Answering Topband Queries in Time Series Data

LI LING

A Thesis submitted for

the Degree of Master of Science

Department of Computer Science

School of Computing

National University of Singapore

· 2007 ·

Abstract

Top k queries are queries that request for k answers having the highest or lowest values for some attribute, expression, or function. These queries arise naturally in many database applications where users are interested in finding records that are closest to the values specified in a query. Example applications include census data analysis, data mining, information retrieval and similarity search of multimedia data. For example, rather than finding all publications on a certain topic, a researcher may want to retrieve the ten most heavily referenced papers on the topic at hand.

There has been a long stream of research work that address the efficient evaluation of top k queries in relational databases. In this thesis, we investigate the usefulness of top k queries in time series data and introduce a new class of queries called $\lceil k \rceil$-topband. Topband queries aim to retrieve objects that are within top k at every time point over a specified time interval. This kind is queries is designed from the observation that objects which exhibit some consistent behavior over a period of time would enable decision-makers to assess, with greater confidence, the potential merits of the objects. A rank-based approach is proposed to evaluate topband queries efficiently. Experiment results on both synthetic and real world datasets indicate that the proposed approach is efficient and scalable, and has direct applications in real world scenarios.

Contents

List of Figures

List of Tables

Acknowledgments

First and most importantly, I am extremely grateful to my supervisor A/P Lee Mong Li and A/P Wynne Hsu. They have given me the most valuable guidance that an adviser can give her students. Their helpful comments, suggestions and insightful criticism are invaluable to my research work.

I am also very grateful to my friends from database group for their continuous support and those valuable discussions and suggestions.

Finally, I would like to express my love and gratitude to my family who have always been supporting and encouraging me.

Chapter 1

Introduction

Time series data are of growing importance in many new database applications. A time series (or time sequence) is a sequence of real numbers, each number representing a value at a time point. Typical examples include stock prices or currency exchange rates, weather data, etc. Recently, there has been an explosion of interest on time series databases due to its usefulness in knowledge discovery. Many high level representations of time series [11, 14, 19, 21, 24, 28, 30, 37], and distance functions for sequence and/or subsequence matching are proposed [1, 11, 31, 32, 36]. However, all these works are focused on similarity matchings which include range queries, best-match queries and k-nearest neighbor queries.

We observe that time series data is also very useful in decision marking because it captures historical data. Oftentimes, decisions that are made based on one time point observation may not be as reliable or durable as decisions that are made based on observations over a period of time. In fact, many real world applications such as online stock trading and analysis, traffic management systems, weather monitoring, disease surveillance and performance tracking, have large repositories of historical data. Finding objects that exhibit some consistent behavior over a

period of time would enable decision-makers to assess, with greater confidence, the potential merits of the objects.

In this work, we define a class of queries call *topband* to retrieve objects with some persistent performance over time. The states of an object over time constitute a *time series*. We will first illustrate with examples the relevance of topband queries in various applications.

Example 1. Stock Portfolio Selection. In selecting a portfolio of stocks for long-term investment, investors would have greater confidence in stocks that consistently exhibit above industry average in growth in earnings per share and returns on equity. These stocks are more resilient when the stock market is bearish and may be a better choice than volatile stocks. We can issue a topband query to return a set of stocks whose growth in earnings per share or return on equity are consistently above the 50th percentile over a period of time.

Example 2. Targeted Marketing. The ability to identify "high value" customers is valuable to companies who are keen on marketing their new products or services. These "high value" customers usually have been with the company for some time and have regular significant transactions. Marketing efforts that are directed to this group of customers are likely to be more profitable than those to the general customer base. The topband query allows these "high value" customers to be retrieved. This will allow the company to develop appropriate strategies that will further its business goals.

Example 3. Awarding Scholarships. Organizations that provide scholarships have many criteria for selecting suitable candidates. One of the selection

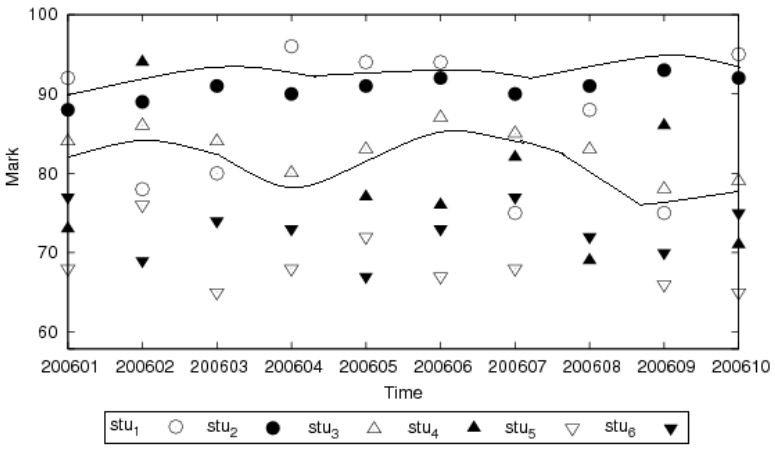


Figure 1.1: Example *student* dataset with $\{stu_2, stu_3\}$ being consistently in the top 3.

criteria often requires the students to have demonstrated consistent performance in their studies or leadership roles. The topband query can be used to retrieve this group of potential scholarship awardees.

Formally, given a time series dataset, we define the $\lceil k \rceil$-topband as the set of time series which are ranked among the top k at *every* time point. The parameter $\lceil k \rceil$ denotes the size of the answer set, and ranges between 0 to k.

Figure 1.1 shows a sample *student* dataset which records the test marks of six students in the first ten months in 2006, assuming there is one test per month. A $\lceil 3 \rceil$-topband query to retrieve students who are consistently in the top 3 for every test over the ten months will yield $\{stu_2, stu_3\}$. Note that the size of the answer set does not need to be 3.

We also introduce a variation of $\lceil k \rceil$-topband queries. The purpose is to retrieve a set of time series that outperforms a particular time series. For example, suppose we want to find a set of stocks whose gains are always greater than some reliable stock such as IBM for the past month. Again, we can retrieve the set

of stocks whose gains are higher than IBM for each day in the last month and compute the intersection. Note that the number of stocks that outperform IBM would vary from day to day, that is, the value k may be changed from one time point to another. We call such queries k*-topband, to indicate the changing k at different time points.

So far, we have posed a strict condition on $\lceil k \rceil$/k*-topband queries. That is, the candidate object must perform well at *EVERY* time point. However, in practice, it is very hard to find such objects. For example, when awarding scholarships (Example 3), the formulation of topband queries requires good performances for all time points. But there may be extenuating circumstances beyond a student's control which may lead to a temporary drop in his/her performance. In this case, it should be relaxed to disregard the students' performance for a few time points. In this work, we will apply the *Haar Wavelet Transform* technique to the original dataset to get the candidate objects for the less restrictive topband queries. The disadvantage of utilizing *Haar Wavelet Transform* is that the result set may contain some objects which do not perform well at few time points. However, the cost/space cost will be reduced while processing the less restrictive topband queries, as well as it is more close to the real practice. We will discuss more about the less restrictive version of topband queries in Chapter 5.

1.1 Contribution

A naive approach to process a $\lceil k \rceil$-topband query is to consider it as a set of top-k queries over a continuous time interval. For each time point in the time interval, we obtain the top-k answers and compute their intersections. It is clear that this straightforward approach can be potentially expensive with many redundant

computations.

We address this shortcoming and develop a rank-based approach to evaluate topband queries efficiently. The time series at each time point are ranked; the time series with the highest value at a time point has a rank of 1. We observe that the rank of a time series is only affected when it intersects with other time series. Referring to our example in Figure 1.1, for the January test, stu_1 is ranked first while stu_2 is ranked second. However, in the February test, the rank of stu_1 drops. Note that the time series for stu_1 intersects with that of stu_2, stu_3 and stu_4 between the two tests.

Based on this observation, we design an efficient algorithm to construct a compact *RankList* structure from a time series dataset. With this structure, we can quickly answer topband queries. Furthermore, the *RankList* structure can be implemented on top of any relational database system. In the following chapters, we will describe how to utilize existing approaches to answer topband queries, as well as our proposed method.

The main contributions of this thesis are summarized below.

1. We give a formal definition of topband queries and explain how a traditional relational database system handles such queries. We also describe how topband queries can be answered using SQL and existing top k methods and highlight the drawbacks of these approaches.

2. We propose a technique that utilizes rank information to answer topband queries efficiently. Algorithms to construct, search and update the *RankList* structure are presented.

3. We present a suite of comprehensive experiment results to show the efficiency and scalability of our proposed method. We also demonstrate that

top-band queries are able to retrieve interesting results from two real world *stock* and *student* datasets.

1.2 Organization

The rest of the thesis is organized as follows. Chapter 2 provides a review of related work. We first discuss how similarity queries and top k queries are evaluated in time series data. Then we show how topband queries are different from top k and skyline queries.

Chapter 3 presents the proposed technique that utilizes rank information to answer topband queries efficiently. More specifically, it describes the RankList structure which can capture the rank information of each time series. Various algorithms to construct, search and update the RankList are given as well as the analysis of their time complexity.

Chapter 4 describes how topband queries can be answered using existing SQL and top k methods and highlights the drawbacks of each method. We show how the proposed RankList structure can be implemented on top of relational database.

Chapter 5 presents a suit of comprehensive experiment results to show the efficiency and scalability of the proposed method, as well as the direct application of topband queries in real world scenario.

Finally, we conclude in Chapter 6 with directions for future work.

Chapter 2

Related Work

In time series database, much research effort has been put on retrieving similar matches which include range queries, best-match queries and k-nearest neighbor queries. Given the inherent high dimensionality of time series data, this problem becomes even complex. In this chapter, we first review various high level representations of time series and distance functions for sequence/subsequence matching.

Given that a naive approach to process topband queries is to obtain the top-k answers at every time point and compute their intersections, we will review the various methods to process kNN and top k queries. Note that top k query is a special case of kNN query.

Finally, we discuss the differences between topband queries, top k, and skyline queries, thus providing a clear idea of topband queries.

2.1 Similarity Queries in Time Series Data

Research in time series data has been concentrated on answering similarity queries which include range queries, best-match queries and k nearest neighbor queries. Two main approaches are developed in order to process similarity queries efficiently. One is to reduce the dimensionality on the data and the other one is to transform the data into a string of symbols. In this section, we will review how existing approaches utilize these two approaches. Furthermore, we will discuss the distance measured used in time series data to measure the similarity of two sequences.

2.1.1 Dimension Reduction on Data

The most promising solutions to answer similarity queries involve performing dimensionality reduction on the data, then indexing the reduced data with a spatial access method. Four major dimensionality reduction techniques have been proposed in the previous work. They are Discrete Fourier Transform [1,32], Piecewise Aggregate Approximation [24,25], Discrete Wavelet Transform [11] and Singular Value Decomposition [36].

[1] discuss DFT approach. The basic idea of this approach is that obtaining DFT coefficients using the Algorithm Fast Fourier Transform (FFT), cutting off all but the first few Fourier coefficients and calculating the square root of the sum of the squared differences of these coefficients. If the difference is below a user-defined threshold, then the two sequences are considered to be similar. The reason to choose DFT is because it is the most well known, its code is readily available, it does a good job of concentrating the energy in the first few coefficients and the amplitude of the Fourier coefficients is invariant under shifts. [32] further

propose to use the last few Fourier coefficients of a time sequence in the distance computation since every coefficient at the end is the complex conjugate of a coefficient at the beginning and as strong as its counterpart. In this way, the search time of the index can be reduced by more than 50 percent in most cases. However, DFT suffers the problems that it cannot capture the feature of time localization.

[11] discuss to use Haar Wavelets to reduce the dimensionality. Haar transform can be seen as a series of averaging and differencing operations on a discrete time function. One advantage of DWT over DFT is that DWT can capture the feature of time localization. However, it is only defined for sequences whose length is an integral power of two.

To overcome the drawbacks of DFT and DWT, [25] propose Piecewise Aggregate Approximation (PAA) approach. In order to reduce the data to N dimensions, PAA approach divides the data into N equi-sized "frames" and calculates the mean value of the data falling within a frame, taking a vector of these values to be the data reduced representation. PAA requires each segment is of the same length, while [24] relax this requirement by allowing the segments to have arbitrary lengths. This approach is called APCA (Adaptive Piecewise Aggregate Approximation). APCA can capture the shapes of time series data more accurately than PAA and have a less response time.

2.1.2 String of Symbols

Another approach to answer similarity queries is to transform data into a string of symbols, then index these symbols accordingly. Three pieces of work [2, 3, 22] have discussed this approach.

[2] present a shape definition language, called SDL, for retrieving objects

based on shapes contained in the histories associated with these objects. Eight symbols are proposed in [2] to describe transitions of objects from one time instant to the following one and a four-layer hierarchical storage structure, which also acts as an index structure, is designed to store these symbols. The advantage of [2] is its ability to perform blurry matching and efficient implementability. However, SDL can be only used to do blurry matching, not the exact matching.

[3] and [22] also translate the data into a string of symbols by calculating the amplitude difference between two adjacent samples. [3] adopt signature files to index the text-string while suffix tree is utilized as index in [22].

All these three transformation techniques aim to capture the shape information of the time series, but losing the actual data values in the process.

2.1.3 Distance Measure

Besides Euclidean distance, which is the most well known distance measure, dynamic time warping (DTW) [4] is also a much more robust distance measure for time series. One advantage using DTW is that DTW allows similar shapes to match even if they are out of phase in the time axis. [23] show that PAA [25] can be adapted to allow indexing under DTW. [26] propose a modification of DTW called Derivative Dynamic Time Warping (DDTW). Instead of considering values in the Y axis of the datapoints, DDTW considers the first derivative of the sequences. Compared to DTW, DDTW can avoid "singularities" and can find obvious, natural alignments in two sequences simply even if a feature in one sequence is slightly higher or lower than its corresponding feature in the other sequence.

2.2 kNN Queries in Relational Database

A typical approach to answer kNN queries is *partitioning approach*, which partitions the data space recursively and stores information about the partitions in the nodes. *Cell* method [35], *R-tree* approach [33], *k-d-tree* approach [20] and *Quad-tree* approach are key methods in partitioning approach.

2.2.1 Cell Method

The cell method [35] is a straightforward technique for solving the best match or nearest neighbor problem. The algorithm divides the data space into identical cells and stores the data objects inside a cell in a list which is attached to the cell. During nearest neighbor search the cells are visited in order of their distance to the query point. The search terminates if the nearest point which has been determined so far is nearer than any cell not visited yet. Although this procedure minimizes the number of records examined, it is extremely costly in space and time, especially when the dimensionality of the space is large.

2.2.2 R-Tree

[33] propose an approach that uses R-tree for nearest neighbor search. Two metrics are computed for each Minimum Bounding Rectangle (MBR) for ordering and pruning search. One metric is MinDist, which is the minimum possible distance from the query point to the rectangle. The other metric is MinMaxDist. This is computed as the maximum possible distance from the query point to the nearest data point inside the rectangle. The algorithm traverses the R-tree and stores for every visited rectangle a list of subrectangles ordered by their MinMaxDist. Three pruning strategies are adopted when traversing:

1. An MBR M and the query point P with MinDist(P, M) greater than the MinMaxDist(P, M') of another MBR M' is discarded because it cannot contain the Nearest Neighbor(NN).

2. An actual distance from P to a give object O which is greater than the MinMaxDist(P, M) for an MBR M can be discarded because M contains an object O' which is nearer to P.

3. Every MBR M with MinDist(P, M) greater than the actual distance from P to a given object O is discarded because it cannot enclose an object nearer than O.

The algorithm is terminated when there is no items in the list. One disadvantage of this algorithm is that it traverses the index in a depth-first fashion. Subnodes are stored before descent, but once a branch has been chosen, its processing has to be completed, even if sibling branches appear more likely to contain the NN. The algorithm therefore accesses more partitions than actually necessary. Furthermore, R-tree cannot scale well when the number of dimensions is up to 16.

2.2.3 k-d-Tree & Quad-Tree

k-d-tree [20] and *Quad-tree* are both multidimensional tree structures that extend the binary search tree to multidimensional data. Both of them accomplish the three functions of the binary search tree: storing the records, dividing space into hyperrectangles and providing a directory among the hyperrectangles. The critical exception is that we have to choose at each internal node one of k keys to use as a discriminator in a multidimensional tree.

The algorithm to construct a k-d-tree is to choose for the discriminator that coordinate j for which the spread of attribute values is maximum for the subcollection represented by the node. The partitioning value is chosen to be the median value of this attribute. The algorithm to construct quad-tree is to partition the search space into four quadrants.

Range search with k-d-tree is straightforward. Starting at the root, the k-d-tree is recursively searched in the following manner. When visiting a node that discriminates by the jth key, one compares the jth range of the query with the discriminator value. If the query range is totally above (or below) that value, then one need only search the right subtree (respectively, left) of that node; the other son can be pruned from the search because any node it contains does not satisfy the query in that particular key. If the query range overlaps the node's key, then both children need to be searched. This can be accomplished by searching both children recursively. Range searching with Quad-tree is similar.

These two structures are most effective in situations where little is known about the nature of the queries or a wide variety of queries are expected.

2.3 Top k Queries in Relational Database

The work in [16, 17] first address top k queries when dealing with queries containing image content. They use *grade* to represent the extent to which that object fulfills the condition, where the larger the grade is, the better the match. They observe that for queries with non-boolean attributes, like *"color = 'red'"* or *"shape = 'round'"*, grade may be intermediate values between 0 and 1 instead of the exact value 0 or 1. They call such non-boolean attributes *multimedia attributes*. The result of such queries with multimedia attributes should be a sorted

list items in its database that match the query the best.

The work in [16,17] assume each of these multimedia attributes have a native sub-system that answers top k queries involving only the corresponding attribute. In the first phase of the proposed Fagin's Algorithm A_0, for each condition on the corresponding attribute, the query processing system obtains a set L of streams of top matches from the corresponding sub-system. This process terminates until there are at least k objects in the intersection of L. In the second phase, Algorithm A_0 computes the score of each of the retrieved objects, and returns the best k objects. Some research [17] further address the problem that certain subqueries may obtain extra weights. However, Algorithm A_0 is unable to provide an accurate estimation in the presence of correlation among attributes and skewed distribution.

The work in [18] generalize Fagin's Algorithm A_0([16]) as Fagin's threshold algorithm [18] (TA). TA assumes that each attribute of the multidimensional data space has an index list. The index list can be utilized to access the data items in descending order of the "local" score for the given attribute with regard to an elementary query condition. There are two modes to access data utilized in TA. One is referred to as "sorted access", which will output the graded set of all objects, one by one, along with their grades under the query, in sorted order based on grade. The other one is termed as "random access". It will output the grade of a given object. In the first step, TA does sorted access in parallel to each of the sorted lists. When an object R is seen under sorted access in some list, TA does random access to the other lists to find the corresponding grade of R. Then it will compute the grade of R. If the grade value is one of the k highest it has seen, remember R and its grade. In the second step, for each list, TA defines the *threshold value* τ to be the grade of the last object seen under sorted access

of that list. As soon as at least k objects have been seen whose grade is at least equal to τ, then halt. In the last step, TA outputs the k objects that have been seen with the highest grades.

One disadvantage of the TA method([18]) is that it moves to the next object only after probing all needed sources of the current object. This in turn incurs more access cost. To overcome this drawback, [34] propose to calculate the probability that the total score exceeds a threshold that would make the item interesting for the top k result based on the assumption of the data distribution. If this probability is sufficiently low, it drops the data item from the candidate list. However, this method would result in some false dismissals.

[8] also aim at avoiding the overly conservative best-score/worst-score bounds of the TA method([18]). It proposes an efficient evaluation of top k queries over a (distributed) "relation" whose attributes are handled and provided by autonomous sources accessible over the web with a variety of interfaces. The expected score is estimated, and upper and lower bounds for the scores are explored in order to prune objects in the first few steps instead of scanning the whole values of an object. In this way, [8] spend less time to process top k queries compared to [18].

2.4 Map Top k Queries to SQL Selection Queries

Another stream of research work to answer top k queries is to map top k queries into SQL selection queries [6,7,9,10,12,13,15,29]. The work in [9,10] illustrate the inefficiencies inherent in a relational DBMS to handle top k queries, and proposes adding a *STOP AFTER* clause to SQL to allow query writers and query tools to explicitly limit the cardinality of a query result. A *STOP* operator, which

produces the top or bottom k tuples of its input stream in a specified order and discards the remainder of the stream, as well as two implementation methods *Scan-Stop* and *Sort-Stop*, are proposed in [9] to efficiently process *STOP AFTER* clause. Furthermore, [10] present additional strategies based on the use of range partitioning techniques and semi-join-like methods to process the *STOP AFTER* clause. However, both work suffer from the drawback that the techniques in [9,10] can only be used after evaluating the score for each object. Hence, these strategies require a preprocessing step to compute the scoring function itself involving one sequential scan of all the data.

To overcome the drawbacks in [9, 10], the work in [12] examine how a top k query can be mapped to a multi-attribute range query. The key issue is to determine an appropriate *search distance* d which would retrieve the k best matches for the query. [12] use the histogram-based statistics on the relations to determine the *search distance*. Unfortunately, using only relatively coarse histograms to identify such a precise value for d is not possible. Therefore, there are two scenarios when processing top k queries by [12]. The first scenario is called *pessimistic scenario*. The pessimistic heuristic uses a largest possible value for d of the selection query, and usually results in an answer set much larger than k points. However, it guarantees that the actual top k points are included in the answer set. The other scenario is *optimistic scenario*. It uses a smallest possible value for d, resulting in a smaller selection query and thus less access cost than the pessimistic strategy. However, the resultant selection query usually returns far less than k points. When this happens, the query must be "re-started" by using a larger d, which in turn incurs extra access cost.

In order to find more precise value of d, [6] introduce a single value in each histogram bucket computed using a variation of the fractal dimension concept,

which models multidimensional data skews within buckets. Using this value, a more precise value of d can be determined from the optimistic scenario to the pessimistic scenario in [12]. Furthermore, [7] propose to use the particular workload of query to find the optimal value of d. However, histogram-based approach still has drawbacks in maintenance overhead and scaling.

In order to overcome the histogram drawbacks, the work in [13] propose a sampling-based approach to map a top k query to a multi-attribute range query. A sampling set S is first chosen and the first several points are retrieved from S according to their distance with the query point q in ascending order. These points are used subsequently to determine the appropriate *search distance* to map to selection queries. Compared to the histogram-based approach in [9], the sampling-based approach [13] has advantages in terms of estimation accuracy, run-time efficiency and resource usage. Recently, [15] compute the search distance by taking into account imprecision in the optimizer's knowledge of data distribution and selectivity estimation.

2.5 Topband vs. Top-k and Skyline Queries.

Top k query requests for k answers having the highest or lowest values for some attribute, expression, or function, whereas skyline query retrieves objects which are not dominated by other objects on every attribute. Both of them aim to retrieve objects with outstanding values.

Topband query is different from top k and skyline queries such that it aims to retrieve the set of objects that show some consistent performance over time. However, mapping a time series dataset to a multi-dimensional dataset, and using top-k or skyline query methods may not be able to retrieve the desired set of

id	test 1	test 2	average mark
stu_1	92	79	85.5
stu_2	88	89	88.5
stu_3	84	86	85
stu_4	77	94	85.5
stu_5	72	76	74
stu_6	78	73	75.5

Table 2.1: Computing the average scores of the students' January and February tests to illustrate top-k query.

objects.

To illustrate, we continue with the example in Figure 1.1 and consider the performance of the students for only the January and February tests. A $\lceil 3 \rceil$-topband query over [200601, 200602] will retrieve the students stu_2 and stu_3 since their test scores for January and February are consistently within the top 3 highest.

A top-k query retrieves k objects which have the highest scores based on some monotonic function [9]. Table 2.1 lists the January and February test marks for the students and their averages. A top-3 query will retrieve students stu_1, stu_2 and stu_4, but stu_1 has not done well in the February test and stu_4 has not done well in the January test.

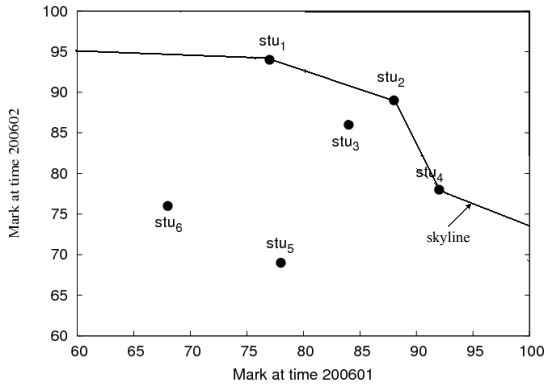


Figure 2.1: Mapping the January and February test marks in Figure 1 to a 2-D space to illustrate skyline query.

Now let us map the January and February test marks of the students in Figure 1.1 to a two-dimensional space as shown in Figure 2.1. The x-axis and y-axis in Figure 2.1 represent the marks of the students in January and February respectively. A skyline query retrieves a set of points from a multi-dimensional dataset which are not dominated by any other points [5]. Figure 2.1 shows the results of a skyline query (stu_1, stu_2, stu_4). Note that stu_1 and stu_4 are retrieved although they have not done well in one of the two tests. Further, $stu3$ who has consistently scored above 85, is not retrieved by the skyline query.

In the following chapters, we will describe a rank based approach to answer topband queries and show how the proposed method can be implemented on top of a relational database system.

Chapter 3

Answering Topband Queries with Rank Method

A time series s is a sequence of values that change with time. We use $s(t)$ to denote the value of s at time t, $t \in [0, T]$. A time series database TS consists of a set of time series s_i, $1 \leq i \leq N$. Given a time series database TS, an integer k, and a time point t, a top-k query will retrieve k time series with the highest values at t. We use top-$k(TS, k, t)$ to denote the set of top k time series at t.

A $\lceil k \rceil$-topband query over a time interval $[t_u, t_v]$ will retrieve the set of time series $U = \bigcap U_t$ where U_t = top-k(TS, k, t) $\forall$ $t \in [t_u, t_v]$. Note that the size of U is between 0 to k. In this chatper, we present an approach that utilizes rank information to efficiently process topband queries.

The rank of the various time series at each time point can be obtained by sorting the values of the time series at each time point. We observe that the rank of a time series s at a time point t, denoted by rank(s,t), is affected by the intersection of s with other time series between the time points $t-1$ and t. Figure 3.1 illustrates how the rankings of a set of time series may be affected by

intersections.

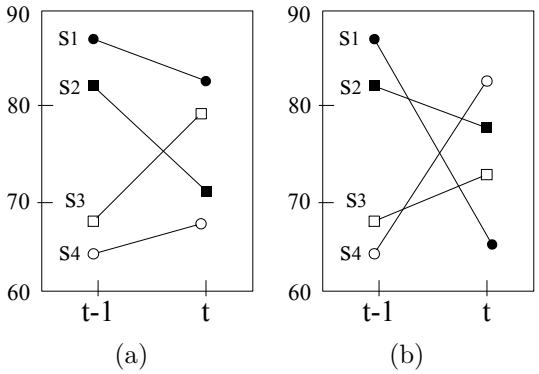


Figure 3.1: Rankings of time series

There are three cases:

1. A time series s does not intersect with any other time series between the time points $t-1$ and t. In this case, rank(s,t) = rank(s,$t-1$).

   For example, the time series s_1 and s_4 in Figure 3.1(a) do not intersect with other time series between t_1 and t_2. Therefore, there is no change in their rankings at these time points.

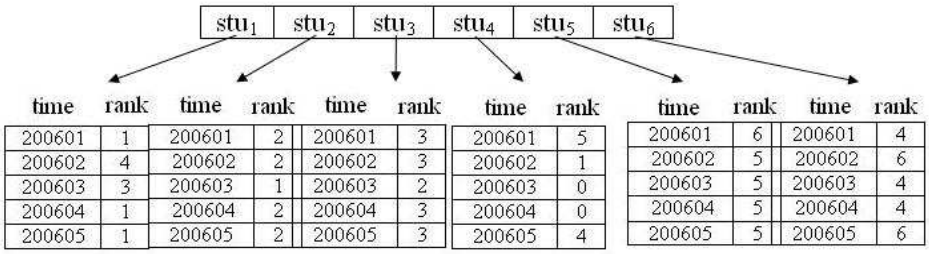
2. A time series intersects with other time series between the time points $t-1$ and t, leading to a change in the ranking of the time series.

   For instance, the time series s_2 and s_3 in Figure 3.1(a) intersects with each other between t_1 and t_2. We have rank(s_2,t_1)=2, rank(s_2,t_2)=3, and rank(s_3,t_1)=3, rank(s_3,t_2)=2.

3. A time series intersects with other time series between the time point $t-1$ and t, but there is no change in the ranking of the time series.

   For example, both of the time series s_2 and s_3 Figure 3.1(b) intersects with s_1 and s_4 between the time point t_1 and t_2. However, their ranks are 2 and 3 respectively at both time points.

We can construct an inverted list for each time series to store the rank information. Each entry in the list consists of the rank of the time series at the corresponding time point. We call this structure *RankList*. There are two options for the RankList design. The first option is that we store the rank information for a time series at every time point (see Figure 3.2). The second option is that we only store the rank information for a time series at the time points at which the rank is different compared to its previous time point (see Figure 3.3). That is, an entry is only created in the inverted list of a time series when its ranking is affected by an intersection. In order to differentiate the two structures, we call the first one *RankList_original* and the second one *RankList_simplified*. Further, if an existing time series does not have any value at some time point, then it will be ranked 0 at that time point.



time	rank	time	rank	time	rank	time	rank	time	rank	time	rank
200601	1	200601	2	200601	3	200601	5	200601	6	200601	4
200602	4	200602	2	200602	3	200602	1	200602	5	200602	6
200603	3	200603	1	200603	2	200603	0	200603	5	200603	4
200604	1	200604	2	200604	3	200604	0	200604	5	200604	4
200605	1	200605	2	200605	3	200605	4	200605	5	200605	6

Figure 3.2: RankList_original constructed for student dataset in Figure 1.1 from January to May

A $\lceil k \rceil$-topband query can be quickly answered with the RankList structure by traversing the list of each time series and searching for entries with rank values greater than k. The result is the set of time series which do not have such entries in their lists.

For example, to answer a $\lceil 3 \rceil$-topband query issued over the *student* dataset in Figure 1.1, we traverse the list of stu_1 in Figure 3.3 and find that the rank in

the second entry is greater than 3. Hence, stu_1 will not be in the answer set. In contrast, there are no entries in the lists of stu_2 and stu_3 with rank values greater than 3, and $\{stu_2, stu_3\}$ are the results of the $\lceil k \rceil$-topband query. Similarly, such query can be answered by traversing the list in *RankList_original*. Note that we can stop searching a list whenever an entry in the list with rank value greater than k is encountered.

To answer k*-topband query with the *RankList_simplied* structure, we need to find the ranks of the specified time series s_1 at various time points. Then we traverse the list of each time series to look for entries with rank values greater than the rank of s_1 at the corresponding time point. The result is a set of time series which do not have such entries in their lists. Note that the entry of s_1 at some time point t may not exist because its rank at t is the same as its rank at $t-1$. In this case, we need to look for the entry with the largest time point that is smaller than t. For example, to retrieve the students who always outperform stu_6, we traverse the list of stu_2 and compare the ranks in the first two entries with the corresponding entries in the list of stu_6. When we encounter the third entry of stu_2, we find that the entry with time *200604* does not exist in the list of stu_6. In this case, we locate the second entry with time *200603* since *200603* is the largest time which is smaller than *200604* in the list of stu_6 and compare the rank in the third entry of stu_2 with the rank in that entry accordingly.

Compared to *RankList_simplied*, answering k*-topband query with the *RankList_original* is simpler. This is because *RankList_original* stores the rank information of a time series at every time point. For example, to retrieve the students who always outperform stu_6, we traverse the list of stu_2 and compare the ranks in every entry with the corresponding entry in the list of stu_6. Since stu_2 holds higher ranks than stu_6 at every entry, stu_2 is a candidate answer.

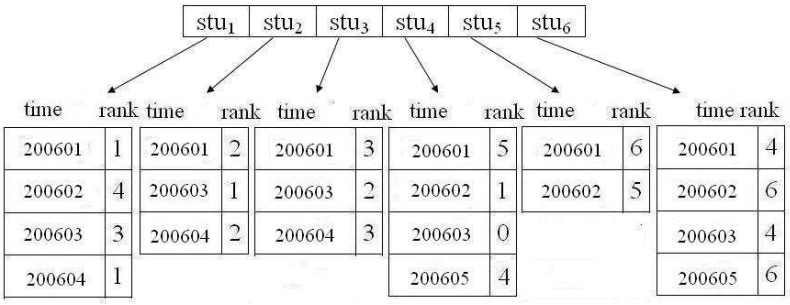


Figure 3.3: RankList_simplified constructed for student dataset in Figure 1.1 from January to May

The RankList structure can be extended to retrieve time series which are consistently at the bottom k. If we have N time series in the dataset, then we traverse the list of each time series and search for entries with rank values greater than $N - k$ or 0. The result is the set of time series which do not have such entries in their lists.

Next, we present the algorithms to construct the RankList structure as well as search and update.

3.1 RankList Construction

Algorithm 1 shows the steps to construct the inverted list structure *RankLst_simplied* that captures the rank information for each time series in a dataset. The algorithm utilizes two arrays called *PrevRank* and *CurrRank* to determine if the ranking of a time series at the current time point has been affected by some intersection.

Lines 3 and 5 initialize each entry in the *PrevRank* and *CurrRank* array to 0. This is because of the possibility of missing values for some time series. If a time series s has a value at time t, CurrRank[s] will be initialized to 1 (lines 7-8). The

Algorithm 1 BuildRankList

```
1: Input: TS - time series database with attributes id, time and value
          T - total number of time points in TS
2: Output: RankLst_s - RankList_simplied structure for TS
3: initialize int [] PrevRank to 0;
4: for each time point t from 0 to T do
5:    initialize int [] CurrRank to 0;
6:    let S be the set of tuples with time t;
7:    for each tuple p ∈ S do
8:       initialize CurrRank[p.id] to 1;
9:    for each pair of tuples p, q ∈ S do
10:      if p.value < q.value then
11:         CurrRank[p.id]++;
12:      else
13:         CurrRank[q.id]++;
14:   for each time series s in TS do
15:      if CurrRank[s] != PrevRank[s] then
16:         Create an entry <t, CurrRank[s]> for time series s in RankLst_s;
17:         PrevRank[s] = CurrRank[s];
18: return RankLst_s;
```

Algorithm 2 $\lceil k \rceil$-topband Search

```
1: Input: RankLst - RankList structure of TS;
          t_start, t_end - start and end time points;
          integer k;
2: Output: A - set of time series that are in top k over [t_start, t_end];
3: initialize A to contain all the time series in TS;
4: for each time series s in A do
5:    locate the entry <t, rank> for s in the RankLst with the largest time point
      that is less than or equal to t_start;
6:    if entry not exist then
7:       A = A - s;
8:       CONTINUE;
9:    while t ≤ t_end do
10:      if rank > k or rank = 0 then
11:         A = A - s;
12:         break;
13:      else
14:         entry = entry.next;
15: return A;
```

Algorithm 3 k*-TopbandSearch

1: **Input:** RankLst - RankList structure of TS;
   t_start, t_end - start and end time points;
   s - a specified time series;
2: **Output:** A - set of time series that outperform s over [t_start, t_end];
3: **for** each time point t from t_start to t_end **do**
4: locate the entry e for s in the RankLst with the largest time point that is less than or equal to t_start;
5: let k = e.rank;
6: let U(t) = $\lceil k \rceil$-topband(RankLst, t, t, k);
7: let A = $\bigcap U_t \ \forall$ t $\in$ [t_start, t_end];
8: return A;

algorithm scans the database once and compares the values of the time series at each time point (lines 9-13). If the ranking of a time series s changes from time point $t-1$ to t, we create an entry in the inverted list of s to record its new rank (lines 14-17).

Algorithm 1 can also be utilized to construct the *RankList_original*. The *if* condition (line 15) must be omitted since an entry needs to be created at every time point for *RankList_original*.

3.2 Topband Search

Algorithm 2 finds the set of time series that are consistently within the top k in the specified time interval. It takes as input the inverted list structure, *RankLst_simplied* or *RankList_original*, for the time series dataset, an integer k, and the start and end time points t_start, t_end. The output is S, a set of time series whose rank is always higher than k over [t_start, t_end].

S is initialized to be the set of all the time series in the dataset (line 3). The entries for each time series in the RankLst is sorted by time. For each time series s, we check if its rank is always higher than k in the specified time interval (lines

4-14). The entry with the largest time point that is less than or equal to t_start is located (line 5). If the entry does not exist or there is no value of s or the rank value of the entry is larger than k, then s is removed from S (lines 6-12). Otherwise, we continue to check the ranks of the entries for s until the end time point is reached.

Algorithm 3 finds the set of time series that outperforms a specific time series (k*-topband queries). We need to determine the value of k at each time point. This can be obtained by checking the rank of the specified time series in RankList structure (lines 4-5). Then we call Algorithm 2 using the various values of k to retrieve the desired set of time series (line 6) before computing their intersection (line 7) to get the final answer.

Alternatively, we can first obtain the ranks of the specified time series at the various time points from the RankList structure and then carry out an index scan to retrieve the set of outperforming time series. This removes the need for an intersection operation to compute the final set of answers.

3.3 RankList Updates

Insertion involves adding new values to an existing time series or adding a new time series into the dataset. The insertion of any new value may affect the rankings of existing time series. Hence, we need to compare the new value with the values of existing time series at the same time point.

3.3.1 Insertion

Algorithm 4 shows the necessages changes made to a *RankList_simplied* structure when a new value is inserted. It takes as input a tuple $<p, t, p(t)>$ to be inserted.

Algorithm 4 Insert

```
1: Input: TS - database with attributes id, time and value
          RankLst_s - RankList_simplied structure of TS;
          <p, t, p(t)> - a tuple to be inserted;
2: Output: RankLst_s - updated RankList_simplied structure for TS
3: initialize int CurrRank to 1;
4: let S be the set of tuples with time t in TS;
5: for each tuple s ∈ S do
6:   if p(t) > s.value then
7:     locate the entry e for s.id in RankLst_s which has the largest time point
       that is less than or equal to t;
8:     let PrevRank = e.rank;
9:     if e.time = t then
10:      increment e.rank by 1;
11:    else
12:      create an entry <t, PrevRank + 1> for s.id and insert into RankLst_s;
13:    locate the entry e at time t + 1 for s.id in RankLst_s;
14:    if entry does not exist then
15:      create an entry <t+1, PrevRank> for s.id and insert into RankLst_s;
16:  else
17:    CurrRank++; /* p(t) < s.value */
18: locate the entry e for p in the RankLst_s which has the largest time point
    that is less than or equal to t;
19: if e.rank ≠ CurrRank then
20:   if e.time = t then
21:     replace e.rank with CurrRank;
22:   else
23:     create an entry <t, CurrRank> for p and insert into RankLst_s;
24:   locate the entry e' at time point t + 1 for p in RankLst_s;
25:   if e' does not exist then
26:     create an entry <t + 1, 0> for p and insert into RankLst_s;
27: return RankLst_s;
```

The algorithm checks for the set of existing time series S whose values are smaller than $p(t)$ at time point t (lines 6-15). We obtain the rank of $s \in$ S from the entry which has the largest time point that is less than or equal to t and store it in the variable *PrevRank* (line 7-8). Next, we try to retrieve the entry <t, rank> for s. If the entry exists, we increase the rank by 1 (lines 9-10). Otherwise, we insert a new entry for s at t (lines 11-12).

Updating the rank of s at t may affect its rank at time $t+1$. Lines 13-15 check if an entry exists for s at $t+1$. If the entry does not exist, implying that its rank at $t+1$ follows the entry prior to t, we need to create an entry with *PrevRank* at $t+1$ and insert into RankList (lines 14-15). Finally, we update the rank for the corresponding time series p of the new value at t using *CurrRank* (lines 20-24). Note the algorithm also checks the entry for p at $t+1$ (line 25). If the entry does not exist, indicating that p does not have a value at time $t+1$ (since p does not have a value at time t), we insert an entry with rank 0 for p (lines 25-26).

The logic to update a *RankList_original* structure is simpler when a new value is inserted. All what we need to do is to obtain the set of time series S whose ranks will be affected (whose values are smaller than $p(t)$) at time point t; update the rank of the time series in S by incrementing by 1; and update the rank for the corresponding time series p. Algorithm 5 shows the details steps.

3.3.2 Deletion

Similarly, the deletion of a value from a time series dataset may affect entries in the RankList_simplied structure.

Algorithm 6 takes as input a tuple <p, t, $p(t)$> to be deleted and obtains the set of existing time series S whose values are smaller than the deleted value at time t (lines 5-20). We retrieve the rank of $s \in$ S from the entry which has the

Algorithm 5 Insert

```
1: Input: TS - database with attributes id, time and value
          RankLst_o - RankList_original structure of TS;
          <p, t, p(t)> - a tuple to be inserted;
2: Output: RankLst_o - updated RankList_original structure for TS
3: initialize int CurrRank to 1;
4: let S be the set of tuples with time t in TS;
5: for each tuple s ∈ S do
6:    if p(t) > s.value then
7:       locate the entry e for s.id in RankLst_o which has the time point that is
         equal to t;
8:       increment e.rank by 1;
9:    else
10:      CurrRank++; /* p(t) < s.value */
11: locate the entry e for p in the RankLst_o which has the time point that is
    equal to t;
12: replace e.rank with CurrRank;
13: return RankLst_o;
```

largest time point that is less than or equals to t and store it in *PrevRank* (line 6-7). Next, we try to retrieve the entry $<t$, rank$>$ for s. If the entry exists, we decrease the rank by 1 (lines 8-9). Otherwise, we insert a new entry for s at time t (lines 10-11).

Updating the rank of s at time t may affect its rank at $t + 1$. Lines 12-14 checks if an entry exists for s at $t+1$ and creates a new entry if it does not exist. Finally, we update the rank for the corresponding time series p of the deleted value at $t + 1$ (lines 17-18) and insert an entry $<t$, 0$>$ for p (line 19) to indicate the missing value of p at t.

Similarly, updating the *RankList_original* structure is simpler when a value is deleted. First, the set of time series S whose values are smaller than the deleted value at time t is retrieved. Second, updating the rank of the time series in S by decrementing by 1. Third, set the rank of p at t to 0. Algorithm 7 shows the details steps.

Algorithm 6 Delete

1: **Input:** TS - database with attributes *id*, *time* and *value*
RankLst_s - RankList_simplied structure of TS;
<p, t, $p(t)$> - a tuple to be deleted;
2: **Output:** RankLst_s - updated RankLst_simplied structure for TS
3: let S be the set of tuples with time t in TS;
4: **for** each tuple $s \in$ S **do**
5: **if** $p(t) > s$.value **then**
6: locate the entry e of $s.id$ in RankLst_s with the largest time point that is less than or equal to t;
7: let PrevRank = e.rank;
8: **if** e.time $= t$ **then**
9: decrement e.rank by 1;
10: **else**
11: create an entry <t, PrevRank-1> of $s.id$ and insert into RankLst_s;
12: locate the entry e at time point $t + 1$ for $s.id$ in RankLst_s;
13: **if** entry does not exist **then**
14: create an entry <$t + 1$, PrevRank> for $s.id$ and insert into RankLst_s;
15: locate the entry e for p in the RankLst_s with the largest time point that is less than or equal to t;
16: locate the entry e' at time $t + 1$ for p in RankLst_s;
17: **if** e' does not exist **then**
18: create an entry <$t + 1$, e.rank> for p and insert into RankLst_s;
19: create an entry <t, 0> for p and insert to RankLst_s;
20: return RankLst_s;

3.4 Time & Space Complexity

The time complexity for the various operations on the RankList structure is polynomial. Suppose we have N time series and T time points in the dataset. In the worst case, each time series will intersect with every other time series at every time point. Therefore, the time complexity to build the RankList structure is O(T^*NlogN) where ($NlogN$) is the time taken to sort the values of the time series at each time point.

The *Search* algorithm examines the list entries with time points in the specified time interval. Since the rank information of each time series at each time

Algorithm 7 Delete

```
1: Input: TS - database with attributes id, time and value
          RankLst_o - RankList_original structure of TS;
          <p, t, p(t)> - a tuple to be deleted;
2: Output: RankLst_o - updated RankLst_original structure for TS
3: let S be the set of tuples with time t in TS;
4: for each tuple s ∈ S do
5:    if p(t) > s.value then
6:       locate the entry e of s.id in RankLst_o with the time point that is equal
         to t;
7:       decrement e.rank by 1;
8: locate the entry e for p in the RankLst_o with the time point that is equal to
   t;
9: set e.rank to 0;
10: return RankLst_o;
```

point is recorded at most once, we have at most T entries in each list. Hence the time complexity for *Search* is O(N*T) in the worst case.

The *Insert* (*Delete*) algorithm compares the new value (deleted value) with every existing values at the same time point to update the ranks. In the worst case, ranks for all time series the existing values correspond to need to be updated. Hence, the time complexity for both *Insert* and *Delete* is O(N).

The space complexity for the *RankList_original* structure is O(N*T) whereas it depends on the number of intersection points for the *RankList_simplied* structure. In the worst case, every time series has a ranking which is different from its ranking at previous time point at every time point. Hence, the space complexity of the RankList_simplied structure is O(N*T). In practice, we expect the size of the *RankList_simplied* to be much smaller than the size of the original dataset. This is because for topband queries to be meaningful, the rankings of the time series should oscillate within a limited range. Further, we can apply smoothing techniques (i.e. *Haar Wavelet Transform*) to reduce the size of the *RankList_simplied* structure (see Section 5.1).

Chapter 4

Answering Topband Queries in Relational System

A time series database can be stored in a relational table R with three attributes: the id or name of the time series s, time point t, and the value of s at t. We use the triple $<s, t, s(t)>$ to denote a tuple in the relation R. For example, the time series dataset in Figure 1.1 can be stored in a relational table as shown in Table 4.1. In this chapter, we presents processing topband queries in relational database system. In the first part, we describe how existing SQL and top k methods process topband queries and highlight the drawbacks of these two approaches. In the second part, we present building RankList structure on top of relational database system, as well as searching and updating it using SQL command.

id	**time**	**mark**
stu_1	200601	92
stu_2	200601	88
stu_3	200601	84
stu_4	200601	77
stu_5	200601	72
stu_6	200601	78
stu_1	200602	79
stu_2	200602	89
stu_3	200602	86
stu_4	200602	94
stu_5	200602	76
stu_6	200602	73
stu_1	200603	80
stu_2	200603	91
stu_3	200603	84
stu_5	200603	70
stu_6	200603	76
stu_1	200604	96
stu_2	200604	90
stu_3	200604	80
stu_5	200604	73
stu_6	200604	75
stu_1	200605	94
stu_2	200605	91
stu_3	200605	83
stu_4	200605	78
stu_5	200605	75
stu_6	200605	72

Table 4.1: Example student relation

4.1 Answering Topband Queries with Existing Methods

A $\lceil k \rceil$-topband query can be mapped to an SQL query which requires a nested loop. For example, a $\lceil 3 \rceil$-topband query to retrieve students with consistent performance for the period January to May is equivalent to the following standard SQL query:

```
SELECT c.id FROM student c
WHERE c.time ≥ 200601 and c.time ≤ 200605
  and ( SELECT count(c1.id) FROM student c1
        WHERE c1.time = c.time
          and c1.id <> c.id
          and c1.mark > c.mark ) < 3
GROUP BY c.id
HAVING count(c.time) = 200605 - 200601 + 1
```

The general approach for evaluating this SQL query is:

1. For each tuple $<s, t, s(t)>$ in relation R, we retrieve the set of tuples at time point t. If the number of tuples whose values are larger than $s(t)$ is less than k, then the tuple $<s, t, s(t)>$ is a top k result at t and is stored in an intermediate relation. This is a nested loop which is expensive and cannot be removed.

2. Group the tuples in the intermediate relation according to their time series id. If the number of tuples in each group s is the same as the number of time points in the specified time interval, then s is an answer to the $\lceil k \rceil$-topband query.

We can utilize an early pruning strategy to optimize the above SQL query. A tuple $<s, t, s(t)>$ can be skipped if there exists another tuple with id s and is not ranked top k previously. However, the improved SQL query still requires nested loops to compute the $\lceil k \rceil$-topband result. Experiment results in Chapter 5 reveal that even the improved SQL approach remains expensive.

Next, we examine how we can leverage the top-k operator proposed in [7] to answer $\lceil k \rceil$-topband queries. This involves mapping the top-k query at each time point to a range query. The search distance can be estimated using any of the methods in [7, 12, 15]. The basic framework is as follows:

1. For each time point t in the specified time interval, estimate the search distance *dist* such that it encompasses at least k tuples with values greater than *dist*. Note that the search distance could vary for the different time points.

2. Use the estimated distances to retrieve the set of top k tuples at each time point, and compute the intersection.

For instance, our example query to retrieve the students who are consistently within top 3 for the period January to May is equivalent to the following range query. The function *distance(c.time)* in the range query denotes the estimated search distance at the time point *c.time*.

```
SELECT c.id from student c
WHERE c.time ≥ 200601 and c.time ≤ 200605
  and c.mark > distance(c.time)
GROUP BY c.id
HAVING count(c.time) = 200605 - 200601 + 1
```

There are two drawbacks to this approach. First, the intermediate relation to store the top k results at each time point is proportional to k and the number of time points. Second, many of these computations are wasted since the final result will not have more than k tuples.

Finally, we discuss how k*-topband queries can be answered using SQL. Given that a time series s will be specified in such queries, we can use the values of s at the various time points to retrieve the set of time series that have larger values than s. For example, the SQL query to retrieve the students that outperform stu_6 for the period Jan to May is:

```
SELECT c.id FROM student c
WHERE c.time ≥ 200601 and c.time ≤ 200605
   and c.mark > ( SELECT c1.mark FROM student c1
                  WHERE c1.time = c.time
                    and c1.id = 'stu6')
GROUP BY c.id
HAVING count(c.time) = 200605 - 200601 + 1
```

Again, many of the computations in the above SQL query for k*-topband could be wasted.

4.2 RankList Implementation

The proposed RankList structure can be easily built on top of a relational database. We define a relation called *RankTable* which consists of three attributes: time series *id*, time point *time*, and the rank of the time series at time point *rank*. The key of the relation is {*id, time*}. This relation can be subsequently indexed by the B+-tree for fast access.

4.2.1 RankList_original Implementation

id	time	rank
stu_1	200601	1
stu_1	200602	4
stu_1	200603	3
stu_1	200604	1
stu_1	200605	1
stu_2	200601	2
stu_2	200602	2
stu_2	200603	1
stu_2	200604	2
stu_2	200605	2
stu_3	200601	3
stu_3	200602	3
stu_3	200603	2
stu_3	200604	3
stu_3	200605	3
stu_4	200601	5
stu_4	200602	1
stu_4	200603	0
stu_4	200604	0
stu_4	200605	4
stu_5	200601	6
stu_5	200602	5
stu_5	200603	5
stu_5	200604	5
stu_5	200605	5
stu_6	200601	4
stu_6	200602	6
stu_6	200603	4
stu_6	200604	4
stu_6	200605	6

Table 4.2: Example *RankTable_original* relation

The inverted list structure in Figure 3.2 can be mapped to the *RankTable_original* relation in Table 4.2. $\lceil k \rceil$-topband queries can now be answered by issuing SQL queries over the relation *RankTable_original* to retrieve time series whose rank is higher than k. Our running example query to retrieve the students who are consistently within the top 3 for the period Jan to May is equivalent to the following SQL query:

```
(S1) SELECT c.id FROM RankTable_original c
     WHERE c.time = 200601
       and NOT EXISTS ( SELECT c1.id
                        FROM RankTable_original c1
                        WHERE c1.id = c.id
                          and c1.time ≥ 200601 and c1.time ≤ 200605
                          and ( c1.rank = 0 OR c1.rank > 3 ) )
```

The condition *[c.time = 200601]* in the above SQL query S_1 guarantees that the subquery is executed only once for each time series.

k*-topband queries require a subquery to retrieve the values of k since k is dependent on the rank of the specified time series at various time points. For example, the following SQL query retrieves the students that outperform stu_6 for the period Jan to May:

```
(S2) SELECT c.id FROM RankTable_original c
     WHERE c.time = 200601
       and NOT EXISTS
         ( SELECT c1.id
           FROM RankTable_original c1
           WHERE c1.id = c.id
             and c1.time ≥ 200601
             and c1.time ≤ 200605
             and c1.rank > ( SELECT c2.rank
                             FROM RankTable_original c2
                             WHERE c2.id = 'stu6'
                               and c2.time = c1.time ) )
```

The RankTable_original relation needs to be updated when a new value is inserted into the dataset. This can be accomplished by issuing a set of SQL statements as shown in statements $S_3 - S_5$ when the tuple <stu_4, 200603, 78> is inserted into the time series relation:

(S_3) CREATE VIEW V(id, time, rank)
AS (SELECT c.id, c.time, c.rank
FROM RankTable_original c
WHERE EXISTS (SELECT * from student
WHERE id = c.id
and time = 200603
and mark < 78))

(S_4) UPDATE RankTable_original SET rank = rank + 1
WHERE id IN (SELECT id from V)
and time = 200603

(S_5) UPDATE RankTable_original SET rank = (SELECT count(*))
WHERE id = 'stu_4'
and time = 200603

The statement S_3 creates a view V to store the set of time series whose ranks are affected by the addition of the new value. It contains entries that has the time points which equal to the time point of the new value. Statement Statement S_4 is used to update the ranks of the time series at time point t which are affected by the new value. Finally S_5 updates the rank information of the time series with the newly inserted value.

Similar SQL statements can be issued to update the RankTable_original relation when a value is deleted from the dataset.

4.2.2 RankList_simplified Implementation

id	time	rank
stu_1	200601	1
stu_1	200602	4
stu_1	200603	3
stu_1	200604	1
stu_2	200601	2
stu_2	200603	1
stu_2	200604	2
stu_3	200601	3
stu_3	200603	2
stu_3	200604	3
stu_4	200601	5
stu_4	200602	1
stu_4	200603	0
stu_4	200605	4
stu_5	200601	6
stu_5	200602	5
stu_6	200601	4
stu_6	200602	6
stu_6	200603	4
stu_6	200605	6

Table 4.3: Example *RankTable_simplified* relation

The inverted list structure in Figure 3.3 can be mapped to the *RankTable_simplified* relation in Table 4.3. $\lceil k \rceil$-topband queries can now be answered by issuing SQL queries over the relation *RankTable_simplified* to retrieve time series whose rank is higher than k. Our running example query to retrieve the students who are consistently within the top 3 for the period Jan to May is equivalent to the following SQL query:

```
(S6) SELECT c.id FROM RankTable_simplied c
     WHERE c.time = 200601
       and NOT EXISTS ( SELECT c1.id
                        FROM RankTable_simplified c1
                        WHERE c1.id = c.id
```

```
                and c1.time ≥ 200601 and c1.time ≤ 200605
                and ( c1.rank = 0 OR c1.rank > 3 ) )
```

The condition *[c.time = 200601]* in the above SQL query S_6 guarantees that the subquery is executed only once for each time series.

k*-topband queries require a subquery to retrieve the values of k since k is dependent on the rank of the specified time series at various time points. For example, the following SQL query retrieves the students that outperform stu_6 for the period Jan to May:

```
SELECT c.id FROM RankTable_simplified c
WHERE c.time = 200601
  and NOT EXISTS
   ( SELECT c1.id
     FROM RankTable_simplified c1
     WHERE c1.id = c.id
       and c1.time ≥ 200601
       and c1.time ≤ 200605
       and c1.rank > ( SELECT c2.rank
                       FROM RankTable_simplified c2
                       WHERE c2.id = 'stu6'
                         and c2.time = ( SELECT max(time)
                                         FROM RankTable_simplified
                                         WHERE id = c2.id
                                           and time < c1.time) ) )
```

The RankTable_simplified relation needs to be updated when a new value is inserted into the dataset. This can be accomplished by issuing a set of SQL

statements as shown in statements $S_7 - S_{10}$ when the tuple <stu_4, 200603, 78> is inserted into the time series relation:

(S_7) CREATE VIEW V(id, time, rank)
AS (SELECT c.id, c.time, c.rank
FROM RankTable_simplified c
WHERE EXISTS (SELECT * from student
WHERE id = c.id
and time = 200603
and mark < 78)
and c.time = (SELECT max(c1.time)
FROM RankTable_simplified c1
WHERE c1.id = c.id
and c1.time ≤ 200603))

(S_8) INSERT INTO RankTable_simplified (id, time, rank)
SELECT p.id, 200603+1, p.value from V p
WHERE p.id NOT IN
(SELECT c.id
FROM RankTable_simplified c
WHERE c.id = p.id
and c.time = 200603+1)

(S_9) INSERT INTO RankTable_simplified (id, time, rank)
SELECT p.id, 200603, p.value + 1 FROM V p
WHERE p.id NOT IN
(SELECT c.id
FROM RankTable_simplified c
WHERE c.id = p.id
and c.time = 200603)

(S_{10}) UPDATE RankTable_simplified SET rank = rank + 1

WHERE id IN (SELECT id from V)

and time = 200603

The statement S_7 creates a view V to store the set of time series whose ranks are affected by the addition of the new value. It contains entries that has the largest time points less than or equal to the time point of the new value. Statement S_8 inserts entries at time point $t + 1$ for the set of time series whose ranks are affected at time point t. Statements S_9 and S_{10} are used to update the ranks of the time series at time point t which are affected by the new value.

Similar SQL statements can be issued to update the RankTable_simplified relation when a value is deleted from the dataset.

Compared to *RankList_original*, *RankList_simplified* is more complicated to implement when a new value is added/deleted. However, the tradeoff is that *RankList_simplified* has less entries compared to *RankList_original* which is more efficient when answering topband queries. We will show the details experiment result in the next section.

Chapter 5

Performance Study

In this chapter, we present the results of three sets of experiments to evaluate the efficiency and scalability of the proposed method. We implement the algorithms in Chapter 3 in Java (JDK version 1.5.01). The synthetic data generator produces time series datasets with attributes *id*, *time* and *value*. Table 5.1 shows the range of values for the various parameters and their default values.

Parameter	**Range**	**Default**
Number of time series N	[100, 500]	100
Number of time points T	[5000, 20000]	10000
k	[50, 250]	50
Length of query interval L	[5000, 10000]	10000
Percentage of intersection points	[2%, 10%]	5%

Table 5.1: Parameters of dataset generator

The first set of experiments examines and compares the time taken to construct, search and update the *RankList_original* and *RankList_simplified* structures, as well as the space requirements. We also investigate how the time and space requirements are affected by applying the **Haar Wavelet Transform** [11] technique, which is conceptually simple, fast and memory efficient.

In the second set of experiments, we map the RankList structure to a relational

table called RankTable, and compare the proposed method (*Rank_original* & *Rank_simplified*) with the SQL approach (*Nested*) and the top-k range query method (*Top-k*).

The third set of experiments evaluates the effectiveness of topband queries on two real-life datasets, *stock* and *student*, as well as the tradeoff of the smoothing method.

All the experiments are carried out on a 2.58GHz Pentium 4 PC with 1.00 GB RAM, running WinXP. Each experiment is repeated 5 times, and the average time taken is recorded.

5.1 Experiments on RankList Structure

We first carry out a set of experiments on the synthetic dataset to examine how the number of intersection points affects the *RankList_original* and *RankList_simplified* structures. We set the number of time series N to 100 and the number of time points T to 10000, giving us a total of 1 million data points. We vary the number of intersection points from 10% to 50% of the total possible number of intersection points.

We also use the Haar Wavelet Transform to smooth the dataset. The Haar transform allows a time series to be viewed in multiple resolutions through a series of averaging and differencing operations. For example, the values {9, 7, 3, 5} can be transformed as follows:

Resolution	Averages	Coefficients
4	{9, 7, 3, 5}	
2	{8, 4}	{1, -1}
1	{6}	{2}

The two values {8, 4} at resolution 2 are obtained by taking the average of the first two numbers {9, 7} and the last two numbers {3, 5} at resolution 4 respectively. The two numbers {1, -1} in the coefficients part of resolution 2 are the differences of {9, 7} and {3, 5} divided by two respectively. This process continues until a resolution of 1 is reached. The Haar transform returns $(6, 2, 1, -1)$ which is composed of the last average value 6 and the coefficients on the rightmost column (2, 1, and -1).

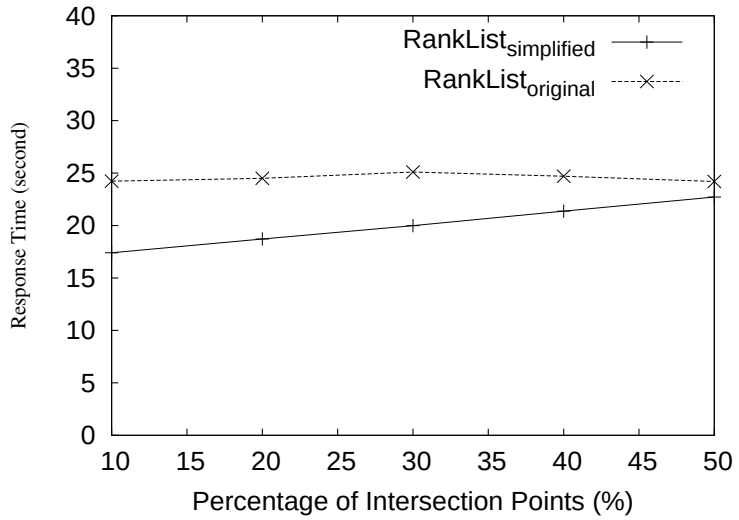


Figure 5.1: Time to construct RankList_simplified vs. RankList_original

Different degrees of smoothing can be achieved by limiting the size of the Haar transform. A parameter *threshold* is used to indicate the size of the Haar transform. For example, if we fix the size of the Haar transform to be 2 (threshold = 2/4 = 0.5), then the resulting time series is reconstructed as (6+2, 6+2, 6-2,

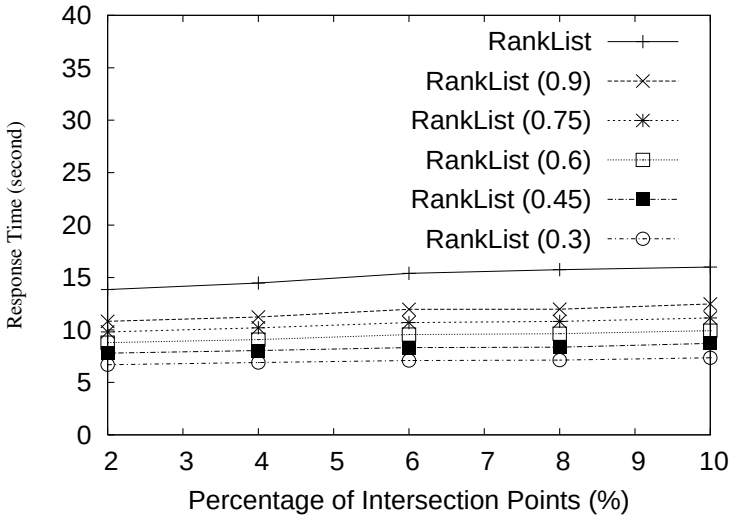


Figure 5.2: Time to construct RankList_simplified

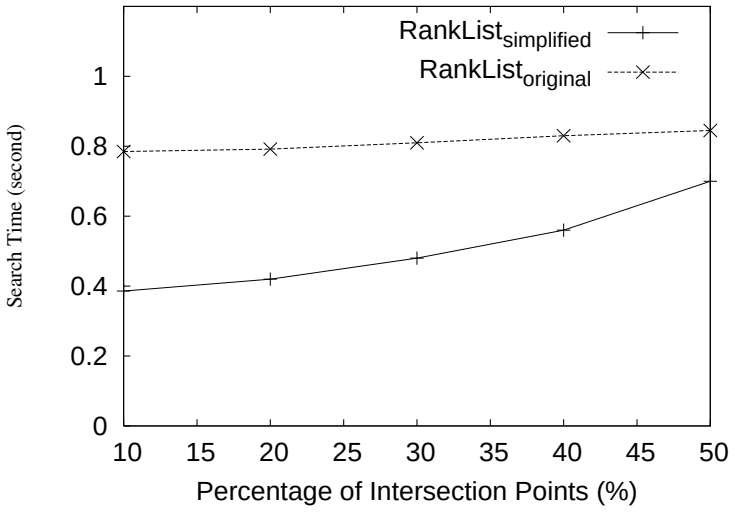


Figure 5.3: Time to search RankList_simplified vs. RankList_original

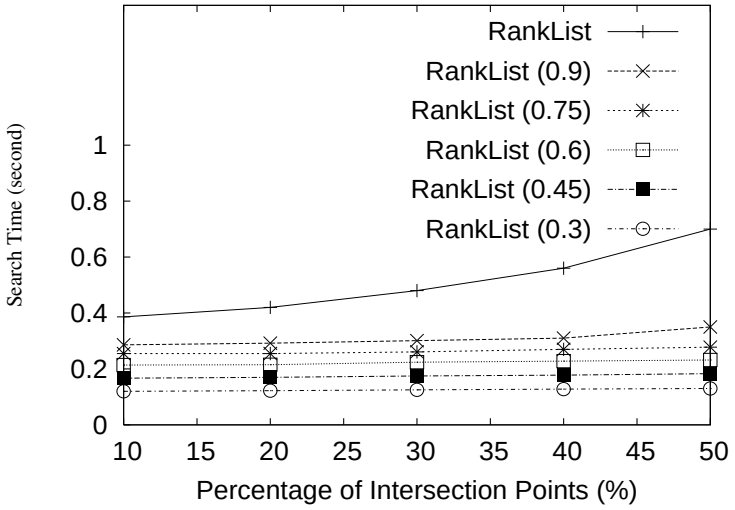


Figure 5.4: Time to search RankList_simplified

6-2) = (8, 8, 4, 4).

Figure 5.1 shows the total time taken to build the *RankList_simplified* and *RankList_original* structures by varying the number of intersection points. The time used to build the *RankList_simplified* structure increases as the number of intersection points increases, whereas, it is almost constant to build *RankList_original* structure. Furthermore, as the number of intersection points increases, the time cost to build *RankList_simplified* structure goes more close to the time cost to build *RankList_original* structure. This is expected as *RankList_simplified* records rank information changes. As discussed in the beginning of Chapter 3, more number of intersections will result in more changes in rank information. Therefore, more entries are created to record changes and more time cost to build the *RankList_simplfied* structure. *RankList_original* structure records the rank information at every time point no matter there is a change or not. Therefore, the number of entries does not depend on the number of intersection points. Thus, the time cost is almost constant to build the *RankList_original* structure.

Figure 5.2 plots the total time taken to build the *RankList_simplified* structure as the smoothing threshold, indicated in brackets, varies from 0.3 to 0.9. In this experiment, we focus on studying the smoothing effect on the *RankList_simplified* structure. This is because smoothing technique aims to reduce the number of intersection points. However, the number of intersection points has little effect on the *RankList_original* structure (as shown in Figure 5.1). Figure 5.2 indicates that as the smoothing threshold decreases, the time to construct the *RankList_simplified* structure decreases despite the increase in the percentage of intersection points.

Next, we examine the time cost to search the RankList structures. Figure 5.3 shows that as the number of intersection points increases, the time cost to search the *RankList_simplified* structure increases whereas it is almost constant to search the *RankList_original* structure. This is because as the number of intersection points increases, more entries are created in *RankList_simplified* structure, therefore, more entries need to be searched. For *RankList_original* structure, there is no increase in the number of entries, therefore, the search time is almost constant. Again, searching the *RankList_simplified* structure takes less time than searching *RankList_simplified* structure, since the previous one has less entries to search.

Figure 5.4 shows that the time taken to search the *RankList_simplified* decreases as the smoothing threshold value decreases. This is expected since the smoothing step will reduce the number of intersection points.

We also examine the cost to update the *RankList_simplified* and *RankList_original* structures by varying the number of insertions/deletions from 20 to 100. Figure 5.5 shows that the update time increases linearly with the number of insertions/deletions for both structures. *RankList_original* takes less time to update than *RankList_simplified*. This is because when updating, *RankList_original* looks

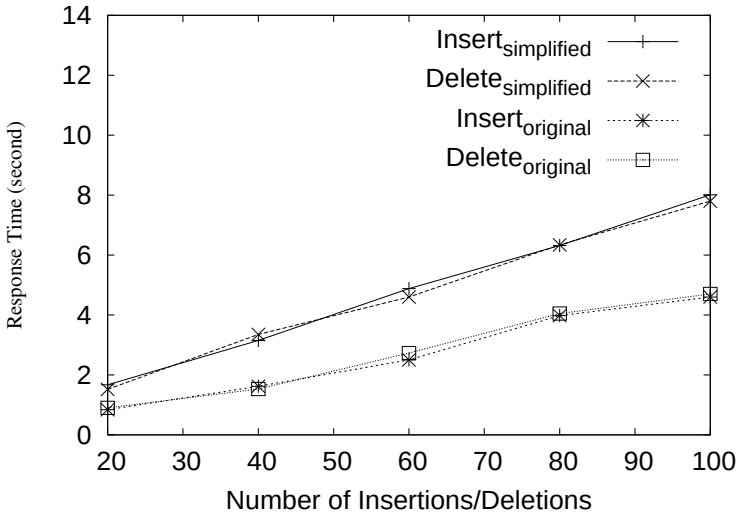


Figure 5.5: Update cost

at one specific entry whereas *RankList_simplified* needs to take care of the rank information at both the specific time point and the time point after that.

Figure 5.6 gives the space requirements of the *RankList_simplified* and *RankList_original* structures. As expected, the space for *RankList_simplified* increases as the number of intersection points increases. Using Haar Transform to smooth the time series also has an effect on the number of entries in the *RankList_simplified*. We see that as the smoothing threshold decreases from 0.9 to 0.3, the space required by the *RankList_simplified* structure decreases despite the increase in the percentage of intersection points. Furthermore, the space cost for the *RankList_original* is constant as the number of intersection points varies. Again, *RankList_simplified* takes less space compared to *RankList_original*.

In conclusion, *RankList_simplified* structure has advantage in constructing, searching and space requirement compared to *RankList_original*, whereas *RankList_original* is very efficient when updating. As time series data is mainly historical data, that is, it is seldom to insert/delete a value for the past period,

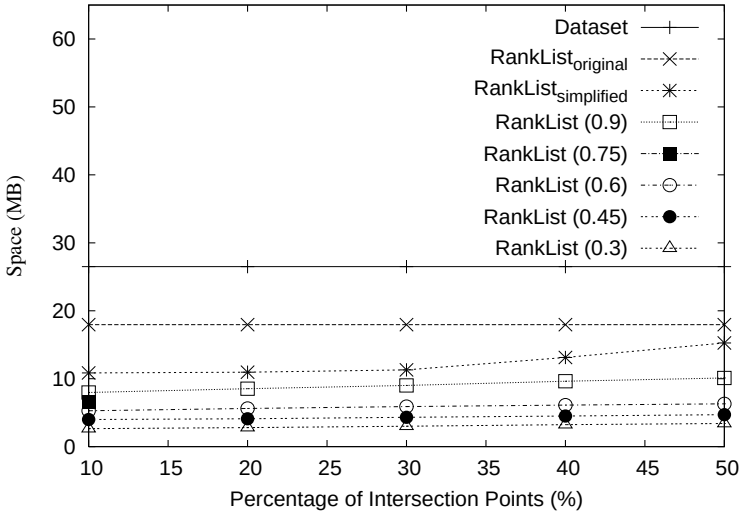


Figure 5.6: Space requirement of RankList

RankList_simplified structure is a good choice when answering topband queries.

5.2 Experiments on Topband Queries

In this section, we present the results of the second set of experiments that compare the proposed *Rank_original* and *Rank_simplified* methods (rank methods for short) with the nested SQL and top-k methods to answer topband queries. We use Oracle9i as the underlying relational database system to store the dataset and create an index on the attribute *time*.

We map the *RankList_original* and *RankList_simplified* structures to relational tables called *Rank_original* and *Rank_simplified* respectively, and issue SQL queries on these relations as described in Section 4.2. The $\lceil k \rceil$-topband queries used for the nested method and the top-k method are similar to the nested SQL query and the top-k query described in Section 4.1. We use the method in [7] to estimate the distance for the range query at various time points in top-k query

approach.

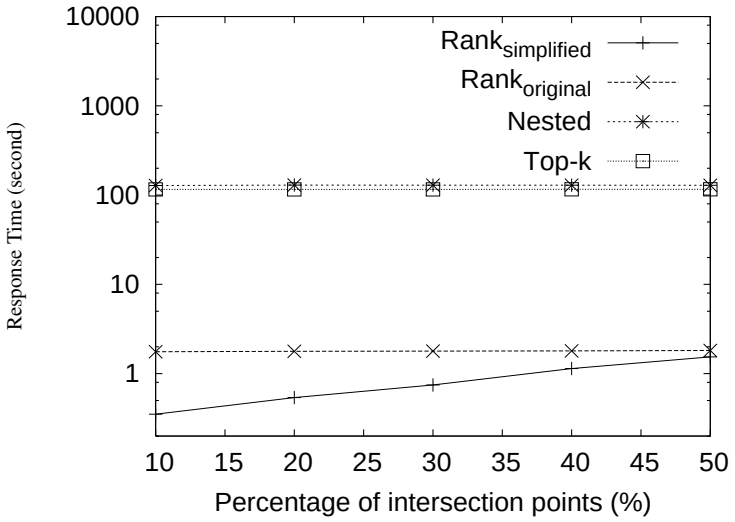


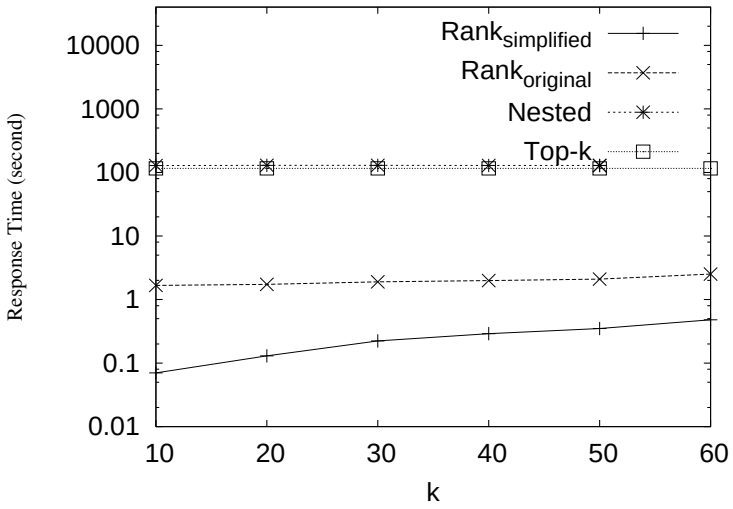
Figure 5.7: Effect of number of intersection points with the response time in log scale

5.2.1 Effect of Number of Intersection Points

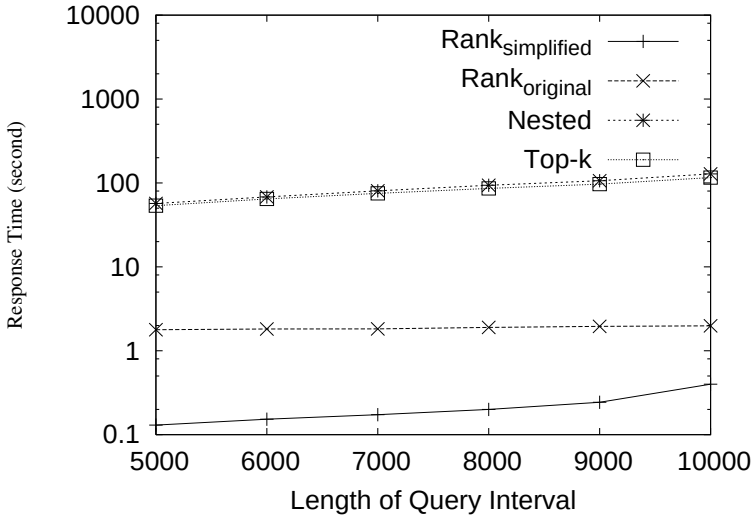
We first study the effect of the number of intersection points. Figure 5.7 shows the time taken by the three methods in log scale. We observe that the runtimes of the *Rank_original*, nested SQL and the top-k approaches are not affected by the increase in the number of intersection points. This is expected since *Rank_original* needs to search the same number of entries regardless of the number of intersection points, whereas the last two approaches require a complete database scan regardless of the number of intersection points. In contrast, the time taken by the *Rank_simplfied* approach is a small fraction of that required by the SQL and top-k methods and it takes less time than *Rank_original* method. This is expected as less tuples are searched by this method.

5.2.2 Effect of Query Selectivity

The selectivity of $\lceil k \rceil$-topband queries is determined by the value of k and the length of the query interval.



(a) varying k



(b) varying length of query interval

Figure 5.8: Effect of query selectivity with the response time in log scale

We first study the performance of the four approaches by varying the value of k in the queries. Figure 5.8(a) shows that the nested approach and top-k

approach are not affected by k. This is because the number of tuples processed is the same regardless of the value of k. In contrast, the time taken by the proposed *Rank_original* and *Rank_simplified* approaches increase as k increases. This is because as k becomes larger, more tuples need to be processed to retrieve results.

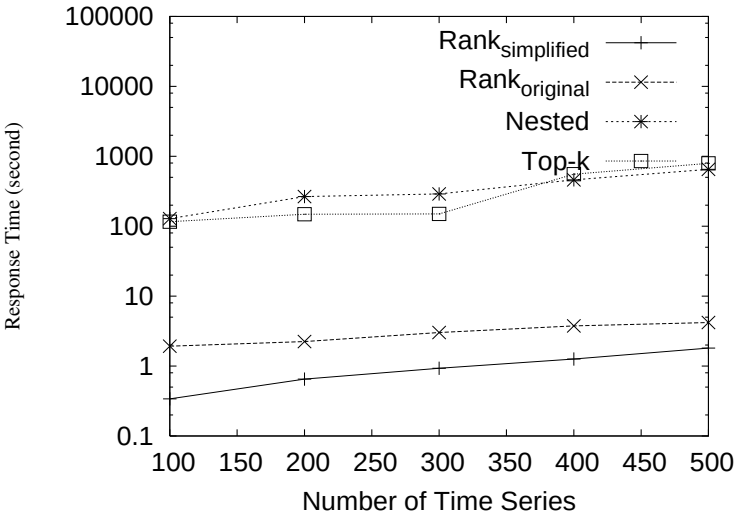
Next, we vary the length of the query interval. The result is shown in Figure 5.8(b). We observe that the runtime of all four approaches increase as the length of the query interval increases. This is expected as more tuples are processed. However, the *Rank_original* and *Rank_simplified* approaches outperform the nested and the top-k approaches by a large margin. *Rank_simplified* further outperforms *Rank_original* method as less tuples are processed.

5.2.3 Scalability

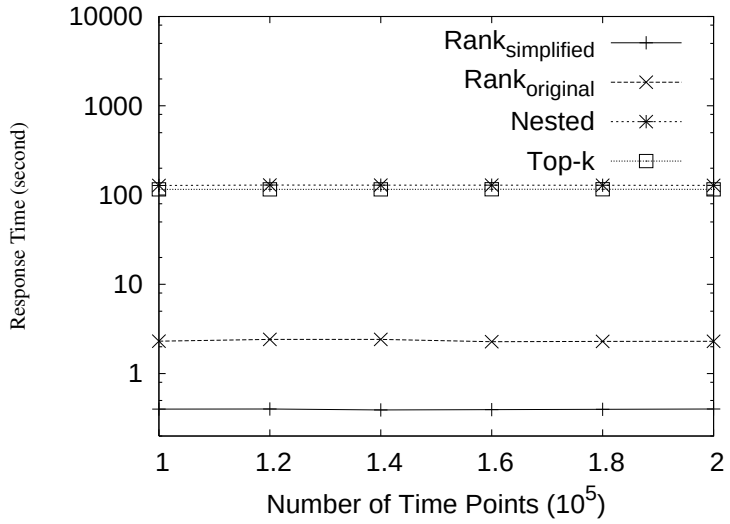
The size of a time series dataset is determined by the number of time points and the number of time series in the database. We first investigate the effect of varying number of time series in the dataset. Figure 5.9(a) shows the time taken by all the four approaches to process $\lceil k \rceil$-topband queries where k=50.

We observe that the runtime increases with the number of time series. The *Rank_simplified* and *Rank_original* approaches outperform the nested and top-k approach by a factor of 1000 and 100 respectively. The poor performance of nested and top-k approach is due to many wasted computations and large intermediate results at every time point. Note that when the number of time series exceeds 400, the top-k approach performs even worse than the nested approach. This is mainly due to the time needed to compute the search distances for each time point.

Next, we vary the number of time points in the dataset. The length of the



(a) varying number of time series



(b) varying number of time points

Figure 5.9: Scalability with the response time in log scale

query interval is fixed at 10,000. Figure 5.9(b) shows that the time taken by all four approaches are independent of the underlying dataset size. Clearly, the proposed approach is efficient and scalable.

5.2.4 Experiments on k*-topband queries

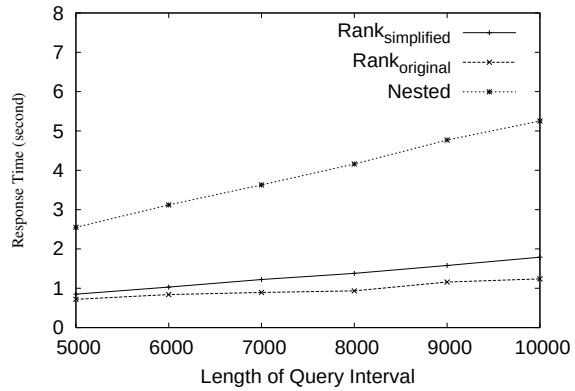
We also carry out a similar set of experiments to evaluate the cost of processing k*-topband queries. Similar trends and effects are observed as the various parameters are varied (see Fig. 5.10). The nested SQL approach takes a shorter time to process k*-topband queries compared to processing k*-topband queries because aggregation is not needed in the subquery. Furthermore,the proposed *Rank_simplified* method takes longer time than the *Rank_original* approach to process k*-topband queries because we need to compute the rank of the specified time series at each time point. Overall, the time taken by the rank approach is still less than the nested approach by a large factor.

Overall, the rank approach outperforms the nested SQL and the top-k approaches. This is due because the rank method is able to prune the non-promising time series the moment their rank falls below k. This allows the elimination of a large number of time series as more time points are processed. In contrast, the top-k approach is unable to perform such elimination.

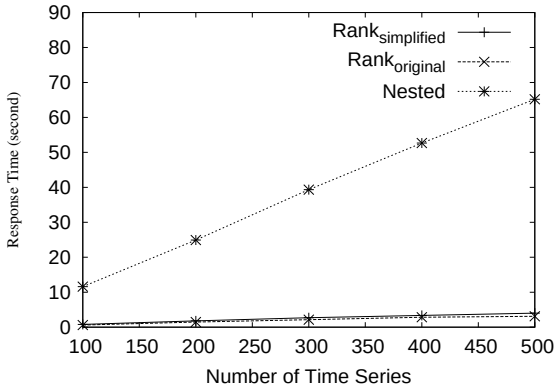
5.3 Experiments on Real World Datasets

In this final set of experiments, we demonstrate how topband queries are useful in two real world scenarios.

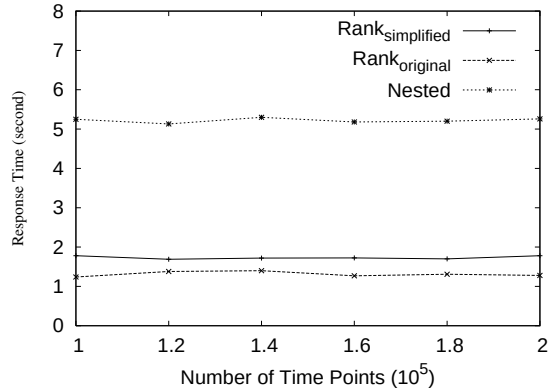
Stock Dataset. In Chapter 1, we introduce a less restrictive version of topband



(a) varying length of query interval



(b) varying number of time series



(c) varying number of time points

Figure 5.10: Experiments on k*-topband queries

query by applying smoothing technique. In this set of experiment, We first examine the effect of smoothing on a real world *stock* dataset [27] and its tradeoff on precision and recall. The stock dataset records the daily prices for 408 stocks from 1995 to 2003. We retrieve the opening and closing prices for each stock and compute their gains for each day. The percentage of number of intersection is 30.4% for this dataset.

We first examine the search time as the smoothing threshold varies from 0.3 to 1. Figure 5.11 shows that the time taken to search the *RankList_simplified* increases as the threshold increases. This is expected as the number of entries will increase when increasing threshold value. It takes less time to search *RankList_simplified* than searching *RankList_original* since less entries in *RankList_simplified*.

Next we examine the space requirement as well as the the tradeoff of the smoothing method. We define the precision and recall of topband queries as follows:

$$precision = \frac{n_t}{n_t + n_f} \times 100\% \tag{5.1}$$

$$recall = \frac{n_t}{n_t + n_m} \times 100\% \tag{5.2}$$

where n_t is the number of time series that are correctly retrieved for a given topband query, n_f is the number of time series which are wrongly retrieved, and n_m is the number of time series which should have been retrieved but are not.

We vary the smoothing threshold from 0.3 to 1 and record the recall and precision of the topband query as well as the space requirement of the *RankList_simplified* and *RankList_original* structures. As expected, the space requirements for the

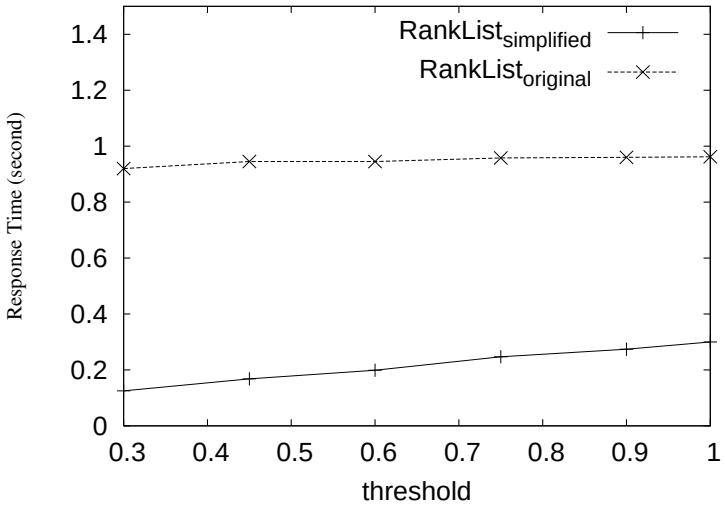


Figure 5.11: Time to search RankList for *stock* dataset

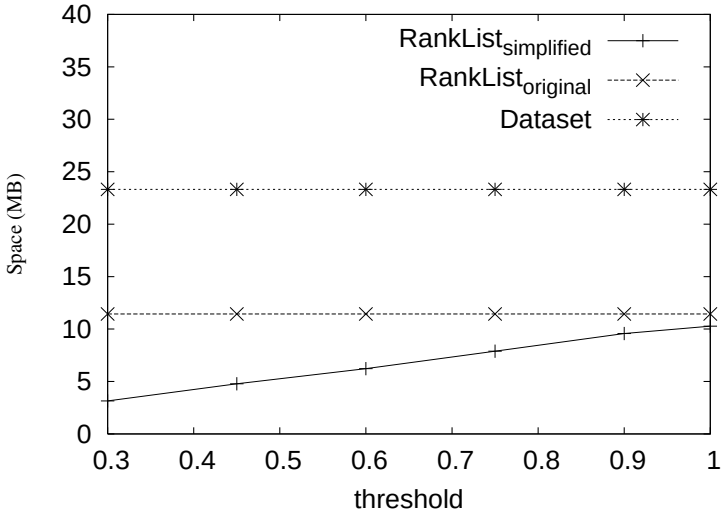


Figure 5.12: Space of RankList for *stock* dataset

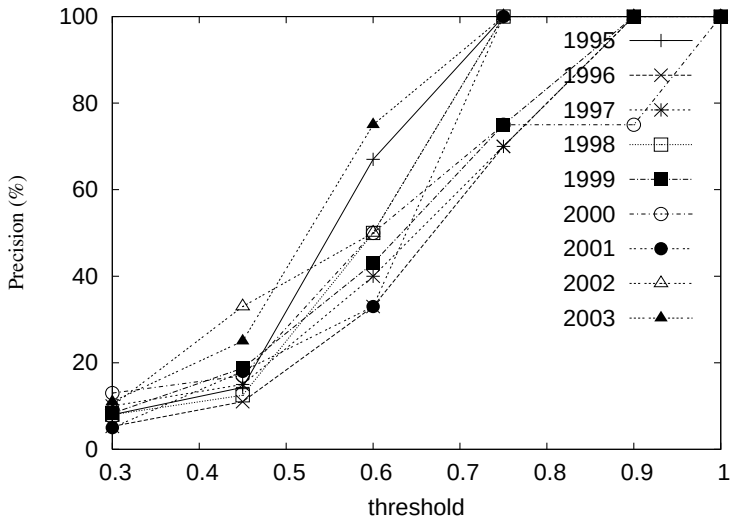


Figure 5.13: Precision vs. smoothing threshold for the *stock* dataset

RankList_simplified structure decreases as the threshold is decreased (see Figure 5.12). The recall is 100% for all values of the smoothing threshold. Figure 5.13 shows that the precision decreases as the threshold is decreased. Furthermore, when the threshold decreases beyond 0.6, the loss of precision accelerates. At this threshold of 0.6, a space reduction of 73.3% is obtained.

Next, we compute the average gains of all the stocks over the same period and issue a top-k query to retrieve a set of k stocks, where k varies from 10 to 50. We also issue topband queries to retrieve a second set of k stocks over the period 1997 - 2000. We compare the performance of these two sets of stocks over the period of 2001 - 2003 by computing their gains for each year as well as their total gains over the three years. Table 5.2 shows that the set of stocks retrieved by topband queries consistently attain higher gains than the stocks retrieved by top-k queries. This further strengthens our confidence in the ability of topband queries to identify the potential merits of a portfolio of stocks.

k	**2001**	**2002**	**2003**	**Total**
10	8.3%	16.4%	34.8%	18%
20	6.76%	9.8%	17.97%	10.6%
30	7.5%	9.6%	6.75%	8.2%
40	10.9%	11.9%	13.3%	11.8%
50	5.2%	4%	7.2%	5.1%

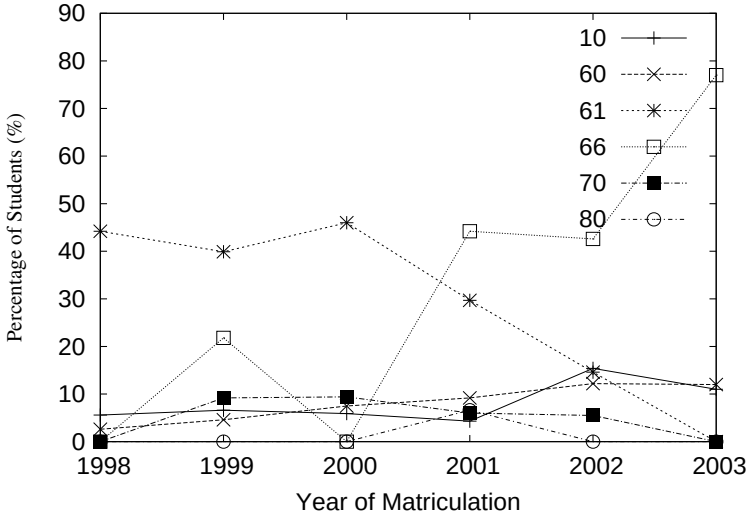
Table 5.2: Percentage gains of stocks retrieved by topband over top-k queries.

Student Dataset. We obtain a 8-year student dataset from our department. This dataset has 3800 records that capture the students' GPA for each semester. Other attribute in the records include studentID, gender, entrance exam code and year of enrolment.

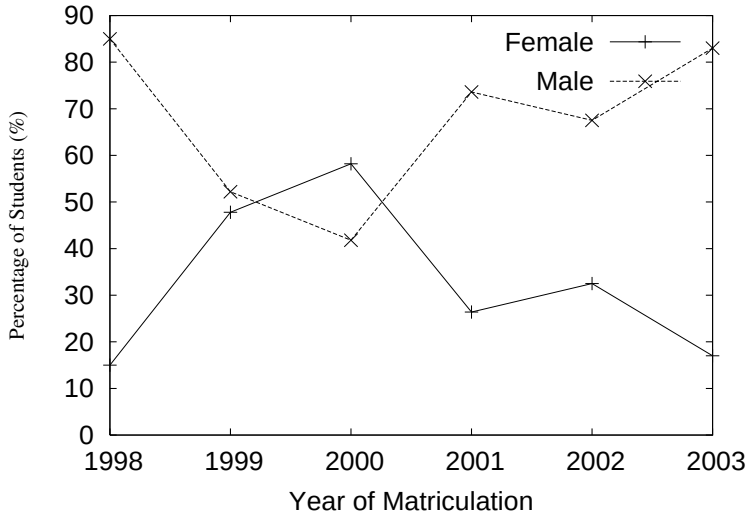
We group the students according to their year of enrolment and issue $\lceil k \rceil$-topband queries to retrieve the top 20% students that show consistent performance throughout their undergraduate studies. Figure 5.14(a) shows the percentage of consistent performers grouped by *entrance exam code* (after normalization). The entrance exam code gives an indication of the education background of the students.

We observe that from 1998 to 2000, the majority of the consistent performers (top 20%) are students with an entrance code 61. However, from 2001 onwards, the top consistent performers shifted to students with entrance code 66. An investigation reveals that due to the change in the educational policy, the admission criteria for students with entrance code 61 has been relaxed, leading to a decline in the quality of this group of students. This trend has been confirmed by the data owner. Students with entrance code of 66 have in the past not been ranked highly. The sudden increase in quality of this group of students is unexpected and has motivated the user to gather more information to explain this phenomena.

Figure 5.14(b) shows the consistent performers grouped by gender. We ob-



(a) group by entrance exam code



(b) group by gender

Figure 5.14: Top 20% students for each batch

serve that there is no specific trend separating the male and female consistent performers. This result dispels the commonly held belief that females do not perform well in computer science subjects. Publishing this statistics will certainly encourage females to apply to engineering/computer science faculties.

Chapter 6

Conclusion and Future Work

Motivated by the need to answer top k (or bottom k) queries over a period of time, we have introduced a new class of topband queries for time series dataset. The $\lceil k \rceil$-topband query retrieves the set of time series that is always within top k for all time points in the specified time interval T. We have also discussed a less restricted version of $\lceil k \rceil$-topband query, that is, k*-topband, which retrieves a set of time series which always outperforms a particular time series over some time interval. Depending on the application, the formulation of topband queries can be relaxed to retrieve a set of time series which are within top k in at least T' time points.

We have examined how topband queries can be answered using standard nested SQL approach as well as top-k methods. In order to evaluate topband queries efficiently, we have designed a structure to capture the rank information of the time series data. The proposed structure can be easily implemented on any relational database system. The results of extensive experiments on both synthetic and real world datasets indicate that the proposed approach is efficient and scalable, and it outperforms the SQL and Top-k methods by a large magin.

We have also demonstrated that topband queries can be applied directly to select a portfolio of stocks that have potential as well as identify shifts in student quality.

Research into topband queries is quite new. To the best of our knowledge, this is the first piece of work to address this problem. While the rank based method is effective is improving the performane of topband queries, the trade-off lies in the space complexity. In the worst case, where every time series intersects with each other at every time point, the space required will increase dramatically. We have shown that applying smoothing techniques such as the Haar Wavelet Transform is able to reduce the space requirement. Future work could include the incorporation of statistics information and error tolerance in order to accelerate the overall processing time and reduce space complexity.

Bibliography

[1] R. Agrawal, C. Faloutsos, and A. Swami. Efficient similarity search in sequence databases. *Int. Conference on Foundations of Data Organization and Algorithms*, pages 69–84, 1993.

[2] R. Agrawal, G. Psaila, E. L. Wimmers, and M. Zait. Querying shapes of histories. *VLDB*, pages 502–514, 1995.

[3] H. Andre-Jonsson and D. Z. Badal. Using signature files for querying time-series data. *PKDD*, pages 211–220, 1997.

[4] D. Berndt and J. Clifford. Using dynamic time warping to find patterns in time series. *AAAI-94 Workshop on Knowledge Discovery in Databases*, pages 229–248, 1994.

[5] S. Borzsonyi, D. Kossmann, and K. Stocker. The skyline operator. *IEEE ICDE*, pages 421–430, 2001.

[6] N. Bruno, S. Chaudhuri, and L. Gravano. Performance of multiattribute top-k queries on relational systems. *Tech. Rep. CUCS-021-00*, 2000.

[7] N. Bruno, S. Chaudhuri, and L. Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM TODS*, pages 153–187, 2002.

[8] N. Bruno, L. Gravano, and A. Marian. Evaluating top-k queries over web-accessible databases. *IEEE ICDE*, pages 369–380, 2002.

[9] M. J. Carey and D. Kossmann. Processing top n and bottom n queries. *IEEE Data Engineering Bulletin*, pages 12–19, 1997.

[10] M. J. Carey and D. Kossmann. Reducing the braking distance of an sql query engine. *VLDB*, 1998.

[11] K. Chan and W. Fu. Efficient time series matching by wavelets. *IEEE ICDE*, pages 126–133, 1999.

[12] S. Chaudhuri and L. Gravano. Evaluating top-k selection queries. *VLDB*, pages 397–410, 1999.

[13] C.-M. Chen and Y. Ling. A sampling-based estimator for top-k selection query. *IEEE ICDE*, 2002.

[14] G. Das, D. Gunopulos, and H. Mannila. Finding similar time series. *PKDD*, pages 88–100, 1997.

[15] D. Donjerkovic and R. Ramakrishnan. Probabilistic optimization of top n queries. *VLDB*, pages 411–422, 1999.

[16] R. Fagin. Combining fuzzy information from multilple systems. *PODS*, pages 216–226, 1996.

[17] R. Fagin. Fuzzy queries in multimedia database systems. *PODS*, pages 1–10, 1998.

[18] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *ACM PODS*, pages 102–113, 2001.

[19] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series database. *ACM SIGMOD*, pages 419–429, 1994.

[20] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic time. *ACM Transactions on Mathemtical Software*, pages 209–266, 1977.

[21] A. C. Gilbert, Y. Kotidis, and S. Muthukrishnan. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. *VLDB*, pages 79–88, 2001.

[22] Y.-W. Huang and P. S. Yu. Adaptive query processing for time-series data. *ACM SIGKDD*, pages 282–286, 1999.

[23] E. Keogh. Exact indexing of dynamic time warping. *VLDB*, pages 406–417, 2002.

[24] E. Keogh, K. Chakrabarti, S. Mehrotra, and M. Pazzani. Locally adaptive dimensionality reduction for indexing large time series databases. *ACM SIGMOD*, 2001.

[25] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra. Dimensionality reduction for fast similarity search in large time series databases. *Knowledge and Information Systems*.

[26] E. Keogh and M. Pazzani. Derivative dynamic time warping. *In First SIAM International Conference on Data Mining*, 2001.

[27] E. J. Keogh and T. Folias. The ucr time series data mining archive.

[28] F. Korn, H. V. Jagadish, and C. Faloutsos. Efficiently supporting ad hoc queries in large datasets of time sequences. *ACM SIGMOD*, pages 289–300, 1997.

[29] C. Li, K.-C. Chang, L. F. Ilyas, and S. Song. Ranksql: Query algebra and optimization for relational top-k queries. *ACM SIGMOD*, pages 131–142, 2005.

[30] S. Park, S.-W. Kim, and W. W. Chu. Segment-based approach for subsequence searches in sequence databases. *SAC*, pages 248–282, 2001.

[31] D. Rafiei and A. Mendelzon. Similarity-based queries for time-series data. *ACM SIGMOD*, pages 13–24, 1997.

[32] D. Rafiei and A. Mendelzon. Efficient retrieval of similar time sequences using dft. *Int. Conference on Foundations of Data Organization*, pages 249–257, 1998.

[33] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. *ACM SIGMOD*, pages 71–79, 1995.

[34] M. Theobald, G. Weikum, and R. Schenkel. Top-k query evaluation with probabilistic guarantees. *VLDB*, 2004.

[35] T. Welch. Bounds on the information retrieval efficiency of static file structures. *Technical Report 88, MIT*, 1971.

[36] D. Wu, D. Agrawal, and A. E. Abbadi. Efficient retrieval for browsing large image databases. *ACM CIKM*, pages 11–18, 1996.

[37] B.-K. Yi, H. V. Jagadish, and C. Faloutsos. Efficient retrieval of similar time sequences under time warping. *IEEE ICDE*, pages 201–208, 1998.