Optimizing multi-top-k queries over uncertain data streams

Tao Chen, Lei Chen, *Member, IEEE,* M. Tamer Özsu, *Fellow, IEEE,* and Nong Xiao

Abstract—Query processing over uncertain data streams, in particular top-k query processing, has become increasingly important due to its wide application in many fields such as sensor network monitoring and internet traffic control. In many real applications, multiple top-k queries are registered in the system. Sharing the results of these queries is a key factor in saving the computation cost and providing real time response. However, due to the complex semantics of uncertain top-k query processing, it is nontrivial to implement sharing among different top-k queries and few works have addressed the sharing issue. In this paper, we formulate various types of sharing among multiple top-k queries over uncertain data streams based on the frequency upper bound of each top-k query. We present an optimal dynamic programming solution as well as a more efficient (in terms of time and space complexity) greedy algorithm to compute the execution plan of executing queries for saving the computation cost between them. Experiments have demonstrated that the greedy algorithm can find the optimal solution in most cases, and it can almost achieve the same performance (in terms of latency and throughput) as the dynamic programming approach.

Index Terms—data streams, uncertain data streams, top-k query, multi-query optimization.

1 Introduction

There are many applications in which data naturally occur in the form of a sequence of values, such as sensor data, financial tickers, on-line auctions, Internet traffic, web usage logs, and telephone call records [12], [13]. These data can be modeled as data streams, which are unbounded data sets produced incrementally over time [6]. Often, due to possible errors caused by limitations of monitoring equipment, human operator mistakes, and interference in data transfer, these data may be incomplete, unreliable, or noisy, with the result that uncertainty is inherent in these data stream applications. Much work has focused on uncertain data streams [10], [18], [2], [20], [16], and the semantics of possible worlds has been widely adopted in dealing with them [25], [26], [17], [28], [9], [22]. A possible world is a possible instance combination of tuples [25], usually within a sliding window. For a given timestamp, the tuples in the sliding window result in a number of possible worlds, whose number exponentially increases with the number of tuples in the sliding window.

The applications listed above typically issue a large number of monitoring queries. These are queries that are registered to the system, and they are executed periodically. An important sub-class of these queries are top-k queries. For example traffic monitoring applications typically wish to determine the top-k speeds of cars that pass through a control point, and volcano monitoring applications monitor top-k readings from sensors that produce uncertain data streams [20], [16]. We discuss traffic monitoring in more detail below. Previous work has considered executing these queries one-at-a-time, but there are considerable benefits to handling them collectively by exploiting similarities. This is the well known multi-query optimization problem, which is known to be very hard in relational DBMSs. This problem is particularly difficult over uncertain data streams.

Consider the car traffic monitoring application mentioned above where sensors are used at a monitoring point to detect speeds of cars. There are three sensors deployed at the monitoring point. Sensor S_0 is activated alone, and sensors S_1 and S_2 are activated at the same time. These sensors send the speeds of cars every 5 minutes. Due to external factors, the speed readings may be inaccurate, or may be correct only with some probability. Table 1 shows sample uncertain data from 9:01 AM to 9:20 AM. We assume that the sliding window size is 20 minutes, which includes six speed readings, each with a certain probability; e.g., it records a speed of 80 km per hour at 9:05 with a probability of 0.3, and so on. There may be dependencies among the speed readings such as, for example, R2 and R3 cannot be true at the same time, and neither can R5 and R6. These are called *generation rules* and denoted as $GR_1 = \{R2, R3\}$ and $GR_2 = \{R5, R6\}$. Traffic controllers would be interested in knowing, for example, the top-3 speed readings in the last 20 minutes. Thus, a continuous query would be run periodically to compute the results. We use CQL, an expressive SQL-based declarative language for registering continuous queries against streams and updatable relations [3] to describe this continuous query:

```
Q1:  SELECT      top-3
     FROM        SENSOR [Range 20 mins]
     WHERE       SENSOR.location=X
     FREQUENCY   2 mins
```

The semantics of FREQUENCY is similar to SLIDE semantics, which is the upper bound specification [11]. FREQUENCY specifies the upper bounds on the re-execution intervals of their queries. That is, the interval between two consecutive executions is no more than the upper bound.

- *E-mail: taochen@nudt.edu.cn*
- *E-mail: leichen@cse.ust.hk*
- *E-mail: tamer.ozsu@uwaterloo.ca*
- *E-mail: nongxiao@nudt.edu.cn*

TABLE 1

The uncertain data set in one sliding window

ID	Time	Speed(km)	Prob.	Sensor
R1	09:05:00 AM	80	0.3	S_0
R2	09:10:00 AM	65	0.4	S_1
R3	09:10:00 AM	45	0.5	S_2
R4	09:15:00 AM	30	1	S_0
R5	09:20:00 AM	50	0.8	S_1
R6	09:20:00 AM	25	0.2	S_2

TABLE 2

The possible worlds of uncertain data set

Possible world	Prob.	Top-2	Top-3
W1=R1,R2,R4,R5	0.096	R1,R2	R1,R2,R5
W2=R1,R2,R4,R6	0.024	R1,R2	R1,R2,R4
W3=R1,R3,R4,R5	0.12	R1,R5	R1,R5,R3
W4=R1,R3,R4,R6	0.03	R1,R3	R1,R3,R4
W5=R1,R4,R5	0.024	R1,R5	R1,R4,R5
W6=R1,R4,R6	0.006	R1,R4	R1,R4,R6
W7=R2,R4,R5	0.224	R2,R5	R2,R4,R5
W8=R2,R4,R6	0.056	R2,R4	R2,R4,R6
W9=R3,R4,R5	0.28	R5,R3	R3,R4,R5
W10=R3,R4,R6	0.07	R3,R4	R3,R4,R6
W11=R4,R5	0.056	R5,R4	R4,R5
W12=R4,R6	0.014	R4,R6	R4,R6

TABLE 3

The top-2 & top-3 probability of each tuple in one sliding window

ID	Top-2 Prob.	Top-3 Prob.
R1	0.3	0.3
R2	**0.4**	0.4
R3	0.38	**0.5**
R4	0.202	**0.784**
R5	**0.704**	**0.8**
R6	0.014	0.173

To answer top-k queries over an uncertain data stream, the semantics of possible worlds is applied. Table 2 shows the possible worlds in the above example. As the sum of the probabilities of R2 and R3 is less than 1, there are three choices for R2 and R3. There are two choices for R5 and R6 since the sum of the probabilities of them is equal to 1. There are two choices for R1. There is only one choice for R4. Thus, there are 2*3*1*2=12 possible words. The probability of the possible world W12, for example, is $(1-0.3)*(1-0.4-0.5)*1*0.2 = 0.014$. Answering the query, "What are the top-3 speed readings in the last 20 minutes?" requires searching all the possible worlds listed in Table II to determine three highest speed readings, each of them is associated with an aggregate probability derived from the possible worlds that it ranks as one of the top-3 readings. The top-k probability of a tuple in the sliding window is the sum of the probabilities of the possible worlds containing the tuple as one of the top-k ranked tuples. A possible answer to the above query can be computed by the $Pk-topk$ algorithm [20], which returns k tuples in the sliding window that have the highest top-k probability among all tuples. In this example, $Pk-topk$ computes the top-3 probability of each tuple as shown in the third column of Table 3. For example, the Top-3 probability of R1 is the sum of the probabilities of the possible worlds W1, W2, W3, W4, W5 and W6 which contain R1 as one of the top-3 ranked tuples. Thus, the Top-3 probability of R1 is $0.096+0.024+0.12+0.03+0.024+0.006 = 0.3$. The results of Q1 are R3, R4, and R5 which have the highest top-3 probability among all; their probabilities are highlighted with bold font in the third column of Table 3. The second column of Table 3 shows the answers to Q2, R2 and R5.

```
Q2:  SELECT      top-2
     FROM        SENSOR [Range 20 mins]
     WHERE       SENSOR.location=X
     FREQUENCY   2 mins
```

Due to complex possible worlds, the top-2 probability of each speed reading is not the same as its top-3 probability; therefore, the results of Q2 are not the largest two readings of the results of Q1. In other words, we cannot obtain the results of the top-2 query directly from the results of the top-3 query. How to reuse results of a top-k query to compute results of another top-k' query where $k \geq k'$ is challenging. This is the problem that we address in this paper. Among the monitoring queries registered in the system, there are likely to be many with different frequencies and with different k values. In addition to Q1 and Q2 discussed above, consider the following monitoring queries:

```
Q3:  SELECT      top-4
     FROM        SENSOR [Range 20 mins]
     WHERE       SENSOR.location=X
     FREQUENCY   3 mins
Q4:  SELECT      top-3
     FROM        SENSOR [Range 20 mins]
     WHERE       SENSOR.location=X
     FREQUENCY   5 mins
Q5:  SELECT      top-5
     FROM        SENSOR [Range 20 mins]
     WHERE       SENSOR.location=X
     FREQUENCY   5 mins
Q6:  SELECT      top-2
     FROM        SENSOR [Range 20 mins]
     WHERE       SENSOR.location=X
     FREQUENCY   7 mins
```

Our objective is to find the solution which maximizes the sharing among queries Q1 to Q6. The commonalities between these queries fall into three categories: some have the same frequency, but different k values ($\{$Q1, Q2$\}$ and $\{$Q4, Q5$\}$), some have the same k values, but have different frequencies ($\{$Q1, Q4$\}$), others have different k values and different frequencies ($\{$Q2, Q3$\}$ and $\{$Q5, Q6$\}$). We have already argued that queries in the first category require care in sharing results. Queries in the second category can share results at times that are multiples of the frequencies; for example, Q1 and Q4 can share the result when they are executed at time $t = 10$ and so on; in fact only one of them needs to be executed at $t = 10$. When Q1 and Q4 are executed according to their frequency upper bounds, the execution time of Q1 and Q4 are shown in Figure 1. However, this level of sharing is not efficient and can be improved. For example, if Q4 can be refreshed every two minutes, the results of Q4 are returned whenever Q1 is executed; we do not need to execute Q4 individually any more. In this situation, the execution time of Q1 and Q4 are shown in Figure 2. Thus, we could improve sharing when queries are refreshed more often than specified. Queries in the third category cannot share any results with each other. These are the sharing problems studied in this paper. Note, again, that the semantics of FREQUENCY is an upper bound specification, similar to SLIDE semantics [11], which ensures that it is possible to re-execute queries more often when the system is lightly loaded, as well as enabling queries with different FREQUENCY to be executed simultaneously to share computation. Thus, a query whose frequency is specified as three minutes can be executed every one minute, every two minutes, or as long as the interval between two executions is no more than three minutes.

To address this problem, we first consider how to share the results of queries with the same frequency, but different

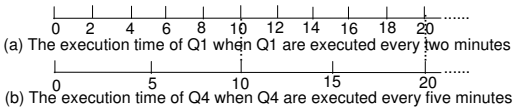


Fig. 1. The execution time of Q1 and Q4 with their frequency upper bounds

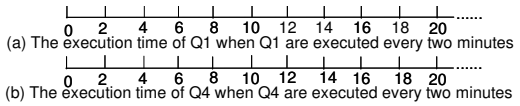


Fig. 2. The execution time of Q1 and Q4 when Q4 is executed every 2 minutes

k values. That is, some intermediate results of the queries in the first category can be shared. For the second category, we exploit the FREQUENCY semantics discussed above to execute the queries more often if their execution time can be synchronized with those queries that have the same k but lower frequency. When we execute queries with lower frequency, results of queries with higher frequency can also be returned. There is almost no additional overhead when sharing between the queries in these two categories, and the queries significantly benefit from joint execution. For the third category, we also exploit the FREQUENCY semantics to share computation. When the queries with higher frequency are executed more often, additional overhead may be incurred since the k values are different. The additional overhead might be more than the savings in computation time if queries are refreshed too often. Therefore, we need a strategy to find an optimal running schedule (execution plan) for these queries to balance the additional overhead and the savings in computation time. In this paper, we report an optimal dynamic programming solution and a faster greedy algorithm. The theoretical analysis and experimental results confirm that our proposed solutions can significantly reduce the computation cost through sharing.

The rest of the paper is organized as follows. We review the related work and state the differences from our approaches in Section 2. In Section 3, we discuss how *Pk-topk* can be used to process a single query. In Section 4, we solve the sharing problem for queries with the same frequency upper bounds but with different k values. We propose a combination rule to combine the groups with different frequency upper bounds without additional overheads in Section 5. We formalize the sharing problem for multiple top-k queries after combination in Section 6. Then, we propose an optimal dynamic programming solution (Section 7) and a faster greedy algorithm (Section 8) to solve the sharing problems. We show how our algorithms can be extended to multiple data streams in Section 9. Experimental evaluations are given in Section 10. Finally, we conclude in Section 11.

2 Related Work

Top-k queries over uncertain data have received recent attention. Soliman et al. [25] first presented U-top*k* and U-*k*Ranks to compute top-*k* tuples over uncertain data. A U-top*k* query returns the vector of *k* tuples that has the maximum probability in all possible worlds. A U-*k*Ranks query returns *k* tuples in which the *i*-th tuple ranks as the largest i in all possible worlds where $1 \leq i \leq k$. Hua et al. [17] proposed PT-k query, a probabilistic threshold top-k query, which returns all the tuples whose probability of being in the top-k is greater than a threshold p. Expected rank method [9] first defined the rank of a tuple in each possible world, and then computed the expected rank of each tuple across all possible worlds. It returned the k best tuples according to the expected rank. Zhang et al. [28] defined the Global-Top*k* probability of a tuple as the sum of the probabilities of all possible worlds whose top-*k* answer contains this tuple, and then returned the *k* tuples with the highest Global-Top*k* probability. Li et al. [22] proposed a unified approach to ranking in probabilistic databases, which defines a unified parameterized ranking function (PRF) to compute the total ordering of all the tuples and selects the k best tuples under this ordering. All of the above works consider only individual top-k query processing over uncertain data.

There are many works focusing on uncertain data streams. However, most of them concentrate on computing statistical aggregates and clustering data [10], [18], [2]. The first work on top-k queries over uncertain data stream was presented by Jin et al. [20], which used P*k*-Top*k* definition to rank tuples by their probability of being the top-*k* among all possible worlds and returned exactly k tuples. Hua et al. [16] presented a probabilistic threshold method to continuously monitor top-*k* uncertain data streams. However, neither of these works are applicable to solving the sharing problems of multiple top-k queries over uncertain data streams.

For multiple queries over certain data, traditional methods target detecting common parts across multiple queries [24], which is the base solution for multiple top-k queries; that is, identifying similar top-k queries. There are also some works that focus on multi-query optimization over data streams. However, many of them concentrate on sharing execution of filters, joins, selections, and aggregations [14], [23], [4], [21], [27], and none of them can be applied to multiple top-k query processing over uncertain data streams due to the complex semantics of uncertainty.

Krishnamurthy et al. [21] present a sharing scheme for aggregation over data streams. This is the first paper to deal with two kinds of variations for sharing aggregates. It chops the input stream into time slices for different windows, and breaks the input data into disjoint sets of tuples for different predicates. In our problem, the top-k probability of each tuple is associated with the possible worlds which is related with all the tuples in a sliding window. Thus, the way to break input streams and input data in [21] cannot obtain the correct top-k result using the semantics of uncertainty.

Our problem is also different from real-time scheduling problems for the following two reasons. First, each task in real-time scheduling has fixed deadlines which must be met [7], [1], [15]. The latest work for periodic real-time scheduling also makes the assumption of deadline periodic tasks [7]. For our problem, each query has a frequency upper bound which is more flexible and complex, but no fixed deadlines. We must ensure that the interval between two consecutive executions is no more than the frequency upper bound. Second, the goal of real-time scheduling is to meet the deadlines of tasks or to minimize the number of late tasks. Our goal is to share the

computation among queries as much as possible and satisfy the frequency semantics at the same time.

Recent work on scheduling in DSMSs presented scheduling methods at the level of tuples and operators [5], [8], [19]. They choose which tuples or which operators to process at any given time. The goals are to reduce output latency or the sizes of inter-operator queues. We extend SLIDE semantics to FREQUENCY semantics. SLIDE semantics means the query can be executed at a frequency which is no more than SLIDE, while FREQUENCY means the interval between two consecutive executions is no more than FREQUENCY which is more flexible, but also more complex.

3 Processing a Single Query

We first discuss how to process a single query, which provides basis for further discussion of sharing and query optimization.

Let $\mathcal{T}$ be an uncertain stream containing a sequence of tuples T_i, each of which is associated with a membership probability $Pr(T_i)$. We use a time-based sliding window over $\mathcal{T}$. Suppose the current window is S with size W. A possible world pw from S is a possible combination of a set of tuples with probability $Pr(pw)$. When tuples are independent without any generation rules, we have $Pr(pw) = \prod_{T_i \in pw} Pr(T_i) \prod_{T_i \notin pw}(1 - Pr(T_i))$. We focus on the case where the tuples are dependent with generation rules.

Definition 3.1: A generation rule is a set of dependent tuples, $\mathcal{R}$ where each element $\nabla \in \mathcal{R}$ is a set of dependent tuples.

For a tuple T_i which is not involved in any generation rule, we can make up a generation rule with only one tuple T_i. Then, we can easily get $Pr(pw) = \prod_{\nabla \in \mathcal{R}, \nabla \cap pw = T_i} Pr(T_i) \prod_{\nabla \in \mathcal{R}, \nabla \cap pw = \varnothing}(1 - \Sigma_{T_i \in \nabla} Pr(T_i))$.

Definition 3.2: Given a ranking function $f : T_i \rightarrow \mathbb{R}$, the top-$k$ results of a possible world *pw*, denoted as $Top_k(pw)$, are the k tuples who rank highest in *pw*. The top-*k* probability of a tuple T_i is the sum of the probabilities of all possible worlds in which T_i belongs to the top-k answer, $Pr_k(T_i) = \sum_{T_i \in Top_k(pw)} Pr(pw)$.

First, we assume the tuples are independent without any generation rules. We rank the tuples in the sliding window S in descending order according to the ranking function f. $Pr_k(T_i)$ can be computed by the product of the probability of T_i and the probability that no more than $k-1$ tuples rank higher than T_i. Let $Pr^+(T_i, j)$ be the probability that there are j tuples that rank higher than T_i, and $Pr^-(T_i, k-1)$ the probability that no more than $k-1$ tuples rank higher than T_i in S. Then $Pr^-(T_i, k-1) = \Sigma_{j=0}^{k-1} Pr^+(T_i, j)$. Thus, the top-*k* probability of T_i is

$$Pr_k(T_i) = Pr(T_i) * Pr^-(T_i, k-1) = Pr(T_i) * \Sigma_{j=0}^{k-1} Pr^+(T_i, j). \quad (1)$$

For $T_1 \in S$, there is no tuple that ranks higher than T_1, thus, $Pr^+(T_1, 0) = 1$. For tuple $T_2 \in S$, $Pr^+(T_2, 0) = 1 - Pr(T_1)$, and $Pr^+(T_2, 1) = Pr(T_1)$. Generally, for tuple $T_i \in S$ where $1 \leq i \leq n$, $Pr^+(T_i, j)$ is:

$$Pr^+(T_i, j) = \begin{cases} Pr^+(T_{i-1}, j) * (1 - Pr(T_{i-1})) & \text{if } j = 0 \\ Pr^+(T_{i-1}, j-1) * Pr(T_{i-1}) + Pr^+(T_{i-1}, j) & \\ *(1 - Pr(T_{i-1})) \quad \text{if } j < i-1 \text{ and } j \leq k-1 & \\ Pr^+(T_{i-1}, j-1) * Pr(T_{i-1}) \text{ if } j = i-1 \text{ and } j \leq k-1 & \end{cases} \quad (2)$$

After computing all the $Pr^+(T_i, j)$ values, the top-*k* probability of T_i can be computed as $Pr_k(T_i) = Pr(T_i) * \Sigma_{j=0}^{k-1} Pr^+(T_i, j)$.

Theorem 3.1: It costs $O(n * k)$ to compute the result of a top-k query when the tuples are independent without any generation rules.

Now we consider the generation rules between tuples. That is, some tuples cannot appear in a possible world at the same time and these tuples are dependent. We convert the computation of the top-k probability between dependent tuples to the computation of the top-k probability between independent tuples. Then, the sharing computation can also be applied to the situation with generation rules. Recall that the set of generation rules are $\mathcal{R}$ where each element $\nabla \in \mathcal{R}$ is a set of dependent tuples. For each T_i, we compute the set of tuples that are independent of T_i. If T_i ranks higher than all the tuples in ∇, ∇ can be ignored. For each $\nabla \in \mathcal{R}$, if T_i ranks lower than all the tuples in ∇, we combine all tuples in ∇ as a single tuple with the probability $Pr(\nabla) = \sum_{T_j \in \nabla} Pr(T_j)$. If T_i ranks between the tuples in ∇, we combine the tuples in ∇ which rank higher than T_i and ignore the remaining tuples in ∇. After scanning all the generation rules in $\mathcal{R}$, the remaining tuples are independent of T_i. Then, we compute top-k probability of T_i according to equations (1) and (2). Therefore, it costs $O(n * k)$ to compute the top-k probability of T_i. The top-k probability of all the tuples can be computed in $O(n^2 * k)$.

Theorem 3.2: It costs $O(n^2 * k)$ to compute the result of a top-k query when the tuples are dependent with generation rules.

4 Sharing among Queries with Same Frequency Upper Bound

As mentioned in the introduction, there are three types of queries with sharing possibilities. In this section, we consider queries in the first category. The second and third categories, which are the most common, are discussed in Sections 5 and 6. Table 4 outlines the major notations used in this paper.

TABLE 4
Notations

G, F, K, x	groups of queries, the corresponding set of frequencies, the set of k values, and the number of queries in G
$\mathcal{G}$,$\mathcal{F}$,$\mathcal{K}$,g	the group set of queries, the corresponding set of frequencies, the set of k values, and the number of groups
G_i, f_i, k_i^{max}	the id, and the frequency of group i in $\mathcal{G}$, and the largest k value in group i where $1 \leq i \leq g$
t_i^1	the time that G_i is first executed
EP, EP_i	an execution plan for $\mathcal{G}$ and an execution plan for the group G_i in $\mathcal{G}$
$Gcost(t)$	the cost of executing $\mathcal{G}$ in [0, t] for an arbitrary plan
$DPcost(i,t)$	the minimum cost of executing the first i groups in [0, t] for the dynamic programming algorithm where $1 \leq i \leq g$.
$GAcost(t)$	the cost of executing $\mathcal{G}$ in [0, t] for the greedy algorithm
$s(i,t)$	the last time that we execute G_i in [0, t] for the dynamic programming algorithm
$e(t)$	the group number in the first i groups executed at time t for the greedy algorithm

We first start with the formalization of the problem. We divide the set of queries Q into r groups $G = \{G_1, ..., G_r\}$ according to the frequency upper bounds of queries so that queries with the same frequency upper bound are in the same

group. Let G_i denote a group whose queries have the same frequency bound, f_i. Each G_i is sorted in ascending order based on f_i. Let k_i^{max} denote the largest k of queries in G_i. After grouping, we have *intra-group sharing* and *inter-group sharing*. The first category (i.e., queries with the same frequency upper bound but different k values) represents intra-group sharing, while the second and the third categories represent inter-group sharing.

As discussed in Section 1, the results of a top-k' query cannot be derived directly from that of another top-k query even when $k \geq k'$. However, we can reuse the intermediate results of a top-k query to compute answers for a top-k' query when $k \geq k'$.

Now let us consider another top-k' query in G_i ($k' < k$) with the same frequency upper bound. The top-k' probability of T_i is $Pr_{k'}(T_i) = Pr(T_i) * \Sigma_{j=0}^{k'-1} Pr^+(T_i, j)$. Since $k' < k$, we can find that $Pr^+(T_i, j)$ $(0 \leq j \leq k'-1)$ is already computed when we compute the answer for the top-k query. Thus, the top-k' probability of each tuple can be derived when we compute the top-k probability of each tuple.

For example, suppose the tuples in a sliding window over an uncertain data stream after ranking are $\{T_1, T_2, T_3, T_4, T_5\}$ with the corresponding probabilities $\{0.7, 0.2, 1, 0.3, 0.5\}$. To compute the top-3 probability of each tuple, we first initialize $Pr^+(T_1, 0) = 1$. According to equation (2), we have $Pr^+(T_2, 0) = Pr^+(T_1, 0) * (1 - Pr(T_1)) = 1 * (1 - 0.7) = 0.3$, $Pr^+(T_2, 1) = Pr^+(T_1, 0) * Pr(T_1) = 1 * 0.7 = 0.7$, $Pr^+(T_3, 0) = Pr^+(T_2, 0) * (1 - Pr(T_2)) = 0.3 * (1 - 0.2) = 0.24$, $Pr^+(T_3, 1) = Pr^+(T_2, 0) * Pr(T_2) + Pr^+(T_2, 1) * (1 - Pr(T_2)) = 0.3 * 0.2 + 0.7 * 0.8 = 0.62$ and $Pr^+(T_3, 2) = Pr^+(T_2, 1) * Pr(T_2) = 0.7 * 0.2 = 0.14$. We get $Pr_3(T_3) = Pr(T_3) * (Pr^+(T_3, 0) + Pr^+(T_3, 1) + Pr^+(T_3, 2))$. If we also want to compute the top-2 probability of T_3, we can get it easily: $Pr_2(T_3) = Pr(T_3) * (Pr^+(T_3, 0) + Pr^+(T_3, 1))$ since $Pr^+(T_3, 0)$ and $Pr^+(T_3, 1)$ are already computed in the computation of the top-3 probability.

Theorem 4.1: It costs $O(n * k)$ to compute the results of both top-k query and top-k' query when $k \geq k'$ without any generation rules.

Now we consider the generation rules between tuples. For each T_i, we compute the set of tuples that are independent of T_i. After scanning all the generation rules in $\mathcal{R}$, the remaining tuples are independent of T_i. Then we compute the top-k probability of T_i according to equations (1) and (2). At the same time, we can also get the top-k' probability of T_i where $k \geq k'$ by sharing the intermediate computations. Therefore, it costs $O(n*k)$ to compute the top-k probability of T_i. The top-k probability of all the tuples can be computed in $O(n^2 * k)$.

Consider the uncertain table and the query Q1 in Section 1. Recall that we have two generation rules: $GR_1 = \{R2, R3\}$ and $GR_2 = \{R5, R6\}$. Let us compute $Pr_3(R4)$. $R4$ is ranked lower than all the tuples in GR_1. There is no more than one tuple which can appear in a possible world. We combine all the tuples in GR_1 into a combination tuple R_{GR_1} with probability $Pr(R_{GR_1}) = Pr(R2) + Pr(R3) = 0.4 + 0.5 = 0.9$. $R4$ is ranked between tuples in GR_2. There is only $R5$ in GR_1 which is ranked higher than $R4$. After the process, the independent tuples which are ranked higher than $R4$ is $\{R1, R_{GR_1}, R5\}$. Now, this problem is converted to the computation of the top-k probability between independent tuples. According to equations (1) and (2), we have $Pr_3(R4) = 0.784$. We can also compute $Pr_2(R4) = 0.202$ by reusing the intermediate results in the computation of its top-3 probability the same way as above.

Theorem 4.2: It costs $O(n^2 * k)$ to compute the results of both top-k query and top-k' query when $k \geq k'$ with generation rules.

5 Combination Rule to Combine Groups with Different Frequency Upper Bounds

According to Theorem 4.2, queries with same frequency upper bounds can share computation. For queries with different frequency upper bounds, our objective is to share computation among queries that have different frequency upper bounds, which include the second and the third types of queries described in Section 1. We can exploit the FREQUENCY semantics to execute some queries more often so that they can be synchronized with those queries that have a higher k but lower frequency. In this case, there is no overhead to execute queries more often based on the Theorem 4.2. Thus, we present the following combination rule to combine some groups with different frequency upper bounds.

Combination rule: As explained in Section 4, Q is divided into r groups $G = \{G_1, ..., G_r\}$ where $i < j$ implies $f_i < f_j$. If $k_i^{max} \geq k_j^{max}$, we set the frequency of G_j to f_i, and thus the results of G_j are returned when G_i is executed. G_i and G_j are combined into the same group with the frequency bound f_i and the largest k value k_i^{max} as defined in the previous section. This combination step will continue until there are no pair of groups (G_i, G_j) such that $k_i^{max} \geq k_j^{max}$ if $i < j$.

Lemma 5.1: Assume that the query groups after combination are $\mathcal{G} = \{G_1, ..., G_g\}$; for any two groups G_i, G_j in $\mathcal{G}$, if $i < j$, then we have $f_i < f_j$ and $k_i^{max} < k_j^{max}$.

Type 2 queries (same k value) are distributed to different groups (different frequency bounds), which can be processed together with type 3 queries since different groups have different frequency bounds.

Consider the example in Section 1 where there is a set $Q = \{q_1, q_2, ..., q_6\}$ of six top-k queries submitted by different users. Note that the unit time is a minute in this example. According to the frequency upper bound, we can divide Q into four subgroups: $G_1 = \{q_1, q_2\}$, $G_2 = \{q_3\}$, $G_3 = \{q_4, q_5\}$, and $G_4 = \{q_6\}$. That is, $G = \{G_1, G_2, G_3, G_4\}$, which is ranked in ascending order based on f_i. The corresponding set of frequency upper bounds is $\{f_1 = 2, f_2 = 3, f_3 = 5, f_4 = 7\}$, and the set of the largest k values is $\{k_1^{max} = 3, k_2^{max} = 4, k_3^{max} = 5, k_4^{max} = 2\}$. Since $k_3^{max} > k_4^{max}$, we set the frequency of G_4 to f_3. Then as shown in Section 4, the results of G_4 can be computed based on the intermediate results generated from G_3 when G_3 is executed. Thus, we combine G_3 and G_4 into the same group with f_3 and k_3^{max}.

After combination, the group set is $\mathcal{G} = \{G_1, G_2, G_3\}$, the corresponding set of frequency bounds is $\{f_1 = 2, f_2 = 3, f_3 = 5\}$, and the set of the largest k values is $\{k_1^{max} =$

$3, k_2^{max} = 4, k_3^{max} = 5\}$. In the rest of this paper, we name the combined query group $\{G1, G2, G3\}$, as **Query Example1**.

For each group after combination, we can use intra-group sharing described in Section 4. We denote this level of sharing after combination as INCO. That is, INCO includes three steps. The first step is the division step which divides the set of queries into groups according to the frequency upper bounds so that queries with the same frequency upper bound will be assigned to the same group; the second step is the combination step which combines some query groups with different frequency upper bounds based on Theorem 4.2; the third step is the intra-group sharing for each combined group.

6 SHARING AMONG QUERIES BETWEEN GROUPS AFTER COMBINATION

In this section, we address inter-group sharing for the combination groups. Sharing among these queries is achieved by selectively executing groups of queries with higher frequency together with groups of queries with lower frequency. In this case, the cost of executing selected groups depends on the largest k of the groups.

As mentioned in Section 4, the execution cost of each query depends on n (the number of tuples in a sliding window) and k. Without loss of generality, we omit parameter n in the cost computation. For **Query Example1**, if we execute them according to their original frequency upper bounds, the total cost in 5 minutes is $3*2+4+5 = 15$. If G_2 is executed every two minutes, G_1 and G_2 can be executed simultaneously. In 5 minutes, G_1 and G_2 are executed at times 2 and 4. The cost to execute G_1 and G_2 at time 2 is $k_2^{max} = 4$ since they can share the intermediate results according to Section 4. The total cost in 5 minutes is $4 * 2 + 5 = 13$. Thus, the total cost can be saved although G_2 is executed more frequently. This is the main idea of sharing the results among queries with different k and different frequencies f. Clearly, there are many alternative schedules for running the queries at different frequencies as long as they satisfy the frequency upper bound constraints. How to select an optimal one with less computation cost is the problem that we investigate here. In the following, we give the related definitions and define the problem of finding an optimal execution plan for a group of top-k queries.

Definition 6.1: The query cycle for the groups in $\mathcal{G}$ is the interval between two consecutive executions of G_g.

Note that by the time G_g is executed for the first time, all G_i $(1 \leq i < g)$ have executed multiple times since their frequency upper bounds are lower, hence they execute more frequently. Consequently, the execution plan of $\mathcal{G}$ can be defined as follows.

Definition 6.2: An execution plan for a set of query groups $\mathcal{G}$, $EP_{\mathcal{G}} = \{EP_1, ...EP_i, ..., EP_g\}$, indicates the execution time of each query group, EP_i, in each query cycle, where execution time of each query group $EP_i = \{t_i^1, t_i^2, ..., t_i^m, ..., t_i^{N_i}\}$ and $EP_g = \{t_g^1\}$, where t_g^1 is the first (and only) execution of G_g in a given cycle and $t_g^1 \leq f_g$, N_i is the number of times G_i is executed in a query cycle ($\lceil t_g^1/f_i \rceil$ is lower bound of N_i), $t_i^m - t_i^{m-1} \leq f_i$, and $t_g^1 - t_i^{N_i} \leq f_i$.

Note that we only need to consider execution plans in one query cycle. For **Query Example1**, we give an execution plan $EP = \{EP_1, EP_2, EP_3\}$ where $EP_1 = \{t_1^1 = 2, t_1^2 = 4, t_1^3 = 5\}, EP_2 = \{t_2^1 = 2, t_2^2 = 5\}, EP_3 = \{t_3^1 = 5\}$, as shown in Figure 3. That is, G_1 is executed at times 2, 4, and 5; G_2 is executed at times 2 and 5; and G_3 is executed at time 5. The interval between two consecutive executions of G_1 is no more than $f_1 = 2$, which is also the case for G_2 and G_3.

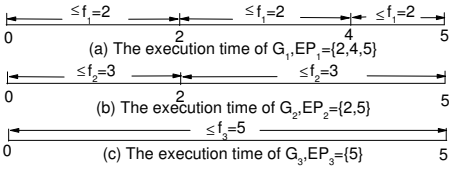


Fig. 3. EP_i in the execution plan $EP = \{\{2, 4, 5\}, \{2, 5\}, \{5\}\}$

Given an execution plan, its corresponding execution cost is defined as follows.

Definition 6.3: Given a set $\mathcal{G}$ of query groups and its execution plan $EP = \{EP_1, EP_2, ..., EP_g\}$, let t_g^1 be the time when G_g is first executed. We define $Gcost(t)$ as the total cost of executing queries in $\mathcal{G}$ in [0, t] ($t \leq t_g^1$). The execution cost of each query is its actual execution time. The cost-per-unit time is $Gcost(t_g^1)/t_g^1$.

$Gcost(t)$ can be computed as follows:

$$Gcost(t) = \begin{cases} Gcost(t-1) & \text{if there is} \\ & \text{no group executed at time } t \\ Gcost(t-1) + k_o & \text{if there is} \\ & \text{a group with the } k_o \text{ value executed at time } t \end{cases} \quad (3)$$

Different plans have different t_g^1 which result in different total cost $Gcost(t_g^1)$. Finding the optimal plan (in terms of computation cost) is a problem that we will address, which is formally defined as follows.

Definition 6.4: Given a set $\mathcal{G}$ of query groups, for $i = 1, 2, ..., g$, G_i has a frequency upper bound f_i and a top k value k_i^{max}, and $f_i < f_j$, $k_i^{max} < k_j^{max}$ if $i < j$. We define the sharing problem of $\mathcal{G}$ as finding an execution plan EP for $\mathcal{G}$ that minimizes the total execution cost-per-unit time.

To compare different plans, we use the cost-per-unit time as the metric. The plan with the minimum cost-per-unit time is considered the best plan. A naive method to find the best execution plan that minimizes the cost-per-unit time is to exhaustively enumerate all the possible plans, which is inefficient since there will be an exponential number of plans to investigate.

Given **Query Example1** and an execution plan $EP = \{EP_1, EP_2, EP_3\}$ where $EP_1 = \{2, 4, 5\}, EP_2 = \{2, 5\}, EP_3 = \{5\}$, for each time t less than five minutes, we compute $Gcost(g, t)$. As shown in Figure 3, no group is executed at time 1. We have $Gcost(1) = 0$. At time 2, G_1 and G_2 can be executed simultaneously. The execution cost at this time depends on k_2^{max} value of G_2 since $k_2^{max} > k_1^{max}$. Thus, $Gcost(2) = Gcost(1) + k_2^{max} = 4$. Similarly, at time 3, no group is executed, $Gcost(3) = Gcost(2)$, and at time 4, G_1 is executed, $Gcost(4) = Gcost(3) + k_1^{max} = 7$. Finally, G_3 is executed at time 5, and the results of all the groups are returned. The execution cost at this time depends on k_3^{max} value since k_3^{max} is the largest. Thus we

have $Gcost(5) = Gcost(4) + k_3^{max} = 12$. The cost-per-minute is $12/5 = 2.4$. Given another execution plan where $EP_1 = \{2, 4\}, EP_2 = \{2, 4\}, EP_3 = \{4\}$ as illustrated in Figure 4, we obtain the execution cost-per-minute $9/4 = 2.25$. After enumerating all possible plans, the most efficient plan can be found as $EP = \{\{2, 4\}, \{2, 4\}, 4\}$. That is, G_2 is executed at time 2, and the results of both G_1 and G_2 are returned, followed by G_3's execution at time 4, returning the results of G_1, G_2 and G_3. It is clear that enumerating all possible execution plans is inefficient, thus, we propose a dynamic programming-based approach in the next section.

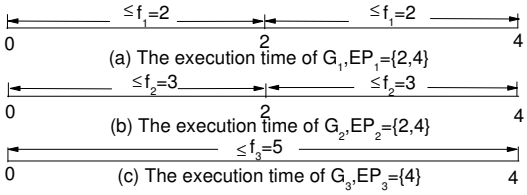


Fig. 4. EP_i in the execution plan $EP = \{\{2, 4\}, \{2, 4\}, 4\}$

7 Dynamic Programming Solution

In order to find the optimal execution of the groups that minimize the cost-per-unit time without enumerating all the possible plans, we analyze the execution of $\mathcal{G}$. Note that G_g has the first (and only) execution of $\mathcal{G}$ in a given cycle. That is, in the optimal execution, G_g has only one execution time, while other groups may have more than one execution time to satisfy the frequency upper bounds. Thus, we analyze the execution of G_g. There are f_g choices for first execution of G_g. Once we set the time t_g^1 for first execution of G_g, we need to find the optimal execution for the other groups $\{G_1, G_2, .., G_{g-1}\}$ in [0, t_g^1].

Let $DPcost(i, t)$ be the minimum cost of executing the first i groups in [0, t] where $1 \leq i \leq g$ and $0 \leq t \leq f_g - 1$. The problem here is to compute $DPcost(i, t)$ for the optimal execution of the first i groups in [0, t]. We propose a dynamic programming approach (denoted as DP) to compute $DPcost(i, t)$. When $i = g$ and $t = t_g^1 - 1$ where $1 \leq t_g^1 \leq f_g$, we get $DPcost(g, t_g^1 - 1)$. Clearly, there are f_g choices for t_g^1 and for each choice, the cost of executing all the groups in [0, t_g^1] is $DPcost(g, t_g^1) = DPcost(g, t_g^1 - 1) + k_g^{max}$. The cost-per-unit time is $DPcost(g, t_g^1)/t_g^1$. Thus, an execution plan that minimizes $(DPcost(g, t_g^1 - 1) + k_g^{max})/t_g^1$ among all the possible t_g^1 values is the best plan.

We compute $DPcost(i, t)$ recursively. When $i = 1$, the optimal execution of G_1 is to execute it every f_1 time units. Thus, we have

$$DPcost(1, t) = (\lfloor t/f_1 \rfloor) * k_1^{max} \quad (4)$$

This is illustrated in Figure 5(a). If $2 \leq i \leq g$ and $0 \leq t < f_i$, no executions of G_i are necessary. Thus, the execution of the first i groups in [0, t] with minimum cost $DPcost(i, t)$ is the same as the execution of the first $i - 1$ groups in [0, t]. We have

$$DPcost(i, t) = DPcost(i - 1, t) \quad (5)$$

for $t < f_i$, as shown in Figure 5(b). When $2 \leq i \leq g$ and $f_i \leq t \leq f_g - 1$, assume that the last time to execute G_i is t_i^l in the optimal execution of the first i groups in [0, t], where $t - f_i + 1 \leq t_i^l \leq t$. $DPcost(i, t)$ is equal to the minimum costs of executing the first i groups in [0, $t_i^l - 1$] and in [$t_i^l + 1$, t], plus the cost of executing G_i at time t_i^l. Recall that the execution of the first i groups in [$t_i^l + 1$, t] is equal to the execution of the first i groups in [0, $t - t_i^l$]. Thus, the minimum cost of executing the first i groups in [$t_i^l + 1$, t] equals to the minimum cost of executing the first i groups in [0, $t - t_i^l$]. Thus, we obtain

$$DPcost(i, t) = DPcost(i, t_i^l - 1) + k_i^{max} + DPcost(i, t - t_i^l) \quad (6)$$

Figure 5(c) illustrates this computation of $DPcost(i, t)$. There are only f_i possible values for t_i^l, namely $t_i^l = t - f_i + 1, t - f_i + 2, ..., t$. Since the optimal execution plan must use one of these values for t_i^l, we need to check them all to find the best. Thus, the recursive definition for the minimum cost $DPcost(i, t)$ of executing the first i groups in [0, t] becomes

$$DPcost(i, t) = \begin{cases} (\lfloor t/f_i \rfloor) * k_i^{max} & \text{if } i = 1 \text{ and } 0 \leq t \leq f_g - 1 \\ DPcost(i-1, t) & \text{if } 2 \leq i \leq g \text{ and } 0 \leq t < f_i \\ min_{(t-f_i+1) \leq t_i^l \leq t}(DPcost(i, t_i^l - 1) + k_i^{max} + & \\ DPcost(i, t - t_i^l)) & \text{if } 2 \leq i \leq g \text{ and } f_i \leq t \leq f_g - 1 \end{cases} \quad (7)$$

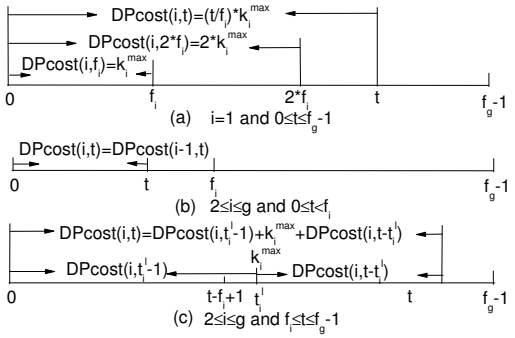


Fig. 5. The illustration of computing $DPcost(i, t)$

According to Equation 7, We develop a dynamic programming solution to compute $DPcost(i, t)$.

Rather than computing the solution recursively, we compute the minimum cost by using a tabular, bottom-up approach. Algorithm 1 uses a table $DPcost[1 - g, 1 - [f_g - 1]]$ for storing the $DPcost(i, t)$ values, and an auxiliary table $s[1 - g, 1 - [f_g - 1]]$ that records the time t_i^l which has achieved the minimum cost in computing, that is, $s(i, t) = t_i^l$ such that $DPcost(i, t) = DPcost(i, t_i^l - 1) + k_i^{max} + DPcost(i, t - t_i^l)$. The inputs are two sequences $\mathcal{F} = \{f_1, f_2, ..., f_g\}$ and $\mathcal{K} = \{k_1^{max}, k_2^{max}, ..., k_g^{max}\}$.

In Algorithm 1, there are three nested loops. The first loop takes at most g values, the second loop takes at most f_g values, and the third loop takes at most f_g values. Thus, the nested loop structure yields a running time of $O(g * f_g * f_g)$ for this algorithm. It requires $O(g * f_g)$ space to store the $DPcost$ and s tables.

To correctly implement the bottom-up approach, we must determine which entries of the table are used in computing $DPcost(i, t)$. Equation (7) shows that $DPcost(i, t)$ of executing the first i groups in [0, t] depends only on $DPcost(i-1, t)$, $DPcost(i, t_i^l - 1)$, and $DPcost(i, t - t_i^l)$. Thus, the algorithm should fill $DPcost$ table in a manner that corresponds to solving the problem on increasing values of i and t. Table

TABLE 5
The values of $DPcost(i,t)$ for Query Example1

	$t=0$	$t=1$	$t=2$
$i=1$	$DPcost(1,0)=0$	$DPcost(1,1)=0$	$DPcost(1,2)=3$
$i=2$	$DPcost(2,0)=0$	$DPcost(2,1)=Gocst(1,1)=0$ (since $t=1$ and $0 \le t < f_2$)	$DPcost(2,2)=DPcost(1,2)=3$ (since $t=2$ and $1 < t < f_2$)
$i=3$	$DPcost(3,0)=0$	$DPcost(3,1)=Gocst(2,1)=0$ (since $t=1$ and $0 \le t < f_3$)	$DPcost(3,2)=DPcost(2,2)=3$ (since $t=2$ and $1 < t < f_3$)

TABLE 6
The values of $DPcost(i,t)$ for Query Example1

	$t=3$	$t=4$
$i=1$	$DPcost(1,3)=3$	$DPcost(1,4)=6$
$i=2$	$DPcost(2,3)=min_{1\le t_i^l\le 3}(DPcost(2,t_i^l-1)+4+DPcost(2,t-t_i^l))=4$(since $t=3$ and $f_2 \le t < f_3$) $(DPcost(2,0)+4+DPcost(2,2))=7\ t_2^l=1$ $(DPcost(2,1)+4+DPcost(2,1))=4\ t_2^l=2$ $(DPcost(2,2)+4+DPcost(2,0))=7\ t_2^l=3$	$DPcost(2,4)=min_{2\le t_i^l\le 4}(DPcost(2,t_i^l-1)+4+DPcost(2,t-t_i^l))=7$(since $t=4$ and $f_2 < t < f_3$) $(DPcost(2,1)+4+DPcost(2,2))=7\ t_2^l=2$ $(DPcost(2,2)+4+DPcost(2,1))=7\ t_2^l=3$ $(DPcost(2,3)+4+DPcost(2,0))=8\ t_2^l=4$
$i=3$	$DPcost(3,3)=DPcost(2,3)=4$ (since $t=3$ and $1 < t < f_3$)	$DPcost(3,4)=DPcost(2,4)=7$ (since $t=4$ and $1 < t < f_3$)

Algorithm 1 The Dynamic Programming DP($\mathcal{F}$,$\mathcal{K}$,$DPcost$,s)

```
for each timestamp t in [0, f_1 − 1] do
    let the minimum cost DPcost[1,t] be 0;
    set the last time s[1,t] to execute group G_1 in [0, t] as -1;
end for
for each timestamp t in [f_1, f_g − 1] do
    set the minimum cost DPcost[1,t] as (⌊t/f_1⌋) * k_1^max;
    if group G_1 is executed at t according to frequency f_1 then
        let the time s[1,t] when G_1 is last executed be t;
    else
        set the time s[1,t] as the time s[1,t−1] when G_1 is last executed ;
    end if
end for
for each query group G_i from group G_2 to group G_g do
    for each timestamp t in [0, f_i − 1] do
        let the minimum cost DPcost[i,t] be the total cost DPcost[i−1,t];
        set the last time s[i,t] to execute group G_i in [0, t] as -1;
    end for
    for each timestamp t in [f_i, f_g − 1] do
        set the initial value of DPcost[i,t] as ∞;
        for each timestamp t_i^l in [t − f_i + 1, t] do
            compute the minimum cost temp = DPcost[i, t_i^l − 1] + k_i^max +
            DPcost[i, t − t_i^l] in [0, t];
            if temp is less than the computed minimum cost DPcost[i,t] then
                let the minimum cost DPcost[i,t] be the new minimum cost temp;
                set the last time s[i,t] to execute G_i with DPcost[i,t] as t_i^l;
            end if
        end for
    end for
end for
```

5 and Table 6 illustrate this procedure for **Query Example1**. Each horizontal row contains the entries for the first i groups in [0, f_g-1]. The algorithm computes the rows bottom-up, and left to right within each row. The minimum execution costs are $DPcost(3,1)=0$, $DPcost(3,2)=3$, $DPcost(3,3)=4$, and $DPcost(3,4)=7$. Remember that an execution plan that minimizes $(DPcost(3,t_3^1-1)+k_3^{max})/t_3^1$ among all the possible t_3^1 ($1 \le t_3^1 \le 5$ where 5 is the value of f_3) values is the best plan. Thus, the execution plan with the minimum $(Gcost(3,3)+k_3^{max})/4=(Gcost(3,3)+5)/4$ is the optimal plan. $Gcost(3,3)=Gcost(2,3)$ since $t=3$ and $t < f_3$. $Gcost(2,3)=min_{1\le t_i^l\le 3}(Gcost(2,t_i^l-1)+4+Gcost(2,t-t_i^l))$ since $t=3$ and $f_2 \le t < f_3$. We can see that $Gcost(2,3)=(Gcost(2,1)+4+Gcost(2,1))=4$ according to Table 6.

When queries are executed according to their frequency upper bounds, there is no sharing between them. The following Theorem proves that the cost-per-unit time of no sharing is $\Omega(log_2\ x)$ times of that of our DP approach on average.

Theorem 7.1: $E(m/m_g^*|x,F,K)=\Omega(log_2\ x)$ where m_g^* is the cost-per-unit time of the DP approach, m is the cost-per-unit time of no sharing approach, and F, K and x are the set of frequencies, the set of k values and the original number of queries, respectively.

Proof: Please see online supplemental materials.

8 Greedy Algorithm

Theorem 7.1 states that DP can save considerable computation cost compared with no sharing. However, the running time of DP is $O(g*f_g*f_g)$ and it requires $O(g*f_g)$ space to store the $DPcost$ and s tables. In this section, we present a much faster greedy algorithm (denoted as GA) to compute the execution plan of executing all the queries. Recall that when we use DP to compute $DPcost(i,t)$, where $1 \le i \le g$ and $0 \le t \le f_g-1$, we have to check all the possible timestamps $(t-f_i+1, t-f_i+2, ..., t)$ for the last time to execute G_i, which is not efficient. Therefore, we propose a faster greedy algorithm that chooses the time to execute G_i without investigating all the candidate timestamps. The greedy rule tries to choose a time for executing G_i that saves as much computation as possible. The time cost and space cost of the new proposed GA are both $O(f_g)$.

When $i=1$, the optimal execution of G_1 is to execute it at time f_1, thus $t_1^1=f_1$. When $2 \le i \le g$, we set the time that G_i will be first executed according to the following rule.

Greedy rule: The first time to execute G_i (t_i^1) is the last time when G_{i-1} is executed in [0, f_i].

This greedy rule is used because we would like to share the results of G_i with some other groups to save computation and to minimize the added cost of executing G_i. Meanwhile, we want to execute G_i as late as possible since the cost of executing G_i is the largest among the first i groups. Thus, we set the last time of executing G_{i-1} as the first time to execute G_i; the incurred added cost is $k_i^{max}-k_{i-1}^{max}$.

The question now is to compute the last time of execution of G_{i-1} in [0, f_i]. Let $GAcost(t)$ be the total cost of executing the first i groups in [0, t] according to the greedy rule. After we set the first time to execute G_1 as $t_1^1=f_1$, we have $GAcost(t)=0$ when $0 \le t < t_1^1$, and we also have

$$GAcost(t_1^1)=GAcost(t_1^1-1)+k_1^{max} \quad (8)$$

as shown in Figure 6(a). Assume the first time to execute G_{i-1} is t_{i-1}^1 and the first time to execute G_i is t_i^1 where $t_i^1 \le f_i$. According to greedy rule, we have $t_i^1=(f_i/t_{i-1}^1)*t_{i-1}^1$(The "/" is integer division operator). For $t_{i-1}^1 < t < t_i^1$, as t_{i-1}^1

is the query cycle of the first $i-1$ groups, we can define $GAcost(t)$ as follows:

$$GAcost(t) = GAcost(t^1_{i-1}) + GAcost(t - t^1_{i-1}) \quad (9)$$

This is illustrated in Figure 6(b). When G_i is executed at time t^1_i, we have

$$GAcost(t^1_i) = GAcost(t^1_i - 1) + k^{max}_i \quad (10)$$

Thus, the recursive definition for the cost $GAcost(t)$ becomes

$$GAcost(t) = \begin{cases} 0 & \text{if } i = 1 \text{ and } 0 < t < t^1_i \\ GAcost(t^1_{i-1}) + GAcost(t - t^1_{i-1}) & \text{if } 2 \le i \le g \\ & \text{and } t^1_{i-1} < t < t^1_i \\ GAcost(t^1_i - 1) + k^{max}_i & \text{if } 1 \le i \le g \\ & \text{and } t = t^1_i \end{cases} \quad (11)$$

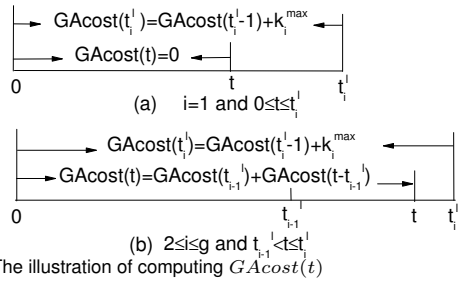


Fig. 6. The illustration of computing $GAcost(t)$

When $i = g$, the cost-per-unit time can be computed as $GAcost(t^1_g)/t^1_g = GAcost(t^1_g - 1) + k^{max}_g/t^1_g$.

To keep track of the execution of the first i groups at time t, we use $e(t)$ to record which group is executed at time t. That is, $e(t)$ is set to be the group number that is executed at time t. Recall that t^1_{i-1} is the query cycle of the first $i-1$ groups. Thus, we obtain:

$$e(t) = \begin{cases} 0 & \text{if } i = 1 \text{ and } 0 < t < t^1_i \\ e(t - t^1_{i-1}) & \text{if } 2 \le i \le g \text{ and } t^1_{i-1} < t < t^1_i \\ i & \text{if } 1 \le i \le g \text{ and } t = t^1_i \end{cases} \quad (12)$$

We can use a table with $f_g - 1$ columns, $GAcost[f_g - 1]$, to record these costs; the detailed steps are given in Algorithm 2. Similarly, we can use a table with $f_g - 1$ columns, $e[1 - f_g]$, to store all the $e(t)$ values. Algorithm 2 has a running time of $O(f_g)$. It requires $O(f_g)$ to store the e tables, and $O(g)$ to store $\mathcal{K}$ and $\mathcal{F}$ sequences. According to Lemma 5.1, different groups have different frequency upper bounds, thus $f_g \ge g$. Algorithm 2 has a space complexity of $O(f_g)$.

Algorithm 2 The Greedy Algorithm GA($\mathcal{F}$,$\mathcal{K}$,c,e)

```
for each timestamp t in [0, f_1 − 1] do
    let the cost GAcost[t] of executing group G_1 in [0, t] be 0;
    set the group number e[t] which is executed at time t as 0;
end for
let the cost GAcost[f_1] of executing group G_1 in [0, f_1] be f_1;
set the group number e[f_1] which is executed at time f_1 as 1;
set the time t_m when G_1 is first executed as f_1;
for each group G_i from G_2 to G_g do
    set the time t_l when G_{i−1} is first executed as t_m;
    let the time t_m when G_i is first executed be (f_i/t_l) * t_l;
    for each timestamp t in [t_l + 1, t_m − 1] do
        let the cost GAcost[t] of executing the first i groups in [0, t] be
        GAcost[t_l] + GAcost[t − t_l];
        set the group number e[t] which is executed at time t as e[t − t_l]
    end for
    set the cost GAcost[t_m] be GAcost[t_m − 1] + k^max_i;
    set the group number e[t_m] which is executed at time t_m to be i;
end for
```

Table 7 shows the procedure of computing $GAcost(t)$ for **Query Example1**. In the table, each horizontal row contains the cost for executing the first i groups in [0, t^1_i]. When $i = 1$, the corresponding row records the cost for [0,t_l] where $t_l = f_1$. For each row i, the costs at timestamps after the first execution time of G_i are not listed; they will be computed when we consider the next group. The minimum cost-per-minute for all three groups is $GAcost(4)/4 = 9/4$. The execution plan with this cost-per-minute is $\{(2, 2), (4, 3)\}$. That is, we execute G_2 at time 2 and the results of G_1 and G_2 are returned. Then, we execute G_3 at time 4 and the results of all the groups are returned.

Obviously, GA is not optimal in some cases. Consider another example where $\mathcal{G} = \{G_1, G_2, G_3\}$, named **Query Example 2**, with the set of corresponding frequency upper bounds $\{30, 51, 60\}$, and the set of k values $\{2, 3, 4\}$. Using the DP approach, the optimal execution is $\{(30, 2), (60, 3)\}$; that is, G_2 is executed at time 30 and the results of G_1 and G_2 are returned, followed by G_3 executed at time 60 and the results of all the groups returned. But according to GA, the best execution is $\{(30, 1), (51, 3)\}$. That is, G_1 is executed at time 30 and the results of G_1 are returned, then, G_3 is executed at time 51 and the results of all the groups are returned. Although GA does not always yield the optimal execution plan, it has lower time and space complexity than DP – both are $O(f_g)$, which is lower than that of DP.

We have already shown that the execution plan obtained by DP is optimal. We now analyze the bound of GA relative to DP in terms of the cost-per-unit time for the execution plan, which shows that the cost-per-unit time of the execution plan obtained by GA is at most 2 times than that computed by DP. In fact, from the experimental results that we report later, the worst case seldom happens, and on average the cost-per-unit time of GA is almost the same as that of DP.

Recall that GA follows the greedy rule for the time t^1_i to execute G_i where $t^1_i \le f_i$. Let m_i be the cost-per-unit time of executing the first i groups using GA. Then, $m_i = GAcost(t^1_i)/t^1_i$. If we use DP, the cost-per-unit time for the first i groups is $m^*_i = min(DPcost(t-1) + k^{max}_i)/t$ where $1 \le t \le f_i$. In order to compare GA with DP, we want to compute the performance bound for each step i in terms of cost-per-unit time (i.e., we compare m_i with m^*_i at each step).

For GA, we first compute the relationship between m_i and m_{i-1} in two consecutive steps $i-1$ and i. According to the greedy rule, the first time to execute G_i, (i.e.,t^1_i), is the last time of executing G_{i-1} in [0, f_i]. The added cost at time t^1_i is a fixed value, that is $k^{max}_i - k^{max}_{i-1}$. Thus, t^1_i is a key factor in computing the relationship between m_i and m_{i-1}. We wish t^1_i to be as large as possible to minimize the added cost-per-unit time from m_i to m_{i-1}. The following lemma shows the lower bound of t^1_i.

Lemma 8.1: Suppose t^1_i is the last time to execute group G_{i-1} in the execution of first $i-1$ groups with m_{i-1} in [0, f_i]. Then, $t^1_i \ge f_i/2$.

Proof: Please see online supplemental materials. □

Lemma 8.1 proves that the last time to execute group G_{i-1} is no less than $f_i/2$. We use it to compute the relationship between m_i and m_{i-1}.

TABLE 7
The values of $GAcost(t)$ for Query Example1

	$t=0$	$t=1$	$t=2$	$t=3$	$t=4$
$i=1$	$GAcost(0)=0$	$GAcost(1)=0$	$GAcost(2)=3$		
$i=2$			$GAcost(2)=GAcost(2)+4-3=4$ (since $t=2$ and $t=t_2^1$) $t_2^1=2$		$GAcost(4)=GAcost(2)+GAcost(4-2)=8$ (since $t_2^1=2$)
$i=3$					$GAcost(4)=GAcost(4)+5-4=9$ (since $t=4$ and $t=t_3^1$) $t_3^1=4$

Theorem 8.1: Let m_i be the cost-per-unit time of executing the first i groups $\{G_1, G_2, ..., G_i\}$ where $1 \leq i \leq g$ in GA. Then $m_i \leq m_{i-1} + 2(k_i^{max} - k_{i-1}^{max})/f_i$.

Proof: Please see online supplemental materials. □

Theorem 8.1 shows the relationship between two steps in GA in terms of cost-per-unit time. Suppose there is a performance bound of GA and DP at step $i-1$. Then, we can use Theorem 8.1 to get the performance bound of these two methods at the next step i.

Theorem 8.2: Let $DPcost(i, t-1)$ be the minimum cost of executing the first i groups in [0, $t-1$] using the DP approach. Then, we have $m_i \leq 2 * min(DPcost(i, t-1) + k_i^{max})/t$ where $1 \leq t \leq f_i$.

Proof: Please see online supplemental materials. □

Theorem 8.2 shows that the performance bound of GA relative to DP is 2 for each step. Corollary 8.1 shows the performance bound for the final step.

Corollary 8.1: Let m_g and m_g^* denote the cost-per-unit time of the solution computed by the DP and the GA approaches, repectively. Then, $m_g/m_g^* \leq 2$.

9 Extension to Multiple Streams

The above discussion focuses on a single stream, but our solutions can be easily extended to multiple streams. If the number of streams is N, and the number of tuples of each stream in a sliding window is w, then $n = w * N$. The top-k probability of each stream is the sum of the top-k probabilities of w tuples of each stream in the sliding window [16].

Suppose the set of multiple streams is $\{\mathcal{T}_1, \mathcal{T}_2, ...\mathcal{T}_N\}$, where each $\mathcal{T}_i$ contains a sequence of tuples $T_{i1}, T_{i2}, ...$, each of which is associated with a membership probability $Pr(T_{ij})$. We use a time-based sliding window over the streams. Suppose the current window is S with size w. Each stream has w tuples in the sliding window. A possible world pw from S is a possible combination of set of the tuples from all the streams. The top-k probability of stream $\mathcal{T}_i$ is $Pr_k(\mathcal{T}_i) = \sum_{1 \leq j \leq w} Pr_k(T_{ij})$ where $Pr_k(T_{ij})$ is the top-k probability of the tuple T_{ij}. According to the top-k probability of $\mathcal{T}_i$, *Pk-topk* returns k streams with the highest top-k probabilities.

We have shown in Section 4 how to compute the top-k probability of each tuple T_{ij}. Once that is computed, it takes $O(w)$ to compute the top-k probabilities of each stream. Then, it costs $O(w*N) = n$ to compute the top-k probabilities of all the streams. As the cost to compute the top-k probability of all the tuples is $O(n^2 * k)$, it costs $O(n^2 * k) + O(n) = O(n^2 * k)$ to compute top-k probabilities of all the data streams.

Suppose there is another top-k' query over the same set of streams in the system. The top-k' probability of each stream depends on the top-k' probability of each tuple in this stream. In Section 4, we have shown that we can use the intermediate results of computing top-k probability of one tuple to compute top-k' probability of this tuple where $k' < k$. Thus, the results of top-k' query can be returned when we compute the results of top-k query over multiple streams. The cost to compute top-k query over multiple streams is $O(n^2 * k)$, which is the same as that of single stream.

10 Experiments

In this section, we report experimental results comparing the dynamic programming approach (DP), the greedy algorithm (GA), intra-group sharing between queries with same frequency upper bounds (IN), INCO described in Section 5, and no sharing approach (NS). All the algorithms were implemented using Microsoft Visual C++ V6.0 and the experiments were conducted on a PC with a 3.0 GHz Pentium 4 CPU, 1.0 GB of RAM, running the Microsoft Windows XP operating system.

10.1 Experiment Setting

Query Sets. We use a synthetic query set where the frequency upper bound f and the k values of each query are drawn from uniform or Gaussian distributions. We developed several generators, each of which generates one distribution on f as well as k, using uniform and Gaussian distributions. First, we specify the number of queries x submitted to the system. Assuming that the set of multiple top-k queries are $Q = \{q_1, q_2, ..., q_x\}$ with the corresponding sets $F = \{f_1, f_2, ..., f_x\}$ and $K = \{k_1, k_2, ..., k_x\}$, we generate all the values for both F and K from uniform or Gaussian distributions. The default parameters for f_i for the uniform distribution range from 1 to 40, while they are 40 and 5 for the Gaussian distribution. For the uniform distribution k_i ranges from 2 to 100 for the single stream, 2 to 20 for the synthetic data of multiple streams, and 2 to 5 for the real data of multiple streams. For Gaussian distribution, the default parameters for k_i are 100 and 10 for the single stream, 5 to 20 for the synthetic data of multiple streams, and 2 to 5 for the real data of multiple streams.

We refer to the results derived from these two distributions by the concatenation of the short names for each distribution for F then K. For example, uu indicates that f and k are both generated by uniform distribution. We show the results of uu in the following, and comment on other combinations briefly.

Data Sets. We use one synthetic data set and one real data set for single stream. Suppose there are n tuples in one sliding window, each of which has a value and a probability. We generate n synthetic tuples from an uniform distribution for the values and the probabilities. The method discussed in Section 4 is used to compute top-k probability of each tuple. To create an uncertain data stream, we repeatedly generate n tuples from uniform distributions for both the property and the probability. For the real single stream data set, we use the International Ice Patrol (IIP) Iceberg Sightings Database3

[20], which collects information on iceberg activity in North Atlantic to monitor iceberg danger near the Grand Banks of Newfoundland by sighting icebergs, plotting and predicting iceberg drift, and broadcasting all known icebergs to prevent icebergs threatening. Each sighting record contains the date, location, shape, size, number of days drifted, etc. It can be considered as a tuple in the sliding window. We choose the number of days drifted as the property of the tuple and convert its visibility into existential probability with six values: 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, and 0.2. To detect iceberg drift and prevent icebergs threats, it is important to continuously report the top-k records with the highest numbers of days drifted for a given interval; e.g., the top-2 records with the highest numbers of days drifted in the last one month. This real application is also used by Jin et al. [20]. We gathered 44440 sighting records from 1998 to 2007. Then, we created an uncertain data stream by repeatedly choosing these records randomly. Then, we randomly chose two consecutive tuples in the stream as a generation rule where the sum of probabilities of these two tuples are no more than 1.

For the synthetic data set of multiple streams, suppose there are N streams and w is the number of tuples of each stream in one sliding window. We repeatedly generate w tuples from uniform distributions for each of the streams. We also use the seismic data sets collected by the wireless sensor network that monitors Reventador, which is an active volcano in Ecuador (http://www.eecs.harvard.edu/∼werner/projects/volcano2005 /data.shtml). Many sensors are deployed at Reventador; we chose 8, 10, 12, 14 and 16 sensors where data from each sensor can be considered as an uncertain stream. The seismic data reported by each sensor is treated as an uncertain stream, and each data record is considered as a tuple. The probability of each tuple in a stream is $1/w$ where w is the number of tuples of each stream in a sliding window, similar to the experimental configuration in [16]. To detect the eruption, it is interesting to continuously report the top-k monitoring streams/locations with the highest seismic values in a given interval; e.g., the top-2 monitoring streams/locations with the highest seismic values in the last 60 seconds. The answer to these queries can be used to detect the locations where eruptions possibly happen. This real application is also described in [16].

Test parameters. Table 8 outlines the test parameters used in this paper. The values of F and K are already described in the above statements of the query sets. The default value of x is 20. The default of n for single stream is 10000. The default values of N and w for the synthetic data set of multiple streams are 100 and 100, while they are 16 and 1000 for real data set of multiple streams.

The metrics. We measure the following to test performance:
1. The execution cost-per-second;
2. The throughput: number of queries executed-per-second;
3. The latency: average latency-per-query. Latency deserves some explanation. It, in effect, computes the latencies of queries due to system load, i.e., some queries that are supposed to execute before time t_i may in fact be delayed and may be executed at time t_j ($t_j > t_i$). Latency measures this phenomenon. The latency of each query is defined as the sum of the additional time between any two consecutive executions.

10.2 Experimental Results

We conduct the experiments with different numbers of queries, different numbers of tuples in a sliding window, and different k values for single stream. For multiple streams, we additionally test different numbers of streams. All these parameters affect the system performance.

10.2.1 Single Stream Synthetic Data Sets

We first report the results on single stream synthetic data under different parameters.

The results with different numbers of queries: We first study the effects of x, the total number of queries. Figure 7(a) shows that the total execution cost-per-second increases as x increases; however, the costs of both DP and GA increase at a much slower rate than that of NS. The cost gap between DP and NS, and GA and NS, quickly increases as x becomes larger. Although the cost of INCO increases slower than that of IN and NS, its cost is still greater than DP and GA at each point.

The throughput is shown in Figure 7(b). GA almost achieves the same throughput as DP. As the number of queries increases, the throughput of DP and GA grow quickly as more queries share computations. The throughputs of DP and GA are much better than those of INCO, which are better than IN or NS.

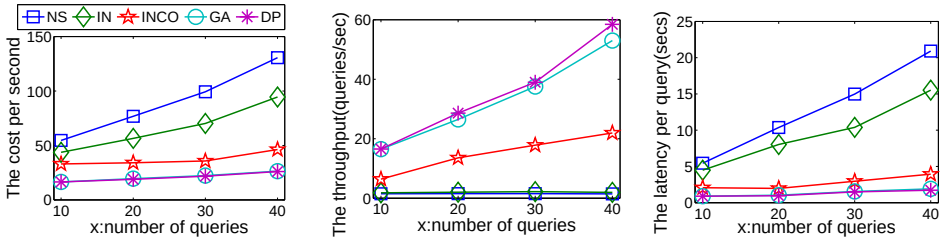
Figure 7(c) shows the latency of each method as the number of queries increases. The latency of NS increases quickly since more queries are executed later than their frequency upper bounds. The latency of IN increases slower than NS, but is still much worse than the latencies of DP and GA, which are better than that of INCO. All the results in Figure 7 demonstrate that DP and GA scale well w.r.t the number of queries.

The results with different number of tuples in a sliding window: In this experiment, we examine the effects of n, the number of tuples in a sliding window. Figure 8(a) shows the cost-per-second with different n values. The costs of DP and GA are almost the same, and are lower than that of INCO, which is much lower than those of IN and NS. The throughput is shown in Figure 8(b). Again, GA achieves almost the same throughput as DP. As n increases, the throughput of each method decreases as the time of executing each query becomes larger. The throughputs of DP and GA are still higher than INCO, which are much higher than those of IN and NS. Figure 8(c) shows the latency of each method as the number of tuples increases. The latency of each method increases with the increase of n. Again, the latencies of DP and GA increase at a slower rate than those of INCO, which increases at a much slower rate than NS and IN. When n becomes larger, the cost of each query increases. That is the reason that the cost-per-second, the throughput, and the latency become worse. However, the cost-per-second and the latencies of DP and GA increase at a much slower rate than those of INCO, NS and IN, and the throughputs are higher than those of INCO, IN and NS at each point. This is because the queries can share more computations in DP and GA than in INCO, IN, and NS.

The results with different k ranges: It is interesting to study the cost, throughput and latency for various k. We test

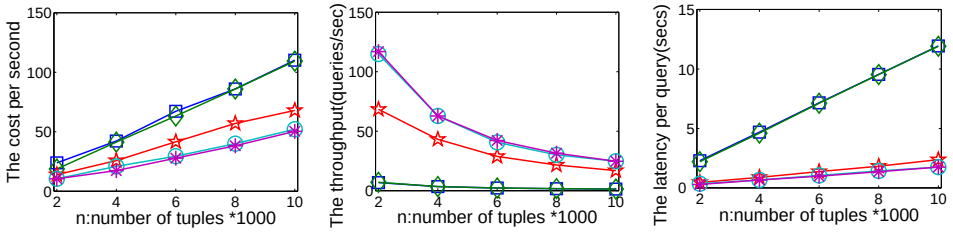
TABLE 8
The test parameters

parameter	meaning	value
x	the number of queries	10, 20, 30, 40
N	the number of streams of multiple streams	[20,100] for the synthetic data and [8,16] for the real data
w	the number of tuples of each stream in one sliding window in the context of multiple streams	[20,100] for the synthetic data and [200,1000] for the real data
n	the number of tuples in one sliding window	from 2000 to 10000 while it is $n = N * w$ for multiple streams
F	the set of frequency upper bounds for all x queries	
K	the set of the k values for all x queries	



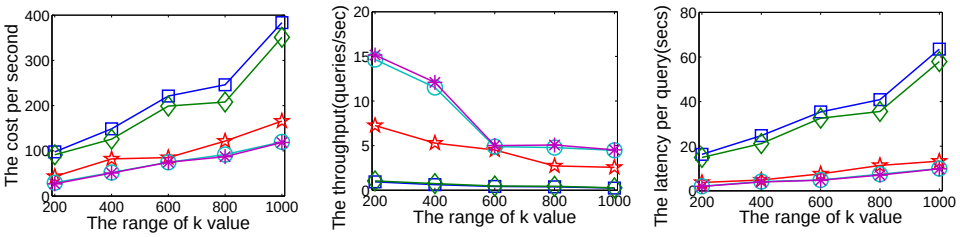
(a) The cost-per-second (b) The throughput (c) The latency

Fig. 7. The performance of DP, GA, INCO, IN and NS with different numbers of queries



(a) The cost-per-second (b) The throughput (c) The latency

Fig. 8. The performance of DP, GA, INCO, IN and NS with different numbers of tuples



(a) The cost-per-second (b) The throughput (c) The latency

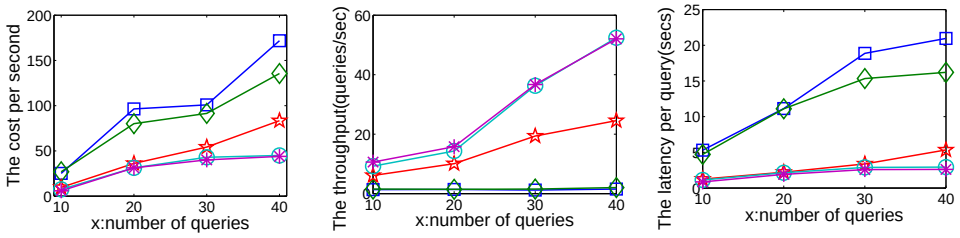
Fig. 9. The performance of DP, GA, INCO, IN and NS with different ranges of k

k with different ranges, [2,200], [2,400], [2,600], [2,800], and [2,1000]. The trend of each measure in Figure 9 is similar to that in Figure 8. The results confirm that the cost of the query depends on n and k. Figure 9 shows that DP and GA also scale well w.r.t the range of k, while NS and IN do not. In this figure, the throughput of each method drops with the increase of k. This is because the cost of executing a query is $O(n_2 * k)$, which is proportional to k.

We also conduct experiments for other distribution combinations. The trends of each measure for ug, gu and gg are similar to those for uu. For the gg combination, we observe that the ratios of the cost-per-second of NS and IN relative to DP and GA are larger than those of uu, ug and gu, and similar trends exist for the latency and throughput. After analyzing all the data combinations carefully, we find that the reason for different performance on gg compared to that of other combinations is that queries can share more computations in gg as the values of f and k are more centralized.

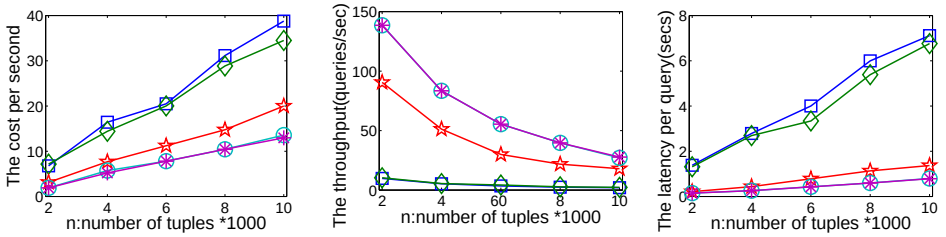
10.2.2 *Single Stream Real Data Sets*

The results over the real data set are shown in Figures 10, 11, and 12. The parameters in these experiments are set to the values described earlier for the synthetic data set of single stream. We observe that the performance of GA and DP are almost the same, and they are better than INCO, which is much better than IN and NS. This confirms that our proposals



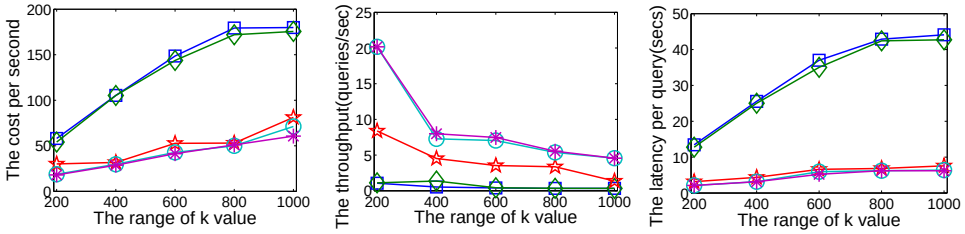
(a) The cost-per-second (b) The throughput (c) The latency

Fig. 10. The performance of DP, GA, INCO, IN and NS with different ranges of x



(a) The cost-per-second (b) The throughput (c) The latency

Fig. 11. The performance of DP, GA, INCO, IN and NS with different number of tuples



(a) The cost-per-second (b) The throughput (c) The latency

Fig. 12. The performance of DP, GA, INCO, IN and NS with different ranges of k

also work well on real data.

10.2.3 Multiple Stream Synthetic Data Sets and Real Data Sets

In these experiments, we use synthetic data sets and real data sets for multiple streams to conduct the experiments with different number of streams, different number of tuples in a sliding window, different number of queries, and different k ranges. The parameters in these experiments are set to the values described earlier. As the number of streams increases or the number of tuples of each stream in the sliding window increases, the total number of tuples increases. For synthetic data sets, Figures 13 and 14 show the results when the total number of tuples increases. The trends are the same as those of the single stream. Figures 15 and 16 show that the performance of our algorithms are much better than IN and NS with different range of x and k. For real data sets, we can observe the similar trends as those of synthetic data sets. Due to the space limit, we put the results for the real data sets in the online supplemental material.

10.2.4 The Runtime of DP and GA

Next, we study the runtime of DP and GA with different ranges of f: [1,10000],[1,20000],[1,30000], and [1,40000]. Figure 17 shows that when the range of f increases, the runtime of GA and DP also increases. The runtime of GA is less than that of DP in each range of f. The results verify our theoretical analysis of the complexity of DP and GA.

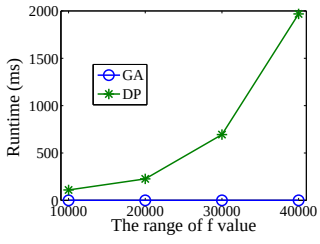
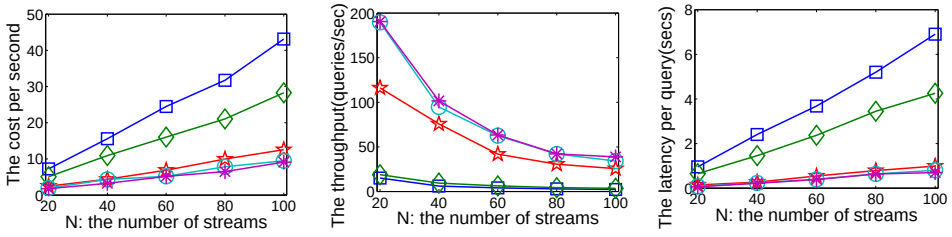


Fig. 17. The runtime of DP and GA with different range of f

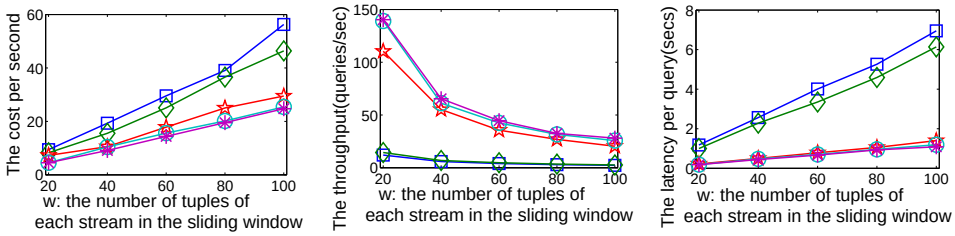
10.2.5 The ratio of DP and NS in terms of the cost-per-unit time

Finally, in order to compare the ratio of DP and NS in terms of the cost-per-unit time, we use the results of real data sets



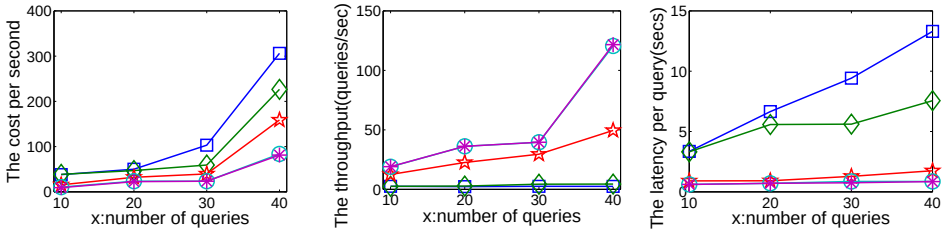
(a) The cost-per-second (b) The throughput (c) The latency

Fig. 13. The performance of DP, GA, INCO, IN and NS with different ranges of N



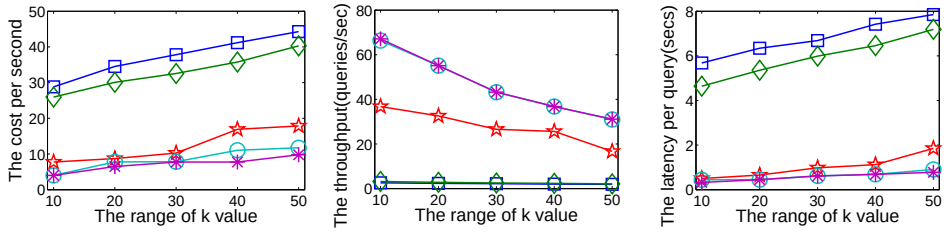
(a) The cost-per-second (b) The throughput (c) The latency

Fig. 14. The performance of DP, GA, INCO, IN and NS with different ranges of w



(a) The cost-per-second (b) The throughput (c) The latency

Fig. 15. The performance of DP, GA, INCO, IN and NS with different ranges of x



(a) The cost-per-second (b) The throughput (c) The latency

Fig. 16. The performance of DP, GA, INCO, IN and NS with different ranges of k

for multiple stream when x changes from 40 to 100. We see from Table 9 that the ratio of DP and NS in terms of the cost-per-unit time can satisfy the bound $log_2\ x$, which verifies our theoretical analysis in Section 7.

TABLE 9
The ratio of DP and NS in terms of the cost-per-unit time and the bound

x	DP	NS	The ratio	$log_2\ x$
40	168.3900	31.3742	5.3671	5.3219
60	232.6250	39.0000	5.9647	5.9069
80	338.6500	39.0000	8.6833	6.3219
100	383.6250	47.0000	8.1622	6.4919

11 Conclusions

Querying over uncertain data streams, in particular top-k querying is important. There are many applications that require this functionality as discussed in the Introduction. These applications always involve a large number of similar top-k queries. Although there have been some studies considering top-k querying over uncertain data streams, all of them consider

individual queries and cannot be directly used for sharing computation among multiple top-k queries. This sharing problem is very challenging for the uncertain top-k queries with different frequency upper bounds and different k values. In this paper, we formulate the problem, and present an optimal dynamic programming solution and a greedy algorithm. We show that a naive method of enumerating all possible plans is not efficient, and the dynamic programming is optimal as it satisfies the optimal substructure and overlapping subproblems. The solution computed by the dynamic programming is the best plan for executing queries. Although the greedy algorithm is not optimal, it is more efficient than the dynamic programming solution in terms of time and space. The experiments we conducted verify the effectiveness of our proposed solutions.

Our approaches can also support other top-k definitions. There are seven top-k definitions in literature so far, as discussed in Section 2. According to the similarity between results of different top-k queries, we divide these definitions into two classes. In the first class, top-k results are completely disjoint from top-$k+1$ results; PT-k, Global-top$k$, PT-top$k$ and U-top$k$ are in this class. In the second class, top-k results are a subset of top-$k+1$ results; U-kRanks, and Expected Rank are in this class. Our approaches can support both categories of definitions.

REFERENCES

[1] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: a performance evaluation. In *Proceedings of the 14th International Conference on Very Large Data Bases*, pages 1–12, 1988.

[2] C. C. Aggarwal and P. S. Yu. A framework for clustering uncertain data streams. In *Proceedings of IEEE 24th International Conference on on Data Engineering*, pages 150–159, 2008.

[3] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: Semantic foundations and query execution. *The Very Large Data Bases Journal*, 15(2):121–142, 2006.

[4] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *Proceedings of the 30th International Conference on Very Large Data Bases*, pages 336–347, 2004.

[5] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas. Operator scheduling in data stream systems. *The VLDB Journal*, 13(4):333–353, 2004.

[6] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 1–16, 2002.

[7] V. Bonifaci, H. L. Chan, A. M. Spaccamela, and N. Megow. Algorithms and complexity for periodic real-time scheduling. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1350–1359, 2010.

[8] D. Carney, U. Cetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 838–849, 2003.

[9] G. Cormode, F.F.Li, and K.Yi. Semantics of ranking queries for probabilistic data and expected ranks. In *Proceedings of IEEE 25th International Conference on Data Engineering*, pages 305 – 316, 2009.

[10] G. Cormode and M. N. Garofalakis. Sketching probabilistic data streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 281–292, 2007.

[11] L. Golab, K.G. Bijay, and M. T. Özsu. Multi-query optimization of sliding window aggregates by schedule synchronization. In *Proceedings of the 15th ACM International Conference on Information and Knowledge Management*, pages 844–845, 2006.

[12] L. Golab and M. T. Özsu. Issues in data stream management. *SIGMOD Record*, 32(2):5–14, 2003.

[13] L. Golab and M. T. Özsu. *Data Stream Management*. Morgan & Claypool, 2010.

[14] M. Hammad, M. J. Franklin, W. Aref, and A. Elmagarmid. Scheduling for shared window joins over data streams. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 297–308, 2003.

[15] J. Haritsa, M. Carey, and M. Livny. Earliest-deadline scheduling for real-time database systems. In *Proceedings of the 12th Real-Time Systems Symposium*, pages 232–242, 1991.

[16] M. Hua and J. Pei. Continuously monitoring top-k uncertain data streams:a probabilistic threshold method. *Journal of Distributed & Parallel Databases*, 26:29–65, 2009.

[17] M. Hua, J. Pei, W. Zhang, and X. Lin. Ranking queries on uncertain data: A probabilistic threshold approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 673–686, 2008.

[18] T. S. Jayram, A. McGregor, S. Muthukrishnan, and E. Vee. Estimating statistical aggregates on probabilistic data streams. In *Proceedings of theTwenty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 243–252, 2007.

[19] Q. Jiang and S. Chakravarthy. Scheduling strategies for processing continuous queries over streams. In *Proceedings of the 21st British National Conference on Databases*, pages 16–30, 2004.

[20] C. Q. Jin, K. Yi, L. Chen, J. X. Yu, and X. M. Lin. Sliding-window top-k queries on uncertain streams. In *Proceedings of the 34th International Conference on Very Large Data Bases*, pages 301–312, 2008.

[21] S. Krishnamurthy, C. Wu, and M. Franklin. On-the-fly sharing for streamed aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 623–634, 2006.

[22] J. Li, B. Saha, and A. Deshpande. A unified approach to ranking in probabilistic databases. In *Proceedings of the 35th International Conference on Very Large Data Bases*, pages 502–513, 2009.

[23] S. Madden, M. Shah, J. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 49–60, 2002.

[24] T. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13(1):23–52, 1988.

[25] M. A. Soliman, I. F. Ilyas, and K. C.-C. Chang. Top-k query processing in uncertain databases. In *Proceedings of IEEE 23rd International Conference on on Data Engineering*, pages 896–905, 2007.

[26] K. Yi, F. Li, G. Kollios, and D. Srivastava. Efficient processing of top-k queries in uncertain databases with x-relations. *IEEE Transactions on Knowledge and Data Engineering*, 20(12):1669–1682, 2009.

[27] R. Zhang, N. Koudas, B. C. Ooi, and D. Srivastava. Multiple aggregations over data streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 299–310, 2005.

[28] X. Zhang and J. Chomicki. On the semantics and evaluation of top-k queries in probabilistic databases. In *Proceedings of IEEE 24th International Conference on Data Engineering Workshops*, pages 556–563, 2008.

Tao Chen is an Assistant Professor in Beijing Institute of Radiation Medicine, China. Her research interests include uncertain databases, bioinformatics.

Lei Chen is an Associate Professor in Hong Kong University of Science and Technology. His research interests include uncertain databases, graph databases.

M. Tamer Özsu is a Professor of Computer Science at the University of Waterloo. Dr. Özsu's current research focuses on large scale data distribution and management of unconventional data (e.g., XML, graphs, RDF, streams). He is a Fellow of ACM and IEEE, and a member of Sigma Xi.

Nong Xiao is currently a Professor in the College of Computer at the National University of Defense Technology. His research interests mainly include storage and grid computing.