SISD ranges from 2.2 to 6.0.



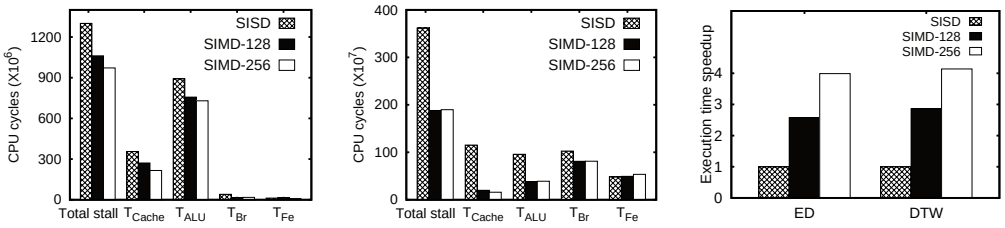
(a) EEG-MK (b) TAO-MK

Figure 3.15. [Motif discovery] vary query length

3.5.3 kNN Classification

We show the breakdown of CPU stalls of UCR Suite based kNN classification problem on Weather dataset in Figure 3.16. We set k=1 which is the default

setting in [33]. This problem is less computational intensive (one candidate per sequence) when compared to the subsequence search problem ($O(n)$ subsequences per sequence) and the motif problem ($O(n^2)$ subsequence pairs per sequence). Even though it is less computational intensive, the SIMD-based solution still saves ∼50% stall cycles for DTW as compared to the SISD-based solution (cf. Figure 3.16). Next we show the execution time speedup of the methods on the kNN classification problem in Figure 3.16(c). kNN classification problem is less computational intensive, Thus, the speedup by SIMD is lower than before. Nevertheless, SIMD still outperforms all other methods.



(a) UCR-ED based breakdown (b) UCR-DTW based breakdown (c) speedup

Figure 3.16. Breakdown of CPU stalls and speedup, kNN Classification

3.6 Chapter Summary

3.6.1 Conclusion

In this work, we conduct performance profiling on existing solutions for time series problems. We find that the performance bottleneck is caused by CPU stalls. We have redesigned vectorized lower-bound and distance functions with SIMD instructions for time series problems. Through our experimental results and analysis, we have two key findings, which will shed light on the design and

implementation of algorithms on modern commodity CPUs, which are used in Chapter 5.

Firstly, the performance bottlenecks of different time series applications are different. Even for the same time series algorithm, it may incur different bottlenecks on different datasets, depending on the pruning power of each specific lower bound function.

Secondly, the characteristics of modern CPUs (e.g., branch prediction unit, hardware prefetching, vectorization) play important roles in the execution time of an implementation. Frequently-used functions (e.g., lower-bound and exact distance computations) need to be redesigned in order to unlock the full potentials of modern commodity CPUs.

3.6.2 Research Directions

Emerging processor architectures have new characteristics and lead to opportunities for further optimization. For example, the 'Many Integrated Core' (MIC) architecture [53] combines a large number cores on a single chip (e.g., Intel Xeon Phi), so that the access time of data items across different cores may depend on the distances between those cores. It becomes important to distribute the workload and transfer data carefully among different cores / threads.

Although our proposed techniques can accelerate existing algorithms by 2-6 times in a single machine, they would take a few hours for very long queries (especially for DTW similarity search). It becomes important to investigate parallel algorithms that run on multiple machines. Some open issues include how to distribute the load among machines, and how to reduce the communication

cost among machines.

Chapter 4

Exploit Every Bit: Effective Caching for High-Dimensional Nearest Neighbor Search

4.1 Introduction

The k nearest neighbor (kNN) search takes a query point q and a point set $\mathcal{P}$ as input, and returns k points of $\mathcal{P}$ that are nearest to q. It has a wide range of applications in multimedia information retrieval [31], where multimedia objects (e.g., images, audio, video) are modeled as data points with dimensionality in orders of hundreds [93, 39].

Due to *the curse of dimensionality* in the high dimensional space [98], the query efficiency of exact indexing methods degenerates to that of linear scan. Re-

cent research, in both computer vision [10, 20] and database [93, 39, 40] communities, focus on finding approximate results for the kNN query. Locality sensitive hashing (LSH) [48, 29, 42] is an attractive approach as it offers c-approximate kNN results[1] at a sub-linear time complexity of $|n|$. It reduces the dimensionality of data by hashing similar data items into the same hash bucket with high probability. Unlike conventional and cryptographic hash functions, LSH aims to maximize the probability of a "collision" for similar data items. The structure of LSH consists of a collection of hash tables. Each hash bucket contains a list of object identifiers (object IDs) rather than actual points. The actual data points are often stored in another file. At the query time, we process a query q in two phases:

1. *the candidate generation phase*: retrieve a candidate set of object identifiers from hash tables,

2. *the candidate refinement phase*: for all object identifiers in the candidate set, fetch their data points from data point file in the hard disk, then compute their distances to q and determine the k nearest results.

We mainly consider disk-based LSH [93, 39], which is suitable for very large datasets that cannot fit into the memory. Existing LSH methods require fetching a large set of candidates (typically hundreds or thousands) from the disk and thus incur significant disk I/O costs. Therefore, the candidate refinement phase turns out to be the performance bottleneck in recent LSH methods. To verify this, we execute the state-of-the-art LSH method (C2LSH [39]) on three real

[1] A point p is called a c-approximate NN of q if $dist(q,p) \leq c \cdot dist(q,p^*)$, where c is the approximation ratio and p^* is the exact NN of q.

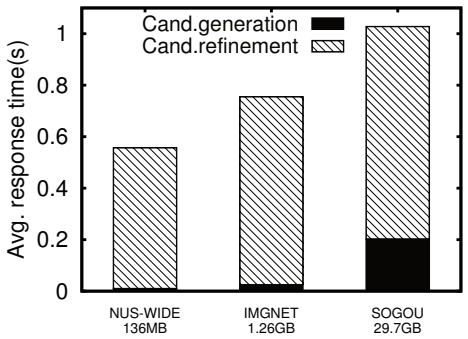


Figure 4.1. Running time (wall-clock) of C2LSH

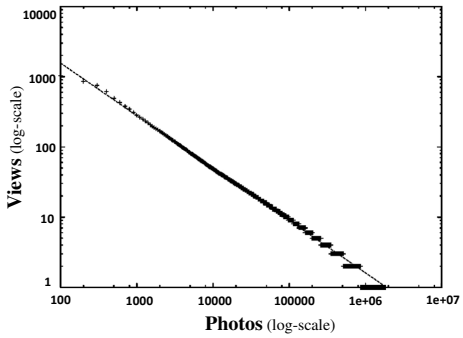


Figure 4.2. Total number of views per photo

high-dimensional datasets stored on the disk (used in the experimental study). Figure 4.1 depicts the average running time (wall-clock time) per query of C2LSH on these datasets. The candidate refinement stage is the performance bottleneck and it motivates us to optimize the candidate refinement time.

In this chapter, we exploit a query log and devise caching techniques to reduce the candidate refinement time of high-dimensional kNN search. Caching can benefit from the temporal locality of queries as observed in typical query logs [65]. Similarly, we expect that the query logs in multimedia retrieval systems exhibit temporal locality. For example, In Flickr, a small fraction of photos receive most of the views (see Figure 4.2, adopted from [94]). Although there exist caching techniques [34, 85] for kNN search on distance-based indexing methods [27, 52], they are not applicable to LSH methods. LSH methods require lookup of objects by object identifiers; however, such lookup operation cannot be supported by the caches in [34, 85].

In our caching problem, the main research question is: *how to exploit the*

limited memory size and query workload to reduce the candidate refinement time. In order to boost the cache hit ratio, we propose to cache conservative approximations of data points (i.e., representing each point in a few bits). Such conservative representation provides lower and upper distance bounds, which can be used to prune unpromising candidates and detect true kNN results early.

4.1.1 Technical Challenges

(1) *Given the query workload (and the number of bits for encoding an approximated point), which scheme is the most effective for encoding data points?*

(2) *Given a cache size, what is the optimal number of bits for encoding a data point?*

Interestingly, we discover that we can cast challenge (1) as a histogram optimization problem. Traditional histograms have been designed for the selectivity estimation problem [51]; however, they are not effective with respect to our optimization goal. This motivates us to find the most effective histogram for our problem. Our method exploits both the data distribution and the query workload to develop compact approximations (of data points) that lead to tight distance bounds.

For challenge (2), it is non-trivial to find the optimal number of bits for encoding a data point in the cache. If each point occupies too few bits, then the cache hit ratio becomes high but those cached points lead to loose distance bounds. If each point occupies too many bits, then they provide tight distance bounds but the cache hit ratio becomes low. In this chapter, we will develop a cost model to find the optimal number of bits for encoding a data point.

The novelties of this chapter are: (i) we formulate a novel histogram optimization problem for reducing the candidate refinement cost in kNN search, (ii) our proposed solution is generic; it is applicable to not only LSH methods, but also to exact tree-based indexes.

4.1.2 Technical Contributions

Our technical contributions are summarized as follows.

- we formulate an appropriate histogram metric for our problem, and design an algorithm to construct an optimal histogram with respect to the novel histogram metric for challenge (1) ⟨Section 4.3⟩;
- we extend our solution for exact tree-based indexes (e.g., iDistance, VP-tree) ⟨Section 4.3.6⟩;
- we devise a cost model for estimating the performance of our solution and for automatic tuning parameter in our solution, to address challenge (2) ⟨Section 4.4⟩;
- we demonstrate the superiority of our caching solution on three real datasets, in particular, the largest dataset (SOGOU, 29.7 GB) has a real query log ⟨Section 4.5⟩.

The remainder of this chapter is organized as follows. We formulate our caching problem in Section 4.2 and present our histogram-based caching method in Section 4.3, then provide a cost model and the optimal parameter setting in Section 4.4. We conduct our experimental study in Section 4.5.

4.2 Definition and Problem Statement

4.2.1 Definitions

We represent points and the distance both in the point form and the vector form interchangeably.

Definition 4.1 (Point) *A point p is defined as a d-dimensional tuple $p = (p.1, p.2, ..., p.d)$. Its vector form is defined as: $\overrightarrow{p} = [p.1, p.2, ..., p.d]$.*

Definition 4.2 (Distance metric) *The Euclidean distance $dist_q(c)$ of a data point c from a query point q is defined as:*

$$dist_q(c) = \sqrt{\sum_{j=1}^{d}(q.j - c.j)^2} = ||\overrightarrow{q} - \overrightarrow{c}||.$$

Then we define the kNN search problem as:

Definition 4.3 (kNN search problem) *Given a query point q and a point set $\mathcal{P}$, the kNN search returns a subset $R \subset P$ of k points such that $dist_q(p) \leq dist_q(p')$ for any $p' \in P - R$.*

In this chapter, we just return the identifiers of points in R but not the actual points. This is reasonable for multimedia information retrieval applications.

As discussed in the introduction, we focus on accelerating *disk-based LSH methods* (e.g., C2LSH [39]) without affecting their query results. These methods access (i) a hash-based index $\mathcal{I}$, whose hash buckets store point identifiers, and (ii) a sequential file for the point set $\mathcal{P}$, which supports direct access of data

point by identifier[2].

During query processing, we first retrieve a candidate set $C(q)$ from the index $\mathcal{I}$ as follows:

Definition 4.4 (Candidate set $C(q)$) *Given a query point q, the index $\mathcal{I}$ reports a set of identifiers for candidate points $C(q) = \{id_i\}$.*

Then, we fetch the corresponding data points by identifiers from the file of $\mathcal{P}$.

4.2.2 Research Objective

For typical disk-based LSH methods, the candidate refinement time T_{refine} is usually much longer than candidate generation time T_{gen} (see Figure 4.1). In this chapter, we aim to reduce T_{refine} significantly by caching points in RAM. To boost the cache hit ratio, we propose to cache *compact approximate points* (p'_{id}), which will be elaborated in Section 4.3.

Figure 4.3 illustrates the framework of our caching problem for kNN search on a high dimensional dataset. Our kNN search procedure (cf. Section 4.3) consists of three phases: (1) candidate generation, (2) candidate reduction, (3) candidate refinement. Both Phases 1 and 3 apply existing work directly and they incur I/O. In Phase 1, we apply an existing index $\mathcal{I}$. In Phase 3, we apply a multi-step kNN search method [80, 56], which will be elaborated in Section 4.2.3. Phase 2 incurs no I/O and it runs our proposed technique to reduce the number of candidates before entering Phase 3.

[2] An alternative is to store $\mathcal{P}$ based on the distribution of clusters in data [52]. We will test its effect in experiments.

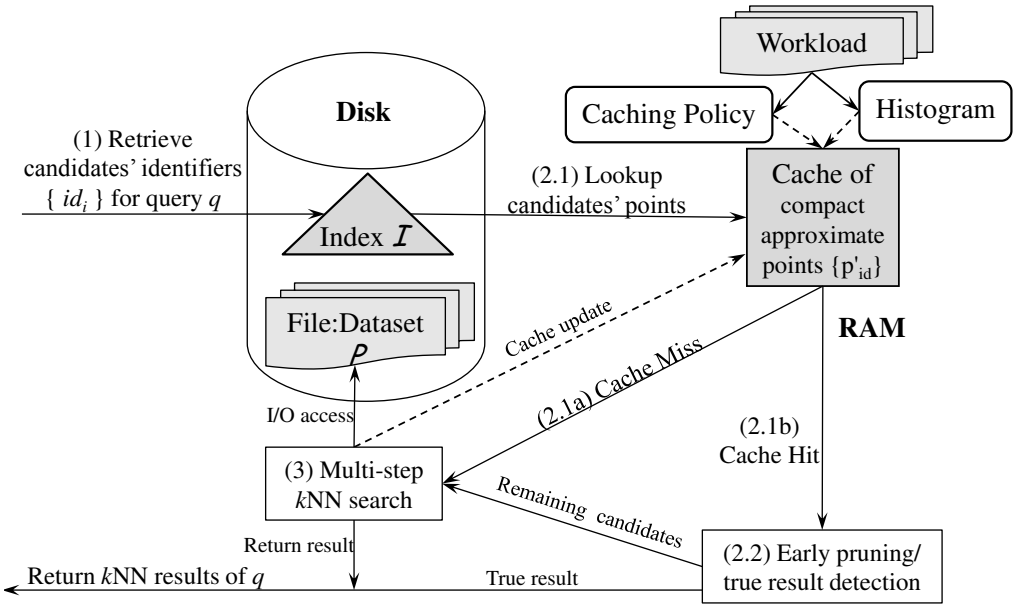


Figure 4.3. Framework of caching on a high-dimensional dataset

Since the candidate refinement time T_{refine} is dominated by the I/O cost, we express it as: $T_{refine} \approx T_{io} \cdot C_{refine}$, where T_{io} is the disk I/O cost for fetching a data point and C_{refine} is the remaining candidate size for the refinement phase.

In general, C_{refine} is the sum of (i) the number of candidates not in the cache, or (ii) the number of candidates in the cache but they cannot be pruned:

$$\begin{aligned} C_{refine} =& (1 - \rho_{hit}) \cdot |C(q)| + \rho_{hit} \cdot (1 - \rho_{prune}) \cdot |C(q)| \\ =& (1 - \rho_{hit} \cdot \rho_{prune}) \cdot |C(q)| \end{aligned} \tag{4.1}$$

where ρ_{hit} is the cache hit ratio, and ρ_{prune} is the ratio of the number of pruned candidates to the number of cache hits.

The scope of our problem: Our goal is to minimize the value of C_{refine}

before candidate refinement (and thus minimize T_{refine}). We can reduce C_{refine} by two orthogonal aspects: (i) a caching policy that offers high hit ratio, and (ii) compact approximation of points that provides tight distance bounds for pruning.

The study of (i) caching policies is orthogonal to our problem. In fact, our proposed solution can be applied with existing web caching policies [65], e.g., Least-Recently-Used (LRU) and Highest-frequency-first (HFF). LRU is a dynamic caching policy, whereas HFF is a static caching policy that requires a query workload $\mathcal{WL}$[3] to decide the initial cache content. We will elaborate HFF in Section 4.4. Our focus is issue (ii), which will be discussed in Section 4.3. Formally, we define our problem as:

Definition 4.5 (Caching problem) *Given a cache size $\mathcal{CS}$, a workload of queries $\mathcal{WL}$, and a point set P, determine the cache content such that it minimizes $\sum_{q \in \mathcal{WL}} C_{refine}(q)$.*

Besides LSH methods, we will discuss how to adapt our proposed solution for tree-based indexes (e.g., R-tree, X-tree, SR-tree) [18] in Section 4.3.6.

Note that our solution offers speedup without affecting the quality of query results. If an exact tree-based index is used, the query results remain exact. If an LSH method is used, the quality of its query results is preserved.

[3] It is usually the historical query log.

4.2.3 Multi-step kNN Search

We illustrate multi-step kNN search methods [80, 56] in this section. Kriegel et al. [56] present an efficient method that requires both lower and upper distance bounds functions. For example, suppose the candidate set of q is $C(q) = \{p_1, p_2, p_3, p_4\}$. Figure 4.4 depicts the lower and upper distance bounds of these candidates as intervals. Their exact distances (from q) are shown as gray dots; However, they can be obtained only after fetching the exact points from the disk. First, it calculates the k-th smallest lower distance bound lb_k and the k-th smallest upper distance bound ub_k, among the candidates, the values of lb_2 and ub_2 are shown in Figure 4.4. Since the upper distance bound of p_1 is less than lb_2, lb_3 and lb_4, p_1 must be a result so we need not fetch p_1 from the disk. The lower distance of p_4 is larger than ub_2, p_4 cannot be a result so we also need not fetch p_4. It suffices to fetch p_2 and p_3 from the disk.

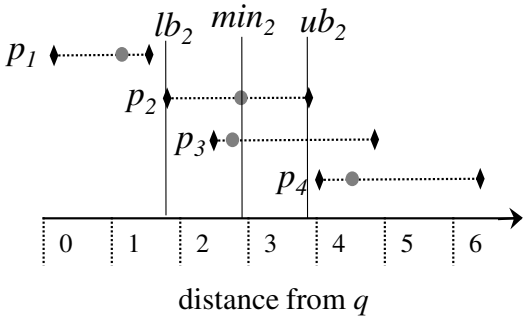


Figure 4.4. Multi-step kNN methods, $k = 2$

4.3 Histogram-based Caching for kNN Search

Histograms, generally, are designed to provide selectivity estimates on a single column attribute of a relational table. In this section, we utilize a histogram to define compact approximate representations of points. Then, we present a kNN search algorithm with our proposed histogram-based cache. Finally, we formulate the novel metric to build an effective histogram for kNN search.

4.3.1 Histogram and Approximate Points

We first define a histogram as follows:

Definition 4.6 (Histogram) *A histogram $\mathcal{H}$ is defined as an array of buckets which cover a domain interval (e.g., [0,..,N_{dom}])*[4]*. Let B be the number of buckets in $\mathcal{H}$. Each bucket (say, the i-th bucket) stores: (i) an interval $[l_i..u_i]$ of values, and (ii) the total frequency $freq_i$ of values in the interval.*

Figure 4.5b shows an example histogram. In our problem, we only care about the bucket position i and its interval $[l_i..u_i]$, but not its frequency $freq_i$. Each bucket position can be represented by a binary code. In general, for a histogram with B buckets, the code length is: $\tau = \lceil \log_2(B) \rceil$. We will examine histogram construction methods in Section 4.3.3 and discuss the tuning of τ in Section 4.4.

With a histogram $\mathcal{H}$, we can convert an exact point p into an approximate point p' as follows:

[4] N_{dom} is the largest dimension value of all points in all dataset

Definition 4.7 (Bucket lookup) *Given a value v, we define the function $\mathcal{H}(v) = i$, such that v is covered by the interval of the bucket i, i.e., $l_i \leq v \leq u_i$.*

Definition 4.8 (Approximate point p') *Given a d-dimensional point $p = (p.1, p.2, \cdots, p.d)$, we define an approximate point p' with respect to the global histogram $\mathcal{H}$ as:*

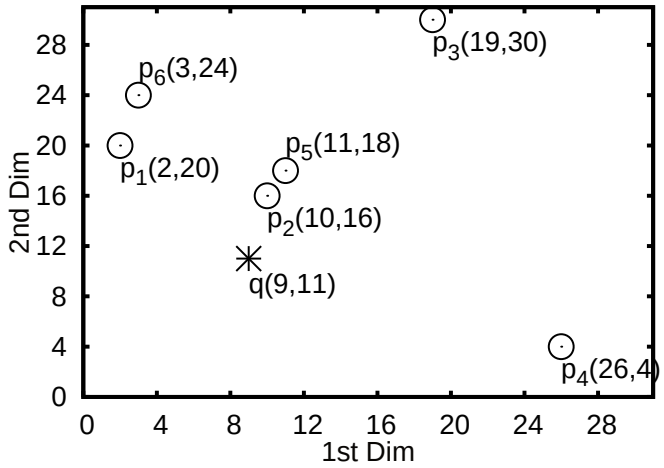
$$p' = (\ \mathcal{H}(p.1), \mathcal{H}(p.2), \cdots, \mathcal{H}(p.d)\ ) .$$

Specifically, we approximate each dimension value by a τ-bit code. In general, if the dataset has different domain sizes for different dimensions, then we may apply normalization to scale each dimension.

For example, consider the point $p_1 = (2, 20)$ in Figure 4.5(a). According to the example histogram in Figure 4.5(b), the values 2 and 20 are mapped to the codes 00 and 10 respectively. Thus, we can represent p_1 by a bit-string p'_1 : " 00 10 ". Figure 4.5(c) shows the cache content which stores the approximate points p'_1, p'_2, p'_3, p'_4. To achieve a compact cache, we pack the bit-string encoding of each point into one or multiple consecutive words in memory [5].

In subsequent discussion, we use a global histogram $\mathcal{H}$ to define all dimensions of an approximate point p'. We will discuss alternative histograms in Section 4.3.6.

[5] With this implementation, an approximate point occupies $\frac{d \cdot \tau}{L_{word}}$ words, where L_{word} is the memory word size (in bits). The value of L_{word} (typically 32, 64) is fixed by the CPU. During kNN search, we can extract these cache items by performing bitwise operations on these words.



(a) a dataset $\mathcal{P}$, with $d = 2$

position / code i	interval $[l_i..u_i]$	frequency
00	[0..7]	3
01	[8..15]	2
10	[16..23]	4
11	[24..31]	3

(b) a histogram $\mathcal{H}$, with $\tau = 2$

$p'_1 : |00|10|$
$p'_2 : |01|10|$
$p'_3 : |10|11|$
$p'_4 : |11|00|$

(c) a cache, with $\mathcal{CS} = 16$ bits

Figure 4.5. Example of histogram-based coding

4.3.2 kNN Search Algorithm

Algorithm 4.5 elaborates the framework of kNN search with histogram-based caching. The corresponding steps in Figure 4.3 are also labeled in this algorithm. First, we retrieve a set $C(q)$ of candidates from the index $\mathcal{I}$ (Line 2), and then check whether they are in the cache. For each candidate c_i found in the cache (Lines 5–6), we compute its lower/upper distance bounds to q as follows.

$$dist_q^+(p') = \sqrt{\sum_{j=1}^{d} \max\{|q.j - p^l.j|, |q.j - p^u.j|\}^2}$$

$$dist_q^-(p') = \sqrt{\sum_{j=1}^{d} \begin{cases} 0 & \text{if } p^l.j \le q.j \le p^u.j \\ \min\{|q.j - p^l.j|, |q.j - p^u.j|\}^2 & \text{otherwise} \end{cases}}$$

where $p^l.j = l_{\mathcal{H}(p'.j)}$ and $p^u.j = u_{\mathcal{H}(p'.j)}$. For those candidates missing in the cache, we will fetch their points from the disk (in the final phase).[6]

The next phase (Lines 7–13) focuses on reducing the candidate size (which do not incur disk accesses). Among all candidates $C(q)$, we derive the k-th minimum lower bound distance lb_k, the k-th minimum upper bound distance ub_k (Lines 7–8). First, we prune candidates having $c_i.lb$ larger than ub_k (Lines 10–11), as they cannot be among k nearest neighbors. Second, we identify candidates having $c_i.ub$ less than lb_k (Lines 12–13), as they must be results and moved to the result set $\mathcal{R}$. Obviously, the effectiveness of this phase depends on the tightness of distance bounds (and the histogram $\mathcal{H}$). We will explore this issue in the remaining subsections.

Finally, in the refinement phase, we apply a multi-step kNN search method [80, 56] (which incurs disk I/O), as described in Section 4.2.3, with the remaining candidate set $C(q)$. This search would fetch data points from $\mathcal{P}$ when necessary. The update of the cache Ψ is optional; it is only required when a dynamic caching policy (e.g., LRU) is used.

[6] An optimization is to fetch those points from disk immediately, in order to tighten the bounds lb_k and ub_k to be mentioned soon. However, this optimization is not effective when the hit ratio is low (as few candidates can be pruned) or high (as lb_k and ub_k are tight already).

Algorithm 4.5 kNN Search (Query q, Result size k)

Disk data: Index $\mathcal{I}$, Dataset file P
Memory data: Cache Ψ, histogram $\mathcal{H}$
• Phase 1: candidate generation
1: Result set $\mathcal{R} := \emptyset$
2: retrieve the candidate set $C(q)$ from $\mathcal{I}$
• Phase 2: candidate reduction
3: **for** each $c_i \in C(q)$ **do** ▷ part 2.1: cache lookup
4: $c_i.lb := 0; c_i.ub := +\infty$
5: **if** Ψ contains p'_{c_i} **then** ▷ cache hit
6: $c_i.lb := dist_q^-(p'_{c_i}); c_i.ub := dist_q^+(p'_{c_i})$
7: $lb_k :=$ the k-th minimum of $\{c_i.lb : c_i \in C(q)\}$
8: $ub_k :=$ the k-th minimum of $\{c_i.ub : c_i \in C(q)\}$
9: **for** each $c_i \in C(q)$ **do** ▷ part 2.2
10: **if** $c_i.lb > ub_k$ **then** ▷ early pruning
11: remove c_i from $C(q)$
12: **else if** $c_i.ub < lb_k$ **then** ▷ true result detection
13: move c_i from $C(q)$ to $\mathcal{R}$
• Phase 3: candidate refinement
14: **if** $|\mathcal{R}| < k$ **then**
15: Multi-step-kNN($C(q), \mathcal{P}, \mathcal{R}$) ▷ Ref. [80, 56]
16: return $\mathcal{R}$

Example: Assume $k = 1$ and consider the query q and dataset $\mathcal{P}$ in Figure 4.5a. We show the running steps of kNN search (Algorithm 4.5) in Table 4.1. Suppose that the index $\mathcal{I}$ reports the candidate set for q as: $C(q) = \{p_1, p_2, p_3, p_4, p_5, p_6\}$. Since the candidates p_5 and p_6 are missing in the cache, we must access their points from the disk. The cache is used to retrieve approximate points (p'_1, p'_2, p'_3, p'_4) of four candidates, as shown in Figure 4.5c. Then, we compute their lower/upper distance bounds from q, and obtain the distance threshold $ub_k = dist_q^+(p_2) = \sqrt{\max\{(9-8),(9-15)\}^2 + \max\{(11-16),(11-23)\}^2} = \sqrt{6^2+12^2} = 13.42$. We can prune p_3 and p_4 as their lower bound distances are above 13.42.

Finally, we apply multi-step kNN search [80, 56] on the remaining candidates p_1, p_2, which costs at most 2 disk accesses. In summary, this example incurs at most 4 disk accesses: 2 for p_5, p_6, and at most 2 for p_1, p_2.

Table 4.1. kNN search on the cache, $k = 1$

cache (code array)		rectangle	$[lb_i..ub_i]$	pruned ?
p_1 : |00|10|		([0..7],[16..23])	[5.39..15]	
p_2 : |01|10|	⇒	([8..15],[16..23])	[5.00..**13.42**]	
p_3 : |10|11|		([16..23],[24..31])	[14.76..24.41]	yes
p_4 : |11|00|		([24..31],[0..7])	[15.52..24.60]	yes

4.3.3 Histogram Solutions for kNN Algorithm

In relational databases, histograms are used to summarize the data distribution and provide result size estimations for selection queries [51]. However, they have not been used for supporting kNN search. This raises interesting questions:

4.3.3.1 Are existing histograms effective for kNN search?

(1) Heuristic histograms (e.g., equi-width, equi-depth) [50]. In order to describe them, we use the notations in Definition 4.6, and denote $F[x]$ as the frequency of value x (in a table column). For equi-width histogram, all buckets have the same width $(u_i - l_i)$; whereas for equi-depth histogram, all buckets have approximately the same sum of frequencies $(\sum_{x=l_i}^{u_i} F[x])$.

(2) V-optimal histogram [51], which minimizes the average estimation error

of selection queries according to the *sum squared error* (SSE) metric [51]:

$$\mathcal{M}_{SSE}(\mathcal{H}) = \sum_{i=1}^{B} \sum_{x=l_i}^{u_i} (F[x] - AVG([l_i, u_i]))^2$$

where $AVG([l_i, u_i]) = \frac{\sum_{x=l_i}^{u_i} F[x]}{u_i - l_i + 1}$ is the average frequency of values in bucket i.

We illustrate the effectiveness of these histograms on kNN search by an example in Figure 4.6. We will evaluate its effectiveness on real datasets in the experimental section. For ease of illustration, we consider a 1-dimensional dataset $\{3, 4, 10, 12, 22, 24, 30, 31\}$ and each value x in it has frequency $F[x] = 1$. Suppose that $q = 17$ is the only query in the query workload $\mathcal{WL}$. Assume that the cache can hold all approximate points and the code length is $\tau = 2$. Thus, each histogram has $B = 2^\tau = 4$ buckets. Figure 4.6 depicts the buckets (intervals) of equi-width, equi-depth, and V-optimal histograms. Both equi-depth and V-optimal histograms are the same in this example.

For each histogram, we run the kNN algorithm (in Section 4.3.2) to find 2NN ($k = 2$) at $q = 17$, and show the running steps in the figure. Histogram buckets are shown in the first column. By using distance bounds (lb_k, ub_k) in the second column, we can prune unpromising candidates and detect true result early in order to reduce the remaining candidate size. In this example, the ideal histogram for our problem has zero remaining candidates (see the last column). On the other hand, equi-width has 6 remaining candidates (in buckets ②,③,④), and equi-depth (and V-optimal) has 4 remaining candidates (in buckets ②,③). These histograms would incur higher cost than the ideal histogram in the candidate

refinement phase.

Differences from traditional histograms: Traditional histograms are designed for the selectivity estimation problem. For example, a traditional histogram metric like $\mathcal{M}_{SSE}(\mathcal{H})$ [51]internal minimize the average estimation error over all histogram buckets. It is fine to have a bucket with large width $(u_i - l_i)$, as long as the values within the bucket have similar frequencies. However, for the kNN problem, such a bucket causes loose distance bounds and thus not effective in shrinking the candidate set. Furthermore, $\mathcal{M}_{SSE}(\mathcal{H})$ does not exploit query workload information, which is important for deriving an effective histogram for our problem.

In Section 4.3.4.1, we will formulate a new histogram metric for building an effective histogram for kNN search. Experiments show that our novel histogram incurs lower I/O cost than existing histograms (e.g., equi-width, equi-depth, V-optimal) on kNN search by at least 50%.

4.3.3.2 What is the optimal histogram for kNN search?

We demonstrate an example of the optimal histogram for kNN search in the last row in Figure 4.6. Interestingly, in this histogram, the buckets close to q are tight, and other buckets can be loose. It allows us to derive tighter bounds (lb_k, ub_k), prune candidates in buckets 1, 4, and detect the candidates in buckets 2, 3 as true results. Since there are no candidates in the refinement step, it incurs zero I/O cost in this example.

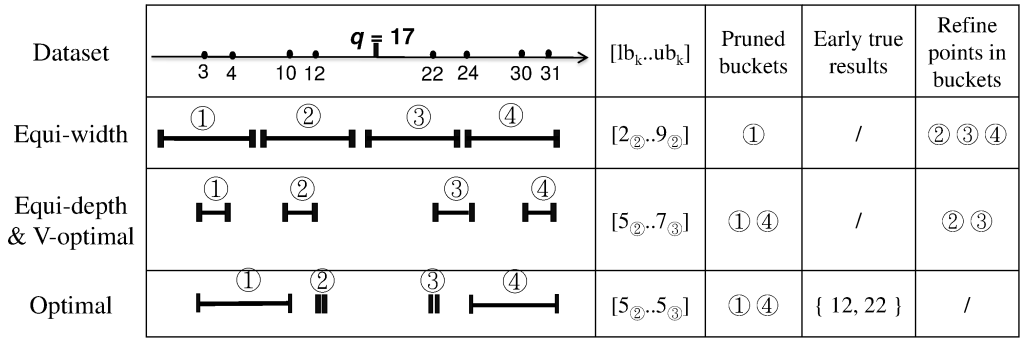


Figure 4.6. Effectiveness of histograms, with $B = 4$ buckets, on 2NN search, $\mathcal{WL}$ = { q }

4.3.4 Effective Histogram Metric

4.3.4.1 Histogram Metric for kNN Search

This section formulates a histogram metric that captures the effectiveness of a histogram for kNN search. In the following discussion, we associate notations with the superscript q if they depend on q (e.g., $C^q_{refine}, lb^q_k, ub^q_k$).

Our goal is to minimize the remaining candidate size C^q_{refine} (cf. Eqn. 4.1) before the candidate refinement phase in Algorithm 4.5. Note that C^q_{refine} is influenced by the distance bounds lb^q_k, ub^q_k (obtained at Lines 7–8), which are derived from the content of $\mathcal{H}$. For each candidate c that hits in the cache Ψ, we can skip it in candidate refinement in two cases:

- (i) if $dist^+_q(c) \leq lb^q_k$, then it must be a true result
- (ii) if $dist^-_q(c) \geq ub^q_k$, then it cannot be a result

Otherwise, the candidate requires refinement.

With the above observation, we proceed to define our histogram construction problem as follows.

Definition 4.9 (Optimal kNN histogram problem) *Let $V = \{v_1, \cdots, v_n\}$ be the set of distinct dimensional values of data points in $\mathcal{P}$.*

Given the code length τ, the query workload $\mathcal{WL}$, and the cache Ψ (i.e., a set of points), the problem is to build a histogram $\mathcal{H}$ with $\mathcal{B} = 2^\tau$ buckets such that (i) it covers all values in V, and (ii) it minimizes the following metric:

$$\mathcal{M}_{kNN}^{\mathcal{WL}}(\mathcal{H}) = \sum_{q \in \mathcal{WL}} \sum_{c \in C(q) \wedge \Psi} refine_{\mathcal{H}}(c) \text{ (M1)}$$

$$\text{where } refine_{\mathcal{H}}(c) = \begin{cases} 0 & \text{if } dist_q^-(c) \geq ub_k^q \\ 0 & \text{if } dist_q^+(c) \leq lb_k^q \\ 1 & \text{otherwise} \end{cases}$$

This optimal histogram construction problem is challenging as the number of combinations of buckets lead to a huge search space $O\binom{n}{\mathcal{B}-1}$, which is $O(n^{\mathcal{B}})$ when $\mathcal{B} \ll n$. Thus, we propose an approximate solution below.

4.3.4.2 Approximate Histogram Metric

Besides the huge search space $O(n^{\mathcal{B}})$, the above metric does not use histogram bucket intervals explicitly, thus rendering it inconvenient to develop a solution.

To tackle these issues, we propose to approximate the metric (M1), $\mathcal{M}_{kNN}^{\mathcal{WL}}(\mathcal{H})$, in a form that can be efficiently solved. In order to minimize $\mathcal{M}_{kNN}^{\mathcal{WL}}(\mathcal{H})$, we

should maximize the number of candidates that can satisfy $refine_{\mathcal{H}(c)} = 0$ for a given query q. Note that there are two cases for $refine_{\mathcal{H}}(c) = 0$: **(Case i)** the lower bound of c is larger than the k-th upper bound (i.e., $dist_q^-(c) \geq ub_k^q$), so c is not a result point and, **(Case ii)** the upper bound of c is smaller than the k-th lower bound (i.e., $dist_q^+(c) \leq lb_k^q$), so c must be a result point.

Observe that at most $|C(q)| - k$ candidates can satisfy $dist_q^-(c) \geq ub_k^q$ in **(Case i)**, but at most k candidates can satisfy $dist_q^+(c) \leq lb_k^q$ in **(Case ii)**. Since $|C(q)| - k \gg k$, we focus on **(Case i)** and plan to minimize ub_k^q.

Recall that $ub_k^q = k^{th} \min_{c \in C(q) \wedge \Psi} dist_q^+(c)$ is contributed by k points in $C(q) \wedge \Psi$. By denoting these k points as $b_1^q, b_2^q, \cdots, b_k^q$, we have $ub_k^q = \max_{1 \leq r \leq k} dist_q^+(b_r^q)$. We then define the error vector of a candidate in Def. 4.10.

Definition 4.10 (Error vector) *Given a histogram $\mathcal{H}$ and a candidate c, we define the error vector of c as $\overrightarrow{\epsilon(c)} = [\epsilon(c).1, \epsilon(c).2, ..., \epsilon(c).d]$ where $\epsilon(c).j = u_{\mathcal{H}(c.j)} - l_{\mathcal{H}(c.j)}$.*

By using Lemma 4.1 (stated below), we derive: $ub_k^q \leq \max_{1 \leq r \leq k} ||\epsilon(b_r^q)|| + dist_q(b_r^q)$. Note that each term $dist_q(b_r^q)$ is a constant value (depending on q) that cannot be optimized. As a heuristic, we approximate the minimization of ub_k^q by minimizing the following Metric (M2):

$$\mathcal{M}2_{kNN}^{\mathcal{WL}}(\mathcal{H}) = \sum_{q \in \mathcal{WL}} \sum_{r=1}^{k} ||\overrightarrow{\epsilon(b_r^q)}||^2 \quad \text{(M2)}$$

Next, for convenience, we define a multi-set $\mathcal{QR}$ to store all b_r^q for queries in

the workload $\mathcal{WL}$:

$$\mathcal{QR} = \{b_r^q \ : \ q \in \mathcal{WL}, dist(q, b_r^q) \leq ub_k^q, r \in [1, k]\} \tag{4.2}$$

Then, we define $F'[x]$ as the frequency of x in coordinates of candidates in $\mathcal{QR}$, where $x \in [0, N_{dom}]$:

$$F'[x] = \texttt{COUNT}\ \{b_r^q.j = x \ : \ b_r^q \in \mathcal{QR}, j \in [1, d]\} \tag{4.3}$$

By Lemma 4.2 (stated below), we express Metric (M2) into the following form using histogram bucket information.

$$\mathcal{M}2_{kNN}^{\mathcal{WL}}(\mathcal{H}) = \sum_{i=1}^{B} \sum_{x=l_i}^{u_i} F'[x] \cdot (u_i - l_i)^2 \qquad \text{(M3)}$$

In the following analysis, we employ vectors to express distance computations in a concise manner.

Definition 4.11 (Distance on vectors) *Let q be a query point and c be a candidate point. The Euclidean distance is $dist_q(c)$, and the upper distance bound $dist_q^+(c)$ can be expressed in terms of the dot product of vectors:*

$$dist_q^+(c) = ||(\overrightarrow{q} - \overrightarrow{c^u})||$$

where $\overrightarrow{c^u} = [c^u.1, c^u.2, ..., c^u.d]$ are defined by a histogram $\mathcal{H}$ as follows:

$$c^u.j = \begin{cases} l_{\mathcal{H}(c.j)} & \text{if } |q.j - l_{\mathcal{H}(c.j)}| > |q.j - u_{\mathcal{H}(c.j)}| \\ u_{\mathcal{H}(c.j)} & \text{otherwise} \end{cases}$$

Lemma 4.1 (Distance inequality) *Any candidate c satisfies*

$$dist_q^+(c) - dist_q(c) \leq ||\overrightarrow{\epsilon(c)}||$$

Proof. Let $\overrightarrow{A} = \overrightarrow{c} - \overrightarrow{q}$ and $\overrightarrow{\Delta_c} = [\Delta_c.1, \Delta_c.2, ...\Delta_c.d]$. Calculate $\Delta_c.j$ as:

$$\Delta_c.j = \begin{cases} \epsilon(c).j & \text{if } c^u.j > c^l.j \\ -\epsilon(c).j & \text{otherwise} \end{cases}$$

Since $||\overrightarrow{\Delta_c}|| = ||\overrightarrow{\epsilon(c)}||$ and $dist_q(c) = ||\overrightarrow{A}||$ always holds, we have:

$$\begin{aligned} &dist_q^+(c) = ||(c^u - \overrightarrow{q})|| \leq ||(\overrightarrow{\Delta_c} + \overrightarrow{c} - \overrightarrow{q})||^2)^{\frac{1}{2}} \\ &= (\overrightarrow{\Delta_c} \cdot \overrightarrow{\Delta_c} + 2\overrightarrow{\Delta_c} \cdot (\overrightarrow{c} - \overrightarrow{q}) + (\overrightarrow{c} - \overrightarrow{q}) \cdot (\overrightarrow{c} - \overrightarrow{q}))^{\frac{1}{2}} \\ &= (||\overrightarrow{\Delta_c}||^2 + 2\overrightarrow{\Delta_c} \cdot \overrightarrow{A} + ||\overrightarrow{A}||^2)^{\frac{1}{2}} \\ &\leq (||\overrightarrow{\Delta_c}||^2 + 2||\overrightarrow{\Delta_c}|| \cdot ||\overrightarrow{A}|| + ||\overrightarrow{A}||^2)^{\frac{1}{2}} \text{(by Cauchy inequality [88])} \\ &= ||\overrightarrow{\Delta_c}|| + ||\overrightarrow{A}|| = ||\overrightarrow{\epsilon(c)}|| + dist_q(c) \end{aligned}$$

Then, we have: $dist_q^+(c) - dist_q(c) \leq ||\overrightarrow{\epsilon(c)}||$. □

Lemma 4.2 (Metric transformation)

$$\sum_{q \in \mathcal{WL}} \sum_{r=1}^{k} ||\overrightarrow{\epsilon(b_r^q)}||^2 = \sum_{i=1}^{B} \sum_{x=l_i}^{u_i} F'[x] \cdot (u_i - l_i)^2$$

where $u_i - l_i$ denotes the width of bucket i.

Proof.

$$\sum_{q\in\mathcal{WL}}\sum_{r=1}^{k}||\overrightarrow{\epsilon(b_r^q)}||^2 = \sum_{b_r^q\in\mathcal{QR}}||\overrightarrow{\epsilon(b_r^q)}||^2 \text{ by Eqn.4.2}$$

$$= \sum_{b_r^q\in\mathcal{QR}}\sum_{j=1}^{d}(\epsilon(b_r^q).j)^2 \quad \text{by Def. 4.10}$$

$$= \sum_{b_r^q\in\mathcal{QR}}\sum_{j=1}^{d}(u_{\mathcal{H}(\epsilon(b_r^q).j)} - l_{\mathcal{H}(\epsilon(b_r^q).j)})^2$$

$$= \sum_{x=1}^{N_{dom}} F'[x]\cdot(u_{\mathcal{H}(x)} - l_{\mathcal{H}(x)})^2 \quad \text{by Eqn.4.3}$$

$$= \sum_{i=1}^{B}\sum_{x=l_i}^{u_i} F'[x]\cdot(u_i - l_i)^2 \quad \text{group by bucket id}$$

□

4.3.5 Efficient Solution

We proceed to present an efficient solution for the simplified histogram metric $\mathcal{M}2_{kNN}^{\mathcal{WL}}(\mathcal{H})$(**M3**). First, we represent the inner sum in Eqn. 4.4 as follows:

$$\Upsilon([l_i,u_i]) = \sum_{x=l_i}^{u_i} F'[x]\cdot(u_i - l_i)^2 \tag{4.4}$$

Then, we propose an efficient algorithm to construct histogram $\mathcal{H}$ that minimizes the metric $\mathcal{M}2_{kNN}^{\mathcal{WL}}(\mathcal{H})$.

We assume that the value domain is: $1..N_{dom}$.[7] Let $OPT(n,m)$ be the minimum $\mathcal{M}2_{kNN}^{\mathcal{WL}}(\mathcal{H})$ value for the histogram covering the interval $[1..n]$ with at

[7] We can extend this method to handle other value domain, e.g., by applying discretization on floating-point values.

most m buckets. If t is the optimal splitting position for the last bucket, then we have: $OPT(n,m) = OPT(t,m-1)+\Upsilon([t+1,n])$, where $\Upsilon([t+1,n])$ is the metric value contributed by the last bucket ($[t+1,n]$), and $OPT(t,m-1)$ is the minimum metric value for the histogram covering $[1..t]$ with at most $m-1$ buckets. By considering all splitting positions, we take $OPT(n,m)$ as the minimum sum as follows:

$$OPT(n,m) = \min_{1\le t<n} \{OPT(t,m-1) + \Upsilon([t+1,n])\} \tag{4.5}$$

With Eqn. 4.5, we apply the dynamic programming approach to calculate $OPT(n,m)$ and the split positions, for all $1 \le n \le N_{dom}$ and $1 \le m \le B$. Finally, we obtain the optimal histogram $\mathcal{H}$ through these split positions.

Lemma 4.3 (Monotonicity of Υ) *if $l_1 \le l_2$, Then $\Upsilon([l_1,u_i]) \ge \Upsilon([l_2,u_i])$.*

Proof. According to Eqn. 4.3, then $F'[x] \ge 0$. Since $l_1 \le l_2$, then, $\sum_{x=l_1}^{u_i} F'[x] \ge \sum_{x=l_2}^{u_i} F'[x]$ and $(u_i-l_1)^2 \ge (u_i-l_2)^2$. Consider $\Upsilon([l_i,u_i]) = \sum_{x=l_i}^{u_i} F'[x]\cdot(u_i-l_i)^2$ then we can conclude $\Upsilon([l_1,u_i]) \ge \Upsilon([l_2,u_i])$. □

Through Lemma 4.3, our algorithm can terminate when $\Upsilon([t+1,n]) \ge OPT(n,m)$. This technique can significantly reduce the running time when n is very large. The details are presented in Algorithm 4.6.

Time Complexity: This algorithm is only executed once in the offline phase. It has a total of $O(N_{dom}\cdot B)$ calculations for $OPT(n,b)$. In the worst case, each calculation involves $O(N_{dom})$ values of t (Eqn. 4.5). Thus, its time complexity is $O(N_{dom}^2\cdot B)$. It is independent of the dimensionality d and the data size $|\mathcal{P}|$.

Histogram maintenance: We expect that the distribution of queries in the

Algorithm 4.6 Build-kNN-Histogram (Bucket number B, Value domain size N_{dom}, Frequency array F')

```
1: let H be an empty histogram
2: OPT :=new 2D array(1..N_dom, 1..B)                    ▷ for OPT values
3: pos :=new 2D array (1..N_dom, 1..B)                   ▷ for split positions
4: for m from 1 to B do
5:     for n from 1 to N_dom do
6:         if m = 1 then
7:             OPT(n, 1) := ϒ([1, n])
8:         else
9:             OPT(n, m) = +∞
10:            for each t from n − 1 to 1 do
11:                if OPT(n, m) > OPT(t, m-1) + ϒ([t+1, n]) then
12:                    OPT(n, m) := OPT(t, m − 1) + ϒ([t + 1, n])
13:                    pos(n, m) := t
14:                else if ϒ([t + 1, n]) ≥ OPT(n, m) then
15:                    break                              ▷ by Lemma 4.3
16: n := N_dom
17: for m from B to 1 do
18:     if m = 1 then
19:         l := 1, u := n
20:     else
21:         l := pos(n, m) + 1, u := n
22:         n := pos(n, m)
23:     insert the bucket [l..u] to H
24: Return H
```

workload does not change rapidly. Following the practice in search engines [65], we propose to perform updates and rebuild the cache periodically (e.g., daily).

4.3.6 Extensions

4.3.6.1 Adaptation for tree-based indexes

The kNN search on tree-based indexes [19, 27, 52] exhibits interleaving steps between candidate generation and candidate refinement. In this section, we discuss how to adapt our proposed solution to speedup kNN search on tree-based indexes.

We illustrate a general tree structure in Figure 4.7. Each node occupies a disk block. A leaf node stores data points, whereas a non-leaf node stores branching information for its children. Conceptually, we can divide the tree into two parts: (i) the set of non-leaf nodes as the index $\mathcal{I}$, and (ii) the set of leaf nodes as the dataset $\mathcal{P}$. The storage size of $\mathcal{P}$ is generally much larger than that of $\mathcal{I}$. We store the exact $\mathcal{I}$ in memory.

In this scenario, we consider each cache item to be a leaf node (i.e., approximate representations of all points in that node), but not an individual point. We construct the cache as follows. First, we run queries in the query workload $\mathcal{WL}$ and collect the access frequency of each leaf node. Then, we fill the cache with leaf nodes in descending order of access frequency. Finally, with our technique in Section 4.3.5, we can build the histogram $\mathcal{H}$ and determine the approximate representations of data points (in leaf nodes).

Any tree-based kNN search solution $\mathcal{A}$ (e.g., [52, 19]) can utilize the above cache, with some slight modifications described below. During kNN search, before we fetch a leaf node (by its Block ID), we first lookup it in the cache. If the leaf node is not in the cache, then we load it from the disk. Otherwise, we

retrieve the node from the cache, examine its approximate points, and compute lower and upper bounds for that node. In this implementation, our solution provides tight lower and upper bounds for leaf nodes.

In addition, we can further optimize the algorithm as follows. We first compute the lower and upper bound for each point in that node. These bounds can be used to tighten ub_k and prune some unpromising nodes and approximate points. Then, the multi-step kNN search method determines which node should be examined next. As we will illustrate in experiments (cf. Figures 4.16(a,c) in Section 4.5.4.5), the above approximate caching solution performs better than exact caching.

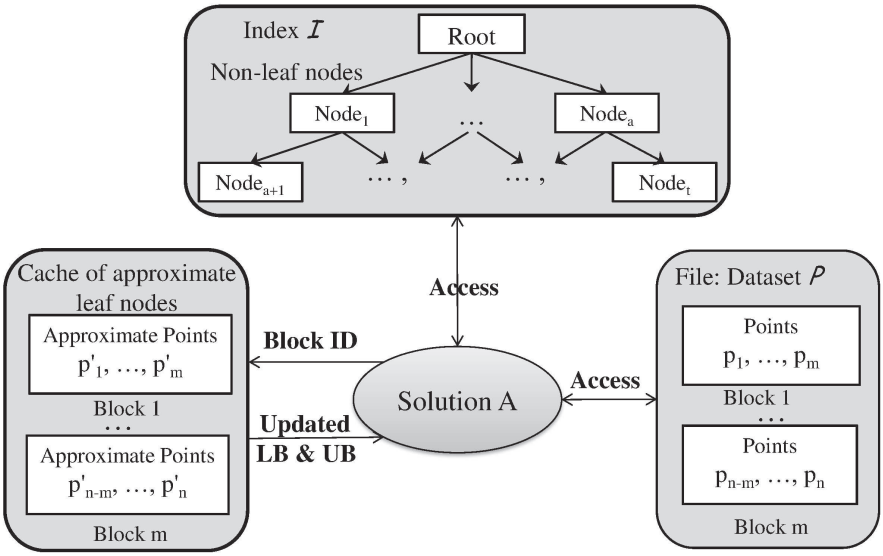


Figure 4.7. Tree-based kNN search with our cache

4.3.6.2 Alternative histogram categories

Besides the global histogram, we may use other histograms to convert a point p into an approximate point p'.

Individual histogram: This approach employs a separate histogram $\mathcal{H}_i$ for each dimension $i = 1..d$. It converts an exact point p to an approximate point p' as follows: $p' = (\ \mathcal{H}_1(p.1), \mathcal{H}_2(p.2), \cdots, \mathcal{H}_d(p.d)\ )$.

We proceed to discuss how to build these d histograms. Observe that our histogram metric M3 is defined by using a frequency array: $F'[x] = \texttt{COUNT}\ \{b_r^q.j = x\ :\ b_r^q \in \mathcal{QR}, j \in [1, d]\}$. We can decompose this array into individual frequency arrays of the form: $F'_j[x] = \texttt{COUNT}\ \{b_r^q.j = x\ :\ b_r^q \in \mathcal{QR}\}$. Then, we can express metric M3 as follows:

$$\sum_{i=1}^{B}\sum_{x=l_i}^{u_i} F'[x]\cdot(u_i-l_i)^2 = \sum_{i=1}^{B}\sum_{x=l_i}^{u_i}(\sum_{j=1}^{d} F'_j[x])\cdot(u_i-l_i)^2$$
$$= \sum_{j=1}^{d}(\sum_{i=1}^{B}\sum_{x=l_i}^{u_i} F'_j[x]\cdot(u_i-l_i)^2)$$

Finally, for each dimension j, we find a histogram $\mathcal{H}_j$ that minimizes $\sum_{i=1}^{B}\sum_{x=l_i}^{u_i} F'_j[x]\cdot(u_i-l_i)^2$ by applying Algorithm 4.6.

Multidimensional histogram: A multidimensional histogram $\mathcal{H}_{MD}$ partitions the space into buckets (i.e., bounding rectangles). Given an exact point p, we compute its approximate point as $p' = \mathcal{H}_{MD}(p)$, which denotes the identifier of the bucket enclosing p.

Due to the curse of dimensionality, a multidimensional histogram is not

effective for approximation. As such, we do not bother to extend our solution for multidimensional histogram. Instead, we use an R-tree based multidimensional histogram and test its effectiveness in the experimental study.

Global vs. Multidimensional histogram: Observe that each approximate point p' is associated with a bounding rectangle. Let w_{br} be the average width of dimensions of a bounding rectangle. We proceed to compare the value of w_{br} for the global histogram and the multidimensional histogram. In the following discussion, we assume that data points fall in the space $[0, 1]^d$ with uniform distribution.

We take an equi-width histogram as a simple global histogram. The number of buckets is 2^τ, where τ is the code length. Then we derive: $w_{br} = \frac{1}{2^\tau}$. Note that this value is independent of the dimensionality d.

Assume that a multidimensional histogram partitions the d-dimensional space such that each rectangle contains at least 2 points (out of n points in the dataset). The average volume of each rectangle is at least $\frac{2}{n}$. Thus, we derive $w_{br} \geq (\frac{2}{n})^{\frac{1}{d}}$. Unfortunately, this value rises rapidly with the dimensionality d.

As an example, we set the data size $n = 1000000$, the dimensionality $d = 100$, and the code length $\tau = 8$. For the global histogram (equi-width), we have: $w_{br} = \frac{1}{2^8} = 0.0039$. For the multidimensional histogram, we have: $w_{br} \geq (\frac{2}{1000000})^{\frac{1}{100}} = 0.877$. This example demonstrates that the global histogram achieves a much smaller w_{br} than the multidimensional histogram.

4.4 Cost Estimation Model

Section 4.4.1 estimates the I/O cost of our proposed solution, as a function of the cache size $\mathcal{CS}$ and the code length τ. In Section 4.4.2, we derive the optimal code length τ (for a given $\mathcal{CS}$) such that it leads to the lowest I/O cost.

Our analysis is based on two assumptions: (i) the distribution of queries follows that of the historical query workload $\mathcal{WL}$, and (ii) the caching policy is HFF (highest-frequency-first).

Specifically, Highest-frequency-first (HFF) [65] is a static caching policy that creates the cache offline and fixes the cache content at runtime. It places the most frequent items into the cache, where the frequency of each cache item p (i.e., candidate) is derived from the query workload $\mathcal{WL}$ as: $freq(p) = |\{q \in \mathcal{WL} : p \in C(q)\}|$.

4.4.1 I/O Cost Estimation

4.4.1.1 Cost Estimation model

The I/O cost in the candidate refinement phase is decided by the remaining candidate size C_{refine}, which is proportional to $1 - \rho_{hit} \cdot \rho_{prune}$ (by Eqn. 4.1).

For ρ_{prune}, we rewrite it as $1 - \rho_{refine}$ where ρ_{refine} is the ratio that a (cache-hit) candidate requires refinement. By using the query workload $\mathcal{WL}$, we estimate ρ_{refine} as the average value $\sum_{q \in \mathcal{WL}} \rho^q_{refine}/|\mathcal{WL}|$, where ρ^q_{refine} is the candidate refinement ratio for a specific query point q.

We aim to estimate the cache hit ratio ρ_{hit} in Section 4.4.1.2 and the can-

didate refinement ratio ρ^q_{refine} for a specific query point q in Section 4.4.1.3.

4.4.1.2 Estimation of ρ_{hit}

Theorem 4.1 (Estimation of ρ_{hit}) *Let ρ_{hit} and ρ^*_{hit} be the cache hit ratio in our proposed histogram based cache method (with equi-width histogram) and in the exact cache method, respectively. Let N_{item} and N^*_{item} be the number of cache items in our cache and in the exact cache, respectively. Let $|\mathcal{P}|$ be the dataset cardinality, and L_{value} be the number of bits for representing a data value. We have:*

$$\rho_{hit} \begin{cases} \leq \frac{L_{value}}{\tau} \cdot \rho^*_{hit} & (if\ N_{item} < |\mathcal{P}|) \\ = 1 & (otherwise) \end{cases}$$

Proof. First, we have: $N_{item} \cdot \tau = {N_{item}}^* \cdot L_{value}$.

The proof for the case $N_{item} \geq |\mathcal{P}|$ is trivial. We thus focus on the case $N_{item} < |\mathcal{P}|$. Let f_i be the query frequency of data point i (according to the query log $\mathcal{WL}$). Without loss of generality, for the HFF caching policy, we arrange the points in the cache in descending frequency order, i.e., $f_1 \geq f_2 \geq \cdots \geq f_n$.

According the definition of hit ratio in HFF, we obtain: $\rho_{hit} = \frac{\sum_{i=1}^{N_{item}} f_i}{\sum_{i=1}^{|\mathcal{P}|} f_i}$ and $\rho^*_{hit} = \frac{\sum_{i=1}^{N^*_{item}} f_i}{\sum_{i=1}^{|\mathcal{P}|} f_i}$

Consider the ratio:

$$\begin{aligned}\frac{\rho^*_{hit}}{\rho_{hit}} &= \frac{\sum_{i=1}^{N^*_{item}} f_i}{\sum_{i=1}^{N_{item}} f_i} = \frac{N^*_{item} \cdot \frac{\sum_{i=1}^{N^*_{item}}(f_i)}{N^*_{item}}}{N_{item} \cdot \frac{\sum_{i=1}^{N_{item}}(f_i)}{N_{item}}} \\ &\geq \frac{N^*_{item}}{N_{item}} \cdot 1 \quad \text{(by Lemma 4.4)} \\ &= \frac{\tau}{L_{value}}\end{aligned} \tag{4.6}$$

Thus, we obtain: $\rho_{hit} \leq \frac{L_{value}}{\tau} \cdot \rho^*_{hit}$ □

Lemma 4.4 (Average weight monotone non-increasing)

$$\forall N^*_{item} \leq N_{item}, \quad \frac{\sum_{i=1}^{N^*_{item}}(f_i)}{N^*_{item}} \geq \frac{\sum_{i=1}^{N_{item}}(f_i)}{N_{item}}$$

Proof. Trivial. □

4.4.1.3 Estimation of ρ^q_{refine} upper bound

Theorem 4.2 (Estimation of ρ^q_{refine} upper bound) *Given a query point q, let b be its k-th upper bound candidate, let D_{max} be the largest candidate distance from q, and let $g_q(x)$ be the probability density function of candidate distances from q. If $g_q(x)$ follows the uniform distribution, then:*

$$\rho^q_{refine} \leq \min\{\frac{||\overrightarrow{\epsilon(b^q_k)}||}{D_{max}}, 1\}$$

Proof. First, we estimate ρ^q_{refine} as:

$$\rho^q_{refine} = \frac{\int_{dist_q(b^q_k)}^{ub^q_k} g_q(x)dx}{\int_0^{D_{max}} g_q(x)dx} = \frac{dist_q(b^q_k)^+ - dist_q(b^q_k)}{D_{max}}$$

According to $dist^+_q(c) - dist_q(c) \leq ||\overrightarrow{\epsilon(c)}||$ (proved in Appendix A), we have:

$$\rho^q_{refine} \leq \frac{||\overrightarrow{\epsilon(b^q_k)}||}{D_{max}}$$

In addition, since $\rho^q_{refine} \leq 1$, we complete the proof. □

Remark: Although we assume $g_q(x)$ to be uniform distribution in the above equation, our estimation is still quite accurate, as shown in our experimental study.

4.4.2 Determining the Optimal τ

For any histogram, we can apply the I/O cost estimation equations in Section 4.4.1 for each τ (from 1 to 32) and then choose the one that gives the lowest estimated I/O cost.

For the equi-width histogram, we provide a closed-form equation to estimate the optimal τ in constant time.

4.4.2.1 ρ^q_{refine} upper bound, for equi-width

For equi-width histogram, we estimate ρ^q_{refine} by Theorem 4.3. Since the terms in Theorem 4.3 are independent of q, we can estimate ρ^q_{refine} for equi-width histogram in constant time.

Theorem 4.3 (ρ^q_{refine} upper bound, for equi-width)

$$\rho^q_{refine} \leq \min\{\frac{\sqrt{d}}{D_{max}}\overline{w}, 1\}$$

where $\overline{w} = 2^{L_{value}-\tau}$ is the bucket width of equi-width histogram, and $D_{max} = cR$ is calculated by using the (R, c)-guarantee in the LSH scheme [93, 39].

Proof. By Lemma 4.2, we have: $\rho^q_{refine} \leq \frac{||\overrightarrow{\epsilon(b^q_k)}||}{D_{max}} = \frac{\sqrt{d}}{D_{max}}\overline{w}$. In addition, since ρ^q_{refine} cannot be larger than 1, we complete the proof. □

4.4.2.2 Determining τ, for equi-width

In this section, we derive the optimal τ for the equi-width histogram. Consider the ratio of $\rho_{hit} \cdot \rho_{prune}$ (for our caching) to ρ^*_{hit} (for exact caching). By Lemma 4.3, we have:

$$\begin{aligned}\frac{\rho_{hit} \cdot \rho_{prune}}{\rho^*_{hit}} \approx & \frac{\frac{L_{value}}{\tau} \cdot \rho^*_{hit} \cdot (1 - \frac{\sqrt{d}}{D_{max}}\overline{w})}{\rho^*_{hit}} \\ = & \frac{L_{value}}{\tau} \cdot (1 - \frac{\sqrt{d}}{D_{max}}(2^{L_{value}-\tau}))\end{aligned}$$

Observe that L_{value}, d are known for a given dataset, and D_{max} can be calculated by the LSH scheme and the query workload.

To find the optimal τ, we simply iterate τ for each value in the range $[1..L_{value}]$, evaluate the ratio $(\rho_{hit} \cdot \rho_{prune})/\rho^*_{hit}$, and then report the τ value leading to the highest ratio.

4.5 Experimental Study

In this section, we experimentally evaluate the performance of our proposed solutions and baseline solutions. Section 4.5.1 introduces the experimental setting. Section 4.5.2 studies the sensitivity of solutions for different configurations (e.g., ordering of the dataset file, caching policy, categories of histograms). Section 4.5.3 demonstrates the accuracy of our estimation equations. Section 4.5.4 compares the performance of solutions with respect to various parameters.

All experiments are conducted on a PC with Intel i7-4770 3.40GHz CPU, 16G RAM, and 64-bit Ubuntu 13.04 operating system. The page (block) size in this system is 4KB(4,096 bytes). All algorithms were implemented in C++, and compiled by g++ 4.7.3 with O3 optimization. All datasets and indexes were stored in hard disk and the OS cache was disabled, as in [106].

4.5.1 Experimental Setup

Datasets and queries: Table 4.2 summarizes the real datasets to be used in our experimental study. They store the feature vectors extracted from images. **SOGOU**, with raw data size 635 GB, is extracted from web images indexed by Sogou[8] (an image search engine in China). We followed [20] to extract a 960-

[8] http://www.sogou.com/labs/dl/p2.html

dimensional GIST descriptor from each image. Sogou also provides the query log (of images) for this dataset. We also use two datasets from [86]: **NUS-WIDE** (extracted from Flickr images), and **IMGNET** (extracted from an online image database)[9]

Table 4.2. Dataset information

Dataset	d	# of $\mathcal{P}$	# of $\mathcal{Q}_{test}$	size per point	file size
NUS-WIDE	150	267,415	50	600 bytes	136 MB
IMGNET	150	2,213,937	50	600 bytes	1.26 GB
SOGOU	960	8,304,965	50	3,840 bytes	29.7 GB

Next, we split the query log into: (i) a query workload $\mathcal{WL}$, and (ii) a testing query set $\mathcal{Q}_{test}$. A sufficiently large $\mathcal{WL}$ is used to populate the cache (in Section 4.2.1), and to construct the histogram (in our solutions).

In each experiment, we execute the queries in $\mathcal{Q}_{test}$ and measure the average query response time per query. We follow [93, 39] and fix the size of $\mathcal{Q}_{test}$ to 50.

Methods for comparison: We consider three baseline methods: NO-CACHE (not using cache), EXACT (caching exact points), and C-VA (caching the whole VA-file)[10]. For C-VA, we tune the number of bits per point so that the VA-file fits into the cache. According to [97], the encoding scheme of VA-file is the same as Equi-Depth.

Our proposed histogram-caching methods share the prefix HC in their names, and apply the following histograms as stated in Section 4.3.1.

- *Global histogram:* HC-W (equi-width), HC-D (Equi-Depth), HC-V (V-

[9] http://staff.itee.uq.edu.au/shenht/UQ_IMH/index.htm. The feature vector of them are 150 dimensions color histogram. However, they do not have real query logs. Following [93, 39], we generate the query log by picking random points from $\mathcal{P}$, and then remove those points from $\mathcal{P}$.

[10] We use VA-file instead of VA$^+$-file. VA$^+$-file [37] requires Karhunen Loeve Transform (KLT), which is not scalable for huge matrices on our datasets.

Optimal), HC-O (our optimal histogram for kNN search).

- *Individual-dimension histogram:* iHC-*.

  It uses d histograms. For each dimension j, it builds a histogram $\mathcal{H}_j$ by the corresponding HC-* method.

- *Multidimensional histogram:* mHC-R.

  First, we build an R-tree with 2^τ leaf nodes (by using a corresponding node fanout). Then, we map the MBR of each leaf node to a bucket.

All methods use the same index $\mathcal{I}$ in the same experiment. In most experiments, we employ C2LSH [39] as the index $\mathcal{I}$. We use the C2LSH implementation in [39] and its parameter tuning functions. At the end of Section 4.5.4, we employ exact kNN search indexes (iDistance and VA-file).

Parameters setting: Unless otherwise stated, we use the following default parameter values. The default result size is $k = 10$. The default cache size $\mathcal{CS}$ is set as 40 MB, 400 MB, 8192 MB for NUS-WIDE, IMGNET, SOGOU, less than 30% of the size, respectively. The default code length, $\tau = 10$, is estimated by using our equations in Section 4.4. We construct our HC-O histogram by Algorithm 2. By default, we run each method by a single thread in each experiment.

4.5.2 Effect of Configurations

Besides the above parameters, the configurations in our solutions include: ordering of the dataset file $\mathcal{P}$, caching policy, and the categories of histograms. We proceed to examine the sensitivity of these choices on our solutions. We

conduct experiments on the SOGOU dataset with the default parameter setting. The experimental results on NUS-WIDE and IMGNET are similar; we omit them due to space reasons.

4.5.2.1 Effect of caching policy

The caching policy is an choice in our solution. We compare the HFF and LRU policies as described in Section 4.2.2. As shown in Figure 4.8, HFF performs better than LRU. Hence, we set HFF as default caching policy in subsequent experiments.

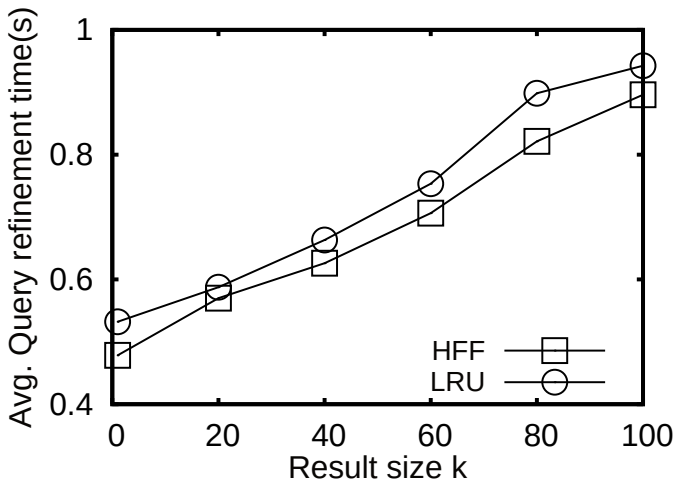


Figure 4.8. Effect of caching policy, EXACT caching

4.5.2.2 Effect of dataset file ordering

We investigate whether the physical ordering of the dataset file $\mathcal{P}$ affects the candidate refinement time T_{refine}. We compare three orderings: (i) the raw ordering in the dataset, (ii) the clustered ordering, which uses the iDistance

ordering [52]. (iii) the sorted key ordering, which uses the SK-LSH ordering [104]. In this experiment, we use the EXACT caching method on SOGOU; we obtained similar results for other methods. Figure 4.9 reports the query refinement time for these orderings. For caching policy HFF, different orderings (Raw, Clustered and Sortedkey) have similar performance. Thus, we use the raw ordering in subsequent experiments.

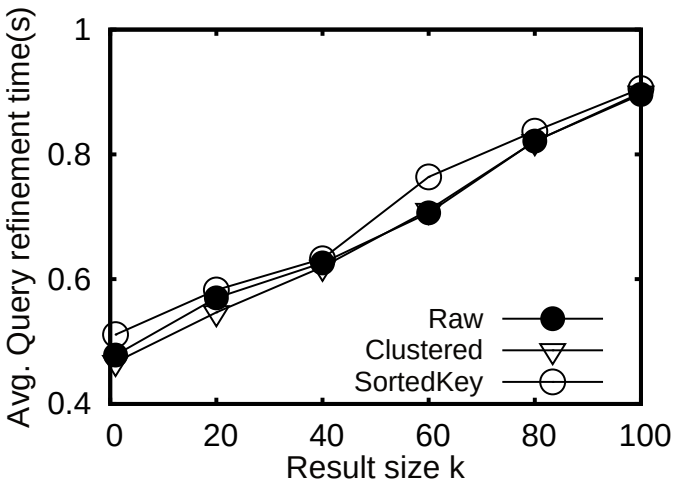


Figure 4.9. Effect of dataset file ordering, EXACT caching

4.5.2.3 Effect of histogram categories

Next, we compare the global histograms (HC-W, HC-D, HC-O) and the individual-dimension histograms (iHC-W, iHC-D, iHC-O), and a multidimensional histogram (mHC-R).

Table 4.3 shows the histogram space (KB), the histogram construction time (s), and the average candidate refinement time T_{refine} during kNN search. Due to the curse of dimensionality, mHC-R is not effective. Global histograms and

individual-dimension histograms have similar T_{refine}. However, individual-dimension histogram suffers from high construction time and occupies more space. For example, it takes 23.8 days to construct iHC-O, but only 35.7 minutes to construct HC-O. Thus, we only use global histograms (HC-W,HC-D, HC-O) in following experiments.

Table 4.3. Effect of histogram categories, on SOGOU

	HC-W	iHC-W	HC-D	iHC-D	HC-O	iHC-O	mHC-R
Space (KB)	8	1,200	8	1,200	8	1,200	1,204
Construction time (s)	0.000	0.004	300	2233	2,140	2.1e6	57.6
Average T_{refine} (s)	0.237	0.230	0.164	0.162	0.123	0.113	0.842

4.5.2.4 Effect of caching the whole VA-file

We compare methods C-VA and HC-D in Figure 4.10. While the SOGOU dataset occupies 29.7 GB, we only vary the cache size in the range 1024–6144 MB, which corresponds to 3.4–20 % of the data. At small cache size, C-VA incurs higher time than HC-D because C-VA caches all points but with fewer bits per point. At large cache size, C-VA and HC-D have similar performance since they maintain cache histogram by equi-depth method. Hence, we ignore C-VA in following experiments.

4.5.3 Cost Estimation

Now we test the accuracy of cost estimation model developed in Section 4.5.3. Figure 4.12 shows the estimated and the measured I/O cost of the method HC-W, as a function of the code length τ. Observe that the estimated cost is close to the measured cost. Also, the default code length ($\tau = 10$) derived from our

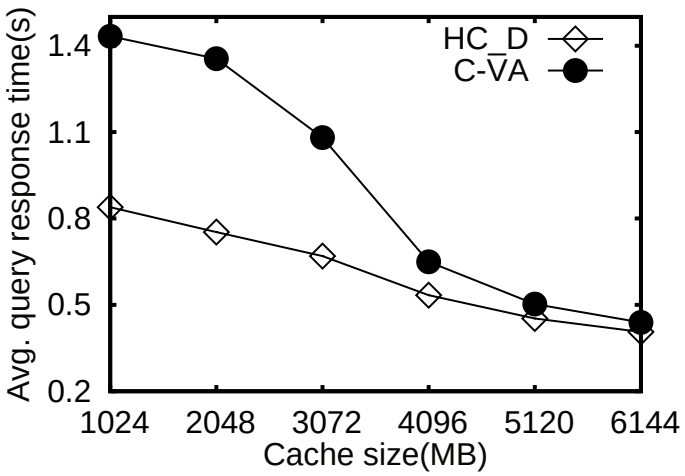


Figure 4.10. C-VA and HC-D comparison

cost model is close to the optimal τ measured in the experiment. We also show the optimal τ for each method on each dataset in Table 4.4.

Table 4.4. Avg. refinement time (s) at default $\tau = 10$ and at optimal τ^*

Dataset	EXACT	HC-W			HC-V		
		Default	Optimal	τ^*	Default	Optimal	τ^*
NUS-WIDE	0.3115	0.0451	0.0451	10	0.0555	0.0555	10
IMGNET	0.3709	0.0672	0.0495	11	0.0203	0.0182	11
SOGOU	0.4803	0.2368	0.2368	10	0.2173	0.1864	8

Dataset	HC-D			HC-O		
	Default	Optimal	τ^*	Default	Optimal	τ^*
NUS-WIDE	0.0110	0.0110	10	0.0087	**0.0087**	10
IMGNET	0.0129	0.0112	11	0.0086	**0.0071**	11
SOGOU	0.1639	0.0839	8	0.1274	**0.0468**	8

4.5.4 Performance Improvement

In this section, we study the performance of our methods with respect to different parameters.

4.5.4.1 The power of early pruning

Early pruning (including true hit detection) plays an important role in our solution. In this experiment, we study the effectiveness of different histograms for supporting early pruning.

Figure 4.11 shows the remaining candidate size of the methods with respect to the number of I/O accesses. The performance of mHC-R (R-tree histogram) is bad due to the curse of dimensionality. Observe that HC-O (using our histogram metric) achieves the best performance. On the other hand HC-V (using the SSE histogram metric [51]) does not minimize the I/O cost. In subsequent experiments, we ignore mHC-R due to its bad performance.

Remark: HC-O incurs lower I/O cost than HC-D by 50%.

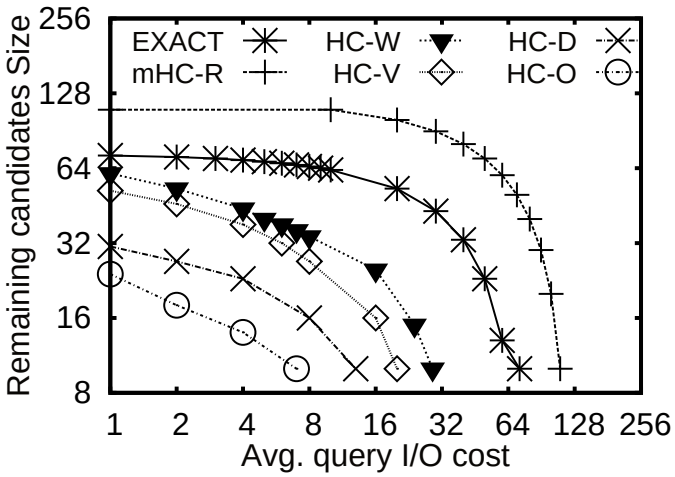


Figure 4.11. Remaining candidate size vs query I/O cost, axes in logscale

Table 4.4 reports the average refinement time of the methods on all datasets with default parameter setting. First, our method HC-O is faster than EXACT by an order of magnitude. Second, although the default code length $\tau = 10$ is not

always the optimal (τ^*), our methods still perform much better than EXACT. In subsequent experiments, we ignore HC-V as its performance is unstable; it is worse than HC-W on NUS-WIDE and better than HC-W on IMGNET and SOGOU.

4.5.4.2 Effect of the cache size $\mathcal{CS}$

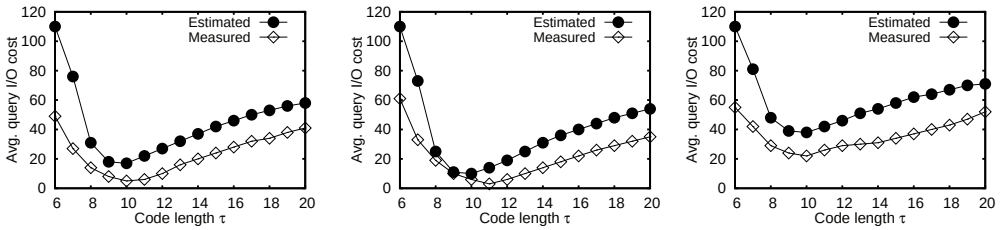
Figure 4.13 plots the average query response time of different methods for different cache size $\mathcal{CS}$. Our caching methods outperform the EXACT caching method. They achieve the best performance when the cache size reaches only 1/3 of the dataset file size. HC-O is the best among all methods. We ignore the baseline methods (NO-CACHE and EXACT) in the following experiments.

4.5.4.3 Effect of the result size k

Then we examine the effect of the result size k on our methods. For readability, we plot the average query response time in log scale in Figure 4.14. The query response time of all methods rises as k increases. HC-O is the best, followed by HC-D, and then HC-W. This result also confirms the effectiveness of our proposed histogram metric (used in HC-O).

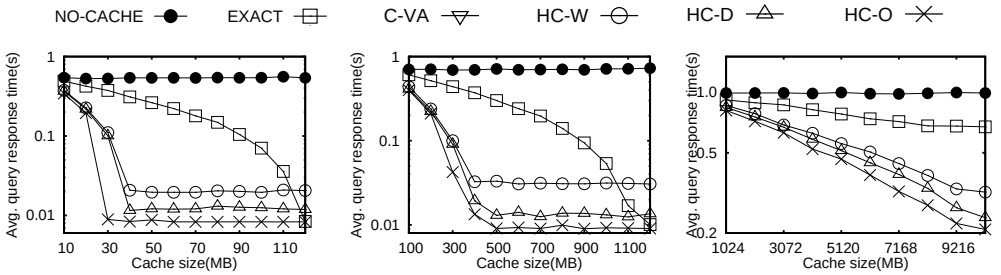
4.5.4.4 Effect of the code length τ

The next experiment investigates the effect of code length τ on our methods. Figure 4.15 (a),(b),(c) show the average values of $\rho_{hit} \cdot \rho_{prune}$, query I/O cost, and refinement time respectively. Due to the space limit, we only show the



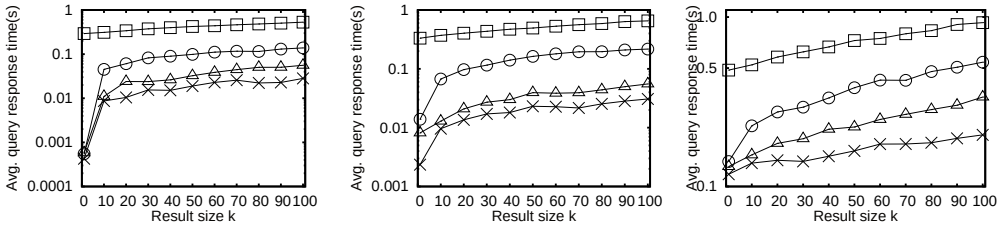
(a) NUS-WIDE (b) IMGNET (c) SOGOU

Figure 4.12. The estimated and the measured query I/O cost of HC-W vs. τ, k, $\mathcal{CS}$ at default setting



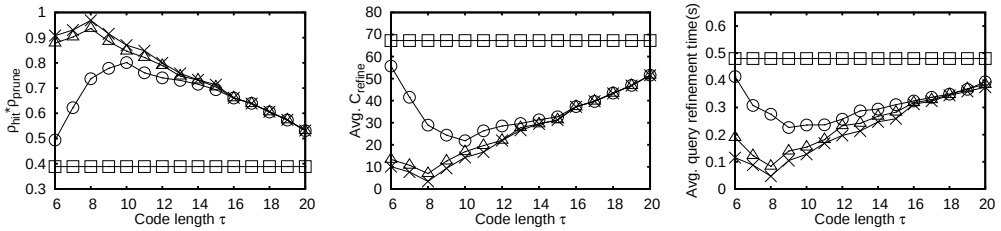
(a) NUS-WIDE (b) IMGNET (c) SOGOU

Figure 4.13. Average response time (in logscale) vs. cache size $\mathcal{CS}$, k, τ at default setting



(a) NUS-WIDE (b) IMGNET (c) SOGOU

Figure 4.14. Average response time (in logscale) vs. result size k, τ, $\mathcal{CS}$ at default setting



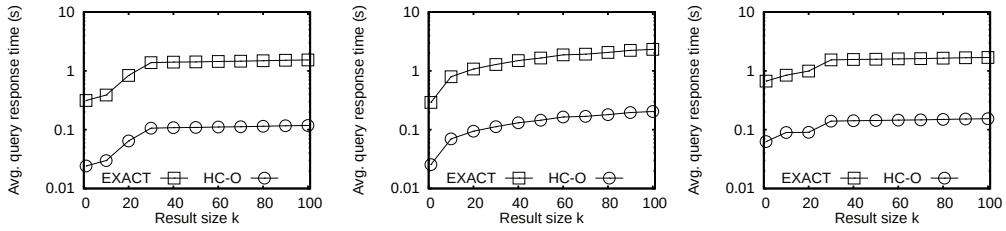
(a) $\rho_{hit} \cdot \rho_{prune}$ (b) I/O cost (c) Refinement time(s)

Figure 4.15. Performances vs. code length τ, on SOGOU, k, $\mathcal{CS}$ at default setting

experimental results on the largest dataset (SOGOU, 29.7G). Observe that different methods can have different optimal values for τ. For example, the optimal τ for HC-W, HC-D, HC-O are 10, 8, 8, respectively. Again, HC-O is the best and its performance is more robust to τ (e.g., at small τ).

4.5.4.5 Experiments on exact kNN search indexes

Finally, we compare the performance of HC-O and EXACT on three exact kNN search indexes: iDistance[11], VA-file[97] and VP-tree [19] on IMGNET. Figure 4.16 shows that the query cost of HC-O is lower than EXACT caching by at least an order of magnitude.



(a) iDistance (b) VA-file (c) VP-tree

Figure 4.16. Exact kNN search indexes, on IMGNET

4.6 Chapter Summary

In high-dimensional kNN search, both exact and approximate kNN solutions incur considerable time in the candidate refinement phase. In this chapter, we investigate a caching solution to reduce the candidate refinement time. Our caching method HC-O is faster than EXACT caching by at least an order of

[11] We use the implementation from:https://code.google.com/p/idistance/

magnitude, on an approximate index (C2LSH) and on exact indexes (iDistance, VP-tree and VA-file).

It is worth noting that our approach is general for any index and achieves promising performance whenever the candidate refinement phase incurs significant time, including other kNN methods [10, 11, 99, 40, 20]. In future, we plan to extend our caching techniques for advanced operations (e.g., kNN join, density-based clustering) on high-dimensional data.

In addition, this chapter investigates how to efficiently utilize main memory. It provides hints to optimize the performance of algorithms in Chapter 5.

Chapter 5

Extracting Top-K Insights from Multidimensional Data

5.1 Introduction

OLAP tools facilitate enterprise knowledge workers (e.g., executives, managers, and analysts) on decision making in business intelligence applications. Their interfaces allow users to navigate the aggregation result by operations (e.g., slicing, dicing, drill up/down). Nevertheless, these tools still require users to specify the dimensions (i.e., group-by attributes) in OLAP queries. This analysis process requires tedious hit-and-trial from the user, on manually posing queries, analyzing results and deciding what is interesting [96]. To alleviate this issue, semi-automatic methods [77, 82] can be used to detect local anomalies or interesting views in an OLAP cube; however, these methods still require the user to specify a target (e.g., an OLAP cell or a dimension). Recent vision

papers [24, 96] and the industry [23] have called for automatic techniques to obtain insights from data, helping users when they are not clear on what they are looking for [81, 46].

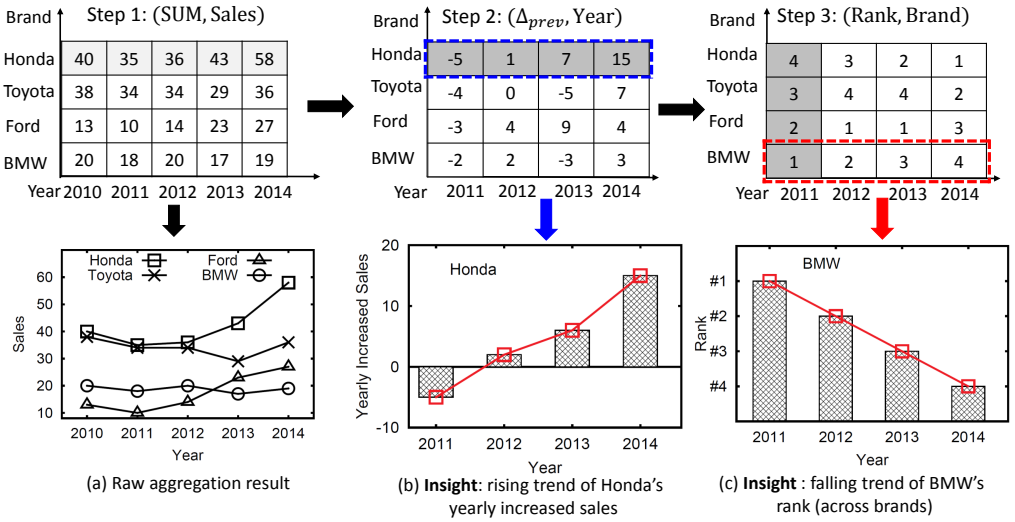
Taking a step further, in this chapter, we take the first attempt to extract interesting insights from facts in a multidimensional dataset (e.g., sales data). We plan to: (i) formulate the concept of insight and propose a meaningful scoring function for it, and (ii) provide efficient solutions to compute top-k insights automatically.

Suppose that we have a car sales dataset with the schema (Year, Brand, Category, Sales). OLAP tools support aggregation on data (e.g., the SUM of Sales by Year and Brand, cf. Step 1 in Figure 5.1). However, aggregation alone does not reveal much information (e.g., clear trend), as illustrated in Figure 5.1(a). In this work, we consider *insight* as an interesting observation derived from aggregation in multiple steps. In following examples, we demonstrate how insights provide valuable information by performing analysis operations (e.g., rank, difference [75]) over aggregation.

Example 1. (Yearly increased sales): We may compare the growth of different brands by the yearly increased sales (cf. Step 2 in Figure 5.1). In Figure 5.1(b), we observe that the yearly increased sales of Honda is rising with years. The aggregation result of the sales of Honda (cf. Figure 5.1(a)) first drops and then rises, whose trend is not intuitive to understand. In contrast, our insight (cf. Figure 5.1(b)) provides a clear rising trend. This insight can be effectively used to reveal the potential market and seek profits.

Example 2. (The rank of yearly increased sales among brands): Re-

garding the brand BMW, we have not found any "interesting" information about it from raw aggregation (cf. Step 1) and yearly increased sales (cf. Step 2). Nevertheless, we can obtain an insight by applying analysis operations on the yearly increased sales. For example, if we rank the yearly increased sales across brands (cf. Step 3), we derive: "the rank of BMW's (across brands) yearly increased sales falls with years", as shown in Figure 5.1(c). Such insight implies that the competitiveness of BMW decreases with years.



(a) Raw aggregation result

(b) **Insight**: rising trend of Honda's yearly increased sales

(c) **Insight** : falling trend of BMW's rank (across brands)

Figure 5.1. Example of insights

The above examples illustrate typical insights extracted from multidimensional data. These insights have two usages in business intelligence applications. First, they provide informative summaries of the data to non-expert users, who do not know exactly what they are looking for [81, 46]. For example, a car seller is looking for interesting patterns in car sales, without knowing the exact patterns in advance. Insights can provide quick and informative summaries of interesting observations in the data and reveal interesting patterns (e.g., the

sales of SUV is the best across Category in BMW). Second, data analysts may customize insights by their needs to guide directions for data exploration. For example, a data analyst focusing on BMW brand may wish to find insights related to BMW, e.g., the rank of BMW's yearly increased sales falls with year (as insight in Figure 5.1(c)). Then, he will continue to investigate the reason why such a trend happens.

Besides the above insights, the dataset may contain many other insights for other combinations of group-by attributes and analysis operations. Even with the aid of OLAP tools, it is tedious for data analysts to enumerate all possible insights and evaluate their "importance" manually. Motivated by this, we propose the **top-k insight problem**, which returns the best k insights with respect to an importance measure. Our problem does not require users to specify any input (e.g., group-by attributes, analysis operations in each step). Although there exist several works on extending the capability of OLAP [77, 102, 101, 1], they do not support automatic extraction of our proposed insights. We leave the discussion of these works in the related work section (cf. Table 2.2).

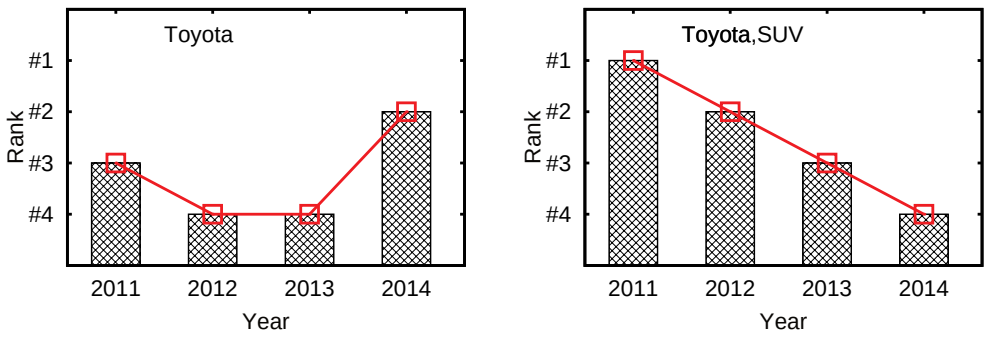
Our problem poses challenges on (i) the effectiveness of the result and (ii) the efficiency of computation. Regarding the effectiveness aspect, existing OLAP tools are lacking means to evaluate the importance of insights. We need a meaningful scoring function that supports generality and comparability among different insights. On the other hand, the efficiency aspect involves the following three technical challenges.

C1) Huge search space: The search space is exponential to the number of dimension attributes d and the depth of insight τ (i.e., the number of steps

in an insight). Also, the search space is polynomial to the domain sizes of dimensions and the combinations of analysis operations. We leave the analysis in Section 5.2.3.

C2) Expensive insight computation: The evaluation of an insight requires applying multiple analysis operations after aggregation (e.g., Steps 2 and 3 in the above example). Each analysis operation may require accessing multiple values in aggregation results.

C3) Non-monotonicity of insight score: As we will explain shortly, the insight score function is not monotonic. For example, there is no insight in BMW with yearly increased sales (cf. Step 2), but there is an insight in the rank of BMW (across Brand) with regard to the yearly increased sales, as shown in Figure 5.1(c). Moreover, there is no insight in the rank of Toyota (as Figure 5.2 (a)) in above example; but there is an insight in its children spaces (e.g., ⟨*, Toyota, SUV⟩ in Figure 5.2(b), where SUV is a value for the Category dimension). Such non-monotonicity prevents us from utilizing existing aggregation computation methods [36, 103], which require the function to be monotonic.



(a) subspace ⟨*, Toyota, *⟩ score = 0.0001

(b) subspace ⟨*, Toyota, SUV⟩ score = 0.37

Figure 5.2. Examples of non-monotonicity

While the problem of automatic insight extraction is challenging, we develop efficient evaluation techniques that render insight extraction feasible for large-scale applications (i.e., the execution time of our solution is sub-linear with data size). Specifically, we devise a suite of optimization techniques to reduce the computation cost. The contributions of this work are:

1. Formulate the top-k insight problem (Section 5.2) and propose a meaningful scoring function for insights (Section 5.3);

2. Propose the architecture of our top-k insight extraction system (Section 5.4) and the computation framework (Section 5.5);

3. Design a suite of optimization techniques — pruning, ordering, specialized cube (Section 5.6) and computation sharing (Section 5.7) to speedup insight extraction;

4. Verify the effectiveness of top-k insights on three real datasets by case study and user study (Section 5.8), and demonstrate the efficiency of our proposal (Section 5.9).

The remainder of this chapter is organized as follows. Section 5.2 formulates our problem and Section 5.3 provides a meaningful score for insights. Section 5.4 describes the architecture of our proposed system and discusses its extensibility. Sections 5.5, 5.6 and 5.7 present the computation framework and a suite of performance optimization techniques. Sections 5.8 and 5.9 demonstrate the effectiveness and efficiency of our proposal, respectively. Section 5.10 concludes this work and discusses the further research topics.

5.2 Problem Statement

In this section, we provide formal definitions of the multidimensional data model, composite extractors, and the score function of insights. Finally, we formulate our insight extraction problem and analyze its search space.

5.2.1 Data Model and Subspace

We are given a multidimensional dataset $\mathbb{R}(\mathcal{D}, \mathcal{M})$ where $\mathcal{D} = \langle D_1, ..., D_d \rangle$ is the list of dimension attributes, and $\mathcal{M}$ is the measure attribute. Let $dom(D_i)$ denote the domain of attribute D_i. We assume that each D_i satisfies $|dom(D_i)| > 1$[1].

Consider the entire OLAP cube defined on the dataset $\mathcal{D}$. Given a cube cell, we can describe its attributes' values by a subspace S and its aggregate value by a measure $S.\mathcal{M}$, as defined below.

Definition 5.1 (Subspace) *A* **subspace** *is defined as an array $S = \langle S[1], ..., S[d] \rangle$, where $S[i]$ can take a value in $dom(D_i)$ or the value $*$ (i.e., 'all' values). Given the dataset $\mathcal{D}$, the aggregate measure $S.\mathcal{M}$ is defined as the aggregation of tuples in $\mathcal{D}$ that match with S.*

For simplicity, we also call $S.\mathcal{M}$ as the measure of subspace S.

It is convenient to analyze the change of cube cells (i.e., subspaces) by varying a single dimension. Therefore, we define a *sibling group* to cover subspaces that differ on a single dimension only.

[1] We discard any attribute D_i with $|dom(D_i)| = 1$ because such an attribute is not meaningful for analysis.

Definition 5.2 (Sibling group) *Given a subspace S and a dimension D_i, a* **sibling group** *is defined as* $\mathsf{SG}(S, D_i) = \{S' : S'[i] \neq * \ \wedge \ \forall j \neq i, S'[j] = S[j]\}$, *i.e., a set of subspaces that differ on D_i only. We also call D_i as a dividing dimension for* $\mathsf{SG}(S, D_i)$.

Example: Table 5.1 illustrates an example dataset (car sales). It contains two dimensions (Year, Brand) and a measure (Sales). By fixing the year (to 2010) and varying the brand, we can compare the sales of different brands in the same year: ⟨2010,Ford⟩, ⟨2010,BMW⟩, ⟨2010,Honda⟩, ⟨2010,Toyota⟩. These four subspaces belong to the sibling group $\mathsf{SG}(\langle 2010,*\rangle,\text{Brand})$.

Tuples	Tuples	Tuples	Tuples
2010, Ford, 13	2010, BMW, 20	2010, Honda, 40	2010, Toyota, 38
2011, Ford, 10	2011, BMW, 18	2011, Honda, 35	2011, Toyota, 34
2012, Ford, 14	2012, BMW, 20	2012, Honda, 36	2012, Toyota, 34
2013, Ford, 23	2013, BMW, 17	2013, Honda, 43	2013, Toyota, 29
2014, Ford, 27	2014, BMW, 19	2014, Honda, 58	2014, Toyota, 36

Table 5.1. Car sales dataset (Year, Brand, Sales)

5.2.2 Composite Extractor

We shall conduct analysis operations on a sibling group in order to derive an observation. First, we introduce an extractor as a basic analysis operation on a sibling group.

Definition 5.3 (Extractor) *An* extractor ξ *takes a sibling group* $\mathsf{SG}(S, D_x)$ *as input, and computes for each subspace* $S_c \in \mathsf{SG}(S, D_x)$ *a derived measure* $S_c.\mathcal{M}'$ *based on (i)* $S_c.\mathcal{M}$ *and (ii)* $\{(S_{c'}, S_{c'}.\mathcal{M}) : S_{c'} \in \mathsf{SG}(S, D_x)\}$, *i.e., the measures of all subspaces in* $\mathsf{SG}(S, D_x)$.

Inspired by Sarawagi et al. [75, 78] and market share analysis[2], we propose four instances of extractors (i.e., Rank, %, Δ_{avg}, Δ_{prev}) and describe their semantics in Table 5.2. The extractor Δ_{prev} imposes an requirement that D_x must be an ordinal attribute, because $prev_{D_x}(S_c)$ refers to the previous subspace of S_c along D_x. The other extractors are applicable to any type of attribute. In addition to these extractors, we also allow data analysts to define their own extractors for their applications.

Extractor ξ	Derived measure $S_c.\mathcal{M}'$ for S_c	Requirement
Rank	the rank of $S_c.\mathcal{M}$ in $\mathsf{SG}(S, D_x)$	nil
%	% of $S_c.\mathcal{M}$ over the SUM of measures in $\mathsf{SG}(S, D_x)$	nil
Δ_{avg}	$S_c.\mathcal{M}-$ the AVERAGE of measures in $\mathsf{SG}(S, D_x)$	nil
Δ_{prev}	$S_c.\mathcal{M} - prev_{D_x}(S_c).\mathcal{M}$	D_x is ordinal

Table 5.2. List of extractors, with the input $\mathsf{SG}(S, D_x)$

Example: We illustrate the output of the above extractors in Table 5.3. Consider the sibling group $\mathsf{SG}(S$,Year), where S=⟨*,Ford⟩. The extractor Rank computes the rank of each S_c among all years. the extractor % calculates the percentage of each S_c among all years. The extractor Δ_{avg} returns the difference of each S_c from the average measure. The extractor Δ_{prev} obtains the difference of each S_c from its previous subspace along Year.

Sib. group $\mathsf{SG}(S, D_x)$	Measure $S_c.\mathcal{M}$	Derived measure $S_c.\mathcal{M}'$ for			
		Rank	%	Δ_{avg}	Δ_{prev}
⟨2010,Ford⟩	13	4	15%	-4.4	
⟨2011,Ford⟩	10	5	11%	-7.4	-3
⟨2012,Ford⟩	14	3	16%	-3.4	4
⟨2013,Ford⟩	23	2	27%	5.6	9
⟨2014,Ford⟩	27	1	31%	9.6	4

Table 5.3. Examples for extractors

[2] https://en.wikipedia.org/wiki/Market_share_analysis

We then introduce a *composite extractor* $\mathcal{C}_e$ to capture a multi-step analysis operator on a sibling group.

Definition 5.4 (Composite extractor) *Given a depth parameter τ, a* **composite extractor** *$\mathcal{C}_e$ is defined as a length-τ array of pairs $(\mathcal{C}_e[i].\xi, \mathcal{C}_e[i].D_x)$ such that it satisfies: (i) $\mathcal{C}_e[1].\xi$ is an aggregate function and $\mathcal{C}_e[1].D_x$ is the measure attribute $\mathcal{M}$, (ii) each $\mathcal{C}_e[i]$ $(i > 1)$ is an extractor, and (iii) adjacent extractors are compatible.* [3]

Composition Taxonomy: We give a well-defined composition closure to ensure the validity of generated composite extractors, as shown in Table 5.4. The result of extractors in the first column serves as the input of extractors in the first row. '✓' means a meaningful composition, '✗' means a meaningless composition.

Example: The composite extractor $\mathcal{C}_e$ = ⟨SUM, (Δ_{prev},Year), (*Rank*, Brand)⟩ is valid according to Table 5.4. It corresponds to the semantic that ranks the value of "difference from previous year" across all brands in the dataset in Table 5.1. However, the composite extractor $\mathcal{C}_e$ = ⟨SUM, (*Rank*, Year), (%, Brand)⟩ is invalid, because it does not make sense to calculate the percentage of ranking positions.

ξ	*Rank*	%	Δ_{avg}	Δ_{prev}
Rank	✓	✗	✓	✓
%	✓	✗	✓	✓
Δ_{avg}	✓	✗	✓	✓
Δ_{prev}	✓	✗	✓	✓

Table 5.4. Composition taxonomy for adjacent extractors

[3] Two extractors $\mathcal{C}_e[i]$ and $\mathcal{C}_e[i+1]$ are compatible if the output set of $\mathcal{C}_e[i]$ can be used as the input of $\mathcal{C}_e[i+1]$.

We assume that the aggregate function is SUM on the measure attribute $\mathcal{M}$ in the dataset. Nevertheless, we will consider other aggregate functions and multiple measure attributes in Section 5.4.2. The depth parameter τ captures the complexity of a composite extractor. When $\tau = 1$, a composite extractor is the same as the aggregate function. We recommend to set τ to 2 or 3, which are analogous to first-order and second-order derivatives in Mathematics, respectively. For the examples in Figure 5.1, we can express steps 1–2 by a depth-2 composite extractor $\mathcal{C}_e = \langle(\mathsf{SUM},\text{Sales}),(\Delta_{prev},\text{Year})\rangle$, and express steps 1–3 by a depth-3 $\mathcal{C}_e = \langle(\mathsf{SUM},\text{Sales}),(\Delta_{prev},\text{Year}),(\text{Rank, Brand})\rangle$.

Next, we define the result set of applying a composite extractor $\mathcal{C}_e$ on a sibling group $\mathsf{SG}(S, D_a)$, as in Definition 5.5.

Definition 5.5 (Result set of composite extractor) *A* **composite extractor** $\mathcal{C}_e$ *takes sibling group* $\mathsf{SG}(S, D_a)$ *as input, and computes the result set* $\Phi = \{(S', S'.\mathcal{M}_\tau) : S' \in \mathsf{SG}(S, D_a)\}$*, where the* $S_c.\mathcal{M}_i$ *denotes the level-*i *derived measure of a subspace* S_c *with respect* $\mathcal{C}_e[i]$*. The value of* $S_c.\mathcal{M}_i$ *is defined recursively as follows.*

At any level $i > 1$*, we obtain each* $S'.\mathcal{M}_i$ *by applying the extractor* $\mathcal{C}_e[i].\xi$ *on the set* $\{(S_c, S_c.\mathcal{M}_{i-1}) : S_c \in \mathsf{SG}(S', \mathcal{C}_e[i].D_x)\}$.

At level i=1, $S'.\mathcal{M}_1$ *is the aggregate result on the measure* $\mathcal{M}$.

Example: Figure 5.3 shows the result set Φ after applying the composite extractor $\mathcal{C}_e = \langle(\mathsf{SUM},\text{Sales}),(\%,\text{Year})\rangle$ on the sibling group $\mathsf{SG}(\langle*,\text{Ford}\rangle,\text{Year})$. For example, at level 2, the derived measure of S_c=$\langle$2014, Ford$\rangle$ is: $S_c.\mathcal{M}_2 = \frac{S_c.\mathcal{M}_1}{\sum_{S' \in \mathsf{SG}(\langle*,\text{Ford}\rangle,\text{Year})} S'.\mathcal{M}_1} = \frac{27}{13+10+14+23+27} = 31\%$, as illustrated in Figure 5.3,

where $S'.\mathcal{M}_1$ is $\mathsf{SUM}(S')$. We will propose an algorithm to compute the result set in Section 5.5.2.

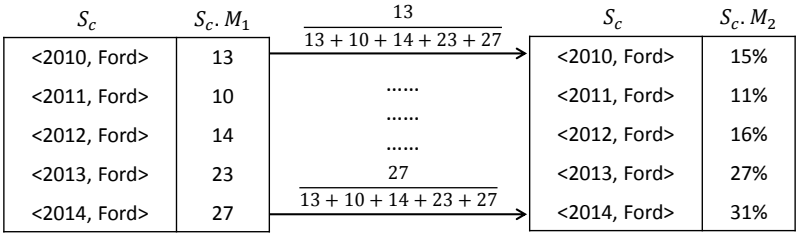


Figure 5.3. Example of composite extractor computation

In some cases, a composite extractor is not applicable to some sibling groups. For example, the composite extractor $\langle$(SUM,Sales),(%,Year)$\rangle$) cannot be applied on the sibling group $\mathsf{SG}(\langle$*,*$\rangle$,Brand) because the subspaces in the group do not have known values in the dimension 'Year'. We discuss how to test the validity of a composite extractor $\mathcal{C}_e$ on a sibling group $\mathsf{SG}(S, D_i)$ as follows.

Definition 5.6 (validity of $\mathsf{SG}(S, D_i)$ and $\mathcal{C}_e$) *A sibling group $\mathsf{SG}(S, D_a)$ is a* **valid input** *for composite extractor $\mathcal{C}_e$ iff for each pair (ξ, D_x) in $\mathcal{C}_e$, D_x=D_a or $S[D_x] \neq *$.*

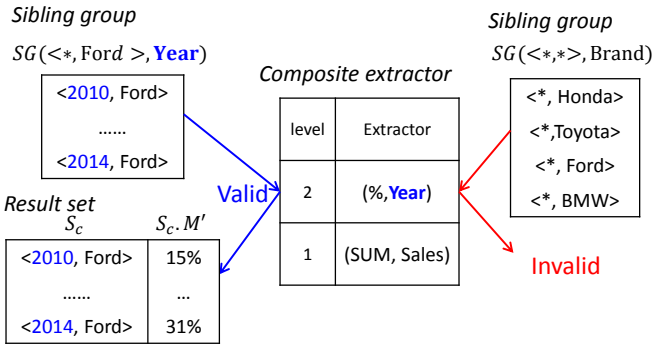


Figure 5.4. Example of $\mathsf{SG}(S, D_i)$ and $\mathcal{C}_e$

Example: In Figure 5.4, a sibling group $\mathsf{SG}(\langle *,\text{Ford}\rangle,\text{Year})$ is valid for the composite extractor $\mathcal{C}_e = \langle(\mathsf{SUM},\text{Sales}),(\%,\text{Year})\rangle)$, as the dimension 'Year' in $\mathcal{C}_e[2]$ has known values in every subspace S_c of $\mathsf{SG}(\langle *,\text{Ford}\rangle,\text{Year})$. However, the sibling group $\mathsf{SG}(\langle *,*\rangle,\text{Brand})$ is not valid for the same $\mathcal{C}_e$ because $\mathsf{SG}(\langle *,*\rangle,\text{Brand})$ does not have known values in the dimension 'Year'.

5.2.3 Problem Definition

Intuitively, business analysts are interested in exceptional facts (e.g., significant differences within a sibling group) and unexpected trends (e.g., rapid rise during a time period). Let Φ be the result set after applying a composite extractor $\mathcal{C}_e$ on a sibling group $\mathsf{SG}(S, D_i)$. We propose to extract two types of "insights" from Φ.

1. **Point insight (outstanding):** Outstanding (No.1 / Last) means that a subspace is remarkably different from others in terms of $S_c.\mathcal{M}_\tau$.
2. **Shape insight (trend):** This insight is applicable when D_i is an ordinal dimension. A rising / falling trend means that $S_c.\mathcal{M}_\tau$ exhibits such a trend when D_i increases.

We are also aware of other types of insights, e.g., those in the Microsoft Power BI product [4]. We will discuss the extensions of our solution for other types of insights in Section 5.4.2.

Formally, we denote a specific insight instance by $(\mathsf{SG}(S, D_i), \mathcal{C}_e, \mathsf{T})$ where T is an insight type. Our problem is to find the top-k insights according to a *score function* $\mathbb{S}(\mathsf{SG}(S, D_i), \mathcal{C}_e, \mathsf{T})$, which we will elaborate in Section 5.3.

Problem 1 (Insight problem) *Given a dataset $\mathcal{R}(\mathcal{D}, \mathcal{M})$ and composite extractor depth τ, find top-k insights $\{(\mathsf{SG}(S, D_i), \mathcal{C}_e, \mathsf{T})\}$ with the highest scores among all possible combinations of sibling groups, composite extractors, and insight types.*

Search Space Size: Before presenting our solutions, we first analyze the search space size of our problem, i.e., the number of possible insights $(\mathsf{SG}(S, D_i), \mathcal{C}_e, \mathsf{T})$, where $\mathsf{SG}(S, D_i)$ is a sibling group, $\mathcal{C}_e$ is a composite extractor, and T is an insight type. In our analysis, let $\mathfrak{D} = \max_{i=1}^{d} |dom(D_i)|$ be the maximum domain size, β be the number of extractor types, and $|\mathsf{T}|$ be the number of insight types. The search space of our solutions is as follows.

Lemma 5.1 (Search space size) *The number of possible insights is $O(|\mathsf{T}| \cdot d \cdot (\beta \cdot d)^{\tau-1} \cdot (\mathfrak{D}+1)^d)$.*

Proof. First, the number of insight types is $|\mathsf{T}|$. For the number of sibling groups, the number of subspaces is $O((\mathfrak{D}+1)^d)$, and there are $O(d)$ choices for D_i.

An extractor (ξ, dim) has $O(\beta \cdot d)$ possible choices. Since a composite extractor contains $\tau - 1$ extractors, there are $O((\beta \cdot d)^{\tau-1})$ possible composite extractors.

By multiplying the above terms, we obtain the number of possible insights: $O(|\mathsf{T}| \cdot d \cdot (\beta \cdot d)^{\tau-1} \cdot (\mathfrak{D}+1)^d)$. □

The scope of this work: The insight problem involves two evaluation metrics: (i) efficiency, and (ii) effectiveness. For efficiency, we propose a computation

framework (in Section 5.5) with optimization techniques (in Sections 5.6, 5.7) to find exact top-k insights efficiently. For effectiveness, we present our methodology to measure the score of an insight (in Section 5.3), and then verify the effectiveness of top-k insights by case study and user study on real datasets (in Section 5.8).

5.3 Meaningful Insight Score

The insight score reflects the *interestingness* of an insight. In order to rank different insights, the insight score metric should exhibit: (i) generality (i.e., applicable to different types of insights), (ii) comparability (i.e, fair across different types of insights).

We first discuss existing works for evaluating the interestingness of information in the literature. In the problem context of [102], the score of a subspace S is defined as:

$$\mathbb{S}(q, S) = \mathsf{Rank}(q, S)^{-1} \cdot \mathsf{ObjCount}(S)$$

where q is a given query tuple, $\mathsf{ObjCount}(S)$ is the number of tuples in S, and $\mathsf{Rank}(q, S)$ is the percentile rank of q among tuples in S. Unfortunately, their score function cannot be readily applied to our insights because (i) our problem does not have any query tuple, (ii) the functions $\mathsf{Rank}(q, S)^{-1}$ and $\mathsf{ObjCount}(S)$ do not capture the characteristics of our insights (e.g., point and shape insights).

The concept of interestingness has also been studied in the context of OLAP cube analysis [77, 76, 100]. All of them define interestingness of a cell value (in the cube) by how surprising that value differs from the expectation. The expectation

is often set by the system in [77, 100], whereas [76] allows a user to specify a list of "known cells" in order to set the expectation for other cells. Following the above works, [28] propose to measure the interestingness of facets in textual documents by using p-value. Unfortunately, all these notions of interestingness are not applicable for our problem as they do not satisfy the properties (e.g., generality, comparability) of insight score.

5.3.1 Insight Score Function

We propose a more appropriate score function for an insight $(\mathsf{SG}(S, D_i), \mathcal{C}_e, \mathsf{T})$ as:

$$\mathbb{S}(\mathsf{SG}(S, D_i), \mathcal{C}_e, \mathsf{T}) = \mathsf{Imp}(\mathsf{SG}(S, D_i)) \cdot \mathsf{Sig}_{\mathsf{T}}(\Phi) \tag{5.1}$$

where Imp measures the impact in the sibling group $\mathsf{SG}(S, D_i)$, $\mathsf{Sig}_{\mathsf{T}}$ measures the significance of type-T insight observed from Φ, and Φ is the result of $\mathcal{C}_e$ on $\mathsf{SG}(S, D_i)$.

The Impact Measure Imp: From the business viewpoint, the impact represents the market share of S. We employ

$$\mathsf{Imp}(\mathsf{SG}(S, D_i)) = \textstyle\sum_{S' \in \mathsf{SG}(S, D_i)} \mathsf{SUM}(S') / \mathsf{SUM}(\langle *, \cdots, * \rangle),$$

so that its value domain is normalized in the interval $[0, 1]$.

The Significance Measure $\mathsf{Sig}_{\mathsf{T}}$: It reveals the uncommonness of an observed insight in the result set Φ. The higher score, the more uncommon/unexpected of that insight. We intend to formulate $\mathsf{Sig}_{\mathsf{T}}$ based on the p-value, which essentially measures how extreme an event is. It also allows fair comparisons among

different types of insights because p-value is already normalized between 0 and 1. In addition, the usage of p-value has been justified by the user-study in [28]. Therefore, we use p-value to measure $\mathsf{Sig}_{\mathsf{T}}$.

5.3.2 The Sig of Insight

In statistics, the p-value is defined as "the probability of obtaining a result equal to or more extreme than what was actually observed, with the given null hypothesis being true" [57]. To achieve generality, we measure the p-value of different types of insights by using different kinds of null hypotheses. We suppose that the null hypotheses for different type of insights are common in real world. We then propose significance functions for point insight and shape insight. The detailed methodologies as follows.

5.3.2.1 Insight evaluation

We describe the significance evaluation procedure for each type of insight in Table 5.5.

Please note that, alternatively, users may customize null hypotheses for personalized analysis and employ their own significance functions. We will discuss them in the extensions in Section 5.4.2.

Measuring Sig of Point Insight. Let $\mathcal{X} = \{x_1, x_2, ..., x_n\}$ be the set of numeric values in the result Φ. In the business domain [8], the sales of products often follow a power-law distribution[4]. Thus, we set the null hypothesis of point insight

[4] https://en.wikipedia.org/wiki/Long_tail

Insight types	Description & Significance definition
Point Insight: *outstanding No.1*	Given a group of numerical values $\mathcal{X} = \{x_1, x_2, ..., x_n\}$, the significance of the biggest value x_{max} being *Outstanding No. 1* of $\mathcal{X}$ is defined by the p-value against the null hypothesis H_0: $\mathcal{X}$ follows a power-law distribution.
Significance calculation: p-value of *outstanding No.1*	i) sort $\mathcal{X}$ in descending order; ii) conduct regression analysis for the values $\mathcal{X}/x_{max}$ using power-law function $\alpha \cdot i^{-\beta}$, where i is index iii) use residuals in regression analysis to train a Gaussian model $N(\mu, \delta)$; iv) obtain the residual ϵ_{max} by $\hat{x}_{max} - x_{max}$ v) calculate the p-value by $P(\epsilon > \epsilon_{max}\|N(\mu, \delta))$. vi) compute the significance of x_{max} by $1 - P(\epsilon > \epsilon_{max}\|N(\mu, \delta))$
outstanding Last	Replace $x_{max} \in \mathcal{X}$ by $x_{min} \in \mathcal{X}$, and the significance calculation is the same as *Outstanding No.1*
Example	cf. Figure 5.7
Shape Insight: *trend*	A time series has an remarkable trend (increase/decrease) with a certain turbulence level (steadily/with turbulence). For a time series $\mathcal{X} = \langle x_1, x_2, ..., x_n \rangle$, the trend insight reflects a relatively sustained trend of $\mathcal{X}_i$. The significance of shape insight is defined by the p-value against the null hypothesis $H_0 : \mathcal{X}$ forms a shape with slope ≈ 0
Significance calculation: trend significance	i) fit $\mathcal{X}$ to a line by linear regression analysis and obtain goodness-of-fit value r^2. ii) compute the *slope* of the $\mathcal{X}$'s fitted line iii) employ a logistic distribution to capture the distributions of slope $L(\mu, \lambda)$, $\mu = 0.2$ and $\lambda = 2$ iv) calculate the p-value by $P(s > \|slope\|\|L(\mu, \delta))$ v) compute the significance of $\mathcal{X}$ by $r^2 \cdot (1 - P(s > \|slope\|\|L(\mu, \delta)))$
Example	cf. Figure 5.8

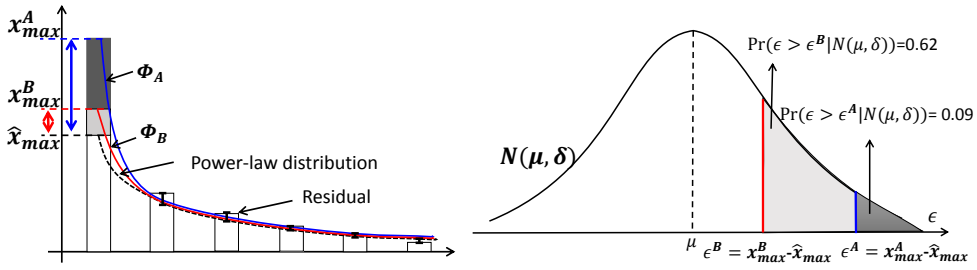
Table 5.5. Insights categories and evaluation procedures

as:

$$H_0 : \mathcal{X} \text{ follows a power-law distribution}$$

The p-value should reveal how surprisingly the maximum value differs from the rest of values in Φ with the hypothesis H_0 is true[5].

First, we sort $\mathcal{X}$ in the descending order and obtain the maximum value x_{max}. Then, we fit the values in $\mathcal{X}/\{x_{max}\}$ to a power-law distribution, like in Figure 5.5(a), where the prediction errors of $x_i \in \mathcal{X}/\{x_{max}\}$ (i.e., subtracting observed value x_i from estimated value $\hat{x}_i$, also called residuals) approximately follow Gaussian distribution $N(\mu, \delta)$. Next, we obtain how surprisingly x_{max} was observed against the hypothesis H_0 is true by (i) deriving x_{max}'s prediction error by $\epsilon_{max} = \hat{x}_{max} - x_{max}$, (ii) calculating the corresponding p-value $p = Pr(\epsilon > \epsilon_{max}|N(\mu, \delta))$, as we depicted in Figure 5.5(b). Finally, we obtain the significance as $\mathsf{Sig}_{\mathsf{T}}(\Phi) = 1 - p$.



(a) Power-law distribution (b) p-value

Figure 5.5. The significance of point insight

We illustrate an example in Figure 5.5. For the result set Φ_A, the prediction error x^A_{max} is large, so we obtain $p = Pr(\epsilon > \epsilon^A_{max}|N(\mu, \delta)) = 0.09$ and derive the significance as $\mathsf{Sig}_{\mathsf{T}}(\Phi_A) = 1-p = 0.91$. For the result set Φ_B, the prediction error x^B_{max} is small, so we derive the significance as $\mathsf{Sig}_{\mathsf{T}}(\Phi_A) = 1-p = 1-0.62 = 0.38$.

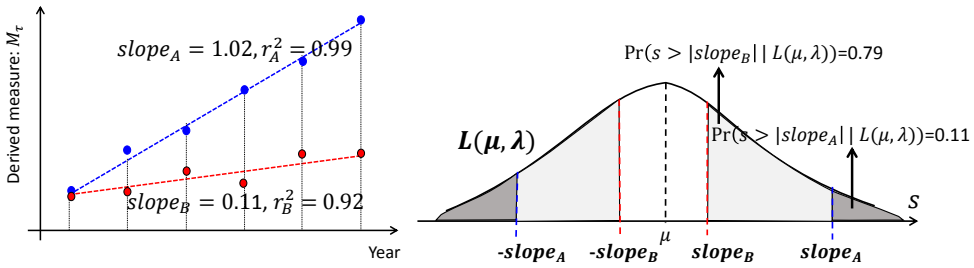
[5]We omit the minimum value discussion, as it is similar with the maximum case.

Thus, Φ_A is more significant than Φ_B.

Measuring Sig of Shape Insight. Let $\mathcal{X} = \langle x_1, x_2, ..., x_n \rangle$ be the time series of values in the result Φ. It is common that the trend is neither rising nor falling. Therefore, we set the null hypothesis as:

$$H_0 : \mathcal{X} \text{ forms a shape with slope } \approx 0$$

In business intelligence applications, data analysts are attracted to a clear rising/falling trend, whose slope is very different from 0. Thus, the p-value should measure how surprisingly the slope differs from 0.



(a) Slope of trends (b) p-value

Figure 5.6. The significance of shape insight

First, we fit $\mathcal{X}$ to a line by linear regression analysis (see Figure 5.6(a)), and then compute its slope *slope* and the goodness-of-fit[6] value r^2. According to [13], the distribution of slopes should follow a logistic distribution $L(\mu, \lambda)$, where μ, λ are constant parameters. In Figure 5.6(b), the p-value is the probability of the slope values equal to or larger than the observed slope of the rising trend[7]. Specifically, we compute the p-value as $p = Pr(s > |slope| \mid L(\mu, \lambda))$. Finally, we define the significance as $\mathsf{Sig}_\mathsf{T}(\Phi) = r^2 \cdot (1 - p)$, where the goodness-of-fit value

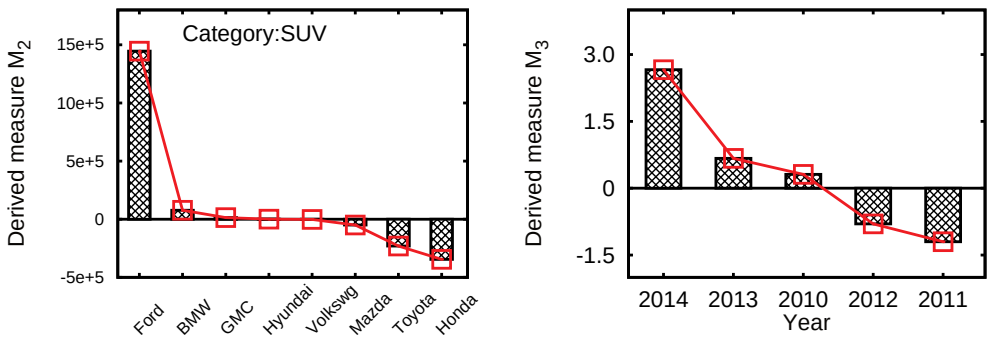
[6] https://en.wikipedia.org/wiki/Coefficient_of_determination

[7] We omit the falling trend discussion, as it is similar with rising trend.

r^2 is used as a weight.

We illustrate an example in Figure 5.6. Consider the shapes of blue dots and red dots in Figure 5.6(a). After fitting blue dots to a line, we obtain the slope $slope_A = 1.02$ and the goodness-of-fit value $r_A^2 = 0.99$. Similarly, after fitting red dots, we obtain $slope_B = 0.11$ and $r_B^2 = 0.92$. As illustrated in Figure 5.6(b), we then compute: $p_A = 0.11$ and $p_B = 0.79$. In this example, since $slope_A > slope_B$, the significance of A (i.e., $0.88 = 0.99 * (1 - 0.11)$) is larger than that of B (i.e., 0.19).

Examples in real dataset. Figure 5.7 shows the significance value of the insight "outstanding No.1" for two different pairs of $(\mathsf{SG}(S, D_i), \mathcal{C}_e)$. Figure 5.7(a) shows that the highest bar is remarkably higher than other bars, thus the significance value is high (0.96). In Figure 5.7(b), since the highest bar is not much higher than other bars, the significance is relatively low (0.36).

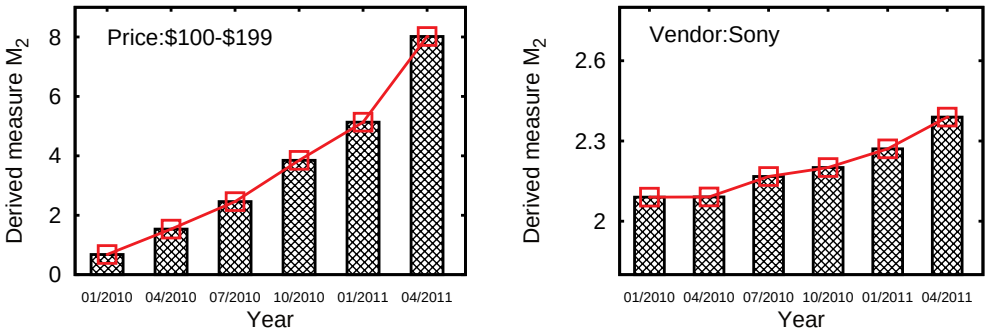


(a) Sig = 0.97 (b) Sig = 0.36

Figure 5.7. The significance of outstanding No.1

As another example, we then illustrate the significance of shape insights. In Figure 5.8(a), the derived measure increases quickly from 01/2010 to 04/2011 (i.e., high slope), thus its significance is high (0.99). However, in Figure 5.8(b),

the derived measure rises slowly, so its significance is low (0.31).



(a) Sig = 0.99 (b) Sig = 0.31

Figure 5.8. The significance of rising trend

5.4 System Architecture

We first describe the architecture of our top-k insight extraction system in Section 5.4.1, and then discuss the extensibility of our system in Section 5.4.2. Finally, we highlight the in-memory techniques in this work in Section 5.4.3.

5.4.1 Architecture Overview

Figure 5.9 depicts the architecture of our system, which consists of three layers.

1. The *system configuration* layer (at the bottom) allows user to configure system settings, e.g., specify a new insight type, or customize the null hypothesis based on the user's belief. We will elaborate this layer in Section 5.4.2.

2. The *insight extraction* layer (in the middle) is the core component of our system. First, it enumerates every possible pair $(\mathsf{SG}(S, D_i), \mathcal{C}_e)$ of sibling group and composite extractor. Then, it feeds each pair $(\mathsf{SG}(S, D_i), \mathcal{C}_e)$ into the computation engine and then invokes the insight engine to compute the score. During this process, the layer maintains the top-k insights list.

3. The *user interface* layer (at the top) is front-end of our system. It presents and visualizes the top-k insights to the users.

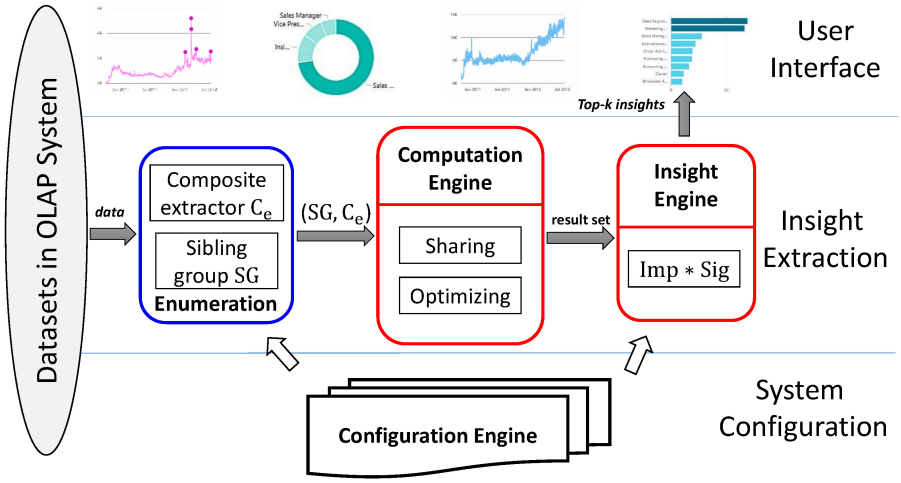


Figure 5.9. Top-k insight extraction system architecture

5.4.2 Extensibility

We have suggested some type(s) of the aggregate function, extractors, dataset, insights, and their score functions so far. Nevertheless, our system is extensible as follows:

Aggregate function and extractors: A composite extractor must take an aggregate function as the level-1 extractor, and then take any other extractor

at higher levels. First, we support many typical aggregate functions in OLAP, e.g., SUM, COUNT, AVERAGE, MAX, MIN. For example, we will consider the aggregate function COUNT, in the user study in Section 5.8. Second, we also allow the data analyst to define his own extractor (i.e., difference from Rank-1) . Regarding the validity of composing extractors, we only need to slightly revise the composite extractor adjunct taxonomy (cf. Table 5.4) to ensure the validation of generated composite extractors.

Dataset: Since our system is built on top of an OLAP system, it can deal any kind of dataset in the OLAP system. For a dataset with multiple measure attributes, user can either choose one measure attribute, or specify a derived measure as a weighted sum of other measure attributes [47] in system configuration layer.

Customization of insights: Our system supports other insight types, e.g., the correlation between two trends, and the seasonality of a trend [4]. Our score functions, e.g., significance functions, are also customizable. Recall in Section 5.3.2 that the significance of an insight type is defined based on p-value, which essentially measures how extreme an event is against a "common observation" in real world. Data analysts may customize their "common observations" by their domain knowledge.

Customization of the search space: Expert users may have some idea about what they are looking for. As such, we enable expert users to declare constraints and limit the search space. For example, an expert user may only consider sibling groups related to BMW in the car sales dataset.

5.4.3 In-memory techniques

With the results and findings in Chapter 3 and 4, we proposed the in-memory insight extraction system as Figure 5.9. Specifically, we illustrate efficient in-memory data cube techniques in Section 5.6.3. We also improve cache locality by sharing computation within / across sibling groups in Section 5.7.

5.5 Insight Extraction

We present a computation framework for *insight extraction layer*.

5.5.1 Computation Framework

Algorithm 5.7 is the pseudo-code of our computation framework for *insight extraction layer* in the system architecture. It employs a heap $\mathcal{H}$ to keep the top-k insights found so far. The algorithm needs to generate all possible instances of composite extractor $\mathcal{C}_e$ and sibling group $\mathsf{SG}(S, D_i)$. Then, it computes the insight from every $(\mathsf{SG}(S, D_i), \mathcal{C}_e, \mathsf{T})$, and updates $\mathcal{H}$ upon finding a better insight.

Generally, the number of sibling groups is much larger than the number of composite extractors. To keep the memory consumption manageable, we adopt the divide-and-conquer approach to generate sibling groups. Specifically, we implement this by a recursive function (Lines 9–18), which consists of two phases.

In Phase I (Lines 9–14), we first check whether the pair $(\mathsf{SG}(S, D_i), \mathcal{C}_e)$ is valid. If yes, then we compute the result Φ of the pair by Algorithm 5.8 which

Algorithm 5.7 Insights (dataset $\mathcal{R}(\mathcal{D}, \mathcal{M})$, depth τ, result size k)

1: initialize a min-heap $\mathcal{H} \leftarrow \emptyset$ ▷ store top-k insights
2: let ub_k be the k-th largest score in $\mathcal{H}$
3: $\mathcal{O} \leftarrow$ enumerate all possible $\mathcal{C}_e$ with depth τ ▷ enumerate $\mathcal{C}_e$
4: **for each** $\mathcal{C}_e$ in $\mathcal{O}$ **do**
5: **for** $i := 1$ to d **do** ▷ enumerate SG
6: initialize subspace $S \leftarrow \langle *, *, \cdots, * \rangle$
7: EnumerateInsight($S, D_i, \mathcal{C}_e$)
8: return $\mathcal{H}$

Function: EnumerateInsight (S, D_i, $\mathcal{C}_e$):
9: **if** isValid ($\mathsf{SG}(S, D_i), \mathcal{C}_e$) **then** ▷ Phase I
10: $\Phi \leftarrow \mathsf{Extract}\Phi(\ \mathsf{SG}(S, D_i), \mathcal{C}_e\ )$ ▷ Alg. 5.8: computation engine
11: **for** each T of insight **do**
12: $\mathbb{S} \leftarrow \mathsf{Imp}(\mathsf{SG}(S, D_i)) \cdot \mathsf{Sig}_{\mathsf{T}}(\Phi)$ ▷ Sec. 5.3: insight engine
13: **if** $\mathbb{S} > ub_k$ **then**
14: update $\mathcal{H}, ub_k$ by $(\mathsf{SG}(S, D_i), \mathcal{C}_e, \mathsf{T})$
15: **for** each value $v \in dom(D_i)$ **do** ▷ Phase II
16: $S^{'} \leftarrow S$, $S^{'}[D_i] \leftarrow v$
17: **for** each j with $S'[D_j] = *$ **do** ▷ enumerate SG
18: EnumerateInsight($S^{'}$, D_j, $\mathcal{C}_e$)

will be elaborated in Section 5.5.2. Next, we compute the score for each insight type and update $\mathcal{H}$ upon finding a better insight.

In Phase II (Lines 15–18), we create a child subspace S' from S by instantiating its value on dimension D_i. For each S', we pick a dimension D_j where $S'[D_j] = *$, and then invoke a recursive call on the sibling group $\mathsf{SG}(S', D_j)$

Example: Given the dataset in Table 5.1, we illustrate the obtained insights in Table 5.6. For ease of illustration, we only consider point insights and a fixed composite extractor $\mathcal{C}_e = \langle(\mathsf{SUM},\mathrm{Sales}),(\Delta_{prev},\mathrm{Year})\rangle$. In Table 5.6, each row shows a sibling group $\mathsf{SG}(S, D_i)$ and its insight score $\mathbb{S}$, i.e., the product of impact Imp and significance Sig. Due to the page limit, we do not show the steps for computing Imp and Sig. When $k = 1$, the top-1 insight corresponds to the first row in Table 5.6.

$\mathsf{SG}(S, D_i)$	$\mathbb{S} = \mathsf{Imp} \cdot \mathsf{Sig}$	Point insight: outstanding No.1
SG(⟨*,*⟩, Year)	$0.61 = 1.00 \cdot 0.61$	⟨2014⟩
SG(⟨*,Honda⟩, Year)	$0.16 = 0.38 \cdot 0.42$	⟨Honda, 2014⟩
SG(⟨*,Toyota⟩, Year)	$0.14 = 0.30 \cdot 0.45$	⟨Toyota,2014⟩
SG(⟨*,Ford⟩, Year)	$0.02 = 0.15 \cdot 0.10$	⟨Ford,2013⟩
SG(⟨*,BMW⟩, Year)	$0.01 = 0.17 \cdot 0.03$	⟨BMW,2012⟩
SG(⟨2014,*⟩, Brand)	$0.07 = 0.25 \cdot 0.27$	⟨2014,Honda⟩
SG(⟨2012,*⟩, Brand)	$0.03 = 0.18 \cdot 0.14$	⟨2012,Honda⟩
SG(⟨2013,*⟩, Brand)	$0.02 = 0.20 \cdot 0.12$	⟨2013,Honda⟩
SG(⟨2011,*⟩, Brand)	$0.01 = 0.17 \cdot 0.04$	⟨2011,Ford⟩

Table 5.6. Insight candidates for $\mathcal{C}_e = \langle(\mathsf{SUM},\mathrm{Sales}),(\Delta_{prev},\mathrm{Year})\rangle$

5.5.2 Computation Engine

We introduce Algorithm 5.8 to apply a composite extractor $\mathcal{C}_e$ on a sibling group $\mathsf{SG}(S, D_i)$ and then compute a corresponding result set Φ. It enumerates each subspace $S' \in \mathsf{SG}(S, D_i)$ (Line 3) and computes the derived measure of S' with respect to $\mathcal{C}_e$ (Line 4). Finally, it inserts each S' with its derived measure into Φ and returns it to the caller.

Algorithm 5.8 ExtractΦ($\mathsf{SG}(S, D_i), \mathcal{C}_e$)

```
1: initialize a result set Φ ← ∅
2: for each value v in dom(D_i) do
3:     S' ← S, S'[D_i] ← v
4:     M' ← RecurExtract( S', τ, C_e )
5:     insert (S', M') into Φ
6: return Φ
```

Function: RecurExtract(Subspace S', $level$, $\mathcal{C}_e$):

```
7: if level > 1 then
8:     initialize a result set Φ_level
9:     D_ξ ← C_e[level].D_x
10:    for each value v in D_ξ do
11:        S_v ← S', S_v[D_ξ] ← v
12:        M'_v ← RecurExtract( S_v, level − 1, C_e )
13:        insert (S_v, M'_v) into Φ_level
14:    M' ← derived S'.M' by applying C_e[level] on Φ_level        ▷ Def. 5.3
15: else
16:    M' ← SUM( S' )                                  ▷ SUM(S'): data cube
17: return M'
```

Conceptually, the computation of the derived measure (at Line 4) involves building trees in the top-down manner and running extractors on tree nodes in the bottom-up manner. This can be implemented by a recursive function 'RecurExtract'.

In this function, the parameter *level* indicates the current level of the extractor in $\mathcal{C}_e$. The initial *level* is τ, which corresponds to the highest level. Let D_ξ be the dimension used by the current extractor $\mathcal{C}_e[level]$ (Line 9). We examine each child subspace S_v (of S'), apply the $level-1$ extractor to it recursively, and insert S_v with its derived measure into a temporary result set Φ_{level} (Lines 10–13). Finally, we apply the current extractor on Φ_{level} to compute the derived measure at the current level.

When we reach the bottom level (i.e., $level = 1$), it suffices to compute the SUM of measures in the subspace S'. This can be obtained efficiently from a data cube.

Example: Consider the composite extractor $\mathcal{C}_e = \langle$(SUM,Sales),(Δ_{prev},Year)$\rangle$ with the sibling group SG($\langle$2013,*$\rangle$,Brand) in Table 5.6. We illustrate the recursive computation of derived measures in Figure 5.10. Each tree node represents a recursive call of 'RecurExtract', and it is associated with a subspace S_v and a derived measure M'_v. In the first phase, we build a tree in the top-down manner. The second phase begins when we reach the bottom level (i.e., $level = 1$). Next, we examine these tree nodes in the bottom-up manner and apply the corresponding extractor on each tree node to compute its derived measure. Then, we obtain the result set Φ={($\langle$2013,Ford$\rangle$,9), ($\langle$2013,BMW$\rangle$,-3), ($\langle$2013,Honda$\rangle$,7), ($\langle$2013,Toyota$\rangle$,-5)}. Finally, we compute the Sig value of Φ (cf. Section 5.3.2)

and the Imp value in order to obtain the insight score (i.e., 0.02).

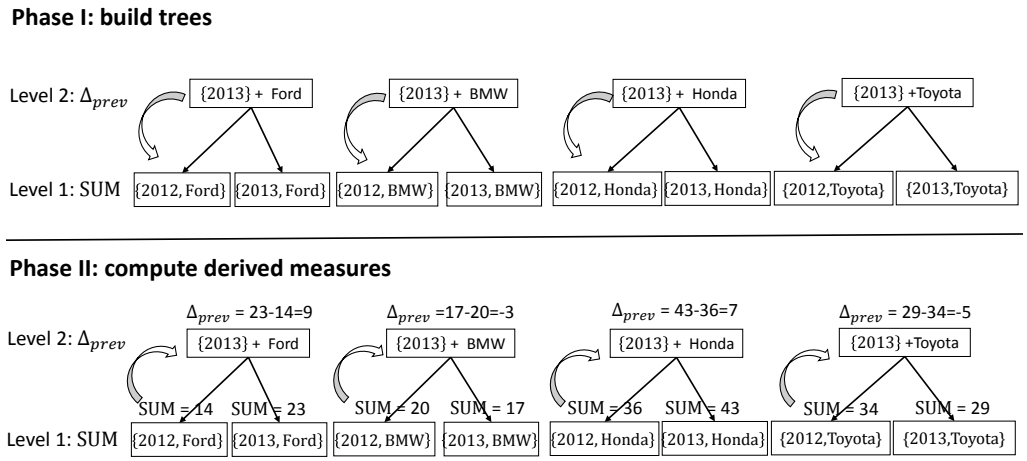


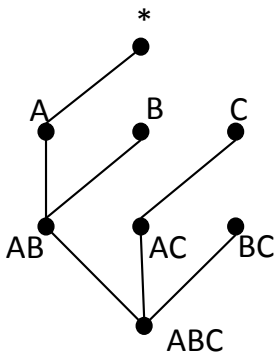
Figure 5.10. Running a composite extractor on a sibling group

$\mathcal{C}_e = \langle(\mathsf{SUM},\text{Sales}),(\Delta_{prev},\text{Year})\rangle$, $\mathsf{SG}(\langle 2013,*\rangle,\text{Brand})$

Data cube optimization: Our framework performs aggregation frequently, e.g., $\mathsf{SUM}(S)$ (Line 16 in Alg. 5.8), and $\mathsf{Imp}(\mathsf{SG}(S, D_i)) = \mathsf{SUM}_{S' \in \mathsf{SG}(S,D_i)}(S')/\mathsf{SUM}(\langle *, *, \cdots, * \rangle)$ (Line 12 in Alg. 5.7). We can construct a data cube and utilize it to reduce the aggregation cost.

A data cube [43, 16] is a collection of *cuboids*, where each cuboid stores the group-by aggregate result for a particular set of dimensions. Figure 5.11(a) illustrates a data cube built for a dataset with schema $(A, B, C, \mathcal{M})$. It contains eight cuboids. The content of cuboid $\langle A, B \rangle$ is shown in Figure 5.11(b).

To compute $\mathsf{SUM}(S)$ efficiently, we propose to store each cuboid as a hash table. Given a subspace S, we can lookup the corresponding entry in the cuboid and then retrieve $\mathsf{SUM}(S)$ in $O(1)$ time.



(a) a data cube

key: A, B	value: $\mathcal{M}$
a_1, b_1	2
a_1, b_2	3
$\cdots$	$\cdots$
a_2, b_1	7
a_2, b_2	9
$\cdots$	$\cdots$
a_3, b_2	8
a_3, b_3	1

(b) cuboid $\langle A, B \rangle$

Figure 5.11. Example of a data cube

5.5.3 Time Complexity Analysis

Since both Algorithms 5.7 and 5.8 spend most of the time on recursive calls, we focus on analyzing the number of recursive calls in these algorithms. We follow the notations in Section 5.2.3. In our analysis, $\mathfrak{D} = \max_{i=1}^{d} |dom(D_i)|$ denotes the maximum domain size, and β denotes the number of types of extractors.

For Algorithm 5.7: It invokes the recursive function 'EnumerateInsight' for all possible insights of $(\mathsf{SG}(S, D_i), \mathcal{C}_e, \mathsf{T})$. Each recursive call examines $|\mathsf{T}|$ types of insights. Combining this with the results in Section 5.2.3, the number of recursive calls to 'EnumerateInsight' is $O(|\mathsf{T}| \cdot d \cdot (\beta \cdot d)^{\tau-1} \cdot (\mathfrak{D}+1)^d)$.

For Algorithm 5.8: It examines each value of attribute D_i and thus calls the recursive function 'RecurExtract' for $\mathfrak{D}$ times at most.

Let j be the current level in the function 'RecurExtract'. When $j > 1$, the function 'RecurExtract' examines each value of attribute $\mathcal{C}_e[i].D_x$ and thus calls the function 'RecurExtract' for $\mathfrak{D}$ times at most.

In summary, the number of recursive calls to 'RecurExtract' is $O(\mathfrak{D} \cdot \prod_{j=2}^{\tau} dom_{max}) =$

$O(\mathfrak{D}^\tau)$.

5.6 Optimization techniques

In this section, we propose optimization techniques to reduce the running time of our solution.

5.6.1 Pruning by Upper Bound Score

Consider the computation of insight score at Lines 10–12 in Algorithm 5.7. The term $\mathsf{Imp}(\mathsf{SG}(S, D_i))$ can be computed efficiently (cf. Section 5.5.2). However, it is expensive to compute Φ as it invokes Algorithm 5.8.

To reduce the cost, we propose an upper bound score

$$\mathbb{S}^{UB}(\mathsf{SG}(S, D_i), \mathcal{C}_e, \mathsf{T}) = \mathsf{Imp}(S) \tag{5.2}$$

and show that it serves as an upper bound of the insight score (cf. Lemma 5.2).

Lemma 5.2 (Upper bound property)

$$\mathbb{S}^{UB}(\mathsf{SG}(S, D_i), \mathcal{C}_e, \mathsf{T}) \geq \mathbb{S}(\mathsf{SG}(S, D_i), \mathcal{C}_e, \mathsf{T})$$

Proof. By Def. 5.2, we have $\mathsf{Imp}(\mathsf{SG}(S, D_i)) = \mathsf{Imp}(S)$ with $S[i] = *$ (Line 17, Alg 5.7). Hence, $\mathbb{S}(\mathsf{SG}(S, D_i), \mathcal{C}_e, \mathsf{T}) = \mathsf{Imp}(S) \cdot \mathsf{Sig}_{\mathsf{T}}(\Phi)$. Since $\mathsf{Sig}_{\mathsf{T}}(\Phi) \leq 1$, we have: $\mathbb{S}(\mathsf{SG}(S, D_i), \mathcal{C}_e, \mathsf{T}) \leq \mathsf{Imp}(S) = \mathbb{S}^{UB}(\mathsf{SG}(S, D_i), \mathcal{C}_e, \mathsf{T})$. □

With this lemma, we can implement the following pruning rule before Line 10 in Algorithm 5.7. We compute $\mathbb{S}^{UB}(\mathsf{SG}(S, D_i), \mathcal{C}_e, \mathsf{T})$ and then compare it with ub_k (i.e., k-th insight score found so far). If $ub_k > \mathbb{S}^{UB}(\mathsf{SG}(S, D_i), \mathcal{C}_e, \mathsf{T})$, then we skip the execution of Lines 10–14.

5.6.2 Subspace Ordering

The effectiveness of the above pruning rule (cf. Section 5.6.1) depends on ub_k (i.e., k-th insight score found so far). To enable effective pruning, it is desirable to obtain a high ub_k as early as possible. Therefore, we propose techniques to reorder both outer and inner loops (Lines 15–18) in Algorithm 5.7.

Ordering of outer-loop (Lines 15–16): Observe that the upper bound score $\mathbb{S}^{UB}(\mathsf{SG}(S', D_i), \mathcal{C}_e, \mathsf{T}) = \mathsf{Imp}(S')$ depends on S' only. Thus, we propose to compute $\mathbb{S}^{UB}$ for each subspace S' at Line 16, and then examine those subspaces in descending order of $\mathbb{S}^{UB}$.

Ordering of inner-loop (Lines 17–18): An intuitive strategy is to order dimensions in ascending order of the domain size $|dom(D_j)|$. When $|dom(D_j)|$ is small, few subspaces will be generated and the average impact of each subspace is expected to be high. This would increase the possibility to obtain a high ub_k early.

5.6.3 Sibling Cube

Our framework incurs significant overhead on (i) hash table lookup operation per computing $\mathsf{SUM}(S)$ (cf. Section 5.5.2), and (ii) sorting operation in

implementing subspace ordering (cf. Section 5.6.2).

In this section, we propose an *sibling cube* in order to reduce the number of lookup operations in hash tables. Furthermore, our sibling cube can avoid redundant sorting operations in our framework.

5.6.3.1 Sibling cube structure

Our *sibling cube* is designed in a fashion that suits better with the operations used in our framework. Specifically, our sibling cube is a collection of the following cuboids:

Definition 5.7 (Cuboid in sibling cube) *A cuboid is labeled by* $\langle \mathcal{D}' \rangle \circ \underline{D_i}$, *where* $\mathcal{D}' \subset \mathcal{D}$ *is a subset of dimensions and* D_i *is a dimension not in* $\mathcal{D}'$.

The cuboid contains a cell for a subspace S *if* $\forall j \in \mathcal{D}', S[j] \neq *$ *and* $\forall j \notin \mathcal{D}', S[j] = *$.

The cell for subspace S *is an array of pairs* $\langle (v_x, \mathcal{M}_x) : v_x \in dom(D_i) \rangle$ *sorted in the descending order of* $\mathcal{M}_x$. *We require that* $\mathcal{M}_x = \mathsf{SUM}(S')$, *where* S' *is a child subspace of* S *with its dimension* D_i *set to* v_x.

Following the example in Section 5.5.2, we consider a dataset with schema $(A, B, C, \mathcal{M})$. We compare a data cube with our sibling cube in Figure 5.12. A cuboid in a data cube contains many cells (see Figure 5.12(a)). On the other hand, a cuboid in an sibling cube contains fewer cells but each cell stores more information (see Figure 5.12(b)).

Compared to the data cube, the sibling cube occupies at most d times the space in the worst case. Nevertheless, the iceberg cube technique [16] can be adapted to shrink our cube size significantly. Specifically, we only store entries whose measures are above $minsup\%$ (e.g., 0.1%) of measure in the dataset. Our experimental study shows that our sibling cube is small enough to fit in main memory.

In the following discussion, we demonstrate the advantages of using the sibling cube over the data cube.

5.6.3.2 Reducing hash table lookup operations

Our algorithms in Section 5.5 execute this operation: "Given a sibling group $\mathsf{SG}(S, D_i)$, retrieve $\mathsf{SUM}(S')$ for each subspace $S' \in \mathsf{SG}(S, D_i)$."

For example, we take $\mathsf{SG}(\langle a_1, * \rangle, B)$ as the sibling group and assume $dom(B) = \{b_1, b_2, b_3\}$. When using a traditional data cube, we issue three lookup operations $\langle a_1, b_1 \rangle, \langle a_1, b_2 \rangle, \langle a_1, b_3 \rangle$ to the cuboid in Figure 5.12(a).

With our sibling cube, it suffices to issue one lookup operation (a_1) to the cuboid in Figure 5.12(b). Then, we can retrieve the list of entries for $\langle a_1, b_3 \rangle, \langle a_1, b_2 \rangle, \langle a_1, b_1 \rangle$ and process the list sequentially.

key: A, B	value: $\mathcal{M}$
a_1, b_1	2
a_1, b_2	3
...	...
a_2, b_1	7
a_2, b_2	9
...	...
a_3, b_2	8
a_3, b_3	1

(a) data cube: cuboid $\langle A, B \rangle$

key: A	value: $B, \mathcal{M}$
a_1	$b_3, 6 \mid b_2, 3 \mid b_1, 2$
a_2	$b_2, 9 \mid b_1, 7 \mid b_3, 5$
a_3	$b_2, 8 \mid b_1, 4 \mid b_3, 1$

(b) sibling cube: cuboid $\langle A \rangle \circ \underline{B}$

Figure 5.12. Data cube vs. sibling cube

In addition to reducing lookup operations, the sibling cube improves the data access locality (e.g., converting random accesses to sequential accesses) and benefits the performance of CPU cache [14].

5.6.3.3 Avoiding sorting operations in loop ordering

When we implement the outer loop ordering (see Section 5.6.2) at Lines 15–16 in Algorithm 5.7, we need to sort subspaces $S' \in \mathsf{SG}(S, D_i)$ in descending order of their upper bound scores (which can be derived from SUM values).

With our sibling cube, we can retrieve a sorted list directly and avoid sorting operations on-the-fly.

We extend the computation framework (Algorithm 5.7) with the above optimization techniques, and then present the optimized computation framework in Algorithm 5.9.

Algorithm 5.9 Insights+Optimized (dataset $\mathcal{R}(\mathcal{D}, \mathcal{M})$, depth τ, result size k)

1: *run Lines 1–3 in Alg. 5.7*
2: construct an sibling cube SibCube ▷ Sec. 5.6.3
3: sort $D_i \in \mathcal{D}$ by ascending domain size ▷ Sec. 5.6.2
4: *run Lines 4–8 in Alg. 5.7* ▷ call the function below

Function: EnumerateInsightI (S, D_i, $\mathcal{C}_e$):
5: **if** isValid($\mathsf{SG}(S, D_i), \mathcal{C}_e$) **then**
6: **if** $ub_k \leq \mathbb{S}^{UB}(\mathsf{SG}(S, D_i), \mathcal{C}_e, \mathsf{T})$ **then** ▷ Sec. 5.6.1
7: $\Phi \leftarrow$ use SibCube in Extract$\Phi(\mathsf{SG}(S, D_i), \mathcal{C}_e)$ ▷ Alg. 5.8
8: *run Lines 11–14 in Alg. 5.7*
9: sorted list $L \leftarrow$ SibCube$[S \circ D_i]$ ▷ Sec. 5.6.3
10: **for** each value-measure pair $(v, \mathcal{M}) \in L$ **do** ▷ Sec. 5.6.2
11: $S' \leftarrow S$, $S'[D_i] \leftarrow v$
12: $S'.\mathsf{SUM} \leftarrow \mathcal{M}$ ▷ stored value of $\mathsf{SUM}(S')$
13: *run Lines 17–18 in Alg. 5.7* ▷ Line 18: EnumerateInsightI

Algorithm 5.9 is an optimized version of Algorithm 5.7 (cf. Section 5.5.1) that incorporates all optimization techniques in the above subsections. We employ a pruning technique (cf. Section 5.6.1) at Line 6. We apply loop ordering techniques (cf. Section 5.6.2) at Lines 3 and 10. Also, we construct an sibling cube SibCube (cf. Section 5.6.3) at Line 2, and then use it at Lines 7 and 9–10. At Line 12, we store the value of $\mathcal{M}$, obtained at Line 10, in S'.SUM. When the recursive call requires SUM(S'), it can access S'.SUM immediately.

5.7 Computation Sharing

This section presents computation sharing techniques to further accelerate our solution.

5.7.1 Sharing within a Sibling Group

First, we identify sharing opportunities within a sibling group in an example. Then, we devise the condition for sharing.

As an example, suppose that we apply the composite extractor $\mathcal{C}_e = \langle$(SUM,Sales),(%,Year)$\rangle$ on the sibling group SG($\langle$*,BMW$\rangle$,Year). Figure 5.13(a) illustrates the computation process of Algorithm 5.8 on this example. Observe that these trees have the same content at level 1, as highlighted by red rectangles. In order to reduce computation cost, we propose to identify the shared content and compute it only once, as shown in Figure 5.13(b).

We discover that significant computation can be saved when certain condition is satisfied. Specifically, we prove in Lemma 5.3 that, if a sibling group

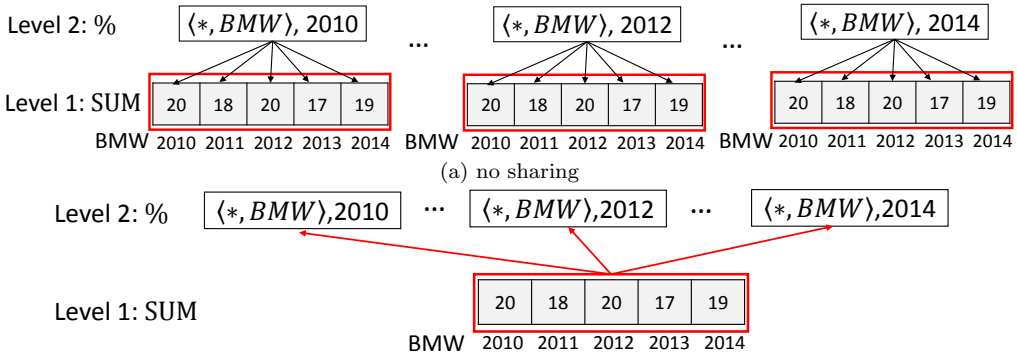


Figure 5.13. Running a composite extractor on a sibling group

$\mathcal{C}_e$=⟨(SUM,Sales),(%,Year)⟩,SG(⟨∗,BMW⟩,Year)

$\mathsf{SG}(S, D_i)$ and the last extractor of $\mathcal{C}_e$ have the same dimension (i.e., $\mathcal{C}_e[\tau].D_x = D_i$), then we can share the intermediate result at level $\tau - 1$.

Lemma 5.3 (Sharing within a sibling group) *Given a composite extractor $\mathcal{C}_e$ and a sibling group $\mathsf{SG}(S, D_i)$, if $\mathcal{C}_e[\tau].D_x = D_i$, then all subspaces of $\mathsf{SG}(S, D_i)$ share the same intermediate result at level $\tau - 1$.*

Proof. Let $S' \in \mathsf{SG}(S, D_i)$ be a subspace. Since S' and S differ on dimension D_i only, we have: $\mathsf{SG}(S', D_i) = \mathsf{SG}(S, D_i)$ ——(★).

According to Definition 5.5, we derive $S'.\mathcal{M}_\tau$ from the set $\Phi' = \{(S_c, S_c.\mathcal{M}_{\tau-1}) : S_c \in \mathsf{SG}(S', \mathcal{C}_e[\tau].D_x)\}$.

By combining (★) with the given condition $\mathcal{C}_e[\tau].D_x = D_i$, we derive: $\mathsf{SG}(S', \mathcal{C}_e[\tau].D_x) = \mathsf{SG}(S, D_i)$. Therefore, Φ' is independent of S' and it can be used to derive $S''.\mathcal{M}_\tau$ for any $S'' \in \mathsf{SG}(S, D_i)$. □

We enhance the computation engine (Algorithm 5.8) with the above sharing idea, and then obtain the optimized version in Algorithm 5.10.

Algorithm 5.10 ExtractΦII($\mathsf{SG}(S, D_i), \mathcal{C}_e$)

1: **if** $D_i \neq \mathcal{C}_e[\tau].D_x$ **then** ▷ test for Lem. 5.3
2: $\Phi \leftarrow \mathsf{ExtractΦ}(\mathsf{SG}(S, D_i), \mathcal{C}_e)$ ▷ Alg. 5.8
3: **else**
4: initialize a result set $\Phi \leftarrow \emptyset$
5: $S' \leftarrow S$, $S'[D_i] \leftarrow dom(D_i).first$
6: $(M', \Phi_\tau) \leftarrow \mathsf{RecurExtractII}(\ S', \tau, \mathcal{C}_e\ )$
7: **for each** value $v \in dom(D_i)$ **do**
8: $S' \leftarrow S$, $S'[D_i] \leftarrow v$
9: $M' \leftarrow$ derived $S'.M'$ after applying $\mathcal{C}_e[\tau]$ on Φ_τ
10: insert (S', M') into Φ
11: **return** Φ

Function: RecurExtractII(Subspace S', $level$, $\mathcal{C}_e$):
12: initialize a result set Φ_{level}
13: $D_\xi \leftarrow \mathcal{C}_e[level].D_x$
14: **if** $level > 2$ **then**
15: **for each** value v in D_ξ **do**
16: $S_v \leftarrow S'$, $S_v[D_\xi] \leftarrow v$
17: $(M'_v, \Phi_{temp}) \leftarrow \mathsf{RecurExtractII}(\ S_v, level - 1, \mathcal{C}_e\ )$
18: insert (S_v, M'_v) into Φ_{level}
19: **else**
20: $\Phi_{level} \leftarrow$ SibCube$[S' \circ D_\xi]$ ▷ Sec. 5.6.3
21: $M' \leftarrow$ derived $S'.M'$ by applying $\mathcal{C}_e[level]$ on Φ_{level} ▷ Def. 5.3
22: **return** (M', Φ_{level})

Algorithm 5.10 applies Lemma 5.3 (cf. Section 5.7.1) to accelerate the computation of $\mathcal{C}_e$ on $\mathsf{SG}(S, D_i)$. First, we check the condition in Lemma 5.3. If it is not satisfied, then we revert back to calling Algorithm 5.8. Otherwise, we invoke the function 'RecurExtractII' to compute an intermediate result set Φ_τ (Line 6) and then reuse it to obtain the derived measure for each subspace in $\mathsf{SG}(S, D_i)$.

Note that 'RecurExtractII' returns a pair (M', Φ_{level}), where Φ_{level} is the intermediate result set for computing the derived measure M' of subspace S'. Actually we only need Φ_{level} at level τ and only M' at other levels.

5.7.2 Sharing across Sibling Groups

We proceed to investigate sharing opportunities across multiple sibling groups.

Consider our computation framework in Algorithm 5.7. After fixing the composite extractor $\mathcal{C}_e$ (at Line 4), we enumerate sibling groups and apply $\mathcal{C}_e$ on each of them. In this example, we assume $\mathcal{C}_e$=⟨(SUM,Sales),(%,Brand)⟩. Figure 5.14(a) illustrates the computation process when we apply $\mathcal{C}_e$ on multiple sibling groups: $\mathsf{SG}(\langle 2010, *\rangle$,Brand) $\cdots$ $\mathsf{SG}(\langle 2014, *\rangle$,Brand), then $\mathsf{SG}(\langle *$,Honda⟩,Year) $\cdots$ $\mathsf{SG}(\langle *$,Ford⟩,Year). Observe that, at level 2, the derived measures in green rectangles are the same as those in red rectangles. This happens because some subspace (⟨2010,Honda⟩:42%) appears in more than one sibling groups ($\mathsf{SG}(\langle 2010, *\rangle$,Brand) and $\mathsf{SG}(\langle *$,Honda⟩,Year)).

We illustrate how this method works with the example in Figure 5.14(b). We employ a temporary hash table Ψ to store the derived measure $S'.M'$ for subspace S' that we have processed before (in other sibling groups). Initially, Ψ

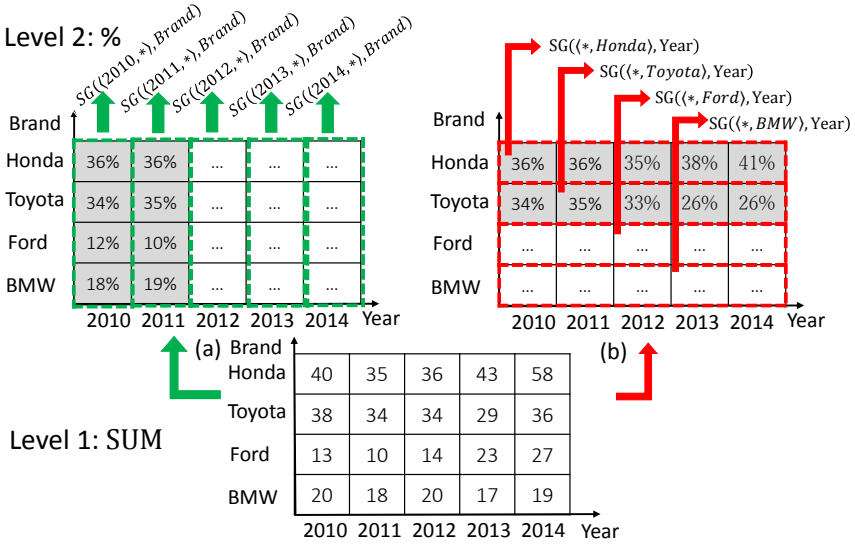
is empty. First, we examine SG($\langle 2010, * \rangle$, Brand), and process four subspaces $\langle$2010, Ford$\rangle$, $\langle$2010, BMW$\rangle$, $\langle$2010, Honda$\rangle$, $\langle$2010, Toyota$\rangle$. Since Ψ is empty, we need to compute the derived measures for the above subspaces and then insert them into Ψ. Similarly, we populate Ψ when we examine SG($\langle 2011, * \rangle$, Brand), $\cdots$, SG($\langle 2014, * \rangle$, Brand). Finally, when we examine SG($\langle *$,Honda$\rangle$,Year), we can find its subspaces in Ψ and thus retrieve their derived measures from Ψ directly.

We then discuss how to incorporate the above techniques into our algorithms. First, we apply the above technique and obtain an efficient computation engine in Algorithm 5.11).

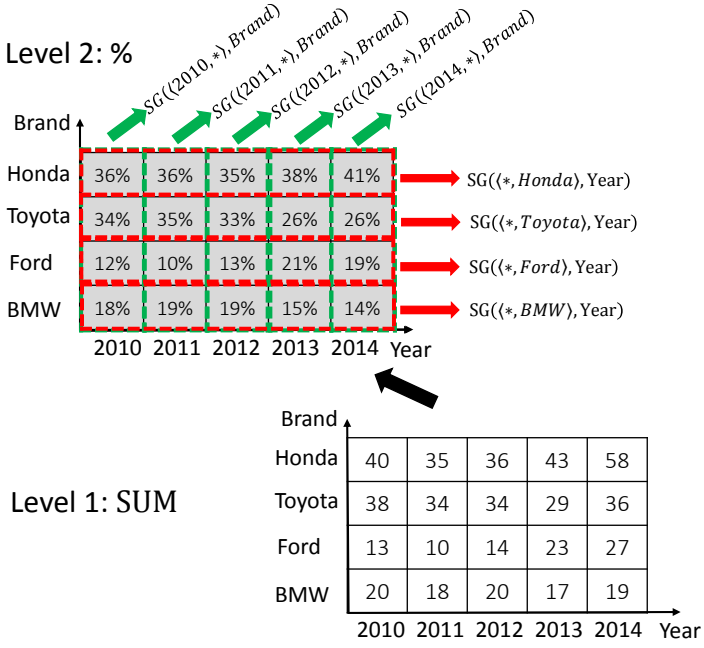
Algorithm 5.11 ExtractΦIII(SG(S, D_i),$\mathcal{C}_e$, hash table Ψ)

```
1: initialize a result set Φ ← ∅
2: for each value v in dom(D_i) do
3:     S' ← S, S'[D_i] ← v
4:     if Ψ contains S' then
5:         M' ← Ψ[S']                                  ▷ get from Ψ
6:     else
7:         M' ← RecurExtract( S', τ, C_e )             ▷ function in Alg. 5.8
8:         Ψ[S'] ← M'                                  ▷ store into Ψ
9:     insert (S', M') into Φ
10: return Φ
```

To save computation cost, we need to detect the shared content and reuse it, as depicted in Figure 5.14(b). We present Algorithm 5.11 to integrate Algorithm 5.8 with a sharing technique (cf. Section 5.7.2). It employs a hash table Ψ to store the derived measure $S'.M'$ for subspace S' that we have processed before (in other sibling groups). If Ψ contains S', then we can retrieve its derived measure from Ψ immediately. Otherwise, we need to compute the derived measure and then store it into Ψ.



(a) no sharing



(b) sharing across sibling groups

Figure 5.14. Running a composite extractor on multiple sibling groups

$\mathcal{C}_e$=⟨(SUM,Sales),(%,Brand)⟩

Second, by using all techniques in Sections 5.6,5.7, we have an efficient computation framework in Algorithm 5.12 for insight extraction layer.

Algorithm 5.12 Insights+Sharing+Optimized (dataset $\mathcal{R}(\mathcal{D}, \mathcal{M})$, depth τ , result size k)

```
1: create a hash table Ψ
2: run Lines 1–3 in Alg. 5.9
3: for each C_e ∈ O do
4:     clear Ψ
5:     for i := 1 to d do
6:         initialize subspace S ← ⟨∗, ∗, · · · , ∗⟩
7:         EnumerateInsightIII( S, D_i, C_e, Ψ )
8: return H
```

Function: EnumerateInsightIII (S, D_i, $\mathcal{C}_e$, Ψ):

```
9: if isValid ( SG(S, D_i), C_e ) then
10:     if ub_k ≤ S^UB(SG(S, D_i), C_e, T) then
11:         if D_i ≠ C_e[τ].D_x then
12:             ExtractΦIII( SG(S, D_i), C_e, Ψ )          ▷ Alg. 5.11
13:         else
14:             ExtractΦII( SG(S, D_i), C_e )              ▷ Alg. 5.10
15:         run Line 8 in Alg. 5.9
16: run Lines 9–13 in Alg. 5.9                  ▷ Line 13: EnumerateInsightIII
```

Algorithm 5.12 is an integrated version of Algorithm 5.9 that incorporates sharing computation techniques in Section 5.7. We employ sharing within a sibling group (cf. Section 5.7.1) at Line 14. We apply sharing across sibling groups (cf. Section 5.7.2) at Line 12. Hash table Ψ will be flushed for each composite extractor at Line 4.

5.8 Effectiveness Study

In this section, we evaluate the effectiveness of our top-k insight extraction system by (1) case study, (2) insight utility study, and (3) human effort study on real datasets.

5.8.1 Case Studies

We collect the following two real datasets (i.e, car sales and tablet sales), and then demonstrate the insights obtained from these datasets.

Car sales dataset[8]: The dataset contains 276 tuples. Each tuple (i.e., a car) has 4 dimensions and a measure *Sales*. The domain sizes of dimensions are: *Year* (5), *Brand* (8), *Category* (8) and *Model* (55).

Tablet sales dataset[9]: The dataset contains 20,685 tuples. Each tuple (i.e., a tablet) has 11 dimensions and a measure *Sales*. The domain sizes of dimensions are: *Year* (11), *CPU* (2), *OS* (7), *Connectivity* (5), *Price* (23), *Region* (9), *Country* (54), *Product* (2), *Resolution* (18), *Size* (9) and *Vendor* (157).

Table 5.7 shows the top-2 insights on car sales and tablet sales, respectively, at τ=2 and τ=3. For convenience, we have omitted $*$ in sibling groups in Table 5.7. For example, SG({SUV},Year) is equivalent to SG($\langle$*,*,SUV,*$\rangle$, Year). We then elaborate some of these insights from Figure 5.15 to 5.18.

Insights from car sales: We first compare our insight with a raw aggregation result on car sales. Figure 5.15(a) refers to the shape insight in Table 5.7(a).

[8] http://www.goodcarbadcar.net/p/sales-stats.html
[9] This is a private real dataset collected from the industry.

Insight	score	SG(S,D_i)	composite extractor $\mathcal{C}_e$
Top-1	0.31	SG({SUV},Brand)	⟨(SUM,Sales),(Δ_{avg},Category)⟩
Point	When measuring the importance of SUV sales for a certain Brand, Ford is Outstanding No.1.		
Top-2	0.30	SG({SUV},Year)	⟨(SUM,Sales),(%,Category)⟩
Shape	There is a rising trend of SUV's market share.		

(a) car sales, top-2 insights with $\tau = 2$

Top-1	0.32	SG({SUV},Year)	⟨(SUM,Sales),(%,Year),(Δ_{avg},Category)⟩
Point	In 2014, SUV exhibits most advantage over other categories than ever.		
Top-2	0.25	SG({Ford},Year)	⟨(SUM,Sales),(%,Brand),(Δ_{avg}, Year)⟩
Shape	There is a falling trend of Ford's market share.		

(b) car sales, top-2 insights with $\tau = 3$

Top-1	0.96	SG({*},Year)	⟨(SUM,Sales),(Δ_{prev},Year)⟩
Shape	The yearly increase of tablet sales is slowing down.		
Top-2	0.64	SG({WiFi},Year)	⟨(SUM,Sales),(Δ_{prev},Year)⟩
Shape	The yearly increase of sales of WIFI based tablets is slowing down.		

(c) tablet sales, top-2 insights with $\tau = 2$

Top-1	0.99	SG({ * },Year)	⟨(SUM,Sales),(Δ_{prev},Year),(Δ_{avg},Year)⟩
Point	2012/04-07's yearly increase of tablet sales is remarkably lower than ever.		
Top-2	0.96	SG({Tablet},Year)	⟨(SUM,Sales),(%,Year),(Rank,Year)⟩
Shape	There is a rising trend of Tablet (vs. eReader) sales.		

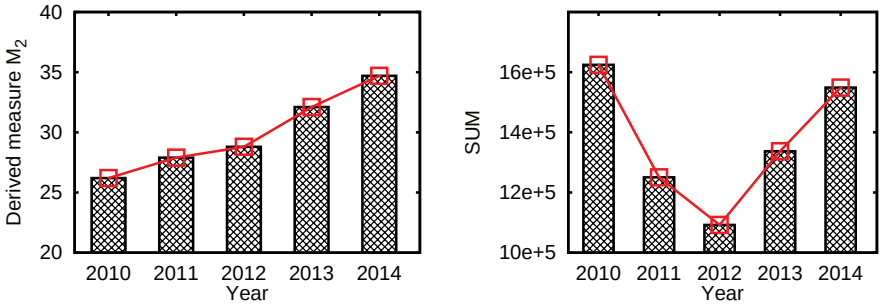
(d) tablet sales, top-2 insights with $\tau = 3$

Table 5.7. Case studies of insights on real datasets

Its SG(⟨*,*,SUV,*⟩, Year) means that we compare SUV cars by year. Its $\mathcal{C}_e$ = ⟨(SUM,Sales),(%, Category)⟩ means that we analyze the percentage of SUV's sales among all categories. Figure 5.15(a) shows that such a percentage rises with year. On the other hand, the raw aggregation result for the same SG(⟨*,*,SUV,*⟩, Year) does not reveal much information.

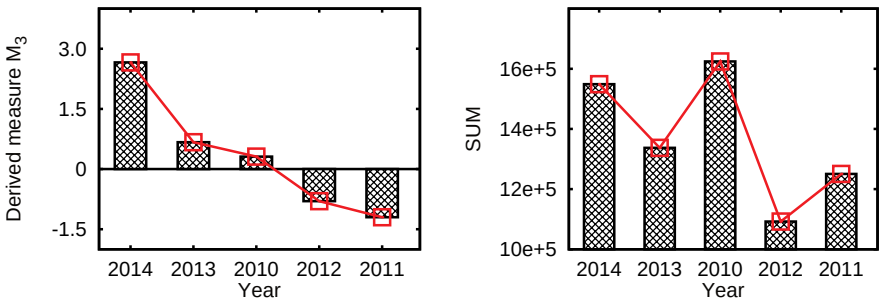
Figure 5.16(a) refers to the point insight (outstanding No.1) in Table 5.7(b). Its SG(⟨ *,*,SUV,* ⟩, Year) means we compare SUV cars by year. Its $\mathcal{C}_e$ = ⟨(SUM,Sales),(%,Year),(Δ_{avg}, Category)⟩ means we analyze SUV's yearly share over the average yearly share of all categories. Figure 5.16(a) shows that 2014, SUV exhibits most advantages over the other years. However, the raw aggregation result in Figure 5.16(b) does not reveal this information.

Insights from tablet sales: Then we compare our insight with a raw aggregation result on tablet sales. Figure 5.17(a) refers to the top-1 shape insight in



(a) insight: rising trend (b) raw aggregation result

Figure 5.15. Car sales shape insight: SG(⟨∗, ∗,**SUV,**∗⟩**, Year)**

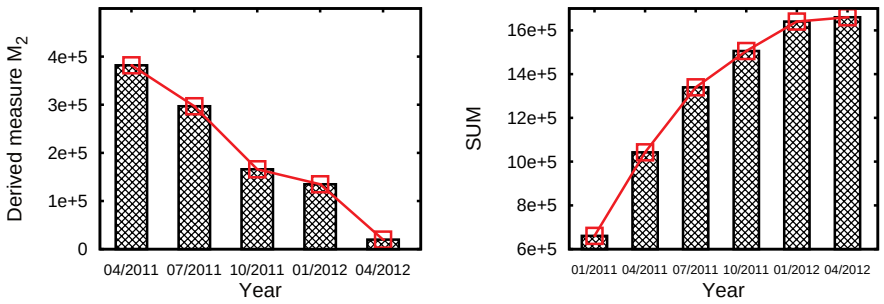


(a) Point insight: Outstanding No.1 (b) raw aggregation result

Figure 5.16. Car sales point insight: SG(⟨ ***,*,SUV,*** ⟩**, Year)**

Table 5.7(c). Its SG($\langle *, \cdots, * \rangle$, Year) means that we compare the tablet sales by year. Its $\mathcal{C}_e = \langle$(SUM,Sales),(Δ_{prev},Year)$\rangle$ means that we analyze the incremental sales between successive years. As shown in Figure 5.17(a), the incremental sales falls with year. In contrast, the raw aggregation result in Figure 5.17(b) only shows a rising trend, but it is not as informative as the above insight.

Figure 5.18(a) refers to the point insight (outstanding Last) in Table 5.7(d). Its SG($\langle$ *,$\cdots$,* $\rangle$,Year) means we compare tablet sales by year. Its $\mathcal{C}_e$ = $\langle$(SUM,Sales), (Δ_{prev}, Year), (Δ_{avg}, Year)$\rangle$ means we analyze the incremental sales of each year over the average of incremental sales among all years. Figure 5.18(b) shows that 07/2012 is the "outstanding last" when compared with the other years. On the other hand, the raw aggregation result in Figure 5.18(b) does not reveal the above insight.

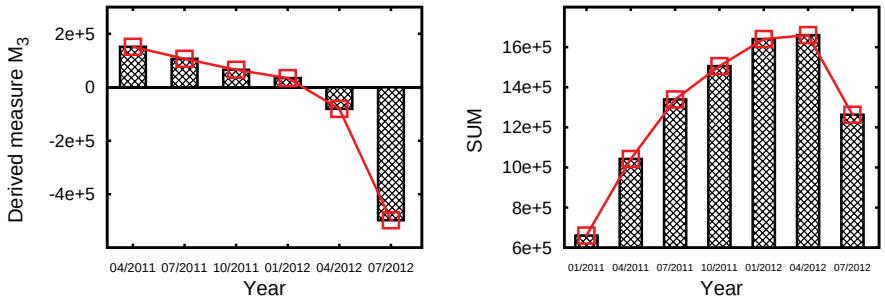


(a) insight: falling trend (b) raw aggregation result

Figure 5.17. Tablet sales shape insight: SG($\langle *, \cdots, * \rangle$, Year)

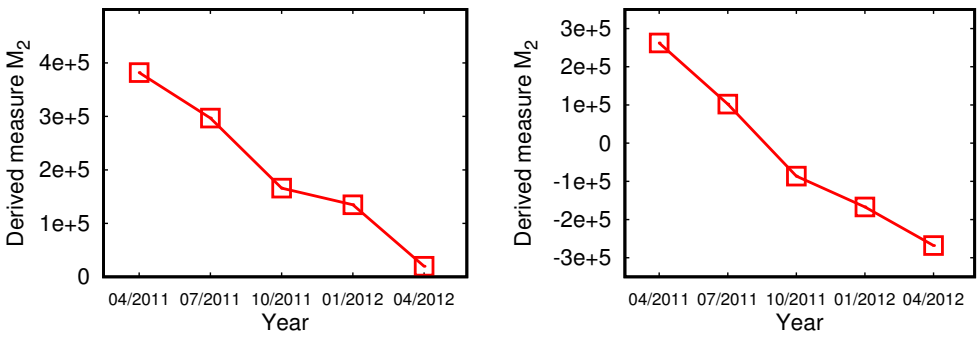
Top-2 insights in tablet sales:

In this section, we discuss the two insights in Table 5.7(c). We illustrate the significance of these two insights, i.e., $\mathsf{Sig}_{\mathsf{Shape}}(\Phi_1)$ and $\mathsf{Sig}_{\mathsf{Shape}}(\Phi_2)$, in Figures 5.19(a) and (b), respectively.



(a) Point Insight: Outstanding Last (b) raw aggregation result

Figure 5.18. Tablet sales point insight: SG(⟨ *,· · ·,* ⟩, Year)



(a) $\mathsf{Sig}_{\mathsf{Shape}}(\Phi_1)$=0.96 (b) $\mathsf{Sig}_{\mathsf{Shape}}(\Phi_2)$ =0.99

Figure 5.19. Significances of top-2 insights in Tablet sales

The top-1 shape insight is with SG({*}, Year) and $\langle$(SUM,Sales),(Δ_{prev},Year)$\rangle$. We compute its insight score as:

$$\begin{aligned}&\mathbb{S}(\mathsf{SG}(*, \mathsf{Year}), \langle(\mathsf{SUM}, \mathsf{Sales}), (\Delta_{prev}, \mathsf{Year})\rangle, \mathsf{Shape})\\ &= \mathsf{Imp}(\mathsf{SG}(*, \mathsf{Year}) \cdot \mathsf{Sig}_{\mathsf{Shape}}(\Phi_1)\\ &= 1 \cdot 0.96 = 0.96.\end{aligned}$$

The top-2 shape insight is with SG({WiFi},Year) and $\langle$(SUM,Sales),(Δ_{prev},Year)$\rangle$. We compute its insight score as:

$$\begin{aligned}&\mathbb{S}(\{\mathsf{WiFi}\}, \mathsf{Year}), \langle(\mathsf{SUM}, \mathsf{Sales}), (\Delta_{prev}, \mathsf{Year})\rangle, \mathsf{Shape})\\ &= \mathsf{Imp}(\mathsf{SG}(\{\mathsf{WiFi}\}, \mathsf{Year}) \cdot \mathsf{Sig}_{\mathsf{Shape}}(\Phi_2)\\ &= 0.643 \cdot 0.99 = 0.64.\end{aligned}$$

5.8.2 Insight Utility Study

In this section, we assess the utility of our top-k insights by 6 domain experts from a leading IT company.

Intern dataset: This dataset is obtained from the University Relationship (UR) team of the above IT company from 2012 to 2016. It contains 1,201 tuples. Each tuple (i.e., an intern) has 15 dimensions. The domain sizes of dimensions are: *Year (4)*, *Group (50)*, *Name (1109)*, *FullTime (2)*, *Start Quarter (13)*, *End Quarter (13), Duration (4)*, *Mentor (300)*, *Nationality (16)*, *Degree (3)*, *Origin (20)*, *University (200)*, *Department (813)*, *Research Area (511)*, *Advisor (831).*

The aggregate function is COUNT in this study.

Study methodology: We first extract top-5 insights with depth-2 and depth-3 composite extractors, and illustrate these insights in Table 5.8. Due to commercial reasons from the data provider, we anonymize some attributes by pseudo-values (e.g., A, B, C, D).

In the following user study, we invite the 3 UR managers and 3 data analysts (from the above IT company) and call them *the domain experts* because they have conducted analysis on this dataset before. We conducted one-to-one interviews with them, collected their comments on our insights, and also asked them to rate the insights by the following two metrics:

1. **Usefulness**: (from 1 to 5), a higher score indicates a more useful insight.

2. **Difficulty**: (from 1 to 5), a higher score indicates that the insight is more difficult to be obtained by using an existing data analysis tool (i.e., Microsoft Excel PivotTable).

Results and Feedback: In these interviews, the domain experts appreciated our top-k insights and found them to be quite useful. They agreed that most of our insights are actionable. For example, The UR team may take actions to balance the nationality ratio based on these insights, and analyze the famous groups by the check-in and check-out interns in each quarter.

We report the ratings of our top-5 insights by the domain expert in Table 5.8. Tables 5.8(a) and (b) illustrate the top-5 insights extracted from the intern dataset with depth-2 and depth-3 composite extractors, respectively. Each

insight has five attributes: usefulness score, difficulty score, sibling group, composite extractor, and its meaning in English. We conducted one-on-one interviews with 6 domain experts (i.e., managers, data analysts) from the leading IT company. In these interviews, we provided the last three attributes to them in a questionnaire, and then asked them to rate the insights by usefulness and difficulty. We reported the average ratings in Table 5.8.

The average usefulness score of depth-2 insights and depth-3 insights are 3.24 and 3.76, respectively. On the other hand, the average difficulty score of depth-2 insights and depth-3 insights are 2.88 and 4.12, respectively. In summary, the domain experts agreed that depth-3 insights are more useful, however, these insights are harder to be summarized by their data analysis tool.

5.8.3 Human Effort Study

In this section, we measure the time taken by users to obtain our insights by using two existing tools: (i) SQL queries, (ii) Microsoft Excel PivotTable. Observe that these tools cannot be readily applied to extract our top-k insights. Nevertheless, we can provide users with a sibling group SG and a composite extractor $\mathcal{C}_e$, then ask users to compute the result of applying $\mathcal{C}_e$ on SG.

In this study, we invited 4 senior database researchers, who are proficient in SQL queries and familiar with Microsoft Excel PivotTable (i.e., a data analysis tool). To make this study manageable, we chose the top-3 insights obtained from car sales with depth-2 composite extractors. This car sales dataset and the depth of composite extractor (i.e., $\tau = 2$) were chosen because it is smallest and simple for human effort study. We provided the participants with the sibling groups and

Usefulness	Difficulty	SG(S,D_i)	Composite Extractor $\mathcal{C}_e$
2.8	2.6	SG({*}, End Quarter)	⟨(COUNT),(Δ_{Prev}, EndQuarter)⟩
Top-1 Insight		We consider the increase of check-out interns in each quarter with the previous quarter. The largest increase happens in 2016Q2	
2.6	2.2	SG({3 months}, Nationality)	⟨(COUNT),(Δ_{avg}, Duration)⟩
Top-2 Insight		The internship duration of interns from Country B is always 3 months.	
3.6	3.6	SG({6 months}, Start Quarter)	⟨(COUNT),(Δ_{Prev}, StartQuarter)⟩
Top-3 Insight		Consider the increase of check-in interns whose internship duration is 6 months among successive start quarters. The largest increase happens in 2015Q1.	
3.6	3.0	SG({PhD}, Nationality)	⟨(COUNT),(Δ_{avg}, Degree)⟩
Top-4 Insight		Consider all interns from Country A. The number of PhD interns is obviously higher than the number of interns with other degrees.	
3.6	3.0	SG({Undergraduate}, Year)	⟨(COUNT),(Rank, Degree)⟩
Top-5 Insight		Regarding the rank (i.e., rank by the number of interns) of each degree among all years, the rank of undergraduates is the lowest in 2013.	

(a) Top-5 insights from Intern Dataset with $\tau = 2$

3.8	3.6	SG({PhD}, Year)	⟨(COUNT), (%, Year),(Δ_{avg}, Degree)⟩
Top-1 Insight		Consider the percentage of interns in each year by their degrees. The percentage of PhD interns is the highest in 2014.	
3.4	4.4	SG({2015Q2}, Group)	⟨(COUNT), (%, Group), (Δ_{avg}, EndQuarter)⟩
Top-2 Insight		Group A at 2015Q2 is the best in terms of the percentage of check-out interns among all quarters.	
3.4	4.4	SG({2016Q1}, Group)	⟨(COUNT), (%, Group), (Δ_{prev}, StartQuarter)⟩
Top-3 Insight		Group B at 2016Q1 is the best in terms of the increased percentage of check-in interns among all successive quarters.	
4.4	4.0	SG({University H}, Group)	⟨(COUNT), (%, Group), (Δ_{avg}, University)⟩
Top-4 Insight		Group C is the most popular group among all research groups for the interns from University H.	
3.8	4.2	SG({2014Q3}, Group)	⟨(COUNT), (%, Group), (Δ_{avg}, EndQuarter)⟩
Top-5 Insight		Group D at 2014Q3 is the worst in terms of the percentage of check-out interns among all quarters.	

(b) Top-5 insights from Intern Dataset with $\tau = 3$

Table 5.8. User study result on the intern dataset with COUNT

Given information	Analysis tool	User 1	User 2	User 3	User 4
(SG,$\mathcal{C}_e$) pairs	SQL Query	20.2	23.4	34.7	38.5
	Excel PivotTable	12.6	10.8	17.3	16.1

Table 5.9. Study on human effort (in minutes)

composite extractors of those insights, and then asked them to compute the result in each case by two methods: (i) SQL queries and (ii) Microsoft Excel PivotTable, respectively. For each participant, we measure the total time of computing all three insights by using SQL queries and Microsoft Excel PivotTable, respectively. We exclude the time of loading data into the database and Excel files.

We reported the time taken in Table 5.9. On overage, the participants spent 29.2 minutes with SQL queries and 14.2 minutes with Microsoft Excel PivotTable to complete the above task. In contrast, our system just takes 0.17 seconds to compute the top-3 insights.

5.9 Performance Evaluation

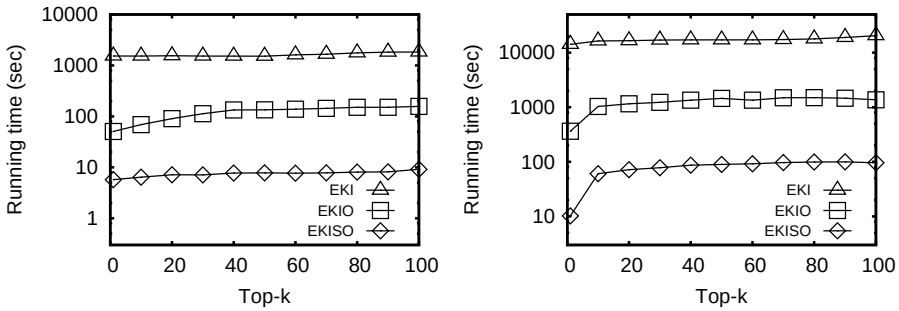
We proceed to evaluate the performance of our solutions. In this work, we store all the raw data and data cube in main memory. The performance of our solutions are measure by the total running time. We conducted all experiments (with single thread) on a machine with an Intel i7-3770 3.4GHz processor, 16GB of memory. We implemented our three solutions in C#. We denote the baseline solution (cf. Algorithm 5.7) for Extracting top-K Insights problem as EKI. EKIO (cf. Algorithm 5.9) applies all optimization techniques in Section 5.6. EKISO (cf. Algorithm 5.12) applies techniques in Section 5.6 (optimizations) and Section 5.7 (computation sharing).

We report the running time (i.e., wall clock time) of solutions in our experiments. Before running experiments, we load the datasets from disk to main memory. SUM is used as the aggregate function. As discussed in Section 5.2.2, unless otherwise stated, the depth of composite extractor τ is set to 2 or 3 by default.

First, we show the efficiency of our solutions on a real dataset. Then, we investigate the efficiency and scalability of our solutions on TPC-H data with respect to various parameters.

5.9.1 Real dataset: Tablet sales

Among the real datasets described in Section 5.8, we use only the tablet sales dataset as it is much larger than the car sales and intern dataset. In Figure 5.20, we vary the result size k and report the running time of solutions on the tablet sales dataset. EKIO performs better than EKI by 10 times, implying the power of our proposed sibling cube and optimization techniques. Since EKISO employs computation sharing techniques, it further outperforms EKIO by an order of magnitude.



(a) at τ=2 (b) at τ=3

Figure 5.20. Runtime on tablet sales vs. result size k

5.9.2 TPC-H dataset

TPC-H[10] data: By default, we generate TPC-H data with scale factor set to 1. We extract the *lineitem* table, which contains 6,001,215 tuples and 16 dimensions. We use *l_extendedprice* (ranging from 901.00 to 104949.50) as measure. We use the following 6 dimensions; their domain sizes are as follows: *l_shipdate*(2526), *l_discount*(11), *l_returnflag*(3), *l_shipinstruct*(7), *l_shipmode*(4) and *l_linestates*(2).

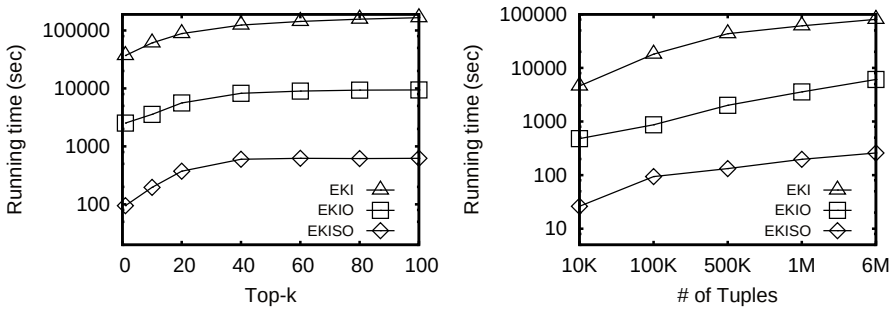
We then study the efficiency and scalability of our methods for various parameters. The default parameter setting is: the number of tuples N=1,000,000, the depth of composite extractor τ=2, the result size k=10, and the number of dimensions d=6.

Effect of result k: Figure 5.21(a) compares the performance of our solutions by varying k from 1 to 100. EKISO is two orders of magnitude faster than EKI. It allows us to obtain the top-1, top-10, top-100 results at 94s, 137s, 622s, respectively. Its running time scales sub-linearly with k.

Effect of number of tuples N: Then we test the performance of our solutions with respect to N. According to Figure 5.21(b), EKISO outperforms EKI by at least two orders of magnitude. Their performance gap widens as N increases. The running time of EKISO also rises sub-linearly with N.

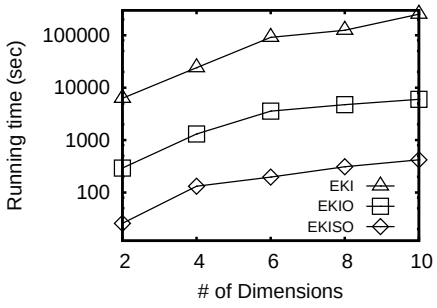
Effect of dimensions d: In Figure 5.21(c), we vary the number of dimensions d from 2 to 10. In addition to the 6 dimensions mentioned earlier, we include 4 more dimensions when $d > 6$: *l_suppkey(1000)* *l_tax(9)*, *l_linenumber(7)* and *l_quantity(50)*. When d is small (≤ 4), we can obtain top-10 results in 1–2

[10] http://www.tpc.org/tpch/

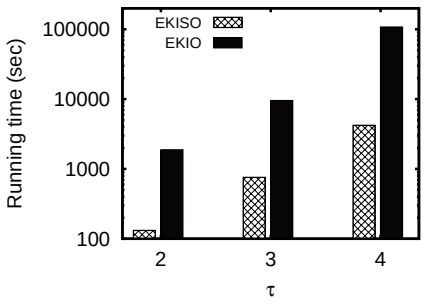


(a) vary result size k

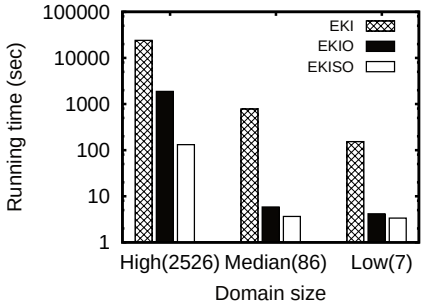
(b) vary number of tuples N



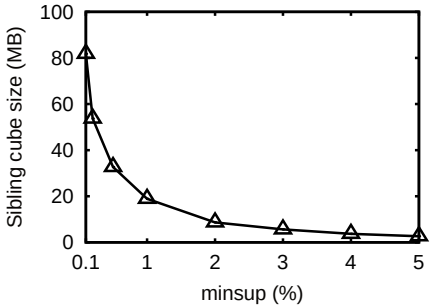
(c) vary dimensionality d



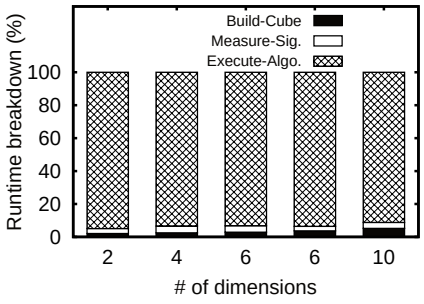
(d) vary depth τ



(e) vary domain size



(f) cube size vs minsup



(g) running time breakdown

Figure 5.21. Performance results on the TPC-H data

minutes. At large values of d, the running time becomes high due to the huge combinations of composite extractors and sibling groups. The performance gap widens with d, and EKISO achieves three orders of magnitude improvement over EKI at d=10.

Effect of depth τ: Next, we examine the performance of our solutions with respect to the depth of composite extractor (from 2 to 4). As illustrated in Figure 5.21(d), the running time rises with the depth because the number of possible composite extractors increases rapidly with τ. EKISO again outperforms the other solutions. We omitted EKI as it is too slow.

Effect of domain size: In this experiment, we vary the domain size of the *l_shipdate* dimension and fix the domain size of other dimensions. The default domain size of *l_shipdate* is 2562 values (one per day). We obtain smaller domain sizes by changing the granularity: 84 values (one per month), 7 values (one per year). As displayed in Figure 5.21(e). EKISO is significantly faster than EKI. EKISO and EKIO perform similarly at low and median domain sizes because the total number of sibling groups is quite small.

5.9.2.1 Sibling cube evaluation

Finally, we evaluate the space and running time overhead of our sibling cube.

Space: Figure 5.21(f) displays the sibling cube size with respect to the iceberg constraint *minsup*. The size of sibling cube ranges from 87.6MB to 1.68MB. All of them can fit in main memory.

Running time breakdown: Figure 5.21(g) shows the breakdown of the run-

ning time of EKISO. We vary the dimensionality d from 2 to 10. Observe that the sibling cube construction time and insight significance measurement time occupy only less than 5% of the total running time.

5.10 Chapter Summary

5.10.1 Conclusion

This work investigates how to extract top-k insights from multidimensional data. We propose a systematic computation framework for this problem, and a suite of performance optimization techniques (e.g., pruning, ordering, sibling cube and computation sharing). Our effectiveness studies (e.g., case study, utility study) have demonstrated that top-k insights reveal meaningful observations on different real datasets. Our best solution EKISO outperforms the baseline solution by several orders of magnitude.

5.10.2 Future work

This work takes the first attempt to extract insights hidden in the data. We want to pursue several promising directions in the future to support both expert data analysts and non-expert executives or managers. First, we plan to incorporate user feedback in insight extraction. Second, for massive datasets, we will investigate how to extract insights efficiently in a distributed environment.

Chapter 6

Conclusion

6.1 Conclusion

In the area of in-memory data management systems, there are many open research questions for both academia and industry to address. In Chapter 1, we present the big picture of in-memory data management systems and discuss the achievements of existing works. We then explore some unsolved problems on in-memory data management.

A major challenge is to identify the problems in existing works or define new promising directions. To tackle it, we survey both in-memory data management techniques (software-level) and the characteristics of CPU and main memory (hardware-level). We discover that the speed gap between CPU and memory is becoming large. This led us to exploit every cycle in modern CPUs in Chapter 3. We also realized existing in-memory data management, in particular, similarity search on high dimensional dataset, did not fully unlock the potential of main

memory. This motivated us to exploit every bit in main memory in Chapter 4.

With the above two fundamental bricks (CPU utilization, RAM utilization), we faced another challenge: how to apply these techniques to applications ? To do so, we first define the concept of *insight* and design an in-memory computation framework to extract top-k insights from multidimensional datasets. Our system can extract insights from different datasets (i.e., sales, business) automatically and effectively, which is not only useful for non-expert users, but also reduce the manual effort of the data experts.

6.2 Future Research

The road for more efficient and intelligent in-memory data management systems is wide open. We now present opportunities for future research.

Our first two works only focus on specific research problems (i.e., accelerate distance computation, utilize main memory for efficient similarity search). In each case, we only exploit these characteristics of existing hardware (e.g., CPUs, RAM). However, both of them can be extended to more general scenarios. On one hand, we can use the SIMD vectorization techniques in Chapter 3 to accelerate the algorithms in Chapter 4 and Chapter 5. On the other hand, we are not aware of existing compilers can convert hardware oblivious algorithms/code to hardware conscious version by exploiting the underlying CPU architecture automatically. If we can provide such tool by extending the techniques in Chapter 3, it will be not only useful for database community but also for compiler community. In Chapter 4, we study how to exploit every bit to accelerate the

similarity search on high dimensional data. How to extend our proposed techniques for in-memory scenario (i.e., the data and index are in the main memory) is an interesting problem.

In Chapter 5, we propose the concept of insight and provide a prototype for insight extraction. it has raised several new issues: (i) different statistical models of insight, (ii) ground truth insights in the dataset and (iii) human-in-the-loop exploration, etc. In addition, the challenge of data exploration is still open, i.e., build an efficient and effective interactive/automatic data exploration system.

We believe that in-memory data management has a bright future ahead. Eventually, in-memory data management will be able to process terabytes of data within seconds on a laptop.

Bibliography

[1] Ibm cogons. http://www.ibm.com/analytics/us/en/technology/cognos-software/.

[2] Ibm watson analytics. https://www.ibm.com/analytics/watson-analytics.

[3] Intel 64 and IA-32 architecutres optimization reference manual. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[4] Quick insights in microsoft power bi. https://goo.gl/ASuJlp.

[5] Sap hana. http://www.sap.com/product/technology-platform/hana.html.

[6] Source codes and datasets for experimental study. http://goo.gl/mwDTxP.

[7] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and David A Wood. Dbmss on a modern processor: Where does time go? In *VLDB*, pages 266–277, 1999.

[8] Chris Anderson. *The long tail: Why the future of business is selling more for less.* Hyperion, 2008.

[9] Ira Assent, Ralph Krieger, Farzad Afschari, and Thomas Seidl. The ts-tree: efficient time series search and retrieval. In *EDBT*, pages 252–263. ACM, 2008.

[10] Vassilis Athitsos, Jonathan Alon, Stan Sclaroff, and George Kollios. Boostmap: A method for efficient approximate similarity rankings. In *CVPR*, pages 268–275. IEEE, 2004.

[11] Vassilis Athitsos, Marios Hadjieleftheriou, George Kollios, and Stan Sclaroff. Query-sensitive embeddings. *ACM Transactions on Database Systems (TODS)*, 32(2):8, 2007.

[12] Vassilis Athitsos, Panagiotis Papapetrou, Michalis Potamias, George Kollios, and Dimitrios Gunopulos. Approximate embedding-based subsequence matching of time series. In *SIGMOD*, pages 365–378. ACM, 2008.

[13] Narayanaswamy Balakrishnan. *Handbook of the logistic distribution.* CRC Press, 2013.

[14] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M Tamer Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*, pages 85–96, 2013.

[15] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *ICDE*, pages 362–373. IEEE, 2013.

[16] Kevin Beyer and Raghu Ramakrishnan. Bottom-up computation of sparse and iceberg cube. In *SIGMOD Record*, pages 359–370. ACM, 1999.

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

[18] Christian Böhm, Stefan Berchtold, and Daniel A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys (CSUR)*, 33(3):322–373, 2001.

[19] Leonid Boytsov and Bilegsaikhan Naidan. Learning to prune in metric and non-metric spaces. In *Advances in Neural Information Processing Systems*, pages 1574–1582, 2013.

[20] Jonathan Brandt. Transform coding for fast approximate nearest neighbor search in high dimensions. In *CVPR*, pages 1815–1822. IEEE, 2010.

[21] Shirley Browne, Jack Dongarra, Nathan Garner, George Ho, and Philip Mucci. A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[22] Alessandro Camerra, Themis Palpanas, Jin Shieh, and Eamonn J. Keogh. isax 2.0: Indexing and mining one billion time series. In *ICDM*, pages 58–67. IEEE, 2010.

[23] Surajit Chaudhuri. What next?: a half-dozen data management research goals for big data and the cloud. In *PODS*, pages 1–4. ACM, 2012.

[24] Surajit Chaudhuri, Umeshwar Dayal, and Vivek Narasayya. An overview of business intelligence technology. *Communications of the ACM*, pages 88–98, 2011.

[25] Shimin Chen, Anastassia Ailamaki, Phillip B Gibbons, and Todd C Mowry. Improving hash join performance through prefetching. *ACM Transactions on Database Systems (TODS)*, 32(3):17, 2007.

[26] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. Efficient implementation of sorting on multi-core simd cpu architecture. *PVLDB*, 1(2):1313–1324, 2008.

[27] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB*, pages 426–435, 1997.

[28] Debabrata Dash, Jun Rao, Nimrod Megiddo, Anastasia Ailamaki, and Guy Lohman. Dynamic faceted search for discovery-driven analysis. In *CIKM*, pages 3–12. ACM, 2008.

[29] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Symposium on Computational Geometry*, pages 253–262, 2004.

[30] Ritendra Datta, Dhiraj Joshi, Jia Li, and James Z Wang. Image retrieval: Ideas, influences, and trends of the new age. *ACM Computing Surveys (CSUR)*, 40(2):5, 2008.

[31] Ritendra Datta, Dhiraj Joshi, Jia Li, and James Ze Wang. Image retrieval: Ideas, influences, and trends of the new age. *ACM Computing Surveys (CSUR)*, 40(2), 2008.

[32] Kyriaki Dimitriadou, Olga Papaemmanouil, and Yanlei Diao. Explore-by-example: An automatic query steering framework for interactive data exploration. In *SIGMOD*, pages 517–528. ACM, 2014.

[33] Hui Ding, Goce Trajcevski, Peter Scheuermann, Xiaoyue Wang, and Eamonn J. Keogh. Querying and mining of time series data: experimental comparison of representations and distance measures. *PVLDB*, 1(2):1542–1552, 2008.

[34] Fabrizio Falchi, Claudio Lucchese, Salvatore Orlando, Raffaele Perego, and Fausto Rabitti. Caching content-based queries for robust and efficient image retrieval. In *EDBT*, pages 780–790. ACM, 2009.

[35] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. Fast subsequence matching in time-series databases. In *SIGMOD*, pages 419–429. ACM, 1994.

[36] Min Fang, Narayanan Shivakumar, Hector Garcia-Molina, Rajeev Motwani, and Jeffrey D Ullman. Computing iceberg queries efficiently. In *VLDB*, pages 299–310, 1999.

[37] Hakan Ferhatosmanoglu, Ertem Tuncel, Divyakant Agrawal, and Amr El Abbadi. Vector approximation based indexing for non-uniform high dimensional data sets. In *CIKM*, pages 202–209. ACM, 2000.

[38] Ada Wai-Chee Fu, Eamonn Keogh, Leo Yung Lau, Chotirat Ann Ratanamahatana, and Raymond Chi-Wing Wong. Scaling and time warping in time series querying. *The VLDB Journal*, 17(4):899–921, 2008.

[39] Junhao Gan, Jianlin Feng, Qiong Fang, and Wilfred Ng. Locality-sensitive hashing scheme based on dynamic collision counting. In *SIGMOD*, pages 541–552. ACM, 2012.

[40] Jinyang Gao, Hosagrahar Visvesvaraya Jagadish, Wei Lu, and Beng Chin Ooi. Dsh: data sensitive hashing for high-dimensional k-nnsearch. In *SIGMOD*, pages 1127–1138. ACM, 2014.

[41] Hector Garcia-Molina and Kenneth Salem. Main memory database systems: An overview. *IEEE Transactions on knowledge and data engineering (TKDE)*, 4(6):509–516, 1992.

[42] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 1999.

[43] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and subtotals. *Data Mining and Knowledge Discovery*, pages 29–53, 1997.

[44] Joachim Gudmundsson, Patrick Laube, and Thomas Wolle. Computational movement analysis. In *Springer handbook of geographic information*, pages 423–438. Springer, 2011.

[45] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach (5. ed.)*. Morgan Kaufmann, 2012.

[46] Stratos Idreos, Olga Papaemmanouil, and Surajit Chaudhuri. Overview of data exploration techniques. In *SIGMOD*, pages 277–281. ACM, 2015.

[47] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys (CSUR)*, 40(4), 2008.

[48] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*, pages 604–613. ACM, 1998.

[49] Hiroshi Inoue, Moriyoshi Ohara, and Kenjiro Taura. Faster set intersection with simd instructions by reducing branch mispredictions. *PVLDB*, 8(3):293–304, 2014.

[50] Yannis E. Ioannidis. The history of histograms (abridged). In *VLDB*, pages 19–30, 2003.

[51] H. V. Jagadish, Nick Koudas, S. Muthukrishnan, Viswanath Poosala, Kenneth C. Sevcik, and Torsten Suel. Optimal histograms with quality guarantees. In *VLDB*, pages 275–286, 1998.

[52] H. V. Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. idistance: An adaptive b^+-tree based indexing method for nearest neighbor search. *ACM Transactions on Database Systems (TODS)*, 30(2):364–397, 2005.

[53] Saurabh Jha, Bingsheng He, Mian Lu, Xuntao Cheng, and Huynh Phung Huynh. Improving main memory hash joins on intel xeon phi processors: An experimental approach. *PVLDB*, 8(6):642–653, 2015.

[54] Martin Kaufmann and Donald Kossmann. Storing and processing temporal data in a main memory column store. *PVLDB*, 6(12):1444–1449, 2013.

[55] Eamonn Keogh and Chotirat Ann Ratanamahatana. Exact indexing of dynamic time warping. *Knowledge and Information Systems*, 7(3):358–386, 2005.

[56] Hans-Peter Kriegel, Peer Kröger, Peter Kunath, and Matthias Renz. Generalizing the optimality of multi-step k-nearest neighbor query processing. In *SSTD*, pages 75–92, 2007.

[57] Martin Krzywinski and Naomi Altman. Points of significance: Significance, p values and t-tests. *Nature methods*, pages 1041–1042, 2013.

[58] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *ICDE*, pages 38–49. IEEE, 2013.

[59] Christian Lemke, Kai-Uwe Sattler, Franz Faerber, and Alexander Zeier. Speeding up queries in column stores. In *International Conference on Data Warehousing and Knowledge Discovery*, pages 117–129, 2010.

[60] Cuiping Li, Beng Chin Ooi, Anthony KH Tung, and Shan Wang. Dada: a data cube for dominant relationship analysis. In *SIGMOD*, pages 659–670. ACM, 2006.

[61] Xiaolei Li, Jiawei Han, Zhijun Yin, Jae-Gil Lee, and Yizhou Sun. Sampling cube: a framework for statistical olap over sampling data. In *SIGMOD*, pages 779–790. ACM, 2008.

[62] Yuhong Li, Leong Hou U, Man Lung Yiu, and Zhiguo Gong. Discovering longest-lasting correlation in sequence databases. *PVLDB*, 6(14):1666–1677, 2013.

[63] P. Y. Lum, G. Singh, A. Lehman, T. Ishkanov, M. Vejdemo-Johansson, M. Alagappan, J. Carlsson, and G. Carlsson. Extracting insights from the shape of complex data using topology. *Scientific Reports*, 3, 2013.

[64] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Multi-probe lsh: Efficient indexing for high-dimensional similarity search. In *VLDB*, pages 950–961, 2007.

[65] Evangelos P. Markatos. On caching search engine query results. *Computer Communications*, 24(2):137–143, 2001.

[66] Ryszard S Michalski. A theory and methodology of inductive learning. In *Machine learning*, pages 83–134. Springer, 1983.

[67] Abdullah Mueen, Eamonn J. Keogh, Qiang Zhu, Sydney Cash, and M. Brandon Westover. Exact discovery of time series motifs. In *SDM*, pages 473–484. SIAM, 2009.

[68] Emmanuel Alexander Müller. *Efficient knowledge discovery in subspaces of high dimensional databases*. PhD thesis, RWTH Aachen University, 2010.

[69] Panagiotis Papapetrou, Vassilis Athitsos, Michalis Potamias, George Kollios, and Dimitrios Gunopulos. Embedding-based subsequence matching in time-series databases. *ACM Transactions on Database Systems (TODS)*, 36(3):17, 2011.

[70] Jian Pei, Yidong Yuan, Xuemin Lin, Wen Jin, Martin Ester, Qing Liu, Wei Wang, Yufei Tao, Jeffrey Xu Yu, and Qing Zhang. Towards multidimensional subspace skyline analysis. *ACM Transactions on Database Systems (TODS)*, pages 1335–1381, 2006.

[71] Hasso Plattner. A common database approach for oltp and olap using an in-memory column database. In *SIGMOD*, pages 1–2. ACM, 2009.

[72] Thanawin Rakthanmanon, Bilson J. L. Campana, Abdullah Mueen, Gustavo E. Batista, M. Brandon Westover, Qiang Zhu, Jesin Zakaria, and Eamonn J. Keogh. Searching and mining trillions of time series subsequences under dynamic time warping. In *KDD*, pages 262–270. ACM, 2012.

[73] Jun Rao and Kenneth A Ross. Cache conscious indexing for decision-support in main memory. In *VLDB*, pages 78–89, 1999.

[74] Kenneth A. Ross. Efficient hash probes on modern processors. In *ICDE*, pages 1297–1301. IEEE, 2007.

[75] Sunita Sarawagi. Explaining differences in multidimensional aggregates. In *VLDB*, pages 7–10, 1999.

[76] Sunita Sarawagi. User-adaptive exploration of multidimensional data. In *VLDB*, pages 307–316, 2000.

[77] Sunita Sarawagi, Rakesh Agrawal, and Nimrod Megiddo. Discovery-driven exploration of olap data cubes. In *EDBT*, pages 168–182. ACM, 1998.

[78] Sunita Sarawagi and Gayatri Sathe. i3: intelligent, interactive investigation of olap data cubes. In *ACM SIGMOD Record*, page 589. ACM, 2000.

[79] Doruk Sart, Abdullah Mueen, Walid A. Najjar, Eamonn J. Keogh, and Vit Niennattrakul. Accelerating dynamic time warping subsequence search with gpus and fpgas. In *ICDM*, pages 1001–1006. IEEE, 2010.

[80] Thomas Seidl and Hans-Peter Kriegel. Optimal multi-step k-nearest neighbor search. In *SIGMOD*, pages 154–165. ACM, 1998.

[81] Thibault Sellam and Martin L Kersten. Meet charles, big data query advisor. In *CIDR*, 2013.

[82] Thibault Sellam, Emmanuel Müller, and Martin L Kersten. Semi-automated exploration of data warehouses. In *CIKM*, pages 1321–1330. ACM, 2015.

[83] Jin Shieh and Eamonn J. Keogh. *i*sax: indexing and mining terabyte sized time series. In *KDD*, pages 623–631. ACM, 2008.

[84] Ali H Shoeb and John V Guttag. Application of machine learning to epileptic seizure detection. In *ICML*, pages 975–982. IEEE, 2010.

[85] Tomás Skopal, Jakub Lokoc, and Benjamin Bustos. D-cache: Universal distance cache for metric access methods. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 24(5):868–881, 2012.

[86] Jingkuan Song, Yang Yang, Yi Yang, Zi Huang, and Heng Tao Shen. Inter-media hashing for large-scale retrieval from heterogeneous data sources. In *SIGMOD*, pages 785–796. ACM, 2013.

[87] Shriram Sridharan and Jignesh M Patel. Profiling r on a contemporary processor. *PVLDB*, 8(2):173–184, 2014.

[88] J Michael Steele. *The Cauchy-Schwarz master class: an introduction to the art of mathematical inequalities*. Cambridge University Press, 2004.

[89] Ayesha Sultana, Norfaeza Hassan, Chengkai Li, Jun Yang, and Cong Yu. Incremental discovery of prominent situational facts. In *ICDE*, pages 112–123. IEEE, 2014.

[90] Bo Tang, Shi Han, Man Lung Yiu, Rui Ding, and Dongmei Zhang. Extracting top-k insights from multi-dimensional data. In *SIGMOD*, pages 1509–1524. ACM, 2017.

[91] Bo Tang, Man Lung Yiu, and Kien A Hua. Exploit every bit: Effective caching for high-dimensional nearest neighbor search. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 28(5):1175–1188, 2016.

[92] Bo Tang, Man Lung Yiu, Yuhong Li, and Leong Hou U. Exploit every cycle: Vectorized time series algorithms on modern commodity cpus. In *International Workshop on In-Memory Data Management and Analytics*, pages 18–39. Springer, 2016.

[93] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. Quality and efficiency in high dimensional nearest neighbor search. In *SIGMOD*, pages 563–576. ACM, 2009.

[94] Roelof van Zwol. Flickr: Who is looking? In *Web Intelligence*, pages 184–190. IEEE, 2007.

[95] Manasi Vartak, Sajjadur Rahman, Samuel Madden, Aditya Parameswaran, and Neoklis Polyzotis. SEEDB: efficient data-driven visualization recommendations to support visual analytics. *PVLDB*, pages 2182–2193, 2015.

[96] Abdul Wasay, Manos Athanassoulis, and Stratos Idreos. Queriosity: Automated data exploration. In *IEEE Congress on Big Data*, pages 716–719. IEEE, 2015.

[97] Roger Weber and Stephen Blott. An approximation based data structure for similarity search. Technical report, Citeseer, 1997.

[98] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, pages 194–205, 1998.

[99] Yair Weiss, Antonio Torralba, and Robert Fergus. Spectral hashing. In *NIPS*, pages 1753–1760, 2008.

[100] Ping Wu, Yannis Sismanis, and Berthold Reinwald. Towards keyword-driven analytical processing. In *SIGMOD*, pages 617–628. ACM, 2007.

[101] Tianyi Wu, Dong Xin, and Jiawei Han. Arcube: supporting ranking aggregate queries in partially materialized data cubes. In *SIGMOD*, pages 79–92. ACM, 2008.

[102] Tianyi Wu, Dong Xin, Qiaozhu Mei, and Jiawei Han. Promotion analysis in multi-dimensional space. *PVLDB*, pages 109–120, 2009.

[103] Dong Xin, Jiawei Han, Xiaolei Li, Zheng Shao, and Benjamin W Wah. Computing iceberg cubes by top-down and bottom-up integration: The starcubing approach. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, pages 111–126, 2007.

[104] Liu Yingfan, Cui Jiangtao, Huang Zi, Li Hui, and Shen Hengtao. Sk-lsh : An efficient index structure for approximate nearest neighbor search. *PVLDB*, 7(9):745–756, 2014.

[105] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 27(7):1920–1948, 2015.

[106] Yi Zhang and Jun Yang. Optimizing i/o for big array analytics. *PVLDB*, 5(8):764–775, 2012.

[107] Jingren Zhou and Kenneth A. Ross. Implementing database operations using simd instructions. In *SIGMOD*, pages 145–156. ACM, 2002.

[108] Haohan Zhu, George Kollios, and Vassilis Athitsos. A generic framework for efficient and effective subsequence retrieval. *PVLDB*, 5(11):1579–1590, 2012.