# ZINC: Efficient Indexing for Skyline Computation

Bin Liu Chee-Yong Chan
Department of Computer Science
School of Computing
National University of Singapore
{liubin,chancy}@comp.nus.edu.sg

## ABSTRACT

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 TSS indexing scheme for skyline queries.

## 1. INTRODUCTION

Given a set of data records $D$, a skyline query returns the interesting subset of records of $D$ that are not dominated (with respect to the attributes of $D$) by any records in $D$. A data record $r_1$ is said to dominate another record $r_2$ if $r_1$ is at least as good as $r_2$ on all attributes, and there exists at least one attribute where $r_1$ is better than $r_2$. Thus, a skyline query essentially computes the subset of "optimal" records in $D$, which has many applications in multi-criteria optimization problems.

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 (TO)*, where the best value for a domain is either its maximum or minimum value. However, in many applications, some of the attribute domains are *partially ordered (PO)* such as interval data (e.g. temporal intervals), type hierarchies, and set-valued domains, where two domain values can be incomparable. A number of recent research work [2, 8] has started to address the more general skyline computation problem where the data attributes can include a combination of TO and PO domains. SDC$^+$ [2] is the first method proposed for the more general skyline query problem, which is an extension of the well-known BBS index method [7] 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 PO domains is TSS [8], which is also based on BBS. Unlike SDC$^+$, TSS uses a precise representation of a PO value $v_p$ by mapping $v_p$ into a TO value $v_t$ and a set of interval values. The TO value $v_t$ corresponds to the ordinal value of $v_p$ in a topological sorting of the PO domain values. By using a precise representation, TSS avoids the overhead incurred by SDC$^+$ to filter out false positive skyline records.

Recently, a new index method called ZB-tree [5] has been proposed for computing skyline queries for TO 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 for TO domains, one question that arises is whether the ZB-tree approach can outperform the state-of-the-art BBS-based TSS approach, when extended to handle PO domains. In this paper, 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 PO 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 is 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 TSS as well as two ZB-tree variants based on different encoding schemes.

For simplicity and without loss of generality, we assume in this paper that for TO domains, smaller values dominate larger values.

The rest of this paper is organized as follows. Section 2 surveys related work and Section 3 provides more background on ZB-tree which is the basis of our proposed ZINC approach. In Section 4, we introduce our novel nested encoding scheme and the proposed ZINC method. Section 5 presents our experimental evaluation results. Finally, we conclude in Section 6.

## 2. RELATED WORK

Skyline query processing has attracted a lot of research. In this section, we review only the work that handles data with both TO and PO domains.

Prior to ZB-tree, the state-of-the-art approach for data with only TO domains was BBS [7]. Not surprisingly, the first work that handles PO domain is based on BBS [2]. The main idea is to map each PO attribute into an approximate representation consisting of a pair of TO attributes. The transformed data is then indexed using BBS. Due to the approximate representation, this approach requires post-processing of false positive skylines. Although this limitation is alleviated with some optimization technique to allow partial progressive skyline computation, the overhead of dominance comparisons can be high.

The state-of-the-art approach for computing skylines with PO domains is TSS [8]. This approach is also based on BBS, but unlike the approach in [2] which maps each PO domain value to a single interval value, TSS uses a precise representation by mapping each PO domain value into an ordinal number with respect to a topological ordering of the PO domain values and a set of interval values. In this way, the overhead of post-processing false positive skylines is avoided.

LatticeSky [6] is an efficient approach to process skyline queries for low-cardinality PO 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 tuples by making use of the lattice structure. LatticeSky works well when the PO attribute domains have low cardinality such that the lattice structure can fit in main-memory.

Another recent direction is the work on skyline computation for continuous streaming data with PO domains [9, 4]. 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.

Yet another recent direction is the work on dynamic skyline queries which are skyline queries where the user preferences are specified at run-time. Specifically, for data with categorical attributes, the partial orders representing the user's value preferences for such attributes are given as part of the input skyline query. Three different approaches have been proposed for dynamic skyline queries. The first approach is based on TSS [8], where an R-tree index is built for every combination of PO domain values. The second approach is the IPO-Tree method [10], which is based on materializing a set of dominating relationship views and using a subset of these materialized views at run-time to process an input dynamic skyline query. The third approach is Adaptive-SFS [3], which is based on re-sorting the data based on the run-time user's preference specification. In this paper, our focus is on skyline queries where the partial orders are static; extending our approach to handle dynamic queries is part of our future work.

## 3. ZB-TREE METHOD

In this section, we review the ZB-tree method [5], which our proposed method is based upon. This method is designed for data where all the attributes have TO 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). Fig. 1(b) depicts an example of Z-order curve on the set of 2D data points shown in Fig. 1(a). By ordering data points in non-descending order of their Z-addresses, ZB-tree has two very 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 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. Fig. 1(d) depicts the ZB-tree built on the dataset shown in Fig. 1(a), where the minimum and maximum leaf node capacity are $1$ 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 Fig. 1(c) 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 is easily derived by appending 0s (resp., 1s) to the common prefix of the Z-addresses of the two endpoints of the corresponding curve segment.

The ZB-tree method utilizes an in-disk ZB-tree (named *SRC*) and an in-memory ZB-tree (named *SL*) to index the data points and computed skyline points, respectively. Skyline points are computed by invoking ***ZSearch(SRC)*** (shown in Appendix A) 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 point 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 (shown in Appendix A) to check whether the corresponding RZ-region or data point of *E* can be dominated by the 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.

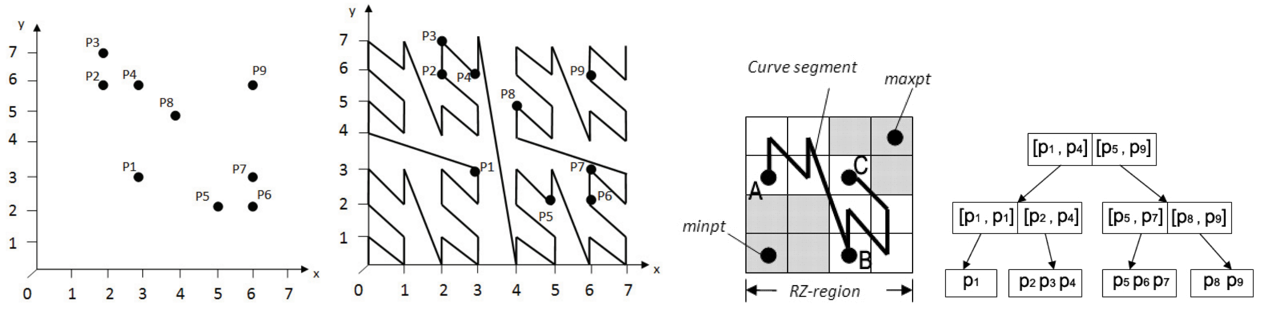
Due to the monotonic ordering property of ZB-tree, each visited data point in a leaf node that is not dominated by any skyline point in *SL* is guaranteed to be a skyline point and is inserted into *SL* and output to the users immediately. The clustering property of ZB-tree enables many index subtree traversals to be efficiently pruned leading to its superior performance over BBS [7].

## 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 TO as well as PO attribute domains. ZINC is basically a ZB-tree that uses a novel encoding scheme to map PO domain values into bitstrings. Once the PO domain values have been mapped into bitstrings, the mapped bitstrings of all the attributes (whether TO or PO domains) of the records will be used to construct a ZB-tree index. Thus, the index construction and search algorithms for ZINC is equivalent to those of ZB-tree except that ZINC uses a different method for dominance comparisons between PO domain values.

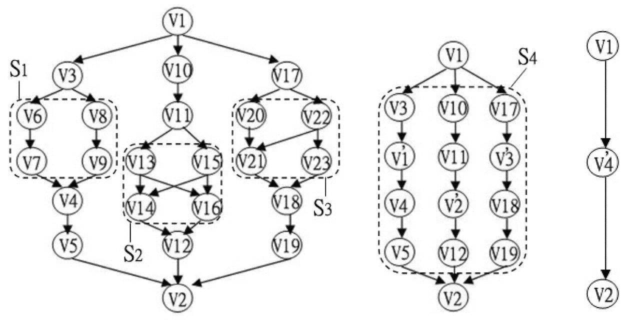
### 4.1 Nested Encoding Scheme

In this section, we introduce a novel encoding scheme, called *nested encoding* (or NE, for short), for encoding values in PO domains. The encoding scheme is designed to be amenable to Z-



(a) 2D data points (b) Z-order curve (c) RZ-region (d) `ZB-tree`

**Figure 1: Z-order curve, RZ-region, and ZB-tree (adapted from [5])**



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

**Figure 2: Partial Order Reduction:** $G_0 \rightarrow G_1 \rightarrow G_2$

order indexing such that when the encoded values are indexed with a `ZB-tree`, the two desirable properties of monotonicity and clustering of `ZB-tree` are preserved.

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 Fig. 2, 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]. $G_1$ 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 4.1. *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 Fig. 2, $S_1$, $S_2$, $S_3$ 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 4.2. *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 4.3. *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,*

[1]The discussion here has been simplified for conciseness. The `PO-Reduce` algorithm described in Section 4.3 actually performs the replacement in two steps: $S_1$ and $S_2$ are replaced in the first step followed by $S_3$ in the second step.

*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 4.4. *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 both the following conditions: (1) $G'$ is not a regular region; and (2) $G'$ is a minimal subgraph of $G$ containing at least two nodes.*

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 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 Fig. 2, 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 $G_0 \to G_1 \cdots \to G_{n-1} \to G_n$, $n \geq 0$, 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 $v \overset{R}{\to} v'$. 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 Fig. 2 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 $v_6 \overset{R_0}{\to} v'_1 \overset{R_1}{\to} v'_4$ and therefore, the depth of node $v_6$ is 2. The containment sequence of $v_3$ is $v_3 \overset{R_1}{\to} v'_4$ and therefore, the 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, $G_0 \to G_1 \cdots \to G_{n-1} \to G_n$, $n \geq 0$, 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 $v_0 \overset{R_0}{\to} v_1 \cdots \overset{R_{k-1}}{\to} v_k$, where each $v_i$ is contained in the region $R_i$, $i \in [0, k)$.

[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 $v_3$.

## 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. $\mathcal{R}(v_i, R_i)$ is 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. $\mathcal{R}(v_i, R_i)$ is 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. However, for `ZINC` to preserve the monotonicity property of `ZB-tree`, the scheme used for encoding irregular regions must satisfy the property that whenever a node $v_x$ dominates another node $v_y$ in $R_i$, the encoded bitstring for $v_x$ must have a smaller value than the encoded bitstring for $v_y$ following our assumption that smaller values dominate larger values. In this paper, we use the bitvector scheme called *Compact Hierarchical Encoding* (CHE) [1] to encode $R_i$; this scheme offers 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 $b_x[1, \cdots, m]$. where $m$ is dependent on the complexity of the irregular region. Given a pair of nodes $v_x$ and $v_y$ in $R_i$, the encoding has the property that $v_x$ dominates $v_y$ iff (1) there exists at

Table 1: Examples of Nested Encodings, $\mathcal{N}(v)$

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

least one bit position $j$ such that $b_x[j] = 0$ and $b_y[j] = 1$, and (2) whenever $b_x[j] = 1$, $b_y[j] = 1$.

As an example of regular region encoding, consider the value $v_9$ in Fig. 2, and let $R_0 = S_1$, $R_1 = S_4$, and $R_2 = G_2$. The containment sequence of $v_9$ is $v_9 \overset{R_0}{\to} v_1' \overset{R_1}{\to} v_4'$, 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., $<< 0, 01 >, < 00, 01 >, < 1, 1 >>$. Similarly, the containment sequence of $v_5$ is $v_5 \overset{R_1}{\to} v_4'$, and $\mathcal{N}(v_5)$ is $< \mathcal{R}(v_4', R_2), \mathcal{R}(v_5, R_1) >$; i.e., $<< 0, 01 >, < 00, 11 >>$.

As an example of irregular region encoding, consider the value $v_{21}$ in the irregular region $R_3 = S_3$. Applying the CHE scheme on $R_3$, the nodes $v_{20}, v_{21}, v_{22}$, and $v_{23}$, are encoded into bitstrings 001, 011, 010, and 110, respectively. The containment sequence of $v_{21}$ is $v_{21} \overset{R_3}{\to} v_3' \overset{R_1}{\to} v_4'$, and $\mathcal{N}(v_{21})$ is $< \mathcal{R}(v_4', R_2)$, $\mathcal{R}(v_3', R_1)$, $\mathcal{R}(v_{21}, R_3)$; i.e., $<< 0, 01 >, < 10, 01 >, < 0, 011 >>$.

Having defined the three different region encodings, we now explain how $\mathcal{N}(.)$ is mapped into a fixed-length bitstring for efficient decoding when used together with Z-order indexing. This requires three refinements to the basic $\mathcal{N}(.)$ scheme described above. First, each node in $G_0$ is encoded with a fixed number of segments. Specifically, $\mathcal{N}(.)$ is extended to consist of a fixed number of $k_{max} + 1$ segments, where $k_{max}$ is the maximum depth of all nodes in $G_0$. When encoding a node $v$ that 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 is 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 are encoded with $w$ bits by padding additional 0 bits. Third, in order to distinguish between regular and irregular region encodings to allow for correct interpretation, each segment starts with a single header bit: a header bit value of 0 (resp., 1) indicates that the segment is a regular (resp., irregular) region encoding. Note that it is not necessary to distinguish between the two regular region encodings since they are encoded using the same two-component format.

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

Table 1 illustrates the nested encodings of four nodes ($v_5$, $v_9$, $v_{15}$, and $v_{21}$) in Fig. 2(a). Since the maximum depth of the nodes in $G_0$, $k_{max}$, is 2, the nested encoding for each node in $G_0$ consists of three segments. Among the four example nodes, all the nodes have a depth of 2 except for $v_5$, which has a depth of 1; therefore, the third segment of $\mathcal{N}(v_5)$ is a dummy segment filled with 0 bits. The first bit for each $\mathcal{N}(.)$ is the header bit; and $v_{21}$ is the only example node that has this bit set in the third segment since $v_{21}$ is the only node (among the example nodes) that is contained in an irregular region (i.e., $S_3$). Since the vertical region $S_1$ (which contains $v_9$) has two chains and the horizontal region $S_2$ (which contains $v_{15}$) has two levels, both $v_9$ and $v_{15}$ require only a single bit for encoding their *X-num* components in the third segment. However, since the encoding of $v_{21}$ in the third segment requires three non-header bits, the size of the third segment is therefore four bits (including the header bit).

**Dominance Comparisons.** The dominance comparison between two nested encodings $\mathcal{N}(v_i)$ and $\mathcal{N}(v_j)$ is performed one segment at a time starting with the first segment. If the $i^{th}$ segment values of $\mathcal{N}(v_i)$ and $\mathcal{N}(v_j)$ are not equal, this means that the comparison is conclusive (i.e., the nodes are either incomparable or one node dominates the other) and the dominance checking terminates at the $i^{th}$ segment. If the $i^{th}$ segment values of $\mathcal{N}(v_i)$ and $\mathcal{N}(v_j)$ are equal, there are two possibilities: if the $i^{th}$ segment is the last segment, then the nodes are incomparable; otherwise, the comparison at the $i^{th}$ segment is inconclusive and the comparison proceeds to the $(i + 1)^{th}$ segment.

As an example, consider the dominance comparison between $\mathcal{N}(v_5)$ and $\mathcal{N}(v_9)$ in Table 1. Since both $v_5$ and $v_9$ are contained in the same virtual node $v_4'$ in $G_2$, their values for the first segment are equal which means the comparison is inconclusive. The comparison then proceeds to the second segment. Since $v_9$ is contained in the virtual node $v_1'$ in $G_1$ which is along the same chain as $v_5$, both $\mathcal{N}(v_5)$ and $\mathcal{N}(v_9)$ have the same X-num values. However, since the Y-num value of $\mathcal{N}(v_9)$ is smaller than that of $\mathcal{N}(v_5)$, the comparison concludes that $v_9$ dominates $v_5$, and the dominance comparison terminates at the second segment.

## 5. PERFORMANCE STUDY

To evaluate the performance of our proposed ZINC, we conducted an extensive set of experiments to compare ZINC against three competing methods: TSS and the two basic extensions of ZB-tree, namely, TSS+ZB and CHE+ZB. Our experimental results show that ZINC outperforms the other three competing methods. Given that both TSS+ZB and CHE+ZB are also based on ZB-tree, the superior performance of ZINC demonstrates the effectiveness of our proposed NE encoding for PO domains.

**Algorithms:** We consider two variants of the main competing method, TSS: an unoptimized variant of TSS (denoted by TSS) and an optimized variant of TSS (denoted by TSS-opt). In TSS, the set of intervals associated with each data / index entry's PO value are stored explicitly with the entry, while in TSS-opt, the intervals associated with an entry are retrieved from a separate precomputed structure. We compare against TSS in this section and TSS-opt in Appendix D.

To compare the effectiveness of our proposed nested encoding scheme, we also introduced two variants of ZB-tree that are based on using different schemes to encode PO domains. The first variant, TSS+ZB, combines the TSS encoding scheme with the ZB-tree method. Each PO domain value $v_p$ of a data point is encoded into a bitstring based on its ordinal value $v_t$ in a topological sorting of the PO domain values. The inclusion of $v_t$ in the derivation of the data point's Z-address is important to ensure ZB-tree's monotonicity property. Each leaf node entry in TSS+ZB stores a data point $p$ together with the interval set representation of each of $p$'s PO attribute values. In each internal node entry of TSS+ZB, besides storing the *minpt* and *maxpt* of the corresponding RZ-region (similar to what is done in ZB-tree), for each PO attribute $A$, a merged interval set for $A$ is also stored which is the union of the interval sets for attribute $A$ of the covered data points. In TSS+ZB, region-based dominance test is applied as follows: if (1) the Z-address of an intermediate skyline point $p_i$ dominates *minpt* of an internal node entry $e_j$, and (2) the interval set of $p_i$ subsumes the interval set of $e_j$ w.r.t. every PO dimension, then the region represented by $e_j$ is dominated by $p_i$ and is pruned from consideration.

The second variant, CHE+ZB, is based on using the *CHE* scheme [1] to encode PO domain values. In contrast to ZINC which uses

**Table 2: 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 |

**Table 3: Properties of PO Domains and Sizes of Indexes**

| | | | Size of Index (MB) | | | |
|---|---|---|---|---|---|---|
| $h$ | Card | Depth | `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 |

the *CHE* scheme for encoding only the irregular regions in a PO domain, `CHE+ZB` encodes a PO domain using only the *CHE* scheme.

**Synthetic datasets:** We generated three types of synthetic datasets according to the methodology in [8]. For TO domains, we used the same data generator as [8] 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* ($ed$)[3], 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 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 [8], all the PO domains for a dataset are based on the same DAG. Table 2 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:** We used a real dataset on movie ratings that is derived from two data sources, *Netflix*[4] and *MovieLens* [5]. Netflix contains more than 100 million movie ratings submitted by more than 480 thousand users on 17770 movies during the period from 1999 to 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 value 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 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 ratings*, which represent, respectively, the movie's average rating (each value is between 0.00 and 5.00) and the 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 the synthetic datasets (Figs. 3(a) to 3(i)) followed by the results for the real dataset (Fig. 3(j)).

**Effect of PO Structure.** Figs. 3(a), 3(b), and 3(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. 3(a) (the y-axis shown is in logarithmic scale), where the height parameter is being varied. The properties of the generated PO domains are shown in the first three columns of Table 3, where *Card* represents the domain cardinality and *Depth* represents the maximum node depth in the DAG; the sizes of the constructed indexes for the four approaches (for 500K dataset) are shown in the last four columns.

For simple partial orders (i.e, height $= 2, 4, 6$ in Fig. 3(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) of the distinct index nodes; `TSS` visited 15% (537 out of 3580), 21% (1524 out of 7255), and 69% (8748 out of 12678) of the distinct index nodes; and `TSS+ZB` visited 30% (604 out of 2012), 19% (1001 out of 5270), and 31% (3153 out of 10172) of the distinct index nodes, respectively, for these three cases.

For complex partial orders (i.e., height $= 8, 10$ in Fig. 3(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 of 58 bits to encode each PO domain value, and `CHE+ZB` actually visited all the index nodes for the skyline computation. Thus, 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

[3]In contrast to [8], which used only the $h$ and $nd$ parameters, the additional $ed$ parameter that we introduced enables a more fine-grained control over the complexity of the generated DAGS.

[4]http://www.netflix.com

[5]http://www.grouplens.org

dominance comparisons in CHE+ZB is about 10 times more than that in ZINC ($9.1 \times 10^8$ vs $9.2 \times 10^7$), and about 3 times more than that in TSS ($9.1 \times 10^8$ vs $2.8 \times 10^8$). 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 shown in Figs. 3(b) and 3(c) for varying node density and edge density, respectively, ZINC outperforms all of CHE+ZB, TSS, and TSS+ZB.

**Effect of Data Distribution.** Fig. 3(d) 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. 3(d), 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. 3(d) corresponding to the five increasing height values are 200, 1780, 4917, 54926, and 286223.

Fig. 3(e) compares the performance for correlated datasets. From the results in Figs. 3(d) and 3(e) we can see that the processing time becomes higher (resp., lower) when datasets are anti-correlated (resp., correlated). The reason is that the number of skyline points becomes larger (resp., smaller) and more (resp., less) computations are incurred.

**Progressiveness.** Fig. 3(f) compares the progressiveness of the algorithms. For each algorithm, we recorded the time it took to output specific percentages of the results (0% for the first returned result, 20%, 40%, 60%, 80% and 100%). The results in Fig. 3(f) indicate that ZINC also outperforms the other methods in terms of progressiveness. While ZINC took only 50% of the total processing time to compute the first 80% of skyline points, TSS+ZB, CHE+ZB, and TSS required 55%, 64%, and 90% of the total time, respectively.

**Effect of Dimensionality.** Fig. 3(g) 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 the 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 many more non-dominated values than TO domains. Again here, ZINC has the best performance.

**Effect of Data Cardinality.** Fig. 3(h) 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.

**Index Construction Time.** Fig. 3(i) 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.

**Performance on Real Dataset.** Fig. 3(j) 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[6]) over the entire partial order domain size is 53%. The results show that ZINC outperforms CHE+ZB, TSS, and TSS+ZB by a factor of 5.5, 15, and 13, respectively.

## 6. CONCLUSIONS

In this paper, we have presented a novel index method, called ZINC, for computing skyline queries on data that contains both TO and PO domains. By combining the strengths of the Z-order indexing method with a novel nested encoding scheme to represent partial orders, ZINC is able to encode partial orders of varying complexity in a concise manner while maintaining a good clustering of the PO domain values. Our experimental results have demonstrated that ZINC outperforms the the state-of-the-art TSS technique for various settings.

## 7. REFERENCES

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

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

[3] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *ICDE*, pages 717–816, 2003.

[4] Y. Fang and C. Y. Chan. Efficient skyline maintenance for streaming data with partially-ordered domains. In *DASFAA*, pages 322–336, 2010.

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

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

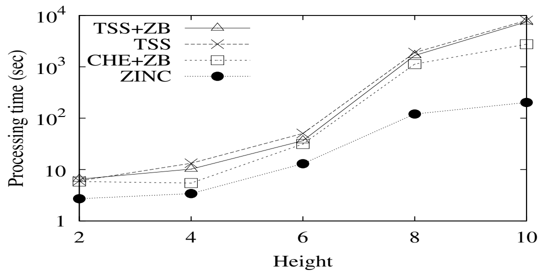
[7] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD*, pages 467–478, 2003.

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

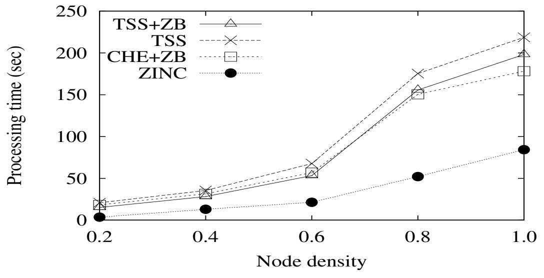
[9] N. Sarkas, G. Das, N. Koudas, and A. K. H. Tung. Categorical skylines for streaming data. In *SIGMOD*, pages 239–250, 2008.

[10] 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. *PVLDB*, 1(1):1032–1043, 2008.

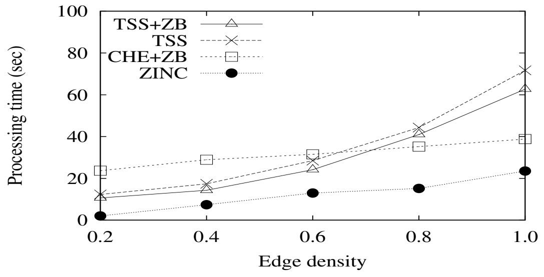
[6]A node $v$ in a PO $P$ is classified as an *irregular node* if the innermost region that contains $v$ in the PO reduction of $P$ is an irregular region; otherwise, $v$ is classified as a regular node.



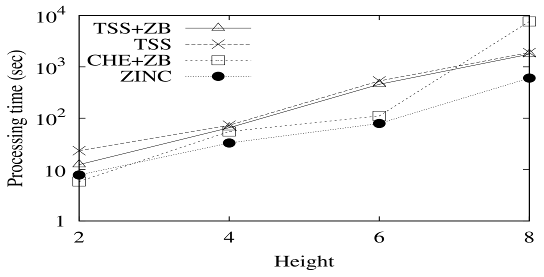
(a) Effect of height



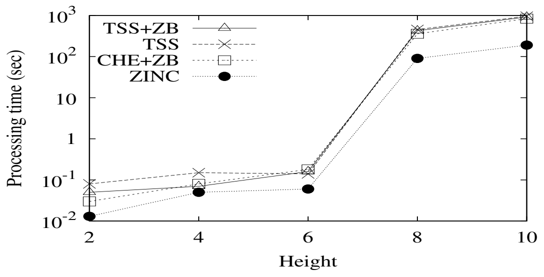
(b) Effect of node density



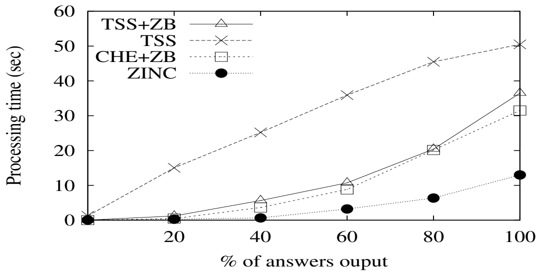
(c) Effect of edge density



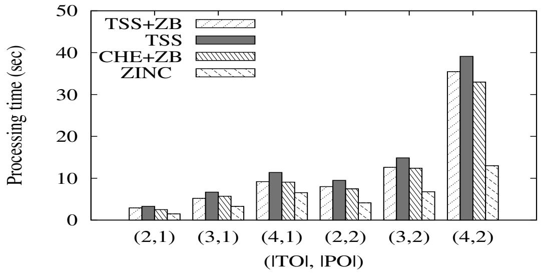
(d) Anti-correlated dataset



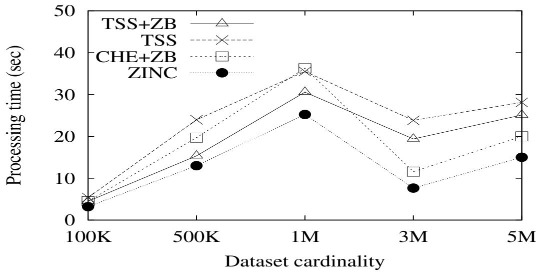
(e) Correlated dataset



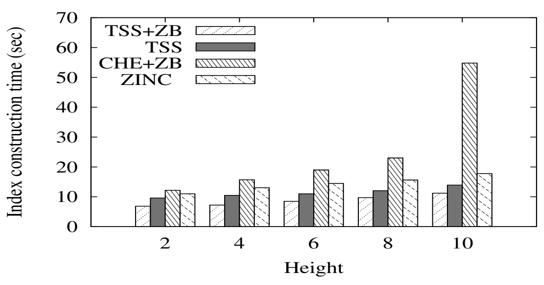
(f) Comparison of progressiveness



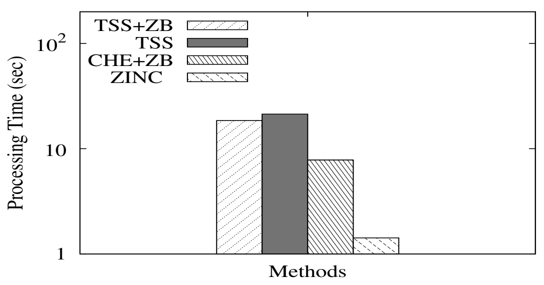
(g) Effect of dimensionality



(h) Effect of data cardinality



(i) Comparison of index construction time



(j) Real dataset

**Figure 3: Experimental Results**

## APPENDIX

## A. ZB-TREE ALGORITHMS

This section presents the details of the two key algorithms of `ZB-tree`, *ZSearch(SRC)* and *Dominate(SL,E)*, that are used in `ZINC`. The main algorithm, *ZSearch(SRC)*, traverses the `ZB-tree`, named $SRC$, to compute the skyline points. The *Dominate(SL,E)* algorithm is invoked to check if an index entry $E$ is dominated by the intermediate skyline points indexed by the in-memory `ZB-tree` $SL$.

**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 child 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: an index or data entry
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 child node c of n do
6              if c's maxpt dominate E's minpt then
7                  return TRUE;
8              else if c's minpt dominate E's maxpt then
9                  q.enqueue(c);
10     else
11         foreach child data point p of n do
12             if p dominate E's minpt then
13                 return TRUE;
14 return FALSE;
```

## B. EFFECT OF PO DOMAINS

In this section, we present additional experimental results to examine the effect of both the regularity of a partially ordered domain as well as the number of partially ordered domains. The experiments were conducted using a subset of the Netflix real dataset consisting of movies that were produced no later than 2000 that have ratings for six years (between 2000 and 2005); the number of such movies is 10,709.

### B.1 Effect of Regularity of PO Domain

In this section, we examine the effect of the regularity of a PO domain by using a different approach to generate PO domains. In

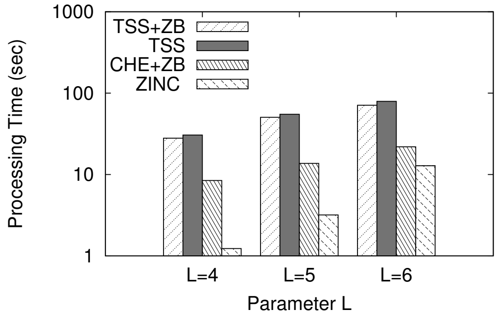


**Figure 4: Effect of Regularity of PO Domain**

this approach, the structure of a PO domain is defined using a number of TO domains (denoted by parameter $L$). As $L$ increases, the generated PO domain becomes more complex and less regular.

Since the Netflix dataset has six years of rating data, we derive $L$ TO domains (with $L \in \{4, 5, 6\}$) using the following approach. First, the movie rating records is partitioned into six subsets based on the year that the rating was given. For a given value of $L$, we choose the $(L-1)$ largest rating subsets (denoted by $R_1, \ldots, R_{L-1}$) and combine the remaining $(7-L)$ subset(s) into one subset (denoted by $R_L$). For each movie $m$, we then compute $L$ average ratings for $m$, with one average rating computed for each $R_i$. Using these $L$ derived rating TO domains, we create a PO domain for movie preference as follows: 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 of the $L$ ratings.

We also derived two TO domains for the dataset: the *average rating* and the *total number of ratings* of a movie over all the six years. In both of these TO domains, higher values are preferred over lower values.

Fig. 4 compares the performance of the four approaches as a function of the parameter $L$. As $L$ increases, the derived PO domain becomes more complex resulting in a larger number of skyline points. Specifically, the number of skyline points for $L = 4$, $L = 5$, and $L = 6$ are 1103, 2412, and 2783, respectively. The respective depths of the PO domains are 13, 15, and 19; and the respective ratios of the size of regular regions (in terms of the number of regular nodes) over the whole PO domain size are 51%, 46%, and 40%. Thus, the PO domain becomes less regular as $L$ increases.

The results show that `ZINC` outperforms the three competing methods in all cases. Observe that the processing time increases 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 a factor of 20 to 5.5.

### B.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 per-

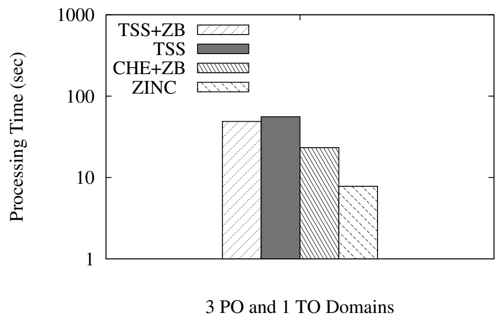


**Figure 5: Effect of number of PO domains**

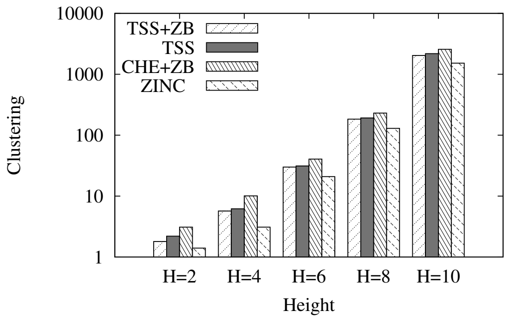


**Figure 6: Comparison of Index Clustering**

formance results in Fig. 5 show that ZINC outperforms CHE+ZB, TSS, and TSS+ZB by a factor of 3.0, 7.2, and 6.3, respectively.

## C. INDEX CLUSTERING COMPARISON

In this section, we compare the clustering effectiveness of the four index methods. Given an index $I$, we define the clustering of $I$, denoted by $clustering(I)$, to be the average distance of each consecutive pair of data points in the leaf level of $I$, the distance of two data points measure the proximity of the points in the multi-dimensional attribute space. Intuitively, an index $I$ with a smaller value of $clustering(I)$ is considered to be more effective in clustering the data points and hence more effective in pruning index nodes to be traversed.

Given two $m$-dimensional data points $p$ and $p'$ (with attributes $A_1,\cdots,A_m$), we define the distance between $p$ and $p'$, denoted by $dist(p, p')$, based on the L2 norm distance function given by the square root of the sum of the squares of the normalized distance, denoted by $ndist(.)$, between p and p' for each attribute; i.e., $dist(p,p') = \sqrt{\sum_{i=1}^{m} ndist(p.A_i, p'.A_i)^2}$. For two TO 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 of that domain.

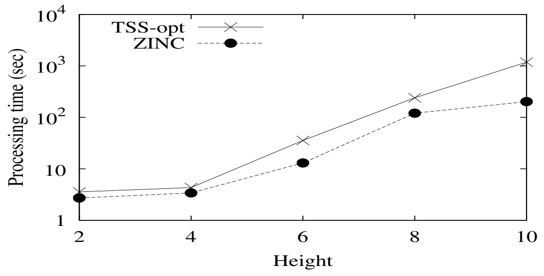
For two PO domain values $v$ and $v'$ in a partial order $G$, we define their normalized distance to be $ndist(v,v') = \frac{L(v,v')+f(v,v')}{3\ L_{max}(G)}$. Here, $L(v,v')$ denote the distance between $v$ in $v'$ in $G$ which is defined in terms of two cases. If $v$ and $v'$ are along the same chain in $G$, then $L(v,v')$ denote the path length between them in $G$; otherwise, $L(v,v')$ is defined to be the length of the shortest path from $v$ to $v'$ via some common ancestor node in $G$. The function $f(v,v') = 0$ if $v$ and $v'$ are along the same chain in $G$; otherwise, $f(v,v') = L_{max}(G)$, where $L_{max}(G)$ denote the path length of the longest chain in $G$. The intuition behind the definition of $ndist()$ is that the distance of two PO values along the same chain are considered to be closer than two PO values that are on different chains; therefore, a penalty of $L_{max}(G)$ is added to the distance between two PO values that are not along the same chain.

Figure 6 compares the the $clustering$ metric for the four methods as a function of the height parameter of the PO domain using the synthetic dataset. Note that the y-axis shown is in logarithmic scale; and an index method 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 about 50%, 62%, and 66% of CHE+ZB, TSS, and TSS+ZB, respectively. When the PO domain becomes more complex (i.e., height is 8 or 10), the performance gain of ZINC reduces because the proportion of irregular regions in the partial order increases.

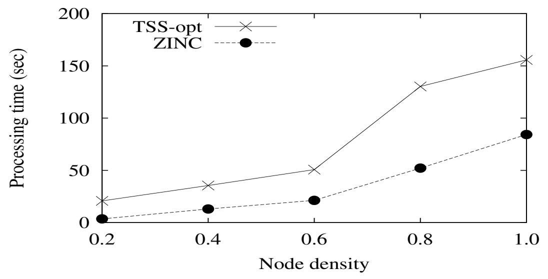
## D. COMPARISON OF ZINC AND TSS-OPT

In this section, we compare the performance of ZINC against an optimized variant of TSS, denoted by TSS-opt. In TSS-opt, the set of intervals associated with each data/index entry's PO value are not stored explicitly in the index structure. Instead, each index entry is associated with a range of ordinal values (denoted by its start and end value) for each PO domain, and the actual collection of intervals associated with an data/index entry's PO domain is retrieved from a separate precomputed structure that is indexed by the start and end pair of ordinal values [8]. In this way, the index structure is more compact with fixed-sized index entries independent of the number of intervals associated with each PO domain value. The implementation of TSS-opt, obtained from the authors of [8], uses a more aggressive optimization that precomputes a three-dimensional boolean array for each PO domain to determine the dominance relationship between an index entry's PO domain value (represented by a pair of start and end ordinal values) and a data entry's PO domain value (represented by an ordinal value). This approach provides very efficient dominance checking in constant time at the cost of requiring more space. In this section, TSS-opt refers to this optimized variant.

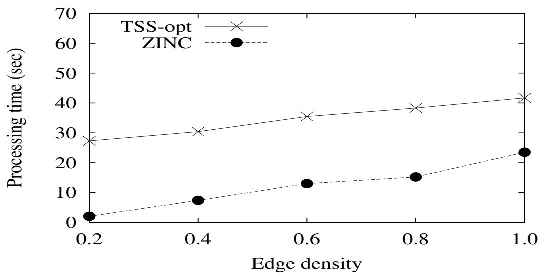
Fig. 7 shows the comparison of ZINC against TSS-opt for the experiments corresponding to those shown in Fig. 3. Observe that TSS-opt outperforms TSS as expected, and ZINC still outperforms TSS-opt demonstrating the effectiveness of the nested encoding scheme used in ZINC.



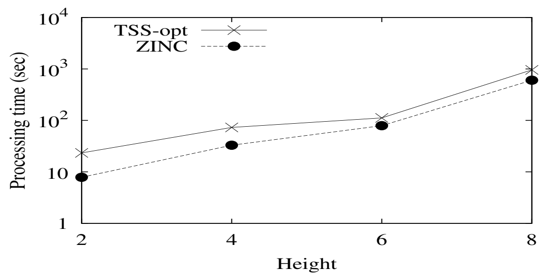
(a) Effect of height



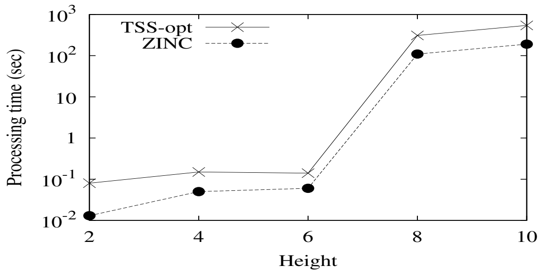
(b) Effect of node density



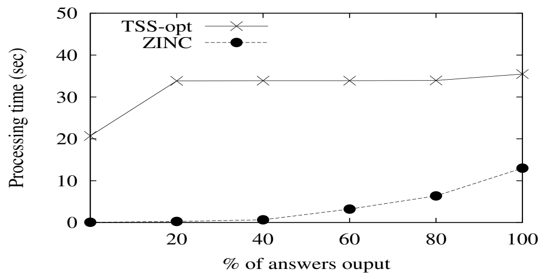
(c) Effect of edge density



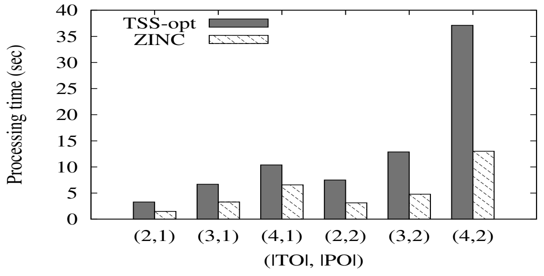
(d) Anti-correlated dataset



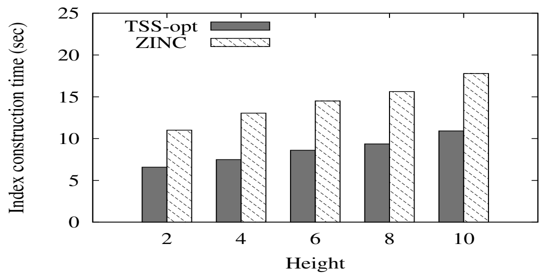
(e) Correlated dataset



(f) Comparison of progressiveness



(g) Effect of dimensionality



(h) Comparison of index construction time

**Figure 7: Comparison of ZINC and TSS-opt**