Efficient Skyline Computation over Low-Cardinality Domains

Michael Morse Jignesh M. Patel H.V. Jagadish
University of Michigan
2260 Hayward Street
Ann Arbor, Michigan, USA
{mmorse, jignesh, jag}@eecs.umich.edu

ABSTRACT

Current skyline evaluation techniques follow a common paradigm that eliminates data elements from skyline consideration by finding other elements in the dataset that dominate them. The performance of such techniques is heavily influenced by the underlying data distribution (i.e. whether the dataset attributes are correlated, independent, or anti-correlated).

In this paper, we propose the Lattice Skyline Algorithm (LS) that is built around a new paradigm for skyline evaluation on datasets with attributes that are drawn from low-cardinality domains. LS continues to apply even if one attribute has high cardinality. Many skyline applications naturally have such data characteristics, and previous skyline methods have not exploited this property. We show that for typical dimensionalities, the complexity of LS is linear in the number of input tuples. Furthermore, we show that the performance of LS is independent of the input data distribution. Finally, we demonstrate through extensive experimentation on both real and synthetic datasets that LS can result in a significant performance advantage over existing techniques.

1. INTRODUCTION

The skyline operator has emerged as an important summarization technique for multi-dimensional datasets. For a dataset D consisting of data points p_1, p_2, ..., p_n, the skyline S is the set of all p_i such that there is no p_j that dominates p_i. p_i is said to *dominate* p_j if p_i is better than p_j in at least one dimension and not worse than p_j in all other dimensions, for a defined comparison function.

An example of the skyline operator in a hotel room selection application is shown in Table 1. In this example, various hotels in a particular city list guest amenities that they contain, such as whether or not they have parking facilities, a swimming pool, and a workout facility for guests. The hotels also list the number of stars that they are rated, and the average price of a room. In this example, a traveler wants to maximize the star rating and boolean-valued amenities of the hotel while minimizing the price. The skyline of this dataset consists of the Soporific Inn, the Drowsy Hotel, and the Celestial Sleep. The Slumber Well is not in the skyline since it has no client amenities and it has a lower rating and costs more than the Soporific Inn. The Nap Motel is not in the skyline because the Soporific Inn also contains a swimming pool, has the same number of stars as the Nap Motel, and costs less.

Hotel Name	Parking Available	Swim. Pool	Workout Center	Star Rating	Price
Slumber Well	F	F	F	$\star$	80
Soporific Inn	F	T	F	$\star\star$	65
Drowsy Hotel	F	F	T	$\star\star$	110
Celestial Sleep	T	T	F	$\star\star\star$	101
Nap Motel	F	T	F	$\star\star$	101

Table 1: A sample hotels dataset.

In this example, the skyline is being computed over a number of domains that have low cardinalities, and only one domain that is unconstrained (the *Price* attribute in Table 1). This dataset characteristic is common in many real applications for several reasons. First, many applications naturally have low cardinality attributes. For example, used car purchase applications often involve the user exploring tradeoffs between the car price (an unconstrained attribute) and several additional attributes with low-cardinality or boolean-valued domains, including the number of doors and the presence or absence of airbags. Second, even seemingly continuous attribute are often naturally searched using a mapping to a low cardinality domain. For example, the car mileage is often mapped to a small number of mileage ranges.

Existing skyline evaluation methods are not designed to exploit the low-cardinality characteristics of such applications, and as a result, are not efficient when used in these cases. The focus of this paper is on developing an efficient skyline algorithm for such applications.

We propose an algorithm called the Lattice Skyline algorithm (LS) that is built around a new paradigm for skyline evaluation. We show that the partial order imposed by the skyline operator over such low-cardinality domains constitutes a *lattice*. We then develop an algorithm that exploits this property and computes the skyline based on the structure of this lattice. The algorithm is very efficient, and for typical dimensionalities has an asymptotic complexity that is linear in the number of input tuples, which can be a big improvement over other techniques. Detailed experimental evaluation comparing LS with existing methods on both real and synthetic datasets shows that in practice LS is indeed significantly more efficient than existing methods.

An additional interesting property of the new lattice-based skyline computation paradigm is that the performance of LS is independent of the underlying data distribution. To understand this property, consider the paradigm used by previous skyline evaluation techniques, such as Block Nested Loops [4] and Sort-First Skyline [9]. These algorithms eliminate data elements from consideration in the skyline by finding other elements in the dataset

VLDB '07, September 23-28, 2007, Vienna, Austria.

that dominate them. The performance of this class of algorithm varies greatly depending on the underlying data distribution; specifically, the performance of these algorithms degrades if the distribution tends towards an anti-correlated distribution. Note that many skyline applications involve datasets in which there is a tradeoff in relative values, which often naturally results in datasets that tend to be anti-correlated. In contrast, LS uses a lattice-structure that is dependent only on the underlying domain characteristics which results in performance that is both predictable and independent of the underlying distribution of the dataset. This property is very desirable, not only from a stability perspective, but also when using the skyline operator in a complex application in which estimates of computational costs can be useful in shaping the user experience (for example in providing progress indicators [8, 18] for complex queries, which has received a lot of attention in recent years).

We acknowledge that previous skyline algorithms which have been designed to be largely independent of the underlying domain characteristics are more general than LS. The generality of these methods implies that they can be applied in any setting. However, we have observed that many skyline applications involve domains with small cardinalities – these cardinalities are either inherently small (such as star ratings for hotels), or can naturally be mapped to low-cardinality domains (such as mileage on a used car). We show that LS produces substantial performance gains for this important class of applications.

The main contributions of this paper are as follows:

1. We develop a new paradigm for skyline computation that is based on constructing a lattice over the underlying domains. We then develop an efficient algorithm that exploits this lattice structure to compute skylines over low-cardinality domains.

2. We show that this method can easily accommodate one unconstrained domain by modifying the lattice-based computation.

3. We show that for low-cardinality datasets of typical skyline dimensionality, the skyline using LS can be evaluated in linear time!

4. We conduct an extensive performance evaluation using both real and synthetic datasets and compare our method with the SFS technique [9] with the LESS optimizations [10], which is currently considered to be the most efficient skyline method that does not require indexing or preprocessing. Our evaluations shows that LS is significantly faster than SFS with the LESS optimizations.

The remainder of this paper is organized as follows: Section 2 discusses related work. In Section 3, we show that the skyline operator over the space of vectors over low-cardinality domains is a lattice, and develop an algorithm for computing skylines using this lattice. In Section 4 we extend the algorithm to accommodate one attribute over an unrestricted domain. In Section 5 we discuss properties of LS and Section 6 presents experimental results. Section 7 discusses applications of LS for discretized attribute domains, and Section 8 contains our concluding remarks.

2. TERMINOLOGY AND RELATED WORK

Terminology: An attribute domain is said to be *low-cardinality* if its value is drawn from a set S=$\{s_1, s_2, ..., s_m\}$ such that the set cardinality m is small. A low-cardinality attribute domain is said to be *totally ordered* if $s_1 < s_2 < ... < s_m$. Skylines usually involve totally ordered attribute domains. Boolean-valued attributes are a special case of totally ordered, low-cardinality attributes. Henceforth, we refer to low-cardinality domains and implicitly assume that they are totally ordered.

Related Work: The skyline is related to several other problems, including maximal vectors [13], the Pareto set, and convex hulls [3]. The skyline was first introduced in the context of database systems in [4]. In this paper, the authors introduce several algorithms for evaluating the skyline, including the block-nested loops (BNL) algorithm, a divide-and-conquer approach, and an indexing technique using B-trees.

The Sort-First Skyline algorithm is proposed in [9], and it is a variant of the BNL algorithm. This technique requires the data to be sorted by a scoring function. Once the data is sorted, the comparison between tuples is simplified since the buffer pool is guaranteed to contain only skyline points. The technique is refined in [10] by eliminating some tuples during the first sort pass with comparisons to a small collection of tuples that fall early in the sort order and combining the final pass of the sort with the first filter pass of the skyline computation. The refined version of the algorithm is called LESS.

Two progressive techniques were proposed in [22]: the Bitmap and the Index techniques. The Bitmap technique operates on skylines over low-cardinality domains, similar to the LS algorithm. The Bitmap technique does not allow one of the attributes to be over an unrestricted domain, so the scope of applications in which it is applicable is more narrow. Bitmap also requires preprocessing, since bitmap indices are required, and the Bitmap technique was also shown to be generally less efficient than the Index technique. Since we are proposing an unindexed technique, we do not compare with either of these indexed techniques; we further discuss our rational for selecting SFS with LESS for comparison in Section 6.

Several techniques using R-trees have been proposed [12,19,20]. Many other problems relating to skyline evaluation have been studied. Techniques to reduce the number of skyline points in high dimensions have also been proposed, including the skyline frequency [7], strong skyline points [25], and the k-dominant skyline [6]. Other techniques to reduce output volume, including Approximately dominating representatives [11] and the k most representative skyline operator [17], have also been studied. Techniques to evaluate skylines in subspaces have been proposed in [24] and [21]. These consider the lattice of dimensional subspaces for skyline evaluation; in contrast, our work views the discrete, well-ordered data space as a lattice and uses that lattice to evaluate the skyline. In [15], a data cube for the dominance relationship is proposed. It uses a lattice structure to develop the D*-tree, which in turn is used to answer several types of dominance queries. However, the dominance relationship is a very different analysis operation than the skyline operation. Also, LS uses a lattice structure on-the-fly to answer skyline queries, as opposed to indexing to evaluate the dominance of a specific point. Skyline evaluation has also been studied in the context of streaming environments in [16, 23] and in the context of partially ordered attribute domains in [5].

3. SKYLINE COMPUTATION FOR LOW-CARDINALITY ATTRIBUTES

Throughout this paper, and without loss of generality, we consider the skyline with the max operator for all attributes. This means that the value T dominates the value F in the boolean case

Figure 1: (a) A Boolean Lattice and (b) the Boolean Lattice with arrows to explicitly indicate the dominance relationship.

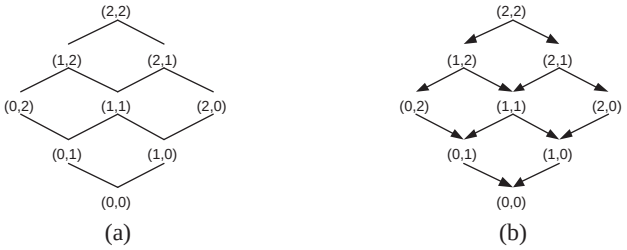

Figure 2: A two-dimensional lattice in which each attribute is drawn from the domain {0,1,2}.

and that larger values dominate smaller ones for low-cardinality and unrestricted attributes.

In this section, we first show that the skyline operator over the space of d-dimensional vectors over low-cardinality domains is a lattice. We then show how this lattice property can be used to develop an efficient skyline algorithm (the Lattice Skyline algorithm). We also give an example of its use and analyze its complexity.

3.1 Skyline and the Low-Cardinality Lattice

The dominance operator '$\prec$' over a dataset defines a partial ordering. (This is easy to see in the dataset in Table 1. The Celestial Sleep dominates the Slumber Well, and hence Celestial Sleep $\prec$ Slumber Well. The ordering is not total since the Celestial Sleep neither dominates nor is dominated by the Soporific Inn).

In this subsection, we show that the partial order that the skyline operator imposes over the space of d dimensional vectors over low-cardinality domains B is a lattice. We let B denote the space of d-dimensional vectors over low-cardinality domains throughout the rest of the paper.

We use the following definition for the lattice of a partially ordered set.

DEFINITION 3.1. *A partially ordered set S with operator '$\prec$' is a lattice if $\forall\ a, b \in S$, the set $\{a, b\}$ has a least upper bound and a greatest lower bound in S.*

We can now use Definition 3.1 to show that the space of vectors B with the skyline operator is a lattice.

THEOREM 3.2. *The space of boolean valued vectors B with the skyline operator '$\prec$' is a lattice.*

PROOF. To show that B with the skyline operator '$\prec$' is a lattice, we must show that each pair $\{x, y\}$ where $x, y \in B$ has (1) a greatest lower bound in B and (2) a least upper bound in B.

Showing (1) involves proving two cases - the case (a) in which x dominates y (or y dominates x) and the case (b) in which x and y are not comparable by the skyline operator.

- CASE 1.a: If x dominates y (y dominates x), then trivially the greatest lower bound q between x and y is y (x).

- CASE 1.b: If x and y are not comparable in the partial order $\prec$, then the greatest lower bound q between x and y is obtained by taking the min between x and y on all dimensions. q is now a lower bound between x and y since in any dimension i, q has a value smaller than or equal to both that of x or y in dimension i, and hence q is dominated by both x and y. q is a greatest lower bound since increasing the value of any attribute a_i on dimension i would no longer result in a lower bound, since the new value of q in dimension i would now be larger than one or both of x or y in that dimension.

Showing (2) also involves proving two cases - (a) in which x dominates y (or y dominates x) and the case (b) in which x and y are not comparable by the skyline operator. This part can be proved in a similar way as above, and is omitted in the interest of space.

□

Since B and the skyline operator are a lattice, we can draw the Hasse diagram for the lattice. The Hasse diagram of B for $d = 3$ in which each low-cardinality attribute is boolean-valued is presented in Figure 1a. In this Figure, the value TTT dominates all other values, so it is at the top of the diagram and it is the upper bound of the set. FFF is dominated by all values so it is the lower bound.

The dominance relationship between elements of B can be further illustrated by adding arrows to the Hasse diagram as shown in Figure 1b. For example, TTF dominates TFF, FTF, and FFF. These are the values in the graph in Figure 1b that are reachable from TTF.

An example Hasse Diagram for a lattice over a two dimensional space in which attribute a_1 is an element of $\{0, 1, 2\}$ and attribute a_2 is also an element of $\{0, 1, 2\}$ is shown in Figure 2a. In Figure 2b, arrows have been added to show the dominance relationship between elements of the lattice.

We now define the concept of an *immediate dominator* of an element of a lattice over B. We let $f(q.a_i)$ denote the number of attribute values in the i^{th} attribute domain that a_i dominates for $q \in B$. For example, in the domain $\{0, 1, 2\}$, value 1 dominates one element.

DEFINITION 3.3. *Let q and q' be elements from B. q is an immediate dominator of q' if and only if q dominates q' and $\sum_{i=1}^{d} f(q.a_i) = \sum_{i=1}^{d} f(q'.a_i) + 1$.*

For example, the immediate dominators of lattice element (1,1) in Figure 2b are (2,1) and (1,2). In this case, $f(1, 1) = 2$ and $f(2, 1) = f(1, 2) = 3$. In general, an element will have d or fewer immediate dominators since an element can only have 1 immediate dominator per dimension. This property of the immediate dominators is used later in the cost analysis of the algorithm.

3.2 Skyline Computation using the Lattice

A dataset D over d low-cardinality attributes does not necessarily contain representatives for each lattice element. For example, the three boolean attributes (Parking Available, Swimming Pool, and Workout Center) in the dataset in Table 1 contains a FTF entry (the Soporific Inn and Nap Motel), but contains no TFF entry.

The method to obtain the skyline of a dataset D consisting of elements of B can be visualized using the Hasse diagram of B. The elements of D that compose the skyline are those in the Hasse diagram that have no path leading to them from another element

Algorithm 1 LS-B: The Skyline for Datasets over Low-Cardinality Domain Attributes.

```
1: Input: Dataset D with n tuples over d low-cardinality attributes, Vector V of size d where V_i is the cardinality of dimension i.
2: Output: A set of skyline points.
3: Let size be the number of entries in the lattice = V_1 * V_2 * ... * V_d.
4: Let the set of designators be {not present, present, dominated}.
5: Let a be an array of designators of size size, initialized to not present.
6: Let F(j) be the one-to-one mapping of j ∈ B to a position in a.
7: for each s ∈ D do
8:    Let l_s be the low-cardinality attribute values of s.
9:    Set a[F(l_s)] to present.
10: end for
11: for t = size − 1 to 0 do
12:    for Each g ∈ immediate dominators of a[t] do
13:       if a[g] = (present or dominated) then
14:          a[t] =dominated
15:       end if
16:    end for
17: end for
18: for each s ∈ D do
19:    Let l_s be the low-cardinality attribute values of s.
20:    if a[F(l_s)] = present then
21:       output s as a skyline point.
22:    end if
23: end for
```

present in D. For example, consider the case in which B is the space of 3 boolean attribute vectors and D consists of four tuples, TTF, FTF, FFT, FFF. FTF is not a skyline value since it is reachable in the diagram in Figure 1b from value $TTF \in D$. Similarly, FFF is reachable from TTF, FTF, and FFT. TTF and FFT are not reachable from any of the values in D, and they are the skyline values.

We can use these observations to develop the LS-B algorithm to find the skyline of a dataset over the space of vectors drawn from low-cardinality domains. Initially, all elements of the lattice of B are marked as *not present* in the dataset. The algorithm then iterates through each tuple t of the dataset D. The element of the lattice that corresponds to t will be marked as *present* (and not yet dominated) in the dataset. After all tuples have been processed, the elements of the lattice that are marked as *present* and which are not reachable by the dominance relationship from any other *present* element of the lattice represent the skyline values. Elements that are present but are reachable by the dominance relationship, and hence are not skyline values, are marked *dominated* to distinguish them from *present* skyline values. The tuples that represent *present* skyline values can then be output with another iterative pass over the dataset. We call the *present, not present,* or *dominated* value of each lattice position the *designator* of that element.

3.3 The LS-B Algorithm

The LS-B algorithm, shown in Algorithm 1, computes the skyline on a dataset D with low-cardinality attribute space B.

In lines 3-5, the algorithm begins by initializing all elements of the array a to *not present*. The size of this array is equal to the product of the domain cardinalities. Each element of the array represents one element of the lattice for B and stores a *designator*.

We let $F(q)$ denote the one-to-one mapping of an element $q \in B$ to a position of the array in line 6. In the boolean case, we can use the binary value of the boolean attributes to determine the array position. For example, if $d = 3$, then element $TFT \in D$ is represented by position 5 of the array a, since the binary equivalent of TFT is $101 = 5$. In the low-cardinality case in our implementation, we choose $F(q)$ to be a linearization of the elements of the lattice, i.e. the ordering becomes $(2,2)$, $(2,1)$, $(2,0)$, $(1,2)$, etc. In lines 7-10, the algorithm iterates over each tuple in D and sets the position in a represented by the value of $q \in D$ to *present*.

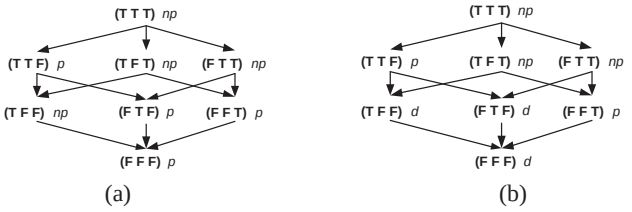

Figure 3: (a) The Boolean Lattice from the example, with present [*p*] and not present [*np*] elements marked. (b) The lattice with dominated values marked as dominated [*d*]. Skyline values are those marked [*p*].

In lines 11-17, the LS-B algorithm iterates through each element of the lattice. If one of the immediate dominators of a lattice position in the Hasse diagram is marked as *present* or *dominated*, indicating that either it is in the skyline or it is dominated by a skyline value, this position in a is marked as *dominated*. The algorithm proceeds through the array beginning at the top of the lattice and ending at the bottom, guaranteeing that the immediate dominators of any element are checked before it.

In lines 18-23, the elements of D are iterated through again, and if the position of a for the boolean-valued attributes of a particular tuple is equal to *present*, then that tuple is a skyline tuple. We note that the second pass through the dataset that outputs the skyline values is not required if the selection predicate in the query contains only attribute values that are in the lattice. In such cases, the query results can be output once skyline values in the lattice are found.

3.4 Example

As an example, suppose a traveler wants to find the skyline of hotels for the boolean valued attributes (availability of parking, swimming pool, and workout center) for the dataset from Table 1. Specifically, the example data is displayed in Table 2.

The lattice element for each element of B is initially marked as *not present*. The LS-B algorithm iterates through each tuple in the input. The lattice position *designator* of each tuple is set to *present*. For example, t_1 is the first tuple considered in the dataset. The *designator* of its boolean attributes, FFF, is set to *present*. The lattice with each lattice value following these actions is displayed in Figure 3a.

Following this, the positions in the lattice that are skyline values are evaluated. The algorithm iterates through the possible values that the space of 3 boolean vectors can obtain. It begins with array position 7 (TTT) and finishes with array position 0 (FFF). For each position, the immediate dominators are checked. The actions for each lattice position, progressing from step 1 to step 8, are shown in Table 3. The lattice following the skyline value evaluation, with each lattice element marked as *np=not present,*

Tuple	Name	Boolean Attribute Values
t_1	Slumber Well	FFF
t_2	Soporific Inn	FTF
t_3	Drowsy Hotel	FFT
t_4	Celestial Sleep	TTF
t_5	Nap Motel	FTF

Table 2: The hotels from the example dataset of Table 1 with the values of their three boolean-valued attributes (parking availability, swimming pool, and workout center).

p=present, or *d=dominated* is shown in Figure 3b. The skyline values are those lattice positions marked as *p*.

The only positions of the lattice that are marked as *present* now are positions TTF and FFT. These tuples are now output as the skyline with another pass through the data.

3.5 Analysis

We now analyze the complexity of the LS-B algorithm for attributes with low-cardinality domains.

THEOREM 3.4. *The complexity of the LS algorithm over a set of low-cardinality attributes is $O(dV + dn)$, where d is the dimensionality, n is the number of data tuples, and V is the product of the cardinalities of the d low-cardinality domains from which the attributes are drawn.*

PROOF. The algorithm makes an initial pass through all n tuples of the data in lines 7-10 of the algorithm. For each tuple, LS-B marks a position in an array as *present* based on the attribute value for each dimension. Since array accesses are $O(1)$, this pass through the data is $O(dn)$.

There are V elements in the lattice. Each is initialized in line 5 of the algorithm. In lines 11-17, each element of the lattice is compared with its immediate dominators, of which there are at most d. We note further that the individual operations in the algorithm are very simple, and that the actual complexity is somewhat better than the asymptotic would suggest. For instance, element $(2, 1)$ of the lattice depicted in Figure 2b has only 1 immediate dominator instead of 2. In short, we expect the algorithm to be efficient in practice, as we show in Section 6. Since there are V total entries in the lattice, each compared with at most d entries, this step is $O(dV)$.

LS-B makes a final pass through the data in lines 19-23, which output the skyline. For each tuple, the algorithm checks an array position based on the attribute value for each dimension to see if its value is *present*. This stage is $O(dn)$. This produces an overall complexity of $O(dV + dn)$ for the algorithm.

□

Analysis: This analysis shows that if n is larger than V, the product of the domain cardinalities of each low-cardinality domain attribute, then the algorithm is linear in n. We expect n to be significantly larger than d for typical skyline datasets (past work has usually experimented with 5-7 dimensions). We also give several examples in Section 6 of low-cardinality datasets in which both skyline computation is important and V is smaller than n. In such cases, the skyline can be evaluated in linear time!

Step	Lattice Pos	D1 (Value)	D2 (Value)	D3 (Value)	Old/New Value
1	TTT	n/a	n/a	n/a	*np* / *np*
2	TTF	TTT (*np*)	n/a	n/a	*p* / *p*
3	TFT	TTT (*np*)	n/a	n/a	*np* / *np*
4	TFF	TTF (*p*)	TFT (*np*)	n/a	*np* / *d*
5	FTT	TTT (*np*)	n/a	n/a	*np* / *np*
6	FTF	TTF (*p*)	FTT (*np*)	n/a	*p* / *d*
7	FFT	TFT (*np*)	FTT (*np*)	n/a	*p* / *p*
8	FFF	TFF (*d*)	FTF (*d*)	FFT (*p*)	*p* / *d*

Table 3: The actions taken during the example, where *p=present, np=not present,* and *d=dominated*. D1, D2, and D3 are the dominators of each position in the example. The value of each such immediate dominator is given in parenthesis.

4. EXTENDING LS TO HANDLE ONE UNRESTRICTED ATTRIBUTE

In this section, we show how to expand the LS-B algorithm to accommodate one attribute u drawn from an unrestricted domain producing the general case LS algorithm. (For example, the domain of u may be the real numbers.)

4.1 Overview

The LS-B algorithm presented in Algorithm 1 marks each lattice position as *present*, *not present*, or *dominated* and uses these designations to find the skyline values. To accommodate an unrestricted domain attribute, in addition to storing the *designator*, each lattice position also stores the best u value in the dataset for that lattice element. For example, if tuples with the lattice value TFF have u attribute values 5, 6, and 7, then the lattice element could store 7 in addition to the *present designator*. In this case, we call 7 the *locally optimal value (*lov*)* of lattice position TFF.

DEFINITION 4.1. *The locally optimal value (*lov*) of an element $q \in B$ is the maximum value of the unrestricted attribute u for any element of a dataset whose low-cardinality attributes are q.*

In the LS-B algorithm presented in the previous section, a lattice element that is marked *present* is in the skyline if none of the lattice positions dominating it are marked as *present*. Now, a lattice element with a lov u is in the skyline if none of the lattice positions dominating it have a lov u' that is better than or equal to u. For example, if TFF has a lov 7 stored in the lattice and TTF has a lov 8, the TFF value is dominated and hence it will not appear in the skyline. In this case, TFF can be marked as dominated. We call the maximum lov contained in an element $q \in B$ and in the elements in B that dominate q the *dominant lattice value (*dlv*)*.

DEFINITION 4.2. *Let A be the set consisting of the locally optimal value of an element $q \in B$ and of the locally optimal values of all $q' \in B$ that dominate q. The dominant lattice value (*dlv*) of q is the maximum value in A.*

Now, a tuple t_i with low-cardinality attribute values q is a skyline value if q is marked *present* and $t_i.u$ is equal to the dlv of q in the lattice. If the *designator* of q is *dominated*, some other lattice entry that dominates q has an lov that is better than or equal to that of q. We can now modify the LS-B algorithm to (1) store the lov for each element of B, (2) find the dlv for each element q of B, and then (3) compare each tuple's u value with the dlv to determine if the tuple is in the skyline.

4.2 The Extended LS Algorithm

Algorithm 2 shows the general LS algorithm, which is an extension of the LS-B Algorithm. Most aspects of the algorithm remain unchanged. The only difference between the two is the values stored for each element of the lattice are different (no longer just storing the *designator* as in the boolean case, but also a value for the unrestricted domain). This information for each lattice element is stored in an array of a defined type L in lines 4 through 6. Each array position stores two pieces of information: (1) the *designator* and (2) the lov of the lattice element.

Each element of the lattice is initialized to *not present* in line 6 of the LS algorithm. In lines 7-15, the algorithm iterates through the elements of the dataset D. If the lattice entry is marked *not present* or the lov is smaller than u, the lattice entry is marked *present* and the lov is updated to u. For example, suppose a dataset consists of data elements over 3 boolean attributes and 1 unrestricted attribute and that the first two data elements of the input

Algorithm 2 LS: The Low-Cardinality Domain Skyline with 1 Unrestricted Attribute Value.

1: **Input:** Dataset D with n tuples over d low-cardinality attributes and 1 unrestricted attribute, Vector V of size d where V_i is the cardinality of dimension i.
2: **Output:** A set of skyline points.
3: Let $size$ be the number of entries in the lattice $= V_1 * V_2 * ... * V_d$.
4: Let the set of *designators* be {*not present, present, dominated*}.
5: Let L be a defined type that contains v, the locally optimal value, and e, an element from the set of *designators*.
6: Let a be an array of type L of size $size$, initialized to *not present*.
7: **for** each $s \in D$ **do**
8: Let $F(j)$ be the one-to-one mapping of $j \in B$ to a position in a.
9: Let l_s be the low-cardinality attribute values of s.
10: Let $pos = F(l_s)$.
11: **if** $a[pos].e =$*not present* or $a[pos].v < s.u$ **then**
12: Set $a[pos].v$ to u.
13: Set $a[pos].e$ to *present*.
14: **end if**
15: **end for**
16: **for** $t = size - 1$ *to* 0 **do**
17: **for** Each $g \in$ immediate dominators of $a[t]$ **do**
18: **if** $a[g].e =$ (*present* or *dominated*) **then**
19: **if** $a[t].e =$*not present* or $a[t].v \leq a[g].v$ **then**
20: $a[t].v = a[g].v$
21: $a[t].e =$*dominated*
22: **end if**
23: **end if**
24: **end for**
25: **end for**
26: **for** each $s \in D$ **do**
27: Let l_s be the low-cardinality attribute values of s.
28: **if** $a[F(l_s)].e =$ *present* and $a[F(l_s)].v = s.u$ **then**
29: output s as a skyline point.
30: **end if**
31: **end for**

are $(T, F, F, 3.2)$ and $(T, F, F, 4.9)$. The TFF lattice position is initially *not present*, indicating that no elements with boolean value TFF have yet been seen in the data. After processing input element $(T, F, F, 3.2)$, TFF is marked as *present* and 3.2 is stored as the lov. After processing $(T, F, F, 4.9)$, the lov is set to 4.9, since 4.9 is the best value for boolean value TFF so far seen.

Now, LS must find the dlv for each element of the lattice. This is done in lines 16-25 of the algorithm. It does this by iterating over the elements of the lattice starting at the top of the lattice and ending with the bottom element. For each such lattice element q, LS checks the dlv values of the immediate dominators of q. The dlv value of q becomes the maximum of the dlv values of the immediate dominators of q marked as *present* or *dominated* and the lov of q. If any of the dlv values of the immediate dominators of q marked as *present* or *dominated* are greater than or equal to the lov of q, q is marked as dominated.

Following this operation, the skyline tuples are those whose low-cardinality value is marked as *present* and have a dlv equal to their u value. In lines 26-31, LS iterates over the elements of D. For each element of D, LS compares the value of u to the dlv for the lattice element. If they are the same and the lattice element is marked *present*, the tuple is an element of the skyline.

4.3 Example

Suppose a traveler is interested in finding the skyline of hotels with regard to the three boolean-valued attributes and the price for the data from Table 1. For this example, we transform the price attribute via a simple flipping function to $200 - price$ so that we are only considering computing the skyline using the *max* operator. Note that this transformation is necessary only to make the example easier to follow by consistent use of the *max* operator. (We could also have negated each value to achieve the same effect.) Our method can easily be adapted to compute the skyline using an arbitrary combination of *max* and *min* operators. The data used in the example with the price transformation is shown in Table 4. We refer to the $200 - price$ value as u.

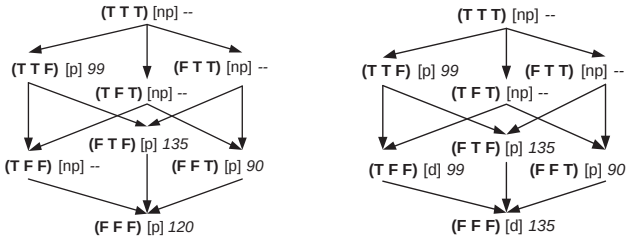

Figure 4: (a) The Boolean Lattice from the example, with [p] present and [np] not present elements marked with their locally optimal values; – means the lattice element is not updated. (b) The lattice with dlvs for each element and with dominated values marked [d]. Skyline values are those marked [p].

The lattice consists of eight entries, one for each boolean value, and each is initialized to *not present*. The LS algorithm now iterates through the input and updates the lattice position for each tuple to the best u value so far present in the data. For example, when LS processes tuple t_1, the lov of FFF is set to 120. Since tuples t_2 and t_5 both contain boolean valued attributes FTF, the lov of FTF is set to 135 (the best u value of either t_2 or t_5). The lattice following these actions is displayed in Figure 4a.

Now, each position in the lattice stores the lov for each lattice element, i.e. the best value that is present in the data *for that element of the lattice*. For example, both t_2 and t_5 have boolean value FTF, but the lov stores only the best value (135) for FTF. LS now finds the dlv for each element of the lattice. For example, FTF has a lov equal to 135, which is better than the lov of FFF. Hence, the FTF element dominates the FFF element, and FFF is marked as *dominated* and its dlv is set to 135.

To find these dominating values, the algorithm iterates through the possible values that the space of 3 boolean vectors can obtain. It begins with TTT and ends with FFF. For each position, the immediate dominators are checked. The actions for each lattice position are shown in Table 5.

The skyline tuples can now be found by iterating over the dataset again. Each tuple t_1-t_5 is compared with its lattice position. If the u value for each tuple is equal to the dlv of the lattice position and that position is marked *present*, that tuple is in the skyline. If the values are not equal or the position is marked *dominated*, then the tuple is not in the skyline. For example, $t_2.u$ is equal to 135 and the dlv of lattice position FTF is 135. FTF is also marked as *present*. Hence, t_2 is in the skyline. However, $t_1.u$ is equal to 120 and the value of FFF's dlv is 135. Moreover, FFF is marked as *dominated*, so t_1 is not in the skyline. The skyline in this example is t_2, t_3 and t_4.

Tuple	Name	Boolean Value	u (200-price) Value
t_1	Slumber Well	FFF	120
t_2	Soporific Inn	FTF	135
t_3	Drowsy Hotel	FFT	90
t_4	Celestial Sleep	TTF	99
t_5	Nap Motel	FTF	99

Table 4: Example data tuples.

Step	Position	Imm. Dom. (Value)	Old/New Value
1	TTT	n/a	[np] – / [np] –
2	TTF	TTT ([np] –)	[p] 99 / [p] 99
3	TFT	TTT ([np] –)	[np] – / [np] –
4	TFF	TTF ([p] 99), TFT ([np] –)	[np] – / [d] 99
5	FTT	TTT ([np] –)	[np] – / [np] –
6	FTF	TTF ([p] 99), FTT ([np] –)	[p] 135 / [p] 135
7	FFT	TFT ([np] –), FTT ([np] –)	[p] 90 / [p] 90
8	FFF	TFF ([d] 99), FTF ([p] 135), FFT ([p] 90)	[p] 120 / [d] 135

Table 5: Example LS actions to find the dlv for each lattice position. Each lattice position is marked [p]=present, [np]=not present, or [d]=dominated with the dlv next to it.

4.4 Analysis

The LS algorithm performs the same sequence of operations as LS-B, with minor differences in the specifics that do not impact the complexity. Hence, the complexity of the LS algorithm for one unrestricted attribute is identical to that of the LS-B algorithm. We omit a formal proof since it is similar to the one presented in Section 3.

5. PROPERTIES OF LS

In the previous section, we showed that LS can have a significant asymptotic complexity advantage over other techniques. In this section, we discuss two properties of LS that are desirable for skyline computation.

1. The performance of LS does not depend on the ordering of the elements of the input.
2. The performance of LS does not depend on the distribution of the input.

The first property is desirable because we want a skyline computation technique to have good performance irrespective of the order of the input elements. For example, the performance of the BNL algorithm of [4] improves significantly if skyline tuples that dominate a large number of data points are present early in the dataset, since this allows BNL to eliminate most of these points in the first elimination pass. On the other hand, if skyline tuples come very late in the dataset order, many passes are needed to eliminate non-skyline points from consideration. SFS [9] addresses this issue by first sorting the data, but requires an expensive sorting operation.

LS achieves the first property because it is intrinsically insensitive to the ordering of the input. No additional costs are incurred such as sorting. For each input element, LS simply reads and writes an element of the lattice. Accessing each element of the lattice has the same fixed cost (an array access), so LS is not sensitive to reorderings of the input elements.

The second property is desirable because we want skyline algorithms to have good performance regardless of whether the input data is correlated, independent, or anti-correlated. Algorithms such as SFS and BNL tend to perform much worse if the input is anti-correlated. The performance of LS does not depend on the input distribution, since finding the skyline values involves the same comparisons with immediate dominators for each element of the lattice irrespective of the dataset distribution. More skyline points may be found if the dataset is anti-correlated, but this also does not result in a difference in performance. This is because during the second pass through the data, each input element is checked with the dlv of the corresponding lattice element to determine if the input element is a skyline point.

6. EXPERIMENTAL EVALUATION

In this section we presents results from an experimental study designed to compare the performance of LS with the best existing method. We have implemented two algorithms: a) our LS algorithm and b) the SFS algorithm [9] with the LESS optimizations described in [10]. (A brief description of the LESS algorithm is presented in Section 2 of this paper.) Throughout this section, we refer to these algorithms simply as LS and LESS, respectively. All methods are implemented in C++. A buffer pool of size 500 pages is used by both implementations for the experiments, and all page requests go through this buffer pool. Page size is set to 4KB for both methods. All experiments are performed on a 1.7GHz Xeon machine running Debian Linux 2.6.

In all experiments, the tuple size is 100 bytes. This tuple size is also used in [10] for their experiments. If the amount of space needed to store the attribute values that the skyline is evaluated over is less than 100 bytes, a random sequence of bits is added to the tuple for padding. This more closely resembles a real database setting in which a projection is applied to the tuples of the skyline that seek information such as that contained in a text field or some other information in addition to the multidimensional skyline values.

The reader will notice that LS requires two scans of the dataset to output the skyline, the first to mark positions in a lattice structure and a second to output skyline points from values derived from the lattice. Our implementation does both of these passes through the dataset for LS, i.e. our LS implementation is outputting not just skyline values but the 100 byte values associated with each skyline tuple. Hence, our comparison with LESS is a fair comparison.

The reason for choosing the LESS algorithm is as follows: LS is a skyline evaluation technique that does not require an index, such as BBS that requires an underlying R-tree, or some other multidimensional index. SFS with the LESS optimizations is currently the best general skyline evaluation technique that also does not require an index to be preconstructed on the data.

Both LS and LESS do not require preprocessing or indexing, which makes them very appealing when the skyline operation is part of a complex query (for example computing the skyline over a subset of the base relation). On the other hand, indexed techniques require precomputing an index, or building an index on-the-fly if one does not exist, which is expensive. To confirm this, we have considered bulk loading an R-tree index on the fly using the R-tree bulk loading technique of [14] and then running BBS [19]. For the datasets that we use in this section, the index construction time is often greater by more than an order of magnitude compared to the LS evaluation time. In the interest of space, we omit these results.

6.1 Datasets

For the datasets, we use both synthetic and real datasets. The use of synthetic datasets allows us to carefully explore the effect of various data characteristics, and is commonly used for skyline evaluation. We generate the synthetic datasets with correlated (CO), independent (IN) and anti-correlated (AC) distributions using the popular skyline dataset generator of [4]. We have modified the generator to generate (a) datasets with d attributes each from low-cardinality domains with domain size of c, and (b) datasets with $d-1$ attributes from low-cardinality domains and 1 attribute from the domain of all real numbers between 0 and 100K.

We generate a number of synthetic datasets by varying three parameters: (1) the data cardinality n, (2) the data dimensionality d, and (3) the number of distinct values for each low-cardinality attribute domain c. Datasets are generated for the CO, IN, and AC distributions by holding two of these three parameters fixed at a default value and varying the third parameter. The parameter settings

Parameter	Settings
d	5, **6**, 7
c	4, 6, **8**, 10, 12
n	100K, 250K, **500K**, 750K, 1M

Table 6: Parameter settings used for varying the dimensionality (d), attribute cardinality (c), and dataset cardinality (n) for the synthetic data experiments, with default parameters shown in bold.

Description	Type	Values	Domain Cardinality
# of Bedrooms	Low-card.	Integer	7
# of Bathrooms	Low-card.	Nearest 1/2 Bath	4
# of Floors	Low-card.	Integer	3
Total Rooms	Low-card.	Integer	10
Contains Garage	Boolean	Yes or No	2
Asphalt Roof	Boolean	Yes or No	2
Colonial Arch.	Boolean	Yes or No	2
Estimated Price	Continuous	Dollar value	nearly 160K

Table 7: Attributes in the Zillow house-price dataset.

used for these three parameters are shown in Table 6, with default parameter settings shown in bold. (The default value of $n = 500K$ is also used in [10]).

We also use two real datasets for our experiments. The first dataset is a house-price information dataset that is obtained from Zillow.com [2]. Zillow lists the number of bedrooms, the number of bathrooms, the estimated price, and other information about houses all over the United States. We obtained a dataset containing more than 160K entries for the local regional area between Yonkers, NY and Stamford, CT. This region corresponds to the area that a New York City commuter might live in north of the city. The dataset contains 8 attributes which are summarized in Table 7. In this dataset, the house price is an unconstrained attribute.

The second real dataset is taken from the Internet Movie Database (IMDB) [1], which contains information about movies and television shows, and ratings of these by actual users. From the IMDB, we have produced a dataset that contains over 161K entries and four attributes. The four attributes are summarized in Table 8. In this dataset, the rating attribute is a value between 0.0 and 10.0 with 1 decimal precision, and the number of reviewers is an unconstrained attribute of the dataset with a range from 0 to 217K.

6.2 Experimental Setup

A buffer pool size of 500 pages is used in all the experiments. For LS, 499 buffer pages are used to store the lattice element entries in an array and 1 page is used to read in the data set. The 499 buffer pool pages are enough for the lattice structure to fit into memory for all tests. For example, for either the CO, IN, or AC synthetic datasets with the default parameters ($d = 6$, $c = 6$) the lattice structure size is $8^5 = 32768$ lattice entries. Each lattice entry uses 34 bits (4 bytes to store the 6^{th} attribute which may be either low-cardinality or from an unrestricted domain, and 2 bits to store the *designator*). Hence, the lattice structure in this case uses 136K of memory (32768*34/8). Note that the buffer pool is of size 500*4K=2000K. Note also that the largest the size of the lattice reaches in these experiments is 8^6*34/8=1088K.

In [10], the authors state that no increase in performance is noticed when setting the EF window size to more than 5 pages. We observed this also in our experiments and even noticed a decrease in performance for some larger EF window sizes. Hence, the EF window size is set to 5 pages in our experiments, which is also done in [10].

Description	Type	Values	Domain Cardinality
Rating	Low-card.	1/10 Increments	101
Color	Boolean	Color or B&W	2
Year	Low-card.	Integer	99
No. of Reviewers	Continuous	# of voters	217K

Table 8: Attributes in the IMDB movie dataset.

6.3 Performance on Synthetic datasets

6.3.1 d-1 Low-Cardinality Attributes and One Continuous Attribute

In this experiment, we evaluate the two algorithms on both correlated, independent, and anti-correlated datasets. In these tests, one attribute is drawn from an unrestricted domain consisting of the set of all real numbers between 0 and 100K and the remaining $d - 1$ attributes are are drawn from low-cardinality domains. In the first test, we vary the dimensionality between 5 and 7 (similar to the performance study of [10]). The results are shown in Figures 5a, 5b, and 5c for the correlated, independent, and anti-correlated datasets, respectively. Figure 5d, shows the number of skyline points for each distribution.

From Figure 5c we observe that LS is an order of magnitude faster than LESS in the AC case. LS is also faster in the independent case for 6 dimensions (about 3X), 7 dimensions (about 4X), and a small advantage for 5 dimensions. In the correlated case, the algorithms perform almost identically for lower dimensions (5 and 6). LESS does achieve an advantage over LS for 7 dimensions in the CO case. Notice that the performance of LS is not varying across distributions, which is expected (see Section 5 for details). The time curve for LS is identical for the CO, IN, and AC distributions, only the scaling in the three graphs is changing. LESS's performance varies with the number of skyline points. The number of skyline points for each distribution is shown in Figure 5d. When the number of skyline points is small (near 10), as in the CO case, LESS performs as well or better than LS. However, when the number of skyline points increases and as the dataset becomes more anti-correlated, LESS requires more computation time as expected. LS has a bigger advantage in the AC case because LESS is not able to eliminate as many tuples with its sort-filter pass as in the IN case. Hence, LESS must perform more comparisons in the AC case.

It is worth noting that the number of skyline points for the 1 unrestricted and $d-1$ low-cardinality domains in Figure 5d never climbs above 4 percent of the 500K dataset size for any of the dimensionalities or distributions. In all other experiments, the number of skyline points for each test is a small percentage of the data (also always less than 4 percent of the dataset size). In other words, low-cardinality domains do not produce a catastrophic case in which nearly the whole dataset is in the skyline.

In the second test, we vary the attribute cardinality between 4 and 12. The results are presented in Figures 6 a, b, and c for the CO, IN, and AC distributions, respectively. Similar to the dimensionality results already presented, LS outperforms LESS by more than an order of magnitude for the AC distribution. For the IN distribution, the performance advantage of LS relative to LESS rises as the domain cardinalities (and hence also the number of skyline points present in the dataset) increases, varying from between 1.5X better when each of the $d - 1$ low-cardinality domains has cardinality 4 to about 2.5X better when the cardinality is 12. For the correlated case, LS and LESS perform about the same when the domain cardinalities are between 4 and 10 while LESS achieves a performance advantage when the domain cardinality reaches 12. This is because the small number of skyline points (similar to the dimensionality

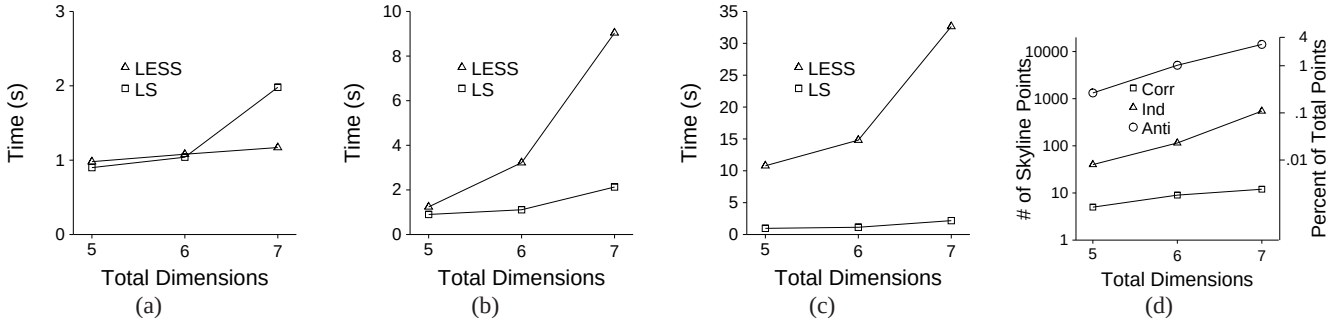

Figure 5: Results for 1 unrestricted and d-1 low-cardinality attributes with varying dimensionality for (a) the CO, (b) the IN, and (c) the AC distributions. (n=500K, c=8) The number of skyline points in each dataset is shown in (d).

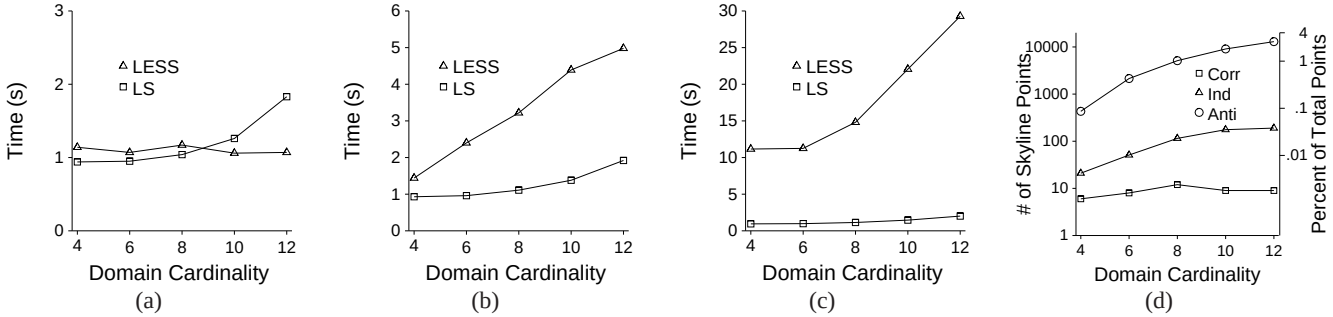

Figure 6: Results for 1 unrestricted and d-1 low-cardinality attributes with varying attribute cardinality for (a) the CO, (b) the IN, and (c) the AC distributions. (n=500K, d=6) The number of skyline points in each dataset is shown in (d).

tests, about 10 total data points) present in the data for the correlated case means that LESS can be very efficient. The number of skyline points for each distribution is shown in Figure 6d. The performance of LESS degrades for the other data distributions as the number of skyline points rises. The performance of LS varies with the sizes of the low-cardinality domains, since larger sizes mean more elements in the lattice. The performance of LS does not vary with the data distribution, but remains constant across each of the three distributions.

In the third test, we vary the input data cardinality between 100K and 1M data tuples. The results for the CO, IN, and AC distribution are presented in Figures 7 a, b, and c, respectively. LS is faster than LESS by an order of magnitude or better for the AC distribution, and about 3X better than LESS on the IN distribution. LS and LESS perform similarly on the CO dataset. The performance of LS decreases approximately linearly as n increases, since the size of n exceeds the cost of the lattice comparisons ($d-1$)*V=164K. for all data sizes except 100K. The performance of LESS degrades faster for the AC distribution because the number of skyline points is greatest for this distribution (see Figure 7d).

6.3.2 d Low-Cardinality Attributes

In this section, we evaluate the performance of LS on datasets that contain d attributes drawn from low-cardinality domains. We again compare LS with LESS and test with synthetically generated datasets from the CO, IN, and AC distributions.

For these experiments, we build the lattice using $d-1$ of the low-cardinality attributes. This allows us to use Algorithm 2 for the skyline evaluation, storing the value of the d^{th} attribute in the lattice. The skyline evaluation using this technique is correct. This results in better performance than building the lattice over all d attributes since the size of the lattice is smaller.

The skyline sizes for the datasets are presented in Figures 8d, 9d, and 10d for varying dimensionality, attribute cardinality, and dataset cardinality, respectively. The reader may notice when observing this data for the correlated and independent data distributions that the number of skyline points decreases as c or d gets larger, which seems counter-intuitive. This occurs because, for these parameter choices, there are a large number of duplicates of the maximal tuple. This repetition occurs because in these cases, the size of the dataspace is smaller than the dataset size. Skyline algorithms cannot simply discard such duplicate skyline values because the skyline query very often is requesting information beyond just skyline values (for example, the name of a hotel) that is unique to each tuple. These duplicates can occur in real datasets. For example, many hotels could offer a workout center, a pool, and free parking. A skyline query for these attributes could then return multiple hotels offering the same features.

In the first test, we vary the dimensionality between 5 and 7 (as in the performance comparison in [10]). The results are shown in Figures 8 a, b, and c for the correlated, independent, and anti-correlated datasets, respectively. LS performs better than LESS by a factor of 5-6X for the AC dataset. On the IN dataset, LS also outperforms LESS when the dimensionality is 6 or 7 (nearly 2X). LS performs about the same as LESS for the correlated dataset for dimensionalities of 5 and 6 and LESS performs better than LS for the correlated dimensionality of 7. The performance advantage for LS for the d low-cardinality attributes is not as great as was achieved in Section 6.3.1 because the number of skyline points is smaller. As is described in Section 6.1, there is a smaller number of skyline points for the correlated case because the number of values expected to be located at the maximum point decreases as the dimensionality increases ($500K/8^5 = 15$ vs. $500K/8^7 > 1$). This trend accounts for the shape of the lines for the number of skyline points in Figure 8d.

In the next experiment, we vary the low-dimensionality cardinal-

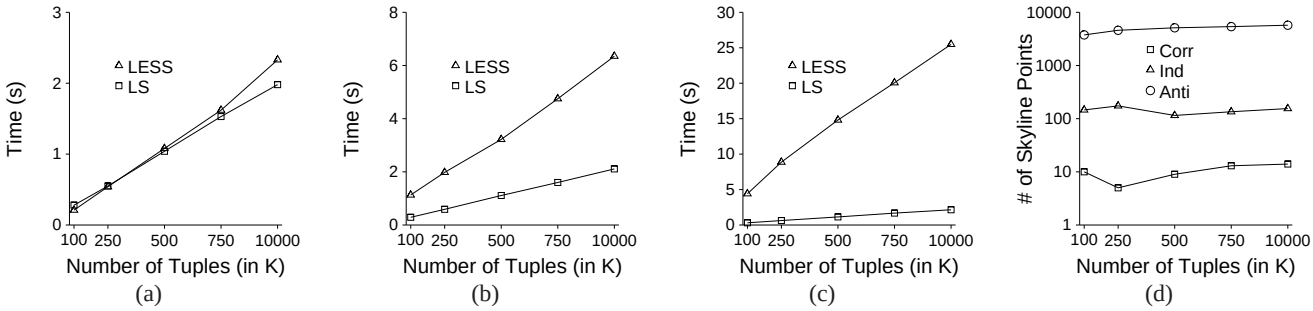

Figure 7: Results for 1 unrestricted and d-1 low-cardinality attributes with varying dataset cardinality for (a) the CO, (b) the IN, and (c) the AC distributions. (c=8, d=6) The number of skyline points in each dataset is shown in (d).

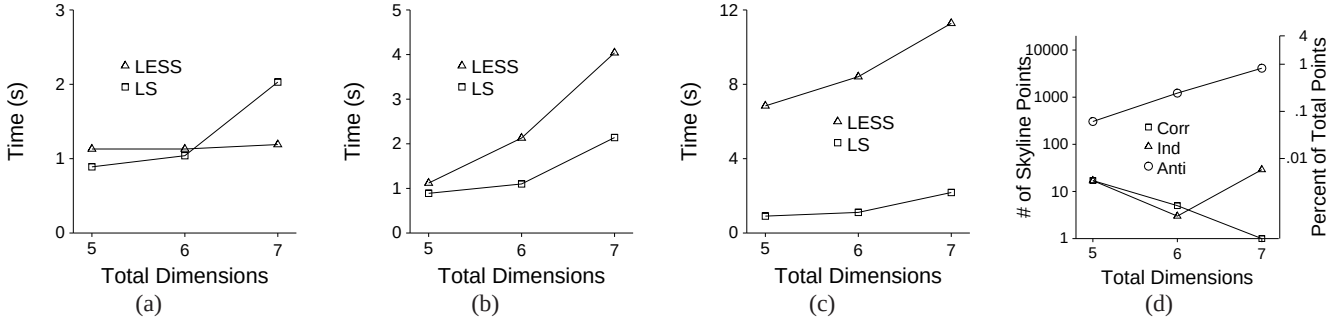

Figure 8: Results for d low-cardinality attributes with varying dimensionality for (a) the CO, (b) the IN, and (c) the AC distributions. (n=500K, c=8) The number of skyline points in each dataset is shown in (d).

ities. The results for this experiment are shown in Figures 9 a, b, and c for the CO, IN, and AC datasets, respectively. LS is faster in the anti-correlated case by nearly an order of magnitude for $c \geq 6$ and is about 4X faster when $c = 4$. LS is faster for the independent case by about 2X when $c \geq 8$ and about $1.5X$ when $c = 4$ or 6. The performance of LESS for the correlated case is better for $c \geq 6$ versus $c = 4$ because the number of skyline points for $c = 4$ is much greater than the other cases. This is because the smaller data space results in more duplicate values (see Section 6.1 for details). Essentially, the performance of LESS for the CO case closely follows the trend set by the skyline size, shown in Figure 9d.

In the third test, we vary the number of data points in each dataset between 100K and 1M. The results are shown in Figures 10 a, b, and c for the CO, IN, and AC datasets respectively. LS is better by nearly an order of magnitude for the AC distribution and by nearly 2X for the IN distribution for cardinalities greater than or equal to 500K. The performance of LESS and LS is similar for the correlated case, for reasons already discussed.

6.4 Performance on Real Datasets

First, we evaluate the performance of LS and LESS on the Zillow dataset. This dataset contains 8 attributes (see Table 7), and our queries compute the skyline with respect to the max operator for the first 7 attributes, since these attributes represent home features that a home buyer may want to maximize. We take the skyline with respect to the min operator for the estimated price of the house.

Using this 8 dimensional dataset, we obtain 5, 6, and 7 dimensional subsets to be used for testing in the following way: for 5 dimensions, we randomly select 4 of the first 7 attributes along with the price attribute (the unrestricted attribute) to obtain 5 attributes in total. We do this 10 times to obtain 10 unique 5 dimensional datasets whose query times are then averaged and reported in this section. A similar operation is done for 6 dimensions. For 7 dimensions, there are seven possible selections of six of the first seven attributes. Each of these seven possible attribute selections, along with the price attribute, make up the 7 dimension attribute subsets.

The performance on the Zillow dataset is shown in Figure 11a. Here, we see that LS outperforms LESS by about an order of magnitude. This behavior is due to the anti-correlated nature of the price attribute with respect to the number of features (bedrooms, bathrooms, etc.) offered by each house. Intuitively speaking, as the number of features rises, the cost of the house also rises. This produces an advantage for LS since its performance is independent of the dataset distribution.

We also evaluate the performance of LS on the IMDB movie dataset. There are four different skyline queries that different users may want to use with this dataset: (1) a query for classic movies that are in black and white, *CBW* (e.g. "Casablanca"), (2) a query for classic movies that are in color, *CC* (e.g. "The Wizard of Oz"), (3) for new movies that are black and white, *NBW* (e.g. "Schindler's List"), and (4) for new movies in color, *NC*. All queries maximize the movie rating and number of reviewers attributes when performing the skyline, to find highly rated movies that have been reviewed by as many voters as possible. Each query either minimizes or maximizes the year and color attributes, depending on whether it is a classic or new movie query for films in color or in black and white. The performance on the IMDB dataset for these four queries is shown in Figure 11b. The performance of LS is about 2X faster than LESS for the *CBW* and *CC* queries and about 1.7X faster on the *NBW* query. LS achieves a modest improvement for the *NC* query. The reason why LESS performs relatively better for the *NC* movie query is that the movie entitled "The Shawshank Redemption" has the largest number of reviews (more than 217K), and one of the best ratings. It, and a few similar movies, dominate a large number of the other entries. Hence, the skyline filter pass of LESS is very effective. There is no similar effect for the other queries.

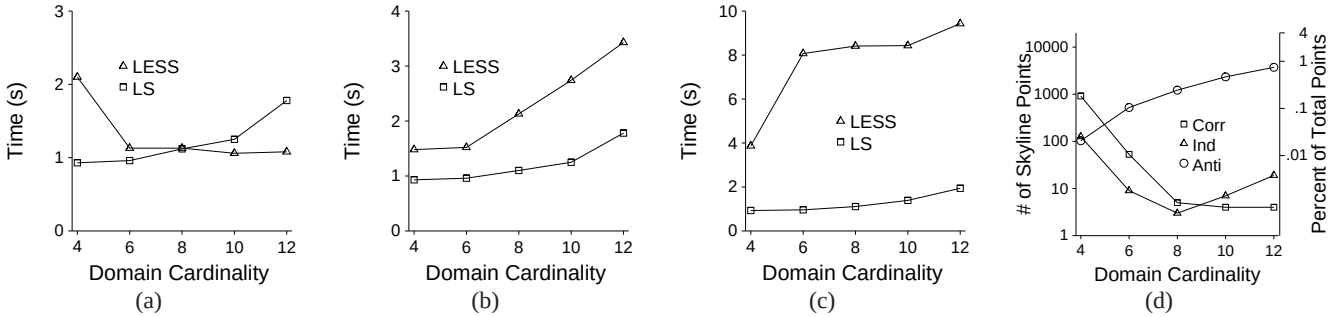

Figure 9: Results for d low-cardinality attributes with varying attribute cardinality for (a) the CO, (b) the IN, and (c) the AC distributions. (n=500K, d=6) The number of skyline points in each dataset is shown in (d).

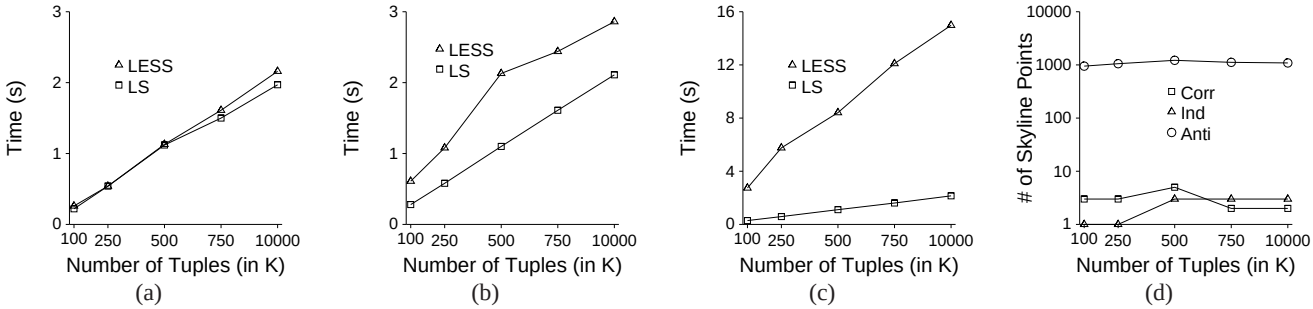

Figure 10: Results for d low-cardinality attributes with varying dataset cardinality for (a) the CO, (b) the IN, and (c) the AC distributions. (c=8, d=6) The number of skyline points in each dataset is shown in (d).

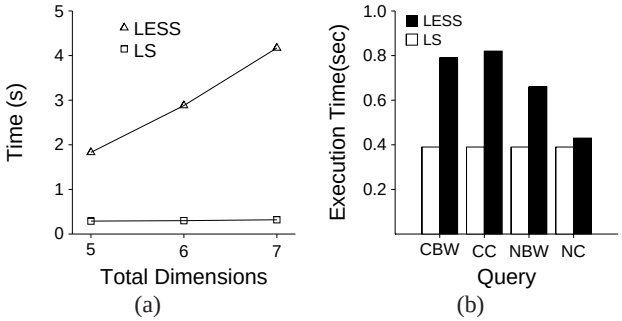

Figure 11: Performance of LS and LESS on (a) the Zillow house-price information dataset, and (b) the IMDB Movie Ratings Dataset.

which means that LESS does more work for these. LS performs the same irrespective of the input. It is also worth noting that the "low" cardinality domains in this example each had cardinalities of approximately 100. Even for this large c value, LS outperforms LESS.

6.5 Performance Summary:

The performance results can be summarized as follows:

- LS typically performs between 5X and an order of magnitude better than LESS on anti-correlated datasets.
- LS performs between about 1.5X and 4X better than LESS on independent datasets.
- LS and LESS perform similarly for the synthetic correlated datasets, with LESS achieving an advantage when $d = 7$ or $c \geq 10$.
- For the real Zillow dataset LS outperforms LESS by an order of magnitude and for the lower dimensional IMDB movie database LS outperforms LESS by up to 2X.

7. DISCRETIZED SKYLINES

In many applications, it may be appropriate to discretize attributes that are over continuous-value domains at coarse granularity. For example, consider the hotel dataset already used as a running example (see Table 1). Now consider what happens if Celestial Sleep were to reduce the price of a room to 66 dollars. The tuples for the Celestial Sleep and Soporific Inn are as follows:

Hotel Name	Parking Available	Swim. Pool	Workout Center	Star Rating	Price
Soporific Inn	F	T	F	$\star\star$	65
Celestial Sleep	T	T	F	$\star\star\star$	66

The Celestial Sleep does not dominate the Soporific Inn since it is still more expensive. Although the Soporific Inn is still in the skyline, including it there in the skyline adds little value since most travelers would prefer to stay at the higher-rated Celestial Sleep for only one extra dollar. This characteristic feature is present in a number of real skyline applications.

As another example, consider the typical car purchase application in which users explore the tradeoffs in price and several additional attributes with low-cardinality or boolean-valued domains. A mileage attribute that may be over a continuous domain can be discretized into a low-cardinality domain. For example, mileage categories might include 30,000-40,000 miles, 40,000-50,000 miles, etc. (Websites such as autotrader.com already allow you to search for cars with mileage under certain increments such as under 75,000). This sort of coarse discretization is often appropriate for continuous valued attributes in many skyline applications because the purpose of skyline computations is often to find candidates for further consideration, and small differences in the value of a continuous attribute can sometimes be ignored.

Definition: We may formally define the discretized skyline if we let $g(q.a_i)$ denote the value of the i^{th} attribute of q in the discretized space.

DEFINITION 7.1. *Element $q \in D$ is said to dominate $q' \in D$ in the discretized space with respect to preference function $\prec_i$ if $\forall i \in d, g(q.a_i) \prec_i g(q'.a_i)$. The discretized skyline A for dataset D is the set of all $p \in D$ such that p is not dominated by any other $q \in D$ in the discretized space.*

This formulation weakens the dominance condition for two purposes. First, it observes that a small advantage in dimension i for q over q' does not necessarily make q more interesting than q' (such as in the case of the Soporific Inn and Celestial Sleep), although discretization may not necessarily solve this problem when values fall close to bucket boundaries. Second, the overall number of skyline points may be reduced, and this is usually desirable.

LS is applicable only to problems with low-cardinality domains, with at most one unconstrained domain. When discretization is appropriate, any continuous attribute can be converted into a low-cardinality attribute. LS can then be applied after such discretization.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed the Lattice Skyline algorithm that is built around a new paradigm for skyline evaluation of datasets whose attributes are drawn from low-cardinality domains. Other skyline evaluation techniques are built around a common paradigm that eliminates points from consideration in the skyline by finding some other dataset element that dominates it. LS uses the structure of the lattice imposed by the skyline operator on the data space of the low cardinality attributes to identify skyline points. This allows LS to have a complexity (for typical skyline dimensionalities and low-cardinality domains) that is linear in the size of the input. It also means that the performance of LS is independent of the data distribution, an important result since the performance of other skyline algorithms typically degrades as the dataset attributes become anti-correlated.

We have shown that LS is applicable to skyline evaluation for three important classes of applications: those in which all attributes come from low-cardinality domains (such as the discretized skyline), those in which attribute domains can be naturally mapped to low-cardinality domains, and those in which one attribute is from an unrestricted domain and all other attributes are from low-cardinality domains. For these applications, LS is also usually significantly faster than existing skyline evaluation methods.

We leave as future work extensions of the LS algorithm when the product of the attribute domain cardinalities is greater than main memory. In such cases, it may be possible to extend LS such that the entire lattice is selectively cached in main memory.

9. ACKNOWLEDGEMENTS

This research was supported by the National Science Foundation (NSF) under grant IIS-0414510. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

10. REFERENCES

[1] The Internet Movie Database. www.imdb.com. Accessed Nov. 6, 2006.

[2] Zillow. www.zillow.com. Accessed Feb. 25, 2007.

[3] C. Böhm and H. Kriegel. Determining the Convex Hull in Large Multidimensional Databases. In *DAWAK*, pages 294–306, 2001.

[4] S. Borzsonyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *ICDE*, pages 421–430, 2001.

[5] C.-Y. Chan, P.-K. Eng, and K.-L. Tan. Stratified Computation of Skylines with Partially-Ordered Domains. In *SIGMOD*, pages 203–214, 2005.

[6] C.-Y. Chan, H. Jagadish, K.-L. Tan, A. Tung, and Z. Zhang. Finding k-Dominant Skylines in High Dimensional Space. In *SIGMOD*, pages 503–514, 2006.

[7] C.-Y. Chan, H. Jagadish, K.-L. Tan, A. Tung, and Z. Zhang. On High Dimensional Skylines. In *EDBT*, pages 478–495, 2006.

[8] S. Chaudhuri, V. R. Narasayya, and R. Ramamurthy. Estimating Progress of Long Running SQL Queries. In *SIGMOD*, pages 803–814, 2004.

[9] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with Presorting. In *ICDE*, pages 717–719, 2003.

[10] P. Godfrey, R. Shipley, and J. Gryz. Maximal Vector Computation in Large Data Sets. In *VLDB*, pages 229–240, 2005.

[11] V. Koltun and C. H. Papadimitriou. Approximately Dominating Representatives. *Theor. Comput. Sci.*, 371(3):148–154, 2007.

[12] D. Kossman, F. Ramsak, and S. Rost. Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In *VLDB*, pages 275–286, 2002.

[13] H. Kung, F. Luccio, and F. Preparata. On finding the maxima of a set of vectors. *Journal of the ACM*, 22(4):469–476, 1975.

[14] S. T. Leutenegger, J. M. Edgington, and M. A. Lopez. STR: A Simple and Efficient Algorithm for R-Tree Packing. In *ICDE*, pages 497–506, 1997.

[15] C. Li, B. Ooi, A. Tung, and S. Wang. DADA: A Data Cube for Dominant Relationship Analysis. In *SIGMOD*, pages 659–670, 2006.

[16] X. Lin, Y. Yuan, W. Wang, and H. Lu. Stabbing the Sky: Efficient Skyline Computation over Sliding Windows. In *ICDE*, pages 502–513, 2005.

[17] X. Lin, Y. Yuan, Q. Zhang, and Y. Zhang. Selecting Stars: the k Most Representitive Skyline Operator. In *ICDE*, 2007.

[18] G. Luo, J. F. Naughton, C. Ellmann, and M. Watzke. Toward a Progress Indicator for Database Queries. In *SIGMOD*, pages 791–802, 2004.

[19] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An Optimal and Progressive Algorithm for Skyline Queries. In *SIGMOD*, pages 467–478, 2003.

[20] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive Skyline Computation in Database Systems. *ACM Trans. Database Syst.*, 30(1):41–82, 2005.

[21] J. Pei, W. Jin, M. Easter, and Y. Tao. Catching the Best View of Skyline: A Semantic Approach Based on Decisive Subspaces. In *VLDB*, pages 253–264, 2005.

[22] K.-L. Tan, P. Eng, and B. Ooi. Efficient Progressive Skyline Computation. In *VLDB*, pages 301–310, 2001.

[23] Y. Tao and D. Papadias. Maintaining Sliding Window Skylines on Data Streams. *IEEE Transactions on Knowledge and Data Engineering*, 18(3):377–391, 2006.

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

[25] Z. Zhang, X. Guo, H. Lu, A. Tung, and N. Wang. Discovering Strong Skyline Points in High Dimensional Spaces. In *EDBT*, pages 478–495, 2006.