Efficient Algorithms for Similarity and Skyline Summary on Multidimensional Datasets

by

Michael David Morse

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2007

For Bethany, my wife, and James, my son.

ACKNOWLEDGEMENTS

A great many people have contributed to my thesis. First, I would like to thank my advisor, Jignesh M. Patel. Jignesh is an outstanding advisor and an outstanding person. I have learned a great deal about research, computer science, databases, and a host of other things from him. I feel extremely fortunate to count myself among his students.

Professor Bill Grosky has mentored me on a number of occasions. He has helped me to obtain funding, develop a number of ideas into competent research, both for this thesis and for other research areas, and learn a few details of Japanese culture.

Professor H.V. Jagadish and Professor Tim Gordon both contributed their time, energy, and experience towards my thesis. I have gained invaluable insight into more than a few of my research problems from conversations with Professor Jagadish and he has been extremely generous in his advise and help. I sincerely thank them.

I have learned a great deal, some useful information and some useless, from my peers in the database lab at Michigan. Adriane Chapman, Jason Yun Chen, Magesh Jayapandian, Dr. Sandeep Tata, Yuanyuan Tian, and Dr. Cong Yu all deserve special thanks in that regard. Thanks also to Dr. Shurug Al-Khalifa, Aaron Elkiss, Prasad Chakka, Dr. Rich Hankins, You Jung Kim, Willis Lang, Dr. Yunyao Li, Bin Liu, Andrew McClory, Dr. Thomas Nadeau, Arnab Nandi, Andrew Nierman, Dr. Kanda Runapongsa, Neamat el Tazi, Nuwee Wiwatwattana, and Dr. Yuquing Wu for making the database lab feel like home. Thanks also to Stephen Reger for help with so many university logistics.

My family has provided me much support throughout my life. My special thanks to my parents for raising me, providing me with an outstanding education, and encouraging me along the way. Special thanks to Joanne for enduring me as an older brother.

Finally, my thanks goes to my wife, Bethany. She has been extremely supportive of my PhD study and work and in all other areas of life. Without her, this thesis would not be possible. My thanks also to my son, James, who, in his first year of life, made sure that I, in my final year of graduate study, didn’t start to sleep too much.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Efficient management of large multidimensional datasets has attracted much attention in the database research community. Such large multidimensional datasets are common and efficient algorithms are needed for analyzing these data sets for a variety of applications. In this thesis, we focus our study on two very common classes of analysis: similarity and skyline summarization. We first focus on similarity when one of the dimensions in the multidimensional dataset is temporal. We then develop algorithms for evaluating skyline summaries effectively for both temporal and low-cardinality attribute domain datasets and propose different methods for improving the effectiveness of the skyline summary operation.

This thesis begins by studying similarity measures for time-series datasets and efficient algorithms for time-series similarity evaluation. The first contribution of this thesis is a new algorithm, called the Fast Time Series Evaluation (FTSE) method, which can be used to evaluate similarity methods whose matching criteria is bounded by a specified ϵ threshold value. We then show that FTSE can be used in a framework that can evaluate a rich range of ϵ threshold-based scoring techniques which we call the Sequence Weighted Alignment (Swale) method.

The second contribution of this thesis is the development of a new time-interval skyline operator, which continuously computes the current skyline over a data stream. We present a new algorithm called *LookOut* for evaluating such queries efficiently, and empirically demonstrate the scalability of this algorithm. In addition, we also examine the effect of the underlying spatial index structure when evaluating skylines. Whereas previous work on skyline computations have only considered using the R*-tree index structure, we show that for skyline computations using an underlying quadtree has significant performance benefits over an R*-tree index.

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. The third contribution of this thesis is a novel technique called 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.

The utility of the skyline as a data summarization technique is often diminished by the shear volume of points in the skyline The final contribution of this thesis is a novel scheme called the Skyline Point Ordering (SPO) which remedies the skyline volume problem by ranking the elements of the skyline based on their importance to the skyline summary, allowing for the most important skyline points to appear first in the skyline result set and providing monotonic top-k skyline queries that simplify the skyline results. We describe two new algorithms, the Skyline First (SF) and the Coverage First (CF), for ranking the skyline points in a dataset based on their summarization importance.

Collectively, the techniques described in this thesis present efficient methods for two common and computationally intensive analysis operations on large multidimensional datasets.

CHAPTER I

Introduction

Driven by many emerging applications, database management systems are increasingly required to provide efficient methods for analyzing large multidimensional datasets. Efficient algorithms to query such datasets are important because the volume of data being managed is typically very large and grows rapidly over time. Accurate techniques to mine and summarize such datasets are also a necessity because the multidimensional nature of the data makes analysis by humans difficult.

One special and common case of multidimensional datasets occurs when one or more dimensions vary with time. For example, scientific datasets are often very large and fit into this multidimensional, time-varying category. As another example, consider buoy sensor data that is used to track ocean currents and obtain weather readings for locations on the surface of the ocean. Yet another example is current hurricane tracking measurements that can be compared with past storm movements to obtain accurate forcasts. In other application areas, the GPS trails of moving objects vary in time and can also generate large datasets. In all of these cases, identifying similar patterns between two time-varying dataset examples is a critical operation. Central to the identification of similar patterns are the similarity measures used to classify and cluster datasets and the methods used to evaluate those similarity measures.

Data summarization techniques have been studied in earnest in the realm of Relational Database Management Systems. Common data summarization methods include finding the minimum or maximum value from a dataset, finding a median value, or finding an average over a set of values. Summarization methods are important for database systems because the volume of data being managed is large. This data volume makes data summarization a necessity.

A new data summarization technique that has recently emerged for multidimensional datasets is the *skyline operator*. Unlike some other summary techniques that consider each dimension of the data in isolation, the skyline concerns itself with multiattribute summarization.

The skyline operator is an elegant summary method over multi-dimensional data sets [43]. Given a data set P containing data points p_1, p_2, ..., p_n, the skyline of P is the set of all p_i in P such that no p_j *dominates* p_i. A commonly cited example for the use of a skyline operator is assisting a tourist in choosing a set of *interesting* hotels from a larger set of candidate hotels. Each hotel is identified by two attributes: a distance from a specific point (such as a location on a beach), and the price for the hotel. To assist a tourist in narrowing down the choices, the skyline operator can be used to find the set of all hotels that are not dominated by another hotel. Hotel a *dominates* hotel b if a is at least as close as b and at least as cheap as b, and offers either a better price, or is closer, or both compared to b. Figure 1.1 shows an example data set and the corresponding skyline; the distance of the hotel from the beach is shown on the x axis and the hotel price is plotted along the y axis. The skyline is the set of points *a, c, d, i,* and *j*.

The skyline can be generalized to multi-dimensional space where a point a dominates another point b if it is as good or better than b in all dimensions, and is better than b in at least one dimension. Implicit in this definition of the skyline operation is the notion of

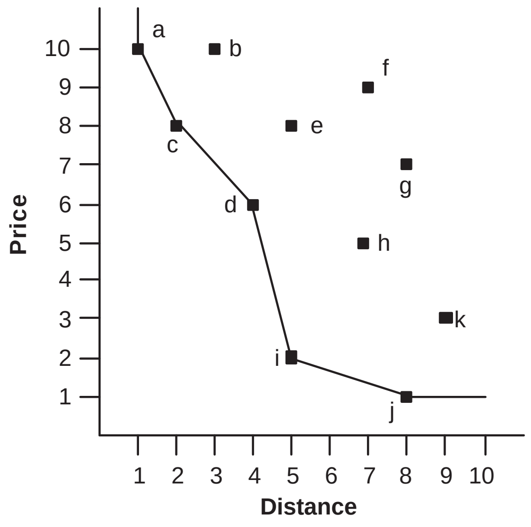


Figure 1.1: Example data set and its skyline.

comparing the *goodness* along each dimension. A common function for determining this property is to use the *min* function. However, skyline computation can easily be extended to consider other functions, such as *max*.

The skyline operator is commonly called the *Pareto set* or the set of *maximal vectors* for a given dataset [24]. The interested reader will note that the three problems are identical to one another; finding the Pareto set of a dataset, finding its maximal vectors, and finding its skyline summary are identical operations and the three resulting data subsets are identical to one another. In a database context, this summarization technique is called the skyline [10].

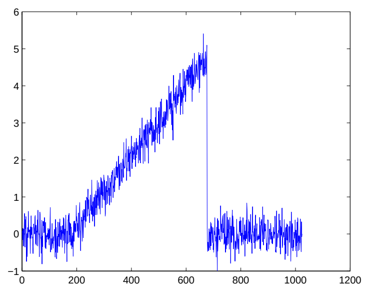
Many of the application areas in which the skyline operator has proven effective also vary in time in at least one dimension. For example, in online settings the price of various commodities changes at least daily. These changing data values form a time-series. This necessitates finding not only efficient algorithms for the evaluation of skylines, but also more efficient techinques for managing temporal and time-series data.

This thesis develops efficient algorithms for similarity measures for multidimensional datasets that vary with time, and methods for processing and producing effective skyline summaries in multidimensional datasets that both vary in time and contain low-cardinality attribute domains. This includes developing algorithms for both finding skylines effectively in the presence of a temporal dimension and maintaining the skyline when data values change over time.

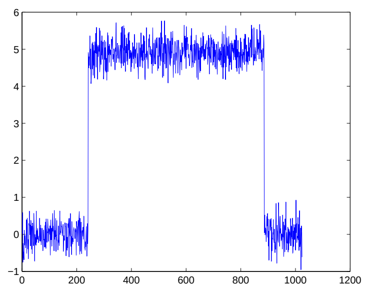
1.1 Contributions

There are many applications for the classification and clustering of time-series, which makes developing effective and efficient measures for the comparison of time-series very important. Identifying similar patterns is a crucial operation in time-series datasets. For example, consider the three time-series examples shown in Figure 1.2. These examples come from the popular Cylinder-Bell-Funnel dataset [2]. Separating the examples of the cylinder from the bell or the funnel for a human is a trivial (but expensive) task; automated techniques have error rates that vary between 15 percent for a Euclidean distance metric and 4 percent for the Dynamic Time Warping (DTW) technique [2]. Not surprisingly, the more accurate techniques such as DTW are also more expensive to evaluate. Our contribution to time-series clustering and classification is two-fold in chapter 2. First, we present the Fast Time-Series Evaluation (FTSE) technique which can evaluate sophisticated techniques quickly. Second, we present a novel scoring method called the Sequence Weighted Alignment that can use FTSE to compare time-series both accurately and quickly.

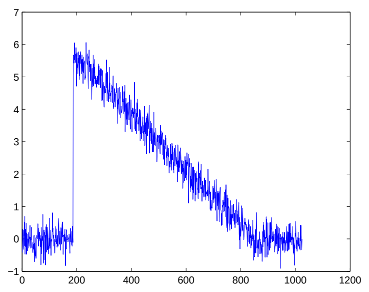
In a number of emerging streaming applications, the data values that are produced have an associated time interval for which they are *valid*. A useful computation over such streaming data is to produce a continuous and valid *skyline* summary. Previous work on skyline algorithms have only focused on evaluating skylines over static data sets, and there are no known algorithms for skyline computation in the continuous setting. In this paper, we introduce the *continuous time-interval skyline* operator, which continuously computes



(a)



(b)



(c)

Figure 1.2: Three example time-series from the Cylinder-Bell-Funnel dataset, depicting (a) a bell, (b) a cylinder, and (c) a funnel.

the current skyline over a data stream. We present a new algorithm called *LookOut* for evaluating such queries efficiently, and empirically demonstrate the scalability of this algorithm. In addition, we also examine the effect of the underlying spatial index structure when evaluating skylines. Whereas previous work on skyline computations have only considered using the R*-tree index structure, we show that for skyline computations using an underlying quadtree has significant performance benefits over an R*-tree index. The details of $LookOut$ are provided in Chapter 2.

The current generation of skyline evaluation methods, including the LookOut technique, follow a common paradigm that removes elements of a dataset D from temporal skyline consideration by finding other elements in D that dominate them, both spatially and with respect to the temporal dimension(s). The distribution of the underlying dataset D heavily influences the performance of the methods that fall into this paradigm. The third contribution of this thesis is a novel technique called the Lattice Skyline Algorithm (LS) that uses a new paradigm to find the skyline for datasets with attributes that are drawn from low-cardinality domains. We show that many temporal skyline applications have such low-cardinality domain 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.

The utility of the skyline as a data summarization technique is often diminished by the shear volume of skyline points, particularly if the dataset is anti-correlated, of high dimensionality, or both. The final contribution of this thesis is a novel scheme called the Skyline Point Ordering (SPO) to rank the elements of the skyline based on their importance to the skyline summary. Skyline point ranking is important for two main reasons. First, it returns the most important skyline points first in the skyline result set, as opposed to other methods that do not specify any ordering. Second, it allows for monotonic top-k skyline queries that simplify the skyline results by only providing k results. We describe two new algorithms, the Skyline First (SF) and the Coverage First (CF), for ranking the skyline points in a dataset based on their summarization importance and expand this discussion to a ranking of temporal skyline data points.

Collectively, this thesis provides efficient algorithms for similarity and skyline evaluation on large multidimensional datasets. datasets In summary, the four main contributions of the thesis are first, the FTSE and Swale methods for the similarity of multidimensional time-series datasets, second, the *LookOut* algorithm for evaluating skylines time-interval continuous skylines, third, the LS method for finding skylines in datasets that have low-cardinality attribute domains, and fourth, the SPO for producing a ranked skyline summary set for multidimensional datasets.

1.2 Thesis Outline

The remainder of this thesis is organized as follows: Chapter II presents the description of the FTSE algorithm and the Sequence Weighted Alignment scoring method and contains a detailed experimental study of these methods compared to other techniques. Chapter III introduces the Time-Interval Continuous Skyline operator and the LookOut algorithm for evaluation of the tics operator on temporal datasets. Chapter IV presents the Lattice Skyline algorithm for evaluation of the skyline for datasets whose attributes are drawn from low-cardinality domains. Chapter V presents the Skyline Point Ordering for the ranking of skyline points, introduces two algorithms for the evaluation of the Skyline Point Ordering for datasets, and evaluates these techniques in a detailed experimental study. Finally, Chapter VI presents our conclusions and directions for future work.

CHAPTER II

A Fast Time Series Evaluation Technique

2.1 Introduction

Techniques for evaluating the similarity between time series datasets have long been of interest to the database community. New location-based applications that generate time series location trails (called trajectories) have also fueled interest in this topic since time series simularity methods can be used for computing trajectory similarity. One of the critical research issues with time series analysis is the choice of distance function to capture the notion of similarity between two sequences. Past research in this area has produced a number of distance measures, which can be divided into two classes. The first class includes functions based on the L1 and L2 norms. Examples of functions in this class are Dynamic Time Warping (DTW) [8] and Edit Distance with Real Penalty (ERP) [17]. The second class of distance functions includes methods that compute a similarity score based on a matching threshold ϵ. Examples of this class of functions are the Longest Common Subsequence (LCSS) [78], and the Edit Distance on Real Sequence (EDR) [18]. Previous research [18, 78] has demonstrated that this second class of methods is robust in the presence of noise and time shifting.

All of the advanced similarity techniques mentioned above rely on dynamic programming for their evaluation. Dynamic programming requires that each element of one time series be compared with each element of the other; this evaluation is slow. The research community has thus developed indexing techniques such as [18, 21, 34, 38, 82] that use an index to quickly produce a superset of the desired results. However, these indexing techniques still require a refinement step that must perform the dynamic programming evaluation on elements of the superset. Furthermore, time series clustering has also been studied [18, 35, 78], and these clustering techniques require pairwise comparison of *all* time series in the dataset, which means that indexing methods cannot be used to speed up clustering applications.

To address this problem, a number of techniques have been developed that impose restrictions on the warping length of the dynamic programming evaluation. The Sakoe-Chiba band, studied in [68], uses a sliding window of fixed length to narrow the number of elements that are compared between two time series. The Itakura Parallelogram, studied in [31], also limits the number of comparisons to accomplish a similar effect as the Sakoe-Chiba band. These techniques that constrain the warping factor are faster, but at the expense of ignoring sequence matches that fall outside of the sliding window. If the best sequence match between two time series falls outside of the restricted search area, then these techniques will not find it.

In this chapter, we propose a novel technique to evaluate the second class of time series comparison functions that compute a similarity score based on an ϵ matching threshold. The popular LCSS and EDR comparison functions belong to this class and can directly benefit from our new evaluation technique. This technique, called the **F**ast **T**ime **S**eries **E**valuation (**FTSE**), is not based around the dynamic programming paradigm nor is it an approximation (i.e. it computes the actual exact similarity measure). Using a number of experiments on real datasets, we show that FTSE is nearly an order of magnitude faster than the traditional dynamic programming-style of similarity computation. In addition, we show that *FTSE is also faster than popular warp-restricting techniques by a factor of 2-3, while providing an exact answer.*

We show that FTSE can evaluate a broader range of ϵ threshold-based scoring techniques and not just LCSS and EDR. Motivated by FTSE's broader ability, we propose the **S**equence **W**eighted **AL**ignm**E**nt (**Swale**) scoring model that extends ϵ threshold based scoring techniques to include arbitrary match rewards and gap penalties. We also conduct an extensive evaluation comparing Swale with popular existing methods, including DTW, ERP, LCSS, and EDR and show that *Swale is generally more accurate than these existing methods*.

The remainder of this chapter is organized as follows: Section 2 discusses the terminology that is used in the rest of the chapter. Section 3 discusses related work and Section 4 describes the FTSE algorithm. Section 5 introduces the Swale similarity scoring method and Section 6 presents experimental results. Finally, Section 7 presents our conclusions.

2.2 Terminology

Existing similarity measures such as LCSS, DTW, and EDR assume that time is discrete. For simplicity and without loss of generality, we make these same assumptions here. Formally, the time series data type T is defined as a sequence of pairs *T* = (p_1, t_1), (p_2, t_2), ... , (p_n, t_n), where each p_i is a data point in a d-dimensional data space, and each t_i is the time at which p_i occurs. Each t_i is strictly greater than each t_{i-1}, and the sampling rate of any two time series is equivalent. Other symbols and definitions used in this chapter are shown in Table 2.1.

Time series datasets are usually normalized before being compared. We follow the normalization scheme for time series data described in [25]. Specifically, for S of length n, let the mean of the data in dimension d be μ_d and let the standard deviation be σ_d. Then, to obtain the normalized data $N(S)$, we can evaluate $\forall\, i\ \in\ n:\ s_{i,d} = (s_{i,d} - \mu_d)/\sigma_d$ on all elements of S. This process is repeated for all dimensions. In this chapter, all data is normalized, and we use S to stand for $N(S)$, unless stated otherwise.

2.3 Related Work

There are several existing techniques for measuring the similarity between different time series. The Euclidean measure sums the Euclidean distance between points in each time series. For example, in two dimensions the Euclidean distance is computed as: $\sqrt{\sum_{i=1}^{n}((r_{i,x} - s_{i,x})^2 + (r_{i,y} - s_{i,y})^2)}$. This measure can be used only if the two time series are of equal length, or if some length normalization technique is applied. More sophisticated similarity measures include Dynamic Time Warping (DTW) [8], Edit distance with Real Penalty (ERP) [17], the Longest Common Subsequence (LCSS) [78], and Edit Distance on Real sequences (EDR) [18]. These measures are summarized in Table 2.2.

DTW was first introduced to the database community in [8]. DTW between two time series does not require the two series to be of the same length, and it allows for time shifting between the two time series by repeating elements. ERP [17] creates g, a constant value for the cost of a gap in the time series, and uses the L1 distance norm as the cost between elements. The LCSS technique introduces a threshold value, ϵ, that allows the scoring technique to handle noise. If two data elements are within a distance of ϵ in each dimension, then the two elements are considered to match, and are given a match reward of 1. If they exceed the ϵ threshold in some dimension, then they fail to match, and no

Symbol	Definition
$R,\ S$	Time series $(r_1, ..., r_m)$ and $(s_1, ..., s_n)$.
r_i	The i^{th} element of R.
$Rest(R)$	R with the first element removed.
M^d	d dimensional MBR.
$M^{d1},\ M^{d2}$	Lower and upper bounds of M

Table 2.1: Symbols and definitions.

		Definition	
DTW(R,S)	=	0	if $m = n = 0$
		∞	if $m = 0$ or $n = 0$
		$dist(r_1, s_1) + min\{DTW(Rest(R), Rest(S)),$ $DTW(Rest(R), S), DTW(R, Rest(S))\}$	otherwise
ERP(R,S)	=	$\sum_1^n dist(s_i, g),\ \ \sum_1^m dist(r_i, g)$	if $m = 0,$ if $n = 0$
		$min\{ERP(Rest(R), Rest(S)) + dist(r_1, s_1)$ $ERP(Rest(R), S) + dist(r_1, g),$ $ERP(R, Rest(S)) + dist(s_1, g)\}$	otherwise
LCSS(R,S)	=	0	if $m = 0$ or $n = 0$
		$LCSS(Rest(R), Rest(S)) + 1$	if $\forall d, \lvert r_{d,1} - s_{d,1}\rvert \leq \epsilon$
		$\max\{LCSS(Rest(R), S), LCSS(R, Rest(S))\}$	otherwise
EDR(R,S)	=	$n,\ m$	if $m = 0,$ if $n = 0$
		$\min\{EDR(Rest(R), Rest(S))+$subcost, $EDR(Rest(R), S) + 1, EDR(R, Rest(S)) + 1\}$	otherwise

Table 2.2: Distance Functions: $dist(r_i, s_i) =$ L1 or L2 norm; subcost$=0$ if $|r_{1,t} - s_{1,t}| \leq \epsilon$, else subcost$= 1$.

reward is issued. The EDR [18] technique uses gap and mismatch penalties. It also seeks to minimize the score (so that a score closer to 0 represents a better match).

In [5], the authors use the Euclidean distance to measure similarity in time series datasets. The Discrete Fourier Transform is used to produce features that are then indexed in an R-tree. Dimensionality reduction is also studied in [15, 34, 38, 42, 63, 82]. Indexing is also studied in [21], which proposes a generic method built around lower bounding to guarantee no false dismissals. Indexing methods for DTW have been the focus of several papers including [33, 39, 69, 83, 86]. Indexing for LCSS [77] and EDR [18] has also been studied. In this chapter, our focus is not on specific indexing methods, but on the design of robust similarity measures, and efficient evaluation of the similarity function. We note that our work is complementary to these indexing methods, since the indexing methods still need to perform a refinement step that must evaluate the similarity function. Traditionally, previous work has not focused on this refinement cost, which can be substantial. Previous works employ a dynamic programming (DP) method for evaluating the similarity function, which is expensive, especially for long sequences. In other words, FTSE can be used to boost the performance of existing LCSS or EDR-based indexing methods since it is faster than traditional DP methods for the refinement step.

The Sakoe-Chiba Band [68] and Itakura Parallelogram [31] are both estimation techniques for restricting the amount of time warping to estimate the DTW score between two sequences. A restriction technique similar to the Sakoe-Chiba Band is described for LCSS in [78] and the R-K Band estimate is described in [65].

Time series may be clustered using compression techniques [20, 36]. We do not compare our algorithms with these techniques because of their inapplicability for clustering short time series.

The FTSE algorithm that we propose bears similarity to the Hunt-Szymanski algorithm [30, 46] for finding the longest common subsequence between two sequences. However, Hunt-Syzmanski is only concerned with string sequences (and not time series), supports only a limited string edit-distance model, and does none of the grid matching that FTSE does to identify matching elements between time series (see Section 2.4).

A more closely set of related work is concerned with clustering of trajectory datasets (such as [18, 35, 78, 81]). In fact, a commonly established way of evaluating the effectiveness of trajectory similarity measures is to use it for clustering, and then evaluate the quality of the clusters that are generated [18, 35, 78]. Common clustering methods such as complete linkage are often used for trajectory data analysis [18, 35, 78], and these methods require that each trajectory in the dataset be compared to every other trajectory. Essentially, for a dataset of size s, this requires approximately s^2 comparisons. As we show in this chapter for such problems, not only is the Swale scoring method more effective, but the FTSE technique is also faster than the existing methods.

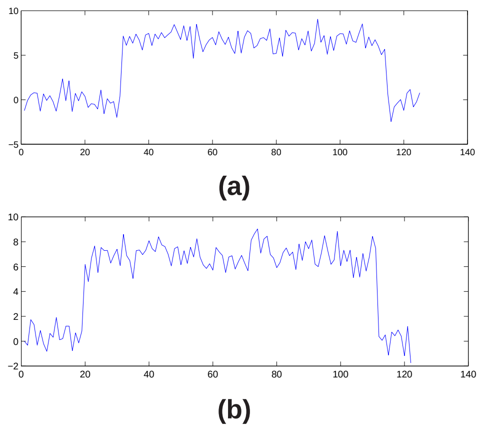


Figure 2.1: Two time series examples of the cylinder class from the Cylinder-Bell-Funnel Dataset.

2.4 Fast Time Series Evaluation

In this section, we introduce the FTSE algorithm. In order to better understand why FTSE is faster than dynamic programming, we first discuss dynamic programming and its shortcomings for evaluating ϵ based comparison functions. We then provide an overview of the FTSE algorithm. We also discuss its operation for LCSS and EDR, provide an example for each, and analyze the cost for each.

2.4.1 Dynamic Programming Overview

Time series comparison techniques such as those shown in Table 2.2 are typically evaluated using dynamic programming. Two time series R and S of length m and n, respectively, are compared using dynamic programming in the following way: First, an m x n two dimensional array A is constructed. Next, each element r_i of R is compared with each element s_j of S for all $1 \leq i \leq m$ and $1 \leq j \leq n$. The result of the comparison of r_i and s_j is added to the best cumulative score between $(r_1, ..., r_{i-1})$ and $(s_1, ..., s_{j-1})$ and stored in A at position (i, j). Once all the mn comparisons have been made and the elements of A are filled in, the final score is stored in $A(m, n)$.

For a concrete example, consider finding the LCSS score between the two time series shown in Figure 2.1. These two time series are from the popular Cylinder-Bell-Funnel (CBF) synthetic dataset [35]. The CBF dataset consists of time series from three classes, cylinders, bells, and funnels. Elements from the same class in the dataset are usually similar to each other. The two time series shown in Figure 2.1 are both from the cylinders class.

The two dimensional array used by the dynamic programming method for the LCSS comparison between R and S is shown in Figure 2.2a, where the ϵ matching criteria is chosen as one-quarter the standard deviation of the normalized time series (a common ϵ value, also chosen in [18]). In Figure 2.2a, black entries in the array at position (i, j) indicate mismatches between r_i and s_j. Gray entries in the array indicate matches between r_i and s_j that do not contribute to the LCSS score of R and S, and light gray colored entries indicate matches between r_i and s_j that are chosen by LCSS as the best alignment between R and S. Notice that the light gray colored entries run approximately from $(0, 0)$ to (m, n) along the grid diagonal. This makes intuitive sense – the alignment between two similar time series should match similar parts of each series (i.e. the front portion of R should not match the final portion of S).

Shortcomings of Dynamic Programming

When evaluating the LCSS of R and S, many of the comparisons made by dynamic programming when filling in the $m \times n$ two-dimensional array are between components of R and S that do not match, and therefore cannot positively impact the score between R and S. Much of the computation can be saved by finding only those elements r_i and s_j of R and S that match. An example of the positive matches between R and S is given in Figure 2.2b. This is the same two-dimensional array that is shown in Figure 2.2a, but the mismatching portions are no longer shown in black. The number of squares in this figure is much smaller than before. Since each square in the array represents work that must be

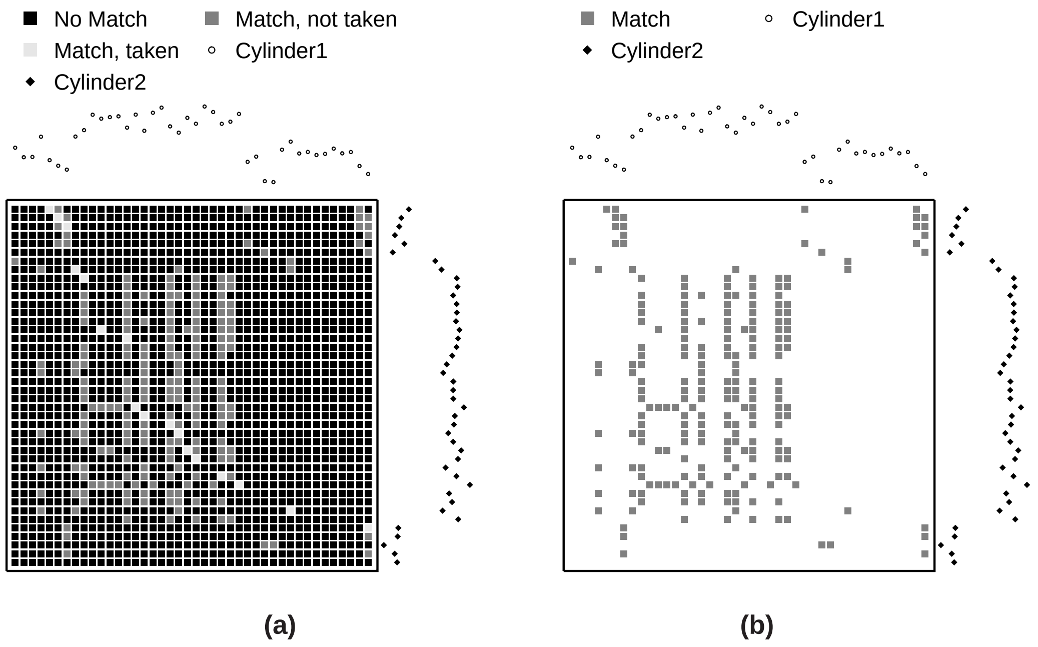


Figure 2.2: (a) The dynamic programming computations necessary to evaluate LCSS between the two cylinder examples from Figure 2.1. The first time series is above the computation array and the second time series is on its right. (b) The matching elements as determined by LCSS between the two time series shown in Figure 2.1. This is what is needed by FTSE to perform the evaluation, in contrast to the larger number of comparisons needed by dynamic programming.

done by the algorithm as well as space that must be used, the evaluation of ϵ threshold scoring techniques can be made more efficient. The main observation that we make is that if only those matches in Figure 2.2b are considered when comparing two time series, considerable computation can be saved since mismatching pairs are ignored.

2.4.2 Overview of FTSE

FTSE identifies the matching elements r_i and s_j between time series R and S *without* using a large two-dimensional array, such as that shown in Figure 2.2a. This is done by treating R and S nonuniformly, rather than treating them in the same way as in dynamic programming. In dynamic programming, both R and S are treated the same (each is lined up on one edge of the two-dimensional array to be compared with the elements of the other sequence).

To find the matching pairs between R and S without comparing each r_i with every s_j, FTSE indexes the elements of R on-the-fly into a grid. Each element of R is placed into a grid cell. Now, to find the elements of R that s_j matches, the grid is probed with s_j. Only the elements of R that reside in the same grid cell as s_j need to be compared with it to see if they match.

Once the matching pairs of R and S are found, the score of LCSS, EDR, or of the more general Swale ϵ scoring functions for R and S and the best alignment between them can be found using only an array of size n and a list containing the matching pairs between the two sequences (in contrast to the mn size array of dynamic programming). This is accomplished by noting that the grid can be probed by order of increasing S position. Hence, when the grid is probed with s_j to find the matching pairs between R and s_j, the matching pairs between the preceding $j-1$ elements of S with R have already been found. Therefore, when considering previous matches between $(s_1, ..., s_{j-1})$ and R for the best cumulative score for a match between r_i and s_j, there is no restriction on the previous matches from s_j. Any of the previous matches that contribute to the best cumulative score for r_i and s_j simply must be between elements of R before position i because the previous matches are inherently before s_j. Thus, high scores by position in R can be indexed into a one dimensional array of size n. The best alignment between R and S can be stored using a list containing matching pairs of elements derived from the grid.

One crucial requirement must be met for the index strategy of FTSE to win over the dynamic programming paradigm: the number of cells in the grid must be less than mn. Since the data is normalized, most elements fall between -3σ and 3σ. If epsilon is chosen as 0.5σ as is done in [77], then the grid contains 6/0.5=12 entries. Since time series are not usually more than 2 dimensional and typically of length considerably greater than 12, the grid size is typically much smaller than mn.

2.4.3 Finding Matches

In this section, we describe how the novel Fast Time Series Evaluation method finds matching pairs between elements of R and elements of S. FTSE measures the similarity between time series R and S with threshold value ϵ. Using ϵ, each pair of elements $r_i \in R$ and $s_j \in S$ can be classified as either a match or a mismatch. The elements r_i and s_j are said to match if $|r_i - s_j| < \epsilon$ in all dimensions. Otherwise, these two elements of R and S are a mismatch.

The first step in the FTSE algorithm is to find all intersecting pairs between elements of R and elements of S. The technique used to obtain these intersecting pairs is shown in Algorithm 1. First, a grid of dimensionality d is constructed (line 4 of the algorithm). The edge length of each element of the grid is ϵ.

In lines 6 to 8 of the algorithm, a Minimum Bounding Rectangle (MBR) is constructed for each element r_i of R. This MBR has a side length of 2ϵ in each dimension, and its center is the point r_i. This construction method ensures that r_i overlaps with no more than 3^d elements in the grid.

The MBR construction is illustrated in Figure 2.3 for one and two dimensions. In one dimension, the MBR of r_i is flattened into a line and intersects with 3 grid elements, as shown in Figure 2.3a. In two dimensions, the MBR of r_i intersects with 9 grid elements, as shown in Figure 2.3b.

A FIFO queue is associated with each cell g of the grid. The queue for each g is used to maintain a reference to all r_i that are within ϵ of g, in order of increasing i. This is done in line 9 of Algorithm 1.

The intersections between R and S are found in lines 11-18 of Algorithm 1. The grid cell g that contains each $s_j \in S$ is located. The elements of R in the queue associated with g are compared with s_j to see if they are within ϵ of one another. For each element r_k of

Algorithm 1 Build Intersection List.

```
1:  **Input:** $R$, $m$, $S$, $n$, $\epsilon$
2:  **Output:** Intersection List $L$
3:  **Local Variables:** Grid $G$, MBR $M$
4:  Initialize $G$: each grid element contains a queue that stores references to all intersecting elements.
5:  for $i$ = 1 to $m$ do
6:      for $k$ = 1 to $d$ do
7:          $M_i^k$=($r_i^k$-$\epsilon$, $r_i^k$+$\epsilon$)
8:      end for
9:      Insert $M_i$ into the queue associated with each grid square $g$ of $G$ which $M_i$ intersects.
10: end for
11: for $i$ = 1 to $n$ do
12:     Obtain queue $q_g$ for grid square $g$ in which $s_i$ lies.
13:     for $k \in q_g$ do
14:         if $|s_i - r_k| < \epsilon$ in all dimensions then
15:             insert $k$ into $L_i$
16:         end if
17:     end for
18: end for
```

R that is within ϵ of s_j, the index of r_k, i.e. k, is inserted into the intersection list L_j of s_j. The entries of L_j are also maintained in order of increasing k.

Note that the size of the grid is likely to be small for the following reason: Since data is normalized with mean zero and standard deviation $\sigma = 1$, most data will fall between -3 and 3. If the ϵ value is not exceptionally small relative to σ (which is common – for example, [77] uses 0.5σ), the size of the grid is reasonably small. Outliers beyond -3 or 3 are rare and can be captured into an additional grid cell.

If the dimensionality is unusually high, the grid may be built on a subset of the overall dimensions since, as is shown in the next section, the number of matching pairs between time series R and S decreases quickly as the dimensionality grows. This way, the technique can still be applicable in higher dimensional spaces.

Cost Analysis of Match Finding

The cost of finding the matches using the grid technique of FTSE is $O(P + m + n)$, where P is the total number of comparison operations between the elements of R and the elements of S made when probing the grid, m is the length of R, and n is the length of S. The cost to insert each element of R into a grid is $O(m)$, and the cost to probe the grid with each element of S is $O(n)$. There are $O(P)$ total comparisons between R and S.

The total number of probe comparisons between R and S will be similar to the total number of matches M, both of which are determined by the size of ϵ. (An element of S will match all elements that are within ϵ in each dimension. It will be compared in the probe phase with elements that are up to 2ϵ away from it in each dimension, and on average within 1.5ϵ in each dimension, since the element will be mapped to 3 grid cells in each dimension that are each of size ϵ.) While this cost is $O(mn)$ in the worst case, in the general case, P will be much less than mn for common ϵ values.

To obtain an average case analysis, we consider two 1 dimensional sequences R and S whose elements are chosen uniformly from the unit space in 1 dimension. Once R and S are normalized, they will be distributed normally with mean 0 and variance 1. The conditional density function of the standard normal random variable Z is provided in Equation 2.1. Since S is normalized, the values of its elements follow a normal distribution and we can consider the value of s_j to be a normal random variable. The probability that a standard normal random variable Z lies between two values a and b, where $a < b$, is given by Equation 2.2. Hence, the probability that the normalized value of r_i, $N(r_i)$, lies within ϵ of the normalized value of s_j, $N(s_j)$, is given in Equation 2.3.

$$\Phi(z) = \frac{1}{\sqrt[2]{2\pi}} \int_{-\infty}^{z} e^{-u^2/2}\, du \tag{2.1}$$

$$P[a < Z \leq b] = \Phi(b) - \Phi(a) \tag{2.2}$$

$$\begin{aligned} P[N(r_i) - \epsilon < N(s_j) &\leq N(r_i) + \epsilon] \\ &= \Phi(N(r_i) + \epsilon) - \Phi(N(r_i) - \epsilon) \end{aligned} \tag{2.3}$$

The expected number of matches M between r_i and the n elements of S is equal to the probability that a particular element of $N(S)$ matches with a value $N(r_i)$ multiplied by n.

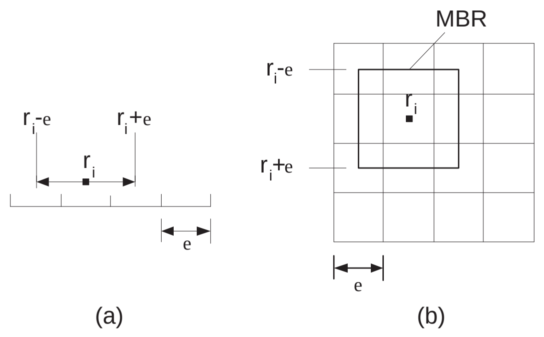


Figure 2.3: A depiction of R sequence elements with MBRs mapped to a (a) one-dimensional and (b) two-dimensional grid.

This is shown in Equation 2.4. We can then find the expected number of matches between R and S by summing over all m, in Equation 2.5. To obtain a solution in terms of mn, we can multiply by 1 (m/m) in Equation 2.6. We can approximate this summation (since we want an estimate in terms of mn) numerically by picking m values uniformly from the unit space for each r_i. We use two values for ϵ, 0.25σ and 0.50σ, which are commonly used ϵ values in [18] and [77], respectively. For $\epsilon = 0.50$, we obtain between $0.26mn$ and $0.27mn$ matches between R and S, and for $\epsilon = 0.25$, we obtain about $0.13mn$ matches between R and S. This is approximately a 4-7X improvement over dynamic programming.

$$E[M|r_i] = n(\Phi(N(r_i)+\epsilon) - \Phi(N(r_i)-\epsilon)) \tag{2.4}$$

$$E[M] = n\sum_{i=1}^{m}(\Phi(N(r_i)+\epsilon) - \Phi(N(r_i)-\epsilon)) \tag{2.5}$$

$$E[M] = mn\sum_{i=1}^{m}\frac{1}{m}(\Phi(N(r_i)+\epsilon) - \Phi(N(r_i)-\epsilon)) \tag{2.6}$$

The expected number of probes P can be found by replacing ϵ in Equation 2.6 with 1.5ϵ, the average maximum distance away from s_j that elements in R can be and still be compared with s_j in the probe phase. Doing so produces about $0.4mn$ probe comparisons when $\epsilon = 0.50$ and about $0.2mn$ probe comparisons when $\epsilon = 0.25$. This is an improvement of 2.5-5X over dynamic programming.

To obtain the average case analysis for 2 dimensions, we consider 2 dimensional time series R and S whose elements are drawn independently from the unit space. The analysis then is similar to the analysis above. The main difference is that $N(r_i)$ must match $N(s_j)$ in both dimensions. Since the values of r_i and s_j in each dimension are independent, we can arrive at Equation 2.7 by performing the same analysis as we did in the 1 dimensional case. If we approximate the number of matches in two dimensions numerically, we obtain between $0.06mn$ and $0.07mn$ matches when $\epsilon = 0.50$ and about $0.02mn$ matches when $\epsilon = 0.25$. This is about a 14-50X improvement over the dynamic programming, which produces mn comparisons.

$$\begin{aligned} E[M] &= mn\sum_{i=1}^{m}\frac{1}{m}[(\Phi(N(r_i^1)+\epsilon)-\Phi(N(r_i^1)-\epsilon)) \\ &* \quad (\Phi(N(r_i^2)+\epsilon)-\Phi(N(r_i^2)-\epsilon))] \end{aligned} \tag{2.7}$$

The expected number of probes P in 2 dimensions can be found by replacing ϵ in Equation 2.7 with 1.5ϵ, the average distance away from s_j that elements in R can be and still be compared with it in each dimension in the probe phase. Doing so produces about $0.16mn$ probe comparisons for R and S when $\epsilon = 0.50$ and about $0.05mn$ probe comparisons when $\epsilon = 0.25$. This is an improvement of 6-20X over dynamic programming for 2 dimensions.

2.4.4 Computing LCSS using FTSE

Once the intersections are found, the LCSS score for the pair R and S can be evaluated using Algorithm 2. An array called $matches$ is maintained that stores at position $matches[i]$ the smallest value k for which i matches exist between the elements of S and r_1, ... , r_k (line 4).

Algorithm 2 LCSS Computation.

```
 1: Input: $R$, $m$, $S$, $n$, $\epsilon$, Intersections $L$
 2: Output: $score$
 3: Local Variables: Array $matches$
 4: Initialize $matches[0] = 0$ and $matches[1 \text{ to } n] = m + 1$.
 5: max=0;
 6: for $j$ = 1 to $n$ do
 7:     Let $c$, a pointer into the $matches$ array, =0.
 8:     Let $temp$ store an overwritten value from $matches$.
 9:     $temp$=$matches[0]$.
10:     for $k \in L_j$ do
11:         if $temp < k$ then
12:             while $matches[c] < k$ do
13:                 $c = c + 1$.
14:             end while
15:             $temp$=$matches[c]$.
16:             $matches[c] = k$.
17:             if $c > max$ then
18:                 $max$ = $c$
19:             end if
20:         end if
21:     end for
22: end for
23: $score$ = $max$.
```

The values in $matches$ are filled by iterating through the elements of S (line 6). Variable c is an index into $matches$ and $temp$ stores an overwritten value from matches. For each of the intersections between r_k and s_j (line 10), k is checked against the value of $temp$ (line 11). Initially, $temp$ is 0 (line 9), so the algorithm proceeds to line 12. Next, c is incremented until the value of $matches[c]$ is not less than k. This indicates that there are $c - 1$ matches between s_1, ... ,s_{j-1} and r_1, ..., $r_{matches[c-1]}$. Adding the match between s_j and r_k makes c matches.

The old value of $matches[c]$ is stored to $temp$ (line 15) and $matches[c]$ is updated to k (line 16). The maximum possible number of matches is stored in max and updated if c is greater than it (lines 17-19). The value of $temp$ is updated because subsequent intersections between R and s_j cannot make use of the intersection between r_k and s_j. This is because the $LCSS$ technique only allows s_j to be paired with one r_k so the previous value is retained as a stand in for the old $matches[c]$ for the next loop iteration. At the end of the algorithm, the $LCSS$ score is stored in max (line 23).

Example for LCSS

To demonstrate the operation of FTSE for LCSS, let R be $r_1 = 2.0$, $r_2 = -0.5$, $r_3 = 1.0$, $r_4 = -2.2$, and $r_5 = -0.4$, and let S be $s_1 = -0.4$, $s_2 = -2.1$, $s_3 = 1.4$, $s_4 = -1.8$. Let $\epsilon = 0.5$.

The matching phase of Algorithm 1 progresses by generating a one dimensional grid in which each grid cell has a side length of 0.5 (the ϵ value). Assume that grid boundaries occur at -2.5, -2, -1.5, -1, -0.5, 0, 0.5, 1, 1.5, 2, and 2.5 (line 4 of Algorithm 1). Next, the algorithm generates MBRs for each element of R (lines 5 to 8). The MBRs for each r_i are (1.5, 2.5) for r_1, (-1, 0) for r_2, (0.5, 1.5) for r_3, (-2.7, -1.7) for r_4, and (-0.9, 0.1) for r_5,

Next, the algorithm inserts each r_i into the grid (line 9). For example, the grid cell with boundaries $(-0.5, 0)$ contains both r_2 and r_5. The grid is then probed with each S value (lines 11-18). First, the grid is probed with s_1. The cell in which it lies, (-0.5,0), contains two MBRs – namely r_2 and r_5. Both elements of R are compared with s_1. Since they are both within ϵ of s_1, 2 and 5 are inserted into intersection list L_1, in that order.

Then, the grid is probed with s_2. The grid in which it is located, ($-2, -2.5$), contains only one element, r_4. Since r_4 and s_2 are within 0.5 of one another, 4 is inserted into L_2. In a similar way, the grid is probed with s_3 and s_4 to produce a match with r_3 for s_3 and with r_4 for s_4.

Next, the operation of Algorithm 2 progresses. The initial state of the $matches$ array is shown in Figure 2.4. The algorithm begins processing the intersection list of s_1. The first value in the intersection list for s_1 is 2 (line 10 of the algorithm), since s_1 intersects with r_2.

Since $matches[0] < 2 < matches[1]$ (lines 12-14), the greatest number of matches possible so far is 1, so the c pointer is set to 1. Hence, the value of $temp$ is updated to the old value of $matches[1]$ (line 15), which is 6 and $matches[1]$ is updated to 2 (line 16).

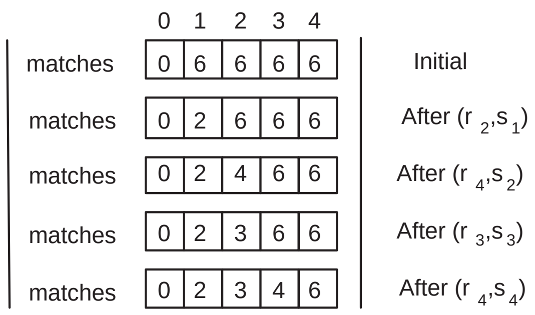

0 1 2 3 4
matches 0 6 6 6 6 Initial
matches 0 2 6 6 6 After (r 2,s 1)
matches 0 2 4 6 6 After (r 4,s 2)
matches 0 2 3 6 6 After (r 3,s 3)
matches 0 2 3 4 6 After (r 4,s 4)

Figure 2.4: The $matches$ array during FTSE LCSS evaluation.

The value of max is updated to 1 (lines 17-18). The new status of $matches$ is shown in Figure 2.4. The next value in the intersection list for s_1 is 5. Since 5 is less than $temp$ (line 11), this intersection cannot be used.

After the processing of the s_1 intersection list, c and $temp$ are reset for the intersections of s_2 (lines 7-9). The first and only value in the intersection list for s_2 is 4 (line 10). Since $matches[1] < 4 < matches[2]$ (lines 12-14), c is set to 2. The value of $temp$ is updated to $matches[2]$ (line 15), and $matches[2]$ is updated to 4 (line 16). The value of max is also updated to 2 (lines 17-18).

The intersection list for s_3 is processed in the same way. Since its only match is with r_3, and because $matches[1] < 3 < matches[2]$, the value of $matches[2]$ is overwritten with 3 (see Figure 2.4). The intersection list of s_4 is also processed, and since s_4 intersects with r_4, and $matches[2] < 4 < matches[3]$, the value of $matches[3]$ is updated to 4, and the max value becomes 3.

Since all the S points have been processed, the algorithm terminates. The best possible number of matches between R and S is stored in max, which is 3. This is the LCSS score.

Cost Analysis of FTSE computing LCSS

The cost of FTSE for computing LCSS is $O(M + Ln)$, where M is the number of matches (discussed in Section 2.4.3) and L is the length of the longest matching sequence between R and S (i.e. the LCSS score). The proof of this is straightforward and hence is omitted (essentially, each matching pair is considered, and the length of the longest match is stored in the array and is iterated over for each element n of S). In the worst case, this length will be equal to the length of $min(m, n)$ (since the LCSS score cannot exceed the length of either sequence), which could be as long as m, making the overall cost $O(mn)$. However, this worst case occurs only when all elements of R and S are matched in the LCSS score, which is not expected to happen often, even for sequences that are quite similar.

To obtain an average case analysis for the size of L in 1 dimension, we again assume time series R and S have their elements drawn uniformly from the unit space. We numerically approximate L by generating one thousand random versions of R and S, each of length one thousand. We then measure the average, maximum, and minimum length of L. For $\epsilon = 0.25$, the average size of L is $0.52m$, the maximum size is $0.54m$, and the minimum size is $0.51m$. For $\epsilon = 0.50$, the average size of L is $0.66m$, the maximum size is $0.68m$, and the minimum size is $0.64m$. The small variation in the sizes of L show that this average case analysis produces repeatable results. It also shows a 1.5-2X improvement over dynamic programming's mn computation to find the best alignment of R and S.

We obtain an average case analysis for 2 dimensions through numerical approximation as well. For $\epsilon = 0.25$, the average size of L is $0.23m$, the maximum size is $0.24m$, and the minimum size is $0.22m$. For $\epsilon = 0.50$, the average size of L is $0.41m$, the maximum size is $0.43m$, and the minimum size is $0.39m$. The smaller size of L in two dimensions is because r_i must match s_j in two dimensions instead of just 1, which produces fewer matches between each r_i and the elements of S (see Section 2.4.3). This analysis shows a 2.5-4X improvement over dynamic programming.

Algorithm 3 EDR Computation.

```
1: **Input:** $R$, $m$, $S$, $n$, $\epsilon$, Intersections $L$
2: **Output:** $score$
3: **Local Variables:** Array $matches$
4: Initialize $matches[0] = 0$ and $matches[1 \text{ to } 2n] = m + 1$.
5: max=0;
6: for $j$ = 1 to $n$ do
7:     Let $c$, a pointer into the $matches$ array, =0.
8:     Let $temp$ store an old value from $matches$,=$matches[0]$
9:     Let $temp2$ store an old value from $matches$,=$matches[0]$
10:    for $k \in L_j$ do
11:        if $temp < k$ then
12:            while $matches[c] < k$ do
13:                if $temp < matches[c] - 1$ and $temp < m - 1$ then
14:                    $temp2 = matches[c]$
15:                    $matches[c] = temp + 1$
16:                    $temp = temp2$
17:                else
18:                    $temp = matches[c]$
19:                end if
20:                $c = c + 1$.
21:            end while
22:            $temp2$=$matches[c]$.
23:            $matches[c]$=$temp + 1$.
24:            $temp$=$matches[c + 1]$.
25:            if $matches[c + 1] > k$, then $matches[c + 1] = k$
26:            if $max < c + 1$, then $max = c + 1$
27:            $c = c + 2$.
28:        else if $temp2 < k$ and $k < matches[c]$ then
29:            $temp2 = temp$
30:            $temp = matches[c]$
31:            $matches[c] = k$
32:            if $max < c$, then $max = c$
33:            $c = c + 1$
34:        end if
35:    end for
36:    for $j = c$ to $max + 1$ do
37:        if $temp < matches[j] - 1$ and $temp < m - 1$ then
38:            $temp2 = matches[j]$
39:            $matches[j] = temp + 1$
40:            $temp = temp2$
41:            if $max < j$, then $max = j$
42:        else
43:            $temp = matches[j]$
44:        end if
45:    end for
46: end for
47: $score$ = $max - (m + n)$.
```

2.4.5 Computing EDR using FTSE

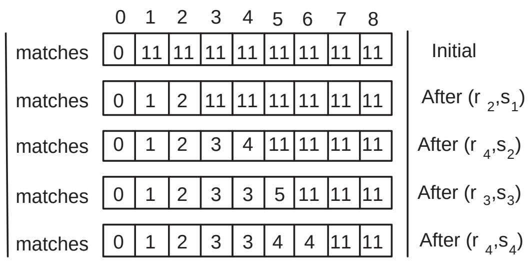
Unlike LCSS, EDR does not reward matches, but rather penalizes gaps and mismatches, so the FTSE algorithm changes slightly. The maximum possible score for EDR(R,S,ϵ) is 0 if R and S are nearly identical. The worst possible score is $-1*(m+n)$, if all m elements of R and all n elements of S incur a gap penalty. A mismatch penalty of -1 between elements r_i of R and s_j of S can thus be viewed as a savings of 1 over two mismatches (which together have a cost of -2 versus the -1 mismatch cost). A match between r_i and s_j has a score of 0, which is a savings of 2 over the gap penalty costs. FTSE for EDR thus scores a match with a $+2$ reward and a mismatch with a $+1$ reward, and considers the baseline score to be $-1 * (m + n)$ instead of zero.

The FTSE algorithm for EDR is presented in Algorithm 3. The $matches$ array is initialized (line 4 of Algorithm 3) similar to Algorithm 2. Since match rewards are being scored with a 2, the array needs to be twice as long. Variables max (line 5), c (line 7), and $temp$ (line 8) are the same as before. Variable $temp2$ stores an overwritten value of the $matches$ array, similar to $temp$. A second such temporary holder is needed because match rewards are scored with a $+2$, hence two values can be overwritten on an iteration.

Most of FTSE for EDR is the same as FTSE for LCSS, such as iterating through the elements of S (line 6) and checking each element of the intersection list for the appropriate $matches$ value (lines 10-12).

Mismatches are handled by lines 13-19. Variable $temp$ stores the value of $matches[c-1]$. Since s_j can obtain a mismatch with any element of R, each value of $matches$ must be incremented (line 15). The overwritten value of $matches$ is stored back into $temp$ (lines 14, 16, 18). Line 13 checks that a previous element has not matched at position c of $matches$ (producing a higher score than a potential mismatch) and that the length of R has not been exceeded.

Lines 22-27 handle a match. The previous value of $matches[c]$ is stored in $temp2$ (line 22) since $matches[c]$ will be updated with a mismatch score (line 23); $matches[c + 1]$ is stored in $temp$ (line 24) since a match is recorded at $matches[c + 1]$ (line 25). The



	0	1	2	3	4	5	6	7	8	
matches	0	11	11	11	11	11	11	11	11	Initial
matches	0	1	2	11	11	11	11	11	11	After (r_2, s_1)
matches	0	1	2	3	4	11	11	11	11	After (r_4, s_2)
matches	0	1	2	3	3	5	11	11	11	After (r_3, s_3)
matches	0	1	2	3	3	4	4	11	11	After (r_4, s_4)

Figure 2.5: The $matches$ array during FTSE EDR evaluation.

maximum score and c counter are updated in lines 26 and 27 respectively.

Lines 28-34 handle the case when the next matching element in intersection list L_j is greater than the previous element by exactly 1. For example, if s_j matches elements r_k and r_{k+1}. In this case, the match with r_{k+1} will not necessarily exceed $temp$, the previously updated $c - 1$ value, but might exceed $temp2$, the previously updated $c - 2$ value. The update code is similar to lines 22-27 already described.

Lines 36-45 handle the case when either s_j has no matches in R or when s_j matches elements only near the beginning of R. In this case, s_j could obtain mismatch scores with the remaining portions of R. This section of Algorithm 3 is similar to the already described lines 13-19.

Example for EDR

We show the operation of FTSE evaluating EDR with the same example as was used for LCSS. Following the intersection list generation of Algorithm 1 already discussed, Algorithm 3 begins by initializing $matches$. This initialized state is seen in Figure 2.5.

The first match is obtained from the intersection list (line 10 of the algorithm). This is the intersection between r_2 and s_1, hence $k = 2$. Since $matches[0] < 2 < matches[1]$, c is set to 1 in lines 13-20. $temp$ and $temp2$ are both set to 11 (lines 22 and 24). $matches[1]$ is set to 1 because s_1 can mismatch with r_1. $matches[2]$ is set to 2 because r_2 matches with s_1. Nothing is done for the match between r_5 and s_1. The updated $matches$ array is shown in Figure 2.5.

The intersection list for s_1 is now empty, so FTSE proceeds to line 36. c is 3 and $max + 1$ is 3, so the loop is taken exactly once. The if condition at line 37 fails, so no changes are made to $matches$.

Next, the intersection between r_4 and s_2 is processed, so $k = 4$. Since $matches[2] < 4 < matches[3]$, c is set to 3 by lines 12-20. No changes are made to the $matches$ array by lines 14-16. Hence, the else condition (line 18) is taken for both $c = 1$ and 2 and $temp = 2$. $matches[3]$ is set to $temp + 1 = 3$ (line 23) and $matches[4]$ is set to 4 (line 25) since $k = 4$. max is updated to 4 (line 26) and c is set to 5 (line 33). Again, lines 36-45 make no changes to $matches$.

The intersection between s_3 and r_3 is next considered. As shown in Figure 2.5, The match between s_3 and r_3 can use the match between s_1 and r_2 at position 2 of $matches$. So, the value of k (3) is recorded at position 4 of $matches$. When processing for s_3 reaches line 36, $temp$ is 4, c is 5, and max is 4. Hence, lines 37-40 record a value of 5 in position $matches[5]$. This is because s_3 builds upon the r_4 and s_2 match with a mismatch between itself and r_5.

Finally, the intersection between r_4 and s_4 is processed. Since the intersection between r_3 and s_3 has resulted in a $match[4]$ value of 3, line 23 will set $match[5]$ to 4, and line 25 will set $match[6]$ to a value of 4. This means that max is also set to 6 (line 27). The final score achieved (line 47) is $-1 * (5 + 4) + 6 = -3$.

Cost Analysis of FTSE computing EDR

The cost of FTSE when evaluating EDR is $O(M + Tn)$, where M is the number of matches between R and S, n is the length of time series S, and T is the value of max in Algorithm 3. This complexity results from iterating over the $matches$ array for each of the n elements of S up to max places in the array. The value of max is bounded between

$min(m, n)$ and $2min(m, n)$. This is because the value of max is increased once for each mismatch and two times for each match that occurs in the final alignment between R and S. While this is still $O(mn)$ in the worst case, FTSE for EDR still achieves efficiencies relative to dynamic programming since it only needs to store the number of matching elements M between R and S. This leads to better performance, which is later quantified experimentally in Section 2.6.2.

2.4.6 Maintaining the Best Matching Pairs

The FTSE algorithm for either LCSS or EDR can be easily modified to find not only the best score between two time series, but also the best sequence of matching pairs that produce that score. Maintaining the best sequence of matching pairs is useful for applications that seek to compute the best alignment between two time series. We now discuss how to modify FTSE for LCSS; a similar discussion for EDR is omitted.

The matching pairs found in Algorithm 1 are maintained in a list of intersections. The list element that contains a particular match can be linked to the previous best set of list elements when the match is considered in line 10 of Algorithm 2 since each match contributes to the best score in at most one position. The best alignment can be found by maintaining an array of the list elements that contain the matching pairs. Each array position corresponds to the last match in the sequence, with the remaining matches chained out behind it.

The following three lines can be added to Algorithm 2 between lines 16 and 17 to maintain the record of the best alignment (where l_k is the list element for match k):

```
$alignment[c] = l_k$.
**if** $c > 0$ $then$ $l_k.next = alignment[c-1]$.
**else** $l_k.next = 0$.
```

The $alignment$ array is of length n, similar to $matches$. It is initialized to all null entries. At the end of the algorithm, the best sequence is maintained in the $alignment$

array, and it can be returned to the user.

2.5 The Swale Scoring Model

The FTSE algorithm can be used to evaluate a broad class of ϵ threshold value based scoring models, of which LCSS and EDR are two examples. This broader class of scoring models includes a new Swale scoring model, which we present below. The Swale scoring model improves over previous approaches in several ways. First, it allows for a sequence similarity score to be based on both match rewards and mismatch penalties. Second, it allows for the match reward and gap penalties to be weighted relative to one another. These weights also allow a user or domain expert with particular knowledge of a certain area to tune the distance function for optimal performance instead of having only one technique for all data domains. If the user has no such domain-specific knowledge, a training dataset can be used to automatically learn the weights (as we do in all the experiments presented in this paper).

More formally, the Swale distance function is defined as:

Definition 2.5.1. *Let R and S be two time series of length m and n, respectively. Let the gap cost be* gap_c *and let the match reward be* $reward_m$. *Then* $Swale(R, S)$ =

$$\begin{cases} n * gap_c, & \text{if } m = 0 \\ m * gap_c, & \text{if } n = 0 \\ reward_m + \quad Swale(Rest(R), Rest(S)), & \text{if } \forall d, |r_{d,1} - s_{d,1}| \le \epsilon \\ max\{gap_c + Swale(Rest(R), S), \; gap_c + Swale(R, Rest(S))\} & \text{otherwise} \end{cases}$$

Next we explain why Swale offers a better similarity measure compared to the best existing ϵ methods, namely LCSS and EDR [18, 78]. For this illustration, consider the sequences shown in Figure 2.6. Sequence A contains six elements. Sequence B has the same six elements as A, but has three additional “noise” elements embedded in it. Sequence C contains ten elements, and sequence D has the same ten elements with three

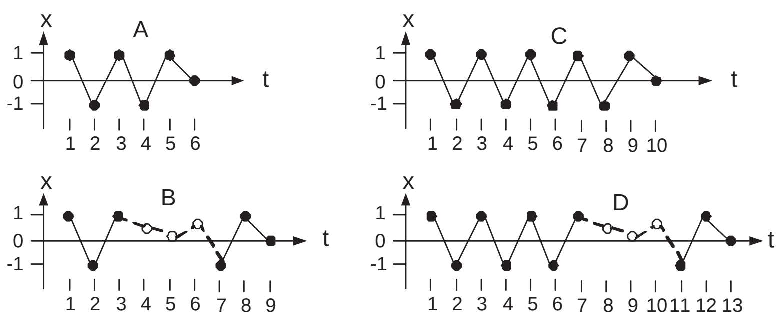


Figure 2.6: Time Series Examples

additional “noise” elements in it. Note that the number of mismatched elements between C and D is the same as that between A and B.

Both EDR and LCSS lose some information when scoring these sequences. EDR scores gaps and mismatches, but does not reward matches. In this sense, it only measures dissimilarity between two sequences. For example, A and B receive the same score as C and D even though C and D have nearly twice as many matching elements.

LCSS rewards matches, but does not capture any measure of dissimilarity between two sequences. For example, the LCSS technique scores C and D identically to C scored with itself, which is not intuitive.

Swale is similar to LCSS because it rewards matches between sequences, but it also captures a measure of their dissimilarity by penalizing gap elements. Swale allows C and D to obtain a higher score than A and B because they have more matching elements while still penalizing them for gap costs.

The Swale scoring function can be evaluated with the same FTSE algorithm described for LCSS by simply changing the last line of Algorithm 2 to $score = max * reward_m + gap_c * (m + n - 2max)$.

Method	**CM**	**ASL**	**CBF**	**Trace**
DTW	53.23	1.31	1.94	521.93
ERP	77.43	1.76	2.68	553.73
LCSS	42.74	0.93	1.41	386.09
EDR	43.69	1.01	1.41	390.87
SC-B$_{DTW}$	10.55	0.78	0.71	104.91
SC-B$_{LCSS}$	14.61	0.80	0.88	132.43
I-Par	15.44	0.86	0.90	141.05
FTSE$_{LCSS}$	5.13	0.78	0.74	80.80
FTSE$_{EDR}$	6.27	0.82	0.85	99.17

Table 2.3: Time in seconds to cluster a given dataset, using techniques that compute the actual alignment.

2.6 Experiments

In this section, we experimentally evaluate the performance of FTSE, and the accuracy of Swale.

2.6.1 FTSE Experimental Evaluation

In this section, we evaluate the effectiveness of the FTSE technique evaluating both LCSS and EDR. Since Swale is evaluated with only a small modification to FTSE for LCSS, its performance is identical to LCSS with FTSE. All experiments are run on a machine with a 1.7 GHz Intel Xeon, with 512MB of memory and a 40GB Fujitsu SCSI hard drive, running Debian Linux 2.6.0. We compare the performance of FTSE against DTW, ERP, LCSS, and EDR. Each technique is evaluated using a traditional, iterative dynamic programming-style algorithm.

The performance of FTSE is dependant on the ϵ value, since this value determines which elements of R and S are close enough to one another to be matched. The emphasis of our work is not on describing how to pick an ϵ value for either LCSS or EDR, but to demonstrate the effectiveness of FTSE for reasonable choices of ϵ. Consequently, we show results with an ϵ value of 0.5σ, where σ is the standard deviation of the data (since we are dealing with normalized data, σ is 1). We have chosen this ϵ value since it was shown to produce good results in [77].

Method	**CM**	**ASL**	**CBF**	**Trace**
DTW	35.23	1.20	1.78	329.17
LCSS	14.24	0.84	1.16	129.42
SC-B$_{DTW}$	7.05	0.76	0.74	72.91
SC-B$_{LCSS}$	6.40	0.74	0.72	66.95
FTSE$_{LCSS}$	2.69	0.72	0.61	48.28
FTSE$_{SC-B}$	2.26	0.70	0.60	40.74

Table 2.4: Time in seconds to cluster a given dataset, using $O(n)$ storage techniques that do not compute the alignment.

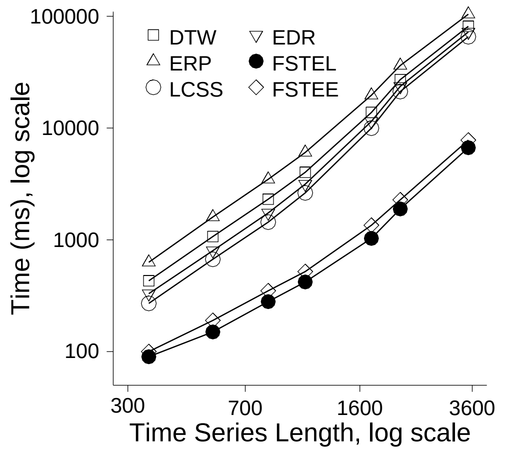


Figure 2.7: Cost of computing similarity scores v/s time series length. (CLIVAR dataset)

In our first experiment we show the average time to perform the similarity comparisons for a complete linkage clustering evaluation. Complete linkage clustering of time series was used in both [78] for LCSS and in [18] for EDR. For a dataset with k time series, each clustering run involves computing approximately $k \times (k-1)$ time series similarity scores.

To perform the complete linkage clustering, our evaluation uses the same datasets used in [78] and in [18], which includes the Cameramouse (**CM**) dataset [23] and the Australian Sign Language (**ASL**) dataset from the UCI KDD archive [76]. Since both of these datasets are two dimensional, we also experiment with the popular Cyliner-Bell-Funnel (**CBF**) dataset of [35] and the **Trace** dataset of [65]. The CBF dataset contains three classes (one each for the cylinder, bell, and funnel shapes) and is synthetically generated. We use 10 examples from each class in the clustering. The Trace dataset is a four class

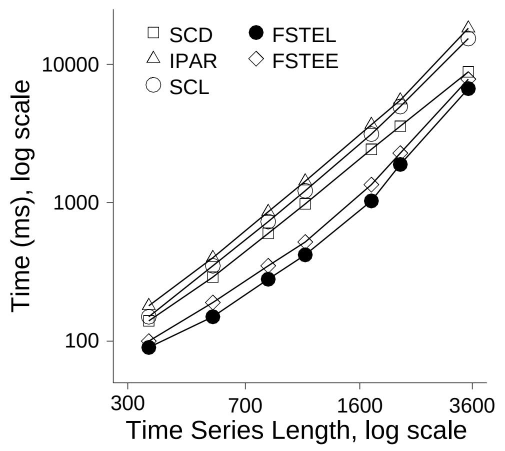


Figure 2.8: Cost of computing similarity scores v/s time series length; comparison with warp-restricting methods. (CLIVAR dataset)

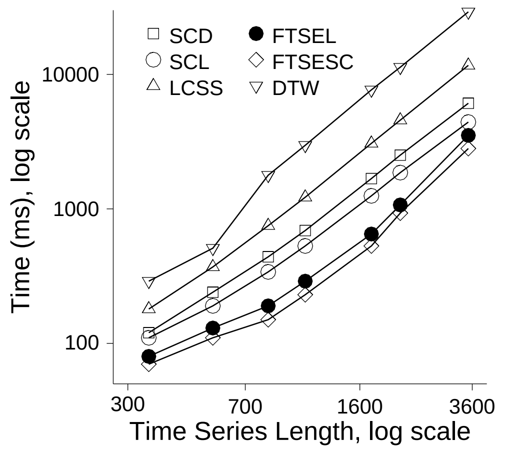


Figure 2.9: Cost of computing similarity scores v/s time series length; comparison with methods that do not compute the actual alignment. (CLIVAR dataset)

synthetic dataset that simulates instrument failures inside of a nuclear power plant. There are fifty examples for each class.

The CM dataset consists of 15 different time series obtained from tracking the fingertips of people in two dimensions as they write words. Three different people wrote out five different words. This gives a total of five distinct class labels (one for each word) and three members for each class.

The ASL dataset contains examples of Australian Sign Language signs. The dataset contains time series in two dimensions for words that are signed by the user, and each word is signed five different times. We choose to use the same 10 word examples of [78]. This gives us a dataset with 10 classes with 5 time series per class.

We also compare with the Sakoe-Chiba Band (SC Band) and Itakura Parallelogram techniques for warp-restricting DTW. A restriction value of 10 percent is used in [86], so we also use this value. A similar technique to the SC Band for LCSS is described in [78], which sets the restriction value to 20 percent. The Itakura Parallelogram is referred to as (I-Par).

The results for the complete linkage clustering test is shown in Table 2.3. For the CM data set, FTSE is faster than the dynamic programming methods by a factor of 7-8 and faster than the warp-restricting techniques by a factor of 2-3. FTSE is faster than DP by a factor of 4-5 and is nearly twice as fast as the SC Band evaluating LCSS for the Trace dataset. FTSE also consistently performs better than dynamic programming on the other datasets. Note that the performance advantage achieved using FTSE relative to the various DP techniques is not as large for ASL and CBF as it is for the CM and Trace datasets. This is because the average sequence length of the ASL and CBF sequences are 45 and 128 respectively, while the average length of the CM is 1151 and the Trace is 255. This indicates that FTSE performs better than DP as length increases, which we also show in the next experiment. The Trace dataset also takes longer to evaluate than others datasets because it contains many more sequence examples (200) than CM (15), ASL (50), or CBF (30).

We also show results for both the DP and SC Band techniques using $O(n)$ storage techniques that produce the best score but do not yield the best alignment between the two sequences in Table 2.4. Essentially, since the i^{th} column of the mn matrix depends only on the $i-1^{th}$ column, 2 column arrays can be used. Similarly, it is a simple extension for FTSE to show that the list of intersections need not be materialized if an alignment between the two time series is not required; we have also implemented this version of FTSE ($FTSE_{LCSS}$ in the table). FTSE can also be implemented with a warp-restriction (in essence, to only consider matches in the same narrow band as the SC Band technique). We have also implemented this version with a restriction value of 20 percent to show that FTSE ($FTSE_{SC-B}$ in the table) can obtain the same score as the SC Band, if desired. In these tests, we limit results to LCSS and DTW evaluation In these tests, FTSE is faster than DP for LCSS by a factor of 7 and by more than 2X when restricting the warping window for the 2 dimensional CM dataset and by a factor of 2.5 over exact DP for LCSS and by a factor of 1.5 when restricting the warping window when evaluating the 1 dimensional Trace dataset.

The second experiment evaluates the effectiveness of the FTSE algorithm as the time series length varies. For this experiment, we use the CLIVAR-Surface Drifters trajectory dataset from [58], which contains climate data obtained in 2003 from free-floating buoys on the surface of the Pacific Ocean. This data contains the longitude and latitude coordinates for each buoy. The time series in this data set vary in length from 4 to 7466 data points.

From the CLIVAR-Surface Drifters dataset, subsets of data are produced such that each subset contains time series of similar length (all time series in a subset are within 10% of the average). For experimentation, subsets of 5 time series each are chosen with the following average time series lengths: 349, 554, 826, 1079, 1739, 2142, and 3500. As before, we report the time needed to perform $k \times (k-1)$ comparisons (the same as was done in the clustering experiments). Since each subset contains 5 time series, this is the time to perform 20 time series comparisons. The results for this experiment are shown in Figures 2.7, 2.8, and 2.9.

Figure 2.7 shows the results for FTSE evaluating LCSS (labeled FTSEL) and EDR (FTSEE). It also shows DTW, ERP, LCSS, and EDR evaluated using dynamic programming. As can be seen in this figure, FTSE is nearly an order of magnitude faster than the dynamic programming techniques. The figure also shows that the performance advantage of FTSE over the DP techniques increases with sequence length.

Figure 2.8 presents results for FTSE and the Sakoe-Chiba Band (SCD evaluating DTW and SCL evaluating LCSS) and Itakura Parallelogram (IPAR) warp-restricting techniques. FTSE is about twice as fast as the Sakoe-Chiba Band and 2-3 times faster than the Itakura Parallelogram technique. SC for LCSS is slower than for DTW because the warping restriction needed for good results (20%) for LCSS is larger than for DTW (10%).

Figure 2.9 presents results for the $O(n)$ storage techniques already discussed. FTSE (FTSEL in the figure) is generally about 3 times faster than the DP methods (LCSS and DTW) and almost twice as fast when the warp-restricted version of FTSE (FTSESC) is compared with the SC Band technique (SCL and SCD).

In summary, compared to existing methods that compute the actual alignment, *FTSE is up to 7-8 times faster than popular dynamic programming techniques for long sequences and 2-3 faster than warp-restricting techniques, while providing an exact answer*.

2.6.2 Experimental Cost Analysis of FTSE

The complexity and average case cost of FTSE have already been analyzed in Sections 2.4.3 and 2.4.4. In this section, we analyze the experimental cost of FTSE to show why it performs better than the other techniques that produce the best alignment, using the CM dataset as an example.

FTSE is more efficient for two reasons: it performs fewer operations than the competing techniques and it requires less space, which improves the algorithm's memory and cache performance.

The number of operations performed by FTSE is dependent on two principle components: the number of matches between elements of R and elements of S obtained by Algorithm 1 and the number of reads or writes to the $matches$ array in Algorithm 2. For the CM dataset, there are about 120 thousand matching pairs on average between any two sequences R and S (since the average length of each time series is 1151 elements, there are a total possibility of $1151 * 1151 = 1.32$ million) and about 300 thousand reads and writes to the $matches$ array. This means that FTSE performs about 420 thousand operations on the CM dataset versus the 1.32 million for DP, which is less than one-third.

The amount of space used by FTSE is dependant on the number of matching pairs generated by Algorithm 1. The $matches$ array and the grid (which contains fewer than 200 grid cells for CM) are of negligible size. For the CM dataset, the number of matching pairs is approximately 120 thousand. The equivalent DP algorithm writes approximately 1.32 million elements.

To test that this considerable space difference actually results in cost savings, we modified Algorithm 2 by allocating an amount of space equivalent to that of the DP algorithm and adding a line between lines 13 and 14 of Algorithm 2 that randomly writes to an element of the allocated space. The new algorithm attains improved performance only from the saved operations, not from memory or cache efficiency. The time this new FTSE takes to cluster the CM dataset is 12.12 seconds (before it was 5.13). This is expected, since DP for LCSS takes 42.74 seconds and the ratio of operations between FTSE and DP is $420/1320$ and $42.74 * 420/1320 = 13.59$ seconds.

2.6.3 Evaluation of the Swale Scoring Model

In this section, we evaluate the effectiveness of the Swale scoring model compared to existing similarity models. For this evaluation, we test the ability of the model to produce high-quality clusters. (Following well-established methodology [18, 35, 78].)

For our evaluation, we used the Cameramouse (CM) dataset [23], and the Australian Sign Language (ASL) dataset (as previously described). In addition, we also obtained an additional dataset from the UCI KDD archive [76] called the High Quality ASL. This dataset differs from the ASL dataset in the following way: In the ASL dataset, several different subjects performed the signing, and lower quality test gloves were used. The High Quality ASL (HASL) dataset consists of one person performing each sign 27 times using higher quality test gloves. Details regarding these differences can be found at [76]. We do not provide detailed results for the Trace and CBF datasets because these datasets have a small number of classes (4 and 3, respectively) and hence do not offer as much room for differentiation as the ASL datasets (all techniques tested on Trace and CBF performed nearly identically).

In the evaluation we perform hierarchical clustering using Swale, DTW, ERP, LCSS, and EDR. (We omit a comparison with the Euclidean distance, since it has been generally shown to be less robust than DTW [18, 33, 78].) Following previous established methods [18,35,78], for each dataset, we take all possible pairs of classes and use the complete linkage algorithm [32], which is shown in [78] to produce the best clustering results.

Since DTW can be used with both the L1-norm [17] and the L2-norm [37] distances, we implement and test both these approaches. The results for both are similar. For brevity, we present the L1-norm results.

The Swale match reward and mismatch penalty are computed using training datasets. The ASL dataset in the UCI KDD archive contains time series datasets from several different signers placed into directories labeled by the signer's name and trial run number. We selected the datasets labeled adam2, john3, john4, stephen2, and stephen4 for test datasets 1-5, respectively, and datasets andrew2 and john2 for training. For the HASL, each word has 27 examples, so we are able to group them into 5 different collections of data, each

	1	2	3	4	5	total
DTW	**40**	**32**	34	37	41	184
ERP	38	**32**	39	40	41	190
LCSS	**40**	30	38	39	41	188
EDR	38	27	39	37	**43**	184
Swale	39	29	**41**	**42**	42	**193**

Table 2.5: Number of correct clusterings (each out of 45) for the ASL dataset. The best performers are highlighted in bold.

with 5 examples, with 2 examples left over. The first such dataset is used for training, and the others are used for testing.

For the training algorithm, we use the random restart method [67]. Since the relative weight of the match reward and gap cost is what is important (i.e. the ratio between them), we fix the match reward to 50 and use the training method to pick various gap costs. The computed mismatch cost for ASL is -8 and for HASL is -21.

The CM dataset does not have enough data to produce a training and a test set. We therefore chose the ASL weight as the default. All techniques correctly clustered the dataset (10 out of 10 correct).

The total number of correct clusterings for each of the five different ASL datasets (out of 45 for each dataset) are shown in Table 2.5. As can be seen in the table, Swale has the overall best performance for the tests. There is a high degree of variability for all the similarity functions from one ASL dataset to the next, but some general trends do emerge. For example, all of the techniques perform well on dataset 5, averaging over 40 correct clusterings out of 45 possible. All of the techniques do relatively poorly on dataset 2, averaging only about 30 correct clusterings out of 45. These two datasets emphasize the variability of data for multi-dimensional time series; two datasets in the same ASL clustering framework produce very different results for all of the tested similarity measures.

The results for the HASL datasets are shown in Table 2.6 and are once again out of a possible 45 for each technique on each test. Overall, DTW, ERP, LCSS, EDR, and Swale

	1	**2**	**3**	**4**	**total**
DTW	8	8	2	5	23
ERP	9	5	4	**7**	25
LCSS	8	**10**	**6**	**7**	31
EDR	13	2	3	6	24
Swale	**18**	**10**	5	**7**	**40**

Table 2.6: Number of correct clusterings (each out of 45) for the HASL dataset. The best performers are highlighted in bold.

obtain fewer correct clusterings on the HASL datasets than they do on the ASL datasets. There is also high variability in accuracy across the datasets, just as in the ASL data presented in Table 2.5. Swale performs much better on the classifications for the HASL datasets than the alternative techniques, obtaining a total of 40 correct total classifications. The closest competitor is the LCSS technique with 31. This dataset also highlights how Swale leverages the combination of the match reward and gap penalty on real datasets for improved accuracy. On HASL dataset 1, EDR, which also uses gap penalties, performs much better than the LCSS technique. Swale also performs very well on this dataset. On HASL dataset 2, the LCSS technique performs better than EDR. Swale performs as well as the LCSS technique on this dataset, and is thus able to obtain the best of both worlds - it does well when EDR does well, and also does well when LCSS does well!

In summary, the results presented in this section demonstrate that Swale is consistently a more effective similarity measuring method compared to existing methods.

2.7 Conclusions

In this chapter, we have presented a novel algorithm called FTSE to speed up the evaluation of ϵ threshold-based scoring functions for time series datasets. We have shown that FTSE is faster than the traditionally used dynamic programming methods by a factor of 7-8, and is even faster than approximation techniques such as the Sakoe-Chiba Band by a factor of 2-3. In addition, we also presented a flexible new scoring model for comparing the similarity between time series. This new model, called Swale, combines the notions of gap penalties and match rewards of previous models, and also improves on these models. Using extensive experimental evaluation on a number of real datasets, we show that Swale is more accurate compared to existing methods. In the next chapter, we will begin our discussion of temporal skyline evaluation.

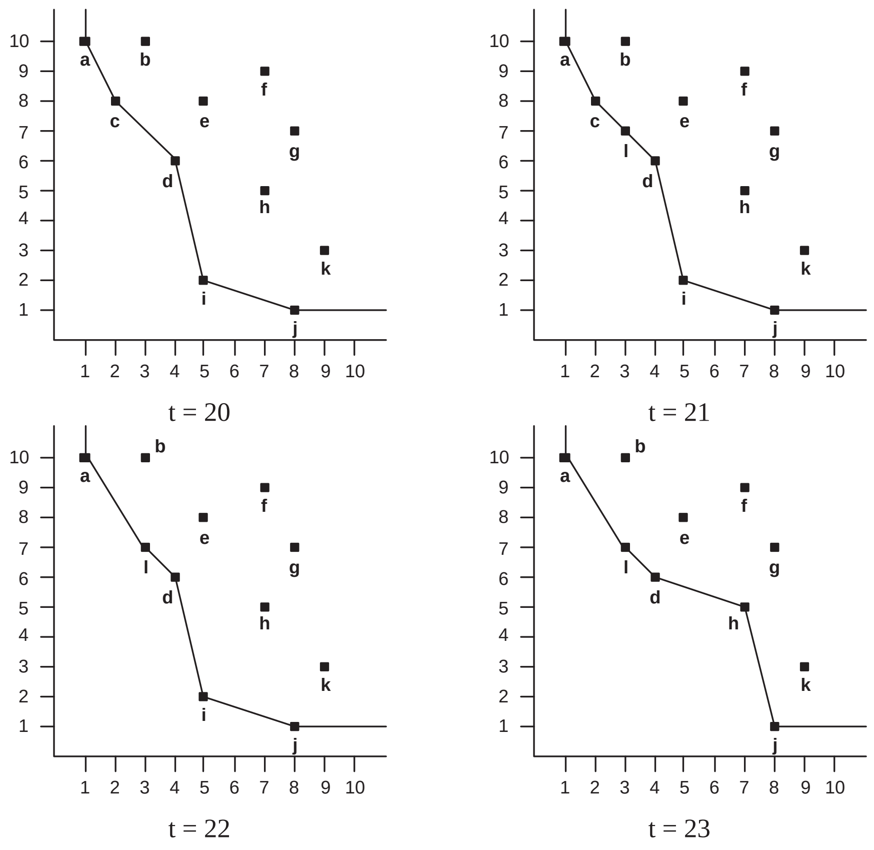
CHAPTER III

The Time Interval Skyline

3.1 Introduction

In this chapter, we begin our detailed discussions of skyline computation which we will discuss throughout the rest of this thesis, focusing on temporal skyline computation in this chapter. In the introduction, we showed that the skyline operator is a useful summarization technique for multi-attribute data sets [43]. We also showed that, if we are given a data set P that contains points p_1, p_2, ..., p_n, p_i is said to be in the skyline of P if no p_j in P dominates p_i.

Most skyline algorithms to-date assume that the data set is static, i.e. the data has no temporal element associated with it, or have dealt with temporal data only in a sliding window context, i.e. the skyline is evaluated only over the most recent n data points. In contrast, the *continuous time-interval skyline* operation involves data points that are continually being added or removed. Each data point has an arrival time and an expiration time associated with it that defines a time interval for which the point is valid. The task for the database then is to *continuously* compute a skyline for the data points that are valid at any given time. The continuous time-interval model used in this chapter is a more general one than the sliding window used in [49, 75], and hence the techniques discussed in this chapter of the thesis may also be used to evaluate such sliding window queries.



p	t_a	t_e
a	1	29
b	3	27
c	4	**22**
d	6	24
e	8	31
f	14	33
g	16	50
h	17	30
i	18	**23**
j	19	45
k	20	40
l	**21**	37

Figure 3.1: The example data with arrival and expiration times. The continuous skyline is shown in transition from time 20 to 23.

Figure 3.1 shows the difference between a conventional skyline query such as that seen in Figure 1.1 and a continuous time-interval skyline over a similar data set. Each data point has an arrival time and an expiration time, as shown in the table on the right hand side of the figure. The figure displays the skyline as it transitions from time 20 to 23. At time 20, the skyline is the same as that in Figure 1.1. The skyline changes at time 21 when data point l arrives. It is part of the new skyline. At time 22, c expires, and the skyline must be modified to remove c from both the data set and the skyline. Notice that b is not in the new skyline, since b is dominated by both a and l. At time 23, data point i expires, and the skyline is modified again, this time introducing a new point into the skyline, point h.

There are a number of emerging streaming applications that require efficient evaluation of the *continuous time-interval skyline*. If we consider the familiar example of choosing hotels, hoteliers routinely run competitive deals with booking agencies such as priceline.com. These hotel operators may wish to submit a bid for their rooms at a particular price for some specified period of time. If bookings increase, they may wish to increase the room cost, or conversely decrease it if bookings do not increase. A user interface on top of the raw priceline data may wish to show the most competitive rooms (with respect to the beach for a given price) to customers, while balancing bids from many hotel companies that all may change with time. At any given time, there may be many continuous skyline queries active in the system, depending on a number of other user preferences (such as distance from a customer-specific point of interest). In such a case, the server needs to efficiently evaluate a large number of skyline queries continuously on data points with arbitrary valid time ranges. Such an application could be useful for online hotel bookers, such as orbitz.com [3].

Another example for the use of continuous skyline evaluation is in the realm of online stock trading. Traders are interested not only in the trading price of a stock, but also in the number of shares trading hands at a price. Since trades are temporal, traders may only be interested in trades within the last hour. Hence, a mechanism for allowing trades to age out of the system after an expiration time is needed. In such a scenario, traders are interested in the skyline (price versus share volume) for many different stocks. Each stock may require a different continuous time-interval skyline operator to keep track of the latest developments. Note that in such applications there can be a large number of skyline queries that the server may need to evaluate continuously, which demand time and space efficient evaluation methods.

In this chapter, we present the first algorithm for efficiently evaluating the continuous time-interval skyline operation. We show that this new algorithm, called *LookOut*, is very scalable as it is both time and space efficient. *LookOut outperforms an iterative algorithm based on currently known methods by at least an order of magnitude in most cases!* We also compare *Lookout* with the $lazy$ and $eager$ methods of [75], and show that it performs better than either of these methods for anti-correlated data sets while evaluating a more general time model than the sliding window queries.

The other contribution that we make in this chapter of the thesis is to explore the choice of index structures for evaluating skyline operations (both in the static and the continuous cases). All previous skyline algorithms that have used spatial indices have employed the R-tree family of indices [26]. For example, the branch and bound algorithm (BBS) [59, 60] uses the R*-tree index [6]. We make an important observation that the MBR overlap involved with the R*-tree's partitioning dramatically increases the number of both index non-leaf and leaf nodes that are examined during a skyline evaluation. In contrast, the non-overlapping partitioning of a quadtree is far superior for computing skylines.

We note that an immediate question that arises with a quadtree index is that it is not a balanced indexing structure. However, it has been shown to be an effective disk-based index [22,28] and some commercial object-relational systems already support quadtrees [44]. The claim that we make and support is that if the speed of skyline computation is critical, a quadtree is far more preferable than an R*-tree. In our skyline experiments, *the quadtree index significantly speeds up skyline computation by up to an order of magnitude or more in some cases, and is never slower than the R*-tree approach*. Using the quadtree also results in smaller memory consumption during the skyline computation. We note that the issue of comparing the R-tree and quadtree for a wider range of spatial operations is beyond the scope of this chapter. Our results show that in systems that support quadtrees, using them is preferable for skyline computation.

It is also worth mentioning that the time-interval model that we use in this thesis is very flexible, and can easily accommodate more specialized streaming data models. For example, our model can be used with data sets that have no expiration time by setting the expiration time of the data in the model to infinity. Similarly, preexisting data or data that does not have any implicit start time, can simply be treated as having a start time of zero. In addition, data that does not have an explicit expiration time, but rather is valid for t seconds from its arrival can simply be handled by noting its arrival time, a, and setting its expiration time to $t + a$.

The remainder of this chapter is organized as follows: Related work is covered in Section 2, and we present our new algorithm in Section 3. In section 4, we consider the effect of the indexing structure for skyline computation. Experimental evaluations are presented in Section 5, and finally Section 6 contains our conclusions.

3.2 Related Work

Now, we discuss work related to skylines both in general, which is also related to the work done in the remaining chapters of the thesis, and specifically for temporal skyline evaluation. We will further highlight some of this related work in the remaining chapters when appropriate.

The skyline query is also referred to as the Pareto curve [61] or a maximum vector [45]. The skyline query is related to several other well-known problems that have been studied in the literature. Nearest neighbor queries were proposed by [66] and studied in [27], top-N were studied in [12], the contour problem in [56], convex hulls in [9,64], multidimensional indexing [53,71,79], and multi-objective optimization in [61,72].

The skyline algorithm was first proposed by Kung et al. [45], which employs a divide-and-conquer approach. Borzsonyi et al. [10] introduced the $skyline$ operation in a database context and showed how the standard indexing structures, B-trees and R-trees, could evaluate skyline queries. Chomicki et al. [19] formulated a generic relational-based approach to compute the skyline, based on the approach of [10]. An algorithm for high dimensional skyline computation was proposed by Matousek [55], and a parallel algorithm was proposed by Stojmenovic et al. [73].

An algorithm to obtain the skyline based on a nearest neighbors approach was introduced by Kossmann et al. [43], which uses a divide-and-conquer scheme for data indexed by an R-tree. Two algorithms were proposed in [74]. One is a bit mapped approach, and the other is an indexed approach using B-trees.

The branch and bound technique for skyline computation (BBS) was proposed by Papadias et al. in [59, 60]. It traverses an R*-tree using a best-first search paradigm, and has been shown to be optimal with respect to R*-tree page accesses. Currently, BBS is the most efficient skyline computation method, and in this paper we compare the $LookOut$ algorithm with BBS. BBS operates by inserting entries into a heap ordered by a specified distance function. At each stage, the top heap entry is removed. If it is a R*-tree node, its children are inserted into the heap. If it is a point, it is tested for dominance by other elements of the growing skyline and is either discarded or added to the skyline. This algorithm requires $O(s \cdot \log(N))$ R*-tree page accesses, where s is the number of skyline points and N is the data set cardinality. [60] also discusses skyline maintenance in the presence of explicit updates, but does not discuss time-interval skylines on streams.

Lin et al. [49] focus on computing the skyline against the most recent n of N elements in a data stream. Their approach indexes data in an R-tree and uses an interval tree to determine when a point is no longer amongst the most recent N points. They also propose a continuous skyline algorithm based around the n of N model which, similar to our algorithm, incorporates a heap to remove elements that have slipped outside the working window. But the similarities to our work end here. The window of size n necessitates a limited scope of elements in the data set and thus in the skyline as well. Consequently, there is not an explicit temporal element to the computation of the skyline. In the temporal case, which we use in this paper, the number of points under consideration is *not* restricted by any N, and at any given point in time new points may arrive, old points may expire, or any combination of the two. Consequently, with our model, the technique proposed in [49] cannot be directly applied. Data reduction in streaming environments is studied in [51].

Tao and Papadias [75] also studied sliding window skylines, focusing on data streaming environments. Their work also focuses on the most recent n window of data points. This is the most similar of the previous work to our work in this paper, and we compare the performance of the two techniques, $eager$ and $lazy$ proposed in the paper with $LookOut$.

Huang et al. [29] studies continuous skyline queries for dynamic datasets. Here, the data is moving in one or more dimensions. To efficiently evaluate continuous skyline queries in the presence of moving data, a kinetic-based data structure is developed. While this work is similar to our work because it requires the continuous evaluation of the skyline as the data changes, the data elements are moving as opposed to arriving at and expiring from the dataset. Since the data model of [75] is closer to our model, we compare $LookOut$ with its $eager$ and $lazy$ techniques.

This paper is a full-length version of the short poster paper [57].

3.2.1 BBS Example

We present the operation of BBS on the dataset shown in Figure 3.2. This dataset consists of 6 data points indexed by an R*-tree. Let us assume that each each internal R*-tree node can hold up to three entries, and that each leaf node can also hold up to three entries.

The BBS algorithm begins by inserting $R1$ into the heap that is ordered by the minimum Manhattan distance. The contents of the heap at each stage of the algorithm are

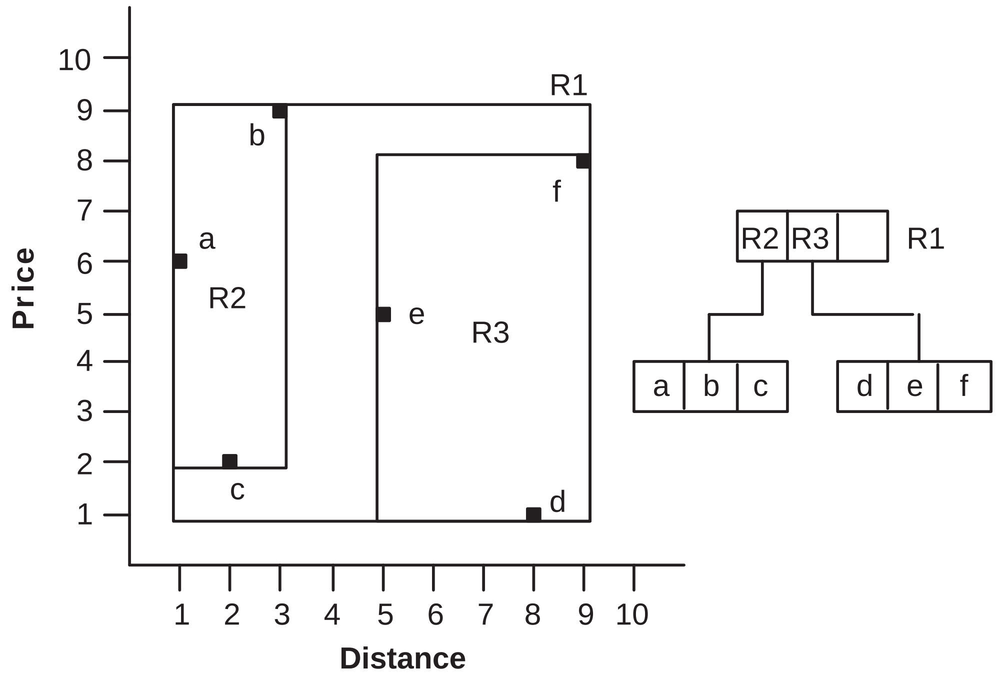


Figure 3.2: A sample dataset indexed by an R-tree used to illustrate the operation of the BBS algorithm.

shown in Table 3.1. $R1$ is popped off the heap and its children, $R2$ and $R3$, are inserted back into the heap. $R2$ has a Manhattan distance of 3, whereas $R3$ has a distance of 6, so $R2$ is popped off the heap and expanded first. The two children that compose its local skyline, c and a, are inserted back into the heap. Note that b need not be inserted back into the heap, since it is dominated by c. Since c is now at the top of the heap, it is popped off and inserted into the set of skyline points. Next, $R3$ is expanded. Two of its children, e and f, are not inserted back into the heap; e is dominated by c, and f is not part of the local skyline of $R3$. The heap now contains a and d. They are both popped off the heap and inserted into the skyline. The heap is now empty, and the BBS algorithm terminates.

3.3 The LookOut Algorithm

In this section, we present our algorithm for efficiently evaluating time-interval continuous skyline queries.

Action	**Heap Contents**	(**Skyline**)
Expand R1	(R2, 3), (R3, 6)	$\emptyset$
Expand R2	(c, 4), (R3, 6), (a, 7)	$\emptyset$
Add c	(R3, 6), (a, 7)	$\{c\}$
Expand R3	(a, 7), (d, 9)	$\{c\}$
Add a	(d, 9)	$\{a, c\}$
Add d	Empty	$\{a, c, d\}$

Table 3.1: Contents of the heap during an iteration of the *BBS* algorithm for the example dataset shown in Figure 3.2.

Algorithm 4 LookOut

```
1:  Input: Index $Tree$, Heap $tHeap$, Current Time $Time$
2:              List $Skyline$, Set $DSP$, Set $NSP$, Time $End$
3:  while $Time < End$ do
4:      if $ndp$ is a new data point then
5:          insert $ndp$ into $Tree$
6:          insert $ndp$ and expiration time into $tHeap$
7:          if isSkyline($Tree$, $ndp$) then
8:              remove points from $Skyline$ dominated by $ndp$
9:              add $ndp$ to $Skyline$
10:         end if
11:     end if
12:     while $tHeap.top.expireTime$ equals $Time$ do
13:         delete $tHeap.top$ from $Tree$
14:         if $tHeap.top$ is a skyline point then
15:             add $tHeap.top$ to $DSP$
16:         end if
17:     end while
18:     for $point \in DSP$ do
19:         $NSP \leftarrow$ MINI($point, tree$)
20:         for $t \in NSP$ do
21:             if $isSkyline$($Tree$, $t$) is $true$, add $t$ to $Skyline$
22:         end for
23:     end for
24:     update $Time$ to the current time.
25: end while
```

3.3.1 Overview

Each data point in the data set is associated with an interval of time for which it is valid. The interval consists of the arrival time of the point and an expiration time for the point. The notation for the interval is (t_a, t_e).

The skyline in the continuous case may change based on one of two events: namely, a) some existing data point i in the skyline may expire, or b) a new data point j may be introduced into the data set.

In the case of an expiration, the data set must be checked for new skyline points that may have previously been dominated by i. These points must then be added to the skyline if they are not dominated by some other existing skyline points. In the case of insertion, the skyline must be checked to see if j is dominated by a point already in the skyline. If not, j must be added to the skyline and existing skyline points checked to see if they are dominated by j. If so, they must be removed.

The $LookOut$ algorithm takes advantage of these observations to evaluate the time-interval continuous skyline. Since the skyline can change only when either a new point arrives or an old point expires, $LookOut$ maintains the current skyline S. A data point p is inserted into a spatial index at time t_a. This point is checked to see if it is in the skyline, and if so, S is updated. If p is dominated, no changes are made to S. When t_e arrives, p is removed from the dataset and deleted from the spatial index. At this time, the dataset is checked to see if any of the points dominated by p are now elements of the skyline. If so, these points are added to S.

$LookOut$ takes advantage of two important properties of hierarchical spatial indices, such as the R-tree family of indices and the quadtree.

1. If p dominates the all corners of a node o (and hence dominates the entire region bounded by the node), then p dominates all of the points contained in o and its children.

2. If all of the corners of a node o dominates a point p (and hence the entire region bounded by the node dominates p), then all of the points contained in o and its children dominate p.

These two observations are later used to prune nodes of the index and to discard new points from skyline consideration by $LookOut$.

Algorithm 5 IsSkyline

```
1:  **Input:** Point $P_{new}$, Index node $Tree$
2:  insert $Tree$ into a heap $BHeap$, with distance 0.
3:  **while** BHeap isn't empty **do**
4:    $Tree \leftarrow$ pop of $BHeap$
5:    **if** $Tree$ is a leaf node **then**
6:      check if one of the entries of $Tree$ dominates $P_{new}$
7:      if so, return false. Otherwise, continue
8:    **else**
9:      **for** $Child \in$ the non-empty children of $Tree$ **do**
10:       **if** minimum corner of $Child$ does not dominate $P_{new}$ **then**
11:         continue
12:       **else**
13:         **if** maximum corner of $Child$ dominates $P_{new}$ **then**
14:           return false
15:         **else**
16:           insert $Child$ into $BHeap$
17:         **end if**
18:       **end if**
19:     **end for**
20:   **end if**
21: **end while**
22: return true
```

3.3.2 Algorithm Description

$LookOut$ may be used with any underlying data-partitioning scheme. In our implementation, we chose to use and evaluate both the R-tree [6] and a disk-based PR quadtree [70]. We use the R-tree because of its ubiquity in multidimensional indexing and its use in other static-data skyline algorithms such as [59]. The quadtree index uses a regular non-overlapping partitioning of the underlying space, and is more effective in pruning portions of the index that need not be traversed for skyline computation (a discussion of these tradeoffs is presented in Section 3.4).

The $LookOut$ algorithm is presented in Algorithm 4. As seen in line 4, when a new data point arrives, $LookOut$ first stores the item into the spatial index. Each data element is also inserted into a binary heap (line 6) that is ordered on the expiration time. This heap is used so that data can be removed from the system when it expires. The element is then checked to see if it is a skyline point by the $isSkyline$ algorithm (line 7), which will be explained shortly. If so, the new point is added to the skyline and those skyline points it dominates are removed. As time passes, the minimum entry in the binary heap is checked to see if its expiration time has arrived (line 12) and, if it has, it is deleted from the index. The skyline points themselves are maintained in a list, so that they may be returned immediately in the event of a skyline query over the data set. (The skyline points can also be stored in an index, but the skyline is small in size and the index overhead often mitigates the benefits of using the index.) A separate heap, ordered on the expiration time, is also maintained for the skyline points so that an expired skyline point may be quickly removed. Those points that have been removed from the skyline (line 18) leave possible gaps that need to be filled by currently available data. The $MINI$ algorithm finds the mini-skyline of points that were dominated by a deleted skyline point and effectively plugs a hole left by a deleted skyline point. Some and possibly all of the points found by $MINI$ may be dominated by some other skyline point. Before adding them to the skyline, $LookOut$ tests if each is in fact a new skyline point with $isSkyline$ (line 21).

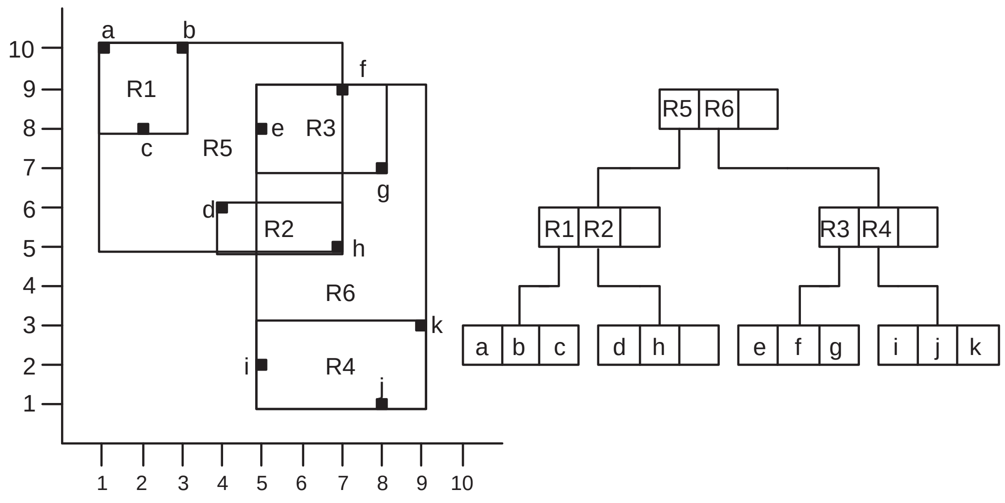


Figure 3.3: An R-tree depicting the data set in Figure 1.1.

The $isSkyline$ algorithm is shown in Algorithm 5. It uses a best-first search paradigm, which is also used in BBS. The index nodes are inserted into a heap based on distance from the origin. When expanding a node in the heap (line 4 of Algorithm 5), the $isSkyline$

Algorithm 6 MINI

```
1: Input: Point $P_{sky}$, Index node $Tree$
2: Output: skyline $miniSkyline$
3: insert $Tree$ into heap $BHeap$, with distance 0.
4: while BHeap isn't empty do
5:   if $BHeap.top$ is a point then
6:     $point \leftarrow$ pop $BHeap$
7:     $pIsDominated \leftarrow FALSE$
8:     for each element $a$ in $miniSkyline$ do
9:       if $a$ dominates $point$ then
10:        $pIsDominated \leftarrow TRUE$
11:      end if
12:    end for
13:    if $pIsDominated$ is $FALSE$ then
14:      insert $point$ into $miniSkyline$
15:    end if
16:  else
17:    $Tree \leftarrow$ pop of $BHeap$
18:    if $P_{sky}$ dominates the maximum corner of $Tree$ then
19:      if $Tree$ is a leaf node then
20:        find the local skyline of just $Tree$
21:        for $point \in$ the local skyline of $Tree$ do
22:          if $P_{sky}$ dominates $point$ then
23:            insert $point$ into $BHeap$.
24:          end if
25:        end for
26:      else
27:        for $Child \in$ the non-empty children of $Tree$ do
28:          insert $Child$ and $Child$'s distance into $BHeap$
29:        end for
30:      end if
31:    end if
32:  end if
33: end while
```

algorithm discards any child node n whose lower left corner does not dominate P_{new} (line 10). This is because any data point in any such n cannot possibly dominate P_{new}, so for the purposes of skyline testing, it can be discarded. If the upper right corner of the child node (which isn't empty) dominates P_{new}, the algorithm can terminate and answer $false$ (line 14). If the node is a leaf (line 5), the elements are compared against P_{new} for dominance. If any such element dominates P_{new}, the algorithm terminates and answers $false$. If the heap ordered on the minimum distance from the origin is ever empty, the algorithm answers $true$.

$MINI$, seen in Algorithm 6, is also a best-first search algorithm and maintains a binary heap. It takes as input a deleted skyline point P_{old}, which must dominate all points under

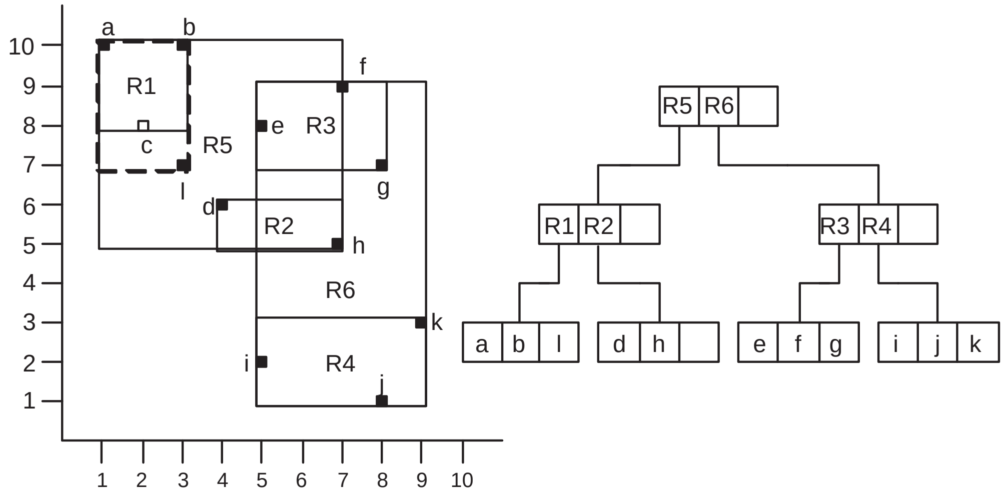


Figure 3.4: An R-tree following the changes made to the data set in Figure 3.1 up to time 22.

consideration. It operates by popping the top element off the heap and inserting its children back into the heap, provided they are not dominated by the growing skyline. It has the extra caveat that all elements it inserts into the heap must be dominated by P_{old}. The algorithm begins by checking if the top heap element is a point (line 5). If the point is dominated by the growing mini skyline, it is ignored; else, it is added to the mini skyline (lines 8-15). If the upper right corner of any internal node is not dominated by P_{old}, then it may be discarded (line 18). If the top of the heap is a leaf (line 19), its local skyline is added to the heap (line 23). If the top is an internal node, those elements which have their upper right corner dominated by P_{old} are inserted back into the heap. $MINI$ terminates when the heap is empty.

3.3.3 Example

We now consider an example execution of the $LookOut$ algorithm on the data set shown in Figure 3.1. Figure 3.1 depicts the example data set beginning at time 20; at this time, the data points in an R-tree might resemble Figure 3.3. Let us assume that each internal R-tree node can hold up to three entries, and that each leaf can also hold three. When l arrives at time 21, the $isSkyline$ algorithm is run to determine if l is a skyline point. First, the root node of the R tree is accessed, and $R5$ is inserted into the heap, with a Manhattan distance (from the origin) of 6. $R6$ is not placed into the heap because its lower left corner does not dominate l. Thus, it may be ignored for the purposes of $isSkyline$. The top of the heap is then popped and processed. $R5$ contains two child nodes, $R1$ and $R2$, but since the lower left corner of neither of these dominates l, they are both discarded. Since the heap is empty, l is added to the skyline. The new node must also be inserted into the R-tree as well.

Action	**Heap Contents**	**(MINI Skyline)**
access root	(R5, 6), (R6, 6)	$\emptyset$
Expand R5	(R6, 6), (R1, 8)	$\emptyset$
Expand R6	(R1, 8), (R3, 12)	$\emptyset$
Expand R1	(R3, 12), (b, 13)	$\emptyset$
Expand R3	(b, 13), (e, 13), (f, 16)	$\emptyset$
Add b	(e, 13), (f, 16)	$\{b\}$
Add e	(f, 16)	$\{b, e\}$
remove f	Empty	$\{b, e\}$

Table 3.2: Contents of the heap during an iteration of the ***MINI*** algorithm.

At time 22, c expires, and must be removed from the data set. Following its removal from the index, the R-tree appears as shown in Figure 3.4. The heap element identifies c as a skyline point. Since c is a skyline point, its removal may mean that preexisting data points must be added to the skyline, so the $MINI$ algorithm is run. The contents of $MINI$'s heap are depicted in Table 3.2. $MINI$ begins by accessing the R-tree root. It adds $R5$ to its heap along with its Manhattan distance (6) and $R6$ with its Manhattan distance (6). Node $R5$ is removed and expanded; the only child of $R5$ that is added to the heap is $R1$, since the upper right corner of $R2$ is not dominated by c. $R6$ is next expanded, since its Manhattan distance is the smallest of any point or node in the heap, and $R3$ is added with a distance of 12. Next, $R1$ is expanded and b is added to the heap. None of the other children of $R1$ are added since they are not dominated by c. $R3$ is expanded and e and f are added to the heap. This ultimately produces b and e as the $MINI$ skyline for entry c. Note that f is not included, since it is dominated by e. The $isSkyline$ algorithm must now be called for both b and e to test if they are in fact skyline points. Neither one is; e is dominated by d and b is dominated by l. Therefore, no skyline change is required with the deletion of c.

3.3.4 Analysis of $LookOut$ in Comparison to BBS

In this section, we examine the quantitative cost of *LookOut* and compare it against the cost of an iteration of the BBS algorithm. Note that since there are no current algorithms for continuous skyline evaluation, repeatedly running BBS can be considered to be the best alternative to $LookOut$. We observe that the only operations that can affect the skyline (and hence the cost of $LookOut$), are either an insert operation or a delete operation. During time intervals when one of these two operations do not occur, the skyline remains the same and $LookOut$ performs no work. During this analysis, we consider indexing with an R-tree.

To determine the cost of an insertion, the costs of several operations need to be evaluated. These operations are: a) the cost of adding an entry to the expiration-time heap, b) the cost of adding an entry to the indexing structure, and c) the cost of running the $isSkyline$ algorithm, to determine if the new point is in the skyline.

The costs of both adding an entry to the heap and of inserting an entry into an index structure are identical for both $LookOut$ and BBS. Consequently, neither one of these operations make $LookOut$ perform either better or worse than BBS. The real difference between the two lies in the cost savings that $isSkyline$ obtains over BBS.

First, we consider the worst case cost of BBS relative to the worst case cost of $isSkyline$ for a signle insertion operation. For BBS, the worst case occurs if all data points are in the skyline. In this case, all of the leaf and non-leaf nodes of the R-tree are inserted into the heap that BBS uses to order elements based on their minimum L1-norm distances. Each data point is also inserted into this heap when their respective leaf nodes are expanded. Since BBS checks each element removed from the heap relative to the growing skyline, this worst case cost is $O(n^2)$.

The worst case for $isSkyline$ for a single insertion occurs if the new data point p that has been inserted overlaps with all leaf and non-leaf nodes of the R-tree. If this occurs, all of the leaf and non-leaf nodes of the R-tree are inserted into the heap ordered on the minimum distances to the origin. Each non-leaf or leaf node is inserted into the heap only once, based on the distance of the node from the origin. For example, the root node of the tree is inserted into the heap, and then expanded. Its children are then inserted into the heap. The root node is never considered by the algorithm again. Each of the nodes in the heap are expanded exactly once and only once, resulting in their children being inserted into the heap. When each leaf node is expanded, the entries in it are compared with the new data point. Each non-leaf or leaf node and each data point are compared in the worst case at most once with the new data point p. In the worst case, all of the entries in the data set are compared with this new point, producing a worst case cost of $O(n)$ comparisons. For example, consider the case when all n data points are elements of the skyline and the new data point overlaps with the leaf level nodes of the tree that contain these points. Then, to determine if the new data point is in the skyline, all n nodes in the dataset will be compared with the new data point.

Second, we compare the average case cost of BBS to the average case cost of $isSkyline$. Since $isSkyline$ only tests whether a single point is dominated by an existing point or not, whereas BBS computes an entire skyline from scratch, the cost savings is dependant on the number of elements in the skyline that BBS evaluates. If this number of skyline points is s, then the average case cost of $isSkyline$ relative to that of BBS is approximately $1/s$.

To determine the cost of a single deletion operation, the costs of the following operations must be evaluated: a) the cost of removing an entry from the expiration-time heap, b) the cost of removing an entry from the indexing structure, and c) the cost of running the $MINI$ algorithm, to determine if alternate points must be added to the skyline.

The costs of both removing an entry from the heap and of removing an entry from the index structure are identical costs, regardless of whether $LookOut$ or BBS is computing the skyline. Consequently, neither one of these operations make $LookOut$ perform either better or worse than BBS. The real difference between the two for a deletion lies in the cost savings that $MINI$ obtains over BBS.

Next, we consider the worst case cost of BBS relative to the worst case cost of $MINI$ for a single deletion. For BBS, the worst case for a deletion is the same as it was in the case of an insertion and occurs if all data points are in the skyline. This worst case cost is $O(n^2)$.

The worst case cost for $MINI$ occurs if the deleted data point is the only element in the skyline. In this case, $MINI$ must evaluate a completely new skyline from scratch. The worst case for $MINI$ then is the same as the worst case cost of BBS, which is $O(n^2)$.

Next, we compare the average case cost of BBS to the average case cost of $MINI$. Since $MINI$ evaluates the skyline relative to a removed skyline point, the cost savings is dependant on the number of elements in the new skyline that were previously dominated by the removed skyline point. If this number of skyline points is s' and the total number of skyline points is s, then the cost of $MINI$ relative to that of BBS is s'/s.

Therefore, the qualitative cost of using $LookOut$ is less than that of iteratively running the BBS algorithm for continuous skyline computation.

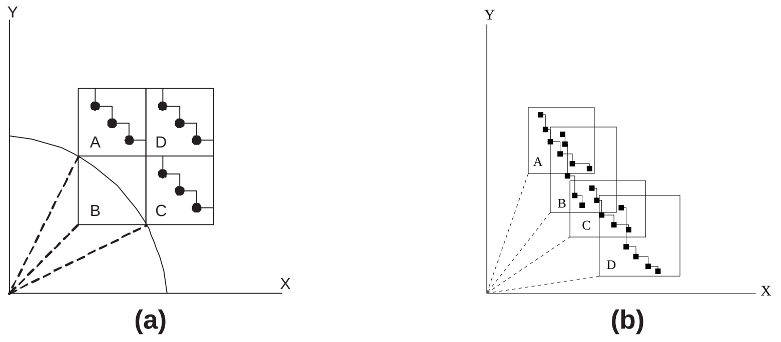


Figure 3.5: Quadtree (a) and R*-tree (b) nodes with local skylines. Distances to each represented by dashed lines.

3.4 Choice of Indexing Structure

In this section, we examine how the choice of the index can impact the performance of both static and continuous time-interval skyline performance. This section examines some of the differences between the ubiquitous R*-tree which has been used for a number of the previously proposed skyline algorithms [43, 59], and the quadtree, which is more efficient for computing skylines. The quadtree has the following two advantages over the R*-tree for evaluating continuous time-interval skylines: 1) Insertion into a quadtree is faster than into a R*-tree, and 2) The quadtree-based traversal reduces the maximum number of heap elements during the best-first search. (The second advantage also applies to skyline computation over static data sets.)

The rationale behind the first point involves the complex node split operation of the R*-tree that involves various sorting and grouping operations on index entries. In contrast, the split operation of the quadtree is much simpler, and merely divides the node in each dimension in half. For point data, such as that managed in skyline queries, the superior performance of the quadtree on inserts and updates has been noted in a study of a commercial DBMS [44]. This study of Oracle Spatial shows that quadtrees are significantly faster for index creation and updates of point data.

The intuition driving the second point above is as follows: First, each time the R*-tree splits, its children are likely to overlap. No dominance relationship can be established between two overlapping leaf or non-leaf nodes, so neither will be able to prune the other from future consideration. Hence, both will be inserted into the heap. Contrast this with the node split of the quadtree, where no overlap exists, and at least one child is automatically dominated (and pruned) each time a split is performed[1]. Second, nodes in the quadtree will produce quite different distances from the origin for their internal data. This is because each quad occupies a region of space derived only from the structure of the quadtree, and not from the data as in the case of the R*-tree. This means that the children of one quad will be fully expanded and mostly removed from the heap before the data contained in neighboring leaf and non-leaf nodes is entered into the heap.

To understand the heap size reduction benefit of quadtrees, consider the example shown in Figure 3.5a. Nodes A, B, and C are inserted into the heap when their parent node is expanded. D is not inserted, because it is automatically dominated by B. B is the first node popped from the heap, and its local skyline points are inserted back into the heap and ordered by the distance function. The distance that A and C are from the origin is represented by the quarter circle. Note that most of the area of B lies within this quarter circle. Any entries in B that lie within this circle are processed and removed from the heap before either A or C is expanded, thus resulting in a smaller heap. Contrast this to the worst case performance of the R*-tree, seen in Figure 3.5b. A, B, C, and D are added to the heap with similar distances. Hence, each is expanded in sequence before any of their individual data elements are processed.

[1]A question that a reader may ask is why not consider an R+-tree instead of a quadtree. While a full exploration of this issue is beyond the scope of this paper, the quick answer is that the R+-tree does not guarantee the pruning property of the quadtree, which is critical to the efficiency for skyline computation. The R+-tree only addresses the non-overlapping problem of the R*-tree, but at the expense of lower page occupancy.

3.5 Experimental Evaluation

In this section, we present experimental results comparing $LookOut$ with BBS, the best known method for computing skylines. We compre the quadtree with the R*-tree [6], a variant of the R-tree. We first present results showing that for static skylines, using the quadtree significantly improves the performance over using an R*-tree. We also show that the heap size is smaller when using the quadtree compared to the R*-tree, implying that a smaller amount of memory is needed for computing skylines with the quadtree approach. (A low memory consumption is critical in streaming environments in which the system is evaluating multiple skylines concurrently.) We then present results for $LookOut$ with the time-interval continuous skyline model.

3.5.1 Experimental Study Goal

In this study, our goal is to compare the performance of the R*-tree and the quadtree for skyline query evaluation. The R*-tree is chosen because indexed skyline query algorithms discussed previously have focused exclusively on the R-tree family of indices. Quadtrees have been shown to manage point data more effectively than the R-tree family in several notable experimental studies [40, 44]. Since skyline queries deal exclusively with point data, it is for this reason we have chosen the quadtree as the best alternative. For a broader comparison beyond the scope of skyline queries for indices in the R-tree family and the quadtree, the interested reader may consult [40, 44].

3.5.2 Data Sets and Experimental Setup

The choice of data sets for experimental evaluation is always a challenging task. While the use of real data sets is preferable, a few selected real data sets don't necessarily bring out the effect of a range of data distributions. Luckily for skyline methods, it has been recognized that there are three critical types of data distributions that stress the effective-

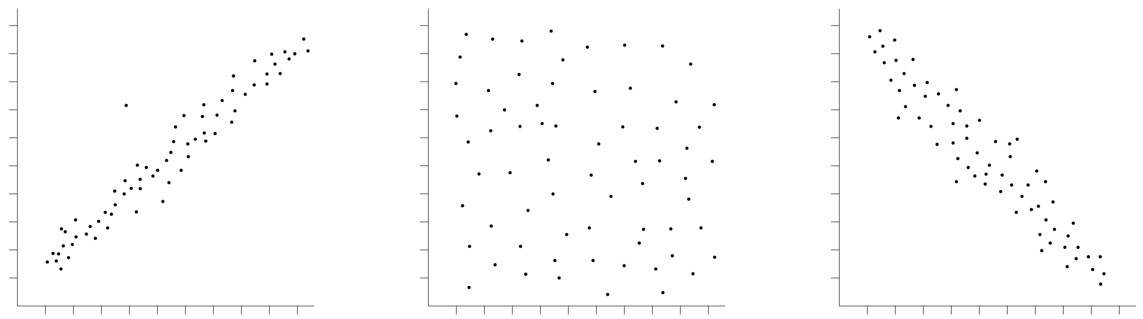


Figure 3.6: Two dimensional examples of (a) correlated data, (b) independent data, and (c) anti-correlated data.

ness of skyline methods [10]. These three distributions are independent, correlated, and anti-correlated. The correlated data set is considered the easiest case for skyline computation since a single point close to the origin can quickly be used to prune all but a small portion of the data from consideration. The anti-correlated data set is considered the most challenging of the three for skyline computation. This is because points in the skyline dominate only a small portion of the entire data set. Larger numbers of skyline points exist for anti-correlated data for a given cardinality relative to either the independent or correlated cases.

To begin the discussion, first consider the different types of data distributions and the varying effects that these distributions have on the cost of computing the skyline operation. The two dimensional case for each of the common data distributions that have been extensively considered in previous work are shown in Figures 3.6 a, b, and c. Only a small portion of the data (and hence only a small part of the data in the index) will be considered during the skyline evaluation of the correlated and independent cases, since each has a data point or points near the origin for sufficiently large cardinality values. These points will dominate all or most of the remaining points in the dataset, quickly pruning away the majority of the data from skyline consideration. The anti-correlated data set is more challenging for skyline algorithms because it produces more skyline points for a given dataset cardinality (on average) than the other distributions. Hence, a greater number of points are considered for inclusion in the skyline, which means that more leaf-level nodes and inner nodes of a spatial index must be traversed by a skyline algorithm. While real datasets may have distributions that differ from these benchmarks, these three distributions present a wide and diverse range of distributions to test the performance of skyline algorithms.

Following well established methodology set by previous research on skyline algorithms [49, 59], we choose to use these three data distributions. We also test our methods on a variety of other data set parameters such as data cardinality and dimensionality. For generating these synthetic data sets, we use the skyline generator generously provided by the authors of [10]; using this, we created a number of data sets varying in cardinalities from 100K to 5M in two dimensions, for the three distributions already mentioned. We also created data sets varying the dimensionality between 2 and 5 while holding the cardinality fixed at 1M entries. We test with these dimensionalities because they have been commonly tested elsewhere for indexed skyline operations [59, 75].

Our experimental platform is built on top of the SHORE storage manager [11], which provides support for R*-trees. We also implemented a quadtree indexing method in SHORE. Our quadtree implementation uses a simple mapping of the quadtree nodes to disk pages. Each leaf level quadtree node is one page in size. Non-leaf nodes are simply implemented as SHORE objects, that are packed into pages in the order of creation.

We implemented both BBS and *LookOut* on top of the SHORE R*-tree and our quadtree index implementation in SHORE. To maintain consistency with the previous approach by Papadias et al. [59], and for ease of comparison, we set the R*-tree node size at 4KB for both leaf and non-leaf nodes. This results in R*-tree leaf node capacities of between 330 and 165 data entries for dimensions 2 and 5, respectively. The linear splitting algorithm is choosen for the nodes of the R*-tree. The non-leaf node capacities vary between 110 for 2 dimensions and 66 for 5 dimensions. We also used a 4KB page size for our quadtree implementation, resulting in leaf node capacities that varied between 424 for 2 dimensions to 131 in 5 dimensions. The leaf node utilization for the R*-tree is 71 percent for 2 dimensional data for both the independent and anti-correlated datasets and 74 and 73 percent for 5 dimensional data for the independent and anti-correlated datasets, respectively. The leaf node utilization for the quadtree is 61 and 53 percent for 2 dimensional data for the independent and anti-correlated datasets, respectively, and 30 and 13 percent for 5 dimensional data for the independent and anti-correlated datasets, respectively. We use a buffer pool size of 128 MB. For the BBS implementation, we followed the algorithm described in [59], and added the local skyline optimizations described in [60].

All experiments were run on a machine with a 1.7GHz Intel Xeon processor, with 512MB of memory and a a 40GB Fujitsu SCSI hard drive, running Debian Linux 2.6.0.

3.5.3 Anti-Correlated Datasets in d Dimensions

The anti-correlated datasets used throughout the experimental section of this chapter as well as later chapters of this thesis are generated using the technique of [10]. This method used to generate these datasets is shown in Algorithm 7.

This method first chooses a value v drawn from a normal distribution, with mean 0.5 and variance 0.25 (line 2 of Algorithm 7). The closest distance that v is from either extreme of the universe, either 0 or 1 depending on which is closer, is determined in lines 3-7 and stored in q. The value of the generated point in each dimension is initialized to v (line 8). Next, uniformly distributed random values are chosen from the interval $(-q, q)$, one for each dimension d (lines 9-13). For a particular dimension d_m, one such uniformly distributed random value is added to $p[d_m]$ and subtracted from $p[d_{m+1}]$.

This results in attribute values that are chosen so that each pair of consecutive dimensions are anti-correlated with respect to each other. In practice, this results in consecutive

Algorithm 7 Anti-Correlated Dataset Generation.

```
1:  **Output:** Anti-correlated point $p$
2:  $v$ = RandomNormal(0.5,0.25)
3:  **if** $v < 0.5$ **then**
4:      $q$ = $v$
5:  **else**
6:      $q$ = 1 - $v$
7:  **end if**
8:  Initialize $p[d]$ = $v$ for all $d \in Dimensions$;
9:  **for** $d < Dimensions$ **do**
10:     $h$ = RandomEqual(-$q$,$q$)
11:     $p[d]$ += $h$
12:     $p[(d+1)\ mod$ Dimensions] -= h
13: **end for**
14: if $p[d] < 0$ or $p[d] > 1$ for any $d \in Dimensions$, repeat.
```

dimensions d_m and d_{m+1} having a correlation coefficient of approximately -0.48, while d_m and d_{m+2} have a correlation coefficient that is typically less than $|0.03|$.

In summary, an anti-correlated dataset chosen in this way will hence have anti-correlated pairs of consecutive attributes, while nonconsecutive attributes will be nearly uncorrelated.

3.5.4 R*-tree v/s Quadtree for Skyline Computation

In this section, we examine the effect of the underlying index structure on the performance of *static* skyline computation. In other words, we show the effect of the choice of index on the BBS algorithm [59, 60]. We focus on two different properties that affect skyline query performance: the data set dimensionality and cardinality. This methodology is consistent with the performance study of [59]. In the interest of space, we only present results for the anti-correlated and independent cases, which is also consistent with previous studies [59, 60].

We measure the number of page accesses in our experiments instead of the disk access cost (DAC) [54] because the DAC is a measure of the number of all nodes of a tree that are read during a query. For members of the R-tree family, this closely matches the number of page accesses, since R-tree nodes are mapped directly to pages. For inner nodes of a packed quadtree, this is not the case since many inner nodes can be mapped to one single page. Hence, we measure page accesses as a more fair comparison for the work done by both data structures.

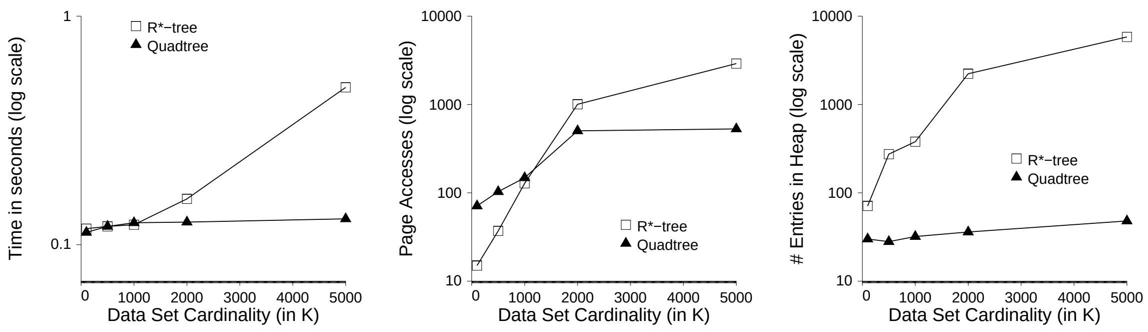


Figure 3.7: Experimental results for the independent data distribution in 2 dimensions for varying cardinality. Graphs show (a) the execution time, (b) the page accesses, and (c) the maximum heap size.

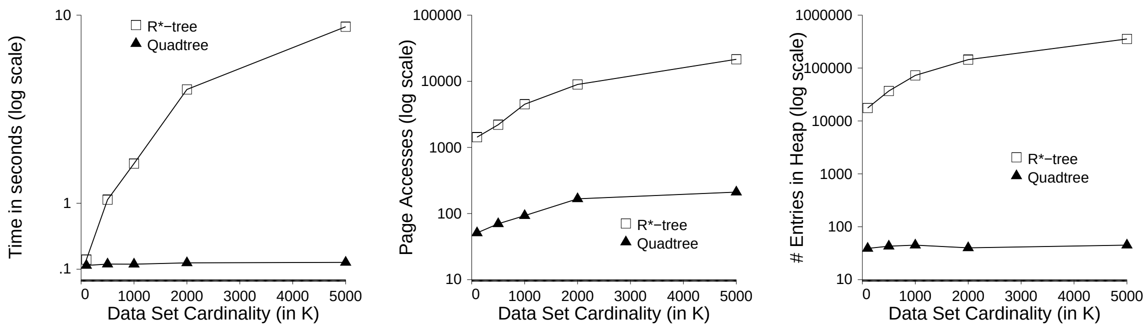


Figure 3.8: Experimental results for the anti-correlated data distribution in 2 dimensions for varying cardinality. Graphs show (a) the execution time, (b) the page accesses, and (c) the maximum heap size.

Effect of Cardinality

In this experiment, we explore the effect of the data cardinality. Following the approach in [59], we fix the dimensionality at 2, and vary the cardinality between 100K and 5M. As in [59], we report three different graphs for each experimental setting: the CPU time vs. cardinality, the maximum size of the heap vs. cardinality, and the number of page accesses vs. cardinality. The results for this experiment are shown in Figures 3.7 a, b, and c.

Figures 3.7a and 3.8a present the execution times for varying cardinality. In the independent case (Figure 3.7a), both the R*-tree and quadtree based methods are comparable until the data set size is over one million entries; for the larger data sizes, the quadtree approach is significantly faster. For the anti-correlated data set (Figure 3.8a) the quadtree approach is significantly faster, and its relative performance improves as the data cardinality increases – for the 5000K data it is two orders of magnitude faster than the R*-tree approach. This difference occurs because of the lower page accesses for the quadtree and smaller maximum heap size.

In Figure 3.7b, we notice that the number of page accesses for the independent case for the R*-tree is 2-4 times that of the quadtree for the 2M and 5M data file sizes. For these two file sizes, the quadtree outperforms the R*-tree by significant amounts (see Figure 3.7b).

From Figure 3.8b, we observe that the R*-tree performs about an order of magnitude more page accesses than the quadtree. These increased page accesses are attributable to the better pruning techniques of the quadtree caused by the node overlaps of the R*-tree (as discussed in Section 3.4). The quadtree accesses fewer leaf and non-leaf nodes than the R*-tree because it can prune away more nodes that are dominated by the discovered skyline points. As a side note, for the 100K data size the quadtree approach actually performs a few more reads (31 versus 21), which is attributable to a larger tree height for the quadtree (5 versus 3) relative to the R*-tree index, and the relatively simple packing of quadtree nodes in our implementation.

The maximum heap sizes in Figure 3.7c and Figure 3.8c show a large improvement for the quadtree method for all file sizes, since the quadtree is accessing fewer leaf and non-leaf nodes than the R*-tree due to its non-overlapping space partition. In addition, the nodes that it does access are processed much more serially than the R*-tree, whose overlapping leaf and non-leaf nodes are expanded into the heap at similar times because they have similar distances from the origin. This results in the fewer page accesses and the smaller maximum heap size for the quadtree (see Section 3.4 for the detailed analysis).

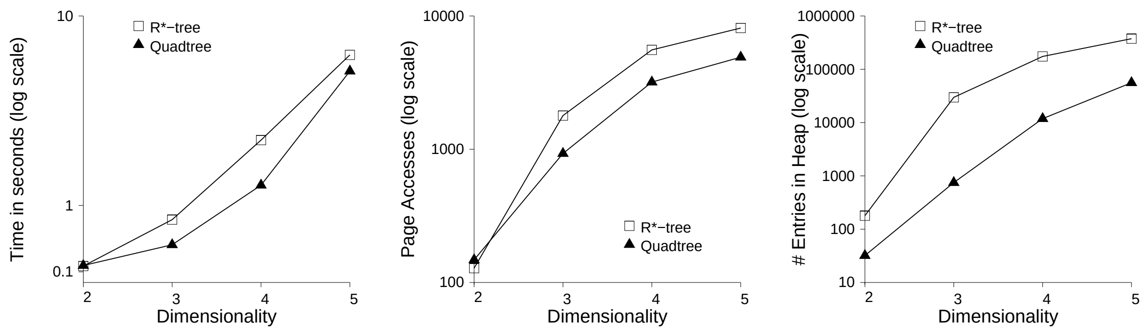


Figure 3.9: Experimental results for the independent data distribution with fixed dataset cardinality (1M tuples) for varying dimensionality. Graphs show (a) the execution time, (b) the page accesses, and (c) the maximum heap size.

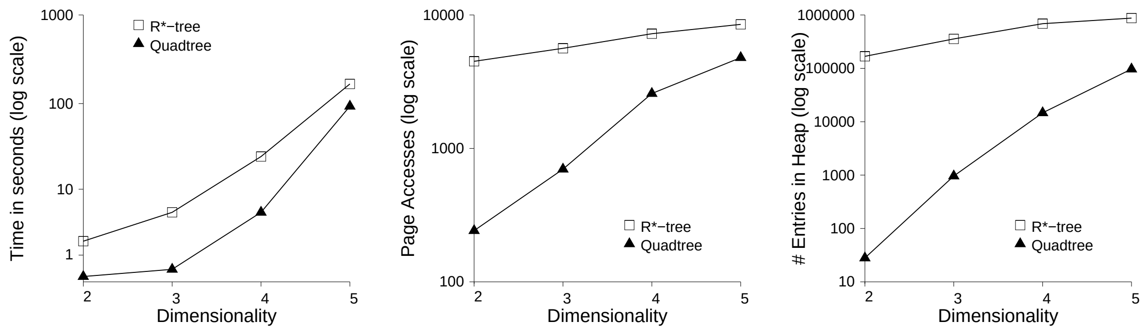


Figure 3.10: Experimental results for the anti-correlated data distribution with fixed dataset cardinality (1M tuples) for varying dimensionality. Graphs show (a) the execution time, (b) the page accesses, and (c) the maximum heap size.

The Effect of Dimensionality

In this experiment, we examine the effect of data dimensionality. As in [59], we fix the data cardinality at 1 million tuples, and vary the dimensionality from 2 to 5. The results for this experiment are shown in Figures 3.9 and 3.10.

The execution time graphs for increasing dimensionality are seen in Figure 3.9a for the independent and Figure 3.10a for the anti-correlated data sets. For the independent case (Figure 3.9a), the quadtree is about 2-4 times faster when dimensionality is higher than 2. For the anti-correlated data set (Figure 3.10a), the quadtree is more than an order of magnitude faster than the R*-tree when the dimensionality is lower than five. These benefits are because the quadtree approach incurs significantly fewer pages accesses and has fewer entries in the heap.

In Figure 3.9b, the number of page accesses for the independent case for the R*-tree is similar for 2 dimensions. As seen in the previous section, the quadtree and R*-tree had comparable performance for 2 dimensions when the cardinality was less than 2M. For higher dimensionality, the quadtree obtains cost savings of about 2-3 times. This is attributable to the higher chance for dead space and overlap amongst R*-tree nodes as dimensionality increases, which exposes the relative superior pruning of the quadtree, leading to more page accesses and heap accesses for the R*-tree (as discussed in Section 3.4).

In Figure 3.10b, the R*-tree performs about an order of magnitude more page accesses than the quadtree in two and three dimensions, about three times more page accesses in four dimensions, and about twice as many page accesses in five dimensions. There are two competing factors at work here. First, the superior pruning of the quadtree results in a lower number of page accesses, relative to the R*-tree. Second, the increasing dimensionality means that more skyline points exist for the anti-correlated data set and the quadtree has to access more data pages to find them all. The R*-tree accesses slightly more pages as well (about twice as many in five dimensions as in two), but the fact that it was already accessing so many in two dimensions means that the increase in the rate of page access with dimensionality is not as remarkable as that of the quadtree.

The maximum heap size in Figure 3.9c shows a savings of about an order of magnitude for the quadtree over the R*-tree. This is again attributable to the pruning techniques of the quadtree, as previously discussed.

Figure 3.10c shows a similar trend for heap size as dimensionality increases as was witnessed for the number of page accesses. The same two competing factors are causing this. First, the superior pruning of the quadtree gives rise to a smaller maximum heap size. Second, the increasing rate of page accesses with dimensionality means more pages will have similar distances as the data set fans out. Thus, more pages will be expanded and insert their entries into the heap at about the same time (see Section 3.4 for details).

Summary

In summary the quadtree index is a much more efficient indexing structure than the R*-tree for computing skylines. The benefits of using a quadtree are generally higher for larger data sets, and for lower dimensionality. In many cases, the quadtree approach is more than an order of magnitude faster than the R*-tree approach. The benefits are the most significant for the anti-correlated data set. In addition to being fast, the quadtree approach also results in a significantly smaller maximum heap size.

3.5.5 The Continuous Time-Interval Skyline

In this section, we examine the performance of *LookOut* relative to a naive method of executing the BBS algorithm to compute the skyline whenever anything changes. This method is referred to as $CBBS$, and can be considered the best alternative method for computing continuous skylines.

For this experiment, the data structures are entirely memory resident, to mimic the application of continuous skyline in streaming applications where such main memory assumptions are common and often the preferred environment (for example [49] also assumes that there is enough main memory). In the naïve $CBBS$ case, a binary heap ordered on data point expiration time is maintained, so that when a point expires, it can be deleted and the BBS skyline algorithm run to reevaluate the skyline. Whenever a data entry arrives, it is inserted into both the heap and the R*-tree, and the skyline is reevaluated by rerunning BBS.

As before, we used synthetic data sets and vary both the cardinality and the dimensionality. For the dimensionality tests, we vary *d* between 2 and 5 and fix the cardinality at

10K entries. For the cardinality test, we fix *d* at 2 and vary the data set cardinality from 10K to 50K.

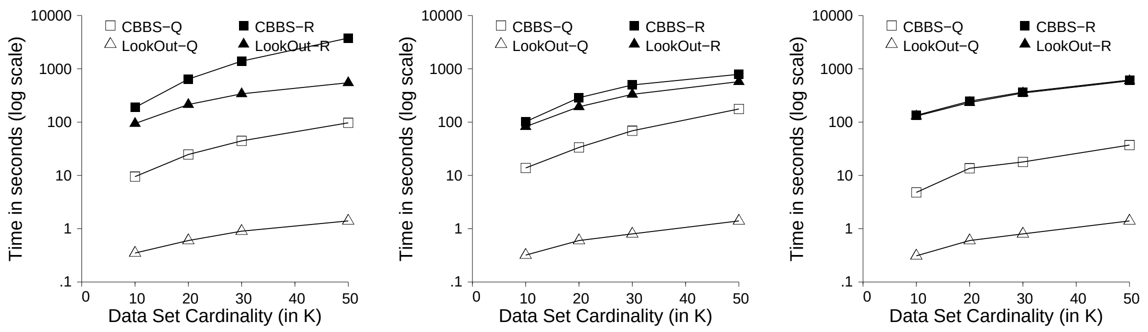


Figure 3.11: Experimental results for the time-interval continuous skyline with random time interval length in 2 dimensions with varying cardinality. Graphs show (a) the anti-correlated, (b) the independent, and (c) the correlated cases.

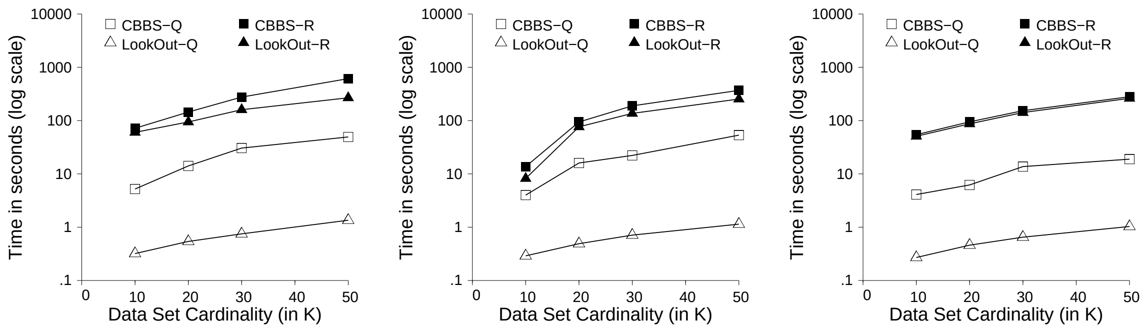


Figure 3.12: Experimental results for the time-interval continuous skyline with 1-10 % time interval length in 2 dimensions with varying cardinality. Graphs show (a) the anti-correlated, (b) the independent, and (c) the correlated cases.

Two different techniques are used to assign data points an arrival and an expiration time, and results for both techniques are presented. For the first technique, we randomly pick an arrival time between 0 and 100K. Then, we pick the departure time randomly between the arrival time and 100K. For the second technique, data points are again assigned an arrival time between 0 and 100K, but the expiration time is chosen randomly between 1 and 10 percent of the total time interval, i.e. between 1000 and 10000 time points later than the arrival time. The data generated using the first technique is used to evaluate the performance of $LookOut$ when the time interval varies widely; the second data generation

Table 3.3: Delays in processing inserts and deletes for LookOut, varying cardinality, with 2 dim. data. Delays in ms.

Card-inality in K	**Max Anti-Corr. Delay**	**Max Indep. Delay**	**Avg Anti-Corr Delay**	**Avg Indep. Delay**
10	1.21	1.55	0.0291	0.0273
20	1.42	1.72	0.0250	0.0235
30	3.31	2.99	0.0236	0.0223
50	3.41	4.57	0.0221	0.0214

technique is used to evaluate the effect on performance when the time-intervals have a constrained size.

The results that we present are generated by running each data set from time 0 to 100,001. During this time, each data point will arrive and be deleted following its expiration. The skyline is continuously updated over the course of the 0 to 100,000 time interval. We present results indicating the throughput in *elements per second (eps)* that can be achieved by $LookOut$. Note that this metric reflects the time to insert or delete an element as it arrives or expires, plus the time to update the continuous skyline. For each experiment, we consider the implementation of $CBBS$ and $LookOut$ using the R*-tree and the quadtree. For $CBBS$ we use the label $CBBS$-R and $CBBS$-Q for the R*-tree and the quadtree index implementations respectively. Similarly $LookOut$ - R and $LookOut$ - Q refer to the R*-tree and quadtree implementation of $LookOut$, respectively. The y axis in all figures uses a log scale to show the workload execution time.

Cardinality

In this test, we vary the data cardinality from 10K to 50K. The results of this experiment using data generated with the first technique (expiration times randomly chosen between the arrival time and 100K) are shown in Figure 3.11, for the three data distributions. In these figures, we observe that the execution time for $LookOut$ with a quadtree relative to $CBBS$ is more than two orders of magnitude better. $LookOut$ with the R*-tree is between 2 and 6 times faster than $CBBS$ in the anticorrelated case and almost twice as fast in the independent case for data set cardinalities greater than 20K. In the correlated case, $LookOut$ with the R*-tree achieves only a small improvement over $CBBS$ with an R*-tree. The superior performance of $LookOut$ with respect to $CBBS$ irrespective of the underlying data structure is expected due to the efficiencies of $LookOut$ in updating the skyline with each new insertion or deletion instead of recomputing it from scratch as $CBBS$ does. There is a more marked improvement in $LookOut$ relative to the $CBBS$ algorithm with the quadtree than with the R*-tree because of the improvements of the quadtree over the R*-tree for skyline evaluation already mentioned and because the insertions and deletions with the quadtree are very fast. The overhead of the R*-tree limits the amount of performance improvement that $LookOut$ can achieve.

The results of the experiment using data generated with the second technique (expiration times randomly chosen between 1 and 10 percent of the total time interval) are shown in Figure 3.12, for the three data distributions. The trends in the data are similar, with $LookOut$ with the quadtree again outperforming $LookOut$ with the R*-tree and $CBBS$, regardless of indexing structure by at least an order of magnitude. $LookOut - R$ also outperforms $CBBS - R$ by a factor of 2-3 in the anti-correlated case.

Table 3.3 presents data on the maximum and average processing delays for $LookOut$ for this experiment. These results indicate that $LookOut$ can process about 45,248 eps for the anti-correlated data set, and about 46,728 eps for the independent data set. (Note 1000/0.0221 = 45,248.)

Dimensionality

The results for the data set dimensionality tests using data generated with the first technique (expiration times randomly chosen between the arrival and 100K) are presented in

Figure 3.13. We observe that the execution time for $LookOut$ with an R*-tree relative to $CBBS$ with an R*-tree is about twice as fast for the independent case for each dimensionality, and between 2 and 9 times better for the anti-correlated case, depending on the dimensionality. The best algorithm is again $LookOut$ with the quadtree, as it only incrementally recomputes the skyline on inserts and deletes, and uses the faster inserts and deletes methods of the quadtree. It is an order of magnitude better than $CBBS$ with an underlying quadtree for all data distributions for all dimensionalities and is more than 2 orders of magnitude better in high dimensionality for the anticorrelated case. The results of the experiment using data generated with the second technique (expiration times randomly chosen between 1 and 10 percent of the total time interval) are shown in Figure 3.14.

From these figures we observe that the execution time for $LookOut$ is less than $CBBS$ with each respective data structure. With the quadtree, $LookOut$ is more than an order of magnitude faster than $CBBS$ with the quadtree. With the R*-tree, $LookOut$ is faster on average by a factor of 2 to 3. In the anti-correlated case, the rate of increase for the $CBBS$ algorithm is higher as d increases than it is for ***LookOut***, indicating that ***LookOut*** scales better for increasing d.

In Table 3.4, we present the maximum and average processing delays for the $LookOut$ algorithm. These results indicate that $LookOut$ can support a throughput rate of about 36,630 eps and 34,364 eps for the independent and anti-correlated cases, respectively, at a dimensionality of 2. For dimensionality 5, the throughput rates are about 3,849 eps and 2,950 eps for the independent and anti-correlated cases.

3.5.6 Comparison with the eager and lazy techniques

In this section, we compare the performance of $LookOut$ with that of the $eager$ and $lazy$ techniques of [75]. The code for these techniques was obtained from the first author's website and compiled for Linux. Following the approach of [75], we experiment with

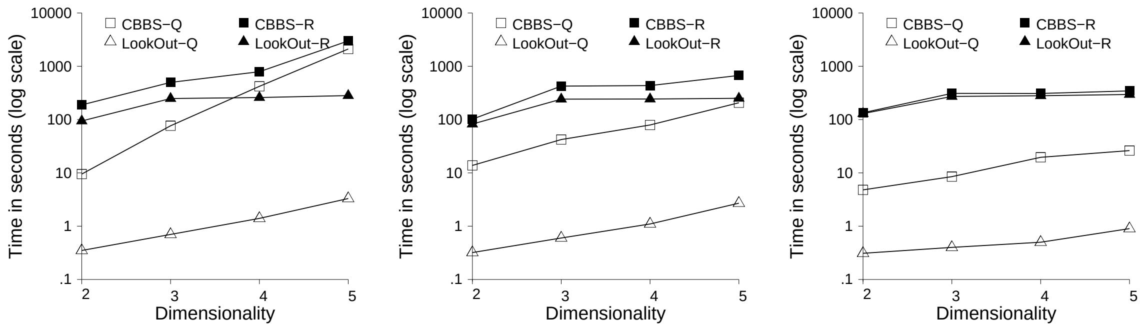


Figure 3.13: Experimental results for the time-interval continuous skyline with random time interval length with fixed cardinality (10K tuples) and varying dimensionality. Graphs show (a) the anti-correlated, (b) the independent, and (c) the correlated cases.

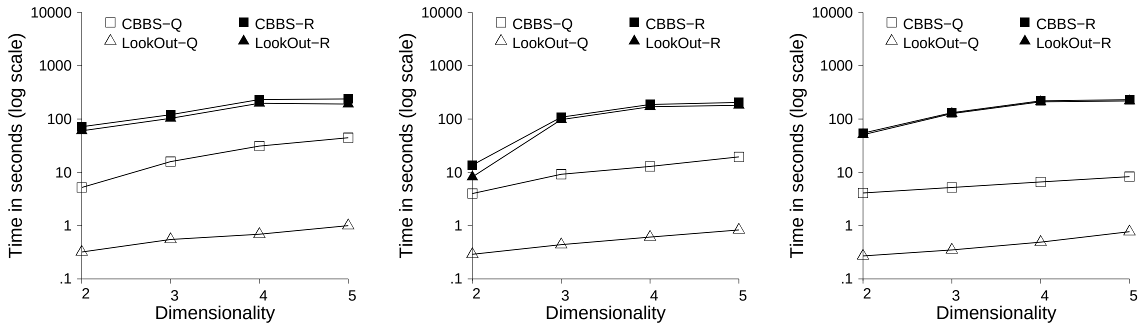


Figure 3.14: Experimental results for the time-interval continuous skyline with 1-10 % time interval length with fixed cardinality (10K tuples) and varying cardinality. Graphs show (a) the anti-correlated, (b) the independent, and (c) the correlated cases.

dimensions 2, 3, and 4, windows of size 200, 400, 800, and 1600 tuples, and report back the per tuple processing times in milliseconds. The independent and anti-correlated data sets were generated with the data set generator also provided from the first author's website. Each data set was modified for $LookOut$ by assigning each tuple an expiration time equal to the arrival time plus a number of time units equal to the size of the window. This means that both $LookOut$ and $lazy$ and $eager$ have the exact same data points available for inclusion in the skyline at any one time and produce the same skyline results. The results for varying dimensionality with a fixed 800 tuple window size are presented in Figure 3.15a for independent data and in Figure 3.15b for anti-correlated data. The results for 3 dimensionals with a varying tuple window size are presented in Figure 3.16a for

Table 3.4: Delays in processing inserts and deletes for LookOut, for varying dim., 10K cardinality. Delays in ms.

Dimen-sion-ality	Max Anti-Corr. Delay	Max Indep. Delay	Avg Anti-Corr. Delay	Avg Indep. Delay
2	1.21	1.55	0.0291	0.0273
3	1.54	5.65	0.0613	0.0499
4	3.26	9.40	0.1297	0.0959
5	4.45	14.80	0.3390	0.2598

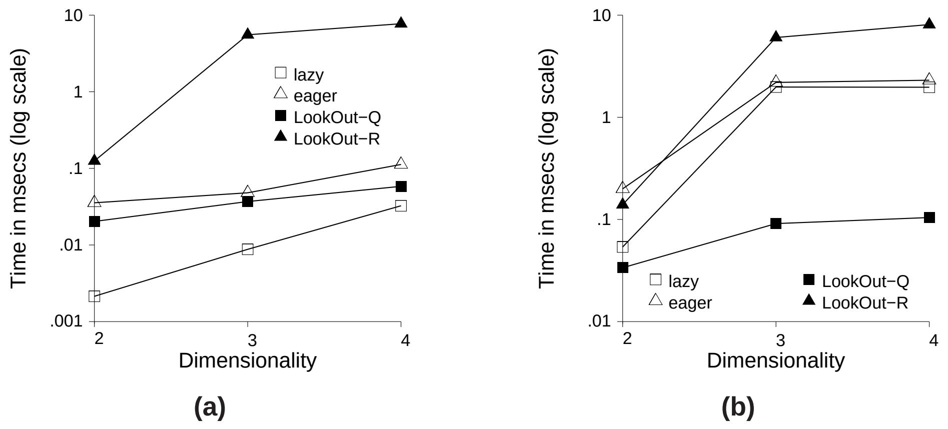


Figure 3.15: The per tuple processing costs for varying dimensionality and a fixed window of size 800 for (a) independent and (b) anti-correlated data.

independent data and in Figure 3.16b for anti-correlated data.

In Figures 3.15a and 3.15b, $LookOut$ using the quadtree performs better than the $eager$ technique for the independent data set and about an order of magnitude better than either $eager$ or $lazy$ for the anti-correlated data set for dimensionality 3 and 4, while handling the more general expiration time model. It achieves similar results for all window sizes in Figures 3.16a and 3.16b. The performance advantage is largely due to the better update performance of the quadtree in these experiments, since $LookOut$ with the R*-tree was much slower, particularly for the independent data set. $LookOut$ does not perform as well as the $lazy$ technique for the independent dataset. This is because the size of the skyline is much smaller than in the anti-correlated case, so the benefits of using the quadtree is much reduced. The important observation from these experiments is that the performance

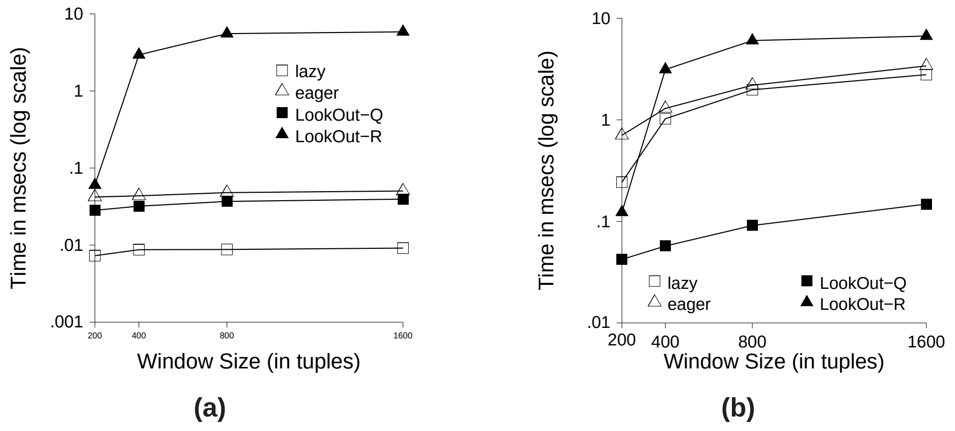


Figure 3.16: The per tuple processing costs for 3 dimensional data and a varying window size for (a) independent and (b) anti-correlated data.

of $LookOut$ is better than $eager$ in all cases and better than $lazy$ in the anti-correlated case even when handling the more restricted time model.

3.6 Conclusions

In this chapter, we have introduced the continuous time-interval skyline operation. This operation continuously evaluates a skyline over multidimensional data in which each element is valid for a particular time range. We have also presented $LookOut$, an algorithm for efficiently evaluating continuous time-interval skyline queries. Detailed experimental evaluation shows that this new algorithm is usually more than an order of magnitude faster than existing methods for continuous skyline evaluation.

We have also exposed several inherent problems with using the R*-tree index for evaluating a skyline. The primary reason for the inefficiency of the R*-tree for skyline computation is the overlap of the bounding box keys, which results in poor subtree pruning of the index non-leaf and leaf nodes that are examined during the skyline computation. We have shown that the quadtree index is a much more efficient index structure for evaluating skylines. The non-overlapping partitioning characteristics of the quadtree leads to a natural decomposition of space that can more effectively prune the index nodes that must be searched. An extensive experimental evaluation shows that using a quadtree can result in a continuous skyline evaluation method that can achieve high throughput and can also dramatically speed up traditional static skyline computation.

In the next chapter, we will develop algorithms to find the skyline for datasets in the presence of low-cardinality attribute domains that is far more efficient and effective than the more general techniques discussed in this chapter.

CHAPTER IV

Computing Skylines Using Lattices

4.1 Introduction

In the thesis introduction, we observed that the skyline operator has emerged as an important summarization technique for multi-dimensional datasets. Recall that 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 other dimensions, for a defined comparison function.

In the previous chapter, we developed methods for evaluating the skyline in the presence of temporal data using the $LookOut$ algorithm. While the temporal dimensions of data in this context are assumed to follow the time interval continuous model, $LookOut$ makes no assumptions about the dataset attributes that are not temporal. In this chapter, we introduce the Lattice Skyline algorithm, that can evaluate static skylines more efficiently than other techniques if all of the dataset attributes are drawn from low-cardinality domains.

An example of the skyline operator in a hotel room selection application is shown in Table 4.1. 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,

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 4.1: A sample hotels dataset.

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.

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 4.1). 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 chapter of the thesis 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 [10] and Sort-First Skyline [19]. These algorithms eliminate data elements from consideration in the skyline by finding other elements in the dataset 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 [16, 52] 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 chapter 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 [19] with the LESS optimizations [24], 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 chapter 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 and discuss extensions of LS for evaluating temporal skylines. 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.

4.2 Terminology and Further Related Work

In this section, we discuss terminology used in this chapter of the thesis, and also further highlight work related to skyline computation over low-cardinality attribute domains.

4.2.1 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.

4.2.2 Related Work

The Sort-First Skyline algorithm is proposed in [19], 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 [24] 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 [74]: 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 Bitmap is applicable is more narrow. Bitmap also requires preprocessing, since Bitmap indices are required. 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 4.6.

Techniques to evaluate skylines in subspaces have been proposed in [84] and [62]. 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 [48], 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.

4.3 Skyline Computation for Low-Cardinality Attributes

Throughout this chapter, 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 and that larger values dominate smaller ones for low-cardinality and

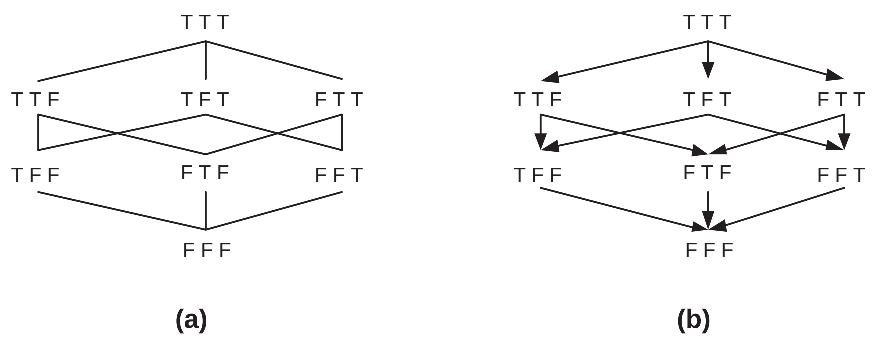


Figure 4.1: (a) A Boolean Lattice and (b) the Boolean Lattice with arrows to explicitly indicate the dominance relationship.

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.

4.3.1 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 4.1. 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 chapter.

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

Definition 4.3.1. *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 4.3.1 to show that the space of vectors B with the skyline operator is a lattice.

Theorem 4.3.2. *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

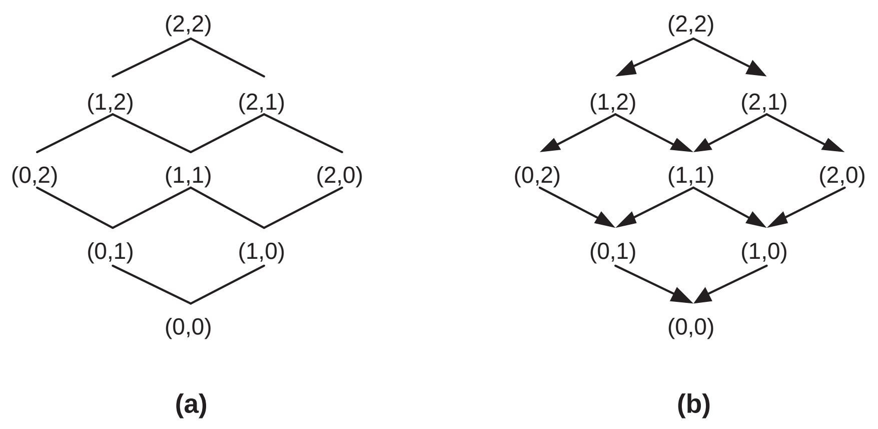


Figure 4.2: A two-dimensional lattice in which each attribute is drawn from the domain {0,1,2}.

boolean-valued is presented in Figure 4.1a. 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 4.1b. For example, TTF dominates TFF, FTF, and FFF. These are the values in the graph in Figure 4.1b 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 4.2a. In Figure 4.2b, 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 4.3.3. *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 4.2b 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.

4.3.2 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 4.1 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 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 4.1b 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

Algorithm 8 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 $j \in B$ to a position in $a$.
7: for each $s \in D$ 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 $g \in$ 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 $s \in D$ 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
```

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.

4.3.3 The LS-B Algorithm

The LS-B algorithm, shown in Algorithm 8, 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*.

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.

4.3.4 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 4.1. Specifically, the example data is displayed in Table 4.2.

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

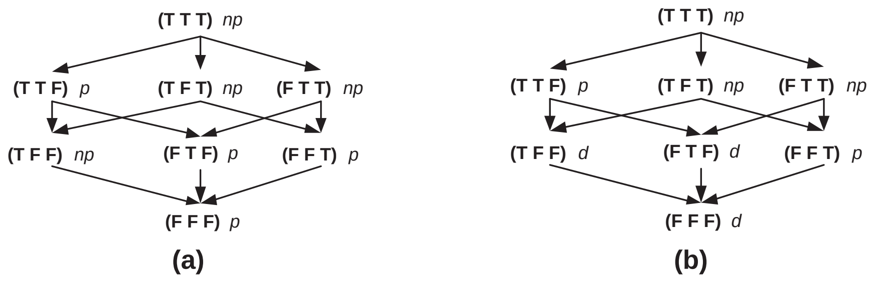


Figure 4.3: (a) The Boolean Lattice from the example, with present [*p*] and not present [*np*] elements marked. (b) The lattice with dominated values marked as dominated [*d*]. Skyline values are those marked [*p*].

value following these actions is displayed in Figure 4.3a.

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 4.3. The lattice following the skyline value evaluation, with each lattice element marked as *np=not present*, *p=present*, or *d=dominated* is shown in Figure 4.3b. 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.

4.3.5 Analysis

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

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 4.2: The hotels from the example dataset of Table 4.1 with the values of their three boolean-valued attributes (parking availability, swimming pool, and workout center).

Theorem 4.3.4. *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 4.2b has only 1 immediate dominator instead of 2. In short, we expect the algorithm to be efficient in practice, as we show in Section 4.6. 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

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 4.3: 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.

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.

$\square$

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 4.6 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!

4.4 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.)

4.4.1 Overview

The LS-B algorithm presented in Algorithm 8 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 4.4.1. *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 4.4.2. *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.

4.4.2 The Extended LS Algorithm

Algorithm 9 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 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.

Algorithm 9 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**
```

4.4.3 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 4.1. 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. 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 4.4. We refer to the $200 - price$ value as u.

The lattice consists of eight entries, one for each boolean value, and each is initialized

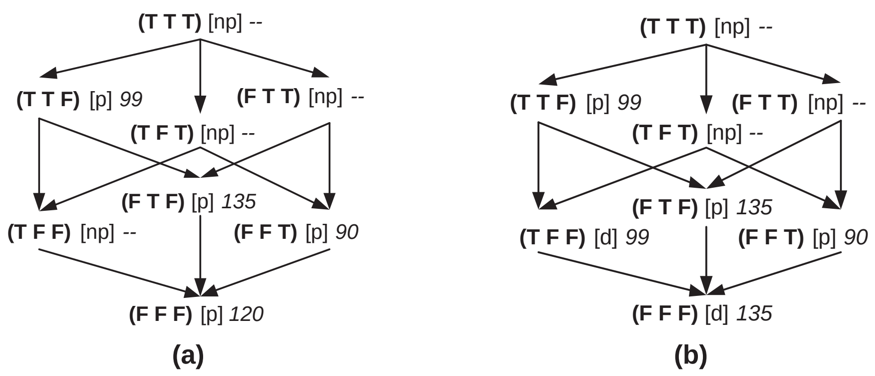


Figure 4.4: (a) The Boolean Lattice from the example, with [p] present and [np] not present elements marked with their locally optimal values; – means the lattice element is not updated. (b) The lattice with dlvs for each element and with dominated values marked [d]. Skyline values are those marked [p].

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 4.4a.

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

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 4.4: 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 4.5: 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.

each position, the immediate dominators are checked. The actions for each lattice position are shown in Table 4.5.

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.

4.4.4 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 4.3.

4.5 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 [10] 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 [19] 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.

4.6 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 [19] with the LESS optimizations described in [24]. 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 [24] 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 an index does not exist, which is expensive. To confirm this, we have also considered bulk loading an R-tree index on the fly using the R-tree bulk loading technique of [47] and then running BBS [59]. For the datasets that we use in this section, the index construction time itself 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.

4.6.1 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 [10]. 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.

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

Table 4.6: 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 4.7: Attributes in the Zillow house-price dataset.

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 used for these three parameters are shown in Table 4.6, with default parameter settings shown in bold. (The default value of $n = 500K$ is also used in [24]).

We also use two real datasets for our experiments. The first dataset is a house-price information dataset that is obtained from Zillow.com [4]. 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 4.7. In this dataset, the house price is an unconstrained attribute.

The second real dataset is taken from the Internet Movie Database (IMDB) [1], which

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 4.8: Attributes in the IMDB movie dataset.

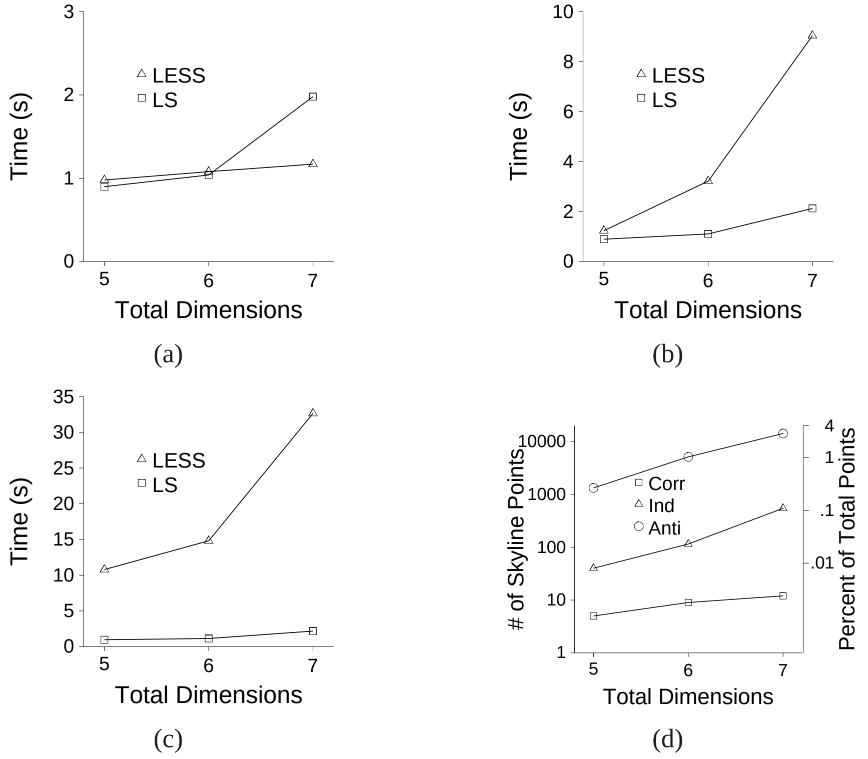


Figure 4.5: Results for 1 unrestricted and d-1 low-cardinality attributes with varying dimensionality for (a) the CO, (b) the IN, and (c) the AC distributions. (n=500K, c=8) The number of skyline points in each dataset is shown in (d).

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 4.8. 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.

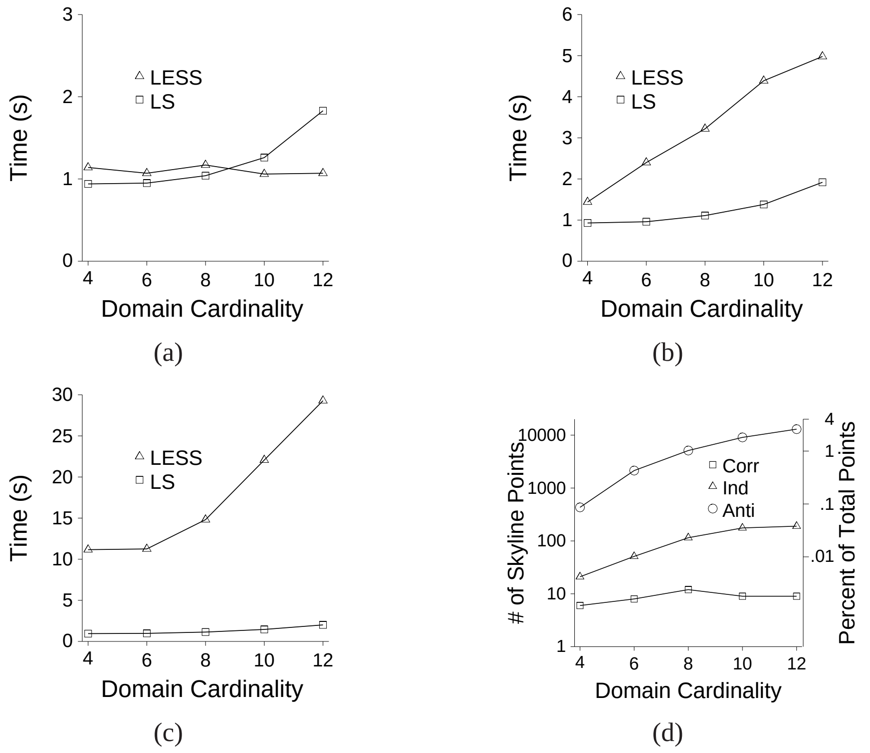


Figure 4.6: Results for 1 unrestricted and d-1 low-cardinality attributes with varying attribute cardinality for (a) the CO, (b) the IN, and (c) the AC distributions. (n=500K, d=6) The number of skyline points in each dataset is shown in (d).

4.6.2 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.

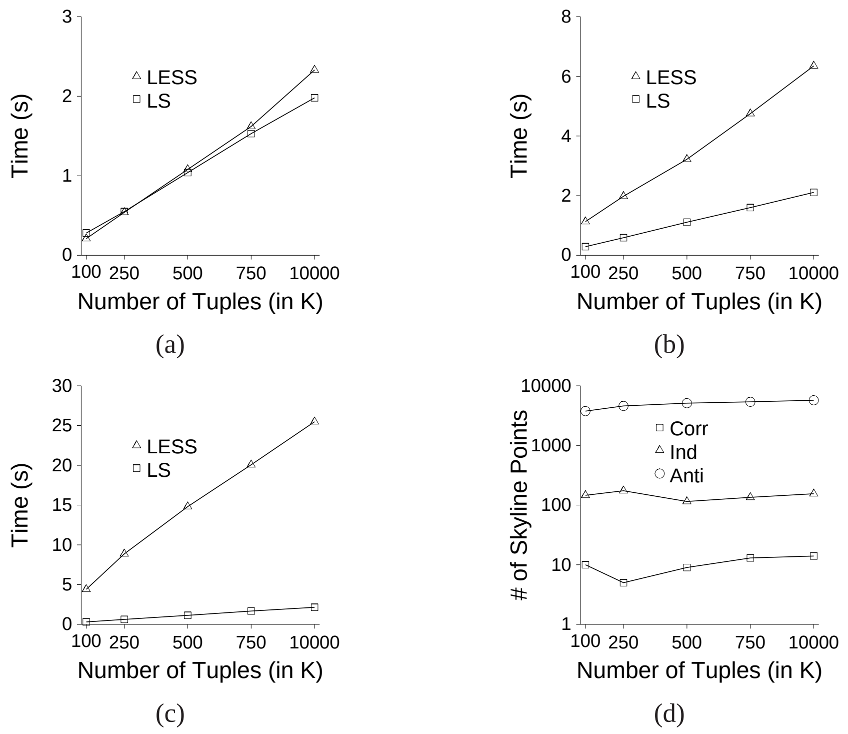


Figure 4.7: Results for 1 unrestricted and d-1 low-cardinality attributes with varying dataset cardinality for (a) the CO, (b) the IN, and (c) the AC distributions. (c=8, d=6) The number of skyline points in each dataset is shown in (d).

In [24], 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 [24].

4.6.3 Performance on Synthetic datasets

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 [24]). The results are shown in Figures 4.5a, 4.5b, and 4.5c for the correlated, independent, and anti-correlated datasets,

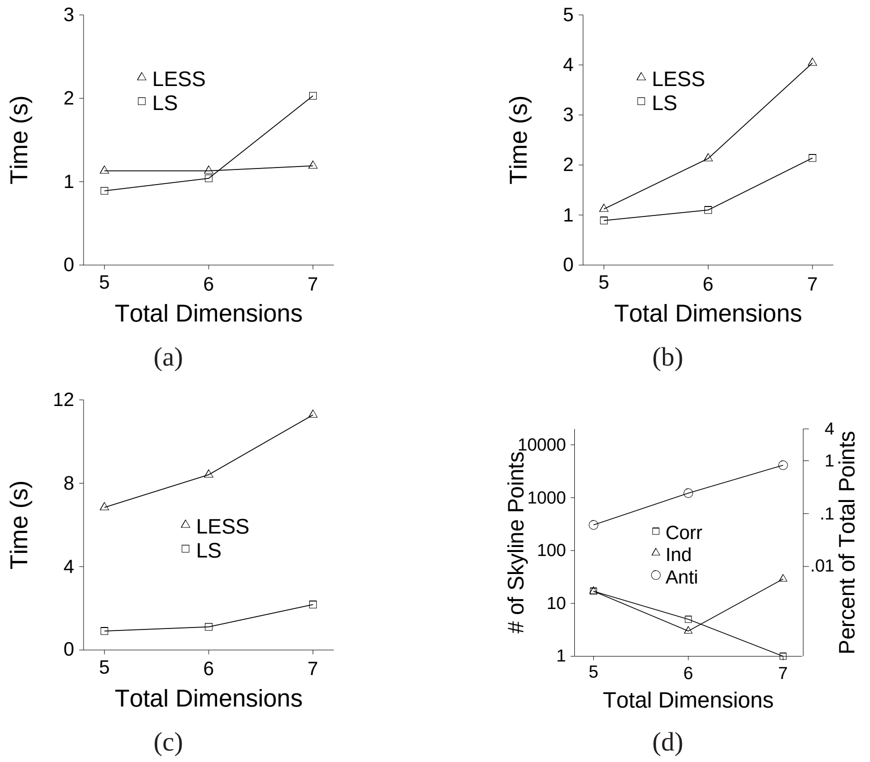


Figure 4.8: Results for d low-cardinality attributes with varying dimensionality for (a) the CO, (b) the IN, and (c) the AC distributions. (n=500K, c=8) The number of skyline points in each dataset is shown in (d).

respectively. Figure 4.5d, shows the number of skyline points for each distribution.

From Figure 4.5c 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 4.5 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 4.5d. When the number of skyline points is small (near 10), as in the CO case, LESS performs as well or

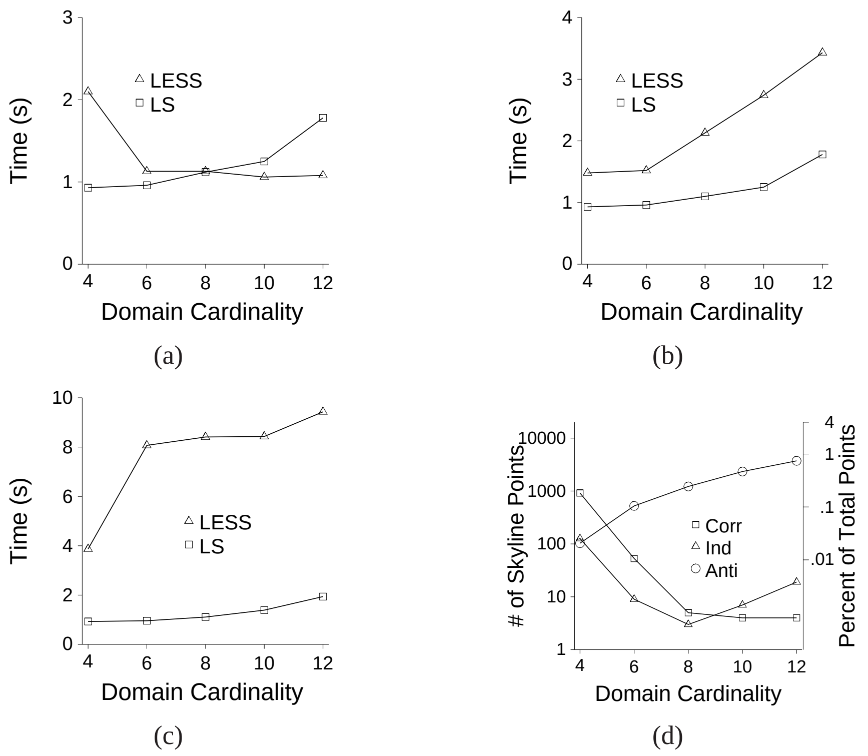


Figure 4.9: Results for d low-cardinality attributes with varying attribute cardinality for (a) the CO, (b) the IN, and (c) the AC distributions. (n=500K, d=6) The number of skyline points in each dataset is shown in (d).

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 4.5d 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.

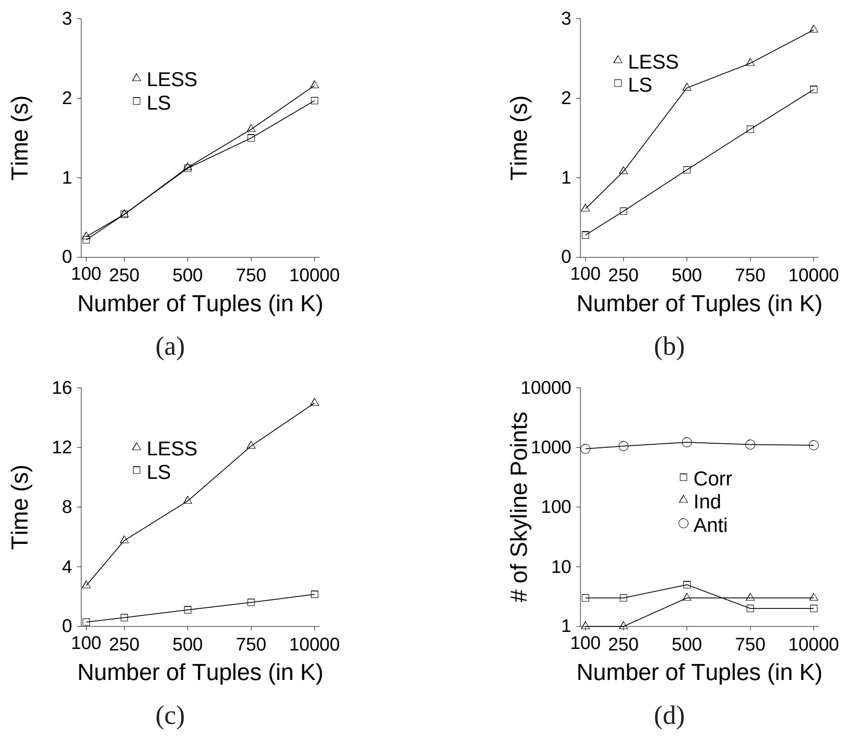


Figure 4.10: Results for d low-cardinality attributes with varying dataset cardinality for (a) the CO, (b) the IN, and (c) the AC distributions. (c=8, d=6) The number of skyline points in each dataset is shown in (d).

In the second test, we vary the attribute cardinality between 4 and 12. The results are presented in Figures 4.6 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 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 4.6d. 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 4.7 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 4.7d).

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 9 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 4.8d, 4.9d, and 4.10d 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 [24]). The results are shown in Figures 4.8 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 4.6.3 because the number of skyline points is smaller. As is described in Section 4.6.1, 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 4.8d.

In the next experiment, we vary the low-dimensionality cardinalities. The results for this experiment are shown in Figures 4.9 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 4.6.1 for details). Essentially, the performance of LESS for the CO case closely follows the trend set by the skyline size, shown in Figure 4.9d.

In the third test, we vary the number of data points in each dataset between 100K and 1M. The results are shown in Figures 4.10 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.

4.6.4 Performance on Real Datasets

First, we evaluate the performance of LS and LESS on the Zillow dataset. The Zillow dataset contains 8 attributes (see Table 4.7), and our queries compute the skyline with respect to the max operator for the first seven 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 4.11a. 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 4.11b. 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. 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

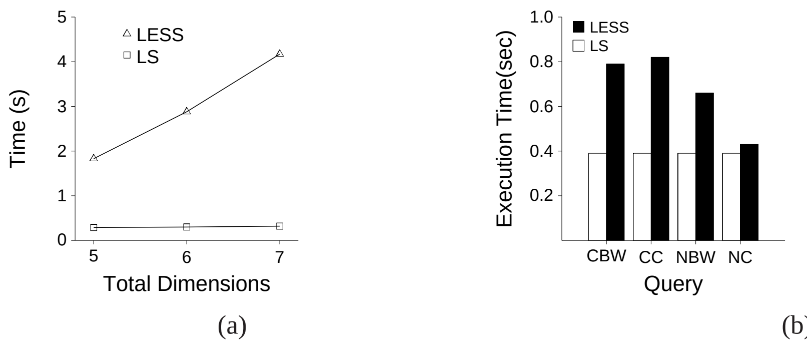


Figure 4.11: Performance of LS and LESS on (a) the Zillow house-price information dataset, and (b) the IMDB Movie Ratings Dataset.

domains in this example each had cardinalities of approximately 100. Even for this large c value, LS outperforms LESS.

4.6.5 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.

4.7 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 4.1). 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	⋆⋆	65
Celestial Sleep	T	T	F	⋆⋆⋆	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 4.7.1. *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). 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.

4.8 Conclusions

In this chapter, 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.

In the next chapter, we discuss the Skyline Point Order, a new way to rank skyline points. By identifying those points that are most valueable to the skyline summary, the technique increases the utility of the skyline operator and makes using the skyline easier in cases when the number of skyline points in a dataset is very large.

CHAPTER V

Measuring Skyline Point Utility

5.1 Introduction

In the previous two chapters, we described skyline evaluation in the presence of temporal data using the $LookOut$ algorithm and over low-cardinality attribute domains using the Lattice Skyline algorithm. The utility of the skyline produced in either of these contexts and in the most general, static context as a meaningful data summarization technique is heavily influenced by the number of data elements in the skyline. The cardinality of the skyline of a dataset is often very large, particularly if the dimensionality of the data is large or the data set elements are anti-correlated. Even when the dataset attributes are independent of one another, the number of elements in the skyline has been shown to be $\Theta(log^{d-1}\, n)$ in expectation in [7]. In such cases, the number of data points in the skyline is on the order of the number of data points in the dataset itself and, in such cases, the value of the skyline as a summary technique is lost.

The potential for skylines with large cardinalities has been noted before [43, 60, 84]. Some techniques to try to reduce the number of skyline points have previously been proposed. For example, some methods consider the skyline in a subset of the dimensionality such as the skyline frequency [14] or in the k-dominant skyline [15] setting. However, these techniques do not guarantee a reduction of skyline points to any particular number,

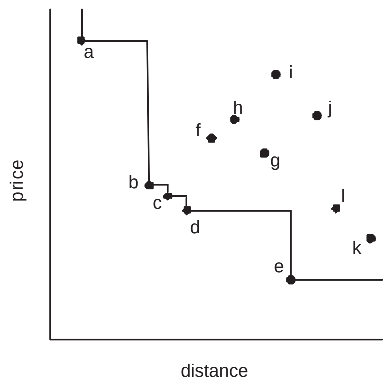


Figure 5.1: An example skyline with five skyline points.

nor does it address the problem in the general case, i.e. when the user is interested in points in all d dimensions. Techniques such as the Approximately Dominating Representatives [41] and the k most representative skyline operator [50] do reduce the skyline to a fixed size, but these lack a monotonic property, which we show is desirable, and they do not rank the elements of the skyline in any particular order, which is the primary focus of this chapter.

Ordering points places the most significant points at the beginning of the skyline result set. Intuitively speaking, not all points in the skyline are equally useful as summary points. For instance, consider the common hotel price versus distance example shown in Figure 5.1. In this example, points b, c, and d in all dominate approximately the same set of points and are all very close to one another. Someone choosing hotel c is likely also to be satisfied with hotel b or d. Hence, someone seeking a condensed skyline summary is likely to satisfied with one such point. Hotels a and e both possess values which are different from any other skyline point. However, a does not dominate any other hotels while e dominates hotels j, k, and l. Thus, e seems like a better summary point than a.

In this chapter, we quantify these intuitive ideas to develop a measure of the importance of each point to the skyline of a dataset. We develop two summarization properites, the dataset summarization and the spatial summarization, for skyline points in datasets and develop a method, called the Skyline Point Ordering (SPO), that quantifies these properties to rank skyline points. Using this method, the skyline points are ranked in order of summarization importance to the skyline so that the most important skyline points are returned to a user first for consideration. A ranked skyline can naturally be extended to a top-k skyline result set by selecting only the first k elements of the ranked skyline. To assess the accuracy of different top-k skyline result sets, we also introduce the Hypperarea Difference and Pareto spread metrics developed in the engineering community to assess the optimality of Pareto sets as methods to measure the summarization accuracy of different top-k skyline sets.

We further propose two different algorithms to evaluate the Skyline Point Ordering. The first, called the Coverage First Algorithm, evaluates the SPO using only the multi-dimensional dataset as input, while the second, called the Skyline First Algorithm, uses both the dataset and the set of skyline points as input.

The rest of this chapter is organized as follows: Section 2 discusses related work. Section 3 describes the HyperArea Distance and Pareto spread measures for top-k skyline accuracy. Section 4 introduces the Skyline Point Ordering and section 5 describes the Coverage First and Skyline First Algorithms. Section 6 contains our experiments and Section 7 concludes.

5.2 Related Work

Methods to reduce the number of skyline points have been proposed. Reductions for high dimensional skylines include the skyline frequency [14], strong skyline points [85], and the k-dominant skyline [13]. These methods do not rank skyline points nor do they guarantee a reduction to any specific number. The skylines in these cases can still be very large.

Several top-k skyline techniques exist, including Approximately Dominating Representatives [41] (ADR) and the k most representative skyline operator [50]. The ADR of a dataset is obtained by postprocessing the skyline by boosting a skyline point by a factor ϵ in each dimension and removing skyline points dominated by the new point until only k skyline points remain in the new set. The problem is shown to be NP-hard and approximation algorithms are developed. The k most representative skyline operator selects the k points in the skyline set that collectively dominate the largest number of other points in the dataset. This problem is also shown in [50] to be NP-hard. We compare our techniques to the FMG technique developed for approximating the k most representative skyline operator. These techniques do not rank skyline points, which is the focus of our work, but top-k skyline methods are similar to our Skyline Point Ordering because the first k ranked skyline points could be treated as the top-k points.

5.3 Motivation and Summary Quality Measures

One of the drawbacks of using the skyline as a summary mechanism for a dataset is the shear volume of data that the skyline may contain. A large volume of data decreases the usefulness of the skyline as a summary. Summarizing the skyline with some smaller number of skyline points is advantageous if the number of skyline points is very large.

Once we decide to summarize the skyline, we must find a good way to obtain summary points for the points in the Pareto set. A number of methods to evaluate the effective summary measure for Pareto sets have been developed in the Engineering community. We will summarize the Hyperarea Difference measure here that is appropriate for measuring the summary accuracy of a top-k skyline (see [80] for more details).

The Hyperarea Difference (HD) metric is a quantitative evaluation of the difference between the size of the dominated spaces of the true (complete) Pareto set P_c and an observed (summary) Pareto set P_s. If we let $HD(P_c,\ P_s)$ denote the hyperarea difference quantity, then:

$$HD(P_c, P_s) \quad = \quad space(P_c) - space(P_s)$$

Here, the term space refers to the area covered by the dominance set of the skyline in 2 dimensions and the volume in 3 or more dimensions.

The hyperarea difference can be quantified as the space difference between the complete skyline, which may contain too many points to be useful as a summary but which still captures the unique optimal values with respect to the dominance relationship, and a potential candidate skyline summary. In [80], the authors note that the better the space of an observed (summary) Pareto set approximates the space of the true (complete) Pareto set, the better the observed (summary) Pareto set approximates the true (complete) Pareto set.

The hyperarea difference measure is one way of accessing the effectiveness of a skyline summary. In the next section, we develop methods to effectively summarize skyline points that try to obtain good spatial summarization as well as dataset summarization.

5.4 Skyline Point Ranking

In this section, we discuss skyline point qualities that should be measure when ranking skyline points. We then discuss quantitative measures for these qualities. Finally, we develop an overall measure of the importance of a skyline point to the overall skyline.

5.4.1 Qualities for Skyline Measure

Any measure that is to rank the relative importance of skyline points to the overall skyline must consider the following two properties of skyline points.

1. **Dataset Summarization:** Since the skyline S is a summary of the larger dataset D, elements $s \in S$ that dominate other points in D add summary value to the skyline. If a point $s \in S$ dominates $p \in D$, then the value of s is preferred to that of p. The more points that s dominates, the better its summary value.

2. **Spatial Summarization:** The points in the skyline are also values that are not dominated in the partial order imposed by the skyline. Each such skyline element occupies a point in space whose importance to the skyline can be measured by how unique a value it is relative to other points in the skyline. A skyline point p that is very near to another skyline point q is not adding much additional summary value; a user interested in the approximate location of p on the skyline can substitute q at little cost.

To see why these two properties of skyline points are important for ranking, consider as an example the dataset shown in Figure 5.1. Data point a does not dominate any other point, i.e. there is no other data point p in D that would prefer a to p. Therefore, the utility of this point for dataset summarization purposes is very low. However, the value of a is very unique since no other point is near it. Hence, the utility of this point for spatial summarization is very high.

These measures are relativistic, meaning that the summary and uniqueness importance of a point p varies depending on the dataset (on the nonskyline points that are dominated by p and on the nearness of other skyline points to p).

5.4.2 Dataset Summarization Measure

In this subsection, we propose a novel measure called the Point Dominance Set of a skyline point as a measure of its dataset summarization properties with respect to the dataset.

Definition 5.4.1. *Point Dominance Set (PDS): The point dominance set of a skyline point*

s of a dataset D is the set of all $p \in D$ such that s dominates p.

$$PDS(s, D) \quad = \quad \{p \in D s.t. s \prec p\} \cup s$$

The point dominance set for each skyline point measures its summarization for elements of D. For example, in Figure 5.1, the point dominance set of point b is $\{b, g, h, i, j\}$. The PDS of e is $\{e, l, k\}$.

We require a numeric measure of the PDS in order to rank skyline elements relative to one another. We can obtain such a measure by normalizing the cardinality of the PDS by the size of the dataset. We call this measure the Normalized Point Dominance of a skyline point.

Definition 5.4.2. *Normalized Point Dominance (NPD): The normalized point dominance of a skyline point s of a dataset D is the cardinality of the PDS of s divided by the total cardinality of D.*

$$NPD(s, D) \quad = \quad \frac{|PDS(s, D)|}{|D|}$$

As an example, consider again Figure 5.1. In the figure, the value of the NPD for b is $|\{b, f, g, h, i, j\}|$/$|D|$=6/12=0.5.

5.4.3 Spatial Summarization Measure

In this subsection, we develop a technique called the Nearest Metadominant Distance as a measure of the spatial summarization properites of a skyline element.

Definition 5.4.3. *Nearest Metadominant Distance (NMD): The nearest metadominant distance of a skyline point s of a dataset D is the distance to the nearest skyline point in D that dominates more points than s dominates.*

$$\begin{aligned} NMD(s, D) &= min\, dist(s, p \in S)\ s.\,t. \\ & NPD(p) > NPD(s) \end{aligned}$$

If no point p has a larger NPD than a point s, then $NMD(s, D) = |U|$, where U is the universe.

The Nearest Metadominant Distance measures the distance from a skyline point s only to those skyline points in the dataset that dominate more points than s dominates. This prevents all skyline points in a cluster from being low-rated do to their spatial similarity. At least one of these points should be highly rated. For example, in Figure 5.1, the three skyline points b, c, and d are all close in space and dominate the same set of points $\{f, g, h, i, j\}$, with the exception of d which also dominates point l. For this reason, the NMD distances of b and c will be small (their distances to point d), while that of d will be the size of U.

If two points have the same NPD, we can adopt the tie-breaking convention to rate points with equal NPD scores based on the value in a particular dimension.

We now introduce the Normalized Nearest Metadominant Distance to normalize the interpoint distances by the size of the universe.

Definition 5.4.4. *Normalized Nearest Metadominant Distance (NNMD): The normalized nearest metadominant distance of a skyline point s of a dataset D is the nearest metadominant distance of s divided by the size of the universe U in which elements of D are drawn.*

$$NNMD(s, D) = \frac{|NMD(s, D)|}{|U|}$$

5.4.4 Complete Ordering

In this subsection, we define the Skyline Point Ordering for a set of skyline points as a combination of the Normalized Point Dominance and the Normalized Nearest Metadominant Distance, thus capturing a combination of the dataset and spatial summarization of each skyline point.

Definition 5.4.5. *Skyline Point Ordering:*

$$SPO(s, D) \;=\; NNMD(s, D) * NPD(s, D)$$

The skyline point which dominates the largest number of points in D will always be the highest ranked skyline point. This is because it will have the largest Point Dominance Set of any skyline point and its Nearest Metadominant Distance will be the size of the Universe.

5.4.5 Using the Skyline Point Ranking to find the Top K Skyline

As we have already discussed, ranking skyline points is important to present users with the most important skyline summary points first in the event that the number of skyline points. It can also be used as a convenient measure of the top k skyline points, where k is an integer between 1 and the cardinality of the skyline. We first define the top k skyline operator using the SPO developed in the previous section.

Definition 5.4.6. *Top-K Skyline: a skyline point s is said to be a top-k skyline point if there does not exist k or more skyline points that have a greater SPO than s.*

This definition takes the first k points in the ranking of the SPO for the top-k. This definition has one very important advantage over the covering definition used in [50] or the ADR used in [41] that are discussed in the related work section and that is its monotonicity.

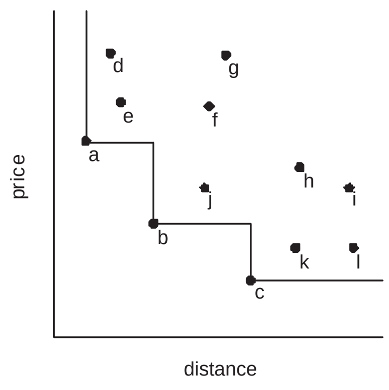


Figure 5.2: An example skyline with three skyline points.

5.4.6 Monotonicity Property

In this section, we discuss the monotonic property for top-k skyline queries. A top-k skyline definition is monotonic if the set of top-k skyline points is a superset of the top-$k-1$ skyline points. In other words, the top k skyline points should be equal to the top $k-1$ skyline points with one additional skyline point that is not in the top-$k-1$ skyline points. This can be written as T_k = $T_{k-1} + s$ where T_k is the top-k skyline points, T_{k-1} is the top-$k-1$ skyline points, and s is a skyline point not in T_{k-1}.

To understand why monotonicity is desireable for the top-k skyline, consider the top-k skyline definition used by the authors of []: The top-k skyline of a dataset D is the set of k skyline points which dominate the largest possible number of points in D.

For example, consider the dataset in Figure 5.2. In this figure, three data elements are in the skyline: a, b, and c. The dominance sets of each point are shown in Table 5.1.

Skyline Point	**Dominance Set**
a	d, e, f, g
b	f, g, h, i, j
c	h, i, k, l

Table 5.1: The skyline points from Figure 5.2 with their dominance sets.

The top-k points using the set coverage definition if $k = 1$ is b, since b has the largest coverage set. If $k = 2$, the top-k skyline is a and c. This seems unintuitive to a user – the best point is b, but the two best points do not contain the best point.

5.5 Top K Skyline Algorithms

In this section, we propose two different algorithms for evaluating the SPO. The first of these algorithms, called the Coverage First Algorithm, evalautes the skyline of the dataset and the Point Dominance Set of each skyline point at the same time. The second of these algorithms, called the Skyline First Algorithm, evaluates the skyline of a dataset using any known skyline algorithm, then uses the skyline to find the Point Dominance Set.

5.5.1 Coverage First Algorithm

The first algorithm we propose is called the *Coverage First Algorithm* (CF) which simultaneously evaluates the cardinality of the PDS of a data point and determines whether or not that point is a skyline point.

The algorithm maintains two sets of data points. The first set consists of those data points that have not yet been dominated by any other data points. This set is called the $Skyline\ Candidate$ (SC) set. The second set consists of those data points that have been dominated by some other data point and hence cannot be skyline points. This set is referred to as the $Dominated\ Points$ (DP) set. These sets are mutually exclusive ($SC \bigcup DP$ = D and $SC \bigcap DP$ = $emptyset$).

The CF algorithm is presented in Algorithm 10. The algorithm begins with all elements of the dataset D in SC and no data points in DP (lines 3-5 of the algorithm). Each element of SC is compared with every other element of SC to determine dominance (lines 6-18). If a point in SC is dominated, it is added to the set of dominated points (lines 15 and 24 of Algorithm 10). If a point is not dominated and is hence a skyline point, it is compared with the set of dominated points (lines 19-23) to determine the size of the PDS for the point. The temporary set RP is used to hold dominated points removed from SC which are later added to DP to prevent them from being considered twice (lines 24-25).

Once the skyline has been found and the PDS cardinality of each point is known, the skyline set is sorted on the PDS cardinalities (line 27). Now, the NMD of each skyline point is found in lines 28-36. First, the NMD is initialized to the size of the universe, where the size of the universe is the L-norm distance from the lower-left (least) corner to the upper-right (greatest) corner in the universe. This is the maximum value of the NMD for any point. Next, each skyline point s is compared with all other points in the skyline (lines 30-34). If a skyline point q has a larger point dominance set than another skyline point p and the euclidean distance from p to q is less than the nearest metadominant distance of p, the nmd of p is set equal to the distance between p and q.

Once a point p has been compared with all other skyline points, its spo can be found (line 35). The skyline points are then sorted on the spo (line 37), which completes the ranking.

5.5.2 Skyline First Algorithm

The second algorithm we propose is called the *Skyline First Algorithm* (SF) which first evaluates the skyline of the dataset before evaluating the cardinality of the coverage set of each skyline point. The Skyline First algorithm is shown in Algorithm 11.

The algorithm begins by first evaluating the skyline using any previously proposed skyline evaluation algorithm. In our implementation, we use the SFS technique of [19] with the LESS [24] optimizations. The set of skyline points S and the remaining points in the dataset D are inputs to SF (line 1). In lines 4-10, the size of the pds set for each skyline point is determined by comparing each element of the skyline set with each element of D.

The remainder of the SF algorithm (lines 11-21) is similar to the CF algorithm (lines 27-37) in how it evaluates the spo of each skyline point.

Algorithm 10 Coverage First Algorithm.

```
1:  **Input:** $D$
2:  **Output:** Ranked Skyline Set $SC$.
3:  $SC$ = $D$.
4:  $DP$ = $\emptyset$.
5:  $RP$ = $\emptyset$.
6:  for all $i \in SC$ do
7:      for all $j \in SC$ and $j \neq i$ do
8:          if $i$ dominates $j$ then
9:              $SC$ = $SC$ - $j$.
10:             $RP$ = $RP$ + $j$.
11:             $i.pds$++.
12:         end if
13:         if $j$ dominates $i$ then
14:             $SC$ = $SC$ - $i$.
15:             $DP$ = $DP$ + $i$.
16:             break.
17:         end if
18:     end for
19:     for all $n \in DP$ do
20:         if $i$ dominates $n$ then
21:             $i.pds$++.
22:         end if
23:     end for
24:     $DP$ = $DP$ + $RP$.
25:     $RP$ = $\emptyset$.
26: end for
27: sort $SC$ by $pds$ size.
28: for all $m \in SC$ do
29:     $m.nmd$ = size of $U$.
30:     for all $n \in SC, m \neq n$ do
31:         if $n.pds < m.pds$ and $dist(m, n) < m.nmd$ then
32:             $m.nmd$ = $dist(m, n)$.
33:         end if
34:     end for
35:     $m.spo$ = $m.pds/\mid D \mid$ * $m.nmd/\mid U \mid$.
36: end for
37: sort $SC$ on $spo$.
```

5.5.3 Algorithm Comparison

We expect the SF algorithm to outperform the CF algorithm when the size of the skyline is small. This is because the CF algorithm performs a block-nested loop computation that is avoided by SF in which points have the cardinalities of their dominance sets determined. Since many of these points end up being dominated, some extra computation is performed during this step. The SF algorithm, in contrast, separates the two steps. The skyline can be found without the expensive block-nested loops calculation, and comparing only skyline points with the dominated set is more efficient than the CF algorithm.

Algorithm 11 Skyline First Algorithm.

```
1: **Input:** dataset $D$, skyline $S$
2: **Output:** Ranked Skyline Set $SC$.
3: $SC = \emptyset$.
4: for $i \in SC$ do
5:     for $j \in D$ do
6:         if $i$ dominates $j$ then
7:             $i.pds$++.
8:         end if
9:     end for
10: end for
11: sort $SC$ by $pds$ size.
12: for all $m \in SC$ do
13:     $m.nmd$ = size of $U$.
14:     for all $n \in SC$, $m \neq n$ do
15:         if $n.pds < m.pds$ and $dist(m, n) < m.nmd$ then
16:             $m.nmd = dist(m, n)$.
17:         end if
18:     end for
19:     $m.spo = m.pds / \mid D \mid * m.nmd / \mid U \mid$.
20: end for
21: sort $SC$ on $spo$.
```

We expect the CF algorithm to outperform the SF algorithm when the size of the skyline is large. This occurs when the size of the skyline approaches the size of the dataset. In this case, finding the skyline and the size of the dominance set in one combined step as is done by CF saves computation that is split up into two steps by the SF algorithm. This is because every point needs to be compared with every other point when (nearly) all points are in the skyline. The size of the skyline will approach the size of the dataset in general when the dataset is anti-correlated and the dimensionality of the dataset is high.

5.5.4 Complexity

The complexity of both algorithms is $O(n^2)$. For the interest of space, we omit a formal proof but give the intuition here. The worst case for both algorithms occurs if all dataset elements are in the skyline. For CF, finding the dominance set for each element (lines 6-26) is $O(n^2)$ since the step then involves comparing each element to all $n - 1$ other dataset elements. The computation of the spo (lines 28-36) also takes $O(n^2)$ time since each element of SC is compared with all other elements in SC. For SF, finding the skyline in the initial step is an $O(n^2)$ operation if all elements are in the skyline. Finding the spo (lines 12-20) is $O(n^2)$ also for the same reasons as for CF.

5.6 Experiments

In this section, we study the performance of the SF and CF algorithms for evaluating the SPO. We then measure the effectiveness of the SPO for summarization of skyline sets. In the performance study, we also compare the SF and CF algorithms to the performance of the FMG algorithm of [50], because the FMG algorithm is used to find the top-k skyline and is the most similar technique to our work. The compiled executables for the Linux operating system for FMG were graciously provided to us by the authors of [50]. The SF and CF algorithms have been implemented in C++. All experiments are performed on a PC running Debian Linux with kernel 2.6.0, a 1.70 GHz Intel Xeon CPU, 512 MB of memory, and a 40 GB Fujitsu SCSI hard drive.

The Coverage First and Skyline First algorithms are implemented in C/C++. Each uses a buffer pool to access data with a buffer pool size of 128 pages with a page size of 4KB, and reads and writes of data pages are modeled using reads and writes to the file system. The FMG binary files work for datasets varying in dimensionality between 2 and 5. The FMG technique requires that a spatial indexing structure first be bult. We do not count the time to build the indexing structure in the query results reported here; all indices for the FMG technique are prebuilt. The SF and CF techniques introduced in this chapter require no indexing.

The FMG technique returns the top-k skyline points, instead of a ranking of all skyline points as is done by our techniques. The query results reported here are for the FMG algorithm run for the top $k = 10$ skyline points. The algorithm does require more time for larger k values. The FMG algorithm also has a parameter which deals with how closely the algorithm approximates the NP hard k dominating representatives definition. This parameter is set to the fastest, least accuracte value (2) for the performance study and to the most accurate, slowest value (32) possible for the accuracy experiments so that the results in both cases are the best possible for FMG.

In this performance experimental study, we first describe the datasets used, including both real and synthetic, then discuss results for low-dimensionality synthetic datasets with dimensions between 2 and 5, which allows for direct comparison between the SF and CF techniques and the FMG method. We then perform higher dimensionality comparisons for SF and CF and finally show results on real datasets. We then conduct accuracy experiments on the top-k results which we later describe.

5.6.1 Datasets

We experiment with both real and synthetic datasets. For the performance study, we use synthetic datasets generated with the dataset generator of [10]. These datasets are correlated, independent, and anti-correlated, and are widely used to evaluate the effectiveness of skyline algorithms because they represent many different types of real datasets in which skyline algorithms are useful. In the first set of experiments, we generate datasets with dimensionalities varying between 2 and 5, because the compiled executables provided for the FM algorithm operate on data with between 2 and 5 dimensions.

In the second set of experiments, we evaluate the performance of SF and CF on higher dimensionality datasets. The compiled binaries for the FMG technique is not applicable for these dimensionalities. For these experiments, we generate datasets with dimensionalities between 6 and 8.

We also experiment with two the real datasets. The first real dataset is the NBA players dataset[1]. This dataset has been used previously to evaluate the effectiveness of skyline

[1]www.basketballreference.com

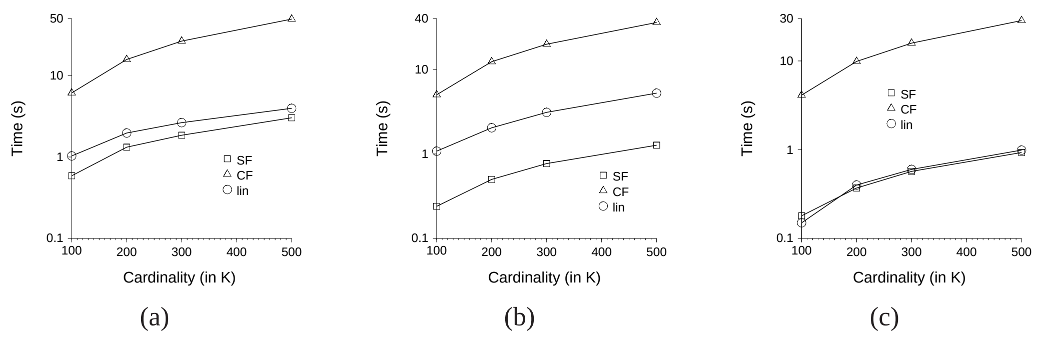


Figure 5.3: Performance results for varying dataset cardinality with fixed dimensionality of 2 for the (a) anti-correlated, (b) independent, and (c) correlated data distributions.

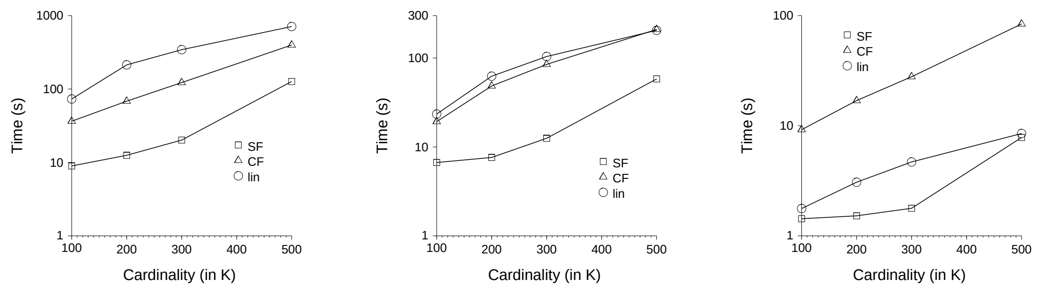
algorithms [13]. It consists of various statistics for NBA players for a season, such as the number of points, the number of rebounds, and the number of free throws that players obtain as single season totals. The dataset contains player-season row entries, so certain players have multiple entries in the dataset, depending on how many seasons they played in the NBA. For example, Michael Jordan has 15 entries in the dataset, corresponding to the 15 seasons he played in the NBA. This dataset contains more than 19 thousand entries.

The second real dataset we use is taken from the Internet Movie Database (IMDB). This dataset contains information about movies and television shows and ratings information from real users about each movie or television show and contains three attributes and more than 160 thousand entries. The three attributes for each tuple are the number of raters for each movie or TV show, the rating, and the year of production.

5.6.2 Low Dimensionality Performance

In this section, we present results for dataset dimensionalities varying between 2 and 5. These are the dimensions for which the FMG algorithm operates, and hence allows for a direct comparison between FMG and the CF and SF algorithms.

We show the results for 2 dimensions in Figure 5.3 a, b, and c for the anticorrelated, independent, and correlated cases, respectively. In all cases, SF performs better than the CF technqiue by nearly an order of magnitude. This is because the skyline is small for



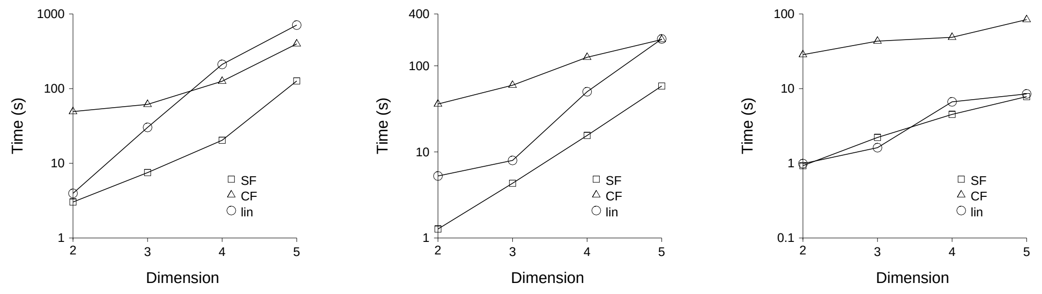
(a) (b) (c)

Figure 5.4: Performance results for varying dataset cardinality with fixed dimensionality of 5 for the (a) anti-correlated, (b) independent, and (c) correlated data distributions.

these two dimensional cases, and it can be found quickly using the SFS skyline algorithm, which improves the performance of SF over CF because CF loops iteratively through the complete dataset. SF also performs better than FMG for the anti-correlated (about 50% faster) and the independent (about 2X faster). This is because the performance of SF improves from the anti-correlated to the independent datasets because of the smaller number of skyline points in the independent case, while the performance of FMG does not improve much. In the correlated case, the performance is nearly identical.

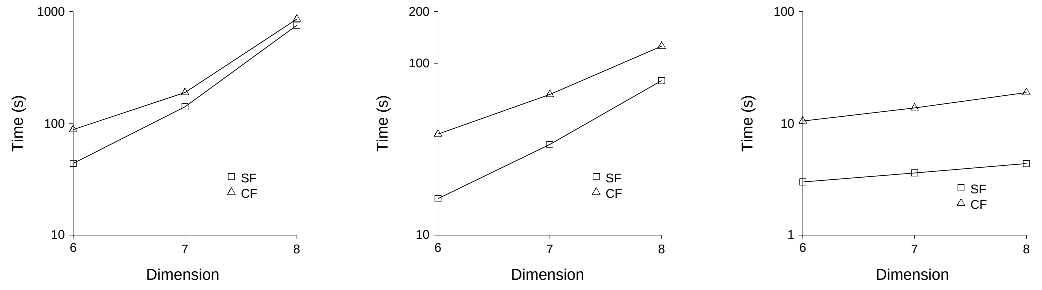
We present the results for each dataset for 5 dimensions in Figure 5.4 a, b, and c for the anticorrelated, independent, and correlated cases, respectively. As in the 2 dimensional case, SF is the best performing technique. SF performs better than CF by 3-10X. SF outperforms FMG by a factor of 4-10X in the anti-correlated and independent cases. Small efficiencies are also obtained in the correlated case. CF also outperforms FMG in the anti-correlated case by 2-4X and achieves small efficiencies in the independent case. FMG is faster than CF in the correlated case by nearly an order of magnitude.

The results for varying the dimensionality for fixed 500 tuple dataset cardinalities are shown in Figure 5.5 a, b, and c for the anti-correlated, independent, and correlated cases, respectively. SF is faster than CF in all cases by 5-10X. Sf is faster than FMG by 2-10X in the anti-correlated case, 2-4X in the independent case, and the two perform nearly



(a) (b) (c)

Figure 5.5: Performance results for fixed dataset cardinality of 500K with varying dimensionality for the (a) anti-correlated, (b) independent, and (c) correlated data distributions.



(a) (b) (c)

Figure 5.6: Performance results for fixed dataset cardinality of 100K with varying high dimensionality for the (a) anti-correlated, (b) independent, and (c) correlated data distributions.

identically in the correlated case. CF does outperform FMG for the 4 and 5 dimensional cases in the anti-correlated case and the two perform nearly identically in the independent case. In all other cases, FMG performs better than CF by 2-10X.

5.6.3 Higher Dimension Performance

In the previous section, we compared the performance of the CF and SF algorithms with that of the FMG technique. Because the FMG technique requires an index to be constructed first, the dimensionality in the previous section of the datasets tested was lower. We use smaller dataset cardinalities in these experiments than before because the larger dimensionalities produce higher running times than before. We use 100 K datasets when varying the dimensionality between 6 and 8 and vary the dataset cardinality between 20 and 100 K for the 8 dimensional dataset tests.

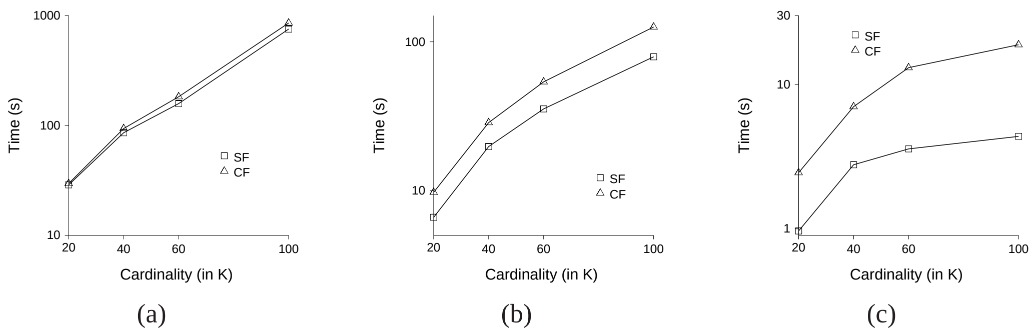


Figure 5.7: Performance results for varying dataset cardinality with fixed dimensionality of 8 for the (a) anti-correlated, (b) independent, and (c) correlated data distributions.

In this section, we evaluate the performance of the CF and SF algorithms on datasets having higher dimensionality. Specifically, we experiment with 6-8 dimensional synthetic datasets using the correlated, independent, and anti-correlated distributions discussed in the datasets section.

The results for varying the dataset dimensionality between 6 and 8 dimensions are presented in Figures 5.6 a, b, and c for the anti-correlated, independent, and correlated distributions, respectively. The SF technique still performs better than the CF dataset in these experiments as was the case for the 2-5 dimensional datsets. However, the performance advantage is smaller because the number of skyline points is greater for the higher dimensionality. This results in the skyline computation used by the SF technique taking a greater amount of time than for the smaller dimensionalities, lowering the advantage relative to CF.

The results for varying the dataset cardinality for 8 dimensions are presented in Figures 5.7 a, b, and c for the anti-correlated, independent, and correlated distributions, respectively. As for the dimensionality experiments just discussed, the SF algorithm still performs better than the CF technique, but the performance advantage for the larger dimensionalities is smaller than before because of the larger number of skyline points in the datasets.

5.6.4 Real Datasets

In this section, we compare the performance of CF, SF, and the FMG algorithms on the two real datasets discussed in Section 5.6.1, the NBA players dataset and the IMDB dataset. From the NBA players dataset, we randomly select 5 dimensional subsets from the overall dataset and average the results for 10 such subsets. The results for both the NBA and the IMDB datasets are shown in Figure 5.8.

For both datasets, the performance of SF is the best of the three methods. It is more than a factor of 2 better than the FMG algorithm for the NBA players dataset and 3 times better on the IMDB dataset and SF is also more than an order of magnitude faster than the CF algorithm on both datasets for reasons already mentioned previously in Section 5.6.2.

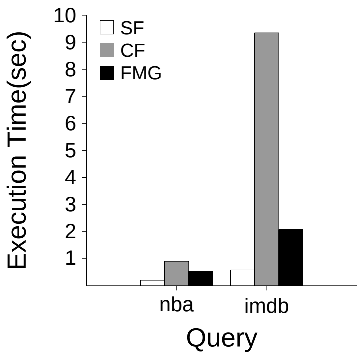


Figure 5.8: Performance results for the NBA players and IMDB datasets.

5.6.5 Accuracy

Next, we measure the overall accuracy of the results returned using the Skyline Point Ordering (SPO) method and compare this to the top-k skyline of FMG. We use two measures of accuracy for these experiments. The first is the Hyperarea Difference (HD) measure discussed in Section 3 that measures the spatial volume of a Pareto set. The second is the number of points in each dataset that are dominated by the top ranked k skyline points for each method (DOM).

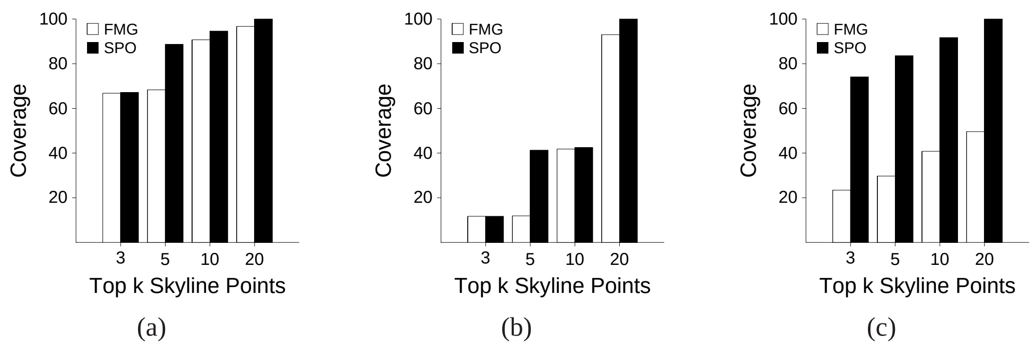


Figure 5.9: Hyperarea Difference accuracy measure for the (a) NBA players, (b) IMDB, and (c) synthetic 5 dimensional anti-correlated datasets for the FMG and Skyline Point Order techniques.

We again use the NBA players and IMDB real datasets and a synthetic 5 dimensional anti-correlated dataset with 500K data points (5A). The three datasets contain 22, 79, and 4079 skyline points for the IMDB, NBA players, and 5A datasets, respectively. We measure the top k=3, 5, 10, and 20 skyline points for each dataset.

The results for the Hyperarea Difference are presented in Figure 5.9 a, b, and c for the NBA players, IMDB, and 5A datasets, respectively. Here, we normalize the volumes by the largest result (typically when k is 20 for the SPO) so that results are represented as a percentage of this best result. In Figure 5.9 a and b, the SPO is covers slightly more volume than the results of FMG when k is 10 and 20 and the two have identical results when k is 3. When k is 10, the SPO results do cover a significantly larger volume. For the results in Figure 5.9 c, the SPO achieves significantly better volume coverage.

The results for the number of points dominated are presented in Figure 5.10 a, b, and c for the NBA players, IMDB, and 5A datasets, respectively. The DOM results are very similar for the IMDB dataset in Figure 5.10. For the NBA dataset, the top-k SPO results do achieve better results than that of the FMG. The top-k SPO results are significantly better for the 5A synthetic data.

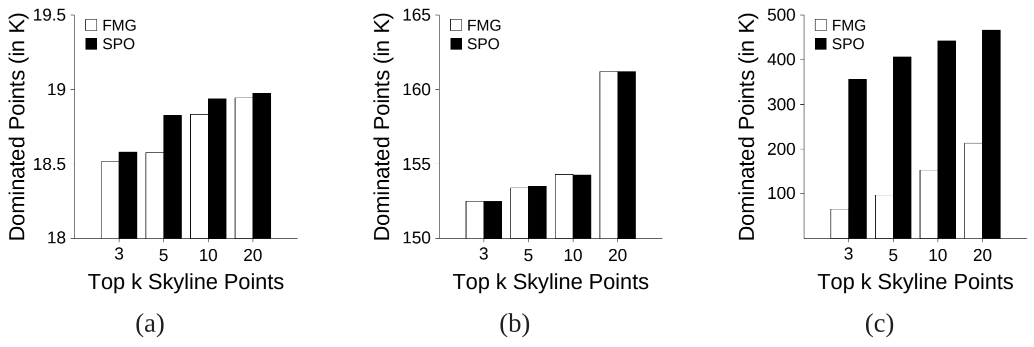


Figure 5.10: The number of dominated points for the (a) NBA players, (b) IMDB, and (c) synthetic 5 dimensional anti-correlated datasets for the FMG and Skyline Point Order techniques.

5.6.6 Summary

We have shown that the Skyline Point Order is generally more accurate than the top-k results of FMG as measured by the HD and DOM methods and that the SPO can be evaluated using the SF algorithm as quickly as the FMG method on correlated datasets and 2-3 times faster on other datasets.

5.7 Discussion

This work is currently the leading research on skyline point ranking. Hence, there are many potential future research opportunities in this area of work. In this section, we discuss a few open questions related to the SPO ranking methodology and point to directions for potential future work in this area. We also discuss the robustness of the SF and CF algorithms which evaluate the SPO.

The SPO method is sensitive to the placement of nonskyline points, which means that moving nonskyline points can result in a new ranking of the elements of the skyline. Consider the example shown in Figure 5.11. This is the same dataset as is shown in Figure 5.2, in which point d has the largest dominance set of any point. In Figure 5.11, points f and h are slightly perturbed, so that they are no longer dominated by point d. Now, point b has the largest dominance set of any point in the dataset, and hence it will outrank d.

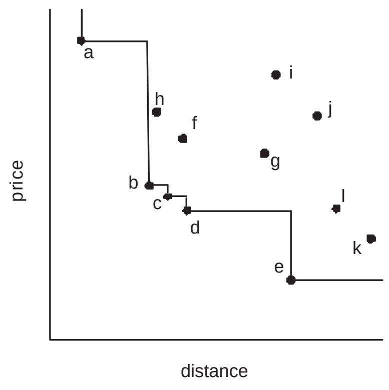


Figure 5.11: The skyline shown in Figure 5.2 with points f and h perturbed.

The ranking of the skyline elements in this case has changed. Point b is a better summary point than d now that it has a larger dominance set. This sensitivity to the placement of nonskyline points is one potential weakness of the Skyline Point Order ranking method. We can define a robust top-k ranked skyline point to be one that remains a top-k skyline point despite perturbing the underlying dataset.

Definition 5.7.1. *A robust top-k skyline point is one that remains a top-k skyline point when nondominant skyline points are perturbed by up to ϵ in any dimension.*

In this chapter, we have discussed the SPO as a method to rank skyline points, as well as a set covering definition for the top-k skyline points developed by [50]. Another method to find important skyline points is to consider which layer of the skyline they might reside. The skyline layers are indicated in Figure 5.12 for the example dataset first discussed in Figure 5.2. In Figure 5.12, there are four skyline layers. Each layer is indicated by how many skyline layers would have to be removed for points in that layer to be on the skyline. Those points in lower numbered layers are closer to the skyline than are points in higher numbered skyline layers. Exploring the potentials of the skyline layers is another area for future work.

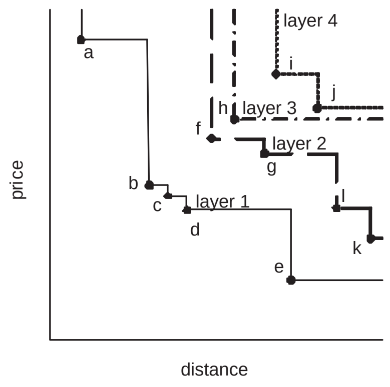


Figure 5.12: The skyline shown in Figure 5.2 with the first through fourth skyline layers indicated.

5.7.1 Cost Analysis for Modifying the SPO

In this subsection, we discuss the cost of the SF and CF algorithms for making modifications to the SPO. The Skyline Point Order is one method to rank skyline points, but modifications to the ranking can be made and still evaluated using the SF or CF algorithms. For example, introducing a log factor to the NNMD or NPD terms will produce a different ordering of the skyline points. For example:

$$SPO(s, D) \;=\; log_{k1}(NNMD(s, D)) \,*\, log_{k2}(NPD(s, D))$$

This modification will not produce a change in the cost of either the SF or CF algorithms and each method will remain unchanged, except for the final line of each that computes the SPO from the NNMD and NPD values. In general, the algorithms remain unchanged for any SPO that is a function of the NNMD and NPD values that are evaluated by each algorithm.

5.8 Conclusion

The skyline operation as a summary method suffers when the number of skyline points is large. We have proposed the novel Skyline Point Order as a method to rank skyline points so that the most important points in the skyline summary appear first in the skyline order, in contrast to other methods that return skyline sets unordered. We have further proposed the hyperarea difference metric as a quantitative measure of the summary value of skyline points. We have also shown that a top-k summary set produced using the Skyline Point Order is more effective summary set than one using an approximation to the k most representative skyline operator of [50] using both the hyperarea difference and the total number of points dominated criteria.

We have proposed two algorithms, the Coverage First and the Skyline First, to evaluate the Skyline Point Order and have shown that the Skyline First approach is more efficient than the nearest competing technique on experiments involving both synthetic and real datasets.

CHAPTER VI

Future Work and Conclusions

6.1 Conclusions

The analysis of large multidimensional datasets is increasing important for database systems. This is because the volume of data for such datasets is very large, and the applications that use and generate multidimensional datasets are plentiful. This necessitates efficient algorithms to mine and summarize these datasets.

In this thesis, we have described efficient algorithms for evaluating time-series similarity and for evaluating skyline sets. In Chapter 2, we have presented the FTSE algorithm for evaluation of time-series similarity measures that are based around an ϵ threshold-based scoring function. We showed that this technique is significantly faster than traditional evaluation measures such as dynamic programming. We have also presented the Swale scoring model that combines the notions of gap penalties and match rewards of previous models for comparing the similarity between time-series datasets. We have shown Swale to be more accurate compared to other existing measures using extensive experimental evaluation.

In Chapter 3, we have presented an algorithm for the efficient evaluation of continuous time-interval skyline queries, called *LookOut*. The *LookOut* algorithm continuously evaluates the skyline operator for temporal datasets in which data elements are valid for certain time ranges. This algorithm is shown to be more than an order of magnitude faster than existing techniques. In Chapter 3, we have also studied the performance of the quadtree for skyline evaluation and determined it to be a superior indexing structure to the R*-tree for skyline queries, both in the static and continuous time-interval contexts.

In Chapter 4, we have examined skyline computation for datasets whose attributes are drawn from low-cardinality domains and described the Lattice Skyline algorithm as a method to find skylines in this context. The Lattice Skyline algorithm differs from pre-existing measures because it does not use the familiar paradigm that eliminates points from skyline consideration by comparison with other points in the dataset. Rather, it uses the structure of the lattice defined by the low-cardinality attribute domains to identify skyline values. Skyline points can then be identified by comparison with the appropriate lattice entry. This gives LS an improved complexity result over other techniques and performance that is independent of the dataset attribute distributions.

The skyline operation as a summary method suffers when the number of skyline points is large. In Chapter 5, we have described a method to rank skyline points called the Skyline Point Order technique. The SPO returns the most important points to the skyline summary first in the skyline order, as opposed to other techniques that return them in an unspecified ordering. We have shown the top-k skyline result of the SPO method to be more accurate a summary than the k most representative skyline method. We have described the Coverage First and the Skyline First algorithms for evaluation of the SPO and have shown experimentally that the SF technique is more effective than competing techniques.

6.2 Future Research Directions

We would like to explore research opportunities in both traditional database research areas such as spatial and temporal data management as well as in interdisciplinary areas.

Spatial: We are interested in storage, indexing, and querying new types of spatial data including moving data, biomedical data types, and scientific data. We are also interested in researching new ways to obtain improved performance for spatial data including using new types of hardware. As we reach the limits of Moore's Law, new types of hardware including dual cores and commodity GPUs offer new alternatives for improving the efficiency of spatial data processing. We are also interested in pursuing new research directions that can result from the skyline and its varients.

Temporal: We are interested in a broad scope of research issues pertaining to the effective querying of temporal data. This thesis has focused on speeding up time-series comparison, and we are interested in pursuing new and faster comparison techniques further. One such direction is to use dictionary-based compression techniques such as Lempel-Ziv to search sequence data for motifs by examining common patterns found in the dictionary after compression. The Swale measure that is developed here treats all time-series datasets the same. We are also interested in classifying different types of time-series datasets based on underlying data characteristics. Designing similarity measures to focus on specific data classes offers new opportunities for improved accuracy.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] The Internet Movie Database. www.imdb.com.

[2] UCR Time Series Classification/Clustering Website.

[3] www.orbitz.com.

[4] Zillow. www.zillow.com.

[5] R. Agarwal, C. Faloutsos, and A. R. Swami. Efficient Similarity Search in Sequence Databases. In *FODO*, pages 69–84, 1993.

[6] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *ACM SIGMOD International Conference on Management of Data*, pages 322–331, 1990.

[7] J. Bentley, H. Kung, M. Schkolnick, and C. Thompson. On the average number of maxima in a set of vectors and applications. *JACM*, 25:536–543, 1978.

[8] D. Berndt and J. Clifford. Using Dynamic Time Warping to Find Patterns in Time Series. In *AAAI-94 Workshop on Knowledge Discovery in Databases*, pages 359–370, 1994.

[9] C. Böhm and H. Kriegel. Determining the Convex Hull in Large Multidimensional Databases. In *Data Warehousing and Knowledge Discovery (DaWaK)*, pages 294–306, 2001.

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

[11] M. Carey, D. DeWitt, M. Franklin, and et al. Shoring Up Persistent Applications. In *ACM SIGMOD International Conference on Management of Data*, pages 383–394, 1994.

[12] M. Carey and D. Kossmann. On Saying ”Enough Already!” in SQL. In *ACM SIGMOD International Conference on Management of Data*, pages 219–230, 1997.

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

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

[15] K.P. Chan and A.W-C Fu. Efficient Time Series Matching by Wavelets. In *ICDE*, pages 126–133, 1999.

[16] Surajit Chaudhuri, Vivek R. Narasayya, and Ravishankar Ramamurthy. Estimating Progress of Long Running SQL Queries. In *SIGMOD*, pages 803–814, 2004.

[17] L. Chen and R. Ng. On the Marriage of Lp-norms and Edit Distance. In *VLDB*, pages 792–803, 2004.

[18] L. Chen, M. T. Özsu, and V. Oria. Robust and Fast Similarity Search for Moving Object Trajectories. In *SIGMOD*, pages 491–502, 2005.

[19] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with Presorting. In *International Conference on Data Engineering (ICDE)*, pages 717–720, 2003.

[20] R. Cilibrasi and P. M. B. Vitanyi. Clustering by Compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545, 2005.

[21] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast Subsequence Matching in Time-Series Databases. In *SIGMOD*, pages 419–429, 1994.

[22] Irene Gargantini. An Effective Way to Represent Quadtrees. *Communications of the ACM*, 25(12):905–910, 1982.

[23] J. Gips, M. Betke, and P. Fleming. The Camera Mouse: Preliminary Investigation of Automated Visual Tracking for Computer Access. In *Proc. of Rehabilitation Engineering and Assistive Technology Society of North America (RESNA)*, pages 98–100, 2000.

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

[25] D. Goldin and P. Kanellakis. On Similarity Queries for Time-Series Data: Constraint Specification and Implementation. In *Constraint Programming*, pages 137–153, 1995.

[26] A. Guttman. R-Tree: A Dynamic Index Structure for Spatial Searching. In *ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[27] G. Hjaltason and H. Samet. Distance Browsing in Spatial Databases. *ACM Transactions on Database Systems (TODS)*, 24(2):265–318, 1999.

[28] Gisli R. Hjaltason and Hanan Samet. Speeding up Construction of PMR Quadtree-based Spatial Indexes. *The VLDB Journal*, 11(2):109–137, 2002.

[29] Z. Huang, H. Lu, B.C. Ooi, and A.K.H. Tung. Continuous Skyline Queries for Moving Objects. *Transactions on Knowledge and Data Engineering (TKDE)*, 18(12):1645–1658, 2006.

[30] J. W. Hunt and T. G. Szymanski. A Fast Algorithm for Computing Longest Common Subsequences. *CACM*, pages 350–353, 1977.

[31] F. Itakura. Minimum Prediction Residual Principle Applied to Speech Recognition. *IEEE Trans. Acoustics, Speech, and Signal Proc.*, Vol. ASSP-23:52–72, 1975.

[32] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, 1988.

[33] E. Keogh. Exact Indexing of Dynamic Time Warping. In *VLDB*, pages 406–417, 2002.

[34] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra. Dimensionality Reduction for Fast Similarity Search in Large Time Series Databases. *KAIS*, 3(3):263–286, 2000.

[35] E. Keogh and S. Kasetty. The Need for Time Series Data Mining Benchmarks: A Survey and Empirical Demonstration. In *SIGKDD*, pages 102–111, 2002.

[36] E. Keogh, S. Lonardi, and C. A. Ratanamahatana. Towards Parameter-Free Data Mining. In *SIGKDD*, pages 206–215, 2004.

[37] E. Keogh and M. Pazzani. Scaling Up Dynamic Time Warping to Massive Datasets. In *PKDD*, pages 1–11, 1999.

[38] E. J. Keogh, K. Chakrabarti, S. Mehrotra, and M. J. Pazzani. Locally Adaptive Dimensionality Reduction for Indexing Large Time Series Databases. In *SIGMOD*, pages 151–162, 2001.

[39] Sang-Wook Kim, Sanghyun Park, and Wesley W. Chu. An Index-Based Approach for Similarity Search Supporting Time Warping in Large Sequence Databases. In *ICDE*, pages 607–614, 2001.

[40] You Jung Kim and Jignesh M. Patel. Rethinking Choices for Multi-dimensional Point Indexing: Making the Case for the Often Ignored Quadtree, 2007.

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

[42] F. Korn, H. Jagadish, and C. Faloutsos. Efficiently Supporting Ad Hoc Queries in Large Datasets of Time Sequences. In *SIGMOD*, pages 289–300, 1997.

[43] D. Kossmann, F. Ramsak, and S. Rost. Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In *Very Large Databases (VLDB)*, pages 275–286, 2002.

[44] Ravi Kanth V Kothuri, Siva Ravada, and Daniel Abugov. Quadtree and R-tree Indexes in Oracle Spatial: A Comparison Using GIS Data. In *ACM SIGMOD International Conference on Management of Data*, pages 546–557, 2002.

[45] 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.

[46] S. Kuo and G. R. Cross. An Improved Algorithm to Find the Length of the Longest Common Subsequence of Two Strings. *ACM SIGIR Forum*, 23(3-4):89–99, 1989.

[47] 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.

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

[49] X. Lin, Y. Yuan, W. Wang, and H. Lu. Stabbing the Sky: Efficient Skyline Computation over Sliding Windows. In *International Conference on Data Engineering (ICDE)*, pages 502–513, 2005.

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

[51] D. Littau and D. Boley. Streaming Data Reduction Using Low-Memory Factored Representations. *Information Sciences*, 176(14):2016–2041, 2006.

[52] Gang Luo, Jeffrey F. Naughton, Curt Ellmann, and Michael Watzke. Toward a Progress Indicator for Database Queries. In *SIGMOD*, pages 791–802, 2004.

[53] Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, and Y. Theodoridis. *R-Trees: Theory and Applications*. Series in Advanced Information and Knowledge Processing. Springer, 2005.

[54] Y. Manolopoulos, Y. Theodoridis, and V.J. Tsotras. *Advanced Database Indexing*. Springer, 1999.

[55] J. Matoušek. Computing Dominances in E^n. *Information Processing Letters*, 38(5):277–278, June 1991.

[56] D. McLain. Drawing Contours from Arbitrary Data Points. *The Computer Journal*, 17(4):318–324, 1974.

[57] Michael Morse, Jignesh M. Patel, and William I. Grosky. Efficient Continuous Skyline Computation. In *International Conference on Data Engineering (ICDE)*, page 108, 2006.

[58] Climate variability and predictability website. http://www.clivar.org.

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

[60] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive Skyline Computation in Database Systems. *ACM Transactions on Database Systems (TODS)*, 30(1):41–82, March 2005.

[61] C. Papadimitriou and M. Yannakakis. Multiobjective Query Optimization. In *ACM Principles of Database Systems*, pages 52–59, 2001.

[62] 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.

[63] I. Popivanov and R.J. Miller. Similarity Search Over Time Series Data Using Wavelets. In *ICDE*, page 212, 2001.

[64] F. Preparata and M. Shamos. *Computational Geometry - An Introduction*. Springer, 1985.

[65] C.A. Ratanamahatana and E. Keogh. Making Time-series Classification More Accurate Using Learned Constraints. In *SIAM International Conference on Data Mining*, 2004.

[66] N. Roussopoulos, S. Kelly, and F. Vincent. Nearest Neighbor Queries. In *ACM SIGMOD International Conference on Management of Data*, pages 71–79, 1995.

[67] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 1995.

[68] H. Sakoe and S. Chiba. Dynamic Programming Algorithm Optimization for Spoken Word Recognition. *IEEE Trans. Acoustics, Speech, and Signal Proc.*, Vol. ASSP-26(1):43–49, 1978.

[69] Y. Sakurai, M. Yoshikawa, and C. Faloutsos. FTW: Fast Similarity Search under the Time Warping Distance. In *PODS*, pages 326–337, 2005.

[70] H. Samet. The Quadtree and Related Hierarchical Data Structures. *Computing Surveys*, 16(4):187–260, 1984.

[71] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.

[72] R. Steuer. *Multiple Criteria Optimization*. Wiley, NY, 1986.

[73] I. Stojmenovic and M. Miyakawa. An Optimal Parallel Algorithm for Solving the Maximal Elements Problem in a Plane. *Parallel Computing*, 7(2):249–251, 1988.

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

[75] Y. Tao and D. Papadias. Maintaining Sliding Window Skylines on Data Streams. *Transactions on Knowledge and Data Engineering (TKDE)*, 18(2):377–391, 2006.

[76] S. Hettich and S. D. Bay. The UCI KDD Archive [http://kdd.ics.uci.edu].

[77] M. Vlachos, M. Hadjieleftheriou, D. Gunopulos, and E. Keogh. Indexing Multi-Dimensional Time-Series with Support for Multiple Distance Measures. In *SIGKDD*, pages 216–225, 2003.

[78] M. Vlachos, G. Kollios, and D. Gunopulos. Discovering Similar Multidimensional Trajectories. In *ICDE*, pages 673–684, 2002.

[79] W.-T. Wong, F. Y. Shih, and T.-F. Su. Thinning Algorithms Based on Quadtree and Octree Representations. *Information Sciences*, 176(10):1379–1394, 2006.

[80] J. Wu and S. Azarm. Metrics for quality assessment of a multiobjective design optimization solution set. *J. of Mechanical Design*, 123(1):18–25, 2001.

[81] J. Yang and W. Wang. CLUSEQ: Efficient and Effective Sequence Clustering. In *ICDE*, pages 101–112, 2003.

[82] B-K Yi and C. Faloutsos. Fast Time Sequence Indexing for Arbitrary Lp Norms. In *VLDB*, pages 385–394, 2000.

[83] Byoung-Kee Yi, H. V. Jagadish, and Christos Faloutsos. Efficient Retrieval of Similar Time Sequences Under Time Warping. In *ICDE*, pages 201–208, 1998.

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

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

[86] Y. Zhu and D. Shasha. Warping Indexes with Envelope Transforms for Query by Humming. In *SIGMOD*, pages 181–192, 2003.