# EFFICIENT INDEXING FOR SKYLINE QUERIES WITH PARTIALLY ORDERED DOMAINS

LIU BIN

NATIONAL UNIVERSITY OF SINGAPORE

2010

# EFFICIENT INDEXING FOR SKYLINE QUERIES WITH PARTIALLY ORDERED DOMAINS

LIU BIN

(B.SC. FUDAN UNIVERSITY, CHINA)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2010

# Abstract

Given a dataset containing multidimensional data points, a skyline query retrieves a set of data points that are not be dominated by any other points. Skyline queries are useful in multi-preference analysis and decision making applications, and there has been a lot of research interest in the efficient processing of skyline queries. While many skyline evaluation methods have been developed on totally ordered domains for numerical attributes, the efficient evaluation of skyline queries on a combination of totally ordered domains for numerical attributes and partially ordered domains for categorical attributes, which is a more general and challenging problem, is only beginning to be studied. The difficulty in handling skyline queries involving partially ordered domains mainly comes from the more complex dominance relationship among values in partially ordered domains. In this thesis, we present a new indexing method named ZINC (for Z-order Indexing with Nested Code) that supports efficient skyline computation for data with both totally and partially ordered attribute domains. The key innovation in ZINC is based on combining the strengths of the ZB-tree, which is the state-of-the-art index method for computing skylines involving totally ordered domains, with a novel, nested coding scheme that succinctly maps partial orders into total orders. An extensive performance evaluation demonstrates that ZINC significantly outperforms the state-of-the-art indexing schemes for skyline queries.

# Acknowledgements

First of all, I gratefully acknowledge my supervisor, Professor Chee-Yong Chan. I truly appreciate his persistent support and continuous encouragement, for sharing with me his knowledge and experience. During the period of my Master study, he provided constant academic guidance and insightful suggestions to my research and taught me his excellent methodology on overcoming difficulties. Meanwhile, he also set an example for me on persistence, rationality and optimistics. His supervision not only was helpful to my study in the university, but also would be instructive to my whole remaining life.

I wish to thank Dr. Wei Ni, Dr. Chang Sheng and Dr. Shi-Li Xiang who keep providing many fruitful discussions and valuable comments in my research work as well as great help in my daily life. I also need to thank Dr. Zhen-Jie Zhang for offering me some important datasets for the experiments in my research work. I also thank Professor Anthony K. H. Tung and Professor Kian-Lee Tan. As my thesis advisory committee members, they provided constructive advice on my thesis work.

I would like to thank my parents for their endless efforts to provide me with the best possible education. They also keep directing me to be an upright, virtuous and kind person. I also must thank my wife for her continuous spiritual support and encouragement during my long period of study. I hope I will make them proud of my achievement.

Last but not least, I would also like to thank my lovely friends in School of Computing for always being helpful over the years as well as the lovely staff who always try their best to solve all the problems in front of me kindly and smilingly.

# List of Tables

# List of Figures

# Table of Contents

# Chapter 1

# Introduction

Given a dataset containing multidimensional data points, a preference query retrieves a set of data points that could not be dominated by any other points. Nowadays, preference query has emerged as an considerably important tool for multi-preference analysis and decision making in real-life. Skyline query is considered to be the most important branch of preference query. While preference query depends upon a general dominance definition, skyline queries explicitly considers total or partial orders at different dimensions to identify dominance. Given a set of data points $D$, a skyline query returns an interesting subset of points of $D$ that are not dominated (with respect to the attributes of $D$) by any points in $D$. A data point $p_1$ is said to dominate another point $p_2$ if $p_1$ is at least as good as $p_2$ on all attributes, and there exists at least one attribute where $p_1$ is better than $p_2$. Thus, a skyline query essentially computes the subset of "optimal" points in $D$, which has many applications in multi-criteria optimization problems. A skyline query is classified as static if all the partially ordered domains remained unchanged at query time; otherwise, if a user can specify a different partially ordered domain to reflect his preference at query-time, it is considered a dynamic skyline query.

## 1.1 Motivation

There has been a lot of research on the skyline query computation problem, most of which are focused on data attribute domains that are *totally ordered*, where any two values are comparable. Usually, the best value for a totally ordered domain is either its maximum or minimum value and a totally ordered domain can be represented as a chain. In our work, regarding totally ordered domains, we assume the smaller value is more preferred. Many approaches are proposed to handle skyline queries with only totally ordered domains and divided into two categories according to whether rely on any predefined index over the dataset. The category of techniques that do not rely on any predefined index include BNL [4], D&C [4], SFS [27], LESS [21], SalSa [3] and OSP [53] methods, while the other category of techniques that require the dataset is already indexed before skyline evaluation contain Bitmap [45], Index [45], NN [31], BBS [39] and ZB-tree [33] methods.

However, in many applications, some of the attribute domains are *partially ordered* such as interval data (e.g. temporal intervals), type hierarchies, and set-valued domains, where two domain values can be incomparable. Since a partial order satisfies inreflexivity, asymmetry and transitivity, a partially ordered domain can be represented as a directed acyclic graph (DAG). A number of recent research work [10, 42] has started to address the more general skyline computation problem where the data attributes can include a combination of totally and partially ordered domains. SDC$^+$ [10] is the first index method proposed for the more general skyline query problem, which is an extension of the well-known BBS index method [38] designed for totally ordered domains. SDC$^+$ employs an approximate representation of each partially ordered domain by transforming it into two totally ordered domains such that each partially ordered value is presented as an interval value. The state-of-the-art index method for handling partially ordered domains is TSS [42], which is also based on BBS. Unlike SDC$^+$, TSS uses a precise rep-

resentation of a partially ordered value by mapping it into a set of interval values. In this way, TSS avoids the overhead incurred by $SDC^+$ to filter out false positive skyline records.

Recently, a new index method called ZB-tree [33] has been proposed for computing skyline queries for totally ordered domains which has better performance than BBS. The ZB-tree, which is an extension of the $B^+$-tree, is based on interleaving the bitstring representations of attribute values using the Z-order to achieve a good clustering of the data records that facilitates efficient data pruning and minimizes the number of dominance comparisons.

Given the superior performance of ZB-tree over BBS, one question that arises is whether we can extend the ZB-tree approach to obtain an index that has better performance than the state-of-the-art TSS approach, which is based on BBS. Since the ZB-tree indexes data based on bitstring representation, one simple strategy to enhance ZB-tree for partially ordered domains is to apply the well-known bitvector scheme [9] to encode partially ordered domains into bitstrings. We refer to this enhanced ZB-tree as CHE+ZB. We also combine the encoding scheme in TSS with ZB-tree to be another variant of ZB-tree named TSS+ZB. Our experimental evaluation shows that while CHE+ZB, TSS+ZB and TSS have comparable performance, the performance of CHE+ZB and TSS+ZB is often suboptimal as the bitvector encoding scheme does not always produce good data clustering and effective data pruning.

Since partially ordered domains are typically used for categorical attributes to represent user preferences (e.g., preferences for colors, brands, airlines), we expect that the partial orders for representing user preferences are not complex, densely connected structures. As an example, consider the partial order shown in Figure 1.1 representing a user's preference for car brands. The partial order shown has a simple structure consisting of one minimal value (representing the top preference for Ferrari), one max-

imal value (representing the least preference for Yugo), and two chains: the left chain represents the user's preference for German brands (with Benz being preferred over BMW) which are incomparable to the right chain representing the user's preference for Japanese brands (with Toyota being preferred over Honda).

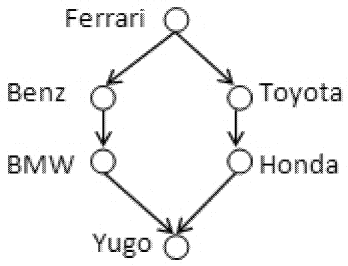



Figure 1.1: Partial order representing a user's preference on car brands.

In our work, we introduce a new indexing approach, called `ZINC` (for Z-order Indexing with Nested Codes), that combines `ZB-tree` with a novel *nested encoding scheme* for partially ordered domains. While our nested encoding scheme is a general scheme that can encode any partial order, the design is targeted to optimize the encoding of commonly used partial orders for user preferences which we believe to have simple or moderately complex structures. The key intuition behind our proposed encoding scheme is to organize a partial order into nested layers of simpler partial orders so that each value in the original partial order can be encoded using a sequence of concise, "local" encodings within each of the simpler partial orders. Our experimental results show that using the nested encoding scheme, `ZINC` significantly outperforms all the other competing methods.

## 1.2 Contributions

In our work, we propose a novel encoding scheme that transforms a partial order into nested layers and encodes all the nodes in the partial order based on the nested lay-

ers. Because each value in the original partial order can be encoded using a sequence of concise, "local" encodings within each of the simpler partial orders, our proposed encoding scheme make it possible to just compare parts of codes while performing dominance comparison between two values in a partially ordered domain. Meanwhile, this encoding scheme maintains the two good properties, i.e., monotonicity property and clustering property, which are provided by `ZB-tree`, to support efficient skyline computation. We also propose a new conception *region* which is common in partial orders and categorize regions into *regular regions* and *irregular regions*. Based on regions, we propose an algorithm to transform a partial order into nested layers. Finally, we conduct an extensive set of experiments and prove that `ZINC` outperforms other existing methods significantly. The experiments are conducted on both synthetic and real datasets. We naturally derive partial orders over real datasets which is novel to the best of our knowledge.

## 1.3 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 surveys related work and Chapter 3 provides more background on `ZB-tree` which is the basis of our proposed `ZINC` approach. In Chapter 4, we introduce our novel nested encoding scheme and describe how `ZINC` evaluates static skyline queries and also propose two variants of `ZB-tree` method which are taken as competitors to `ZINC` in experiments. Chapter 5 presents our experimental evaluation results. Finally, we give a presentation on conclusions and future work in Chapter 6.

# Chapter 2

# Related Work

In this chapter, we review related work on skyline queries, especially the processing of skyline queries with ordered domains.

## 2.1 Skyline Queries with Totally Ordered Domains

After skyline query processing is introduced into database area by [4], researchers devote effort on processing skyline queries with totally ordered domains where the best value for a domain is either its maximum or minimum value.

### 2.1.1 NL, BNL

The first algorithm for processing skyline query is the simple *Nested-Loops* algorithm (`NL` algorithm). It compares every data point with all the data points (including itself), and as a result it can work for any orders. However, obviously `NL` is costly and inefficient. In [4], a variant of `NL` is proposed called *Block Nested-Loops* algorithm (`BNL` algorithm), which is significantly faster and is an a-block-one-time algorithm rather than a-point-one-time as `NL`. `BNL` achieves the efficient processing by a good memory management. The key idea is to maintain in main memory a window, which is used

to keep incomparable data points. When a data point $t_i$ is read from input, $t_i$ is compared to all data points of the window. Based on the comparison, $t_i$ is either discarded, put into the window or put into a temporary file which is allocated in disk and will be considered as input in the next iteration of the algorithm. At the end of each iteration, we can output a part of data points in the window that have been compared to all the data points in the temporary file. These points are not dominated by any other point and do not dominate any points that will be considered in following iterations. Be exactly, these output points are the points that are inserted into the window when the temporary file is empty. Thus, `BNL` achieves the effect of "a-block-one-time". In the best case, the most preferred objects fit into the window and only one or two iterations are needed. Meanwhile, `BNL` has considerable limitations to its performance. First, the performance of `BNL` is affected very much by the discarding effectiveness which `BNL` can not affect at all. Furthermore, there is no guarantee that `BNL` will complete in the optimal number of passes.

### 2.1.2 D&C

*Divide-and-Conquer* algorithm (`D&C` algorithm) [4, 32], as its name indicates, takes a divide-and-conquer strategy. It recursively divides the whole space into a set of partitions, skylines of which are easy to compute. Then, the overall skyline could be obtained as the result of merging these intermediate skylines.

### 2.1.3 SFS, LESS, SalSa, OSP

*Sort-Filter-Skyline* algorithm (`SFS` algorithm) proposed in [27] performs an additional step of pre-sorting before generating skyline points. In this step the input is sorted in some topological sort compatible with the given preference criteria so that a dominating point is placed before its dominated points. The second step is almost the same as the

procedure of BNL, except that in SFS when a point is inserted into the window during a pass, we are sure that it is a most preferred point since no point following it can dominate it. SFS is guaranteed to work within the optimal number of passes since SFS can control the discarding effectiveness. Optimized algorithms, *Linear Elimination Sort for Skyline* (LESS algorithm) and *Sort and Limit Skyline algorithm* (SalSa algorithm), are derived from SFS in [21] and [3]. Finally, the *Object-based Space Partitioning* (OSP algorithm), which is proposed in [53], performs skyline computation in a similar manner, except for that organizes intermediate skyline points in a *left-child/right-sibling* tree, which accelerates the checking of whether the currently read point could be dominated by some intermediate skyline point.

All of the above methods do not rely on any predefined index structure over the dataset. They all require at least one scan through the data source, making them unattractive for producing fast initial response time. Another set of techniques [45, 31, 39, 33] are proposed which require that the dataset are already indexed before skyline evaluation and generally produce shorter response time.

### 2.1.4 Bitmap, Index

The *Bitmap* method is proposed in [45]. This technique encodes in bitmaps all the information needed to decide whether a data point belongs to the skyline. In specific, whether a given data point could be dominated can be identified through some bit-wise operations. This is the first technique utilize the efficiency of bit-wise operations. Meanwhile, the computation of the entire skyline is expensive since it has to retrieve the bitmaps of all data points. Also, because the number of distinct values in a domains might by high and the encoding method is simple, the space consumption might be prohibitive. Another method, called *Index* method, is also proposed in [45]. It partitions the entire data into several lists, indexes each list by a B-tree and uses the trees to find

the local skylines, which are then merged to a global one.

### 2.1.5 NN, BBS

The *branch and bound skyline* (BBS algorithm) proposed in [39] is an optimized method of the *Nearest Neighbor* (NN algorithm) which is proposed in [31] and based upon nearest neighbor search. BBS operates on an R-tree and recursively traverses the R-tree. It performs a nearest neighbor search to find regions/points that are not dominated by the so far found skyline points, and inserts these into a main-memory heap structure. Because BBS visits entries in ascending order of their distances from the origin, each computed point is guaranteed to be a skyline point, and hence can be returned to the user immediately. BBS is presented to be I/O optimal and superior to previous methods. Prior to the publication of the ZB-tree paper [33], BBS was the state-of-the-art approach for data with only totally ordered domains.

### 2.1.6 ZB-tree

ZB-tree proposed in [33] indexes the data points with the help of a Z-order curve which is compatible with the dominance relation. As a result, large number of unnecessary dominance tests are avoided and ZB-tree is found more appropriate in skyline computation than the R-tree. Since our proposed method ZINC is based upon ZB-tree, we will give a description on ZB-tree with more details in Chapter 3.

## 2.2 Skyline Queries with Totally and Partially Ordered Domains

Recently, researchers pay more attention on processing skyline queries with both totally and partially ordered domains, which is common in practice. Difficulty in this area is

mainly due to the more complicated dominance relationship among values in partially ordered domains compared with totally ordered domains.

### 2.2.1 BBS$^+$, SDC, SDC$^+$

Efficient evaluation of skyline queries with both totally and partially ordered domains was first tackled by [10]. Core procedure of BBS$^+$ consists of three phases (1) transform each partially ordered domain into two totally ordered domains, (2) maintain the transformed attributes using an existing indexing scheme and compute the skyline using BBS and (3) prune false positives which are brought in by the lossy transformation in the first phase. As optimized approaches, SDC and SDC$^+$ apply some stratification strategies to data points so that a partial progressiveness could be guaranteed. Limitation of these approaches is the necessary post-processing to eliminate false positives caused by lossy transformation will introduce enormous dominance tests and therefore will harm overall performance significantly. Although this limitation is alleviated with some optimization technique to allow partial progressive skyline computation, the overhead of dominance comparisons still can be high.

### 2.2.2 LatticeSky

LatticeSky is proposed in [36] to efficiently process skyline queries with low-cardinality partially ordered attribute domains using at most two sequential data scans: the first scan is to construct a lattice structure to identify the active dominating domain values, and the second scan is to identify the skyline points by making use of the lattice structure. LatticeSky works well when the partially ordered attribute domains have low cardinality such that the lattice structure can fit in main-memory.

### 2.2.3 IPO-Tree and Adaptive-SFS

Two independent algorithms are proposed in [51] to process dynamic skyline queries with partially ordered domains. The key components in *IPO-Tree* method are the semi-materialization preparation and the important merging property. First of all, materialize result set for each basic dominating relationship in offline style. Then, utilizing the merging property, we can get final result set for any general preference by performing set operation on these materialized result sets. Limitation of this approach are that partial orders on categorical attributes are required to be in a very strict form (something like total orders). Furthermore, cardinalities of involved attributes and dimensionality are required to be quite small since space materialized is in the level of exponential. *Adaptive-SFS* is an evolution on *SFS* algorithm. It starts with a sorted data set. Before processing a user query, it first re-sorts the data set according to the user preference. Unfortunately, the re-sorting could be expensive. Because of the lack of index structure, it has to scan all the concerned data in the processing.

### 2.2.4 TSS

Framework TSS, proposed in [42], can be used to tackle both static and dynamic skyline queries with partially ordered domains. A topological sorting is performed over each partially ordered domain and this sorting assigns each value a topological number. Regarding the static part, sTSS is rather similar with BBS$^+$ except that sTSS introduces additional information, i.e., an additional set of intervals, to capture accurate dominance relationship between values to avoid false positives. Topological numbers and values of totally ordered domains offer the visiting order and guarantee progressiveness of the processing. Currently, sTSS is the state-of-the-art approach in tackling static skyline queries with totally and partially ordered domains. Regarding the dynamic part, dTSS build an R-tree for each group of data points having same values of partially ordered

domains. When a specific query arrives, it first topologically sorts the partially ordered domains and then processes data groups group by group following the topological order and non-dominated points will be inserted into a main memory R-tree. The weakness is obvious that the number of R-trees is considerably large if cardinality and dimensionality of partial orders are not strictly limited.

## 2.3 Other Skyline Related Work

In this section, we review some other skyline related work. This section is not meant to be comprehensive but aim to highlight some of the research directions in this area.

Skyline queries can be seen as a specific case of the Pareto preference queries. The latter one depends upon a more general dominance definition, which is not necessarily derived by taking into account preference orders on well-defined object dimensions compared with skyline queries, which explicitly considers total or partial orders at different dimensions to identify dominance. Pareto preference queries have been investigated in parallel by three research groups, i.e., Chomicki group with work [14, 24, 25, 26, 15], Kießling group with work [30, 50, 28, 29, 23] and Torlone group with work [47, 48, 49]. Accordingly, three Pareto preference operators, i.e., *Winnow* operator, *BMO* operator and *Best* operator, are proposed by these three groups, respectively. All these work mainly focus on four research aspects on Pareto preference quereis: (1) model of preferences, (2) preference algebra, (3) query optimization, and (4) preference query language. Modelling and reasoning with more complex preferences has been proposed in the Artificial Intelligence community. A common model is the CP-net for Conditional Preferences which is studied in [7, 18, 8, 5, 6].

Some related analysis techniques have been proposed as a auxiliary tools for investigation on skyline query processing. A complete space and time complexity analysis for skyline computation was conducted in [22]. Meanwhile, several work [20, 12, 54]

have been proposed for skyline cardinality estimation.

Many work have been done to investigate the relationship between queries with different preferences. Some work [16, 13] investigate a phenomenon that query results could be incrementally refined when preferences are incrementally refined. Some other work [2, 1] focus on the effects of the query refinement on result size or the reuse of skyline results when a query is refined in a progressive fashion. [52, 41] analyze relationship between the skylines in the sub-spaces and super-spaces and propose efficient algorithms for subspace skyline computation. Efficient method on processing skyline queries on high dimensional space is proposed in [11]. Several work [35, 37, 46] have been done to study processing of skyline queries with only totally ordered domains on streaming data. Recently, the work [43] has been proposed to research processing of skyline queries involving partially ordered domains on streaming data. The focus there is on efficient skyline maintenance for streaming non-indexed data which is very different from the focus of our work which is on an index-based approach for static data. Effort is also devoted to probabilistic skyline computation [40] and skyline computation over uncertain data [34].

# Chapter 3

# ZB-tree Method

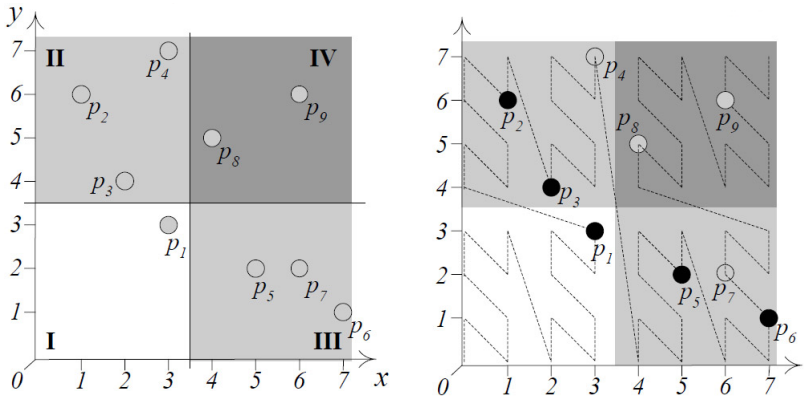
In this chapter, we first review the ZB-tree method [33], which our proposed method is based upon, and then give a brief picture on performance comparison between ZB-tree and BBS which is also presented in [33].

## 3.1 Description of ZB-tree Method

ZB-tree is designed for data where all attributes have totally ordered domains. It first maps each multi-dimensional data point to a one-dimensional Z-address according to Z-order curve by interleaving the bitstring representations of the attribute values of that point. For example, given a 2D data point (0,5), its bitstring representation is (000,101) and its Z-address is (010001). Figure 3.1(b) depicts an example of Z-order curve on a given set of 2D data points shown in Figure 3.1(a). By ordering data points in non-descending order of their Z-addresses, ZB-tree has the following two useful properties. The *monotonic ordering property* states that a data point $p$ can not be dominated by any point that succeeds $p$ in the Z-order. The *clustering property* states that data points ordered by Z-addresses are naturally clustered into regions, which enables very efficient region-based dominance comparisons and data pruning.

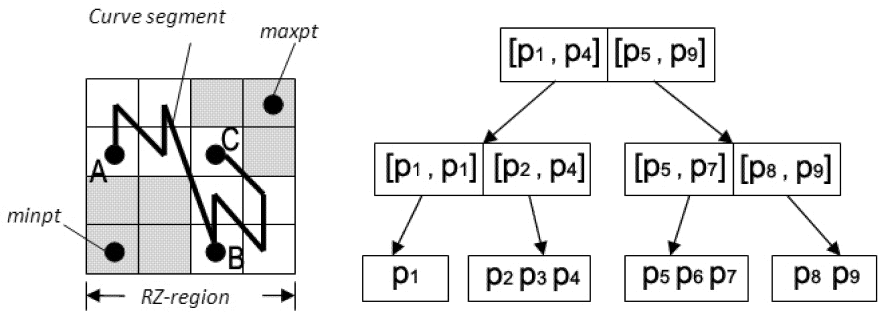
A ZB-tree is a variant of $B^+$-tree using Z-addresses as keys. The data points are stored in the leaf nodes sorted in non-descending order of their Z-addresses. Figure 3.2(b) depicts the ZB-tree built on the dataset shown in Figure 3.1(a), where the minimum and maximum leaf node capacity is 1 and 3, respectively. Each internal node entry (corresponding to some child node $N$) maintains an interval, denoted by a pair of Z-addresses, representing a segment of the Z-order curve (called the Z-region) covering all the data points in the leaf nodes in the index subtree rooted at $N$. Specifically, an interval is represented by $(minpt, maxpt)$, where $minpt$ and $maxpt$ correspond, respectively, to the minimum and maximum Z-addresses of the smallest square region, called the *RZ-region*, that encloses the Z-region. An example of RZ-region is shown by the $4 \times 4$ square in Figure 3.2(a) where three data points $A$, $B$, and $C$ are bounded; the $minpt$ and $maxpt$ indicated are the minimum and maximum Z-addresses of the enclosed square RZ-region. The $minpt$ (resp., $maxpt$) of an RZ-region can be easily derived by appending 0s (resp., 1s) to the common prefix of Z-addresses of the two endpoints of the corresponding curve segment.

Another point worth mentioning is about organization of data points in ZB-tree, which is not exactly the same as in $B^+$-tree. In $B^+$-tree, all data points are tightly packed to minimize the storage overhead. Nevertheless, applying the same data organization principle to ZB-tree would result in large RZ-regions which is not quite helpful in pruning search space. Following the example shown in Figure 3.1(b), all the 9 data points should be allocated into 3 seperate leaf nodes with maximum leaf node capacity being 3. Among these 3 leaf nodes, $p_7$, $p_8$ and $p_9$ are allocated in the third node and resulting RZ-region turns out to be large. Because this large RZ-region can not be dominated by any data point, the corresponding leaf node as well as all the enclosed data points need to be visited. Actually, we can see that points $p_8$ and $p_9$ can be pruned when point $p_1$ has been identified as a skyline point. As a result, data organization




(a) 2D data points (b) Z-order curve

Figure 3.1: An example of Z-order curve




(a) RZ-region (b) ZB-tree

Figure 3.2: Example of RZ-region and ZB-tree

in ZB-tree strategically trade some storage overhead for pruning efficiency through putting as many data points in the same RZ-region as possible into a node instead of filling up the entire node capacity. As shown in Figure 3.2(b), point $p_1$, rather that points $p_1$ to $p_3$ can be put into the first leaf node. Then, points $p_2$ to $p_4$ are inserted into the second one, while points $p_5$ to $p_7$ into the third one. Finally, points $p_8$ and $p_9$ are allocated into the last one. Although this data point organization in ZB-tree requires some extra storage overhead, the search performance is significantly improved since unnecessary node traversal and comparisons between incomparable nodes are avoided.

The ZB-tree method utilizes an in-disk ZB-tree (named *SRC*) and an in-memory

ZB-tree (named *SL*) to index data points and computed skyline points, respectively. Skyline points are computed by invoking ***ZSearch(SRC)*** as shown in Algorithm 1 to recursively traverse *SRC* in depth-first manner to find regions or data points that are not dominated by the current skyline points in *SL*. Given two RZ-regions $R$ and $R'$, the ZB-tree exploits the following three properties of RZ-regions to optimize dominance comparisons: (P1) If *minpt* of $R'$ is dominated by *maxpt* of $R$, then the whole $R'$ is dominated by $R$. (P2) If *minpt* of $R'$ is not dominated by *maxpt* of $R$ and *maxpt* of $R'$ is dominated by *minpt* of $R$, then some points in $R'$ could be dominated by $R$. (P3) If the *maxpt* of $R'$ is not dominated by the *minpt* of $R$, then no point in $R'$ can be dominated by any point in $R$.

For each visited index entry (either internal or leaf entry) $E$, ZSearch invokes ***Dominate(SL,E)*** algorithm as shown in Algorithm 2 to check whether the corresponding RZ-region or data point of $E$ can be dominated by skyline points in *SL*. ***Dominate(SL,E)*** traverses *SL* in a breadth-first manner and performs dominance comparison between each visited entry and $E$ based on properties P1 to P3. In particular, if $E$ is an internal entry and it is dominated by some skyline point due to $P1$, then the search of the index subtree rooted at the node corresponding to $E$ is pruned.

Due to the monotonic ordering property of ZB-tree, each visited data point in the leaf node that is not dominated by any skyline point in *SL* is guaranteed to be a skyline point and can be inserted into *SL* and output to the users immediately. The clustering property of ZB-tree enables many index subtree traversals to be efficiently pruned leading to its superior performance over BBS [38].

## 3.2 Performance Evaluation of ZB-tree against BBS

Performance evaluation of ZB-tree against BBS is conducted on both synthetic and real datasets.

**Algorithm 1**: ZSearch(*SRC*)

```
   Input:  SRC: ZB-tree indexing source data points;
   Local: s: Stack;
   Output: SL: ZB-tree indexing skyline points;
 1 s.push(SRC's root);
 2 while s is not empty do
 3     n = s.pop();
 4     if not Dominate(SL,n) then
 5         if n is an internal node then
 6             foreach children node c of n do
 7                 s.push(c);
 8         else
 9             foreach data point c in n do
10                 if not Dominate(SL,c) then
11                     SL.insert(c);
12 output SL;
```

**Algorithm 2**: Dominate(*SL*,*E*)

```
   Input:  SL: ZB-tree indexing skyline points
   E: the index entry under dominance comparison
   Local: q: Queue;
   Output: TRUE if E is dominated, FALSE otherwise;
 1 q.enqueue(SL's root);
 2 while q is not empty do
 3     n = q.dequeue();
 4     if n is an internal node then
 5         foreach children node c of n do
 6             if c's maxpt can dominate E's minpt then
 7                 return TRUE; /* P1 */
 8             else if c's minpt can dominate E's maxpt then
 9                 q.enqueue(c); /* P2 */
10     else
11         foreach children data point p of n do
12             if p can dominate E's minpt then
13                 return TRUE;
14 return FALSE;
```

Among them, synthetic datasets are generated based on *anti-correlated* distribution and *independent* distribution. The data dimensionality varies from 4 to 16 and the data cardinality ranges from $10K$ to $10000K$ in order to evaluate scalability of `ZB-tree` against `BBS`. The *elapsed time* and the *I/O cost* are employed as the main performance metrics. Regarding implementation, since Z-addresses can be used to derive orginal attribute values, only Z-addresses are kept in `ZB-tree`, while data points are kept in the R-tree adopted by `BBS`. While varying data dimensionality from 4 to 16, `ZB-tree` keeps outperforming `BBS` for both distributions regarding elapsed time. The superior performance of `ZB-tree` depends on the fact that `ZB-tree` can determine whether a skyline point or an RZ-region is dominated at upper-level nodes of *SL* and result in shorter elapsed time than `BBS` which needs to reach the leaf nodes of the main memory R-tree every time. The gap between performance of the two algorithms increases as data dimensionality increases until the dimensionality reaches 12 where over 95% of data points are skyline points. Regarding I/O cost, `ZB-tree` incurs lower I/O cost than `BBS` in low data dimensionality and similar I/O cost as `BBS` in high data dimensionality due to the curse of dimensionality. While varying data cardinality from $10K$ up to $10000K$, the elapsed time of both algorithms increases and `ZB-tree` produces a shorter elapsed time. The performance comparison regarding I/O cost is not presented due to space consideration.

Performance evaluation is also conducted on 3 real datasets, i.e., NBA, HOU and FUEL datasets, which follow anti-correlated, independent and correlated distribution, respectively. The experimental results of the real datasets show that `ZB-tree` clearly outperforms `BBS` for both the elapsed time and the I/O cost.

In summary, `ZB-tree` is capable to outperform `BBS` with both synthetic and real datasets under various settings. `ZB-tree` has become state-of-the-art approach in tackling skyline queries with only totally ordered domains.

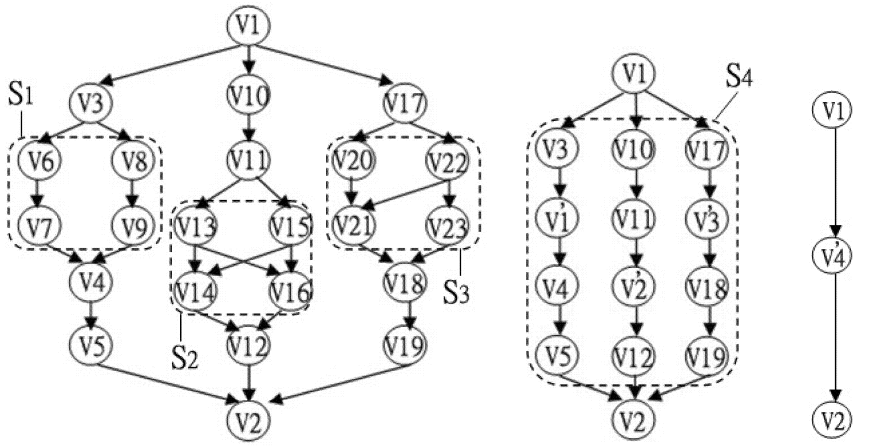
# Chapter 4

# ZINC

In this section, we present our proposed indexing method named ZINC (for Z-order Indexing with Nested Code) that supports efficient skyline computation for data with both totally as well as partially ordered domains. ZINC is basically a ZB-tree that uses a novel encoding scheme to map partially ordered domain values into bitstrings. Once the partially ordered domain values have been mapped into bitstrings, the mapped bitstrings of all the attributes (whether totally or partially ordered domains) of the records will be used to construct a ZB-tree index. Thus, the index construction and search algorithm for ZINC is equivalent to those of ZB-tree except that ZINC uses a different method for dominance comparisons between partially ordered domain values.

## 4.1 Nested Encoding Scheme

In this section, we introduce a novel encoding scheme, called *nested encoding* (or NE, for short), for encoding values in partially ordered domains. The encoding scheme is designed to be amenable to Z-order indexing such that when the encoded values are indexed with a ZB-tree, the two desirable properties of monotonicity and clusteredness of ZB-tree are preserved.




(a) $G_0$ (b) $G_1$ (c) $G_2$

Figure 4.1: Graph reduction

We represent a partial order by a directed graph $G = (V, E)$, where $V$ and $E$ denote, respectively, the set of vertices and edges in $G$ such that given $v, v' \in V$, $v$ dominates $v'$ iff there is a directed path in $G$ from $v$ to $v'$. Given a node $v \in V$, we use $parent(v)$ (resp., $child(v)$) to denote the set of parent (resp., child) nodes of $v$ in $G$. A node $v$ in $G$ is classified as a minimal node if $parent(v) = \emptyset$; and it is classified as a maximal node if $child(v) = \emptyset$. We use $min(G)$ and $max(G)$ to denote, respectively, the set of minimal nodes and maximal nodes of $G$.

Given a partial order $G_0$, the key idea behind nested encoding is to view $G_0$ as being organized into nested layers of partial orders, denoted by $G_0 \rightarrow G_1 \cdots \rightarrow G_{n-1} \rightarrow G_n$, $n \geq 0$, where each $G_i$ is nested within a simpler partial order $G_{i+1}$, with the last partial order $G_n$ being a total order. As an example, consider the partial order $G_0$ shown in Figure 4.1, where $G_0$ can be viewed as being nested within the partial order $G_1$ which is derived from $G_0$ by replacing three subsets of nodes $S_1 = \{v_6, v_7, v_8, v_9\}$, $S_2 = \{v_{13}, v_{14}, v_{15}, v_{16}\}$ and $S_3 = \{v_{20}, v_{21}, v_{22}, v_{23}\}$ in $G_0$ by three new nodes $v'_1$, $v'_2$ and $v'_3$, respectively, in $G_1$[1].

[1]Note that the presentation here has been simplified for conciseness. The `PO-Reduce` algorithm in Section 4.3 actually performs the replacement in two steps, where $S_1$ and $S_2$ are first replaced in the one step followed by $S_3$ in another step.

$G_1$ in turn can be viewed as being nested within the total order $G_2$ which is derived from $G_1$ by replacing the subset of nodes $S_4 = \{v_3, v'_1, v_4, v_5, v_{10}, v_{11}, v'_2, v_{12}, v_{17}, v'_3, v_{18}, v_{19}\}$ by one new node $v'_4$ in $G_2$. We refer to the new nodes $v'_1$, $v'_2$, $v'_3$ and $v'_4$ as *virtual nodes*; and each virtual node $v'_j$ in $G_{i+1}$ is said to *contain* each of the nodes in $S_j$ that $v'_j$ replaces. By viewing $G_0$ in this way, each node in $G_0$ can be encoded as a sequence of encodings based on the nested node containments within virtual nodes.

In the following, we present a formal definition of our nested encoding scheme.

## 4.2 Horizontal, Vertical, and Irregular Regions

**Definition 1.** *Given a partial order G, a non-empty subgraph G′ of G is defined to be a* region *of G if G′ satisfies all the following conditions: (1) every minimal node in G′ has the same set of parent nodes in G; i.e., parent*$(v) = parent(v'), \forall\ v, v' \in min(G')$*; (2) every maximal node in G′ has the same set of child nodes in G; i.e., child*$(v) = child(v'), \forall\ v, v' \in max(G')$*; and (3) only a minimal or maximal node in G′ can have a parent or child node in* $G - G'$*; i.e.,* $parent(v) \cup child(v) \subseteq G', \forall\ v \in G' - min(G') - max(G')$.

In the above example shown in Figure 4.1, $S_1$, $S_2$, $S_3$ and $S_4$ are regions. A region $R$ in a partial order $G_1$ can be replaced by a virtual node $v'$ to derive a simpler partial order $G_2$ while "preserving" the dominance relationship between the nodes in $R$ and nodes in $G_1 - R$. Specifically, the dominance relationships in $G_1$ are preserved in $G_2$ in the sense that (1) if a node $v$ in $G_2$ dominates $v'$, then $v$ also dominates each node of $R$ in $G_1$; and (2) if a node $v$ in $G_2$ is dominated by $v'$, then $v$ is also dominated by each node of $R$ in $G_1$.

For our nested encoding scheme to be amenable for Z-order indexing, a region ideally should have a simple "regular" structure so that its encoding is concise. In this

paper, we classify a region into a *regular* or an *irregular* region depending on whether the region can be encoded concisely. In the following, we introduce two types of regular regions, namely, *vertical regions* and *horizontal regions*.

**Definition 2.** *A region $G'$ of a partial order $G$ is defined to be a* vertical region *if $G'$ satisfies all the following conditions: (1) the nodes in $G'$ can be partitioned into a disjoint collection of $k$ non-empty chains $C_1, \cdots, C_k$, $k > 1$, where each chain $C_i$ represents a total order, such that $child(v) \cap C_j = \emptyset$ for each $v \in C_i, C_i \neq C_j$; and (2) $G'$ is a maximal subgraph of $G$ that satisfies condition (1).*

**Definition 3.** *A region $G'$ of a partial order $G$ is defined to be a* horizontal region *if $G'$ satisfies all the following conditions: (1) the nodes in $G'$ can be partitioned into $k$ non-empty, disjoint subsets $S_0, \cdots, S_{k-1}$, $k \geq 1$; (2) $min(G') = S_0$ such that $child(v) = S_1, \forall\ v \in S_0$; (3) $max(G') = S_{k-1}$ such that $parent(v) = S_{k-2}, \forall\ v \in S_{k-1}$; (4) for each $i \in (0, k-1)$ and for every node $v \in S_i$, $parent(v) = S_{i-1}$ and $child(v) = S_{i+1}$; and (5) $G'$ is a maximal subgraph of $G$ that satisfies conditions (1) to (4).*

For a horizontal region $R$ where the nodes are partitioned into $k$ subsets, $S_0, \cdots, S_{k-1}$, as defined, we refer to $R$ as a *$k$-level horizontal region*, and refer to a node in $S_i$, $i \in [0, k-1]$ as a *level-$i$ node*.

**Definition 4.** *Consider a region $G'$ of a partial order $G$. $G'$ is defined to be a* regular region *if $G'$ is either a vertical or horizontal region. $G'$ is defined to be an* irregular region *if it satisfies all the following conditions: (1) $G'$ is not a regular region; and (2) $G'$ is a minimal subgraph of $G$ that satisfies condition (1).*

Note that a vertical region corresponds to a collection of total orders while a horizontal region corresponds to a weak order[2]. We have defined a regular region to be a

[2]A partial order G is defined to be a *weak order* if incomparability is transitive; i.e., $\forall v_1, v_2, v_3 \in G$, if $v_1$ is incomparable with $v_2$ and $v_2$ is incomparable with $v_3$, then $v_1$ is incomparable with $v_3$.

maximal subgraph in order to have as large a regular structure as possible to be encoded concisely. In contrast, an irregular region is defined to be a minimal subgraph so as to minimize the number of nodes encoded using a lengthy encoding. For example, the regions $S_1$, $S_2$ and $S_3$ shown in $G_0$ in Figure 4.1, respectively, are vertical, horizontal and irregular regions.

## 4.3 Partial Order Reduction Algorithm

In this section, we present an algorithm, termed `PO-Reduce`, that takes a partial order $G_0$ as input and computes a *reduction sequence*, denoted by $G_0 \rightarrow G_1 \cdots \rightarrow G_{n-1} \rightarrow G_n$, $n \geq 0$, that transforms $G_0$ into a total order $G_n$, where each $G_{i+1}$ is derived from $G_i$ by replacing some regions in $G_i$ by virtual nodes. The reduction sequence will be used by our nested encoding scheme to encode each node in $G_0$.

Given an input partial order $G_i$, algorithm `PO-Reduce` operates as follows:(1) Let $S = \{S_1, \cdots S_k\}$ be the collection of regular regions in $G_i$; (2) If $S$ is empty, then let $S = \{S_1\}$, where $S_1$ is an irregular region in $G_i$ that has the smallest size (in terms of the number of nodes) among all the irregular regions in $G_i$. (3) Create a new partial order $G_{i+1}$ from $G_i$ as follows. First, initialize $G_{i+1}$ to be $G_i$. For each region $S_j$ in $S$, replace $S_j$ in $G_{i+1}$ with a virtual node $v'_j$ such that $parent(v'_j) = parent(v)$ with $v \in min(S_j)$ and $child(v'_j) = child(v)$ with $v \in max(S_j)$. (4) If $G_{i+1}$ is a total order, then the algorithm terminates; otherwise, invoke the `PO-Reduce` algorithm with $G_{i+1}$ as input.

The time complexity of `PO-Reduce` to reduce a partial order $G_0$ is $O(|V_0|^2 \times |E_0|)$, where $|V_0|$ and $|E_0|$ are total number of nodes and edges in $G_0$, respectively.

When a node $v$ in a region $R$ is being replaced by a virtual node $v'$, we say that $v$ is *contained in* $v'$ (or $v'$ *contains* $v$), denoted by $v \xrightarrow{R} v'$. Clearly, the node containment can be nested; for example, if $v$ is contained in $v'$, and $v'$ is in turn contained in $v''$, then $v$ is also contained in $v''$. Given an input partial order $G_0$, we define the *depth* of a

node $v$ in $G_0$ to be the number of virtual nodes that contain $v$ in the reduction sequence computed by algorithm `PO-Reduce`. As an example, consider the value $v_6$ in Figure 4.1 and let $R_0 = \{v_6, v_7, v_8, v_9\}$ and $R_1 = \{v_3, v'_1, v_4, v_5, v_{10}, v_{11}, v'_2, v_{12}, v_{17}, v'_3, v_{18}, v_{19}\}$. The containment sequence of $v_6$ is $v_6 \xrightarrow{R_0} v'_1 \xrightarrow{R_1} v'_4$ and therefore, depth of node $v_6$ is 2. The containment sequence of $v_3$ is $v_3 \xrightarrow{R_1} v'_4$ and therefore, depth of node $v_3$ is 1.

Thus, given an input partial order $G_0$, algorithm `PO-Reduce` outputs the following: (1) the partial order reduction sequence, $G_0 \rightarrow G_1 \cdots \rightarrow G_{n-1} \rightarrow G_n$, $n \geq 0$, where $G_n$ is a total order; and (2) the node containment sequence for each node in $G_0$. If a node $v_0$ in $G_0$ has a depth of $k$, we can represent the node containment sequence for $v_0$ by $v_0 \xrightarrow{R_0} v_1 \cdots \xrightarrow{R_{k-1}} v_k$, where each $v_i$ is contained in the region $R_i$, $i \in [0, k)$.

Given a partial order $G_i$, we use $V_i$ and $E_i$ to denote the set of nodes and edges of $G_i$, respectively, and $|V_i|$ and $|E_i|$ denote the total number of nodes and edges of $G_i$, respectively. In `PO-Reduce`($G_i$), as shown in Algorithm 3, we first partition the node set of $G_i$, i.e., $V_i$, into a number of partitions by invoking function Partition($G_i$) (resp., Partition'($G_i$)) so that each partition has the same parent set (resp., child set), i.e., for any two different values $v_i$ and $v_j$ belonging to the same partition, we have $parent(v_i) = parent(v_j)$ (resp., $child(v_i) = child(v_j)$). We store those partitions having 2 or more nodes in a global variable $L$ (resp., $L'$), which would be used by following functions. The task of Partition($G_i$) (resp., Partition'($G_i$)) can be accomplished straightforwardly in a cost of $O(|E_i|)$ because no edge needs to be visited more than once. Function Search-VR($G_i$) and Search-HR($G_i$) are used to identify vertical regions and horizontal regions, respectively. With a guarantee that all found regular regions (either vertical or horizontal regions) are non-overlapped, we replace each of these with a virtual node. If no regular region can be found, we will invoke the function Search-Min-IRR($G_i$) to search for the minimal irregular region and replace it by a virtual node. After the replacement of either regular regions or the minimal irregular region, we need to output

the corresponding node containment as well as the structure of the obtained partial order $G_{i+1}$ as a step of the partial order reduction sequence. If $G_{i+1}$ is a total order, the program terminates. Otherwise, we invoke `PO-Reduce`($G_{i+1}$) for further partial order reduction.

In Search-VR($G_i$), as shown in Algorithm 4, for each node set in $L$, we view the node set as the set of minimal nodes of the potential vertical region and store it in a local variable *min-set*. We proceed to obtain the corresponding chain below each node of *min-set* and store maximal node of each such chain in *max-set*. Then, we partition the *max-set* into a number of partitions so that each partition own the same child set, i.e., for any two values $v_i$ and $v_j$ belonging to the same partition, we have $child(v_i) = child(v_j)$. So far, the corresponding chains of each partition of *max-set* form a vertical region. We insert all the found vertical regions into *VR-set* and proceed to the next un-examined node set in $L$. We also remove the node set, based on which a vertical region is found successfully, from $L$ because the node set can not be a part of another region. Taking $G_i$ which is shown in Figure 4.2(a) as an instance, we store the $\{v_2, v_3, v_4, v_5\}$, which is a node set in $L$, in *min-set*. Then, four corresponding chains are obtained for this node set and *max-set* becomes $\{v_8, v_9, v_{10}, v_{11}\}$. The *max-set* is partitioned into two partitions, i.e., $\{v_8, v_9\}$ and $\{v_{10}, v_{11}\}$, each of which own the same child set. According to the partitioning, we obtain two vertical regions, one of which contains the chains $\{v_2, v_6, v_8\}$ and $\{v_3, v_9\}$, while the other contains the chains $\{v_4, v_{10}\}$ and $\{v_5, v_7, v_{11}\}$. We replace the two vertical regions by virtual nodes $v'_1$ and $v'_2$, respectively and the obtained $G_{i+1}$ is shown in Figure 4.2(b).

Before getting into Search-HR($G_i$), which is presented in Algorithm 5, we give a definition *HR-satisfy* between two node sets, which is describing the relationship between neighbor layers of a weak order.

**Definition 5.** *Given two non-overlapped node sets $S_1$ and $S_2$ in a partial order $G$, $S_1$* HR-satisfies *$S_2$ if $S_1$ and $S_2$ satisfy the following conditions: (1) $|S_1| > 1$, $|S_2| > 1$; (2)*

**Algorithm 3**: PO-Reduce($G_i$)

```
Input: G_i: a partial order;
Global: L: the node sets having same parent set; L': the node sets having same child set;
Output: Node containment sequence and partial order reduction sequence;
1  L = Partition(G_i);
2  L' = Partition'(G_i);
3  VR-set = Search-VR(G_i);
4  HR-set = Search-HR(G_i);
5  S = VR-set ∪ HR-set;
6  if S is not empty then
7      replace every region in S by a virtual node to obtain G_{i+1};
8      output node containment for every replaced region;
9  else
10     IRR = Search-Min-IRR(G_i);
11     replace IRR by a virtual node to obtain G_{i+1};
12     output node containment for the replaced IRR;
13 output structure of G_{i+1};
14 if G_{i+1} is a total order then
15     terminate;
16 else
17     PO-Reduce(G_{i+1});
```

**Algorithm 4**: Search-VR($G_i$)

```
Input: G_i: a partial order;
Output: VR-set: all vertical regions in G_i;
1  min-set = the first node set in L;
2  VR-set = ∅;
3  while min-set is non-empty do
4      foreach node n in min-set do
5          n' = child node of n;
6          while outdegree and indegree of n' is 1 do
7              n' = child of n';
8          put parent of n' in this chain into max-set;
9      partition max-set so that each partition has same child set;
10     foreach partition of max-set do
11         put corresponding chains as a VR into VR-set;
12     remove this node set from L;
13     min-set = the next node set in L;
14     max-set = ∅;
15 return VR-set;
```

**Algorithm 5**: Search-HR($G_i$)

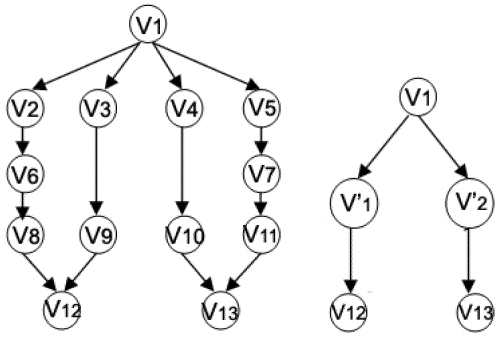
**Input**: $G_i$: a partial order;
**Output**: HR-set: all horizontal regions in $G_i$;

```
1  min-layer = the first node set in L;
2  HR-set = ∅;
3  while min-layer is non-empty do
4      cur-layer = min-set;
5      while exist a non-empty set T so that cur-layer HR-satisfies T do
6          cur-layer = T;
7      if a sequence of layers are found then
8          put the found layers as a HR into HR-set;
9      remove this node set from L;
10     min-layer = the next node set in L which is not included in any found HR-region;
11 return HR-set;
```

**Algorithm 6**: Search-Min-IRR($G_i$)

**Input**: $G_i$: a partial order;
**Local**: $s$, $s'$: minimal and maximal node set of the potential region, respectively; $r$: the current potential region;
**Output**: Min-IRR: the minimal irregular region in $G_i$;

```
1  r = ∅;
2  Bool sig = True;
3  foreach node set s in L do
4      foreach node set s' in L' do
5          r = s ∪ s';
6          foreach node n between s and s' do
7              if introduction of n violates definition of region w.r.t. r then
8                  sig = False; break;
9              else
10                 put n into r;
11         if sig then
12             r is guaranteed to be an irregular region;
13             Min-IRR = minimal found irregular region;
14         else
15             sig = True;
16 return Min-IRR;
```




(a) $G_i$ (b) $G_{i+1}$

Figure 4.2: Example of searching for vertical regions

*each node in* $S_1$ *has the same child set which is* $S_2$*, i.e., for any* $v \in S_1$*,* $child(v) = S_2$*; and (3) each node in* $S_2$ *has the same parent set which is* $S_1$*, i.e., for any* $v \in S_2$*,* $parent(v) = S_1$.

In Search-HR($G_i$), for each node set in $L$, we view it as the first layer of the potential horizontal region and store it in a local variable *min-layer*. We proceed to check whether there exists a node set $S$ so that the maximal layer among all the found layers can HR-satisfies $S$. If so, we add $S$ as the new maximal layer into the potential horizontal region. We keep searching for layers downward until no more qualified layer can be found. At the last, if a sequence of layers are found where any higher layer HR-satisfies its next lower layer, these layers form a horizontal region and we insert this horizontal region into *HR-set*. It could be realized easily to store the node sets in $L$ in an order so that a higher layer must be located before all the lower layers of it. As a result, no re-visit to the same horizontal region is guaranteed.

Once no regular region could be found, any found region must be an irregular region. Function Search-Min-IRR($G_i$), which is shown in Algorithm 6, is used to identify the minimal irregular region. For each pair of node sets $s$ and $s'$ ($s \in L$, $s' \in L'$), we try to identify if there is a region $r$ with $s$ and $s'$ as $min(r)$ and $max(r)$, respectively. In specific, we gradually introduce nodes between $s$ and $s'$ and stop immediately when some node

make it impossible to find a region between $s$ and $s'$. As a consequence, we can find out all irregular regions in $G_i$. Finally, we pick up the minimal one among these irregular regions as the final result. Because order of node sets in $L$ and $L'$ is fixed, we can see that `PO-Reduce` is a deterministic algorithm.

**Theorem 1.** `PO-Reduce` *can complete a reduction of a given partial order* $G_i$ *in the cost of* $O(|V_i|^2 \times |E_i|)$.

As mentioned, Partition($G_i$) is in the cost of $O(|E_i|)$. In Search-VR($G_i$), each edge is visited at most once. Therefore, Search-VR($G_i$) is in the cost of $O(|E_i|)$. In Search-HR, each edge is visited at most twice during the identification of the next layer. Therefore, Search-VR($G_i$) is also in the cost of $O(|E_i|)$. In Search-Min-IRR($G_i$), for each pair $(s, s')$, we need to check whether some node $n$ between them make it impossible to find a region with $s$ and $s'$ being the minimal and maximal node set, respectively. Because this kind of checking is in the cost of $O(|E_i|)$ and the number of such checking is in the level of $O(|V_i|^2)$, Search-Min-IRR($G_i$) is in the cost of $O(|V_i|^2 \times |E_i|)$.

Because all of Partition($G_i$), Search-VR($G_i$) and Search-HR($G_i$) are just in the cost of $O(|E_i|)$, while Search-Min-IRR($G_i$) is in the cost of $O(|V_i|^2 \times |E_i|)$, `PO-Reduce` is in the cost of $O(|V_i|^2 \times |E_i|)$.

## 4.4 Encoding Scheme

In this section, we describe how the nodes in a partial order are encoded using our nested encoding scheme. Consider a node $v_0$ in an input partial order $G_0$, where the reduction sequence of $G_0$ is $G_0 \to G_1 \cdots \to G_{n-1} \to G_n$, $n \geq 0$; and $v_0$ is contained in $k_0$ virtual nodes, $k_0 \in [0, n]$. Let $v_0 \overset{R_0}{\to} v_1 \cdots v_{k_0-1} \overset{R_{k_0-1}}{\to} v_{k_0}$ denote the containment sequence of $v_0$ computed by algorithm `PO-Reduce`. Note that each $v_i$ in the containment sequence is associated with a region: for $i \in [0, k_0)$, $v_i$ is associated with $R_i$; and the last node $v_{k_0}$ is

associated with the total order $G_n$. For notational convenience, we use $R_{k_0}$ to denote $G_n$.

In our nested encoding scheme, the encoding of each node $v_0$ (w.r.t. $G_0$), denoted by $\mathcal{N}(v_0)$, is defined by a sequence of $k_0+1$ segments: $< \mathcal{R}(v_{k_0}, R_{k_0}), \mathcal{R}(v_{k_0-1}, R_{k_0-1}), \cdots, \mathcal{R}(v_0, R_0) >$, where each segment $\mathcal{R}(v_i, R_i)$ represents the *region encoding* of $v_i$ w.r.t. the region $R_i$. In the following, we present the details of the region encoding for the three types of regions (i.e., vertical, horizontal, and irregular).

**Vertical Region Encoding.** Suppose $R_i$ is a vertical region consisting of $c$ chains, where the longest chain has $p$ nodes. Without loss of generality, we number the chains in $R_i$ from left to right by $0, \cdots, c-1$; and number the positions of the nodes within a chain from top to bottom by $0, 1$, etc. $\mathcal{R}(v_i, R_i)$ is defined to be a pair of natural numbers <*X-num,Y-num*>, where *X-num* represents the chain number that contains $v_i$ and *Y-num* represents the position of $v_i$ on that chain. $\mathcal{R}(v_i, R_i)$ can be represented by a bitstring of size $\lceil log_2(c) \rceil + \lceil log_2(p) \rceil$ bits.

**Horizontal Region Encoding.** Suppose $R_i$ is an $\ell$-level horizontal region. If $v_i$ is a level-$j$ node in $R_i$, $j \in [0, \ell-1]$, then for the purpose of dominance comparison it is sufficient to represent the node $v_i$ in $R_i$ by the value $j$. To facilitate efficient decoding, we design the format for horizontal region encoding to be the same as that for vertical region encoding with a pair of natural numbers <*X-num,Y-num*>, where *X-num* and *Y-num* are set to be 0 and $j$, respectively. $\mathcal{R}(v_i, R_i)$ can be represented by a bitstring of size $1 + \lceil log_2(\ell) \rceil$ bits.

**Irregular Region Encoding.** Suppose $R_i$ is an irregular region. In contrast to regular regions which can be encoded compactly, there is no universal "optimal" encoding for irregular regions. In this thesis, we use the bitvector scheme called *Compact Hierarchical Encoding* [9] to encode $R_i$; this scheme supports compact encoding of partial orders and efficient dominance comparison between values in partial orders. Each node $v_x$ in $R_i$ is encoded by a fixed-length bitstring of length $m$, denoted by $b_x[1, \cdots, m]$, with the

interpretation that a 0 bit dominates a 1 bit. Thus, for every pair of distinct nodes $v_x$ and $v_y$ in $R_i$, $v_x$ dominates $v_y$ iff (1) there exists at least one bit position $j$ such that $b_x[j] = 0$ and $b_y[j] = 1$, and (2) whenever $b_x[j] = 1$, $b_y[j] = 1$. Note that the size of the bitstring, $m$, is dependent on the complexity of the irregular region and the bitvector encoding algorithm.

As an example of regular region encoding, consider the value $v_9$ in Figure 4.1 and let $R_0 = \{v_6, v_7, v_8, v_9\}$, $R_1 = \{v_3, v'_1, v_4, v_5, v_{10}, v_{11}, v'_2, v_{12}, v_{17}, v'_3, v_{18}, v_{19}\}$ and $R_2 = G_2$. The containment sequence of $v_9$ is $v_9 \xrightarrow{R_0} v'_1 \xrightarrow{R_1} v'_4$, and $\mathcal{N}(v_9)$ is $< \mathcal{R}(v'_4, R_2), \mathcal{R}(v'_1, R_1), \mathcal{R}(v_9, R_0) >$; i.e., $<< 0,01 >, < 0,01 >, < 1,1 >>$. Similarly, the containment sequence of $v_5$ is $v_5 \xrightarrow{R_1} v'_4$, and $\mathcal{N}(v_5)$ is $< \mathcal{R}(v'_4, R_2), \mathcal{R}(v_5, R_1) >$; i.e., $<< 0,01 >, < 0,11 >>$. As an example of irregular region encoding, consider the value $v_{21}$ and let $R_4 = \{v_{20}, v_{21}, v_{22}, v_{23}\}$ which is an irregular region. The containment sequence of $v_{21}$ is $v_{21} \xrightarrow{R_4} v'_3 \xrightarrow{R_1} v'_4$, and $\mathcal{N}(v_{21})$ is $< \mathcal{R}(v'_4, R_2), \mathcal{R}(v'_3, R_1), \mathcal{R}(v_{21}, R_4)$; i.e., $<< 0,01 >, < 10,01 >, < 0,011 >>$.

Having defined the three different region encodings, we now need to explain how $\mathcal{N}(.)$ can be mapped into a fixed-length bitstring for efficient decoding when used in Z-order indexing. This requires three refinements to $\mathcal{N}(.)$. First, we need to encode each node in $G_0$ with a fixed number of segments. Thus, $\mathcal{N}(.)$ is extended to consist of a fixed number of $k_{max} + 1$ segments, where $k_{max}$ is the maximum depth of all nodes in $G_0$. In the event that a node $v$ has a depth of $k < k_{max}$, we append $k_{max} - k$ additional dummy segments to $\mathcal{N}(v)$ that are filled with 0 bits. Second, for each segment, the size of its bitstring representation should be fixed for all nodes being encoded; i.e., if the longest $x^{th}$ segment encoding is represented by $w$ bits, then all $x^{th}$ segments should be encoded with $w$ bits by padding additional 0 bits. Third, in order to distinguish between the different region encodings, we need to prepend each segment with a single header bit; specifically, a header bit value of 0 (resp., 1) indicates that the following segment is a regular (resp., irregular) region encoding. Note that a single header bit suffices for

Table 4.1: Examples for $\mathcal{N}(v)$

| $v$ | $Segment_1$ | $Segment_2$ | $Segment_3$ |
|---|---|---|---|
| $v_5$ | $0, < 0, 01 >$ | $0, < 00, 11 >$ | $0, < 0, 000 >$ |
| $v_9$ | $0, < 0, 01 >$ | $0, < 00, 01 >$ | $0, < 1, 001 >$ |
| $v_{15}$ | $0, < 0, 01 >$ | $0, < 01, 10 >$ | $0, < 0, 000 >$ |
| $v_{21}$ | $0, < 0, 01 >$ | $0, < 10, 01 >$ | $1, < 0, 011 >$ |

the three region encodings since the two regular region encodings are designed with the same pairwise dominance comparisons.

For convenience, we denote the fixed-length nested encoding of a partially ordered domain value $v$ by $\mathcal{N}(v)$. Once each partially ordered domain value of all data points has been mapped using `NE`, each data point is represented by a fixed-length bitstring which can be indexed using `ZB-tree`.

Table 4.1 illustrates some examples of $\mathcal{N}(v)$ for the partially ordered domain in Figure 4.1(a). Consider $v_5$. Although its depth is only 1, because $k_{max}$ is 2, we have to extend $\mathcal{N}(v_5)$ to 3 segments by appending one dummy segment filled with 0 bits. For $v_9$, $v_{15}$ and $v_{21}$, the depth of each of these values is 2. *Y-num* of the third segment of $v_9$, $v_{15}$ is represented by only 1 bit, while *Y-num* of the third segment of $v_{21}$ is represented by 3-bits. As a result, we must pad two 0 bits to the *Y-num* of the third segment of $v_9$ and $v_{15}$. As shown in Table 4.1, within each segment of an $\mathcal{N}(v)$, the first bit is the header bit indicating whether the segment is regular or irregular. We can see that only the third segment of $\mathcal{N}(v_{21})$ corresponds to an irregular region.

**Dominance Comparisons.** The dominance comparison between two nested encodings $\mathcal{N}(v_i)$ and $\mathcal{N}(v_j)$ is performed a segment at a time starting with the first segment. A segment comparison is said to be inconclusive if (1) the segment values are equal and (2) the segment is not the last segment; otherwise, we say that the segment comparison is conclusive (i.e., the encoded values are comparable or incomparable). If a segment

comparison is conclusive, the dominance comparison terminates; otherwise, if a segment comparison is inconclusive, the comparison proceeds to the next segment and so on until a conclusive comparison is reached.

Clearly, if one segment is regular and the other segment is irregular, then the encoded values are considered incomparable. Given two regular segments, if their X-num values differ, than they are considered incomparable; otherwise, if they have the same X-num values, then we need to also compare their Y-num values to decide whether the segment comparison is comparable (i.e. conclusive) or inconclusive.

As an example, consider the dominance comparison between the encoded values of $v_5$ and $v_9$ shown in Figure 4.1. We begin by comparing their first segments which are both regular. Since their X-num values are equal (i.e., 0), we proceed to compare their Y-num values which are also the same. We conclude that these two values are contained in the same virtual node regarding the first segment. Thus, the first segment comparison is inconclusive and we proceed to compare their second segments which are again both regular segments. Since their X-num values are the same, we compare their Y-num values. Here, the smaller Y-num of $v_9$ (relative to that of $v_5$) indicates that $v_9$ dominates $v_5$ and the segment comparison is conclusive and the dominance comparison terminates.

## 4.5 ZB-tree Variants

In this chapter, we provide details of the two basic variants of `ZB-tree`, namely, `CHE+ZB` and `TSS+ZB`, that we have developed to handle partially ordered domains. These two variants will be taken as competitors to `ZINC` method in experiments.

### 4.5.1 TSS+ZB

The TSS+ZB combines the TSS encoding scheme with the ZB-tree as follows. For each data point, we interleave binary expression of its values in totally ordered domains and topological numbers for its values in partially ordered domains into its Z-address which is a fixed-length bitstring. The reason for taking topological numbers for partially ordered domain values into account while encoding is to ensure the monotonicity property among data points indexed by ZB-tree. We take Z-addresses of data points as keys while constructing ZB-tree for a dataset. In a leaf entry, we store Z-address of the corresponding data point as well as the interval set for each partially ordered domain value because interval sets of partially ordered domain values are exactly where the dominance relationship among partially ordered domain values are encoded. In a dominance comparison between two data points, containment test between interval sets for values of the points in each dimension is the really crucial part. In an internal entry, we store *minpt* and *maxpt* of corresponding RZ-region as done in ZB-tree method and also store a merged union of the interval sets of all covered data points. During skyline query processing, we maintain an intermediate set of skyline points. For fairness, we also apply region-based dominance tests to TSS+ZB, which is enabled by the interval sets stored in internal entries. In specific, if Z-address of an intermediate skyline point $p_i$ can dominate *minpt* of an internal entry $e_j$ and interval set of $p_i$ subsumes the interval set of $e_j$ w.r.t. every partially ordered dimension, then the region represented by $e_j$ is dominated by $p_i$ and could be pruned immediately and safely.

### 4.5.2 CHE+ZB

The CHE+ZB is based on using the *Compact Hierarchical Encoding* [9] to encode partially ordered domain values. The main idea of this encoding method is to assign each node of a partial order with a reasonable label uniquely (called *gene*) following some

rules. As a result, each node can obtain a gene set which is the union of genes assigned to its ancestor nodes. With an implicit order on all the genes, the encoding of each node is a bitstring, each bit of which is set to 1 (resp., 0) based on the existence (resp., non-existence) of the corresponding gene in its gene set. For two nodes $v_1$ and $v_2$, if for any bit where encoding of $v_2$ is 1, the encoding of $v_1$ is 1 and there exists at least one bit where encoding of $v_1$ is 1 and encoding of $v_2$ is 0, then $v1$ is dominated by $v_2$. The *Compact Hierarchical Encoding*, which can precisely encode any partial order, owns a reasonable time complexity.

The encoding method consists of two main parts: *lattice completion* and *encoding algorithm*.

A lattice is a hierarchy where each pair of nodes has a unique smallest common ancestor and a unique greatest common descendant. The key idea in lattice completion is to complete the hierarchy into a full lattice by adding missing *intersection nodes*.

For instance, an hierarchy is given in Figure 4.3, which is not a lattice since for a pair of nodes *TA* and *FVS*, their smallest ancestor is not unique. This hierarchy could be completed into a full lattice by adding a new node *student&employee* as shown in Figure 4.4.

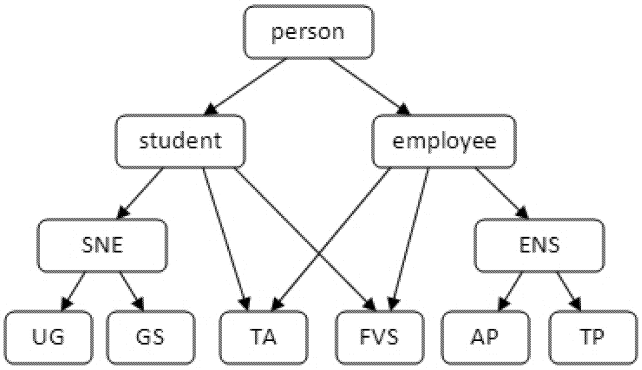



Figure 4.3: The original hierarchy.

The encoding algorithm associates genes to certain nodes of the lattice obtained in lattice completion and computes the code as the union of all genes of a node's ancestors.

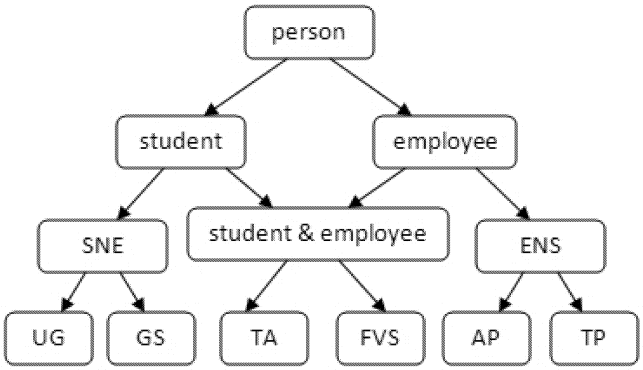



Figure 4.4: The completed lattice.

In particular, we assume the set of genes is $G = \{g_1, g_2, ..., g_n\}$ with an implicit order $g_i$ is in advance of $g_{i+1}$ $(i = 1, 2, ..., n-1)$. Each code is a member of $P(G)$ which is the powerset of $G$. Three functions are computed for each node: (1) The gene function $g$, which associates a gene to each primary node. A *primary node* is a node with a unique parent. (2) The encoding function $\gamma$, is defined as $\gamma(x) = \bigcup\{g(y)|y \in ancestor(x)\}$. (3) The anti-coding function $\nu$, such that $\nu(x)$ is the union of all genes that should not be chosen for any new child of $x$. $\nu(x)$ is also a code.

This algorithm works in an incremental and top-down manner. Encoding a node is different for primary nodes and others: (1) If the new node $x$ is a primary node with parent $y$, we compute $\nu(y)$ by taking all genes of all (a) descendants of $y$ and of (b) children of ancestors of $y$ that are not ancestors of $y$. We then pick the first gene in $G - \{\gamma(y) \cup \nu(y)\}$, using the implicit order we mentioned before. (2) If the new node $x$ has the set of parent $\{y_1, y_2, ..., y_p\}$, the algorithm proceeds with 2 steps. We first look for a *conflict* caused by the introduction of $x$, which is a pair $(y_i, y_j)$ such that $\gamma(y_i) \cap \nu(y_j) \neq \emptyset$. For each such conflict, we identify each ancestor of $y_i$ with gene $g_k$ responsible for the conflict, i.e., $g_k \in \nu(y_j)$ and we *mutate* the ancestor (change his gene to a safer gene). When all mutation are done, we simply compute the code $\gamma(x)$ by taking union of genes of ancestor of $x$. A non-primary node has no personal gene. Figure 4.5 and Table 4.2 display genes and codes for nodes in the partial order shown in Figure 4.4, respectively.

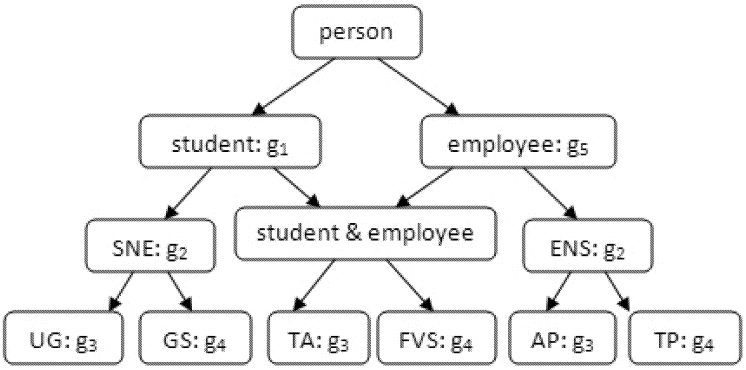
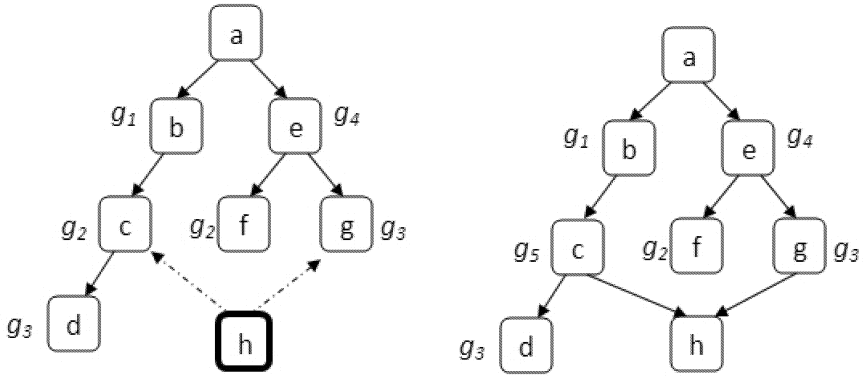



Figure 4.5: Genes for nodes in the lattice.

Table 4.2: Bitvectors for nodes in the partial order.

| $x$ | $\gamma(x)$ |
|---|---|
| person | 00000 |
| student | 10000 |
| SNE | 11000 |
| UG | 11100 |
| GS | 11010 |
| employee | 00001 |
| student & employee | 10001 |
| TA | 10101 |
| FVS | 10011 |
| ENS | 01001 |
| AP | 01101 |
| TP | 01011 |

After the example without involving any conflict, we give an example illustrating how mutation work to tackle conflicts. In Figure 4.6(a), assume we add a node $h$ into the partial order which provokes a conflict between the pair of nodes $(c, g)$. $(\gamma(c) \cap \nu(g) = \{g_2\} \neq \emptyset)$ To tackle this conflict, the gene $g_2$ of $c$ is mutated into $g_5$ and finally produces the coding shown in Figure 4.6(b).




(a) Before mutation (b) After mutation

Figure 4.6: A mutation example

The encoding algorithm is *polynomial* in time, and has been proven to be efficient enough to be used at run-time in building dynamic hierarchies. Although the encoding is still a complex operation in worst time, most of the encoding is actually straightforward. Since the lattice completion algorithm is also *polynomial*, the compact hierarchical encoding gives us a practical tool for encoding any partial order with bitvectors.

Space complexity is more complicated than time complexity. While no non-primary exist in the partial order, the length of a bitvector for one node is guaranteed to be no longer than the length of the maximal anti-chain in the partial order. In the general case, adding non-primary nodes may or may not require new genes (mutations). For instance, the example in Figure 4.5 uses only 5 genes, whereas its the length of its maximal anti-chain is 6. Nevertheless, it is easy to build a lattice with many intersection nodes that would cause a large number of mutations and thus consumes more genes than the length

of maximal anti-chain. The experience is that for practical hierarchies, the upper bound obtained for the hierarchies without non-primary node is still valid.

Readers can refer to [9] for more details.

## 4.6 Metric for Index Clustering

In this section, we present the metric for index clustering. Given an index $I$, let $D_I =< p_1, p_2, \cdots, p_n >$ denote the sequence of data points stored in the leaf level of $I$. We define the clustering of $I$, denoted by $clustering(D_I)$, to be the average "distance" of each pair of consecutive data points $p_j$ and $p_{j+1}$, denoted by $Dist(p_j, p_{j+1})$, in $D_I$. Here, the intuition is that two consecutive data points $p_j$ and $p_{j+1}$ in $D_I$ that are closer in the attribute value space should have a smaller distance value $Dist(p_j, p_{j+1})$; and an index method $I$ with a smaller value of $clustering(D_I)$ is considered to be more effective in clustering the data points and hence more effective in pruning index nodes to be traversed.

Following [44, 19, 17], given two $m$-dimensional data points $p$ and $p'$ (with attributes $A_1$,$\cdots$,$A_m$), the distance between $p$ and $p'$ is defined based on L2 norm distance function to be the square root of the sum of the squares of the normalized distance between p and p' (denoted by $NDist()$) in each dimension, i.e., $Dist(p, p') = (\sum_{i=1}^{m} (NDist(p.A_i, p'.A_i))^2)^{1/2}$. For two totally ordered domain values $v$ and $v'$, $NDist(v, v') = \frac{|v - v'|}{v_{max} - v_{min}}$, where $v_{max}$ and $v_{min}$ denote the maximum and minimum values for that domain.

For two partially ordered domain values $v$ and $v'$ in a partial order $G$, our normalized distance metric is defined in terms of two cases. Let $maxDist(G)$ denote the edge length of the longest chain in $G$. Consider the first case where $v$ and $v'$ are along the same chain in $G$. Let $L(v, v')$ denote the distance of $v$ and $v'$ along that chain (in terms of number of

edges). Thus, $NDist(v, v') = \dfrac{L(v, v')}{maxDist(G)}$. Consider the second case where $v$ and $v'$ are not along the same chain in $G$. Let $v_a$ be common ancestor value of $v$ and $v'$ in $G$. We define $GDist(v, v', v_a) = \max(L(v, v_a), L(v', v_a))+ \min(L(v, v_a), L(v', v_a)) \times maxDist(G)$. The intuition here is that the distance of two partially ordered domain values along the same chain are considered to be closer than two partially ordered domain values that are on different chains. Therefore, $NDist(v, v')$ is defined to be minimum of $\dfrac{GDist(v, v', v_a)}{2 \times maxDist(G)}$ over all common ancestor values $v_a$ of $v$ and $v'$.

# Chapter 5

# Performance Study

To evaluate the performance of our proposed ZINC, we conducted an extensive set of experiments to compare ZINC against TSS, TSS+ZB and CHE+ZB. Our experimental results show that ZINC outperforms the other three competing methods. Given that: (1) both TSS+ZB and CHE+ZB are also based on ZB-tree; (2) ZINC does not use more memory in processing compared with other methods, the superior performance of ZINC demonstrates the effectiveness of our proposed NE encoding for PO domains.

**Synthetic datasets:** In our experiments, we generated three types of synthetic datasets according to the methodology in [42]. For TO domains, we used the same data generator as [42] to generate synthetic datasets with different distributions. For PO domains, we generated DAGs by varying three parameters to control their size and complexity: *height* ($h$), *node density* ($nd$), and *edge density* ($ed$)[1], where $h \in Z^+$, $nd, ed \in [0, 1]$. Each value of a PO domain corresponds to a node in DAG and the dominating relationship between two values is determined by the existence of a directed path between them. Given $h$, $nd$, and $ed$, a DAG is generated as follows. First, a DAG is constructed to represent a poset for the powerset of a set of $h$ elements ordered by subset

[1]In contrast to [42], which uses only the $h$ and $nd$ parameters, the additional $ed$ parameter that we introduced enables a more fine-grained control over the complexity of the DAGS.

Table 5.1: Parameters of Synthetic Datasets

| Parameters | Values |
|---|---|
| $\|PO\|$: no of PO domains | 3, 1, 2 |
| $\|TO\|$: no of TO domains | 1, 2, 3, 4 |
| $h$: DAG height | 6, 2, 4, 8, 10 |
| $nd$: DAG node density | 0.4, 0.2, 0.6, 0.8, 1.0 |
| $ed$: DAG edge density | 0.6, 0.2, 0.4, 0.8, 1.0 |
| $\|D\|$: size of dataset | 500K, 100K, 1M, 3M, 5M |
| Correlation | independent, anti-correlated, correlated |

containment; thus, the DAG has $2^h$ nodes. Next, $(1 - nd) \times 100\%$ of the nodes (along with incident edges) are randomly removed from the DAG, followed by randomly removing $(1 - ed) \times 100\%$ of the remaining edges such that the resultant DAG is a single connected component with a height of $h$. Following the approach in [42], all the PO domains for a dataset are based on the same DAG. Table 5.1 shows the parameters and their values used for generating the synthetic datasets, where the first value shown for each parameter is its default value. In this section, default parameter values are used unless stated otherwise.

**Real dataset:**Our real dataset on movie ratings is derived from two data sources, *Netflix*[2] and *MovieLens* [3]. Netflix contains more than 100 million movie ratings submitted by more than 480 thousand users on 17770 movies between December 31st, 1999 and December 31st, 2005. MovieLens contains more than 1 million ratings submitted by more than 6040 users on 3900 movies. Both these data sources use the same rating scale from 0 to 5 with a higher rating indicating a more preferred movie. Our dataset consists of the ratings for 3098 of the movies that are common to both data sources.

We derived a PO attribute, named *movie preference*, for the 3098 movies as follows: a movie $m_i$ dominates another movie $m_j$ iff the average rating of $m_i$ in one data source

[2]http://www.netflix.com
[3]http://www.grouplens.org

is higher than that of $m_j$, and the average rating of $m_i$ in the other data source is at least as high as that of $m_j$. We also derived two TO attributes for each movie, named *average rating* and *number of reviewers*, which represent, respectively, the movie's average rating (each value is between 0.00 and 5.00) and total number of ratings that it has received over the two data sources. The number of distinct values for these two TO domains are 501 and 219800, respectively. For each of the TO domains, a higher attribute value is preferred.

**Platform and settings:** All the algorithms were implemented in C++ and compiled with GCC. The index/data page size was set to be 4K byte for all the algorithms. Our experiments were carried out on a Pentium IV PC with 2.66GHz processor and 4GB main memory running on Linux operating system. Each reported timing measurement is an average of five runs with cold cache.

In the rest of this section, we first present the results for synthetic datasets (Figs. 5.1(a) to 5.2(c)) followed by the results for real datasets (Fig. 5.2(e) to Fig. 5.2(h)).

## 5.1 Effect of PO Structure

Figs. 5.1(a), 5.1(b), and 5.1(c) compare the effect of the PO structure on the total processing time (including both CPU and I/O) to compute skylines as each of the three parameters (DAG height, node density, edge density) is varied. Note that the complexity of the DAGs increases as each parameter value becomes larger. In the following, we shall focus our discussion on Fig. 5.1(a) (the y-axis shown is in logarithm scale), where the height parameter is being varied. The properties of the generated PO domains are shown in the first three columns of Table 5.2, where *Card* represents the domain cardinality and *Depth* represents the maximum node depth in the DAG; the sizes of constructed indexes for the four approaches (for 500K dataset) are shown in the last four columns.

Table 5.2: Features of each PO domain and sizes of indexes

| $h$ | Card | Depth | Size of Index (MB) | | | |
|---|---|---|---|---|---|---|
| | | | ZINC | CHE+ZB | TSS | TSS+ZB |
| 2 | 3 | 0 | 7.38 | 5.96 | 14.32 | 8.05 |
| 4 | 6 | 1 | 15.07 | 5.90 | 29.02 | 21.08 |
| 6 | 29 | 3 | 29.54 | 12.04 | 50.71 | 40.69 |
| 8 | 112 | 6 | 60.59 | 40.28 | 113.10 | 97.20 |
| 10 | 456 | 7 | 67.57 | 103.32 | 151.25 | 124.17 |

For simple partial orders (i.e, height = $2, 4, 6$ in Fig. 5.1(a)), the number of returned skyline points are 102, 8, and 267, respectively. The performance of all four methods for these three cases are I/O bound with at least 63% of the processing time spent on I/O. While CHE+ZB, TSS, and TSS+ZB have comparable performance, ZINC outperforms all these three methods. ZINC was able to more effectively prune away many unnecessary subtree traversals and visited only a small portion of the index nodes; specifically, only 29% (532 out of 1846), 18% (678 out of 3768), and 24% (1778 out of 7386) of the distinct index nodes of ZINC were visited corresponding to height of $2, 4$, and 6, respectively. In contrast, CHE+ZB visited 78% (1158 out of 1491), 73% (1085 out of 1476), and 82% (2456 out of 3010) distinct index nodes; TSS visited 15% (537 out of 3580), 21% (1524 out of 7255), and 69% (8748 out of 12678) of distinct index nodes; and TSS+ZB visited 30% (604 out of 2012), 19% (1001 out of 5270), and 31% (3153 out of 10172) nodes, respectively, for these three cases.

For complex partial orders (i.e., height = 8, 10 in Fig. 5.1(a)), the performance of all four methods become CPU bound with at least 83% of the total time spent on CPU processing. This is because the complex partial orders result in much more skyline points and dominance comparisons. For example, when the height is 8, there are 112 values in the PO domain and a total of 20493 skyline points. Observe that ZINC continues to outperform the other methods significantly. For CHE+ZB, it requires a bitstring size

of 58 bits to encode each PO domain value, and CHE+ZB actually visited all the index nodes for the skyline computation. Thus, we see that the data points in CHE+ZB are not well clustered resulting in ineffective region-based pruning for its index traversals.

In contrast, due to the effectiveness of NE, ZINC visited only 27% of its index nodes. Consequently, the number of pairwise dominance comparisons in CHE+ZB is about 10 times more than that in ZINC ($9.1\times10^8$ vs $9.2\times10^7$), and about 3 times more than that in TSS ($9.1\times10^8$ vs $2.8\times10^8$). Like CHE+ZB, TSS also visited all its index nodes. Observe that the performance of TSS and TSS+ZB degrades significantly as the complexity of the partial orders increases. The reason is because each pairwise dominance comparison in TSS and TSS+ZB involves not only dominance comparison between two bitstrings but also containment checking between the corresponding two interval sets. The average number of intervals in each interval set are 4 and 5, respectively, for height values of 8 and 10. Consequently, the cost of pairwise dominance comparisons in TSS and TSS+ZB is significantly higher than that of the other algorithms. Finally, with respect to the total processing time, ZINC outperforms CHE+ZB, TSS and TSS+ZB by up to a factor of about 9, 14.5 and 13 times, respectively.

Similarly, for the results corresponding to varying node density and edge density as shown in Figs. 5.1(b) and 5.1(c), respectively, ZINC outperforms all of CHE+ZB, TSS, and TSS+ZB.

## 5.2 Effect of Data Cardinality

Fig. 5.1(d) compares the performance of the algorithms as a function of data cardinality. The number of skyline points for data cardinality values of 100K, 500K, 1M, 3M, and 5M, are 601, 267, 142, 1, and 1, respectively. The processing time decreases for all the methods when the cardinality increases from 1M to 3M; this is due to the fact that there is only one skyline point when the cardinality is 3M, resulting in very effective

index traversal pruning. However, when cardinality increases further from 3M to 5M, although the number of skyline points remains unchanged (with only one point), there is an increase in the number of dominance comparisons and visited index nodes due to the larger data size which results in an increase in the processing time.

## 5.3 Effect of Data Distribution

Fig. 5.1(e) compares the performance for anti-correlated datasets. Again here, ZINC has the best performance. Observe the the performance of CHE+ZB, TSS, and TSS+ZB is satisfactory for simple partial orders, but not for complex partial orders. In particular, when height = 10 (which is not shown in Fig. 5.1(e)), each of CHE+ZB, TSS, and TSS+ZB took more than 3 hours to complete the skyline computation compared to ZINC which took 1.7 hours. The reason for this significant increase in running time is due to the large number of skyline points when the data is anti-correlated. Specifically, the number of skyline points in Fig. 5.1(e) corresponding to the five increasing height values are 200, 1780, 4917, 54926, and 286223.

Fig. 5.1(f) compares the performance for correlated datasets. From the experimental results shown in Fig. 5.1(e) and 5.1(f), we can see that the processing time becomes higher (resp., lower) while datasets are anti-correlated (resp., correlated). The reason is the number of skyline points becomes larger (resp., smaller) and more (resp., less) computations are incurred.

## 5.4 Progressiveness

This set of experiments investigate the progressiveness of the algorithms. For each algorithm, we record the time it requires to output specific percentages of the results (0% for the first returned result, 20%, 40%, 60%, 80% and 100%). In Fig. 5.2(a) we can see

that ZINC also outperforms the other methods in terms of progressiveness. While ZINC needs only 50% of total processing time to compute the first 80% of skyline points, TSS+ZB, CHE+ZB, and TSS require 55%, 64%, and 90% of the total time, respectively.

## 5.5 Effect of Dimensionality

Fig. 5.2(b) investigates the effect of the dataset dimensionality. Each pair of numbers $(t, p)$ along the x-axis represents the number of TO (t) and number of PO (p) domains in the datasets. As the number of skyline points increases with an increase in the data dimensionality, the processing time for all algorithms also increases. For a fixed number of dimensions, the processing time is larger when there are more PO domains, e.g., (2,2) vs (3,1), and (3,2) vs (4,1). The reason is that PO domains always have much more non-dominated values than TO domains. Again here, ZINC has the best performance.

## 5.6 Index Construction Time

Fig. 5.2(c) compares the index construction time as a function of the height parameter. Observe that the construction time for ZINC is slightly higher than that of TSS and TSS+ZB. Although ZINC incurs less I/O time than TSS and TSS+ZB for index construction, the nested encoding used by ZINC is more complex which increases the CPU time spent on encoding and computing node splits. CHE+ZB has the highest index construction time because the encodings produced by CHE+ZB are also the longest resulting in more costly comparisons and hence higher construction time; in particular, when height = 10, the maximum lengths of the encodings produced by TSS+ZB, ZINC, and CHE+ZB are 132, 352, and 848 bits, respectively.

## 5.7 Comparison of Index Clustering

In this section, we compare the clustering effectiveness of the four index methods. Figure 5.2(d) compares the clustering effectiveness of the four methods in terms of the *clustering*($D_I$) metric as a function of the height parameter. The y-axis shown is in logarithm scale. Thus, an index with a smaller y-axis value is considered to be more effective in clustering the data points. The results show that ZINC produces the best clustering. In particular, when height = 6, the clustering value of ZINC is just about 50%, 62%, and 66% of CHE+ZB, TSS, and TSS+ZB, respectively. When the partial orders become more complex (i.e., height is 8 or 10), the performance gain of ZINC reduces because a larger proportion of the partial orders are irregular regions which increase the the proportion of irregular region encoding.

## 5.8 Performance on Real Dataset

Fig. 5.2(e) compares the performance on the real dataset which contains 291 skyline points. The depth of the derived partial order domain is 9, and the ratio of the size of the regular region (in terms of the number of regular nodes) over the entire partial order domain size is 53%.[4] The results show that ZINC outperforms CHE+ZB, TSS, and TSS+ZB by a factor of 5.5, 15, and 13, respectively.

## 5.9 Additional Experiments on Netflix Dataset

In this section, we present additional experimental results on the Netflix real dataset to examine the effect of the regularity of the partial order domain as well as the effect of the number of partial order domains. We focus on the movies that are produced no later

[4] A node $v$ in a partial order $P$ is classified as an *irregular node* if the innermost region that contains $v$ in the PO reduction of $P$ is an irregular region; otherwise, $v$ is classified as a regular node.

than 2000 and have ratings for six years (for every year in the period between December 31st, 1999 and December 31st, 2005). The number of such movies is 10709, which is the cardinality of the derived PO domain.

### 5.9.1 Effect of Regularity of PO Domain

To vary the structure of the PO domain, we introduce a parameter $L \in \{4, 5, 6\}$ which represents the number of dimensions used to construct the PO domain. We expect the number of skyline points to increase with a larger value of $L$. For a given $L = l$, for each movie $m$, we calculate the yearly average rating of $m$ for the $l - 1$ years for which $m$ has the largest number of yearly reviews. Then, we calculate for each movie the average rating over all the remaining years. As a result, each movie has $l$ ratings. Using these $l$ ratings for each movie, a partial order domain is constructed based on the following dominance relationship: a movie $m_i$ dominates another movie $m_j$ iff (1) $m_i$ is no lower than $m_j$ in each of the $l$ ratings, and (2) $m_i$ is higher than $m_j$ in at least one rating.

We also derive two TO domains for each movie: the movie's *average rating* and the total *number of ratings* over all the six years. In both of these TO attributes, higher values are preferred.

Fig. 5.2(f) compares the performance as a function of parameter $L$. The number of skyline points are 1103, 2412, and 2783, respectively, for $L = 4$, $L = 5$, and $L = 6$. The respective depths of the PO domains are 13, 15, and 19; and their respective ratios of size of regular regions (in terms of the number of regular nodes) over the whole domain size are 51%, 46%, and 40%. Thus, the PO domains become less regular as $L$ increases.

The results show that `ZINC` outperforms the three competing methods in all cases. Observe that the performance decreases as a function of $L$ due to the increased number of skyline points. Moreover, as the PO domain becomes less regular with increasing $L$, the performance gain of `ZINC` over the competing methods also decreases. For example,

the performance gain of ZINC over TSS decreases from 20 to 5.5.

### 5.9.2 Effect of Number of PO Domains

In this experiment, we derive three PO domains from the six yearly movie ratings in the Netflix dataset. Each PO domain is constructed from two yearly ratings (i.e., 2000 and 2001, 2002 and 2003, and 2004 and 2005). For each partial order, a movie $m_i$ dominates another movie $m_j$ iff the yearly average rating of $m_i$ is higher than that of $m_j$ in one year and not lower than that of $m_j$ in the other year. As before, we also derive two TO domains; thus the derived dataset has three PO domains and two TO domains. The average ratio of the regular regions over these three PO domains is up to 65%, and there are a total of 2572 skyline points. The performance results in Fig. 5.2(g) show that ZINC outperforms CHE+ZB, TSS, and TSS+ZB by a factor of 3.0, 7.2, and 6.3, respectively.

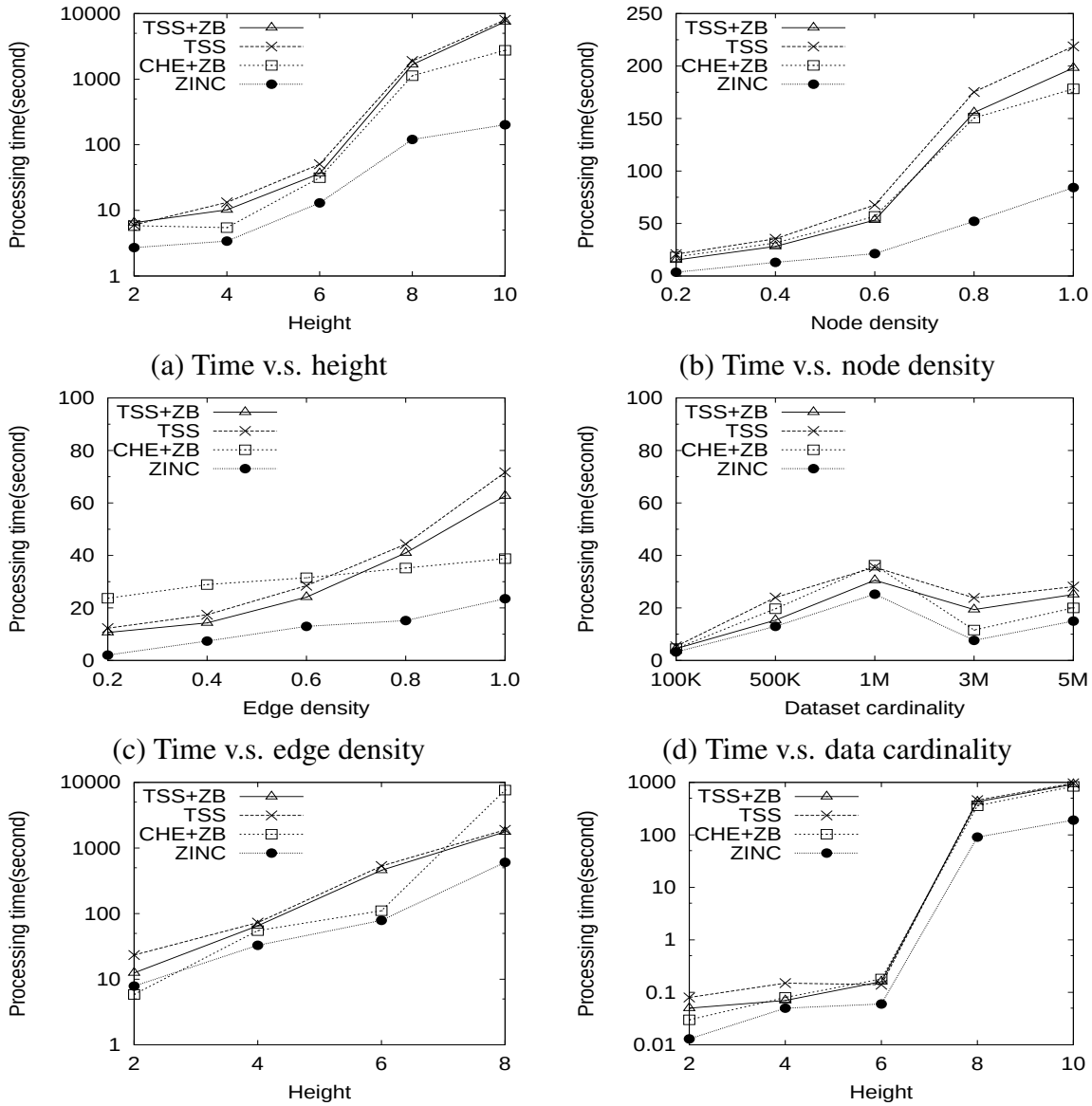
## 5.10 Experiments on Paintings Dataset

The last experiment on real dataset is based upon a smaller real dataset and a simple partial order derivation method. We use a real dataset, denoted by *paintings*, which contains information about more than 22,000 paintings collected from two art gallery websites[5]. Each painting record consists of one totally ordered attribute, *year*, and eight partially ordered attributes (e.g., *size*, *subject*, *main color*, *price*). The partially ordered domains are derived from a survey conducted by the Dia Art Foundation[6] on the preferences of artwork buyers from different countries. In our experiments, we used the preferences of buyers from the US. Here, we elaborate on how the partially ordered domains of our paintings real dataset are derived from the survey regarding

[5]http://artgallery.com.ua, http://www.gallery-worldwide.com
[6]http://awp.diaart.org/km/surveyresults.html

user preferences on painting purchases. Each question in the survey asks for the user preference on a painting-related topic. For instance, in one question, users are asked for their favorite season to be depicted in paintings, and the percentage breakdown for this question is as follows: fall (33%), spring (26%), summer (16%), and winter (15%). For each question, we map the answer values into a partial order based on a threshold value $\alpha$ as follows: if the percentages for two answer values differ by less than $\alpha$, then the two answer values will be treated as incomparable; otherwise, the answer value with a higher percentage dominates the other value. We set $\alpha$ to be 3%. Thus, for the attribute related to season preference, we have *fall* dominates *spring*, *spring* dominates both *summer* and *winter*, and both *summer* and *winter* are treated as incomparable. Based on this approach, we mapped eight questions in the survey into eight partially ordered domains. We believe that the described approach is a reasonable way to map user preferences in a survey to partially ordered domains. The partially ordered domains obtained are only of low or moderate complexity: their cardinalities range from 4 to 14 and the maximum node depth varies from 0 to 2. Correspondingly, the length of `NE` codes varies from 70 bits to 210 bits. In fact, we found that regular regions are very common in the partially ordered domains of this real dataset: the proportion of regular regions in each partially ordered domain is at least 80%.

Fig. 5.2(h) compares the performance for the paintings real dataset. As the partial orders for this dataset is not complex, the performance of all the methods are I/O-bound with at least 70% of the total processing time spent on I/O. There are a total of 2006 skyline points. Similar to the comparison trends for the synthetic datasets, we see that `ZINC` outperforms the three competing methods by at least a factor of 2. In particular, `ZINC` was able to effectively prune away 30% of the index node traversals; in contrast, each of the other methods visited more than 90% of the index nodes.




(a) Time v.s. height

(b) Time v.s. node density

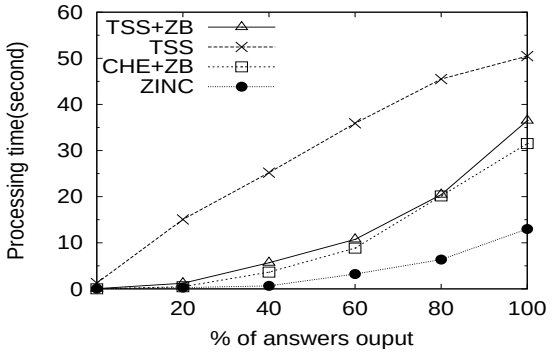
(c) Time v.s. edge density

(d) Time v.s. data cardinality

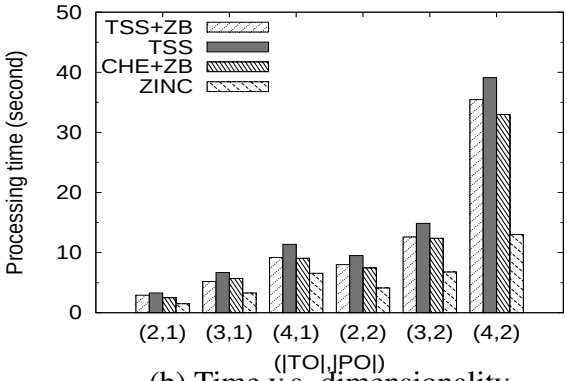
(e) Anti-correlated dataset

(f) Correlated dataset

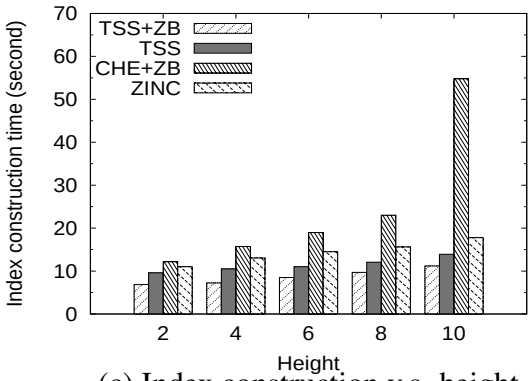
Figure 5.1: Experimental results



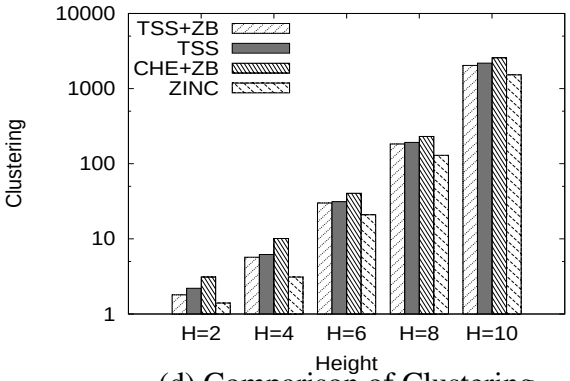

(a) Progressiveness



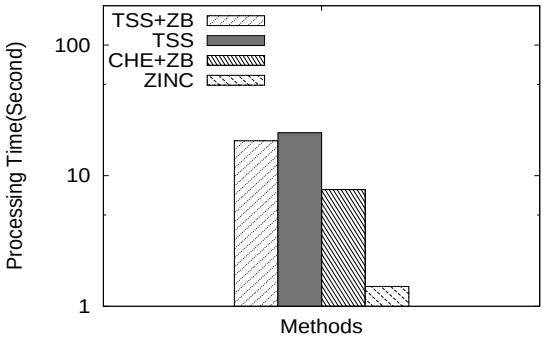

(b) Time v.s. dimensionality



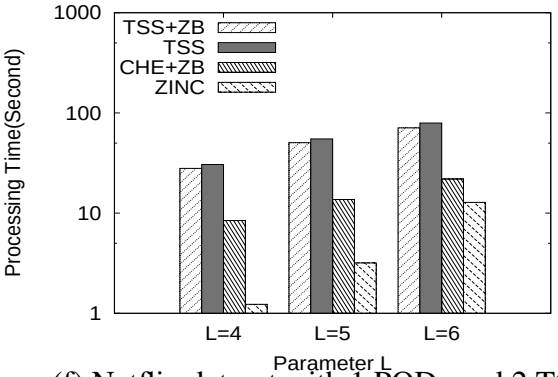

(c) Index construction v.s. height



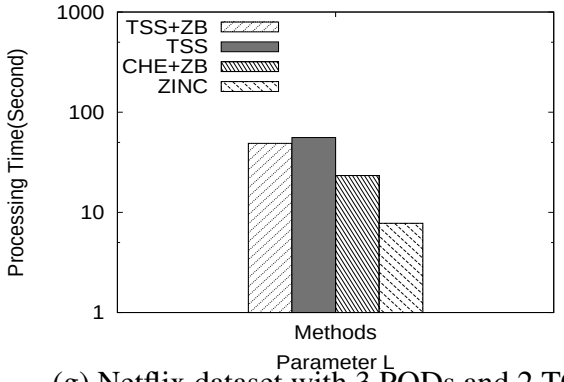

(d) Comparison of Clustering



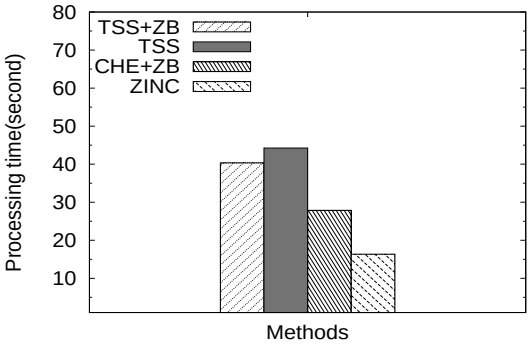

(e) Processing time on real dataset




(f) Netflix dataset with 1 PODs and 2 TODs




(g) Netflix dataset with 3 PODs and 2 TODs




(h) Paintings dataset

Figure 5.2: Experimental results continued

# Chapter 6

# Conclusions and Future Work

In this chapter, we state the conclusions of our existing work and then introduce some future work that might be interesting.

## 6.1 Conclusions

In this thesis, we have reviewed the existing work in the area of skyline queries processing. While most of effort is devoted to processing skyline queries with totally ordered domains only, increasing attention has been attracted by processing of skyline queries with both totally and partially ordered domains which is more general in practice. We also give a picture on lots of other related research areas. After going through these related work, we present the `ZB-tree` method in details which is the basis of our proposed `ZINC` method.

The key contribution of our proposed `ZINC` method is the efficient encoding scheme `NE` which encodes values in partial ordered domains into bitstrings compactly relying upon reduction of the corresponding partial orders. We also develop two variants of `ZB-tree` method which combine `ZB-tree` with `TSS` encoding scheme and another bitstring encoding scheme, respectively. We conduct an extensive set of experiments on

both synthetic and real datasets with various settings to compare ZINC with the existing state-of-the-art method TSS and the two variants of ZB-tree. By combining the strengths of NE and ZB-tree, ZINC achieves an outstanding performance to outperforms the existing state-of-the-art method TSS in processing skyline queries with both totally and partially ordered domains. From the superior performance of ZINC over CHE+ZB and TSS+ZB, we can see that the good effect of ZINC mainly depends on the efficiency of NE scheme.

## 6.2 Future Work

There are two interesting future work. The first one aims to efficiently process skyline queries with a more general case of preferences called *conditional preferences*. The other is about how to efficiently process a batch of skyline queries in parallel with common computation cost amortized.

### 6.2.1 Skyline Queries with Conditional Preferences

Extending handleable domains of skyline queries processing from totally ordered domains to the combination of totally and partially ordered domains is rather good but still not great enough. Within partial orders, conflicting dominance relationship is not allowed and preferences on different attributes are considered independent. These are not completely comply with the preferences met in daily life. A more general model of preferences is *Conditional Preferences* (CPs, for short) which have been studied in AI community. CPs take into account the dependency among difference attributes which is based on some assumptions. For instance, the statement ”I prefer red wine to white wine if meat is served.” asserts that, once meat served, a red wine is preferred to a white wine. Obviously, partial orders could be thought of as a special case of CPs because a

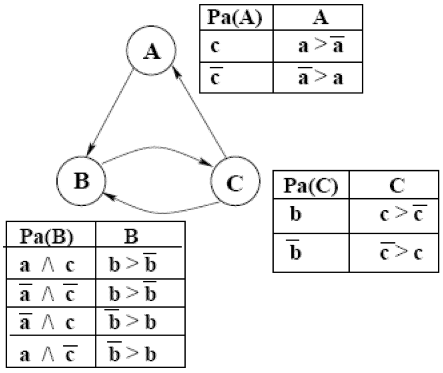



Figure 6.1: An Example for CP-net

value $v_1$ dominates another one $v_2$ in a partial order can be viewed as $v_1$ could dominate $v_2$ under combination any values of other attributes.

Some intuitive representation and rules for CPs are crucial for investigating skyline query with CPs. An elegant formalism to represent CPs is the CP-nets which are proposed and improved in [7, 5, 18]. For example, a cyclic CP-net and corresponding statement tables are shown in Figure 6.1 with three Boolean attributes $A$, $B$ and $C$. The first row in the table associated with $A$ means with presence of value $c$ for attribute $C$ value $a$ is preferred to value $\overline{a}$ and similarly, the second row means with presence of value $\overline{c}$ for attribute C value $\overline{a}$ is preferred to value $a$. Figure 6.2 shows the induced preference ordering of the given CP-net. From this graph of preference ordering we can see the value combination $abc$ is non-dominated.

For skyline queries with CPs, or additionally with some hard constraints, it is hard to efficiently capture all skyline points by using any existing method, e.g., Search-CP method [6], because it needs to recursively scan all possible candidates. Also, existing work may become disabled once corresponding CPs are dynamic.

In our future work, we plan to extend efficient skyline queries processing to a context involving CPs.

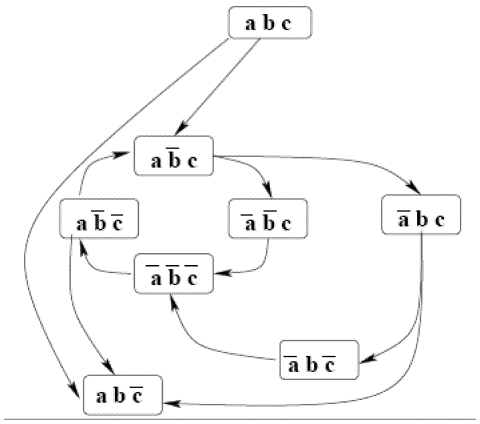



Figure 6.2: Induced Preference Ordering of the CP-net

### 6.2.2 Multiple Skyline Queries Processing

In some real application, such as a second-hand cars sale system and an air ticket booking system, more than one skyline query is simultaneously presented to the system in order to be processed. Taking the air ticket booking system for Air China [1] as an example, according to statistical data, the system receives about 460 thousands queries every day and in average, about 5 queries every second and probably more during peak time. Meanwhile, for a batch of skyline queries received at the same time, they may have difference preferences on some attributes, e.g., *airlines* and *flight models*. Thus, a new problem arises is that if we can process such batch of skyline queries efficiently by sharing common computation cost. We call this problem *Multiple Skyline Queries Optimization* (`MSQO`, for short).

Keep using the air ticket booking as an example. An air ticket booking website receives three simultaneous skyline queries from three users. Each user has her own particular skyline. All the users want to fly to Hong Kong. When the destination is Hong Kong, the recognized best airline choice is Cathay Pacific. Moreover, the first user prefers Singapore Airline to China Airline due to SA's outstanding service. The third

[1] http://www.airchina.com.cn

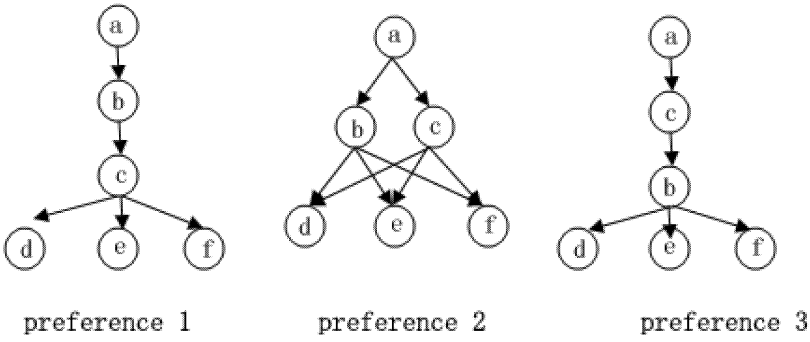



Figure 6.3: Graphic Representation of Preferences in an MSQO Problem

user prefers China Airline to Singapore Airline due to CA's attractive price. The second user is the least fastidious one, to whom both Singapore Airline and China Airline are acceptable. Furthermore, all the users can not endure a transit because a two-hour transit is so tiresome. As a result, the corresponding skyline dominance graphs are shown in Figure 6.3, where every node contains two value features. Meanings of the nodes are listed below:

| | |
|---|---|
| a: Cathay Pacific without transit | d: Cathay Pacific with transit |
| b: Singapore Airline without transit | e: Singapore Airline with transit |
| c: China Airline without transit | f: China Airline with transit |

Based on existing frameworks for processing skyline queries, system has to process the received queries sequentially so that a great amount of common computation will be performed repeatedly. As a result, much unnecessary computation is conducted and users' requirement on response time is hard to be satisfied. We aim to efficiently process a batch of skyline queries that are obtained within a tiny time interval in a real-time fashion. We are going to find out inner similarity among different preferences in real-time and share common computation during processing.

# Bibliography


[1] W. Balke, U. Guntzer, and C. Lofi. Eliciting matters controlling skyline sizes by incremental integration of user preferences. In *DASFFA*, pages 551–562, 2007.

[2] W. Balke, U. Guntzer, and W. Siberski. Exploiting indifference for customization of partial order skylines. In *IDEAS*, pages 80–88, 2006.

[3] I. Bartolini, P. Ciacia, and M. Patella. Efficient sort-based skyline evaluation. In *TODS*, volume 33(4), pages 1–49, 2008.

[4] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[5] C. Boutilier, R. I. Brafman, C. Domshlak, H. H. Hoos, and D. Poole. Cp-nets: A tool for representing and reasoning with conditional ceteris paribus preference statements. In *JAIR*, pages 135–191, 2004.

[6] C. Boutilier, R. I. Brafman, C. Domshlak, H. H. Hoos, and D. Poole. Preference-based constrained optimization with cp-nets. In *Computational Intelligence*, volume 20, pages 137–157, 2004.

[7] C. Boutilier, R. I. Brafman, H. H. Hoos, and D. Poole. Reasoning with conditional ceteris paribus preference statements. In *UAI*, pages 71–80, 1999.

[8] R. I. Brafman and C. Domshlak. Introducing variable importance tradeoffs into cp-nets. In *In Proceedings of UAI-02*, pages 69–76. Morgan Kaufmann, 2003.

[9] Y. Caseau. Efficient handling of multiple inheritance hierarchies. In *OOPSLA*, pages 271–287, 1993.

[10] C. Y. Chan, P. K. Eng, and K. L. Tan. Stratified computation of skylines with partially-ordered domains. In *SIGMOD*, pages 203–214, 2005.

[11] C. Y. Chan, H. V. Jagadish, K. L. Tan, A. K. H. Tung, and Z. Zhang. On high dimensional skylines. In *EDBT*, pages 478–495, 2006.

[12] S. Chaudhuri, N. Dalvi, and R. Kaushik. Robust cardinality and cost estimation for skyline operator. In *ICDE*, page 64, 2006.

[13] Jan Chomick. Iterative modification and incremental evaluation of preference queries. In *FoLKS*, pages 63–82, 2006.

[14] J. Chomicki. Preference queries. *CoRR*, cs.DB/0207093, 2002.

[15] J. Chomicki. Semantic optimization techniques for preference queries. *CoRR*, abs/cs/0510036, 2005.

[16] J. Chomicki. Database querying under changing preferences. *CoRR*, abs/cs/0607013, 2006.

[17] L. Cowen and C. Priebe. Randomized non-linear projections uncover high-dimensional structure. In *AAM*, pages 319–331, 1997.

[18] C. Domshlak and R. I. Brafman. Cp-nets: Reasoning and consistency testing. In *KR-02*, pages 121–132, 2002.

[19] H. L. Fei and J. Huan. L2 norm regularized feature kernel regression for graph data. In *CIKM*, pages 593–600, 2009.

[20] P. Godfrey. Skyline cardinality for relational processing. In *FoIKS*, pages 78–97, 2004.

[21] P. Godfrey, R. Shipley, and J. Gryz. Maximal vector computation in large data sets. In *VLDB*, pages 229–240, 2005.

[22] P. Godfrey, R. Shipley, and J. Gryz. Algorithms and analyses for maximal vector computation. In *VLDB J.*, volume 16(1), pages 5–28, 2007.

[23] B. Hafenrichter and W. Kießling. Optimization of relational preference queries. In *ADC*, pages 175–184, 2005.

[24] J.Chomicki. Querying with intrinsic preferences. In *ICEDT*, pages 34–51, 2002.

[25] J.Chomicki. Preference formulas in relational queries. In *TDS*, pages 427–466, 2003.

[26] J.Chomicki. Semantic optimization of preference queries. In *CBD*, pages 133–148, 2004.

[27] J.Chomicki, P.Godfrey, and J.Kryz. Skyline with presorting. In *ICDE*, pages 717–719, 2003.

[28] W. Kießling and B. Hafenrichter. Optimizing preference queries for personalized web service. In *IASTED*, pages 461–466, 2002.

[29] W. Kießling and B. Hafenrichter. Algebraic optimization of relational preference queries. In *Tecknique Report 2003-1. Institut für Informatik, Universität Ausberg*, 2003.

[30] W. Kießling and G. Köstler. Preference sql - design, implementation, experiences. In *VLDB*, pages 990–1001, 2002.

[31] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *VLDB*, pages 275–286, 2002.

[32] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. In *Jounrnal of the ACM*, pages 469–476, 1975.

[33] K. Lee, B. Zheng, H. Li, and W. C. Lee. Approaching the skyline in z order. In *VLDB*, pages 279–290, 2007.

[34] X. Lian and L. Chen. Monochromatic and bichromatic reverse skyline search over uncertain databases. In *SIGMOD*, pages 213–226, 2008.

[35] X. Lin, Y. Yuan, W. Wang, and H.Lu. Stabbing the sky: Efficient skyline computation over sliding windows. In *ICDE*, pages 502–513, 2005.

[36] M. Morse, J. M. Patel, and H. V. Jagadish. Efficient skyline computation over low-cardinality domains. In *VLDB*, pages 267–278, 2007.

[37] M. D. Morse, J. M. Patel, and W. I. Grosky. Efficient continuous skyline computation. In *ICDE*, page 108, 2006.

[38] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD Conference*, pages 467–478, 2003.

[39] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. In *SIGMOD*, volume 30, pages 41–82, 2005.

[40] J. Pei, B. Jiang, X. Lin, and Y. Yuan. Probabilistic skylines on uncertain data. In *VLDB*, pages 15–26, 2007.

[41] J. Pei, W. Jin, M. Ester, and Y. Tao. Catching the best views of skyline: a semantic approach based on decisive subspaces. In *VLDB*, pages 253–264, 2005.

[42] D. Sacharidis, S. Papadopoulos, and D. Papadias. Topologically-sorted skyline for partially-ordered domains. In *ICDE*, pages 1072–1083, 2009.

[43] N. Sarkas, G. Das, N. Koudas, and A. K. H. Tung. Categorical skylines for streaming data. In *SIGMOD*, pages 239–250, 2008.

[44] K. Shim, R. Srikant, and R. Agrawal. High-dimensional similarity joins. In *ICDE*, pages 301–311, 1997.

[45] K. Tan, P. Eng, and B. Ooi. Efficient progressive skyline computation. In *VLDB*, pages 301–310, 2001.

[46] Y. Tao and D. Papadias. Maintaining sliding window skylines on data streams. In *IEEE TKDE*, volume 18(2), pages 377–391, 2006.

[47] R. Torlone and P. Ciaccia. Finding the best when it's a matter of preference. In *SEBD*, pages 347–360, 2002.

[48] R. Torlone and P. Ciaccia. Which are my preferred items? In *Workshop on Recommendation and Personlization in E-Commerce*, 2002.

[49] R. Torlone and P. Ciaccia. Management of user preferences in data intensive applications. In *SEBD*, pages 257–268, 2003.

[50] W.Kießling. Foundations of preferences in database systems. In *VLDB*, pages 311–322, 2002.

[51] R. C. Wong, A. W. Fu, J. Pei, Y. S.Ho, T. Wong, and Y. B. Liu. Efficient skyline querying with variable user preferences on nominal attributes. In *PVLDB*, volume 1, pages 1032–1043, 2008.

[52] Y. Yuan, X. Lin, Q. Liu, W. Wang, J. X. Yu, and Q. Zhang. Efficient computation of the skyline cube. In *VLDB*, pages 241–252, 2005.

[53] S. Zhang, N. Mamoulis, and D. W. Cheung. Scalable skyline computation using object-based space partitioning. In *SIGMOD*, pages 483–494, 2009.

[54] Z. Zhang, Y. Yang, R. Cai, D. Papadias, and A. Tung. Kernel-based skyline cardinality estimation. In *SIGMOD*, pages 509–522, 2009.