Topband Queries in Time Series Data

Ling Li  Mong Li Lee  Wynne Hsu
Abstract

Top-$k$ queries have been extensively studied in various snapshot databases and data streams for applications where the state of an object is static with respect to time. We observe that decision-makers are often interested in a set of objects that exhibit a certain degree of consistent performance over time. In this paper, we introduce a new class of queries called $\lceil k \rceil$-topband that is able to retrieve objects that are within top $k$ at every time point over a specified time interval. Topband queries can be processed using standard SQL and existing top-$k$ methods. However, the SQL method requires the nested loops to compute the top-$k$ result, while the top-$k$ approach leads to large intermediate results and wasted computations. We design a rank-based approach to address these shortcomings. 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.

I. INTRODUCTION

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. 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 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.

However, in practice, there may be extenuating circumstances beyond a student’s control which may lead to a temporary drop in his/her performance. While the formulation of topband queries requires good performances for all time points, it can be easily relaxed to disregard the students’ performance for a few time points.

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 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 $\{stu_2,$
\begin{table}
\centering
\begin{tabular}{|c|c|c|c|}
\hline
id & test 1 & test 2 & average mark \\
\hline
\textit{stu}1 & 92 & 79 & 85.5 \\
\textit{stu}2 & 88 & 89 & 88.5 \\
\textit{stu}3 & 84 & 86 & 85 \\
\textit{stu}4 & 77 & 94 & 85.5 \\
\textit{stu}5 & 72 & 76 & 74 \\
\textit{stu}6 & 78 & 73 & 75.5 \\
\hline
\end{tabular}
\caption{Computing the average scores of the students’ January and February tests to illustrate top-k query.}
\end{table}

\textit{stu}3}. Note that the size of the answer set does not need to be 3.

**Topband vs. Top-k and Skyline Queries.** Topband queries are different from top-k and skyline queries. Topband queries aim 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 objects.

To illustrate, we continue with the example in Figure 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 \textit{stu}2 and \textit{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 [6]. Table I lists the January and February test marks for the students and their averages. A top-3 query will retrieve students \textit{stu}1, \textit{stu}2 and \textit{stu}4, but \textit{stu}1 has not done well in the February test and \textit{stu}4 has not done well in the January test.

Now let us map the January and February test marks of the students in Figure 1 to a two-dimensional space as shown in Figure 2. The x-axis and y-axis in Figure 2 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 [2]. Figure 2 shows the results of a skyline query (\textit{stu}1, \textit{stu}2, \textit{stu}4). Note that \textit{stu}1 and \textit{stu}4 are retrieved although they have not done well in one of the two tests. Further, \textit{stu}3 who has consistently scored above 85, is not retrieved by the skyline query.

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, 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 \textit{RankList} structure from a time series dataset. With this structure, we can quickly answer topband queries. Furthermore, the \textit{RankList} structure can be implemented on top of any relational database system.

The key contributions of the paper are as follows:

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.

To the best of our knowledge, this is the first work to address the problem of finding objects which exhibit some consistent behavior over time.

The rest of the paper is organized as follows. Section II discusses related work. Section III examines how topband queries can be answered using standard SQL and top-k methods. Section IV presents the details of how topband queries can be efficiently processed with the rank information of time series data. The results of experiments to evaluate the proposed approach is presented in Section V, and we conclude in Section VI.

II. RELATED WORK

In this section, we will review techniques to evaluate top-k and skyline queries.

**Top-k Query Processing.** Many methods have been developed to process top-k queries. Fagin’s algorithm [15], [14] carries out a sorted access by retrieving a list of top matches for each condition in the round-robin fashion until $k$ objects that matches all the conditions are found. The TA algorithm [16] generalizes Fagin’s algorithm by computing the score for each object seen during a sorted access. Subsequent work reduces the access cost in the TA algorithm by computing the probability of the total score of an object [23], and exploring upper and lower bounds for the score [5] for early pruning.

The methods in [6], [7], [9], [3], [4], [10], [12] discuss how to map top-k queries to SQL selection queries. For example, the work by [6], [7] propose adding a STOP AFTER clause to SQL to limit the cardinality of a query result. Methods such as Scan-Stop and Sort-Stop [6], range partitioning and semijoin-like methods [7] are utilized to process the STOP AFTER clause. However, these techniques can only be used after evaluating the score for each object.

Top-k queries can be mapped to multidimensional range queries [4], [9]. The key issue is to determine an appropriate search distance that would retrieve the $k$ best matches. The
search distance can be determined from the statistics on the relations \[9\] or from a multi-dimensional histogram \[4\]. The method by \[12\] computes the search distance by taking into account imprecision in the optimizer’s knowledge of data distribution and selectivity estimation while \[10\] adopts a sampling-based approach to determine the search distance.

All the above studies are focused on retrieving top \(k\) answers at one time point. The work in \[19\] designs two algorithms to address the problem of top-k monitoring over sliding windows. The Top-k Monitoring algorithm re-computes the answer of a query whenever some of the current top-k tuples expire, while the Skyband Monitoring algorithm partially pre-computes future results by reducing top-k to k-skyband queries. These methods are not applicable to topband queries because they consider objects in the sliding windows to have constant values until they expire whereas each time series may change its value at every time point.

**Skyline Query Processing.** The work in \[2\] propose two methods to evaluate skyline queries. The first method partitions the dataset to compute partial skylines in memory. The partial skylines are then merged to obtain the final result. The second method is a block nested loop algorithm that calculates skyline points by scanning the dataset and keeping a list of candidate skyline points in the main memory. Chomichi et al. \[11\] improves the block nested loop approach by sorting the dataset according to a preference function. Candidate points are inserted into a list in ascending order of their scores. Points with lower scores are likely to dominate a larger number of points, thus rendering the pruning more effective.

Index-based methods such as \[20\] and \[13\], \[21\] utilize R-trees and bitmaps to evaluate the skyline without scanning the entire dataset. Balke et al. \[1\] propose a multi-objective retrieval algorithm to process skyline queries. This algorithm is similar to the TA algorithm \[16\] where objects are retrieved in a round robin manner for each condition. A virtual object is created with the minimum values for each condition. The search terminates when an object which dominates the virtual object is encountered.

The work of \[18\], \[22\] address skyline monitoring over sliding windows. The work in \[18\] utilizes an encoding scheme to convert the skyline query problem into the *stabbing* query problem, while \[22\] proposes *lazy* and *eager* methods to continuously output changes in the skyline. The *lazy* method delays computations until the expiration of a skyline point while *eager* method precomputes the *skyline influence time* of a tuple. Again, an object does not change its
value in the sliding window.

III. Answering Topband Queries with Existing Methods

This section discusses how topband queries can be answered using existing SQL and top-k methods. We will also highlight the problems of processing topband queries with these approaches.

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 $[k]$-topband query over a time interval $[t_u, t_v]$ will retrieve the set of time series $U = \bigcap U_t$ where $U_t = \text{top-}k(TS,k,t)$ $\forall t \in [t_u, t_v]$. Note that the size of $U$ is between 0 to $k$.

<table>
<thead>
<tr>
<th>id</th>
<th>time</th>
<th>mark</th>
</tr>
</thead>
<tbody>
<tr>
<td>stu1</td>
<td>200601</td>
<td>92</td>
</tr>
<tr>
<td>stu2</td>
<td>200601</td>
<td>88</td>
</tr>
<tr>
<td>stu3</td>
<td>200601</td>
<td>84</td>
</tr>
<tr>
<td>stu4</td>
<td>200601</td>
<td>77</td>
</tr>
<tr>
<td>stu5</td>
<td>200601</td>
<td>72</td>
</tr>
<tr>
<td>stu6</td>
<td>200601</td>
<td>78</td>
</tr>
<tr>
<td>stu1</td>
<td>200602</td>
<td>79</td>
</tr>
<tr>
<td>stu2</td>
<td>200602</td>
<td>89</td>
</tr>
<tr>
<td>stu3</td>
<td>200602</td>
<td>86</td>
</tr>
<tr>
<td>stu4</td>
<td>200602</td>
<td>94</td>
</tr>
<tr>
<td>stu5</td>
<td>200602</td>
<td>76</td>
</tr>
<tr>
<td>stu6</td>
<td>200602</td>
<td>73</td>
</tr>
<tr>
<td>stu1</td>
<td>200603</td>
<td>80</td>
</tr>
<tr>
<td>stu2</td>
<td>200603</td>
<td>91</td>
</tr>
<tr>
<td>stu3</td>
<td>200603</td>
<td>84</td>
</tr>
<tr>
<td>stu4</td>
<td>200603</td>
<td>70</td>
</tr>
<tr>
<td>stu5</td>
<td>200603</td>
<td>76</td>
</tr>
<tr>
<td>stu6</td>
<td>200603</td>
<td>96</td>
</tr>
<tr>
<td>stu1</td>
<td>200604</td>
<td>90</td>
</tr>
<tr>
<td>stu2</td>
<td>200604</td>
<td>80</td>
</tr>
<tr>
<td>stu3</td>
<td>200604</td>
<td>73</td>
</tr>
<tr>
<td>stu4</td>
<td>200604</td>
<td>75</td>
</tr>
<tr>
<td>stu5</td>
<td>200604</td>
<td>94</td>
</tr>
<tr>
<td>stu6</td>
<td>200605</td>
<td>91</td>
</tr>
<tr>
<td>stu1</td>
<td>200605</td>
<td>83</td>
</tr>
<tr>
<td>stu2</td>
<td>200605</td>
<td>78</td>
</tr>
<tr>
<td>stu3</td>
<td>200605</td>
<td>75</td>
</tr>
<tr>
<td>stu4</td>
<td>200605</td>
<td>72</td>
</tr>
</tbody>
</table>
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 $R$.

The time series dataset in Figure 1 can be stored in a relational table as shown in Table II. Then a $[k]$-topband query can be mapped to an SQL query which requires a nested loop. For example, a $[3]$-topband query to retrieve students with consistent performance for the period January to May is equivalent to the following standard SQL query:

```sql
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 $[k]$-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 $[k]$-topband result.

Experiment results in Section V reveal that even the improved SQL approach remains expensive. Next, we examine how we can leverage the top-$k$ operator proposed in [4] to answer $[k]$-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 [9], [4], [12]. The basic framework is as follows:
1) For each time point \( t \) in the specified time interval, estimate the search distance \( \text{dist} \) such that it encompasses at least \( k \) tuples with values greater than \( \text{dist} \). 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 \( \text{distance}(c.\text{time}) \) in the range query denotes the estimated search distance at the time point \( c.\text{time} \).

```
SELECT c.id from student c
WHERE c.time \geq 200601 and c.time \leq 200605
   and c.mark > \text{distance}(c.\text{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.

IV. ANSWERING TOPBAND QUERIES WITH RANK METHOD

Motivated by the relevance of topband queries in many real world applications, and the inadequacies of using standard SQL and top-k methods to answer such queries, we design 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 \( \text{rank}(s,t) \), is affected by the intersection of \( s \) with other time series between the time points \( t - 1 \) and \( t \). Figure 3 illustrates how the rankings of a set of time series may be affected by intersections.

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, \( \text{rank}(s,t) = \text{rank}(s,t - 1) \).

For example, the time series \( s_1 \) and \( s_4 \) in Figure 3(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(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(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 (see Figure 4). Each entry in the list consists of the rank of the time series at the corresponding time point. Note that it is only necessary to create an entry in the inverted list of a time series when its ranking is affected by an intersection. 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. We call this structure RankList.

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, we traverse the list of $stu_1$ in Figure 4 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.
Note that we can stop searching a list whenever an entry in the list with rank value greater than $k$ is encountered.

![Fig. 4. RankList constructed for student dataset in Table II](image)

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.

### A. Algorithms

In this section, we present the algorithms to construct the RankList structure as well as search and updates.

1) **RankList Construction:** Algorithm 1 shows the steps to construct the inverted list structure $\text{RankLst}$ that captures the rank information for each time series in a dataset. The algorithm utilizes two arrays called $\text{PrevRank}$ and $\text{CurrRank}$ 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 $\text{PrevRank}$ and $\text{CurrRank}$ 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 $t$, $\text{CurrRank}[s]$ will be initialized to 1 (lines 7-9). The algorithm scans the database once and compares the values of the time series at each time point (lines 10-16). 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 17-22).

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 $\text{RankLst}$ for 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 - RankList 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;
17:           PrevRank[s] = CurrRank[s];
18:     return RankLst;

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;
time series dataset, an integer $k$, and the start and end time points $t_{\text{start}}, t_{\text{end}}$. The output is $S$, a set of time series whose rank is always higher than $k$ over $[t_{\text{start}}, t_{\text{end}}]$.

$S$ is initialized to be the set of all the time series in the dataset (line 3). The entries for each time series in the RankList 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 4-18). The entry with the largest time point that is less than or equal to $t_{\text{start}}$ 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 6-13). Otherwise, we continue to check the ranks of the entries for $s$ until the end time point is reached.

3) Insert: 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.

Algorithm 3 takes as input a tuple $<p, t, p(t)>$ to be inserted. The algorithm checks for the set of existing time series $S$ whose values are smaller than $p(t)$ at time point $t$ (lines 6-17). 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 $\text{PrevRank}$ (line 7-8). Next, we try to retrieve the entry $<t, \text{rank}>$ 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 14-17 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 $\text{PrevRank}$ at $t + 1$ and insert into RankList (lines 15-16). Finally, we update the rank for the corresponding time series $p$ of the new value at $t$ using $\text{CurrRank}$ (lines 24-28). Note the algorithm also checks the entry for $p$ at $t + 1$ (line 29). 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 30-32).

4) Delete: Similarly, the deletion of a value from a time series dataset may affect entries in the RankList structure.

Algorithm 4 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 5-25). We retrieve the rank of $s \in S$ from the entry which has the largest time point that is less than or equals to
and store it in \textit{PrevRank} (line 6-7). Next, we try to retrieve the entry \(<t, \text{rank}>\) 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 13-16 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 21-23) and insert an entry \(<t, 0>\) for \(p\) (line 24) to indicate the missing value of \(p\) at \(t\).

\begin{algorithm}
\textbf{Algorithm 3 Insert}
1: \textbf{Input:} TS - database with attributes \textit{id}, \textit{time} and \textit{value}
\hspace{1cm} RankLst - RankList structure of TS;
\hspace{1cm} \(<p, t, p(t)\) - a tuple to be inserted;
2: \textbf{Output:} RankLst - updated RankList structure for TS
3: initialize int \textit{CurrRank} to 1;
4: let \(S\) be the set of tuples with time \(t\) in TS;
5: for each tuple \(s \in S\) do
6: \hspace{1cm} if \(p(t) > s.\text{value}\) then
7: \hspace{2cm} locate the entry \(e\) for \(s.\text{id}\) in RankLst which has the largest time point that is less than or equal to \(t\);
8: \hspace{2cm} let \textit{PrevRank} = \(e.\text{rank}\);
9: \hspace{2cm} if \(e.\text{time} = t\) then
10: \hspace{3cm} increment \(e.\text{rank}\) by 1;
11: \hspace{2cm} else
12: \hspace{3cm} create an entry \(<t, \text{PrevRank} + 1>\) for \(s.\text{id}\) and insert into RankLst;
13: \hspace{2cm} locate the entry \(e\) at time \(t+1\) for \(s.\text{id}\) in RankLst;
14: \hspace{2cm} if entry does not exist then
15: \hspace{3cm} create an entry \(<t+1, \text{PrevRank}>\) for \(s.\text{id}\) and insert into RankLst;
16: \hspace{2cm} else
17: \hspace{3cm} \textit{CurrRank}++; /* \(p(t) < s.\text{value} *\)*/
18: \hspace{2cm} locate the entry \(e\) for \(p\) in the RankLst which has the largest time point that is less than or equal to \(t\);
19: \hspace{2cm} if \(e.\text{rank} \neq \textit{CurrRank}\) then
20: \hspace{3cm} if \(e.\text{time} = t\) then
21: \hspace{4cm} replace \(e.\text{rank}\) with \textit{CurrRank};
22: \hspace{3cm} else
23: \hspace{4cm} create an entry \(<t, \text{CurrRank}>\) for \(p\) and insert into RankLst;
24: \hspace{2cm} locate the entry \(e'\) at time point \(t+1\) for \(p\) in RankLst;
25: \hspace{2cm} if \(e'\) does not exist then
26: \hspace{3cm} create an entry \(<t + 1, 0>\) for \(p\) and insert into RankLst;
27: \hspace{1cm} return RankLst;
\end{algorithm}
B. 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 $O(T^*N \log N)$ where $(N \log N)$ 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 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.

**Algorithm 4 Delete**

1. **Input:** TS - database with attributes id, time and value  
   RankLst - RankList structure of TS;  
   <p, t, p(t)> - a tuple to be deleted;  
2. **Output:** RankLst - updated RankLst 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 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;  
12. locate the entry e at time point t + 1 for s.id in RankLst;  
13. if entry does not exist then  
14. create an entry <t + 1, PrevRank> for s.id and insert into RankLst;  
15. locate the entry e for p in the RankLst 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;  
17. if e' does not exist then  
18. create an entry <t + 1, e.rank> for p and insert into RankLst;  
19. create an entry <t, 0> for p and insert to RankLst;  
20. return RankLst;

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 structure depends on the number of intersection points. 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 structure is O(N*T). In practice, we expect the size of the RankList 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 to reduce the size of the RankList structure (see Section V-A).

C. 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.

The inverted list structure in Figure 4 can be mapped to the RankTable relation in Table III. \([k]\)-topband queries can now be answered by issuing SQL queries over the relation RankTable 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:

(Q1) SELECT c.id FROM RankTable c
WHERE c.time = 200601
and NOT EXISTS ( SELECT c1.id
FROM RankTable c1
WHERE c1.id = c.id
and c1.time >= 200601
and c1.time <= 200605
and c1.rank > 3 )

The condition \([c.time = 200601]\) in the above SQL query guarantees that the subquery is executed only once for each time series.
Note that it is also possible to utilize the following query to accomplish the same purpose:

\[(Q2) \text{ SELECT } c.\text{id} \text{ FROM RankTable } c \]
\[
\text{WHERE } c.\text{time} \geq 200601 \text{ and } c.\text{time} \leq 200605
\]
\[
\text{GROUP BY } c.\text{id}
\]
\[
\text{HAVING } \text{max}(c.\text{rank}) \leq 3
\]

Fig. 5. Time taken to process queries Q1 and Q2 by varying the number of time series

Fig. 6. Time taken to process queries Q1 and Q2 by varying the length of query interval

The major difference between queries Q1 and Q2 is the "GROUP BY" clause in Q2. This clause will require the tuples in each group to be sorted, thus incurring additional time. We carried out two experiments to compare the time cost to process Q1 and Q2 as the number
of time series and length of query interval increase. The experiment settings are the same as presented in the paper. The results are shown in Figure 5 and Figure 6. We observe that the time cost to process Q2 is obviously more than Q1.

<table>
<thead>
<tr>
<th>id</th>
<th>time</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>stu1</td>
<td>200601</td>
<td>1</td>
</tr>
<tr>
<td>stu1</td>
<td>200602</td>
<td>4</td>
</tr>
<tr>
<td>stu1</td>
<td>200603</td>
<td>3</td>
</tr>
<tr>
<td>stu1</td>
<td>200604</td>
<td>1</td>
</tr>
<tr>
<td>stu2</td>
<td>200601</td>
<td>2</td>
</tr>
<tr>
<td>stu2</td>
<td>200603</td>
<td>1</td>
</tr>
<tr>
<td>stu3</td>
<td>200604</td>
<td>2</td>
</tr>
<tr>
<td>stu3</td>
<td>200601</td>
<td>3</td>
</tr>
<tr>
<td>stu3</td>
<td>200603</td>
<td>2</td>
</tr>
<tr>
<td>stu3</td>
<td>200604</td>
<td>3</td>
</tr>
<tr>
<td>stu4</td>
<td>200601</td>
<td>5</td>
</tr>
<tr>
<td>stu4</td>
<td>200602</td>
<td>1</td>
</tr>
<tr>
<td>stu4</td>
<td>200603</td>
<td>0</td>
</tr>
<tr>
<td>stu4</td>
<td>200605</td>
<td>4</td>
</tr>
<tr>
<td>stu5</td>
<td>200601</td>
<td>6</td>
</tr>
<tr>
<td>stu5</td>
<td>200602</td>
<td>5</td>
</tr>
<tr>
<td>stu5</td>
<td>200601</td>
<td>4</td>
</tr>
<tr>
<td>stu6</td>
<td>200602</td>
<td>6</td>
</tr>
<tr>
<td>stu6</td>
<td>200603</td>
<td>4</td>
</tr>
<tr>
<td>stu6</td>
<td>200605</td>
<td>6</td>
</tr>
</tbody>
</table>

TABLE III
EXAMPLE RankTable relation

The RankTable 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 \( S_1 - S_4 \) when the tuple \(<stu_4, 200603, 78>\) is inserted into the time series relation:

\( (S_1) \) CREATE VIEW V(id, time, rank)
AS ( SELECT c.id, c.time, c.rank
FROM RankTable 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 c1
WHERE c1.id = c.id
and c1.time <= 200603 ) )
\((S_2)\) INSERT INTO RankTable (id, time, rank)

\[
\text{SELECT p.id, 200603+1, p.value from V p}
\]

WHERE p.id NOT IN

\[
( \text{SELECT c.id}
\text{FROM RankTable c}
\text{WHERE c.id = p.id}
\text{and c.time = 200603+1})
\]

\((S_3)\) INSERT INTO RankTable (id, time, rank)

\[
\text{SELECT p.id, 200603, p.value + 1 FROM V p}
\]

WHERE p.id NOT IN

\[
( \text{SELECT c.id}
\text{FROM RankTable c}
\text{WHERE c.id = p.id}
\text{and c.time = 200603})
\]

\((S_4)\) UPDATE RankTable SET rank = rank + 1

WHERE id IN ( SELECT id from V )

and time = 200603

The statement \(S_1\) 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 \(S_2\) inserts entries at time point \(t+1\) for the set of time series whose ranks are affected at time point \(t\). Statements \(S_3\) and \(S_4\) 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 relation when a value is deleted from the dataset.

V. PERFORMANCE STUDY

In this section, we present the results of three sets of experiments to evaluate the efficiency and scalability of the proposed method. We implement the algorithms in Section IV-A in Java (JDK version 1.5.01). The synthetic data generator produces time series datasets with attributes \(id\), \(time\) and \(value\). Table IV shows the range of values for the various parameters and their default values.
The first set of experiments examines the time taken to construct, search and update the RankList structure, as well as its space requirements. From the theoretical analysis in Section IV-B, we note that the RankList structure is affected by the number of intersection points in a time series dataset. As such, we investigate the effect of applying smoothing techniques such as the Haar Wavelet Transform [8] 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) 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.

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.

A. 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 structure. 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 2% to 10% of the total possible number of intersection points.

We also use the Haar Wavelet Transform to smooth the dataset. The Haar transform allow 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:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Range</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of time series N</td>
<td>[100, 500]</td>
<td>100</td>
</tr>
<tr>
<td>Number of time points T</td>
<td>[5000, 20000]</td>
<td>10000</td>
</tr>
<tr>
<td>k</td>
<td>[50, 250]</td>
<td>50</td>
</tr>
<tr>
<td>Length of query interval L</td>
<td>[5000, 10000]</td>
<td>10000</td>
</tr>
<tr>
<td>Percentage of intersection points</td>
<td>[2%, 10%]</td>
<td>5%</td>
</tr>
</tbody>
</table>

TABLE IV
PARAMETERS OF DATASET GENERATOR
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 \(2, 1, \text{and } -1\).

Different degrees of smoothing can be achieved by limiting the size of the Haar transform. A parameter \textit{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 (\textit{threshold} = 2/4 = 0.5), then the resulting time series is reconstructed as \((6 + 2, 6 + 2, 6 - 2, 6 - 2) = (8, 8, 4, 4)\).

Figure 7 plots the total time taken to build the RankList structure as the smoothing threshold, indicated in brackets, varies from 0.3 to 0.9. We observe that as the number of intersection points increases, the construction time increases since the rankings of more time series will change with the occurrence of intersections. Figure 7 also indicates that as the smoothing threshold decreases, the time to construct the RankList structure decreases despite the increase in the percentage of intersection points.

Figure 8 shows that the time taken to search the RankList is almost independent of the number of intersection points. The reason could be due to the pruning strategy which provides for early termination of the search. We also observe that the search time 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 structure by varying the number of insertions/deletions from 20 to 100. Figure 9 shows that the update time increases linearly with the number of insertions/deletions.

Figure 10 gives the space requirements of the RankList structure. As expected, the space 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. We see that as the...
smoothing threshold decreases from 0.9 to 0.3, the space required by the RankList structure decreases despite the increase in the percentage of intersection points.

B. Experiments on Topband Queries

In this section, we present the results of the second set of experiments that compare the proposed Rank method 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.

![Fig. 7. Time to construct RankList](image1)

![Fig. 8. Time to search RankList](image2)
We map the RankList structure to a relational table called RankTable, and issue SQL queries on this relation as described in Section IV-C. 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 III. We use the method in [4] to estimate the distance for the range query at various time points in top-k query approach.

1) Effect of Number of Intersection Points: We first study the effect of the number of intersection points. Figure 11 shows the time taken by the three methods in log scale. We observe that the runtimes of the nested SQL and the top-k approach are not affected by the increase in the number of intersection points. This is expected since the two approaches require a complete database scan regardless of the number of intersection points. In contrast, the time
taken by the rank approach is a small fraction of that required by the other two methods. This time taken may be further reduced by the early pruning strategy, as shown by the drop at the 9% and 10% intersection points in Figure 11.

![Figure 11. Effect of number of intersection points with the response time in log scale](image)

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.

We first study the performance of the three approaches by varying the value of $k$ in the queries. Figure 12(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 approach increases 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 12(b). We observe that the runtime of all three approaches increase as the length of the query interval increases. This is expected as more tuples are processed. However, the rank approach outperforms the nested and the top-k approaches by a large margin.

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 13(a) shows the time taken by all the three 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 approach outperforms 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 the many wasted computations and large intermediate results at every time point. Note that when the number of time series exceed 400, the top-k approach performs even worse than the nested approach. The 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 query interval is fixed at 10,000. Figure 13(b) shows that the time taken by all three approaches are independent of the underlying dataset size. Clearly, the proposed approach is efficient and scalable.
4) Experiments in a Non-DBMS Environment: This set of experiments is similar to the experiments presented in Section V-B.1 – Section V-B.3, except that the evaluation is not embedded in a DBMS. The results are shown in Figure 14 – Figure 16. Similar trends are observed, and the proposed method outperforms the top-k method.

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.
C. 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.** We first examine the effect of smoothing on a real world stock dataset [17] 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. We define the precision and recall of topband queries as follows:

\[
\text{precision} = \frac{n_t}{n_t + n_f} \times 100\% \tag{1}
\]

\[
\text{recall} = \frac{n_t}{n_t + n_m} \times 100\% \tag{2}
\]

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 structure. As expected, the space requirement for the RankList structure decreases as the threshold is decreased (see Figure 17).
The recall is 100% for all values of the smoothing threshold. Figure 18 shows that the precision decreases as the threshold is decreased. Further, 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.

![Graph showing space of RankList for stock dataset](image)

**Fig. 17.** Space of RankList for *stock* dataset

![Graph showing precision vs. smoothing threshold for the stock dataset](image)

**Fig. 18.** Precision vs. smoothing threshold for the *stock* dataset

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 V shows that the set of stocks retrieved by topband queries consistently attain higher gains than the stocks retrieved by top-k queries.
<table>
<thead>
<tr>
<th>$k$</th>
<th>2001</th>
<th>2002</th>
<th>2003</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>8.3%</td>
<td>16.4%</td>
<td>34.8%</td>
<td>18%</td>
</tr>
<tr>
<td>20</td>
<td>6.76%</td>
<td>9.8%</td>
<td>17.97%</td>
<td>10.6%</td>
</tr>
<tr>
<td>30</td>
<td>7.5%</td>
<td>9.6%</td>
<td>6.75%</td>
<td>8.2%</td>
</tr>
<tr>
<td>40</td>
<td>10.9%</td>
<td>11.9%</td>
<td>13.3%</td>
<td>11.8%</td>
</tr>
<tr>
<td>50</td>
<td>5.2%</td>
<td>4%</td>
<td>7.2%</td>
<td>5.1%</td>
</tr>
</tbody>
</table>

| \[\lceil k \rceil\]-topband queries \[\text{Percentage gains of stocks retrieved by topband over top-k queries.}\]

This further strengthens our confidence in the ability of topband queries to identify the potential merits of a portfolio of stocks.

**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 19(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 19(b) shows the consistent performers grouped by gender. We observe 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.
VI. CONCLUSION

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 $T$. 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, where $T' \leq T$, or retrieve a set of time series which always outperforms a particular time series over some time interval.

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 margin. 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.

REFERENCES


