Efficient Skyline Evaluation over Partially Ordered Domains*

Shiming Zhang, Nikos Mamoulis, David W. Cheung, and Ben Kao
Department of Computer Science,The University of Hong Kong
Pokfulam Road, Hong Kong
{smzhang, nikos, dcheung, kao}@cs.hku.hk

ABSTRACT

Although there has been a considerable body of work on skyline evaluation in multidimensional data with totally ordered attribute domains, there are only a few methods that consider attributes with partially ordered domains. Existing work maps each partially ordered domain to a total order and then adapts algorithms for totally-ordered domains to solve the problem. Nevertheless these methods either use stronger notions of dominance, which generate false positives, or require expensive dominance checks. In this paper, we propose two new methods, which do not have these drawbacks. The first method uses an appropriate mapping of a partial order to a total order, inspired by the lattice theorem and an off-the-shelf skyline algorithm. The second technique uses an appropriate storage and indexing approach, inspired by column stores, which enables efficient verification of whether a pair of objects are incompatible. We demonstrate that both our methods are up to an order of magnitude more efficient than previous work and scale well with different problem parameters, such as complexity of partial orders.

1. INTRODUCTION

Research on skyline evaluation [3, 18, 22, 25, 31] is generally restricted to dimensions with *totally ordered domains* (TODs) relying on the natural correspondence of user preferences to value ordering in such domains. However, in real applications, data may include attributes that are *categorical* and *partially ordered* in nature, such as interval data (e.g., temporal data), type/class hierarchies (e.g., semantic ontologies), and set-valued domains (e.g., DNA segments). Furthermore, preferences may be indirectly (i) induced from *conditional preference networks* (CP-nets) [4]; (ii) mined by preference learning [16]; or (iii) derived as aggregated preferences in subspaces [21]. In most cases, such indirectly derived preferences can only be modeled as partial orders. Skyline evaluation for objects having attributes with partially ordered domains (PODs) is computationally expensive due to the index-resistance of PODs and the increased cost of dominance checks in them.

Previous work [6, 24, 25, 31] relies on partial-to-total order mapping schemes. A single value in the POD is mapped to a set of values in a two-integer domain; then, a dominance check translates to a set of comparisons in the transformed space. The first work in this direction (i.e., SDC [6]) considers static (i.e., pre-defined) PODs. TSS [24] defines the partial-to-total ordering based on a topological ordering, while extending at the same time the ideas of [6] to verify dominance among incomparable attributes in the partial order. These solutions suffer from a large number of false positives/misses, which are produced by the partial-to-total order mapping and have to be checked explicitly.

This paper identifies some uninvestigated properties of partially ordered preferences and exploits them to optimize skyline evaluation on data with PODs. Two approaches are proposed to solve this problem, namely CPS (*Chain-Product decomposition Skyline*) and SCL (*Strata Cyclic Linked skyline*). Our first method (i.e., CPS) is inspired by the *lattice theorem* [17, 30]: *any partially ordered set can be embedded into a product of chains*. Formally speaking, for a partially ordered set (poset) p there exists a mapping function $\phi : p \mapsto \prod_{i=1}^{d(p)}[c_i]$ (where $[c_i] = \{0, 1, \ldots, c_i - 1\}$ and $d(p)$ is the intrinsic dimensionality of p) that embeds p into the product of $d(p)$ chains. A chain $[c_i]$ is a totally ordered set, meaning that all original dominance relationships in the poset p can be entirely preserved in a chain-product decomposition scheme. Thus, any efficient skyline algorithm for totally ordered domains can directly be applied to compute the skyline in the transformed space, without the need of defining any strong or special dominance conditions like the m-*dominance* in [6] or the t-*dominance* in [24].

Our second method (i.e., SCL) accelerates the dominance test between two objects o and o' by identifying a pair i, j of *conflicting* dimensions, such that $o[i]$ dominates $o'[i]$ and $o'[j]$ dominates $o[j]$. The main observation is that *on the average, an incomparability test is much cheaper than a dominance test in the complete set of dimensions*. A good algorithm should identify a pair of conflicting dimensions by applying less than d comparisons (in a d-dimensional problem). To facilitate incomparability checking, SCL employs a column-store style storage, where objects having the same value in a (partially ordered) domain are stored together in a chunk; incomparability can then be verified by fetching a small number of column segments. To avoid expensive joins for locating objects in different columns, SCL connects the columns using a cyclic link data structure. Our empirical evaluation shows that CPS and SCL outperform the competitors (i.e., SDC and TSS) by a wide margin in response time and I/O and they exhibit better scalability.

2. RELATED WORK

This section reviews related work on skyline evaluation, for TODs and PODs. We also discuss some advanced preference models, which involve preference elicitation and optimization.

*Work supported by grant HKU 715509E from Hong Kong RGC.

2.1 Skyline evaluation in TODs

The skyline query (also known as maximal vector computation) retrieves objects that are not dominated by others in a multidimensional space. [3] was the first work on this problem within the database community, which compared a basic *divide and conquer* (D&C) approach against an algorithm that works in a *block nested loop* (BNL) fashion. BNL is better on the average case, however, it may not scale well because it may require a large number of data passes until the complete skyline is computed. In view of this, [10] proposed *sort filter skyline* (SFS), which sorts the whole dataset using a monotone function (e.g., sum of normalized coordinates or entropy), before applying BNL. Sorting guarantees that an object cannot be dominated by objects that follow in the order. Optimized versions of SFS (i.e., LESS and SaLSa) were later proposed in [14] and [1]. Finally, the *object-based space partitioning* (OPS) skyline algorithm [31] operates in a similar fashion, but organizes the skyline found so far in a *left-child/right-sibling* tree, which accelerates the checking of whether the currently read point is already dominated by the found skyline.

The above scan-based approaches do not rely on any predefined index over the data. A set of other techniques [3, 18, 19, 22, 27] require that the data are already indexed before skyline evaluation. The *branch and bound skyline* (BBS) algorithm introduced in [22] is an optimized approach that operates on an R-tree. BBS identifies the skyline points progressively, prioritizing node and object accesses according to distance to the best point in the search space. BBS is shown to be I/O optimal and superior to previous approaches. [19] proposed a ZBtree that indexes the objects with the help of a Z-order curve, which is compatible with the dominance relation. This way, redundant dominance checks are avoided and the ZBtree is found more appropriate than the R-tree.

A thorough space and time complexity analysis for skyline computation was conducted in [14]; besides, skyline cardinality estimation has been studied in [32]. Skylines in high dimensional spaces tend to be large and hard to interpret and use. For this reason, skyline definitions that consider dimensional subspaces [7, 28] have been proposed. Moreover, efforts have been devoted to dynamic skyline search [12, 8], probabilistic skyline computation [23] and skyline computation over uncertain data [20].

2.2 Skyline evaluation in PODs

While most of the research has focused on totally ordered domains, there has been a number of proposals [6, 24, 25] for skyline evaluation over partially ordered domains involving nominal dimensions. SDC (*Stratification by Dominance Classification*) is the first approach in this direction. For each partially ordered domain (POD), SDC computes a minimum spanning tree of the lattice that defines the partial order and encodes the dominance relationships implied by this tree in a two-integer domain. The transformed domain can be indexed and algorithms like BBS [22] can be applied to compute the skyline. However, this partial-to-total order domain mapping mechanism does not entirely preserve all dominance relationships in the original domain; as a result, *false positives* may be included in the skyline of the transformed space. To alleviate this problem, SDC distinguishes two different domain values in the POD: *completely covered* values of which all dominating paths to other values are included in the spanning tree; and *partially covered* ones. This approach can be extended to organize objects into different disjoint strata based on their covered values in the spanning tree, and the intermediate skyline points can be maintained into two strata on-the-fly: the *completely covered* skyline points (stratum 1) and the *partially covered* ones (stratum 2). Skyline points in stratum 1 can directly be output, but those in stratum 2 could be false positives, so they need to be cross-examined against all skyline points in both strata. SDC$^+$ is an optimized version of SDC that further distinguishes partially covered objects into multiple strata and organizes them in separate indexes to improve efficiency and progressiveness. STARS (*Streaming Arrangement Skyline*) [25] is an extension of SDC$^+$ for streaming data. TSS (*Topologically Sorted Skyline*) [24] avoids the overhead of SDC$^+$ for managing and checking false positives; in addition, skyline points are computed progressively. TSS applies topological sorting on the poset, generating a total order. However, directly computing the skyline using this order may miss some skyline points, as two incomparable values are now ordered. To alleviate this problem, TSS employs an efficient, t-dominance check, which is based on an encoding using a minimum spanning tree on the poset.

TSS has significantly lower cost compared to SDC because it directly verifies misses using the encoded poset, at the expense of increasing the space and I/O overhead, since quadratic space (in the domain size) is required to encode the information of the topological order, which is incompatible with the original partial order. The performance of TSS is sensitive to the size of the skyline and the density of the original poset, as verified by our experiments.

2.3 Preference querying processing

Skyline queries can be viewed as a special case of the Pareto preference operator [9, 13]. The latter is based on a more general dominance definition, which is not necessarily derived by considering preference orders on well-defined object dimensions, while the skyline query explicitly considers total or partial orders at different dimensions to define dominance. A straightforward way to solve a Pareto preference problem is to indirectly define dimensions for each Pareto preference function and treat the problem as a skyline query on top of these custom-based dimensions. Preference frameworks tailored to standard database systems have been introduced in [9, 13]. These models are based on *strict partial orders* and they are semantically rich, easy to handle, and flexible in representing complex preferences which are ubiquitous in real-life applications. Modeling and reasoning with such complex preferences has been studied well in the AI community; a common model is the CP-net [4] and its extensions [5]. Meanwhile, several operators were proposed to evaluate preference queries (e.g., winnow operator and best matches [9]) for the more general preference model. Another interesting line of work addresses the problem of identifying a set of attributes, for which an object is part of the skyline (e.g., the *favorable facet* in [21]). This problem can be viewed as the reverse problem of computing the skyline for a given set of preferences.

3. PRELIMINARIES

Consider a set of objects $\mathcal{O} = \{o_1, o_2, \ldots, o_n\}$ and a set of preferences for them. Assume that each preference is expressed by a partial or total order on an attribute domain. In other words, given a set of d preferences, each preference p_i is applied on a different dimension i, considering each object as a d-dimensional vector $o = (o[1], o[2], \ldots, o[d])$. $\mathcal{D}_i$ denotes the domain of the i^{th} dimension (which can be partially or totally ordered). We use this simplified definition for simplicity. In general, preferences can be defined by combining different attributes in a non-trivial manner (e.g., in a CP-net [4]) to derive partial orders; this way, implicit dimensions can be defined.

A partial order is a binary relation $\preceq$ over a domain $\mathcal{D}_i$, denoted by $(\mathcal{D}_i, \preceq)$, satisfying reflexivity (i.e., $a \preceq a$), asymmetry (i.e., if $a \preceq b$, then $b \preceq a$ cannot hold unless $a = b$), and transitivity (i.e., $a \preceq b$ and $b \preceq c$ implies $a \preceq c$). If the binary relation $\prec$ holds in the set $\mathcal{D}_i$, $(\mathcal{D}_i, \prec)$ is a strict partial order ($a \preceq b \Leftrightarrow a \prec b \vee a =$

b). A total order over a domain $\mathcal{D}_i$ is a special case of partial order, where $a \prec b$ or $b \prec a, \forall a, b \in \mathcal{D}_i$, $a \neq b$. A domain with partial (total) order is a partially (totally) ordered domain POD (TOD).

Given a partial order $p_i = (\mathcal{D}_i, \preceq)$, $\forall o, o' \in \mathcal{O}, o \preceq_{p_i} o' \Leftrightarrow o[i] \preceq o'[i]$ can be interpreted as "*o is better than or as good as o' in the i^{th} dimension w.r.t. p_i*". By aggregating all preferences in all dimensions $\mathcal{P} = \{p_1, p_2, \cdots, p_d\}$, we can determine whether an object is preferable over another. Specifically, an object $o \in \mathcal{O}$ *dominates* $o' \in \mathcal{O}$ with respect to $\mathcal{P}$, denoted by $o \prec_{\mathcal{P}} o$, if and only if o is better than or as good as o' in all dimensions and strictly better than o' in at least one dimension, i.e., $o \prec_{\mathcal{P}} o' \Leftrightarrow \{\forall i \in [1, d], o \preceq_{p_i} o'\} \wedge \{\exists j \in [1, d], o \prec_{p_j} o'\}$. If $o \not\prec_{\mathcal{P}} o'$ and $o' \not\prec_{\mathcal{P}} o$, we say that o and o' are *incomparable*, denoted by $o \sim o'$. The skyline of $\mathcal{O}$ over $\mathcal{P}$, denoted by $\mathcal{S}(\mathcal{O}, \mathcal{P})$, is the superior subset of $\mathcal{O}$ containing all objects that are not dominated by any others in $\mathcal{O}$ with respect to $\mathcal{P}$, i.e., $\mathcal{S}(\mathcal{O}, \mathcal{P}) = \{o \in \mathcal{O} | \nexists o' \in \mathcal{O} \backslash o, o' \prec_{\mathcal{P}} o\}$.

As for any value $v \in \mathcal{D}_i$ in a partial order p_i, the set of *superiors* (resp. *inferiors*) of v is defined by $sup(v) = \{u \in \mathcal{D}_i | u \prec_{p_i} v\}$ (resp. $inf(v) = \{u \in \mathcal{D}_i | v \prec_{p_i} u\}$). Similarly, the *parents* (resp. *sons*) of v are $par(v) = \{u \in \mathcal{D}_i | u \prec_{p_i} v, \nexists z \in \mathcal{D}_i, u \prec_{p_i} z \prec_{p_i} v\}$ (resp. $son(v) = \{u \in \mathcal{D}_i | v \prec_{p_i} u, \nexists z \in \mathcal{D}_i, v \prec_{p_i} z \prec_{p_i} u\}$). The set of *incomparable pairs* $inc(p_i)$ of p_i is defined by $inc(p_i) = \{(v, u) | v, u \in \mathcal{D}_i, v \sim u \text{ in } p_i\}$. $inc(p_i)$ is an irreflexive and symmetric binary relation on $\mathcal{D}_i$ *w.r.t.* p_i, i.e., $(v, u) \in inc(p_i)$ implies that $(u, v) \in inc(p_i)$. The *dominance level* $l(v)$ of value v in a partial order p_i is defined as $l(v) = 0$ if $par(v) = \emptyset$; otherwise, $l(v) = \max\{l(u) | u \in par(v)\} + 1$. Therefore, $v \prec_{p_i} u$ implies $l(v) < l(u)$, $l(v) = l(u)$ implies $v \sim u$, and the dominance level defines a *topological* order for p_i. Finally, a total order of size c can be isomorphically represented by a chain of the first c non-negative integers $\{0 < 1 < \cdots < c-1\}$, denoted by $[c]$. A table is included in Appendix A to denote the most frequently used notation throughout the paper.

4. CHAIN-PRODUCT APPROACH

In this section we describe our first method: *chain-product skyline* (CPS). We first discuss how a partial order can be embedded into a product of chains. Then, we adapt an efficient skyline algorithm over TODs to operate on the transformed space.

4.1 Embedding into Products of Chains

The lattice theorem in combinatorics [29, 30] states that the subset of the Cartesian product of some chains is a partial order and that any partial order can be embedded into a chain product, dually. Figure 1 shows an exemplary product of two chains (i.e., $[3] \times [4]$) as a lattice. A subset of this lattice (shown with solid lines) consists of 8 elements $\{a, b, c, d, e, f, g, h\}$ in a partial order; there are also four nodes of the lattice (i.e., $0, 1, i$ and j) which do not correspond to any data element. Conversely, the partial order defined by the solid arrows in this figure can be embedded into the product of two chains, such that each element is represented by a tuple in the 2D space (e.g., $a \leftrightarrow (0, 1)$, $f \leftrightarrow (2, 2)$).

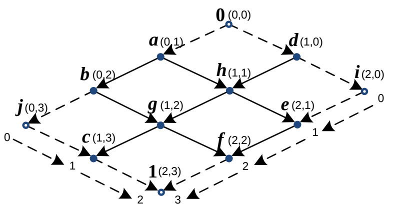


Figure 1: Partial order and a $[3] \times [4]$ Chain product

Formally, there exists a bijective mapping $\phi : \mathcal{D}_i \mapsto \prod_{j=1}^{m_i} [c_j]$ that embeds a partial order $p_i = (\mathcal{D}_i, \preceq)$ into a product of m_i chains. Each element $v \in \mathcal{D}_i$ is encoded as an m_i-tuple (i.e., $\phi(v) = (v[1], v[2], \cdots, v[m_i])$, such that $\forall j \in [1, m_i], v[j] \in [c_j]$). Then, for any pair of elements $(v, u) \in \mathcal{D}_i$, it holds that $v \preceq u \Leftrightarrow \phi(v) \preceq \phi(u)$ (i.e., $v \preceq u \Leftrightarrow \{\forall j \in [1, m_i], v[j] \leq u[j]\}$).

Therefore, the embedding of partial order into a chain product is an isomorphic mapping, and the dominance relationships between values in the poset are preserved by the embedding. The minimum number of chains that *tightly* embed the given partial order p_i is the *dimensionality* of p_i, denoted by $d(p_i)$. Finding $d(p_i)$ is NP-complete when $d(p_i) \geq 3$, while detecting whether $d(p_i) \leq 2$ can be done in polynomial time [29, 30]. Thus, there have been efforts in finding an approximate optimal embedding based on heuristics [29, 30]. In this paper, we adopt the embedding method proposed in [30]. This approach is formally presented in Appendix B. Here, we give a high-level intuitive view of the algorithm.

In a nutshell, given p_i, the chain decomposition method generates a sequence of chains. Each value $v \in p_i$ is mapped to value $v[j]$ in chain $[c_j]$. Each chain preserves a topological order of p_i. Given two incomparable objects u and v in p_i (i.e., $u \sim v$), the method makes sure that there exist two chains $[c_j]$ and $[c_k]$ such that $u[j] < v[j]$ and $u[k] > v[k]$.

If we consider p_i as a directed graph (e.g., Figure 2(a)), then a chain, which is a total order, can be constructed by adding some edges between incomparable values in the graph. For example, in Figure 2(a), a chain has to "include" either the edge (a, d) or the edge (d, a) for the incomparable values a and d. Note that some edges are redundant and need not be added. For example, if we have added edge (d, a) then we need not add (d, b), since the order $d \prec b$ is implied by $d \prec a$ and $a \prec b$. Thus, when constructing a chain, those redundant edges need not be considered. Figure 2(b) shows a graph $G(p_i)$ which captures redundancy between incomparable value pairs $inc(p_i)$ of p_i. When constructing the chains, we only consider vertices in $G(p_i)$ without any incoming edges.

Obviously, the total order given by a chain $[c_j]$ should not conflict with the partial order p_i. This restricts what edges should be added to p_i to obtain a chain. For example, we should not add both (d, a) and (b, d) because that induces the cycle $d \rightarrow a \rightarrow b \rightarrow d$. The two edges are thus *incompatible*. We represent such conflicts by a graph $G^*(p_i)$ in which incompatible pairs are connected by an edge. Figure 2(c) shows an example. By coloring $G^*(p_i)$, we can identify sets of pairs, which, when added to p_i, define a total order (i.e., a chain). For example, the graph in Figure 2(c) can be colored using two colors (0 and 1). The corresponding total orders (chains) are shown in Figure 2(d). The reader can verify that if the ordered pairs colored 0 in Figure 2(c) are included in p_i, this would imply total order C_0. Note that graph coloring guarantees that for each pair of $inc(p_i)$ in a chain, there will be another chain, where the order of this pair is reversed. The next step is to minimize the domains of the chains (see Figure 2(e)). We start a counter 0 for the first value of a chain (e.g., $a = 0$ in C_0). Then we increase the count only if the relationship with the next element is not implied by another chain. For example, since $a \prec b$ is implied by C_1, we set $b = a = 0$ in C_0, but $b \prec d$ is not in C_1, so $d = b + 1 = 1$. This process encodes the coordinates of each value of p_i in the corresponding chain. Figure 2(f) shows the coordinates of all values after the chain-product embedding of p_i.

4.2 Intuitive Skyline Framework

After converting each partial order into a product of totally ordered domains, we can easily transform a hybrid d-dimensional space $\Re^d$ (over totally and partially ordered domains) into a pure totally ordered isomorphic space $\mathcal{N}^{\Phi(d)}$ through a set of isomorphic

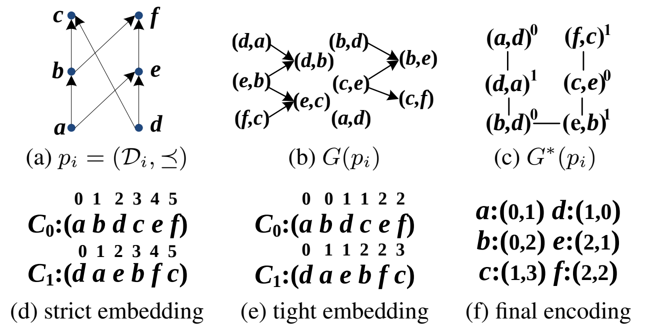


Figure 2: An example on $\phi : \mathcal{D}_i \mapsto [3] \times [4]$

mapping functions $\Phi = \{\phi_i | i \in [1, d]\}$, where $\mathcal{N}$ denotes the domain of integers, ϕ_i the isomorphic mapping function over partial order p_i, and $\Phi(d)$ the resulting dimensionality (obviously, $\Phi(d) = \sum_{i=1}^{d} d(p_i)$). (For clarity, we assume each total order p_i can be isomorphically mapped to a single chain.) Then, each object $o \in \mathcal{O}$ is simply represented as a tuple $(\phi_1(o[1]), \phi_1(o[2]), \cdots, \phi_d(o[d]))$ in the isomorphic space $\mathcal{N}^{\Phi(d)}$.

With this representation, we can directly apply any off-the-shelf skyline algorithm for totally ordered data (see Section 2.1) to $\mathcal{N}^{\Phi(d)}$ to derive the skyline in $\Re^d$. We use CPS (for *Chain-Product decomposition Skyline*) to denote this framework. In order to perform a fair comparison between CPS and the previous proposals (i.e., SDC and TSS) for partially ordered domains, we index the data in the transformed space $\mathcal{N}^{\Phi(d)}$ using a R-tree and we use an adaptation of BBS [22] to compute the skyline. Note that SDC and TSS also use R-trees and BBS during skyline evaluation. In order to minimize dominance checks we embed some ideas of the OSP algorithm [31] in BBS. The skyline points found so far during BBS traversal are organized in a *left-child/right-sibling* (LSRS) skyline tree (as in OSP), which supports efficient dominance checking with the help of bitwise operations. When a candidate point needs to be checked against the current skyline points, we use the LSRS tree to perform the dominance test. In addition, when an intermediate R-tree entry e is accessed we access the LSRS tree to check whether the lower-left corner point e^- of e's MBR is dominated by any of the current skyline points. If yes, then e (and the corresponding R-subtree) is pruned; otherwise, the contents of the R-tree node pointed by e are inserted in the BBS search heap.

5. STRATA CYCLIC LINKED APPROACH

In this section, we propose a new skyline technique over PODs, based on a specialized data storage organization. From Section 3 recall that the *dominance level* $l(v)$ is a function applied on the values v in a partial order p_i to define a *topological* order of p_i (i.e., $v \prec_{p_i} u$ implies $l(v) < l(u)$ and $l(v) = l(u)$ implies $v \sim u$). If, for a pair of objects o and o', there are at least two distinct dimensions i and j, where $l(o[i]) > l(o'[i])$ and $l(o[j]) < l(o'[j])$, then o and o' must be incomparable. This observation inspired us to employ a special storage scheme which facilitates incomparability tests between objects; to prove that a pair of objects is incomparable, it suffices to find two dimensions where they are incomparable. The storage scheme, inspired by column-stores [26, 15], organizes the objects in different columns based on their values in the different dimensions. Each column corresponds to a dimension (i.e., a POD or a TOD). This allows checking for incomparability between objects by comparing their values in the different dimensions, but without having to fetch the entire set of attribute values for two objects that are compared.

The basic column-store architecture has the drawback of requiring expensive joins, when it comes to finding the value of a given object in different dimensions. In order to alleviate this problem, we develop a special *strata cyclic linking* structure, which links all columns, as shown in Figure 3. Each column is stored in a separate file, where all objects are grouped by their values in the corresponding dimension. Each group c in a column is called a *chunk* and corresponds to a value $c.val$ in the corresponding domain. In a column $\mathcal{D}_i$, the chunks are physically ordered according to dominance level of their values in dimension i. In each chunk c of a POD, we encode $c.val$ using the chain-product decomposition encoding (cf. Section 4). This allows for efficient dominance checking between values in the domain; otherwise, traversal of the poset graph might be required. In addition, for each column $\mathcal{D}_i$, we store in a hash table $\mathcal{M}_i$ the addresses of its chunks; given a value v in dimension i, $\mathcal{M}_i$ returns in $O(1)$ time the address of chunk c, where $c.val = v$.

Each chunk consists of multiple *strata*, where objects are grouped based on their value in the next linked dimension. That is, objects in the same chunk, which also have the same domain value in the next linked column, are grouped into the same stratum with a pointer storing the address of the corresponding chunk in the next column. Each stratum is identified by its locating chunk-id and a stratum-id within the chunk (i.e., c_i and s_k in Figure 3). Note that there are links only between every column and the next one in order. Similarly, the last column links back to the first one. The strata in the same chunk are ordered based on the dominance level of their linking chunks at the next column. Appendix C provides details on the construction and maintenance of this data structure.

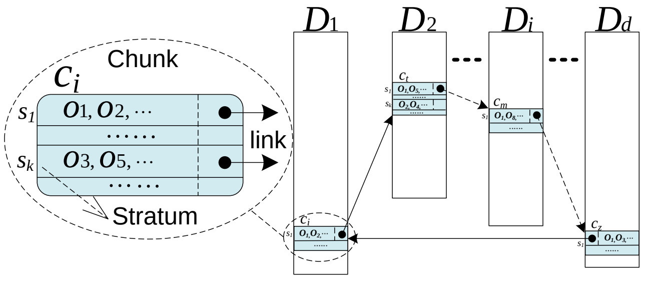


Figure 3: Storage Scheme and Strata Cyclic Links

5.1 Data Strip and Data Strip Level

Now, we define the concepts of *data strip* and *data strip level*, which are used by our algorithm.

DEFINITION 1. *The* ***data strip*** $St(o)$ *and* ***data strip level*** $l_S(o)$ *of an object* $o \in \mathcal{O}$ *are defined as:*

$$St(o) = \bigcup_{i=1}^{d} \{c \in \mathcal{D}_i | o[i] \not\prec_{p_i} c.val\}$$

$$l_S(o) = \sum_{i=1}^{d} l(o[i])$$

$St(o)$ is the set of chunks in all dimensions corresponding to values that are not inferior to the values of o, and $l_S(o)$ is the sum of dominance levels of o across columns; $l_S(o)$ defines a global topological order for the objects. We construct a new column $\mathcal{L}$ which stores the objects and their data strip levels to facilitate incomparability validation. For each object o, $\mathcal{L}$ stores $l_S(o)$ and a file pointer linking to the chunk storing o and having the minimal dominance level across columns. $\mathcal{L}$ is not cyclically linked with other columns. Consequently, the values of any object o can be accessed by following the link from $\mathcal{L}$ to a chunk that contains o and then follow the stratum pointers to all columns cyclically. Meanwhile, within each chunk/stratum c we keep track of the *lower* and *upper* bound of the strip levels of its containing objects as $l_S^-(c)$ and $l_S^+(c)$, i.e., $l_S(o) \in [l_S^-(c), l_S^+(c)], \forall o \in c$. These values help to prune the search space during skyline evaluation. The following example illustrates the details of the storage organization.

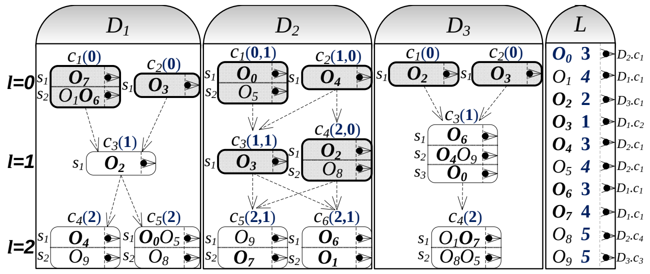


Figure 4: An example of strata cyclic linked column stores

Consider a dataset including 10 objects with 3 PODs, as shown in Figure 4. There are 5, 6 and 4 chunks stored in dimensional columns $\mathcal{D}_1$, $\mathcal{D}_2$ and $\mathcal{D}_3$, respectively. For example, chunk c_1 in $\mathcal{D}_1$, denoted as $\mathcal{D}_1.c_1$, contains two strata containing $\{o_7\}$ and $\{o_1, o_6\}$, respectively. The dashed arrows indicate the dominance relations in the three domains. The links from strata to chunks in other domains are omitted, but they can easily be deduced, e.g., the first stratum in the chunk c_1 of $\mathcal{D}_1$ links to chunk c_5 in $\mathcal{D}_2$, due to object o_7. The labels on the left (e.g., l=1) indicate the dominance levels in the three partial orders. The embedding code of the domain value for a chunk is shown in a bracket over it (e.g., $(2, 1)$ for chunk c_5 of $\mathcal{D}_2$). For each object in $\mathcal{L}$ there is a link to its containing chunk with the smallest dominance level among the three columns. For instance, o_3's link points to c_2 in $\mathcal{D}_1$.

5.2 Skyline Evaluation over Column-stores

The *Strata Cyclic linked Skyline* (SCL) algorithm is based on the following lemmas.

LEMMA 1. *For any object s in the skyline $\mathcal{S}(\mathcal{O}, \mathcal{P})$, strip $St(s)$ includes all objects in $\mathcal{S}(\mathcal{O}, \mathcal{P})$; i.e., $\mathcal{S}(\mathcal{O}, \mathcal{P}) \subseteq St(s)$.*

LEMMA 2. *For a given object $o \in \mathcal{O}$, let $\Lambda_i(o) = \{o' \in \mathcal{O} \backslash o | o'[i] \in sup(o[i]) \vee o'[i] = o[i]\}$. If $\bigcap_{i=1}^{d} \Lambda_i(o) = \emptyset$, then o must be in the skyline; otherwise, o must be dominated by any surviving objects (except o's duplicates).*

Lemma 1 defines a search space for skyline evaluation. SCL should first pick a skyline object τ, whose strip $St(\tau)$ covers a small volume of chunks in all columns. The objects having the minimum $l_S(o)$ value are guaranteed to be in the skyline and their $St(o)$ should be small. For example, $\tau = o_3$ in Figure 4 is a skyline object and the chunks in data strip $St(o_3)$ (shaded in the figure) include all skyline objects (these are $\{o_0, o_2, o_3, o_4, o_6, o_7\}$).

Lemma 2 implies that we can apply an intersection operator over columns to verify the dominance relationships among a set of objects against a given object o, and in turn detect whether o is in the skyline. Algorithm 1 summarizes the procedure for computing the skyline using our storage organization.

After finding the pivot skyline point τ with minimal $l_S(\tau)$, the algorithm sets $St(\tau)$ as the search space. $St(\tau)$ can be determined easily if we know the value $\tau[i]$ of τ in each dimension i; for each value v, such that $\tau[i] \not\prec v$, hash table $\mathcal{M}_i$ is used to find the address of chunk c for which $c.val = v$. Chunks in $St(\tau)$ are accessed in an increasing order of their $l(c)$ values (line 5); this order helps identifying the skyline objects earlier, which are used

Algorithm 1: Strata Cyclic Linked Skyline ($\mathcal{C}$)

```
   Input: $\mathcal{C} = \{\mathcal{D}_0, \mathcal{D}_1, \cdots, \mathcal{D}_{d-1}, \mathcal{L}\}$–Strata cyclic linked columns
   Output: $\mathcal{S}$ – Skyline object $ids$ in $\mathcal{C}$
1  Initialize priority queues $\mathcal{H}$ and $\mathcal{T}$ to be empty        ▷priority on $l_S(\cdot)$
2  Put all top objects of $\mathcal{L}$ into $\mathcal{S}$                         ▷ initial skyline
3  $\tau$ := an object from $S$
4  while $St(\tau)$ is not empty do
5      Remove the chunk $c$ with minimal $l(c)$ from $St(\tau)$ and push its
       contents into $\mathcal{T}$
6      Remove all known skyline/non-skyline objects from $\mathcal{T}$; push all
       skyline ones to $\mathcal{H}$ and erase non-skyline objects from $c$
7      if $\mathcal{T}$ is not empty then
8          $\mathcal{D}_i$ := $c$'s locating column in $\mathcal{C}$, $j := (i + 1) \bmod d$
9          Collect all strata $s \in sup(c)$ in $\mathcal{D}_i$, s.t. $l_S^-(s) < l_S^+(c)$, to $\mathcal{H}$
10         while $\mathcal{T}$ is not empty $\wedge\ j \neq i$ do
11             Remove $\{h \in \mathcal{H} | \nexists t \in \mathcal{T}, h \prec t$ over $\mathcal{D}_j\}$ from $\mathcal{H}$
12             Push $\{t \in \mathcal{T} | \nexists h \in \mathcal{H}, h \prec t$ over $\mathcal{D}_j\}$ into $\mathcal{H}$ and $\mathcal{S}$; Remove
               them from $\mathcal{T}$
13             Update each stratum in $\mathcal{H}$ and $\mathcal{T}$ with its linked strata in $\mathcal{D}_j$
14             $j = (j + 1) \bmod d$                           ▷next linked column
15     if $\mathcal{H}$ is not empty then
16         Erase all remaining objects in $\mathcal{T}$ from the top chunk $c$ in $\mathcal{D}_i$
17         Clear $\mathcal{H}$
18     else Move all objects from $\mathcal{T}$ to $\mathcal{S}$
```

to prune objects in later chunks faster. Assume that c is the next chunk of $St(\tau)$ accessed in this order, and that c is contained in column $\mathcal{D}_i$. Objects in c that are in the skyline are pushed to a heap $\mathcal{H}$. Others that are not marked as non-skyline objects, are pushed to another heap $\mathcal{T}$ (line 6). In addition, all strata in superior chunks of c are pushed to $\mathcal{H}$; these are candidate skyline objects which may dominate those in $\mathcal{T}$ (line 9).

While $\mathcal{T}$ is not empty, the next column is accessed (i.e., $\mathcal{D}_j$, where $j = (i + 1)\ mod\ d$). The key module of the algorithm is to validate the incomparabilities among objects in the two heaps $\mathcal{T}$ and $\mathcal{H}$ across columns (lines 10–14). First, all strata in $\mathcal{H}$, which cannot dominate any strata in $\mathcal{T}$ at the next column $\mathcal{D}_j$, are removed from $\mathcal{H}$. Dually, the algorithm evaluates whether there exist some strata in $\mathcal{T}$ which cannot be dominated by any strata in $\mathcal{H}$ over $\mathcal{D}_j$. All these objects are confirmed to be in the skyline (i.e., they pass the incomparability test), so they are moved to the skyline set $\mathcal{S}$. In addition, they are inserted into $\mathcal{H}$, since they may dominate some remaining candidates in $\mathcal{T}$ (lines 11–12). The dominance among a pair of strata in $\mathcal{T}$ and $\mathcal{H}$ can be determined using their chain-product embedding codes. In addition, the priority of $\mathcal{T}$ and $\mathcal{H}$ on the lower bound of the strip levels can further accelerate dominance checking, since a stratum can only be dominated by strata with strip level smaller than its data strip level.

All surviving strata in $\mathcal{T}$ and $\mathcal{H}$ are intersected with their linked strata in the next column to derive the new $\mathcal{T}$ and $\mathcal{H}$ for the next loop (line 13). After crossing all columns, all objects that remain in $\mathcal{T}$ must be in the skyline if $\mathcal{H}$ is empty; otherwise, they should be dominated by some remaining skyline objects in $\mathcal{H}$ (lines 15–18).

Let us see how SCL operates on the example of Figure 4. First, SCL finds $\tau = o_3$, the object with the smallest $l_S(o)$, and adds it to the skyline $\mathcal{S}$. Strip $St(\tau)$ (all shaded chunks in Figure 4) covers all skyline candidates. The chunks in $St(\tau)$ are ordered based on $l(c)$ and $l_S^-(c)$. The top chunk in $St(o_3)$ (i.e., $\mathcal{D}_1.c_2$) is first picked (line 5). For this, $\mathcal{T}$ is empty (line 7), as $o_3 \in \mathcal{D}_1.c_2$ is already in the skyline. SCL proceeds to pick the next chunk in $St(\tau)$ (i.e., $\mathcal{D}_3.c_2$) in the second loop, for which the same case applies (i.e., $\mathcal{T}$ is empty). The next chunk is $\mathcal{D}_3.c_1$, which contains o_2. Since $Lambda_3(o_2) = \emptyset$, by Lemma 2, o_2 is added to the skyline $\mathcal{S}$. In the next loop, $\mathcal{D}_2.c_2$ is picked and o_4 is added in $\mathcal{S}$ for the same reason. The next loop examines $\mathcal{D}_1.c_1$ and directly adds o_7 to the skyline since it is the only object in the top stratum s_1 of the top-level chunk $\mathcal{D}_1.c_1$; o_7 is also added to $\mathcal{H}$ (since it may prune other objects in $\mathcal{D}_1.c_1$). Objects in $\mathcal{D}_1.c_1.s_2$ (i.e., o_1 and o_6) cannot be confirmed so they are moved to $\mathcal{T}$. By looking at the linked chunks of o_7 and $\mathcal{T} = \{o_1, o_6\}$ (i.e., $\mathcal{D}_2.c_5$ and $\mathcal{D}_2.c_6$), SCL sees that o_7 cannot dominate any objects in $\mathcal{T}$ since $\mathcal{D}_2.c_5$ and $\mathcal{D}_2.c_6$ are incomparable. Hence, o_7 is removed from $\mathcal{H}$. Then, $\mathcal{T}$ is split into two strata (i.e., s_1, s_2 in $\mathcal{D}_2.c_6$). SCL proceeds to the next column $\mathcal{D}_3$ to confirm o_6 as a skyline object and prune o_1, which is dominated by o_6 in $\mathcal{D}_3$, since a complete evaluation loop is finished across three columns. Next, SCL will start a new loop from $\mathcal{D}_2.c_1$, and continue in this fashion until all chunks in $St(o_3)$ have been examined, reporting the final skyline $\mathcal{S} = \{o_3, o_2, o_0, o_4, o_7, o_6\}$.

6. EXPERIMENTAL EVALUATION

We experimentally evaluate the performance of the proposed algorithms against two competitors: SDC$^+$ [6] and TSS [24], using both synthetic and real datasets. The tested algorithms, listed in Table 1, were all implemented in C++ with gcc 4.3.3 and the experiments were conducted on a Linux 2.6.28-15-server with Intel(R) Core(TM)2 Quad 2.66GHz CPU and maximal 4GB RAM. The page size on the server is fixed to 4KB and unless otherwise stated, it uses a memory buffer of 512MB. We evaluate the algorithms in terms of overall response time and page faults starting with a cold buffer.

Algorithm	Description
SDC$^+$	Stratification by Dominance Classification [6] with OSP
TSS	Topologically-sorted Skyline [24] with OSP
CPS	Chain Product Skyline with OSP (Section 4)
SCL	Strata Cyclic Linked Skyline (Section 5)

Table 1: Description of the algorithms

SDC$^+$, TSS, and CPS operate on top of an R*-tree [2]. For each partial order domain, a minimal spanning tree is computed and used by SDC$^+$ and TSS to index the data. This is consistent with the experimental settings in [6, 24]. For CPS, a tight embedding of each partial order into chain products has been computed, as explained in Section 4. Then, an R*-tree is used to index the transformed data incorporating with the chain products; in addition, the OSP scheme of [31] is used to maintain the skyline points, as explained in Section 4.2. For fairness, we use the OSP scheme in the implementation of SDC$^+$ and TSS for maintaining skyline candidates during BBS. On the other hand, SCL uses a different storage and indexing scheme, based on column-wise decomposition.

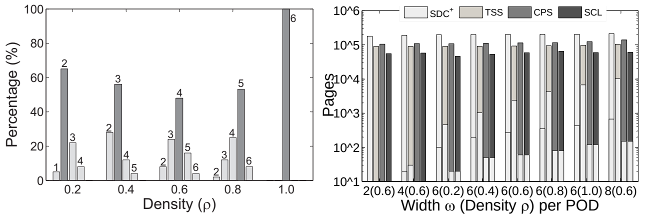
6.1 Description of Data

We conducted our experimental evaluation using both synthetic and real data. Three types of synthetic datasets, *anti-correlated* (AC), *uniform and independent* (UI) and *correlated* (CO) distributions, were generated to model different scenarios according to the methodology in [3]. All datasets are hybrid over *partial order* (PO) and *total order* (TO) domains. For totally-ordered attributes, we used integer values from the normalized domain [1,10000]. Each partially ordered domain is generated as a random directed acyclic graph (DAG), using two parameters: the *width* (ω) and *density* (ρ). Given a pair of ω and ρ values, the complete lattice $\mathcal{G}$ is generated by the product of ω 4-length chains (i.e., $[4] = \{0 < 1 < 2 < 3\}$), which contains 4^ω nodes. Then, each node is randomly removed from $\mathcal{G}$ with a probability (1-ρ). The resulting graph is returned as the final partial order, consisting of $\rho 2^{2\omega}$ nodes on the average. Obviously, $d(\mathcal{G}) \leq \omega$, that is, the derived partial order can be embedded into a product of at most ω chains. Table 2 enumerates the parameters involved in our evaluations and their default values (in bold face), e.g., the data cardinality (n) varies from 100 thousand to 10 million, with a default data size of one million objects.

Parameters	Values
Data Cardinality (n)	100K, 500K, **1M**, 5M, 10M
# of PO dimensions ($\|PO\|$)	**2**, 3, 4
# of TO dimensions ($\|TO\|$)	**2**, 3, 4
Width per POD (ω)	2, 4, **6**, 8
density per POD (ρ)	0.2, 0.4, **0.6** , 0.8, 1.0

Table 2: Parameters and Setting Values

To test the effect of the ω and ρ values to the complexity of the resulting poset, we randomly generated 100 partial orders for $\omega = 6$ and different values of ρ. Figure 5(a) shows the distribution of $d(\mathcal{G})$ in the resulting graphs. For example, for $\rho = 0.4$ more than 50% of the generated graphs have intrinsic dimensionality 3, even though they were generated by a lattice of $\omega = 6$ chains. In our experiments, for each (ω, ρ) pair, we used only the partial orders of the most frequent intrinsic dimensionality (e.g., for $\omega = 6$ and $\rho = 0.4$ we used the generated posets of dimensionality 3).



(a) Dimension w.r.t. density (ρ) (b) Space Requirement

Figure 5: Dimension Distribution & Data Space Consumption

We also used three real datasets in our evaluation, denoted by *Netflix*, *Household* and *Darpa*[1]. *Netflix* contains more than 100 million timestamped movie ratings performed by 480,189 anonymous customers on 17,770 movies between Dec 31, 1999 and Dec 31, 2005. *Household* has about 614,092 tuples between 2000 and 2007 from IPUMS USA census data system. Each tuple records the cost of an American family on six types of expenditures (e.g., electricity, gas, phone, etc) in different groups of families. *Darpa* is extracted from KDD CUP'99 and contains 4,898,431 examples on fraud and intrusion detection data to distinguish between bad and good normal connections. Although there were not explicit partial orders in the attribute domains of these datasets, we conducted some analysis to derive dimensions with meaningful partial orders. Table 3 shows the original dimensionality of the datasets and the number of derived total and partial orders for each of them. The details of the derivation are in Appendix E.

Parameters		Household	Darpa	Netflix
Cardinality(n)		614,092	4,898,431	100,480,507
$\|Skyline\|$		24,896	47,121	13,772
Dimensionality (d)	Orig.	7	31	5
	$\|TO\|$	6	0	1
	$\|PO\|$	1	3	3
	$\Phi(d)$	8	10	7

Table 3: Real Datasets

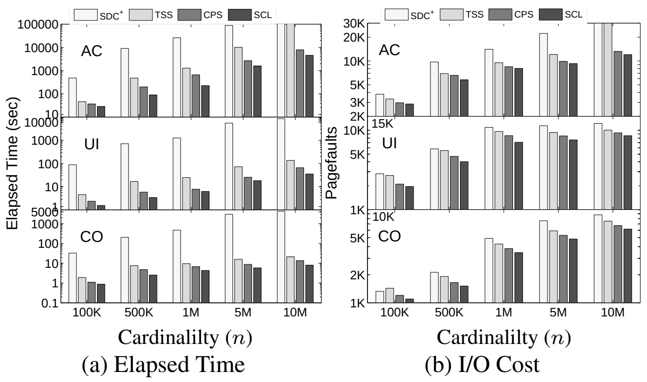
Due to space limitations, a comparison of the data preprocessing costs of the different methods is included in Appendix F. In the subsequent experiments, we do not include this cost when measuring the performance of the algorithms, assuming that the partial orders are mostly static. Thus, domain preprocessing costs are one-time,

[1]Collected from www.netflixprize.com, www.ipums.org, and kdd.ics.uci.edu/kddcup99, respectively.

while object updates (and skyline updates) are independent and can be efficiently incorporated by all indexing schemes (cf. Appendices C.2 and D). Before the skyline evaluation experiments, we present some statistics about the size that the indexes occupy on disk in Figure 5(b), for uniform datasets with different ω and ρ values and default values for the other parameters. The bars are split into two segments: the space for encoding the spanning tree (for SDC$^+$ and TSS) and the chain decomposition (for CPS and SCL) is shown at the bottom and the index size on disk (R*-tree for SDC$^+$, TSS, and CPS, column-store for SCL) is shown at the top. Naturally, partial order encoding occupies negligible space compared to the indexes and is easily stored in memory. TSS occupies less space than the SDC$^+$ and CPS because it uses a single tree of $|PO| + |TO|$ dimensions (as opposed to multiple trees by SDC$^+$), while CPS uses a single tree of higher dimensionality, i.e., $d(|PO|) + |TO|$. SCL occupies less storage than all indexes because (i) it uses linear space to the number of objects and (ii) it "compresses" objects having the same value in a dimension using the chunk representation.

6.2 Experiments on Synthetic Datasets

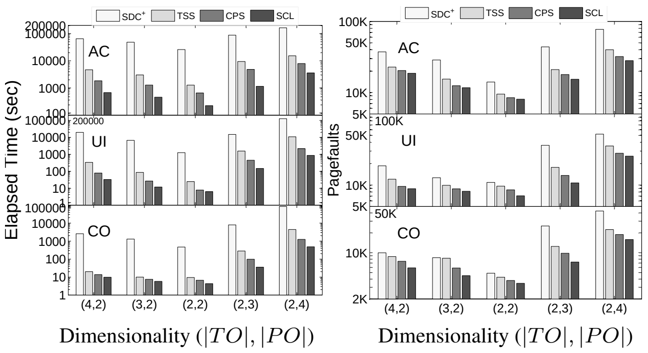
We investigate the performance of our proposed methods against SDC$^+$ and TSS over varying data sizes (n=100K up to 10M), as shown in Figure 6. The response time and I/O cost of all algorithms increases with data cardinality. SCL and CPS are substantially faster than SDC$^+$ and TSS (up to two orders of magnitude). The I/O overhead of all methods increases fast with data cardinality on AC datasets, while the effect is lower for UI and CO datasets. This is due to the fact that the skyline size is close to the database size on AC data and cannot be managed well in the available memory buffer (512M) for large datasets. The effect is smaller for TSS and CPS because they use an effective data structure to manage the currently found skyline and even smaller for SCL, which does not require to hold the whole skyline in memory, but only the points of the skyline which affect the accessed strata during search. SCL also performs best in terms of computational cost, as its effective incomparability testing mechanism avoids accessing and comparing a large proportion of the data. CPS is more efficient than SDC$^+$ and TSS because of its simplicity; the competitors spend more time on validating candidates and on dominance checks for poset values. For example, SDC$^+$ and TSS could not terminate within the maximum allowed time (10^5 sec.) for the 10M AC dataset.



(a) Elapsed Time (b) I/O Cost

Figure 6: Scalability w.r.t. Data Cardinality (n)

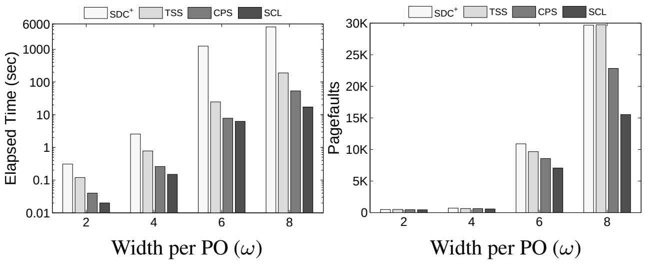
The next set of experiments is conducted on AC, UI, and CO datasets of size one million and investigate the effect of the data dimensionality varying the number of PODs and TODs (Figure 7). Each quadruple of bars indicates the total processing time or page-fault hits of the four algorithms. The x-axis of each plot indicates the setting: the first (resp. second) number corresponds to the number of TODs (resp. PODs) in the correspondind dataset. The overall (and I/O) costs of all methods grow with the number of TODs or PODs due to the increasing problem size. When dimensionality is fixed, the cost grows when more PODs exist (e.g., compare the case of (4,2) against (2,4)), since PODs induce more incomparabilities among domain values and increase the skyline cardinality and the problem complexity. Among algorithms, SCL is the best performing one and scales best with respect to data dimensionality.



(a) Elapsed Time (b) I/O Cost

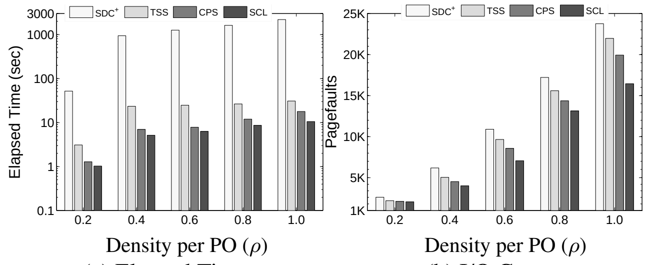
Figure 7: Scalability w.r.t. Dimensionality ($|TO|, |PO|$)

We now study the effect of the PO structure on UI datasets with 1M tuples, two TODs and two PODs (other data distributions show similar trends). In Figure 8, the width (ω) per PO varies from 2 to 8, resulting in an exponential cost increase for all methods and a significant increase in I/O. SCL is on the average 100 times faster than SDC$^+$, which is the slowest method. In general, SCL has a significant performance difference compared to other methods in all settings. Figure 9 evaluates performance, varying node density from 20% to 100%. The cost of all methods increases with growing density. For SDC$^+$ and TSS, the spanning tree covers fewer edges while the topological order is a weaker approximation of the partial order, as density grows. This increases the number of false positives, and negatively affects their CPU and I/O cost. As a result, these methods scale worse than CPS and SCL.



(a) Elapsed Time (b) I/O Cost

Figure 8: Scalability w.r.t. Width per PO (ω)



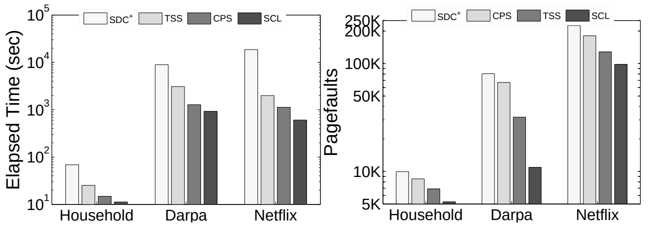
(a) Elapsed Time (b) I/O Cost

Figure 9: Scalability w.r.t. Density per PO (ρ)

6.3 Experiments on Real Datasets

Table 3 lists the intrinsic parameters of the three real datasets. The row labeled *Orig.* shows the original number of dimensions (e.g., 31 attributes in *Darpa*). The next two rows show the derived dimensions, as explained in Appendix C. The last row shows the dimensionality after converting the partial orders to chain products. Note that the chain-product decomposition models each partial order with a relatively small number of chains (i.e., 2 to 4).

Figure 10 shows the costs of all methods on the real datasets. The relative performance of the algorithms is consistent with what we observed on the synthetic data. In terms of response time, SCL is several times to 1-2 orders of magnitude faster than SDC$^+$ and TSS and CPS is worse than SCL but better than SDC$^+$ and TSS. In terms of I/O the difference is smaller, but still significant. The results on *Darpa*, in specific, demonstrate that our proposed methods (i.e., SCL and CPS schema) scale better than the SDC$^+$ and TSS schemes in massive complex PO structures.



(a) Elapsed Time (b) I/O Cost

Figure 10: Performance on Real Datasets

7. CONCLUSIONS

In this paper we proposed two new algorithms for skyline evaluation over data with partially ordered domains. CPS is based on a simple, but intuitive framework; the embedding of posets into chain products followed by skyline evaluation using an off-the-shelf algorithm for totally ordered domains. SCL goes one step further. It exploits the fact that incomparability tests between objects are faster than dominance checks and employs a column-wise storage organization for the data, which facilitates fast incomparability checking. Our experimental evaluation shows that both these methods outperform the state-of-the-art techniques by a wide margin, with SCL being the most efficient method. In the future, we will study the efficient update of the indexing schemes in the case of dynamic partial orders. In addition, we will adapt SCL to apply on arbitrary dimensional subspaces. The column-wise decomposition scheme it uses favors such an adaptation, as opposed to the R*-trees indexes used by other approaches.

8. ACKNOWLEDGMENTS

We would like to thank N. Sarkas, C.Y. Chan, and D. Sacharidis for sharing their codes on SDC, SDC$^+$ and TSS, Hao Li for helping with the implementation, and J. Montero for providing clarifications on the chain-product embedding method.

9. REFERENCES

[1] I. Bartolini, P. Ciaccia, and M. Patella. Efficient sort-based skyline evaluation. *TODS*, 33(4):1–49, 2008.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, 1990.

[3] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, 2001.

[4] C. Boutilier, R. I. Brafman, H. H. Hoos, and D. Poole. Cp-nets: A tool for representing and reasoning with conditional ceteris paribus preference statements. *J. Artif. Intell. Res.*, 21:135–191, 2003.

[5] R. I. Brafman and C. Domshlak. Introducing variable importance tradeoffs into cp-nets. In *UAI*. Morgan Kaufmann, 2003.

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

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

[8] L. Chen and X. Lian. Dynamic skyline queries in metric spaces. In *EDBT*, 2008.

[9] J. Chomicki. Preference formulas in relational queries. *TODS*, 28(4):1–40, 2003.

[10] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting: Theory and optimizations. In *Int. Inf. Sys.*, 2005.

[11] H. S. Christopher D. Manning, Prabhakar Raghavan. *Introduction to Information Retrieval*. Cambridge Univ. Press, 2008.

[12] E. Dellis and B. Seeger. Efficient computation of reverse skyline queries. In *VLDB*, 2007.

[13] M. Endres and W. Kießling. Transformation of tcp-net queries into preference database queries. In *ECAI*, 2006.

[14] P. Godfrey, R. Shipley, and J. Gryz. Algorithms and analyses for maximal vector computation. *VLDB J.*, 16(1), 2007.

[15] P. Hasso. A common database approach for oltp and olap using an in-memory column database. In *SIGMOD*, 2009.

[16] B. Jiang, J. Pei, X. Lin, D. W. Cheung, and J. Han. Mining preferences from superior and inferior examples. In *SIGKDD*, 2008.

[17] B. Korte and J. Vygen. *Combinatorial Optimization: Theory and Algorithms*. 4th Edition, Springer, 2007.

[18] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: an online algorithm for skyline queries. In *VLDB*, 2002.

[19] K. C. K. Lee, B. Zheng, H. Li, and W. C. Lee. Approaching the skyline in z order. In *VLDB*, 2007.

[20] X. Lian and L. Chen. Monochromatic and bichromatic reverse skyline search over uncertain databases. In *SIGMOD*, 2008.

[21] D. Mindolin and J. Chomicki. Discovering relative importance of skyline attributes. *PVLDB*, 2(1):610–621, 2009.

[22] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *TODS*, 30(1):41–82, 2005.

[23] J. Pei, B. Jiang, X. Lin, and Y. Yuan. Probabilistic skylines on uncertain data. In *VLDB*, 2007.

[24] D. Sacharidis, S. Papadopoulos, and D. Papadias. Topologically sorted skylines for partially ordered domains. In *ICDE*, 2009.

[25] N. Sarkas, G. Das, N. Koudas, and A. K. H. Tung. Categorical skylines for streaming data. In *SIGMOD*, 2008.

[26] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-store: A column-oriented dbms. In *VLDB*, 2005.

[27] K. L. Tan, P. K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *VLDB*, 2001.

[28] Y. Tao, X. Xiao, and J. Pei. Subsky: Efficient computation of skylines in subspaces. In *ICDE*, 2006.

[29] W. T. Trotter. *Combinatorics and partially ordered sets: Dimension theory*. John Hopkins Press, 2001.

[30] J. Yáñez and J. Montero. A poset dimension algorithm. *J. Algorithms*, 30(1):185–208, 1999.

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

[32] Z. Zhang, Y. Yang, R. Cai, D. Papadias, and A. K. H. Tung. Kernel-based skyline cardinality estimation. In *SIGMOD*, 2009.

APPENDIX

A. NOTATION USED IN THE PAPER

Table 4 lists the notations used in this paper.

Symbol	Interpretation
$\mathcal{O}, o$	object-set, object
d, n	dimensionality, cardinality of $\mathcal{O}$
$\mathcal{D}_i$	domain of i^{th} dimension
$\mathcal{P}, p_i$	preferences in all dimensions, i^{th} dimension
$\mathcal{S}(\mathcal{O}, \mathcal{P})$	skyline of $\mathcal{O}$ w.r.t $\mathcal{P}$
$o \preceq_{p_i} o'$	o is not worse than o' in i^{th} dimension w.r.t. p_i
$o \prec_{\mathcal{P}} o'$	o dominates o' w.r.t. $\mathcal{P}$
$o \sim o$	o and o' are incomparable w.r.t. $\mathcal{P}$ or p_i
$sup/inf/dup(v)$	superiors/inferiors/duplicates of v in p_i
$par/son(v)$	parents/sons of v in p_i
$[c]$	a chain with c elements $(0<1<\cdots<c-1)$
$inc(p_i)$	the set of incomparable pairs of p_i
$d(p_i)$	intristic dimensionality of partial order p_i
$\phi : p_i \mapsto \prod_{i=1}^{d(p_i)}[c_i]$	a mapping function from p_i to $\prod_{i=1}^{d(p_i)}[c_i]$
$\phi(v), \phi(u)$	image of v and u using mapping function ϕ
$l(v)$	dominance level of the element v in p_i
$G(p_i)$	the directed consistency digraph of p_i
$G^*(p_i)$	the compatibility graph of p_i
$\Re^d, \mathcal{N}^{\Phi(d)}$	hybrid d-dimensional, transformed space
α, β, γ	incompatible *ordered* value pairs in $inc(p_i)$
γ^∂	the inverse ordered pair of γ
$\alpha \to \beta$	including α in p_i induces β in p_i
$\alpha \sim \beta$	α and β are incompatible
$crit(p_i)$	critical pairs of p_i
$\mathcal{K}^s_{p_i}$	strict hypergraph of p_i
$G(\mathcal{K}^s_{p_i})$	induced hypergraph of p_i
$\chi(G)$	chromatic number of a graph $G = (V, E)$
$St(o)$	the data strip of o
$l_S(o)$	the data strip level of o
ω, ρ	width, density of a generated sublattice

Table 4: Table of symbols

B. CHAIN-PRODUCT DECOMPOSITION

Here, we describe the details of the chain-product decomposition algorithm of [30]. Given a poset p_i, the objective is to generate a sequence of chains which are all topological orders of p_i. In addition, if in a chain $[c_j]$, $v < u$ (implying that $v \prec u$), while v and u are incomparable in the original p_i, there should be at least one chain $[c_k]$, in which $u < v$.

We first define some concepts on p_i. An *alternating cycle* is a sequence $\{(v_j, u_j) \in inc(p_i) | u_j \preceq_{p_i} v_{(j+1)\%k}, j \in [1, k]\}$, where k is the *length* of the cycle. The *transitive closure* of p_i is defined as $tr(p_i) = \{(v, u) | v \preceq_{p_i} u, \forall v, u \in \mathcal{D}_i\}$.

DEFINITION 2. *Given a partial order $p_i = (\mathcal{D}_i, \preceq)$, its* ***consistency digraph*** *is a directed graph $G(p_i) = (inc(p_i), E(p_i))$, where nodes are the ordered pairs of values in inc(p_i) and $E(p_i) = \{(\alpha, \beta) | \beta \in tr(p_i \cup \alpha)\}$, i.e., including α in p_i induces β in p_i, denoted $\alpha \to \beta$.*

The addition of an incompatible pair α to p_i implies all the decendants of α in $G(p_i)$. Therefore, the problem reduces to choosing from the vertices $V^*(p_i)$ of $G(p_i)$ with no parents; i.e., $V^*(p_i) = \{\alpha \in inc(p_i) | par(\alpha) = \emptyset$ in $G(p_i)\}$. We say that two pairs $\alpha, \beta \in V^*(p_i)$ are *incompatible*, denoted by $\alpha \sim \beta$, if and only if they can induce an alternating cycle in p_i; i.e., $\exists \gamma \in inc(p_i) \backslash \alpha$ such that $\alpha \to \gamma$ and $\beta \to \gamma^\partial$, where γ^∂ denotes the inverse of ordered pair γ. The *critical pairs* of p_i, with respect to $G(p_i)$ are $crit(p_i) = \{\alpha \in V^*(p_i)\}$ and the *chromatic number* of a graph $G = (V, E)$, denoted $\chi(G)$, is the minimal number of colors needed to color the vertices of G so that no two adjacent vertices share the same color.

DEFINITION 3. *Given a partial order p_i=$(\mathcal{D}_i, \preceq)$, the* ***incompatibility graph*** *is an undirected graph $G^*(p_i) = (V^*(p_i), E^*(p_i))$, where $E^*(p_i) = \{(\alpha, \beta) | \alpha \sim \beta$ in $G(p_i), \forall \alpha, \beta \in V^*(p_i)\}$. The* ***strict hypergraph*** *is the hypergraph $\mathcal{K}^s_{p_i} = (crit(p_i), \mathcal{C})$, where $\mathcal{C} = \{(e_1, e_2, \cdots) \subseteq crit(p_i) | \{e^\partial_1, e^\partial_2, \cdots\}$ forms a strict alternating cycle$\}$. The* ***induced hypergraph*** *from $\mathcal{K}^s_{p_i}$ is $G(\mathcal{K}^s_{p_i}) = (crit(p_i), \mathcal{C}')$, where $\mathcal{C}' = \{e \in \mathcal{C} \mid |e| = 2\}$.*

THEOREM 1. *Given a partial order $p_i = (\mathcal{D}_i, \preceq)$, 1) $G^*(p_i)$ and $G(\mathcal{K}^s_{p_i})$ are isomorphic and then $\chi(G^*(p_i)) = \chi(G(\mathcal{K}^s_{p_i}))$; 2) $d(p_i) = \chi(\mathcal{K}^s_{p_i}) \geq \chi(G(\mathcal{K}^s_{p_i}))$.*

LEMMA 3. *Given a partial order $p_i = (\mathcal{D}_i, \preceq)$ and coloring $G^*(p_i)$, $d(p_i) = \chi(G^*(p_i))$ if and only if no hyperedge in $\mathcal{K}^s_{p_i}$ is equally colored.*

Theorem 1 implies that $d(p_i)$ can be computed by coloring the corresponding incompatibility graph $G^*(p_i)$ if $\mathcal{K}^s_{p_i} = G^*(p_i)$. Otherwise, according to Lemma 3 there exists at least one hyperedge E in $\mathcal{K}^s_{p_i}$ such that some vertices $\{e \in E\}$ share the same color in $G^*(p_i)$. We can add the ordered pairs in $G^*(p_i)$ which share the same color to p_i to extend p_i to a chain. However, some of the resulting chains may contain cycles and in this case they are unacceptable. To address this problem, a recursive procedure has been proposed in [30], denoted by *MinimumRealizer* $(G^*(p_i))$, which returns $\chi(G^*(p_i))$ induced chains. If $\mathcal{K}^s_{p_i} = G^*(p_i)$, then $d(p_i) = \chi(G^*(p_i))$ different chains, which have no cycles, can be derived from p_i based on the different sets of coloring vertices in $G^*(p_i)$. Otherwise, at least one extending chain contains a strict alternating cycle which consists of the vertices of $G^*(p_i)$ but some edges among them cannot be linked in $G^*(p_i)$ (denoted by $\overline{E}$). The problem can be avoided by adding these edges in $G^*(p_i)$; i.e., $G^*(p_i) = G^*(p_i) \cup \overline{E}$. The chromatic number of such an extended incompatibility graph could increase and the procedure is recursively applied until no cycle exists in the induced chains.

Algorithm 2: EmbeddingIntoChainProduct($p_i = (\mathcal{D}_i, \preceq)$)

```
   Output: Bidirection encoding hash map $\mathcal{H}$ for $v \in \mathcal{D}_i$
 1 Compute consistency digraph $G(p_i) = (inc(p_i), E(p_i))$
 2 Compute incompatibility graph $G^*(p_i) = (V^*(p_i), E^*(p_i))$
 3 repeat
 4     Color the incompatibility graph $G^*(p_i)$
 5     $t = \chi(G^*(p_i))$ the chromatic number of $G^*(p_i)$
 6     $\mathcal{C}$ =MinimumRealizer($G^*(p_i)$) [30]      ▷$\mathcal{C} = \{C_1, C_2, \cdots, C_t\}$
 7     if ($\exists c \subseteq C_i$) then ▷exist a cycle $c = \{v_1 \prec v_2 \prec \cdots \prec v_r \prec v_1\}$
 8         $\overline{E} = \{(\alpha, \beta) \notin E^*(p_i) \mid \forall \alpha, \beta \in V^*(p_i), \alpha, \beta \in c\}$
 9         $G^*(p_i) = G^*(p_i) \cup \overline{E}$
10 until (no cycle in $\mathcal{C}$)
11 for $k = 1$ to $t$ do      ▷$C_k = \{v_0 \prec v_1 \prec \cdots \prec v_{n-1}\}, n = |\mathcal{D}_i|$
12     $m_k = 0$
13     for $j = 0$ to $n - 2$ do
14         if ($\exists C \in \mathcal{C} \backslash C_k, v_j \prec v_{j+1}$ in $C$) then
15             $v_j = v_{j+1}, v_j.p_k = v_{j+1}.p_k = m_k$ in $C_k$
16         else $v_j.p_k = m_k$++
```

Algorithm 2 illustrates the method of [30] for embedding a partial order $p_i = (\mathcal{D}_i, \preceq)$ into a product of chains. For partial order $p_i = (\mathcal{D}_i, \preceq)$, each domain value $v \in \mathcal{D}_i$ is encoded as an t-tuple $(v.p_1, v.p_2, \cdots, v.p_t)$, i.e., $\phi : \mathcal{D}_i \mapsto \prod_{l=1}^{t}[m_l]$, where $t = d(p_i)$. Lines 1–10 generate the set of chains $\mathcal{C}$, each of length $n = |\mathcal{D}_i|$. Lines 11–16 compute a *tight* encoding, where the domain of each chain is minimized. The first element of each chain, (corresponding to a value in D_i) is set to 0 and the next element in the chain is set to the previous one if the ordering of the corresponding values in p_i can be implied by another chain; otherwise, it is the previous value increased by 1.

Complexity. The complexity of this algorithm is dominated by the coloring of the incompatibility graph. For a partial order with m values this graph may have as many as $n = O(m^2)$ nodes. Graph coloring is an NP-complete problem. Using an approximate algorithm for coloring can drastically reduce the computation time, at the expense of deriving a sub-optimal coloring. In our implementation, we used the graph coloring algorithm toolkit by Joseph Culberson.[2] The best approximation ratio that can be achieved is O($n(\log\log n)^2/(\log n)^3$), where n is the number of vertices in the graph to be colored, while the complexity of a typical algorithm (DSATUR) is O(n^3).

Reducing the complexity of the decomposition. Given a partial order $p_i = (\mathcal{D}_i, \preceq)$, the *duplicates* of v are $dup(v) = \{u \in \mathcal{D}_i | par(v) = par(u), son(v) = son(u)\}$, which are necessarily incomparable. The partial order p_i can be simplified by merging duplicates and representing them as a single element. The simplified partial order is denoted by $\widetilde{p_i}$. W. T. Trotter [29] has highlighted that $dim(\widetilde{p_i}) \leq dim(p_i)$ and in the embedding of $\widetilde{p_i}$ we may use a smaller number of shorter chains. In practice, we apply the encoding on $\widetilde{p_i}$, and all *duplicate* elements in the original partial order that have the same set of parents and children are encoded by the same image tuple in the chain product. The impact of such a schema is that dominance checking in the transformed space should distinguish among duplicate values and identical ones, because the former imply incomparability, but the latter do not. For example, consider two objects o and o' having equivalent, but incompatible values in poset p_i; e.g., $o[i] = v$, $o'[i] = u$, $u \in dup(v)$. If o dominates o' in the remaining dimensions, o and o' are still incompatible due to p_i. However, if v and u are given the same value in the chain decomposition, o will be found to dominate o'.

C. IMPLEMENTATION DETAILS OF SCL

In this appendix, we discuss issues related to the implementation and maintenance of the strata-cyclic linked structure, used by SCL.

C.1 Construction

Column $\mathcal{D}_i$ is only linked the next in order (i.e., $\mathcal{D}_{i+1 \bmod d}$), therefore we construct the columns one-by-one, in separate files, as follows. We start by creating column $\mathcal{D}_d$, by topologically sorting the objects according to the dominance level of their d-th coordinates (i.e., $l(o[d])$). This creates the chunks for the d-th dimension. For each chunk we defer the division to strata and the generation of pointers, until column $\mathcal{D}_1$ has been constructed. Before flushing each constructed chunk to disk, we update a temporary hash table T_d that enables us to locate for each object o the address of the chunk in $\mathcal{D}_d$ that contains o.

Then, for $i = d - 1$ downto 1 we create column $\mathcal{D}_i$, by performing a topological sort on $l(o[i])$. For each generated chunk c by the sorting, we obtain the locating chunks of c's objects in $\mathcal{D}_{i+1}$, using T_{i+1}. Then, we topologically sort the addresses of these chunks, based on their dominance levels. After sorting we divide c into strata and generate the corresponding pointers (i.e., file pointers to chunk locations in $\mathcal{D}_{i+1}$). After the construction of $\mathcal{D}_i$ is completed, we delete T_{i+1} and construct T_i (to be used in the construction of $\mathcal{D}_{i-1}$).

Finally, $\mathcal{D}_d$ is scanned again and T_1 is used to divide its chunks into strata and generate their pointers to $\mathcal{D}_1$. The construction cost is dominated by sorting the d columns O($dn \log n$), provided that the currently used hash table (i.e., T_{i+1} for the construction of $\mathcal{D}_i$) fits in memory (a realistic assumption). Thus, if each column fits in memory, the data need only be scanned once to create the columns, and each column is scanned once and sorted in memory.

[2]http://webdocs.cs.ualberta.ca/ joe/Coloring/Colorsrc/manual.html

C.2 Maintenance

If the chunks are "packed" in a column file, upon insertion of a new object, the addresses of chunks could be shifted, causing the update of many pointers in the previous column. Therefore, if the dataset is dynamic, changing file addresses of chunks should be avoided as much as possible. Although in our current implementation, we did not consider data updates, the storage scheme could easily be adapted to facilitate changes in the data, in a similar way to hash indexes. First, assuming that the cardinality of the domains is not high, we can allocate one block per chunk. A new object is then inserted to the chunk blocks corresponding to its values, without affecting the addresses of other chunks. If a chunk block overflows, then an overflow block can be allocated and linked to the chunk. This way, each chunk is stored as a linked list of blocks in the corresponding column. Strata pointers always refer to the first block of a chunk. Although this architecture may introduce some additional random accesses within a column (if the sequence of blocks of a chuck have to be accessed), the overhead is manageable compared to having to update a large number of pointers from the previous column. Deletions in the SCL storage structure can be handled similarly. Appendix D discusses how the skyline can be maintained after data updates.

C.3 Impact of Column Ordering

The ordering of domain columns may have impact to the performance, as it determines the number of strata in the chunks. For example, if we reorder the columns in a way such that the number of strata is minimized, then the number of inter-column pointers is minimized, storage space would be saved, and potentially there would be lower I/O. We have experimented with an implementation of SCL with a column ordering which minimizes the number of chunks. For each ordered pair of columns $(\mathcal{D}_i, \mathcal{D}_j)$, we computed the number of inter-column pointers (i.e., number of strata in $\mathcal{D}_i$) that would be required if $\mathcal{D}_i$ is placed before $\mathcal{D}_j$ in the column ordering. Then, we used a dynamic programming algorithm to find the permutation of columns that results in the minimum number of inter-column pointers overall. Nevertheless, the savings in storage and performance achieved by the optimal ordering compared to a random column ordering were not very high (about 10%). On the other hand, finding the optimal ordering has a cost of O(d^2B), where d is the dimensionality and B is the number of blocks in a column, as for all O(d^2) ordered column pairs the number of inter-column pointers should be computed. Thus, the high cost of finding a good column ordering is not compensated by the performance savings it offers.

D. SKYLINE MAINTENANCE

Regardless of the algorithm used to compute the skyline, it can be updated after a new object o is inserted, by comparing o with the existing objects in $\mathcal{S}$; if o is dominated by any object in $\mathcal{S}$, there is no update; if o is incomparable with all objects in $\mathcal{S}$ it is inserted to $\mathcal{S}$; if o dominates some objects in $\mathcal{S}$, these objects are deleted from $\mathcal{S}$ and o is inserted to the skyline.

When an object o is deleted from the dataset, if $\mathcal{S}$ did not contain o, there is no update. If o was part of the skyline, then the skyline should be updated by accessing the objects dominated by o and computing the skyline among them (see [22]). CPS can perform this update by a *constrained skyline* query, as explained in [22].

SCL can perform the update, by setting $\tau = o$ and accessing $St(t)$ to update the skyline, using $\mathcal{S} \setminus o$ as the current skyline and ignoring o (which is also deleted from its containing chunks).

E. GENERATION OF PARTIAL ORDERS FOR REAL DATASETS

In this appendix, we discuss how the partial orders used for the real datasets are generated.

Netflix: Originally, each record in the dataset is a 4-tuple, (*user*, *movie*, *date*, *rating*), indicating the rating that a user gave to a movie and the date when this was recorded. Let r_{ui} be the rating of user u on movie i. Values of r_{ui} range from 1 (star) indicating no interest to 5 (stars) indicating a strong interest. Each user u is associated with a set of items denoted by $R(u)$, which contains all the items for which ratings by u are available. Likewise, $R(i)$ denotes the set of users who rated item i. We derived the new attributes based on a collaborative filtering analysis posted at http://www.netflixprize.com/.

Each user u is modeled using the following attributes: (i) the normalized fraction of movies rated by u: $\frac{R(u)}{\#movies} \times 1000$; (ii) the user's deviation: $\sigma_u = \frac{\sum_{i \in R(u)} (r_{ui} - \mu - \sigma_i)}{\lambda_2 + |R(u)|}$, where μ is the global mean rating, $\sigma_i = \frac{\sum_{u \in R(i)} (r_{ui} - \mu)}{\lambda_1 + |R(i)|}$, and $\lambda_1 = 25, \lambda_2 = 10$; (iii) the associated time deviation: $\sigma_t^u = \frac{\sum_{i \in R(u)} sgn(t_{ui} - t_u) \cdot |t_{ui} - t_u|^\beta}{|R(u)|}$, where μ is the mean of all ratings, $sgn(\cdot)$ is the sign function, t_{ui} is the timestamp of rating r_{ui}, t_u is the mean rating timestamp for user u, and $\beta = 0.4$.

Likewise, each movie is described by following attributes: (i) the fraction of user ratings: $\frac{R(i)}{\#users} \times 1000$; (ii) the observed deviation: $\sigma_i = \frac{\sum_{u \in R(i)} (r_{ui} - \mu)}{25 + |R(i)|}$; (iii) the associated time deviation: $\sigma_t^i = \frac{\sum_{u \in R(i)} sgn(t_{ui} - t_i) \cdot |t_{ui} - t_i|^\beta}{|R(i)|}$, where t_i is the mean rating timestamp for item i.

Using these normalized derived attributes, we apply k-means clustering to divide the users/movies into partitions. For each user cluster, we compute an average rating $\tilde{r}$ and an average deviation $\tilde{\sigma}$. The dominance relationship between two user clusters is defined based on these two values, and a partial order on users is defined. The same process is applied on the movie clusters to define a second partial order. For each clustering, we used the best k, estimated using the *Akaike Information Criterion* based on the minimal *residual sum of squares* (RSS) of all clusterings with up to K clusters [11]. This resulted in 50 customer partitions and 88 movie clusters.

The third partial order on Netflix considers the effect of time on ratings, since an item's popularity may change over time. Time may also have an effect on the behavior of users. For example, a user who tended to give a rating 4 to an average movie, may now rate average movies by 3. For each time unit (i.e., day), we capture the average time deviation for different users on this date and the average time deviation on different movies on this date. Based on these two values, we define a partial order between days. The last partial order is derived from about 40 different date gaps based on user and movie biases by grouping similar time deviations on users and movies.

Given two ratings r_{ui} and r_{vj}, r_{ui} dominates r_{vj} if the triplet of clusters where r_{ui} falls, according to the three derived partial orders, dominates the triplet of r_{vj}. This intuitively means that r_{ui} is more credible than r_{vj}, since professional users give more confident ratings on movies at the right time. We also keep the rating value as a fourth, totally ordered dimension for ranking ratings. Skyline evaluation on the derived dimensions of the ratings can help an analyst to identify the most interesting subset of user ratings in the database.

In a nutshell, the three attributes of the original dataset (user, movie, date) upon which dominance cannot be defined in their original domains are replaced by attributes with meaningful dominance relationships in their domains, captured by the corresponding partial orders.

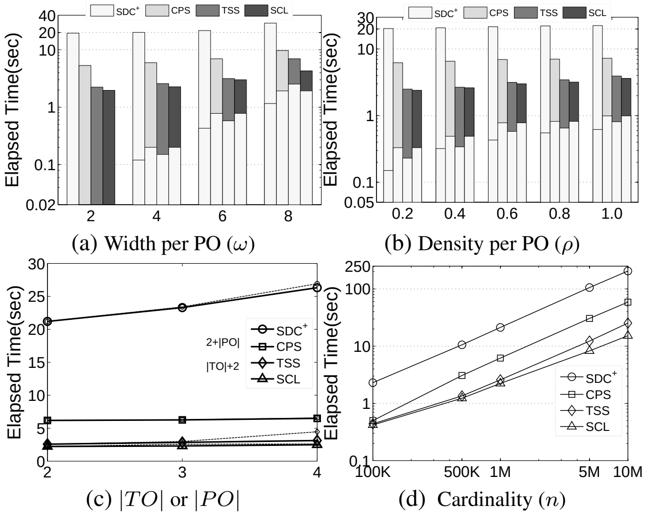
Darpa: The 31 original attributes in this set are divided into three group features (i.e., basic, content, and traffic features), which contain 9, 13 and 9 attributes, respectively. Three partial orders can directly be defined for each object, based on these features. For example, object o dominates o' in the basic partial order, if o dominates o' considering only the attributes in the basic feature class. The dominance graph of each group feature can be easily derived from its contained attributes.

Household: Each original tuple stores six types of annual expenditures (e.g., electricity, gas, phone, etc) of a family and the family class. In addition, the dataset contains the six types of expenditures of the same family on a monthly basis. Therefore, we can group the data by different family class and compute the average total cost per family (including all six types of expenditures) in each quarter for each family class (i.e., four attributes are induced, each corresponding to the average family expenditures in a quarter). We say that family class v dominates family class u if v spends more than or same as u on the average in each of the four quarters, while there exists a quarter where v spends strictly more than u. This way, a partial order can be derived on the family class attribute.

F. PREPROCESSING COST

Here, we present a set of experiments, where we measure the preprocessing overhead by all techniques. These times include the computation of the spanning tree (and the topological order) for SDC$^+$ and TSS and the chain-product embedding for CPS and SCL. They also include the time to generate the corresponding indexes (i.e., R*-trees for SDC$^+$, TSS, and CPS; and the column-store index for SCL). Figures 11(a) and 11(b) plot the elapsed time in log scale as a function of the PO parameters (ω and ρ) of UI data with 2 TO and 2 PO. The time for spanning tree computation (in SDC$^+$ and TSS) and chain-product embedding (in CPS and SCL) are shown at the bottom part of each bar. Observe that chain-product embedding is slightly more expensive, in general, than spanning tree computation, and the costs of these operations are negligible compared to the index construction costs. In addition, generating the corresponding R*-trees is more expensive for SDC$^+$ and CPS, than for TSS. SDC$^+$ may have to generate multiple R*-trees per partial order, while CPS constructs a single multi-dimensional tree. TSS on the other hand, only constructs a one dimensional tree, this is why it has lower cost than the other R*-tree based approaches. SCL has the lowest index-construction cost, because the index is not hierarchical, requiring only one pass over the data to decompose them into columns, while the strata and the links between them can be efficiently created with the help of a hashmap (see Appendix C.1). Figure 11(c) shows how the preprocessing cost is affected by the number of dimensions and the number of total/partial orders in their domains. For each method we plot two lines; one which fixes the number of PO to 2 and varies the number of TO and one that fixes the number of TO to 2 and varies the number of PO. The times only slightly increase with dimensionality and all methods scale similarly. Finally, Figure 11(d) shows that changes in the database size do not affect the relative performance of the methods.

Figure 12 shows the preprocessing costs of all methods on the


(a) Width per PO (ω) (b) Density per PO (ρ)

(c) $|TO|$ or $|PO|$ (d) Cardinality (n)

Figure 11: Preprocessing cost on UI datasets

real data. Chain-product embedding for SCL and CPS is more CPU intensive than spanning tree computation for SDC$^+$ and topological sorting for TSS on larger scale partial orders (i.e., the *Darpa* dataset). However, this time is only a small fraction of the total preprocessing time. For indexing, the trend is similar to that on synthetic data. CPS becomes more expensive than SDC$^+$ for the Darpa dataset, because of the large number of dimensions in the embedding (10 dimensions for 3 partial orders), while only 6 dimensions are derived from the 3 PODs for SDC$^+$. SCL has the lowest indexing cost of all methods.

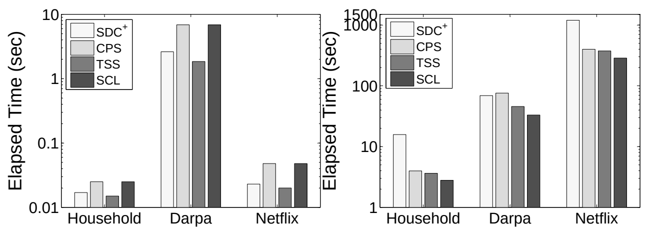

(a) Decomposing POs (b) Indexing Data

Figure 12: Preprocessing of Real Datasets