7-2022

Finding top-m leading records in temporal data

Yiyi WANG
Singapore Management University, yiyiwang.2019@phdcs.smu.edu.sg

FINDING TOP-M LEADING RECORDS IN TEMPORAL DATA

WANG YIYI

SINGAPORE MANAGEMENT UNIVERSITY

2022

Finding Top-*m* Leading Records in Temporal Data

WANG Yiyi

Submitted to School of Computing and Information Systems in partial fulfillment of the requirements for the Degree of Master of Philosophy in Information Systems

Master's Thesis Committee:

Kyriakos MOURATIDIS (Supervisor / Chair)
Associate Professor of Information Systems
Singapore Management University

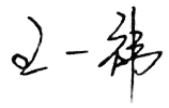
FANG Yuan
Assistant Professor of Computer Science
Singapore Management University

LI Yuchen
Assistant Professor of Computer Science
Singapore Management University

Singapore Management University
2022

I hereby declare that this Master's thesis is my original work and it has been written by me in its entirety.
I have duly acknowledged all the sources of information which have been used in this thesis.

This Master's thesis has also not been submitted for any degree in any university previously.



WANG Yiyi
7 July 2022

Finding Top-*m* Leading Records in Temporal Data

WANG Yiyi

ABSTRACT

A traditional top-*k* query retrieves the records that stand out at a certain point in time. On the other hand, a durable top-*k* query considers how long the records retain their supremacy, i.e., it reports those records that are consistently among the top-*k* in a given time interval. In this thesis, we introduce a new query to the family of durable top-*k* formulations. It finds the top-*m* leading records, i.e., those that rank among the top-*k* for the longest duration within the query interval. Practically, this query assesses the records based on how long they stay ahead of competition. We perform a case study with real NBA data to demonstrate the value of the query. In addition, we present a meaningful problem variant for the special scenario where the data are sparse. We propose a first-cut algorithm for solving the problem, which we later enhance with an early termination condition. We compare the two versions of the algorithm and demonstrate their practicality using synthetic and real datasets.

TABLE OF CONTENTS

ACKNOWLEDGEMENT

I would first like to express my deepest gratitude to my advisor Professor Dr. Kyriakos MOURATIDIS for the continuous support of my research. Your patience, enthusiasm and immense knowledge have inspired and motivated me throughout the whole journey. Your guidance has helped me in all the time of research and writing of this thesis. Your advice on both my research and my life have been invaluable.

Besides my advisor, I would also like to thank my committee members, Professor Dr. FANG Yuan and Professor Dr. LI Yuchen for serving on my thesis committee. Your support, brilliant comments and questions have made my defense an enjoyable moment. Your suggestions have brought in threads of thought that made my thesis so much richer.

Finally, I must express my very profound gratitude to my family for providing me with unfailing support and continuous encouragement throughout my life.

1. INTRODUCTION

Top-*k* queries are well-studied to retrieve a small set of records that best match users' preferences, from a large database. Temporal data, referring to objects that change over time, are pervasive in a variety of domains. Therefore, researchers devote many efforts to query such data recently. In this case, traditional top-*k* queries only filter the records standing out at a certain point-in-time without considering how long they retain the supremacy. Extending top-*k* search to temporal databases, durable top-*k* queries are proposed to return the top objects with durable quality over time. In general, a durable top-*k* query finds the set of objects that are consistently in the top-*k* results throughout a given time interval. Durable top-k queries have many variants and applications. For example, with stock price data, a durable top-*k* query may retrieve the stocks whose price-to-earning ratios are among the lowest 10 in the tech industry for more than 80% of the time over the past year [3]. As a more complex example, with NBA player performance statistics, a recent paper extends durable queries to support the durability claims, such as: "Since 2006, Kobe's 81 points scoring performance has yet to be broken as of today." [5]

In this paper, we consider the problem of finding the top-*m* leading records, which is a variant of durable top-*k* query with practical significance. When we assess the ability of some people or the value of products, their ranking is definitely the most straightforward metric. However, it is also important for the top-ranking records to stay ahead. For example, some advertising statement emphasizes, "The product is

among the top-3 for continuous 10 years", to show that it has the leading position in the industry. In real life, we always recognize a product to be the market leader when it is always among the top few in the industry. For this paper, in order to find the leading records, given a historical temporal dataset, we are intended to retrieve the records that rank among the top k for the longest duration within the query interval. A more formal problem definition is given below.

PROBLEM DEFINITION. Consider a dataset P with n records, where each record $p \in P$ has d time-varying attributes, and a discrete time domain of interest $T = \{1, 2, \ldots, l\}$. At a timestamp t_i, a top-k query asks for the k records having the highest score with respect to a user specified monotone scoring function f. For dataset P, given a query interval I: $[t_b, t_e] \subseteq T$, an integer k and m as inputs, a **top-m leading records** query returns the set of m records, remaining in the top-k between t_b and t_e for the longest duration (i.e., for the most timestamps) in I.

The top-m leading records query has numerous practical applications in different domains, as illustrated next.

EXAMPLE 1. For some review and rating applications (e.g., Yelp, Agoda, etc.), multi-criteria ratings of different objects are updated from time to time. Each user can assign a set of weights representing his/her own preference on the attributes using preference tools (e.g., a sliding bar). With weighted sum scoring function, it is easy to

find the best choice at that time for different users. However, some frauds (e.g., click farming[1]) have huge impact on the outcome of this approach. In order to have an immediate effect on the ratings, a large number of fake reviews are always generated together within a short period. In this case, a top-*m* leading records query only recommends users the choices which keep ranking at the top, and filter out those short-lived sensations. The result ranking is hardly affected by temporary malicious manipulation.

EXAMPLE 2. Estimating the value of different stocks is always worth studying for investors. By integrally analyzing multiple attributes (e.g., open price, close price, volume, etc.), investors can evaluate a stock more accurately. Besides, it is also a wise choice to invest in stable stocks, which keep superior to others for the most time. Therefore, the top-*m* leading records query caters well for such application.

EXAMPLE 3. NBA statistics (e.g., points, rebounds, assist, blocks, and steals per game) are fairly intuitive to estimate various abilities of players. However, such simple data are not enough to evaluate a player and his influence over years comprehensively. Therefore, given the NBA statistics for recent years, a top-*m* leading records query finds the leading players who have the highest aggregate scores for the most time. Such query identifies those players with a certain dominance over world basketball.

[1] Click farming is typically launched by multiple fake or compromised accounts which are used to generate fake reviews to boost or diminish the ratings of listed products and services [14].

2. RELATED WORK

A time series is a sequence of real numbers, each number representing a value at a time point. They have been studied in various database applications, while relatively little efforts have been made for time series of multidimensional data. However, such type of data are currently of growing importance in a wide variety of applications. With major time series related tasks including indexing, clustering, classification, prediction, anomaly detection, and so on, time series data is related to every field of human activities [15, 16, 17].

Most existing related papers focus on the similarity search problem, which searches a database to find multidimensional data sequences that are similar to a given query sequence [7]. Several works propose techniques to compute the similarity between trajectories of moving objects [8, 9], which is the basis of many interesting applications (e.g., determining migration patterns of certain groups of animals). The notion of time series similarity is also essential for many other mining tasks, e.g., clustering, pattern matching, prediction, and anomaly detection. Laftchiev and Liu [10] present the problem of finding the top-*k* matches to a multidimensional query pattern in a multidimensional time series dataset, which is important for the localization of abnormal patterns in very long time series. Other than that, some papers aim to identify the motifs and anomalies to better predict future trends [11, 12]. Last but not least, Jiang et. al [13] present a work on continuous multidimensional top-*k* query

processing in wireless sensor networks. They believe that the recent development of sensor networks has made it possible for people to search information not only in the cyber space, but also in the physical world, someday in the future. In this case, the top-*k* search should be not only multidimensional, but also across the time domain. Also, with multidimensional sensor data, as many queries as the number of user requests have to be posed since each user could assign a set of weights representing his own preference. They develop a framework which effectively monitors user queries and incorporates a special data structure, the dominant graph, to maintain top-*k* query results. They declare that this is the first work for continuous multidimensional top-k query processing in sensor networks.

This thesis is inspired by the durable top-*k* queries presented in [5], which find the records that are among the top-*k* within a surrounding time window. Effectively, they look back in a fixed length window ending at the arrival time of the record, and check whether it has the top score among all other records within this window. Therefore, given a query interval *I'* and a durability threshold τ, they check whether every record arriving during *I'* remains in the top-*k* for at least τ timestamps ending at its arrival time. However, our proposed problem focuses on the whole time series and looks for the leading records overall.

Gao et. al [3] also propose the problem of finding durable top-*k* objects, whose values are among the top-*k* for at least some fraction of the time during a given interval. This

is a one-dimensional time series problem, aiming to retrieve all records considering a given time constraint regardless of the ranking, which is different from ours. All of their solutions must compute for each record whether it is among top-k at every timestamp and compare the sum with a threshold. Since every record only has one time-varying attribute, selection algorithm, which selects the k-th largest value from all records, is used to compute the membership of each record in the top-k. Therefore, it is meaningless to adapt their solution to our problem, which computes which records are among the top-k per timestamp and then rank them to find the final result.

3. PROPOSED SOLUTION

Our basic solution is to retrieve the top-k results at each timestamp according to the user specified scoring function. For each object in those top-k results, we compute the total count of timestamps when it is among the top-k, in order to generate the final top-m leading result set. Such solution has two modules: (i) an algorithm for the computation of the top-k results at every timestamp, and (ii) a method which integrates all the top-k results to obtain the final result.

Algorithm 1 Top-m leading records

Input: a query interval I: [t_b, t_e], an R-tree (RT_i) on the dataset for every timestamp t_i within the query interval I, a scoring function f, an integer k, an integer m
Output: result set C
1. initiate a result set C with size m
2. initiate a top-k result set L_i for each timestamp t_i with size k
3. **for** each timestamp t_i in I
4. retrieve the top-k list L_i by calling **Algorithm 2** with input: RT_i, f, k
5. integrate all the top-k results by calling **Algorithm 3** with input: all L_i, m, I
6. **return** C

end

There are various techniques aiming to answer top-k queries and most of them assume monotone scoring functions. The Branch-and-bound search (*BBS*) [19] is a good choice for this baseline method. For each timestamp, the data are indexed with an R-tree [20,21], which groups nearby objects and encloses them with their minimum bounding rectangle (*MBR*) in the parent node. *MBR*s at the same level are recursively bounded into nodes at the higher level and the leaves of the tree contain pointers to the database objects. Based on R-tree, *BBS* traverses the tree nodes using the best-first search technique, which maintains a heap H containing the entries of nodes visited so far in descending order of their *maxscore*. For a data point, *maxscore* equals its score of f. The *maxscore* of an intermediate entry equals the largest score of any point that may lie in the subtree of it. At each step, *BBS* de-heaps the entry having the largest *maxscore* and en-heaps all the entries in its child node. When it reaches a leaf node, the corresponding data point definitely has the largest score among all the records that have not been visited. Therefore, it can be returned as one of the top-k records. The algorithm terminates when k records are reported.

Algorithm 2 BBS

Input: RT_i, f, k
Output: result set L_i
1. initiate a candidate heap H
2. initiate a result set L_i with size k
3. load the root of RT_i
4. **for** each entry e in the root
5. calculate the *maxscore* of its *MBR (e.maxscore)* according to f
6. en-heap (e, *e.maxscore*) into H
7. **while** (L_i contains less than k records)
8. de-heap the entry e' having the largest *maxscore*
9. **if** e' is a leaf node, **then** add it to L_i
10. **else** en-heap all the entries in its child node with their *maxscore*s
11. **return** L_i

end

After executing the first module, a set L_i containing k objects is generated for each timestamp t_i in the given interval I. The remaining process is to aggregate the timestamp counts for all the candidates in these sets and obtain the top-m leading objects with the most counts. We create a hash table HT, which maps each candidate to its duration counts. Starting with the top-k set L_b of the first timestamp t_b, we iterate over each top-k set. Every time we encounter a candidate object, we look it up in HT. If this object is visited for the first time, we add it to HT with a value of 1. Otherwise, we increase its value by 1. Meanwhile, we also maintain candidate set C that contains the current m leading objects, the m-th largest value m_{max} in C, and largest value r_{max} among the remaining objects, which are not included in C. As traversing top-k sets, we keep updating HT, C, m_{max}, and r_{max}. At the timestamp t_i, if $t_e - t_i \leq m_{max}$, we can stop adding new entries to HT, because if an object has not occurred in HT so far, it is impossible to have a longer duration than current candidates in C. This algorithm terminates when: (i) all top-k sets are visited, or (ii) $r_{max} + (t_e - t_i) \leq m_{max}$, which means

there is no chance for the rest objects to surpass the m-th candidate in C, even if they are in the top-k for all the remaining timestamps.

Algorithm 3 Integrate all top-k lists

Input: $L_i \in [L_b, \ldots, L_e]$, m, I: $t_i \in [t_b, t_e]$
Output: result set C

```
1.  initiate a duration count cnt
2.  initiate a result set C with size m
3.  initiate m_max = 0 to maintain the m-th largest duration count in C
4.  initiate r_max = 0 to maintain the largest duration count outside C
5.  initiate a hash table HT /* HT maps object o to its cnt in the form (o, o.cnt)*/
6.  while (not all L_i has been visited and r_max + (t_e - t_i) > m_max)
7.     for each L_i
8.        for each object o in L_i
9.           if o is not in HT and m_max < (t_e - t_i), then add (o, 1) to HT
10.          else o.cnt++
11.          if there are less than m objects in C, then add o to C
12.          else if o.cnt > m_max, then replace the m-th object with o and m_max = o.cnt
13.          else if o.cnt > r_max, then r_max = o.cnt
14. return C
end
```

Algorithm 4 Optimized top-m leading records

Input: a query interval I: $[t_b, t_e]$, an R-tree (RT_i) on the dataset for every timestamp t_i within the query interval I, a scoring function f, an integer k, an integer m
Output: result set C

```
1.  initiate a result set C with size m
2.  initiate a top-k result set L_i for each timestamp t_i with size k
3.  initiate a duration count cnt
4.  initiate m_max = 0 to maintain the m-th largest duration count in C
5.  initiate r_max = 0 to maintain the largest duration count outside C
6.  initiate a hash table HT /* HT maps object o to its cnt in the form (o, o.cnt)*/
7.  while (r_max + (t_e - t_i) > m_max)
8.     for each timestamp t_i in I
9.        retrieve the top-k list L_i by calling Algorithm 2 with input: RT_i, f, k
10.       for each object o in L_i
9.           if o is not in HT and m_max < (t_e - t_i), then add (o, 1) to HT
10.          else o.cnt++
11.          if there are less than m objects in C, then add o to C
12.          else if o.cnt > m_max, then replace the m-th object with o and m_max = o.cnt
13.          else if o.cnt > r_max, then r_max = o.cnt
14. return C
End
```

The algorithm retrieves top-k lists for every timestamp and integrates them thereafter. With the early termination condition, we may reduce the time cost of integrating process by skipping some of these lists. Therefore, we optimize the algorithm by checking the termination condition before computing top-k results of each timestamp, as shown in Algorithm 4. In such setting, the computation time of retrieving unnecessary top-k lists is saved.

4. A MEANINGFUL VARIANT

In a special scenario, for some tuples, there may be no data in some timestamps. It is unfair to compare such records with those which have data in every timestamp. Intuitively, a record, which is among the top-k in most of its timestamps, is considered

a leading record. Thus, we propose a variant of top-m leading records problem as follows.

VARIANT PROBLEM DEFINITION. Consider a dataset P with n records, where each record $p \in P$ has d time-varying attributes, and a discrete time domain of interest $T = \{1, 2, \ldots, l\}$. At a timestamp t_i, a top-k query asks for the k records having the highest score with respect to a user specified monotone scoring function f. In dataset P, given a query interval I: $[t_b, t_e] \subseteq T$, for each record p, we compute the number of timestamps τ_p that p has data during I. Given an integer k and m as inputs, a **variant top-m leading records** query returns the set of m records, remaining in the top-k for the highest percentage of τ_p.

Adapting the basic algorithm to solve this problem, the first module is the same. After generating the top-k lists, for each record p in the lists, we search it in each timestamp and compute the total number of timestamps τ_p where it occurs. Similarly, we create a hash table HT, which maps each candidate to its duration counts divided by τ_p. Every time we encounter a candidate object, we look it up in HT. Instead of increasing its value by 1 every time, we increase it by $1/\tau_p$. Meanwhile, we also maintain candidate set C that contains the current m leading objects, the m-th largest value m_{max} in C. As traversing top-k sets, we keep updating HT, C and m_{max}. This algorithm terminates when: (i) all top-k sets are visited, or (ii) $m_{max} = 1$.

A weakness of this definition is that if a tuple has data for only one (or just a few) timestamps in I where it ranks high, it will be included in the result although other tuples with slightly lower percentages but much longer durations would be omitted. In applications where so sparse data are possible, the variant's definition could be enhanced with the extra condition that τ_p must be at least θ (where θ is a problem parameter between 1 and the length of the query interval I).

5. NBA CASE STUDY

We use a real-life dataset (NBA[2]), containing the performance per game of each NBA player in each season from 1973 to 2021. Each record has 5 numeric attributes: *rebounds, assists, steals, blocks and points*. A function *f = 0.2*rebounds + 0.2*assists + 0.2*steals + 0.2*blocks + 0.2*points* is used to assess the overall abilities of each player.

With input I = [1973, ..., 2021], k = 10 and m = 10, the query returns {LeBron James, Karl Malone, Shaquille O'Neal, Kobe Bryant, Michael Jordan, Hakeem Olajuwon, Charles Barkley, Kevin Durant, Allen Iverson, Magic Johnson} as the top-10 leading NBA players who are among the top-10 for the most times. All the 10 players retrieved are representative figures, who are included in the official list[3] of the greatest 75 players in NBA history. When it comes to achievements in the NBA, the

[2] Collected from https://www.basketball-reference.com/
[3] ESPN's ranking of the NBA's Top 75 Players: https://www.espn.com.sg/nba/story/_/id/32432119/nba-75-greatest-players-all-complete-list

championship ring has the greatest impact on player honor. Some of them (e.g., LeBron James, Shaquille O'Neal, Kobe Bryant, Michael Jordan, etc.) have won multiple NBA championships, and are famous even to people who have never watched NBA competitions. Moreover, the results also give recognition to the well-known uncrowned kings[4] (i.e., Karl Malone, Charles Barkley and Allen Iverson). With the function averagely considering the five attributes, the query recommends the players showing the greatest overall aptitude, regardless of their positions and awards.

6. EXPERIMENTS

In this set of experiments, we examine the efficiency (in elapsed running time) of the top-*m* leading records algorithm (TLR) and its optimization (OTLR) using *Uniform (UNI)* synthetic datasets.

6.1 Experimental setting

All generated values of attributes range from 0 to 1. In *UNI* datasets, values are uniformly distributed. The datasets are indexed by R-trees and kept in memory. Note that building R-trees belongs to the pre-computation process in our setting. Thus, the running time of that process is excluded. The techniques are implemented in C++ and the experiments run on a machine with Intel i7-8750H CPU at 2.20GHz and 16Gb RAM.

[4] The 12 uncrowned kings in NBA history, who do you think is the most regrettable?: https://daydaynews.cc/en/nba/876241.html

Table 1 lists the problem parameters along with their tested values. The default values for the input parameters are n = 10K, d = 3, k = 50, m =10, and the number of timestamps is 30. In each experiment, we fix four parameters to their default values and vary the value of the fifth one.

Name	Tested value
Synthetic dataset size for each timestamp (n)	1K, 5K, **10K**, 30K, 50K, 100K, 200K
# of attributes (d)	2, **3**, 4, 6, 8
k	5, 10, **50**, 100, 300, 500
m	1, **10**, 50, 100. 200, 300, 500
# of timestamps	5, 10, **30**, 50, 100

Table 1: Tested parameters and their ranges (with default values in bold)

6.2 Experimental Results

6.2.1 Varying n

In the first experiment, we study the effect of data size on the computation time of the proposed algorithms. As shown in Figure 1 below, as n increases, the running time of both TLR and OTLR increases, since fetching more records to retrieve the top-k results costs more time. Also, as expected, OTLR is faster than TLR.

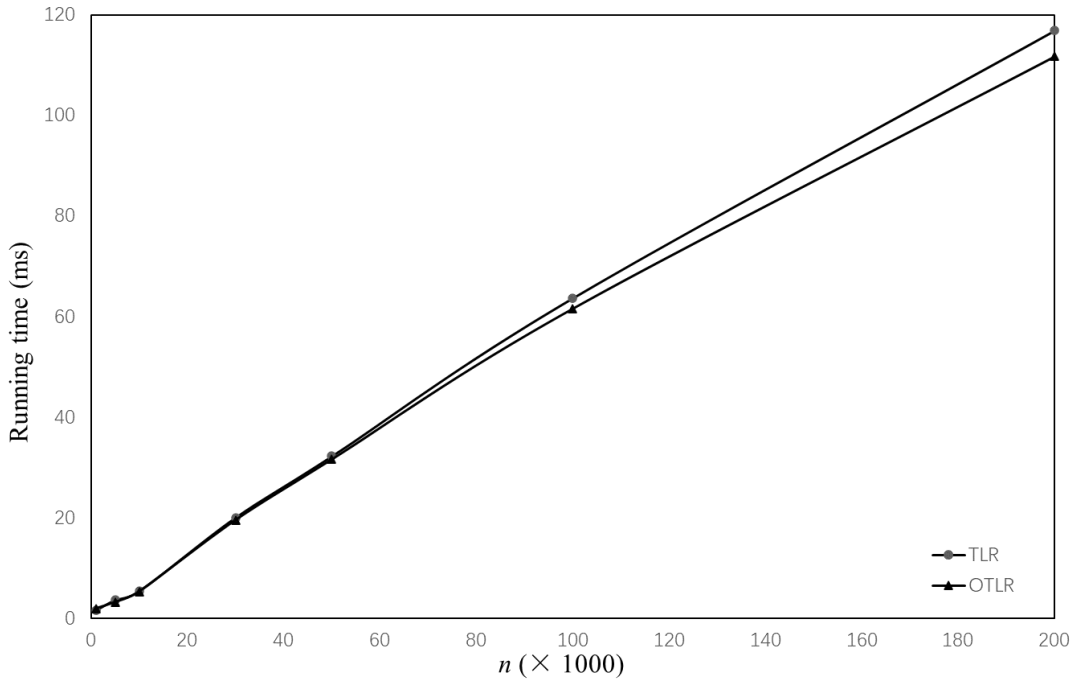


Figure 1: Effect of *n*

6.2.2 Varying d

Figure 2 shows how a growing number of dimensions affects computation time. The trends become increasingly sharp, which means the performance of both algorithms degrades. The curse of dimensionality occurs due to the spatial nature of our problem and index (R-tree) [22].

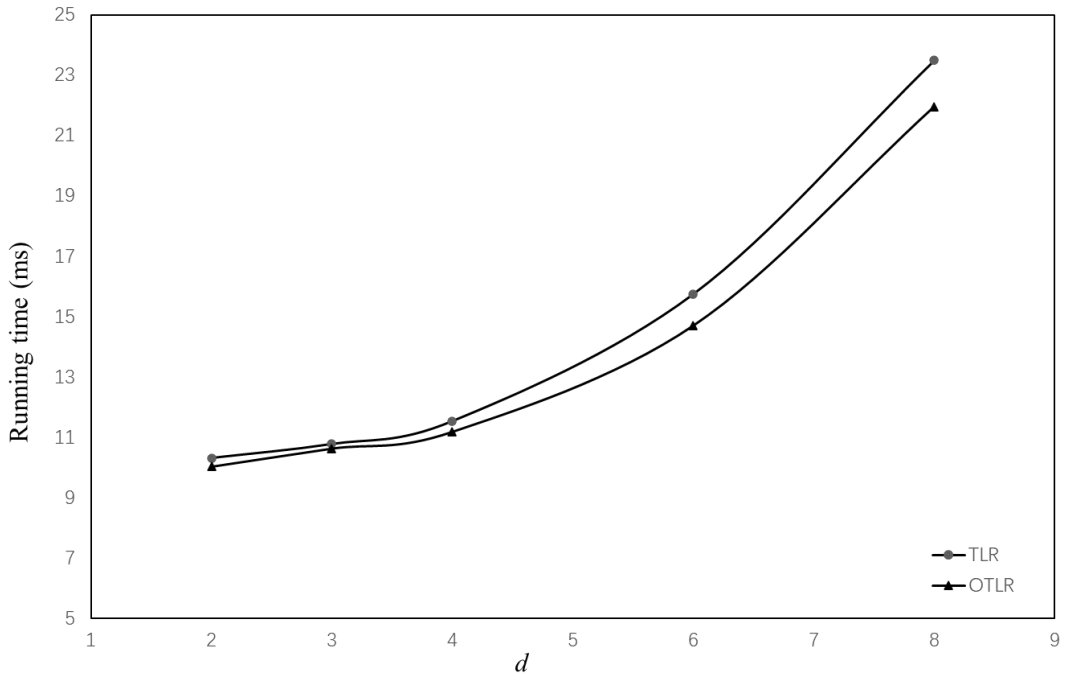


Figure 2: Effect of *d*

6.2.3 Varying k

Naturally, the running time increases with k in general, because both algorithms retrieve more top-k results and finding m leading records from more candidates cost more time.

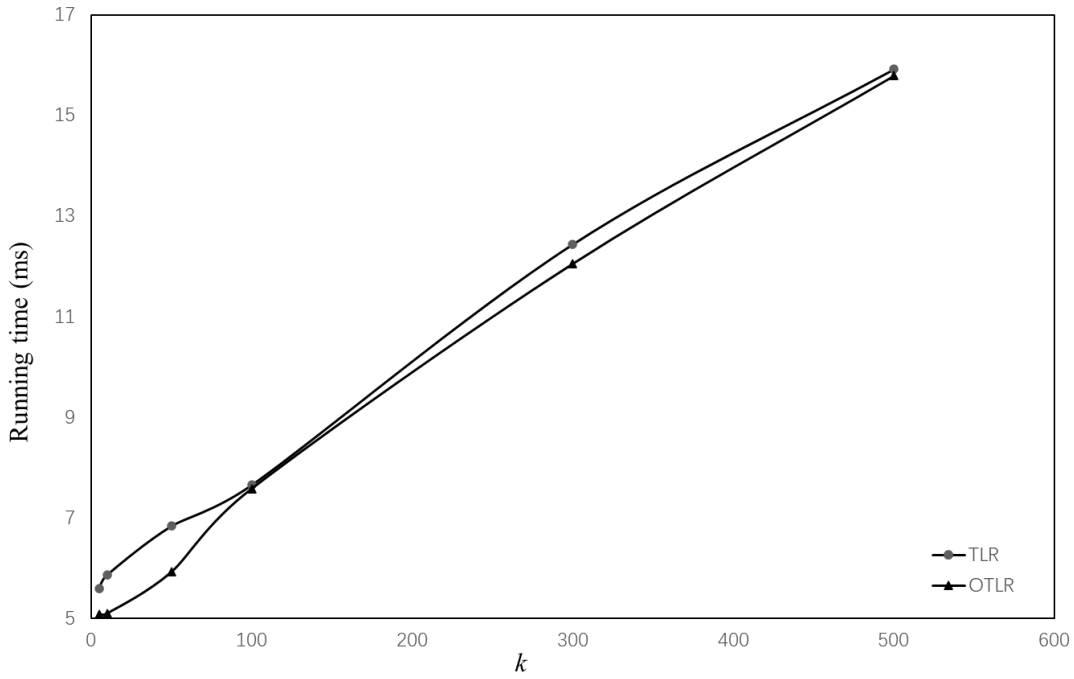


Figure 3: Effect of *k*

6.2.4 Varying m

Comparing with the effect of *k*, varying *m* has little impact on the computation time, which proves that repeated top-*k* operations are the most time-consuming processes. The default value of *k* is 50 and the number of timestamps is 30. With the fixed number of candidates, as *m* increases, the trend tends to flatten out.

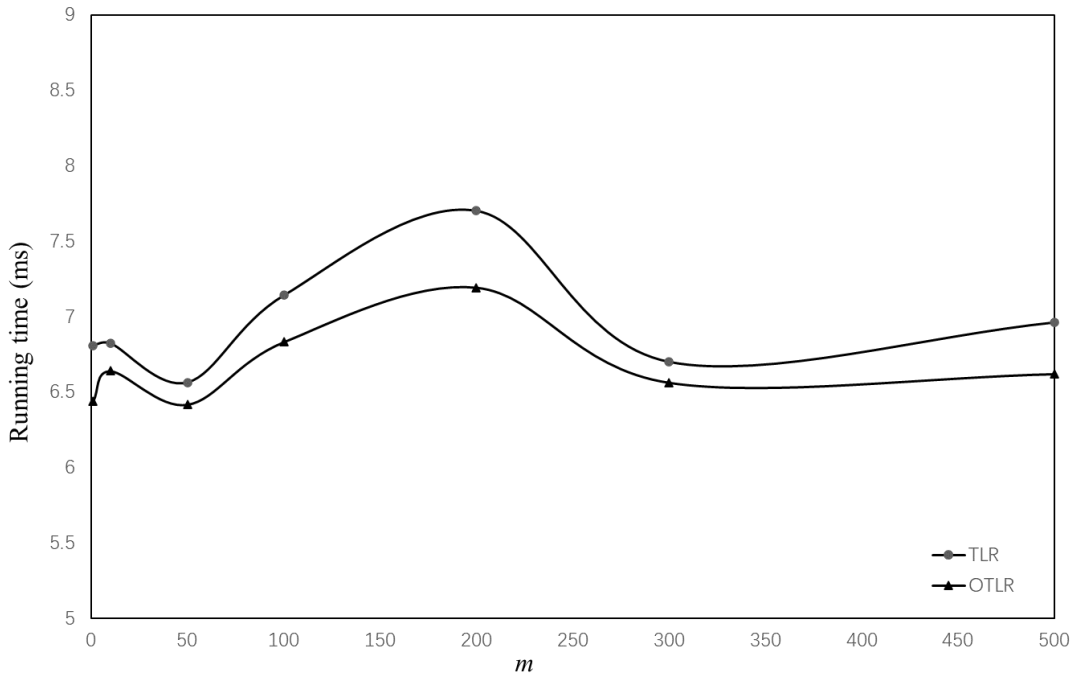


Figure 4: Effect of *m*

6.2.5 Varying number of timestamps

More timestamps is essentially the same as larger cardinality, i.e., *n*, because an R-tree is built on the data of each timestamp. More timestamps mean more R-trees to traverse. As there are more timestamps, the difference between the running time of TLR and OTLR increases. Since TLR runs a top-*k* query for every timestamp anyway, more timestamps naturally implies greater gains for OTLR from the avoidance of more unnecessary top-*k* computations.

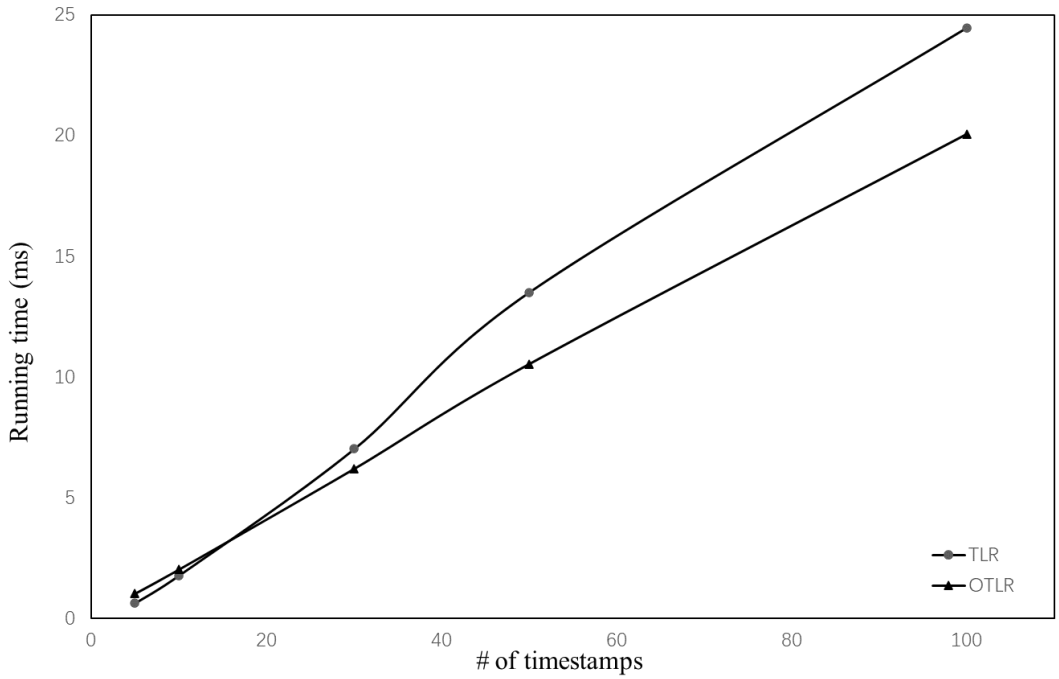


Figure 5: Effect of # of timestamps

6.2.6 Experiments with NBA dataset

As elaborated in Section 5, the data size for each timestamp and the number of dimensions are fixed. The default values for the input parameters are k = 50, m =10, and the number of timestamps equals 30. In each experiment, we fix two parameters to their default values and vary the value of the third one. Since only the data from 1973 to 2021 are complete, the number of timestamps is at most 49.

The experimental results below demonstrate that OTLR is more efficient than TLR. As shown in Figure 6, the running time of TLR grows faster than OTLR with k, because a larger k raises the time cost of top-k lists maintenance for TLR. In Figure 7, similar to our previous experiments in Figure 4, the running time of both TLR and OTLR is insignificantly affected by m. Comparing Figures 8 and 5, the difference

between the running time of TLR and OTLR increases much faster for NBA dataset. That is because for real-life dataset, OTLR has a high probability of meeting the early termination condition when there is a continuity in the position of the tuples in consecutive timestamps.

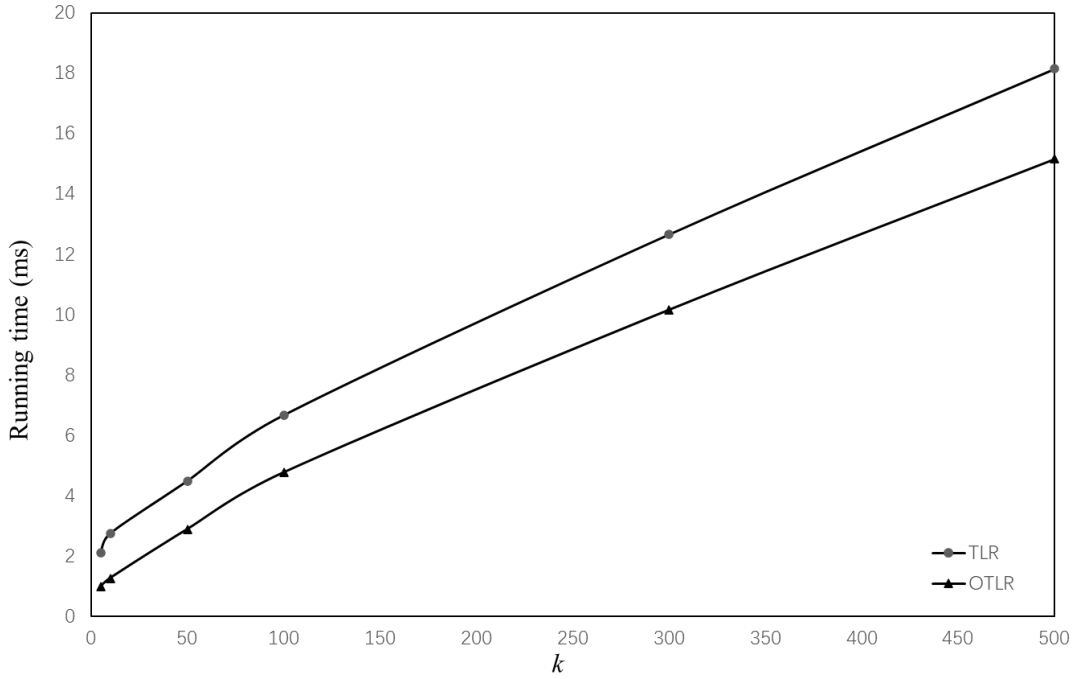


Figure 6: Effect of *k* (NBA)

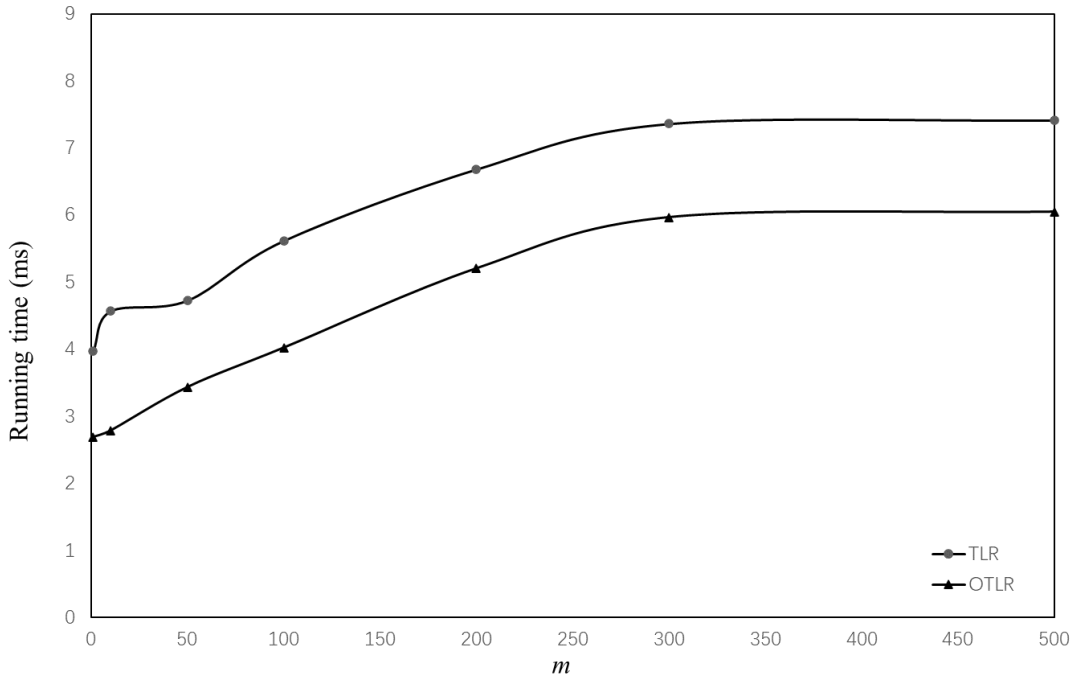


Figure 7: Effect of *m* (NBA)

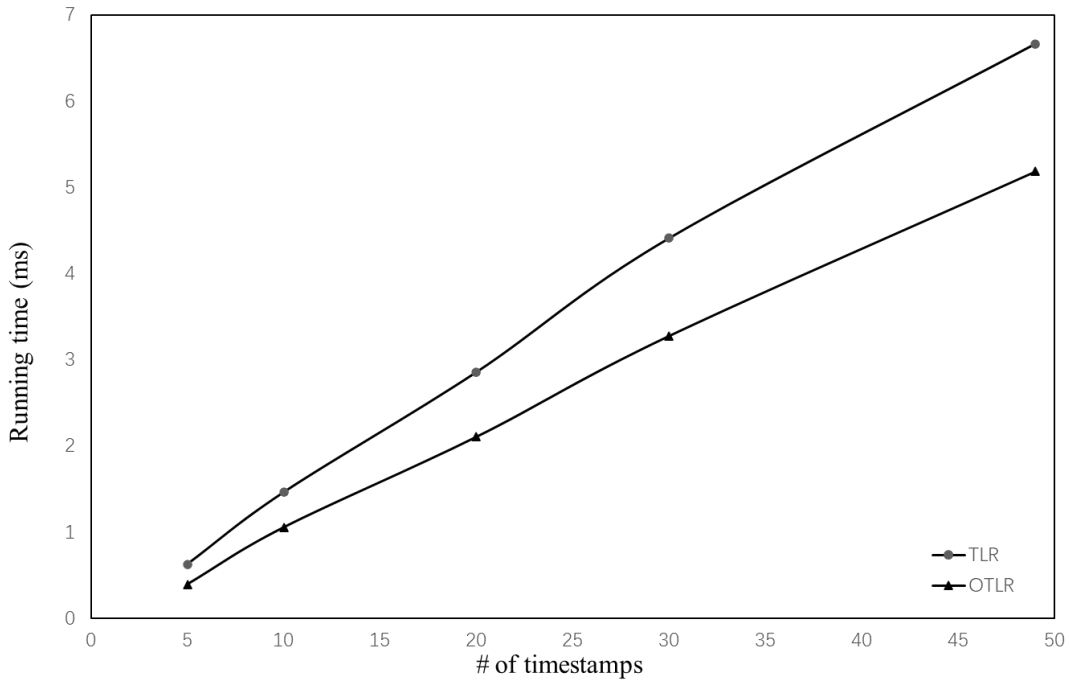


Figure 8: Effect of # of timestamps (NBA)

7. FUTURE WORK

The methods presented in this thesis perform numerous top-*k* computations, namely for every and for many, respectively, timestamps. In addition to taking considerable time, that also creates many unnecessary elements in the hash table. A direction for future work is to improve performance by sharing computations among repetitive top-computations and passing information from one top-*k* query to another. Such considerations are worth a think-through in future research.

References

[1] H. Wang, Y. Cai, Y. Yang, S. Zhang, and N. Mamoulis. Durable queries over historical time series. *TKDE*, 26(3):595–607, 2014.

[2] H. Leong, N. Mamoulis, K. Berberich, and S. Bedathur. Durable top-k search in document archives. In *Proc. of the ACM SIGMOD Conf*., 2010.

[3] J. Gao, P. K. Agarwal, and J. Yang. Durable top-k queries on temporal data. *PVLDB*, 11(13), 2018.

[4] F. Li, K. Yi, and W. Le. Top-k queries on temporal data. In *VLDBJ*, 2010.

[5] J. Gao, S. Sintos, P. K.Agarwal, and J. Yang. Durable top-k instant-stamped temporal records with user-specified scoring functions. In *ICDE*, 2021.

[6] K. Mouratidis, S. Bakiras, and D. Papadias. Continuous monitoring of top-k queries over sliding windows. In *SIGMOD*, 2006.

[7] S.-L. Lee, S.-J. Chun, D.-H. Kim, J.-H. Lee, and C.-W. Chung. Similarity Search for Multidimensional Data Sequences. In *Proceedings of ICDE*, pages 599–608, 2000.

[8] M. Vlachos, G. Kollios, and D. Gunopulos. Discovering similar multidimensional trajectories. In *Proc. ICDE*, 2002.

[9] L. Chen, M. T. Ozsu, and V. Oria. Robust and fast similarity search for moving object trajectories. In *SIGMOD Conference*, 2005.

[10] E. Laftchiev and Y. Liu. Finding multidimensional patterns in multidimensional time series. In *KDD Workshop on MiLeTS*, 2018.

[11] M. Jones, D. Nikovski, M. Imamura, and T. Hirata. Anomaly detection in real-valued multidimensional time series. In *International Conference on Bigdata/Socialcom/Cybersecurity*, 2014.

[12] A. McGovern, D. Rosendahl, R. Brown, and K. Droegemeier. Identifying predictive multi-dimensional time series motifs: an application to severe weather prediction. *DMKD, 22*, 2011.

[13] H. Jiang, J. Cheng, D. Wang, C. Wang, and G. Tan. Continuous multidimensional top-k query processing in sensor networks. In *Proceedings of IEEE INFOCOM*, 2011.

[14] N. Li, S. Du, H. Zheng, M. Xue and H. Zhu. Fake reviews tell no tales? dissecting click farming in content-generated social networks. *China Communications*, 2018.

[15] Philippe Esling and Carlos Agon. Time-series data mining. *ACM Computing Surveys (CSUR),* 2012.

[16] Chotirat Ann Ralanamahatana, Jessica Lin, Dimitrios Gunopulos, Eamonn Keogh, Michail Vlachos, and Gautam Das. Mining time series data. In *Data mining and knowledge discovery handbook*. Springer, 1069–1103, 2005.

[17] J. Paparrizos, C. Liu, A. J. Elmore, and M. J. Franklin. Debunking four long-standing misconceptions of time-series distance measures. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1887–1905, 2020.

[18] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[19] Tao, Y., Hristidis, V., Papadias, D., and Papakonstantinou, Y. 2007. Branch-and-bound processing of ranked queries. *Information Systems* 32, 3, 424–445.

[20] A. Guttman, R-trees: a dynamic index structure for spatial searching, *ACM SIGMOD*, 1984.

[21] N. Beckmann, H. Kriegel, R. Schneider, B. Seeger, The R*- tree: an efficient and robust access method for points and rectangles, *ACM SIGMOD*, 1990.

[22] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? *International Conference on Database Theory*, 1999.