Maintaining Sliding Window Skylines on Data Streams

Yufei Tao and Dimitris Papadias

Abstract—The skyline of a multidimensional data set contains the "best" tuples according to any preference function that is monotonic on each dimension. Although skyline computation has received considerable attention in conventional databases, the existing algorithms are inapplicable to stream applications because 1) they assume static data that are stored in the disk (rather than continuously arriving/expiring), 2) they focus on "one-time" execution that returns a single skyline (in contrast to constantly tracking skyline changes), and 3) they aim at reducing the I/O overhead (as opposed to minimizing the CPU-cost and main-memory consumption). This paper studies skyline computation in stream environments, where query processing takes into account only a "sliding window" covering the most recent tuples. We propose algorithms that continuously monitor the incoming data and maintain the skyline incrementally. Our techniques utilize several interesting properties of stream skylines to improve space/time efficiency by expunging data from the system as early as possible (i.e., before their expiration). Furthermore, we analyze the asymptotical performance of the proposed solutions, and evaluate their efficiency with extensive experiments.

Index Terms—Skyline, stream, database, algorithm.

1 Introduction

Skyline queries are important to applications that require retrieval with respect to user preferences. Fig. 1a shows an example where each 2D point represents a stock record with two attributes: *risk* (x axis) and *commission cost* (y axis). A stock r dominates another r' if and only if the coordinates of r on all axes are smaller than or equal to those of r'. The skyline consists of all stocks that are not dominated by others, i.e., $\{b, e, f\}$ in Fig. 1a. These are the best stocks since if a tuple r dominates another r', then r is preferable to r' by any "preference function" that is monotonic on all attributes. For instance, stock e has lower risk and cost than a, c, and d, meaning that e is better independently of the relative importance of the two attributes. On the other hand, e and f are incomparable since a long-term investor may be willing to pay higher commission to ensure lower risk.

Skyline computation has received considerable attention in relational databases [6], [26], [21], [9], [23] and Web information systems [3]. However, recently, the database community witnessed a paradigm shift to query processing over continuous streams (rather than static data sets stored in disks). The goal is to continuously report the qualifying records for long-standing queries in a real-time manner. Techniques developed in traditional databases are inefficient or simply useless in this scenario because they typically do not consider the special characteristics of streams, such as fast data arrivals, strict limits for response time, etc. Stream-oriented algorithms usually maintain all the data in memory to avoid the expensive overhead of disk I/Os.

This paper studies skyline computation in stream systems that consider only the tuples that arrived in a *sliding window* covering the W most recent timestamps, where W is a system parameter called the window *length*. Specifically, a tuple r is *alive* during its *lifespan* $[r.t_{arr}, r.t_{exp})$, where $r.t_{arr}$ is its arrival time, and $r.t_{exp}$ is the expiry time equal to $r.t_{arr} + W$. Furthermore, the stream is "append-only" [10], [15], meaning that a tuple is not replaced before its expiry. Streams satisfying the above requirements are abundant in sensor-networks where the readings of a sensor are transmitted to a central server periodically, e.g., every $W = 5$ minutes. Records that arrived 5 minutes ago are discarded from the server because they most likely do not reflect the current sensor readings.

Our objective is to maintain the skyline over the live data, and continuously output the skyline changes. Fig. 1b shows an example, where the number beside each point r denotes the arrival time $r.t_{arr}$ of a stock record, assuming that these records are processed in a FIFO (first in, first out) manner, and their lifespan lengths are $W = 5$. The output of the skyline operator is itself a stream; in this example, the stream contains pairs $(+a, 1)$, $(+b, 3)$, $(-a, 6)$, $(+c, 6)$, $(+d, 7)$, $(-b, 8)$, $(+e, 9)$, $(-c, 9)$, $(-d, 9)$, $(+f, 11)$, etc. A pair $(+r, t)$ implies that point r starts belonging to the skyline at time t, while $(-r, t)$ indicates the removal of r from the skyline at time t. All pairs are produced in chronological order, and no skyline modifications occur between two consecutive pairs, i.e., the system captures *all the skyline changes*.

We propose algorithms that utilize the special properties of "stream skylines" to improve space and time efficiency

- *Y. Tao is with the Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Hong Kong. E-mail: taoyf@cs.cityu.edu.hk.*
- *D. Papadias is with the Department of Computer Science, Hong Kong University of Science and Technology, Clearwater Bay, Hong Kong. E-mail: dimitris@cs.ust.hk.*

Manuscript received 6 July 2004; revised 23 Mar. 2005; accepted 8 July 2005; published online 18 Jan. 2006.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0201-0704.

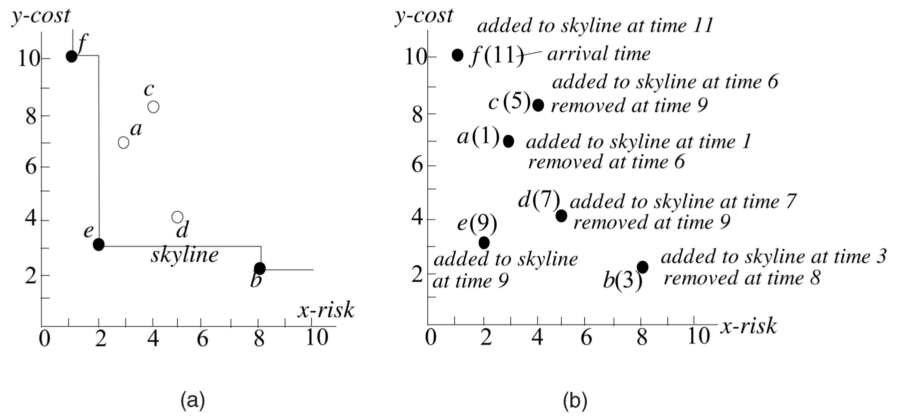

Fig. 1. Skyline examples. (a) Conventional skyline. (b) Stream skyline ($W = 5$).

by expunging tuples from the system as early as possible (before their expiration). Some of these properties are:

- All points dominated by an incoming tuple r can be safely discarded since they are guaranteed not to appear in the skyline in the future. For instance, the arrival of e at time 9 immediately expunges the current skyline points c and d although they have not expired yet.
- An arriving tuple r cannot be directly discarded even if it is dominated by some existing tuple r' in the database—r will expire after r' and, therefore, may become part of the skyline later. For instance, although c, at the time (=5) of its arrival, is dominated by a, c appears in the skyline after the expiration of a at time 6.
- In fact, a tuple r can appear in the skyline for at most a single *continuous time interval*, which can be computed upon the arrival of r. For example, at time 5, we can say that c will be a skyline point during the future time interval [6, 10) (the value 10 is the expiry time of c), *provided that* it will not be dominated by a subsequent arrival.

The above windows are *time-based*. In *count-based* sliding windows [13], [2], [15], a tuple expires after W subsequent tuples have been received, regardless of their arrival times. Although our discussion focuses on time-based windows, the proposed solutions are directly applicable to count-based streams using a simple transformation. Specifically, we manually associate each tuple r with an "artificial arrival time" $r.t_{arr}$, which equals its sequence-id in the stream (i.e., the first tuple has arrival time 1, the second 2, etc.). Its expiry time equals $r.t_{arr} + W$, i.e., the count-based window can be regarded as a time-based one (with length W) defined on the artificial time.

The rest of the paper is organized as follows: Section 2 reviews the related work on stream processing and skyline computation. Section 3 describes two general frameworks for maintaining stream skylines. Section 4 analyzes and compares the performance of alternative frameworks, while Section 5 discusses their practical implementations. Section 6 experimentally evaluates the efficiency of the proposed techniques. Section 7 concludes the paper with directions for future work.

2 Related Work

A fundamental issue in the stream literature is to explore alternative algorithms for traditional database operators. Existing work focuses mostly on aggregate queries and joins. Aggregate methods usually maintain various data summaries (e.g., histograms, sample sets, sketches, etc.) to provide fast approximate results, often with accuracy guarantees [12]. For joins, the common approach is to adapt conventional methods (e.g., block nested loop, hash-join) to their "unblocking" counterparts [24]. In a system that involves a large number of operators (e.g., for supporting multiple concurrent queries), the performance depends on effective "scheduling" [7], i.e., deciding the "best" operator that should be executed next in order to minimize factors such as memory consumption, output rate, etc. Query optimization in such systems [28] must take into account the tuple arrival rates since, intuitively, a faster stream should receive more computing resources to avoid jamming the incoming traffic. It is possible that an excessively large volume of data may arrive at the system within a short period of time (i.e., "bursty" traffic) such that the system (due to its limited capacity) cannot process all incoming records. In this case, some tuples must be "shed," i.e., discarded from the system without passing through any query operator. Evidently, shedding degrades the quality of query results, but the amount of degradation may be reduced through "semantic shedding" [10], which aims at expunging only the least useful data—those expected to have insignificant influence on the result.

None of the above methods consider skyline maintenance on streams. On the other hand, skyline computation on static data (and other related problems such as multi-objective optimization [25] and maximum vectors [22]) has been extensively studied in conventional processing frameworks. In the database context, Borzsonyi et al. [6] develop two solutions based on divide-and-conquer (DC) and block-nested-loop (BNL), respectively. Specifically, DC divides the data set into several partitions that can fit in memory. The skylines in all partitions are computed separately using a main-memory algorithm and then merged to produce the final skyline. BNL essentially compares each tuple in the database with all the other data and outputs the tuple only if it is not dominated. The sort-first-skyline (SFS) [9] sorts

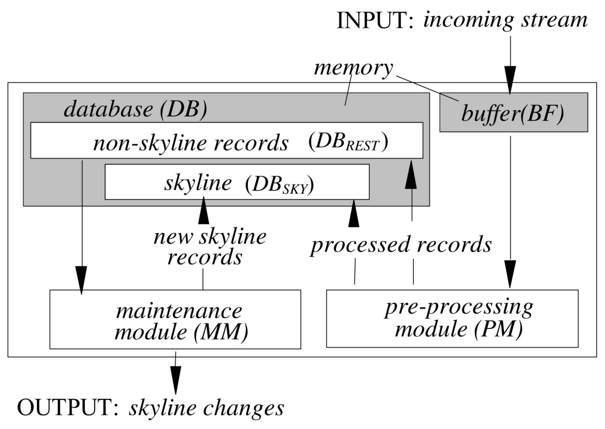

Fig. 2. The architecture of our system.

the database according to a (monotone) preference function, after which the skyline can be found in another pass over the sorted list. Tan et al. [26] propose an algorithm that obtains the skyline from bitmaps (derived from the original data set) using bitwise operations, i.e., calculating the AND/OR of two binary vectors. The authors also provide another method based on relationships between the skyline and the minimum coordinates of individual points. Kossmann et al. [21] and Papadias et al. [23] discover the skyline with nearest-neighbor search on spatial access methods. Balke et al. [3] study skylines in Web information systems applying the "threshold" algorithm of [11].

The above methods assume that all the relevant data are available (in main memory, disk, or distributed servers) before processing. Furthermore, they report a single skyline and terminate. On the other hand, in stream environments, data are not known in advance, but they keep changing as new tuples arrive and old ones expire. The objective is to continuously monitor the skyline changes according to the record arrivals and expirations. Therefore, our problem is more related to incremental maintenance rather than one-time skyline computation, rendering all the existing techniques inapplicable.

3 Frameworks for Tracking Skylines on Streams

Fig. 2 demonstrates the architecture of our system. Arriving tuples are placed in an input buffer (BF), processed by the preprocessing module (PM) in ascending order of their arrival times, and then included into the *database* (DB). Both BF and DB are memory resident. DB is further divided into DB_{sky} and DB_{rest} storing points that are and are not in the current skyline, respectively. Whenever a skyline point expires, some points in DB_{rest} may appear in the new skyline. They are identified by the *maintenance module* (MM), which is also responsible for expunging the obsolete (i.e., dead) data from DB, and outputting the skyline stream.

Lemma 1. *Let r be a tuple in DB. If r is dominated by an incoming record r', then r can be safely discarded, i.e., it will not be part of the skyline in the future.*

TABLE 1
Frequently Used Symbols

Symbol	Description
W	the sliding window size
d	the dimensionality
DB	the set of existing tuples in the system
DB_{sky}	the set of current skyline points
DB_{rest}	the set of non-skyline tuples in DB
EL	the event list of *Eager*
$r.t_{arr}$	the arrival time of a tuple r
$r.t_{exp}$	the expiry time of r
$r.t_{sky}$	the skyline influence time of r

Proof. Trivial because r' will expire after r and, thus, will dominate r in its remaining lifespan. □

Based on the above lemma, we develop two general frameworks for online monitoring of stream skylines. Section 3.1 first describes a "lazy" strategy that delays most computational work until the expiration of a skyline point. Then, Section 3.2 presents an "eager" approach that takes advantage of precomputation to minimize memory consumption. The following discussion uses two-dimensional data, but the proposed methods generalize to arbitrary dimensionality in a straightforward manner. Table 1 summarizes some symbols that will be used frequently throughout the paper.

3.1 The *Lazy* Method

Skyline changes can occur only when 1) a new tuple arrives or 2) some skyline point expires. *Lazy* handles these two situations in its preprocessing module (L-PM) and maintenance module (L-MM), respectively.

Given an arriving tuple r, L-PM checks if it is dominated by any point in DB_{sky}. For example, in Fig. 1b, the arrival of c at time 5 does not affect the skyline (since c is dominated by a); thus, c is placed in DB_{rest} (recall that c cannot be discarded as it will appear in the skyline after a expires). On the other hand, if the new tuple r is not dominated by any skyline point, it is added to DB_{sky}. Furthermore, in this case, r may dominate some skyline points, which are expunged from the system (they cannot reappear in the skyline later, as shown in Lemma 1). For instance, in Fig. 1b, when e arrives at time 9, it immediately belongs to the skyline, while skyline tuples c and d are expunged (even though they have not expired yet).

We define the *dominance region* $r.DR$ of a tuple r to be the area of the data space dominated by r. Specifically, $r.DR$ is an axis-parallel rectangle whose major diagonal is decided by r and the "max corner" of the data space (having the maximum coordinates on all dimensions). Similarly, the *antidominance region* $r.ADR$ of r refers to the area where a point dominating r could fall; $r.ADR$ is a rectangle whose major diagonal is determined by r and the origin of the data space. The gray regions in Fig. 3 illustrate $r.DR$ and $r.ADR$.

Deciding whether r is dominated by any skyline point essentially corresponds to a d-sided emptiness test (d is the dimensionality of the data space), which returns a Boolean

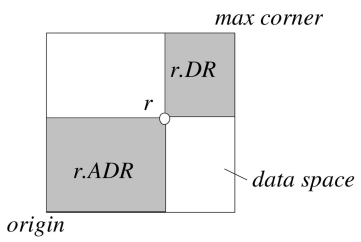

Fig. 3. The dominance and antidominance regions of h.

value indicating whether any tuple of DB_{sky} falls in $r.ADR$ (a "true" result means that $r.ADR$ is empty). On the other hand, finding the existing skyline points dominated by r can be achieved by a d-sided range search, which reports all the tuples of DB_{sky} lying in $r.DR$. These two query types are "d-sided" because d (out of $2d$) boundaries of $r.ADR$ and $r.DR$ coincide with the borders of the data space.

Fig. 4 illustrates the pseudocode of L-PM which, in addition to carrying out the above operations, also maintains the earliest expiry time T_{exp}^{sky} of the current skyline points (i.e., $T_{exp}^{sky} = \min\{r.t_{exp} | r \in DB_{sky}\}$), as well as a pointer to the oldest skyline tuple r_{old} that decides T_{exp}^{sky}. An incoming tuple r changes T_{exp}^{sky} if and only if r dominates (and, hence, evicts) r_{old}. For instance, at timestamp 8 in Fig. 1b, the skyline contains c, d, and T_{exp}^{sky} equals 10 (the expiry time of c). At the next timestamp, the arriving tuple e expunges c and d, and, accordingly, T_{exp}^{sky} is modified to the expiry time 14 of e (the new skyline contains only e).

L-MM is invoked at time T_{exp}^{sky}, and it first eliminates the data from DB_{rest} that already expired at some time before T_{exp}^{sky} (such data remain in the system after expiration, and are discarded together in one execution of L-MM). Since the tuples of DB_{rest} are sorted in ascending order of their arrival times, L-MM scans the sorted list from the beginning, and eliminates the tuples encountered until a live tuple is found.

As a second step, L-MM evicts the expiring skyline point r and updates the skyline based on two observations: 1) the remaining data in DB_{sky} still belong to the skyline and 2) the set S of "candidate" tuples in DB_{rest} that may appear in the skyline must be dominated *exclusively* by r (i.e., not by any other data in DB_{sky}). Hence, L-MM computes the skyline for the data of S using an existing algorithm [22], [9], [23], and inserts the points of this skyline in DB_{sky}. Fig. 5 formally describes the execution of L-MM.

The above discussion presents the general idea of *Lazy*, leaving out several implementation issues including, for example, the structures organizing the data in DB_{sky} and DB_{rest}, the algorithms for performing d-sided emptiness tests and range search. In fact, as discussed in Section 4.1, the above framework can lead to various implementations with different characteristics, depending on the choices of structures and algorithms. A disadvantage of *Lazy*, however, is that DB_{rest} needs to store obsolete data and tuples that will never appear in the skyline. This problem motivates our next methodology.

3.2 The *Eager* Method

Eager aims at 1) minimizing the memory consumption by keeping only those tuples that are or may become part of the skyline in the future and 2) reducing the cost of the

```
Algorithm L-PM //invoked for each incoming tuple
1.  r = an incoming tuple from $BF$
2.  issue a $d$-sided emptiness test to check whether any point of $DB_{sky}$ is in $r.ADR$
3.  if the test result is false ($r.ADR$ is not empty)
4.      insert $r$ to $DB_{rest}$ and return //terminate L-PM
5.  else
6.      issue a $d$-sided range query to find the points of $DB_{sky}$ in $r.DR$
7.      for each tuple $r'$ in the query result
8.          output ($-r'$, current time), and discard it from the system
9.      output ($+r$, current time) and add $r$ to $DB_{sky}$
10.     set $T_{exp}^{sky}$ to the earliest expiry time of the tuples in $DB_{sky}$
```

Fig. 4. Preprocessing module of *Lazy*.

```
Algorithm L-MM //invoked at time $T_{exp}^{sky}$
1.  $r$ = the tuple in $DB_{sky}$ that expires
2.  output ($-r$, $T_{exp}^{sky}$) and remove $r$ from $DB_{sky}$
3.  $r'$ = the first (oldest) tuple in $DB_{rest}$
    //data in $DB_{rest}$ are sorted in ascending order of their arrival times
4.  while $r'.t_{exp} < T_{exp}^{sky}$ //$r'$ expired already
5.      expunge $r'$ and set $r'$ to the next tuple in $DB_{rest}$
6.  compute the skyline for the set $S$ of data in $DB_{rest}$ dominated exclusively by $r$, but not by any other
    skyline point
7.  for each tuple $r'$ in this skyline
8.      output ($+r'$, $T_{exp}^{sky}$), and move $r'$ from $DB_{rest}$ to $DB_{sky}$
9.  set $T_{exp}^{sky}$ to the earliest expiry time of the records in $DB_{sky}$
```

Fig. 5. Maintenance module of *Lazy*.

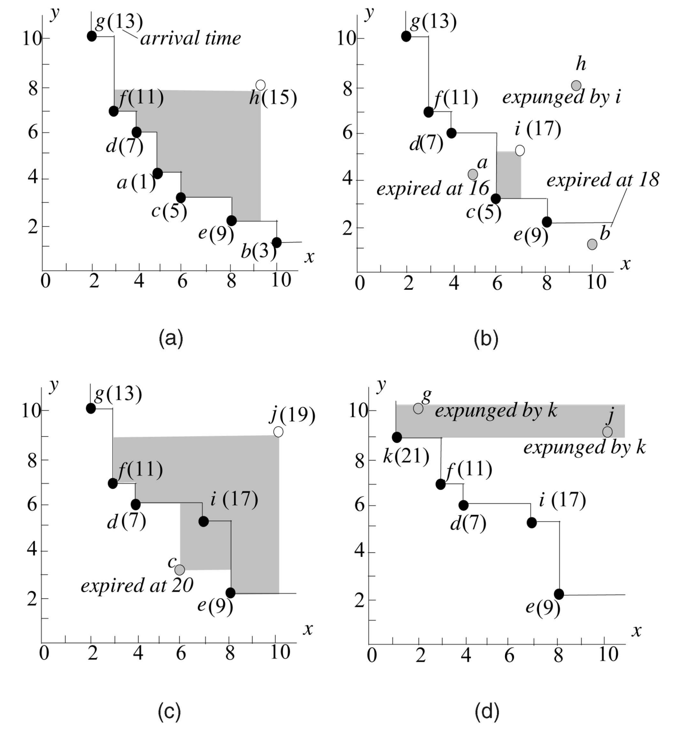

Fig. 6. Execution example of *Eager* ($W = 15$). (a) Skyline from time 13 to 15. (b) Skyline at time 18. (c) Skyline at time 20. (d) Skyline at time 21.

maintenance module (E-MM). It achieves these goals by performing additional work (compared to *Lazy*) in the preprocessing module (E-PM). In the sequel, we first use a concrete example to illustrate the functionality of *Eager*, and then clarify the details of E-PM and E-MM.

In Fig. 6a, points $a, b, \ldots, g$ constitute the content of DB at time 13 (all of them belong to the skyline). The arrival timestamps of individual points are indicated in parentheses, assuming that the length W of the sliding window equals 15. Consider the point h arriving at time 15, which is dominated by a set S of tuples $\{a, c, d, e, f\}$. Notice that h can appear in the skyline only after timestamp 26 when all the tuples in S have expired. We say that 26 is the *skyline influence time* of h, and denote it as $h.t_{sky}$.

Since h is currently not a skyline point, *Eager* places it in DB_{rest}. At the next timestamp 16, a expires, and *Eager* asserts that no point in DB_{rest} can become part of the skyline, without examining DB_{rest} at all. Indeed, the skyline influence time of the only tuple h in DB_{rest} equals 26 and, hence, h does not "influence" the skyline at the current time 16.

Fig. 6b continues the example with an incoming tuple i at time 17. The only point among those in DB_{sky} and DB_{rest} dominating i is c. Following the same reasoning for h, *Eager* sets the skyline influence time $i.t_{sky}$ of i to the expiry time 20 of c, and adds i to DB_{rest}. Furthermore, h can be expunged from the system because it is dominated by i, and (by Lemma 1) is guaranteed not to appear in the skyline later (DB_{rest} contains a single tuple i now). Recall that *Lazy* would not compare i and h (since h is not in DB_{sky}) and, therefore, would not evict h.

The next "event" corresponds to the expiry of b at time 18. Similar to processing the expiration of a, *Eager* simply removes b without any further actions. When j arrives at time 19, it is dominated by $\{c, d, e, f, i\}$, and is included in DB_{rest} with $j.t_{sky} = 32$, which is the expiry time of point i in DB_{rest}. At timestamp 20, c expires; since 20 is also the skyline influence time of i, *Eager* includes i into DB_{sky} (without inspecting other tuples), as shown in Fig. 6c. The last incoming tuple k at time 21 is not dominated by any existing skyline point and, therefore, becomes part of the skyline immediately. Furthermore, it dominates g and j (stored in DB_{sky} and DB_{rest}, respectively), which are evicted from the system. Fig. 6d illustrates the final skyline.

Eager maintains an event list EL that contains entries of the form $e =< e.ptr, e.t, e.tag >$. Field $e.ptr$ is a pointer to the tuple involved in the event, $e.t$ specifies the event time (i.e., when the event will happen), and $e.tag$ indicates the event type. Specifically, if the tuple r referenced by $e.ptr$ belongs to the skyline currently, then $e.tag = {}'\mathrm{EX}'$ (i.e., a keyword indicating the expiry of a point) and $e.t = r.t_{exp}$. Otherwise, $e.tag = {}'\mathrm{SK}'$ (i.e., indicating the future inclusion of r in the skyline) and $e.t = r.t_{sky}$.

We are now ready to clarify the procedures carried out by the preprocessing and maintenance modules of *Eager*. After receiving a tuple r, E-PM discards the data in DB (including DB_{sky} and DB_{rest}) dominated by r. Such data can be retrieved using a d-sided range query (as discussed in Section 3.1), whose search region is $r.DR$. Then, E-PM computes the skyline influence time $r.t_{sky}$ of r, which is equivalent to finding the maximum expiry time of the points in $r.ADR$, i.e., a *d-sided max search*. If r appears in the skyline immediately, E-PM inserts an event $<$ (a pointer to) $r, r.t_{exp}, \mathrm{EX} >$ into EL. Otherwise, the event inserted is $< r, r.t_{sky}, \mathrm{SK} >$.

E-MM, on the other hand, simply outputs events chronologically: An EX event causes the removal of a (just-expired) skyline tuple, while an SK event adds a point r to the skyline, together with a new EX event $< r, r.t_{exp}, \mathrm{EX} >$ in EL. Note that, at any time, each live tuple corresponds to *exactly one* entry in EL: Skyline tuples are associated with EX events, while the rest with SK events. Figs. 7 and 8 show the pseudocodes of these two modules.

To facilitate updating EL and identifying the next event (which has the smallest event time), we index the events in EL with a main-memory B-tree [16] on their event times. Each tuple r in DB is associated with a pointer to its event in this B-tree, so that when r is expunged from the system, its associated event can be efficiently found for removal. From the implementation perspective, *Eager* can manage the data in DB_{sky} and DB_{rest} using the same structure, as explained in detail later. We close this section by proving the correctness of *Eager* (a similar proof was omitted for *Lazy* since it is straightforward).

Lemma 2. *Eager correctly produces the skyline output stream.*

Proof. We aim at establishing two facts: 1) all skyline changes are captured by *Eager*, and 2) every skyline change produced by *Eager* is correct. To prove the first direction, let r be any tuple that ever belongs to the skyline, and t_1 (t_2) be the *actual* timestamps when r starts (stops) appearing in the skyline. We will show that *Eager* indeed outputs two pairs ($+r$, t_1) and ($-r$, t_2).

```
Algorithm E-PM //invoked for each new tuple
1.  r = the incoming record from $BF$
2.  issue a $d$-sided range query (with the dominance region $r.DR$ of $r$) to retrieve the tuples in $DB$
    dominated by $r$
3.  for each tuple $r'$ in the query result
4.      delete its event entry from $EL$, and discard $r'$ from the system
5.      if the event of $r'$ is of type EX then output ($-r'$, current time)
6.  insert $r$ into $DB$
7.  invoke a $d$-sided max search (using $r.ADR$) to obtain the skyline influence time $r.t_{sky}$
8.  if $r.t_{sky}$ equals the current time //$r$ belongs to the skyline now
9.      output ($+r$, current time)
10.     insert <(a pointer to) $r$, $r.t_{exp}$, EX> into $EL$
11. else insert <$r$, $r.t_{sky}$, SK> into $EL$
```

Fig. 7. Preprocessing module of *Eager*.

```
Algorithm E-MM //invoked when the current time equals the minimum event time in $EL$
1.  $e$ = the event with the minimum event time in $EL$
2.  $r$ = the record referenced by $e.ptr$
3.  delete $e$ from $EL$
4.  if $e.tag$ = EX
5.      output ($-r$, $e.t$) and expunge $r$ from the system
6.  else //$e.tag$ = SK
7.      output ($+r$, $e.t$) and insert <$r$, $r.t_{exp}$, EX> into $EL$
```

Fig. 8. Maintenance module of *Eager*.

For the case of ($+r$, t_1), we distinguish two possibilities about the relationship between t_1 and the arrival time $r.t_{arr}$ of r. If $t_1 = r.t_{arr}$, r belongs to the skyline as soon as it arrives, and ($+r$, t_1) is produced by E-PM (executed to handle the arrival of r). The other possibility $t_1 > r.t_{arr}$ means that, when r arrives, it is dominated by a set S of points in DB; nevertheless, until time t_1, r is not dominated by any incoming data (otherwise, by Lemma 1, r cannot appear in the skyline). Hence, t_1 equals exactly its skyline influence time computed by E-PM, which inserts into EL an SK-event of r with t_1 as the event time. This event is not eliminated before time t_1—such elimination requires an arriving tuple dominating r, which cannot happen as mentioned earlier. Hence, the event will be handled by E-MM at time t_1, which outputs ($+r$, t_1).

After r becomes a skyline tuple, an EX event is created in EL. Based on this, we proceed to prove that ($-r$, t_2) is also generated by *Eager*, starting with the case where t_2 is identical to the expiry time $r.t_{exp}$ of r. This indicates that r is not dominated by any subsequent tuple. As a result, its EX-event will be handled by E-MM at time $r.t_{exp}$, which produces ($-r$, t_2). If, on the other hand, t_2 is smaller than $r.t_{exp}$, r is dominated and expunged by a tuple r' arriving at t_2, in which case E-PM, executed to handle the arrival of t_2, outputs ($-r$, t_2).

It remains to prove the second direction: If *Eager* outputs a pair ($+r$, t) or ($-r$, t), then tuple r indeed starts or stops belonging to the skyline at time t. Recall that *Eager* generates ($+r$, t) only at the arrival time of r (i.e., $t = r.t_{arr}$) or its skyline influence time ($t = r.t_{sky}$). The former case happens if and only if r becomes part of the skyline as soon as it arrives. The second case means that, at time t, all the points dominating r have already expired, and r is not dominated by any tuple arriving after it; therefore, r appears in the skyline at time t. On the other hand, *Eager* produces ($-r$, t) only when it expires or is dominated by an incoming tuple. In both cases, r is no longer part of the skyline. □

4 Analytical Study

This section analyzes the proposed *Lazy* and *Eager* frameworks. We aim at identifying 1) the best framework (in terms of the *amortized cost* for processing a tuple) depending on the problem characteristics, and 2) efficient theoretical implementations.

The subsequent discussion considers that N tuples have been received in the stream. Assume that the skyline size at a *single* timestamp varies with time, and let s_{sky} be the maximum size, i.e., s_{sky} is the largest number of points in DB_{sky} at any timestamp. If a is the highest tuple arrival rate per timestamp, then the number of simultaneously live tuples is at most $W \cdot a$, where W is the number of timestamps in a sliding window. The value of s_{sky}, therefore, has an upper bound $W \cdot a$ in the *worst case*, but, as proved in [14], it is much smaller for practical data distributions. We first consider *Lazy* in the next section, and then study *Eager* in Section 4.2.

4.1 Analysis of *Lazy*

We first show a basic lemma.

Lemma 3. *For Lazy, every tuple r is inserted into (removed from) each of DB_{sky} and DB_{rest} at most once.*

Proof. The part of the lemma about DB_{sky} is equivalent to saying that no tuple can "become" a skyline point more than once. In fact, a tuple r ceases to be a skyline point if it expires or is dominated by an incoming tuple r'. In neither case can r reappear in the skyline (the latter case is not possible due to Lemma 1). On the other hand, *Lazy* may add a tuple to DB_{rest} only when the tuple arrives (using the L-PM module). This excludes the possibility of adding r to DB_{rest} twice. □

Given an incoming tuple r, L-PM performs a d-sided emptiness test to determine whether it is dominated by a skyline point; therefore, a total of N tests are performed in history. We use c_{empty} to represent the cost of the most expensive test. If r belongs to the skyline immediately, a d-sided range search is issued (to find the skyline tuples dominated by r). Let us denote the number of range queries as n_{arrsky} (the subscript implies "arriving skyline" points), and the execution time of the ith $(1 \leq i \leq n_{arrsky})$ query as $c^{lazy}_{range}[i]$.

The performance of *Lazy* is also affected by the number n_{skyexp} of skyline tuples that expired. For each expiration, L-MM obtains the skyline for the subset of the tuples (i.e., a "subskyline") in DB_{rest} dominated exclusively by the expiring skyline point. Therefore, totally n_{skyexp} subskylines are computed. Assume the cost of the ith $(1 \leq i \leq n_{skyexp})$ computation to be $c_{subsky}[i]$. Given the amortized time c_{updsky} and $c_{updrest}$ of an update (insertion/deletion) on DB_{sky} and DB_{rest}, respectively, we prove:

Theorem 1. *For Lazy, the amortized time of processing a tuple is*

$$O\left(c_{empty} + \frac{1}{N}\sum_{i=1}^{n_{arrsky}} c^{lazy}_{range}[i] + \frac{1}{N}\sum_{i=1}^{n_{skyexp}} c_{subsky}[i] + c_{updsky} + c_{updrest}\right). \tag{1}$$

Proof. Each term in the complexity bounds one "type" of cost incurred by *Lazy*, namely, the overhead of

1. an empty test,
2. a range search,
3. computing the subskyline after a skyline tuple expires, and
4. updating DB_{sky} and DB_{rest} (according to Lemma 3, every tuple is added/removed into/from DB_{sky} and DB_{rest} $O(1)$ times). □

The above result allows us to discuss the effectiveness of concrete implementations of *Lazy*. A simple approach, referred to as *LL-Lazy*, is to organize the data of DB_{sky} and DB_{rest} using linked lists. Let s_{rest} be the maximum number of *live* tuples in DB_{rest} at timestamps when a skyline point expires. We have:

Corollary 1. *For LL-Lazy, the amortized time of processing a tuple is $O(s_{sky} + \frac{n_{skyexp}}{N} \cdot s_{rest} \cdot (s_{sky} + \log^{\alpha}(s_{rest})))$, where $\alpha = 1$ for dimensionalities $d = 2$ or 3, and $\alpha = d - 2$ for $d > 4$.*

Proof. Both an emptiness test and a range search can be performed in time $O(s_{sky})$ by simply examining all the tuples in DB_{sky}. The set S at Line 6 of Fig. 5 can be computed in $O(s_{sky} \cdot s_{rest})$ time by examining each point in DB_{rest} against every current skyline point. Then, using the algorithm by Kung et al. [22], we can compute a subskyline in $O(|S| \cdot \log^{\alpha}|S|)$ time, where $|S|$ (the size of S) is at most s_{rest}, and α as defined in the corollary. The update cost c_{updsky} and $c_{updrest}$ equal $O(1)$. Therefore, the complexity in the corollary follows (1) by replacing 1) c_{empty} and each $c^{lazy}_{range}[i]$ $(1 \leq i \leq n_{arrsky})$ with s_{sky}, 2) each $c_{subsky}[i]$ $(1 \leq i \leq n_{skyexp})$ with $s_{sky} \cdot s_{rest} + s_{rest} \cdot \log^{\alpha}(s_{rest})$, and 3) c_{updsky} and c_{updsky} with $O(1)$. □

Note that s_{rest} is no more than $W \cdot a$ (the largest number of live tuples at a timestamp). Hence, the above complexity can be written as $O(s_{sky} + \frac{n_{skyexp}}{N} \cdot (W \cdot a) \cdot (s_{sky} + \log^{\alpha}(W \cdot a)))$. $W \cdot a$ is a constant independent of N; thus, if the history is long enough, N will become arbitrarily larger than $(W \cdot a) \cdot \log^{\alpha}(W \cdot a)$. Therefore, *LL-Lazy* is efficient when n_{skyexp} is significantly lower than N, or equivalently, *only a small number of skyline points stay in the system until their expiration*. For example, assume that initially all the live tuples lie near the max corner (as in Fig. 3) of the data space. As time progresses, the data distribution moves slowly toward the origin of the data space, i.e., the coordinates of new tuples tend to be smaller than those of the old ones. Consequently, an existing skyline point has a high chance of being dominated by an incoming tuple, in which case the point is discarded before its expiration. Therefore, the resulting n_{skyexp} is expected to be small. When $n_{skyexp} \cdot (W \cdot a) \cdot \log^{\alpha}(W \cdot a) < N$, the amortized processing time of *LL-Lazy* is essentially $O(s_{sky})$.

If s_{sky} is large (i.e., a skyline includes many tuples), *Lazy* can be improved by creating appropriate indexes on DB_{sky}, resulting in a different technique *I-Lazy*. Specifically, two dynamic indexes (supporting insertions and deletions in any order) are required for emptiness tests and range queries, respectively, both of which have been very well studied (see [1] for a good survey). In particular, a range search solution typically answers a query in time $Q_{range}(n) + k$, where n is the cardinality of the input data set, $Q_{range}(n)$ some function of n, and k the number of retrieved objects. The corresponding cost of an emptiness-test method is $Q_{empty}(n)$ (i.e., no output overhead "$+k$" since the result is merely a Boolean value). Let $U_{range}(n)$ and $U_{empty}(n)$ be the worst-case update complexities of the two structures.

We also need another structure on DB_{sky} to facilitate the retrieval of S before the subskyline computation (Line 6 of Fig. 5). To compute S, for each point r' in DB_{rest}, we check if it is dominated by r (the expiring skyline point). If yes, we execute a d-sided count query that returns the number of points in DB_{sky} falling in $r'.ADR$ (the antidominant region of r'); r' is added to S if and only if the query returns 1. Assuming that the structure answers such a count query in $Q_{count}(n)$ (n is the number of points indexed), and can be updated in $U_{count}(n)$ time, we have:

Corollary 2. *For I-Lazy, the amortized time of processing a tuple is*

$$O\left(Q(s_{sky}) + U(s_{sky}) + \frac{n_{skyexp}}{N}\right.$$
$$\left.(s_{rest} \cdot Q_{count}(s_{sky}) + \cdot s_{rest} \cdot \log^{\alpha}(s_{rest})\right),$$

where $Q(s_{sky}) = \max\{Q_{empty}(s_{sky}), Q_{range}(s_{sky})\}$, $U(s_{sky}) = \max\{U_{empty}(s_{sky}), U_{range}(s_{sky}), U_{count}(s_{sky})\}$, *and* α *is the same as in Corollary 1.*

Proof. This corollary follows (1) in a way similar to Corollary 1. The first notable difference is that S can now be computed in $O(s_{rest} \cdot Q_{count}(s_{sky}))$ time. The second difference is that $c^{lazy}_{range}[i]$ $(1 \le i \le n_{arrsky})$ should be bounded by $Q_{range}(s_{sky}) + k_i$, where k_i is the number of skyline points extracted by the ith range query. Since such points are then expunged from the system, they will not be retrieved by another range query, leading to $\sum_{i=1}^{n_{arrsky}} k_i \le N$. Therefore, the second term (in (1)) can be replaced by $\frac{1}{N}(n_{arrsky} \cdot Q_{range}(s_{sky}) + N)$ which, in turn, can be simplified to $Q_{range}(s_{sky})$. □

I-Lazy has numerous instances with distinct tradeoffs among space, query, and update performance by selecting different structures for *d*-sided emptiness tests, range search, and count queries. In 2D space, for example, all operations can be solved with a *priority search tree* [5] in $O(\log(s_{sky}))$ time. The tree occupies $O(s_{sky})$ space and can be updated in $O(\log(s_{sky}))$ time. The performance of the resulting *I-Lazy* improves the first term $O(s_{sky})$ in Corollary 1 to $O(\log(s_{sky}))$. As another example, in general, *d*-dimensional space, this term becomes $O((s_{sky})^{\frac{d-1}{d}})$ by utilizing the *O-tree* [19] to support the three operations. The space/update complexity an O-tree is identical to that of a priority search tree.

It is worth mentioning that the term $\frac{n_{skyexp}}{N} \cdot s_{rest} \cdot \log^{\alpha}(s_{rest})$ in Corollary 1 (for subskyline computation) is actually a rather pessimistic upper bound in practice. By maintaining an R-tree on DB_{rest}, it is possible to obtain the subskyline much faster in *most cases* using a set of heuristics [23] (as will be evaluated in our experiments). However, the worst-case cost of this heuristic approach is quadratic to s_{rest}, in which case the above term becomes $\frac{n_{skyexp}}{N} \cdot s^2_{rest}$ (i.e., worse than its form in Corollary 1).

4.2 Analysis of *Eager*

A result similar to Lemma 3 also holds for *Eager*:

Lemma 4. *Eager inserts every tuple r into (deletes it from) DB once. Furthermore, Eager adds to (removes from) the event list EL at most one SK and one EX event of r.*

Proof. The part about DB is trivial since only E-PM can insert r into DB, and this module is executed exactly once for r (when it arrives). Similarly, an SK event can only be produced by E-PM and, hence, it can be created only once. On the other hand, an EX event is generated when r becomes a skyline point. As in Lemma 3, no tuple can appear in the skyline twice; thus, at most one EX event exists for r. □

It is easy to see that *Eager* stores at most $O(W \cdot a)$ tuples because expired tuples are always removed. In fact, we can prove a much tighter bound, by defining a $(d+1)$-dimensional *extended space* that shares d axes with the original data space, and has an additional "inverse-time" dimension. Specifically, each tuple r in DB corresponds to a point $(r[1], r[2], \ldots, r[d], 1/r.t_{arr})$ in the extended space, where the first d components are the coordinates of r in the original space, and the last component is the inverse of the arrival time of r.

Theorem 2. *At any timestamp, Eager retains exactly the live skyline points in the extended space.*

Proof. We refer to the skyline in the extended space as "ext-skyline" (to distinguish it from the skyline in the original space). We first prove that *Eager* does not store any tuple r that is not part of the ext-skyline. In fact, since there exists an ext-skyline tuple r' that dominates r, we have 1) r' dominates r also in the original space, and 2) $r.t_{arr} < r'.t_{arr}$ (i.e., r' arrived after r). Therefore, r must have been expunged by E-PM at the arrival time of r'.

It remains to establish that each tuple r in the ext-skyline is indeed retained by *Eager*. This is true because a live r is expunged if and only if it is dominated (in the original space) by another tuple r' arriving after it. This requires that r' should dominate r in the extended space, which contradicts the fact that r belongs to the ext-skyline. □

Given an arriving tuple r, E-PM issues a *d*-sided max query to compute its skyline influence time, and a *d*-sided range query to retrieve the existing database tuples dominated by r (for elimination). Therefore, a total of N max and range searches are performed, and their cost is assumed to be bounded by c_{max} and c^{eager}_{range}, respectively. Given the worst-case time c_{upddb} of updating DB, we have:

Theorem 3. *For Eager, the amortized time of processing a tuple is* $O(c_{max} + c^{eager}_{range} + c_{upddb} + \log(W \cdot a))$.

Proof. The cost of one execution of E-PM is bounded by $O(c_{max} + c^{eager}_{range})$. According to Lemma 4, during the lifespan of r, E-MM is invoked for it at most twice (handling its EX and SK events, respectively). Each application of E-MM expunges (at most) one tuple in time c_{upddb}, and performs $O(1)$ updates to EL. Since EL is managed with a main-memory B-tree (see Section 3.1), updating EL requires $O(\log(W \cdot a))$ cost, noting the fact that the size of EL is no more than $W \cdot a$. The complexity in the theorem results from the above analysis. □

We proceed to consider the concrete implementations of *Eager*, starting with *LL-Eager* that manages DB with a linked list. Since both *d*-sided max and range queries can be answered by scanning the entire DB once, the amortized per-tuple cost of *LL-Eager* equals $O(W \cdot a)$ (setting c_{max}, c^{eager}_{range}, and c_{upddb} in Theorem 3 to $O(W \cdot a)$, $O(W \cdot a)$, and $O(1)$, respectively). Similar to *I-Lazy*, we can design *I-Eager* which enhances the performance of *LL-Eager* with auxiliary structures on DB that facilitate *d*-sided max and range retrievals. We do not discuss the detailed derivation[1] since it is similar to that of Corollary 2.

1. It suffices to mention that the best-known method for *d*-sided max queries is due to Chazelles [8], who proposed a structure that consumes (in our context) linear $O(W \cdot a)$ space, answers a query in $O(\log^{1+\varepsilon}(W \cdot a))$ time (ε is an arbitrarily small constant), and can be updated in $O(\log^3(W \cdot a) \log\log(W \cdot a))$ cost.

```
Algorithm emptiness-test($N$, $r$) /* check if any point lies in $r.DR$; $N$ is the R-tree node
being considered */
1.  if $N$ is a leaf node
2.    return false if any data point in $N$ falls in $r.ADR$, or true otherwise
3.  else
4.    if $r.ADR$ completely covers an edge of any MBR in $N$ then return false;
5.    else
6.      sort the entries in $N$ by the "overlapping percentages" between their MBRs and $r.ADR$
7.      for each entry $E$ in the list (in descending order)
8.        if emptiness-test($E.child$, $r$) = false then return false //recursive execution
9.      return true;
```

Fig. 9. Using an R-tree to answer a *d*-sided emptiness query.

Comparing the performance of *Lazy* and *Eager*, it is clear that *Lazy* is expected to incur less processing overhead when the skyline size at a timestamp is small, or the number (n_{skyarr}) of "skyline expirations" is low. In these cases, its per-tuple processing time is decided by the preprocessing module, which is highly efficient. On the other hand, *Eager* upper bounds the processing time within $O(W \cdot a)$ in *all cases* (the time is even shorter if indexes are adopted). In the next section, we explore an alternative implementation of these two frameworks using R-trees.

5 Practical Implementations with R-Trees

The *I-Lazy* and *I-Eager* in the previous section are based on "theoretical structures" that have attractive asymptotical performance in the worst case. Unfortunately, these methods may incur expensive *actual* (space, query, and/or update) overhead, due to the "hidden constants" in their complexities. Furthermore, they are complex and demand considerable development effort. In the sequel, we discuss the implementations of both frameworks using (main-memory) R-trees [4], [20]. It is well known [27], [1] that although the R-tree does not have provable performance guarantees, it performs reasonably well for real-world data.[2]

I-Lazy maintains two R-trees on DB_{sky} and DB_{rest}, respectively. The first tree is used to perform emptiness tests and range queries in L-PM, while the second one is for efficiently finding a subskyline (Line 6 of Fig. 5). Since the algorithms for range search and obtaining a subskyline can be found in [17] and [23], respectively, we discuss only the emptiness test.

Fig. 9 illustrates the pseudocode *emptiness-test*, which is analogous to range search, except that the search terminates as soon as a point (in DB_{sky}) is found in $r.ADR$ (r is the newly arrived tuple). At each intermediate node N, *emptiness-test* checks whether $r.ADR$ completely covers an edge (or a $(d-1)$-dimensional rectangle in general *d*-dimensional space) of any minimum bounding rectangle (MBR) E in N. In this case, at least one point in the subtree of E must appear in $r.ADR$ and, hence, the algorithm finishes immediately (returning "false"). Otherwise (no edge of any MBR in N is contained in $r.ADR$), *emptiness-test* accesses (the child nodes of) the entries of N in descending order of the "overlapping percentages" between their MBRs and the query region. Specifically, if α is the area of a MBR, and β the overlapping area between the MBR and $r.ADR$, the overlapping percentage equals β/α. The rationale is that a MBR with a high overlapping percentage is expected to have a high chance of enclosing a point in $r.ADR$ (which will lead to an early termination).

Eager, on the other hand, maintains a single R-tree on all the data of DB. Fig. 10 demonstrates an R-tree on the database instance of Fig. 6a ($W = 15$) at timestamp 13 (the numbers in the brackets indicate the arriving times of individual tuples), assuming a node capacity of 3. In addition to an MBR, an intermediate entry E also stores a number $E.t_{exp}^{max}$ (in the parentheses) equal to the maximum expiry time of the points in its subtree. For instance, $E_1.t_{exp}^{max}$ equals 22, the expiry time of d. Also, notice that each leaf entry of the R-tree is linked (with bidirectional pointers) to its event in EL (indexed by a B-tree). The R-tree can be used to accelerate (*d*-sided) max queries (for finding the skyline influence time $r.t_{sky}$), and range search (for retrieving the objects dominated by an incoming tuple r). In the sequel, we focus on the computation of skyline influence time.[3]

Fig. 11 presents the algorithm *skyline-time* based on the "best-first" framework [18]. We illustrate its functionality using point h received by the system at time 15 as an example (see Fig. 10). At the beginning, the algorithm sets $h.t_{sky}$ to the current time 15 (which is the smallest possible value for $h.t_{sky}$), and visits the root of the R-tree. Since the MBR of E_1 is totally contained in the antidominance region $h.ADR$ of h (implying that all records under E_1 dominate h), the child node of E_1 is not accessed, but $h.t_{sky}$ is set to $E_1.t_{sky}^{max} = 22$. The MBRs of the other root entries E_2, E_3 partially intersect $h.ADR$, and are inserted in a heap H, sorted in descending order of their t_{sky}^{max}: $H = \{<E_3, 28>, <E_2, 24>\}$. *Skyline-time* then deheaps the first entry E_3, and accesses its child node N_3, where a tuple f dominating h is found. Since the expiry time 26 of f is larger than the current value 22 of $h.t_{sky}$,

2. Since our goal is simply to demonstrate the speedup achieved by indexing, we use a straightforward adaptation of R*-trees to main memory. Specialized main-memory implementations of R-trees (e.g., cache-conscious) or other multidimensional access methods (especially for high-dimensional spaces, where the performance of the R-tree deteriorates) may yield better results.

3. Zhang and Tsotras [29] develop an algorithm that uses R-trees to perform "max-range" search. This algorithm is of limited use in our case because it targets rectangle objects.

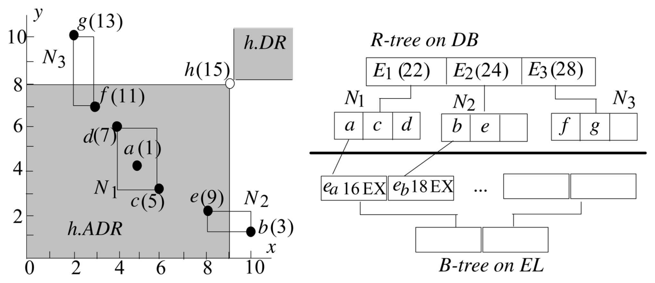

Fig. 10. A main-memory R-tree on DB.

$h.t_{sky}$ is updated to 26. Then, the processing terminates (without retrieving the child of E_2) because $E_2.t_{sky}^{max} = 24$ is smaller than the current $h.t_{sky}$, namely, even if a point in the subtree of E_2 dominates h, it cannot affect $h.t_{sky}$.

6 Experiments

This section experimentally evaluates the efficiency of the proposed techniques, using a Pentium IV 2.5GhZ CPU. For *Lazy*, we consider two implementations *LL-Lazy* and *I-Lazy*, which manage the data in DB_{sky}/DB_{rest} with a linked list and an R*-tree, respectively. *LL-Lazy* computes a subskyline using an adapted version of the SFS method [9] (that presorts only the points exclusively dominated by the expiring skyline tuple), while *I-Lazy* adopts the "constrained-skyline" algorithm of [23]. Similarly, for *Eager*, we examine *LL-Eager* and *I-Eager*. In all cases, each node of an R-tree and a B-tree (for indexing the event list of *Eager*) occupies 512 bytes.

The data space consists of d dimensions whose domains have range [0, 1]. The distribution of the live data does not change with time. We consider two types of distributions popular in the skyline literature: *independent* and *anticorrelated* [21], [23]. Each tuple in an *independent* stream is a point whose d coordinates are obtained uniformly in their respective domains. An *anticorrelated* tuple, on the other hand, has the property that if its coordinate on one dimension is large, then with a high probability its coordinate on another axis is small (we refer the interested readers to [6] about the details of creating *anticorrelated* data).

TABLE 2
Average Skyline Size per Timestamp versus W ($d = 3$)

W	200	400	800	1.6k	3.2k
Independent	34	41	49	55	61
Anti-correlated	1566	2656	4186	6139	8452

TABLE 3
Average Skyline Size per Timestamp versus d ($W = 800$)

d	2	3	4
Independent	45	49	156
Anti-correlated	243	4186	7643

6.1 Amortized Performance

We first evaluate the amortized cost (processing time, space consumption) of each method. Since the result is independent of the network speed, we assume a low arrival rate of 10 tuples/second, and generate streams with various distributions (*independent* and *anticorrelated*), dimensionalities d (between 2 and 4) and sliding window lengths W (from 200 to 3.2k seconds). Each stream contains 500 windows so that the total number of tuples ranges from 1 to 16 million. Table 2 shows the average skyline size (in terms of the numbers of points) per timestamp as a function of W (for $d = 3$), while Table 3 demonstrates the size as a function of d ($W = 800$). Note that *independent* data sets have much smaller skylines than *anticorrelated* ones. Furthermore, the dimensionality has a stronger impact on skyline sizes than the number of the live tuples.

6.1.1 Processing Time

In the first set of experiments, we fix d to 3, and compare the per-tuple processing time of *LL-Lazy*, *I-Lazy*, *LL-Eager*, and *I-Eager* for different values of W. Figs. 12a and 12b illustrate the results for *independent* and *anticorrelated* data sets, respectively. In all cases, the indexed versions incur lower

```
Algorithm skyline-time($r$) //obtain the skyline influence time $r.t_{sky}$ of an arriving tuple $r$
1.  initialize a max-heap $H$ which accepts entries of the form $<E, key>$
2.  $r.t_{sky}$ = current time
3.  for each root entry $E$ in the R-tree root
4.    if $E$ intersects $r.ADR$ and $E.t_{exp}^{max} > r.t_{sky}$ then insert $<E, E.t_{exp}^{max}>$ to $H$
5.    if $E$ is covered by $r.ADR$ and $E.t_{exp}^{max} > r.t_{sky}$ then $r.t_{sky} = E_{exp}^{max}$
6.  while $H$ is not empty
7.    remove the next entry $<E, E.t_{exp}^{max}>$ from $H$
8.    if $E.t_{exp}^{max} < r.t_{sky}$ then return // terminate
9.    if $E$ points to an intermediate node $N$
10.     for every entry $E'$ in $N$
11.       if $E'$ is covered by $r.ADR$ and $E'.t_{exp}^{max} > r.t_{sky}$ then $r.t_{sky} = E'.t_{exp}^{max}$
12.       else if $E'$ intersects $r.ADR$ and $E_{exp}'^{max} > r.t_{sky}$ then add $<E', E'.t_{exp}^{max}>$ to $H$
13.   else //$E$ points to a leaf node $N$
14.     for every record $r'$ in $N$
15.       if $r'$ dominates $r$ and $r'.t_{exp} > r.t_{sky}$ then $r.t_{exp} > r.t_{sky}$
```

Fig. 11. Using an R-tree to compute the skyline influence time.

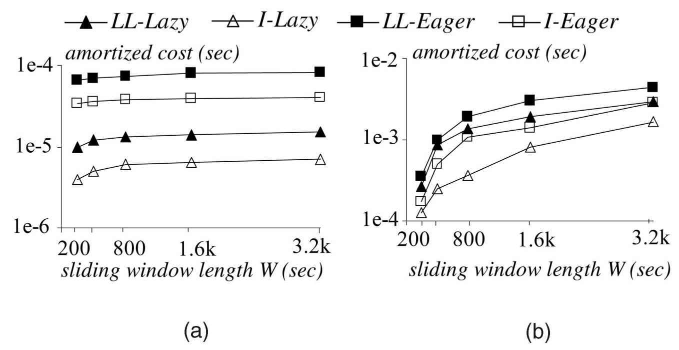

Fig. 12. Amortized cost versus W (d = 3). (a) *Independent*. (b) *Anticorrelated*.

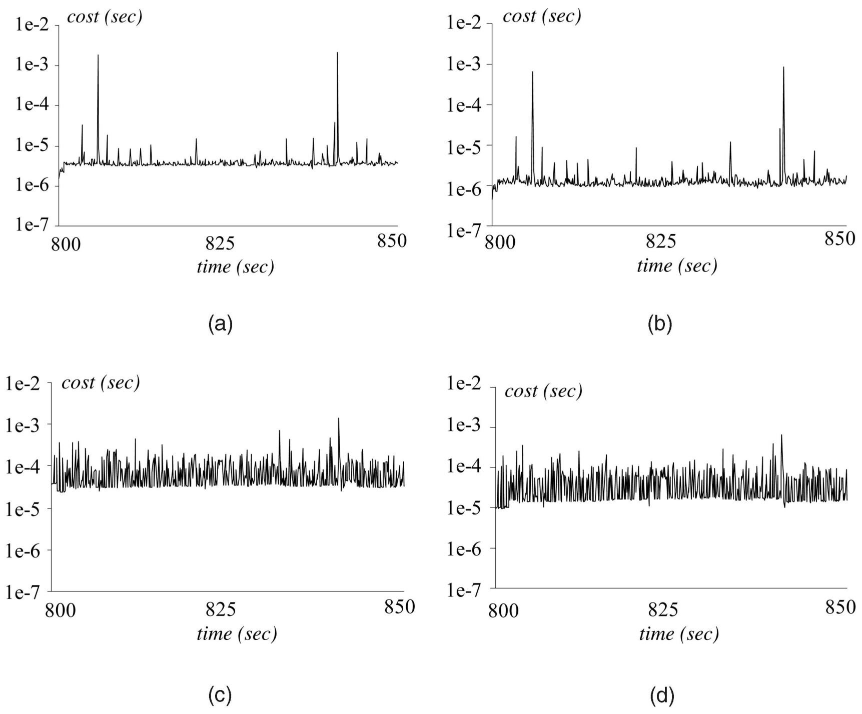

Fig. 13. Processing cost at individual operations (*independent*, $W = 800$, $d = 3$). (a) *LL-Lazy*. (b) *I-Lazy*. (c) *LL-Eager*. (d) *I-Eager*.

cost than their counterparts based on linked lists, confirming the importance of indexes in stream skyline monitoring. In general, *Lazy* outperforms *Eager* in both linked list and indexed implementations.

In order to explain the above phenomena, we select the streams with $W = 800$ (i.e., the median value in Fig. 12) as the representatives. Figs. 13a and 13b plot the cost of all the PM and MM execution for *LL-Lazy* and *I-Lazy* from the 800th to the 850th seconds (i.e., after the system has "warmed up"). The overhead of both methods at some timestamps (i.e., the "spikes") is considerably higher than their amortized cost. These are the timestamps when a skyline point expires, and *Lazy* invokes the expensive maintenance module (L-MM) to eliminate obsolete data and compute a subskyline. The spikes of *I-Lazy* are shorter because it calculates a subskyline using an R-tree [23], which is faster than the algorithm of [9] adopted by *LL-Lazy*.

Figs. 13c and 13d demonstrate the corresponding results for the two versions of *Eager*. The cost incurs less fluctuation, indicating that this framework handles every tuple with similar overhead. Notice that the overhead of *LL-Eager* (*I-Eager*) is significantly higher than that of *LL-Lazy* (*I-Lazy*) for most of the time, which explains their relative superiority in Fig. 12a. Fig. 14 presents the results of the same experiments for *anticorrelated*. The phenomena are analogous to those in Fig. 13, except that the spikes of *Lazy* disappear since it needs to scan a large number of tuples in DB_{sky} for every arrival (due to the frequent skyline changes).

In order to study the effects of dimensionality, in Fig. 15, we set W to 800 and measure the amortized cost as a function of d. The relative order of the four methods (in terms of efficiency) remains the same as in the previous experiments, but their performance deteriorates as the dimensionality increases. The deterioration of the linked-list implementations is due to the increase of skyline sizes (see Tables 2 and 3). For *I-Lazy* and *I-Eager*, an additional factor is that the effectiveness of R-trees drops with the dimensionality [27].

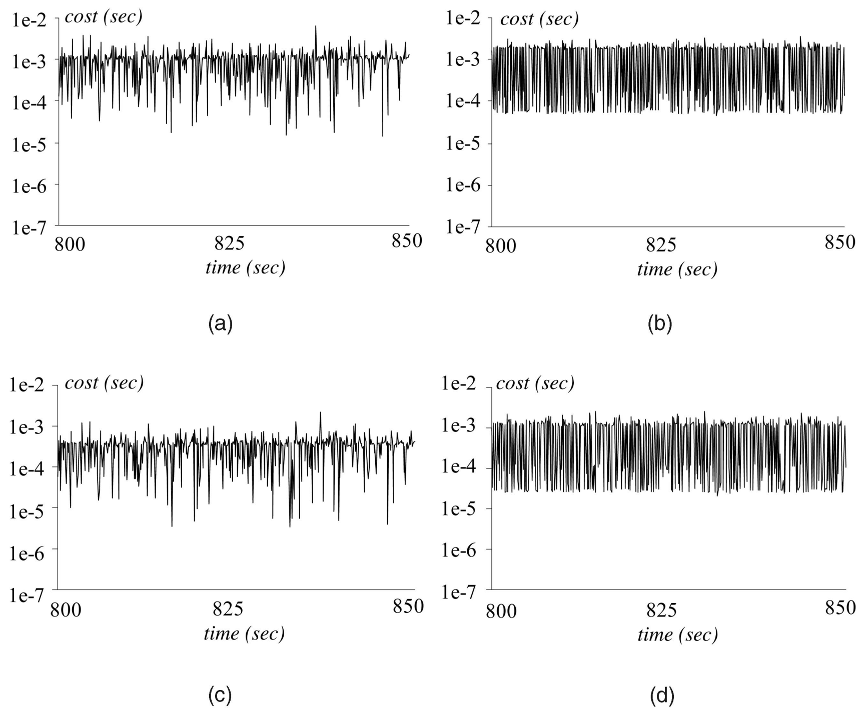

Fig. 14. Processing cost at individual operations (*anticorrelated*, $W = 800$, $d = 3$). (a) *LL-Lazy*. (b) *I-Lazy*. (c) *LL-Eager*. (d) *I-Eager*.

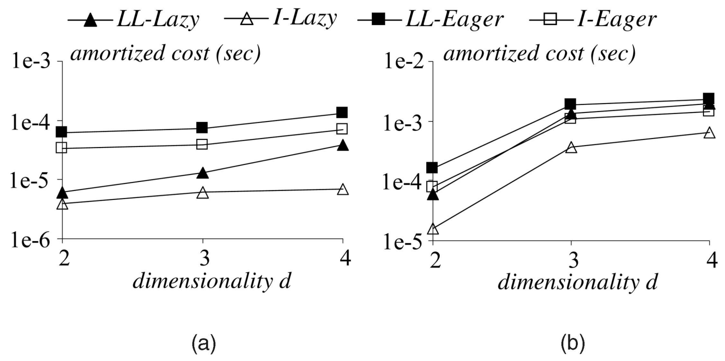

Fig. 15. Amortized cost versus d ($W = 800$). (a) *Independent*. (b) *Anticorrelated*.

6.1.2 Space Consumption

Fig. 16a demonstrates the average amount of memory (per timestamp) consumed by each method in the experiments of Fig. 12a (for *independent* data). It also shows the average number of tuples retained by *Lazy* and *Eager* for each value of W. Observe that *Lazy* stores a larger number of tuples than the window length W. This happens because a live tuple r is evicted by *Lazy* if and only if r is dominated (by a subsequent arrival) when it is in the skyline. This situation seldom happens because an *independent* skyline has a very small size and skyline changes are infrequent. *Eager*, on the other hand, maintains a fraction of the live data because it keeps only the tuples that participate in the skyline. Notice that the indexed implementations are only slightly larger than the corresponding nonindexed versions.

Fig. 16b plots the space overhead for the settings in Fig. 12b. Interestingly, although *Lazy* still retains more tuples (than *Eager*), it actually consumes less space. For *anticorrelated* data sets, both frameworks store a similar number of tuples, because most data will appear in the skyline and, therefore, must be retained. In addition, *Eager* needs to maintain an event list, whose size is comparable to that of the data tuples;[4] hence, it requires more memory than *Lazy*. Fig. 17 evaluates the space efficiency as a function of dimensionality, confirming our earlier findings.

6.2 Performance under Variable Arrival Rate

Having evaluated the amortized behavior of the proposed techniques, we proceed to examine their performance for realistic streams where the arrival rate varies with time, and a large amount of data may be received in a very short time interval. For this purpose, we fix the dimensionality d to 3, and create streams where tuples arrive in a "spiky" manner every 30 seconds. Specifically, from the first to the 29th seconds of each period, the time difference between two consecutive arrivals follows a Gaussian distribution with mean 0.1 (i.e., on average, 10 tuples per second) and variance 0.1. During the 30th second, the difference is obtained with another Gaussian distribution

4. Recall that, to maintain the event list, each tuple needs to keep a pointer to its event, which carries a timestamp and a pointer referencing the tuple. Hence, in d-dimensional space *Eager* needs at least $d + 3$ values to manage a point, as opposed to d of *Lazy* (i.e., just storing its coordinates).

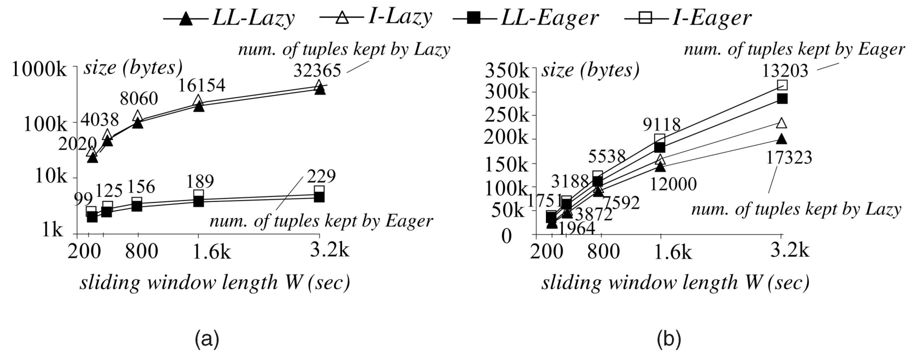

Fig. 16. Per-timestamp space overhead versus W ($d = 3$). (a) *Independent*. (b) *Anticorrelated*.

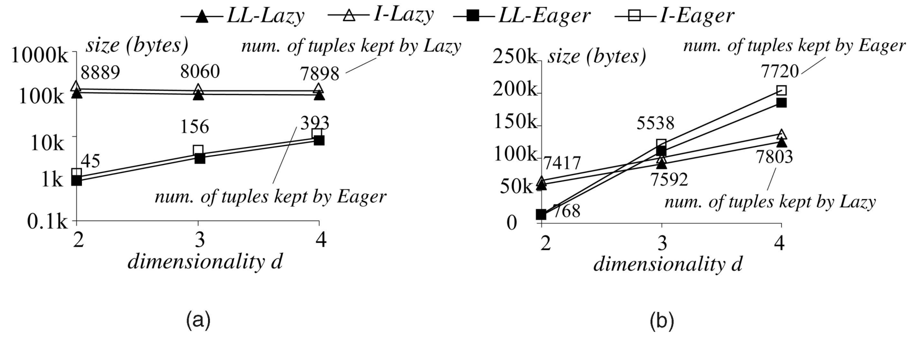

Fig. 17. Per-timestamp space overhead versus d ($W = 800$). (a) *Independent*. (b) *Anticorrelated*.

with mean 5×10^{-5} (i.e., half a million tuples per second) and variance 10^{-3}. As with the experiments of the previous section, the data distribution can be either *independent* or *anticorrelated*. In all cases, a stream lasts for 300 minutes, and a sliding window contains $W = 15$ seconds.[5]

Data must be buffered when they arrive faster than they can be processed. Fig. 18a shows the number of buffered tuples at the end of every second when *LL-Lazy* is used to process an *independent* stream. As illustrated on top of the diagram, the amortized cost (for the entire history) equals 1.4×10^{-5} seconds per tuple. Fig. 18b demonstrates the same results for *LL-Eager*. The buffer size of *LL-Lazy* is small (at most 6) at all times since it can usually finish handling the previous tuple before the next one arrives. On the other hand, the buffer size of *LL-Eager* surges to a large value every 30 seconds because its amortized cost 8.5×10^{-5} is higher than the expected interval (5×10^{-5}) between two arrivals in the spiky traffic.

In Figs. 18c and 18d, we present the memory consumption (including both the buffered and processed tuples) at various timestamps. The space overhead of *LL-Lazy* remains high (around 225k bytes) during the first 15 seconds of each period because, as shown Fig. 16, it needs to retain almost all the live tuples ($W = 15$). The memory consumption decreases after the 15th second because at this time the tuples received during the previous spiky traffic have expired and are eliminated from the system (i.e., only those data that arrived in the current period remain in the memory). The space cost of *LL-Eager* peaks at around 50k at the end of a period (when many data are buffered), and then quickly drops to below 10k after the buffer is cleared (recall that *LL-Eager* keeps only the tuples that may become part of the skyline).

Fig. 19 demonstrates similar results for *anticorrelated* data, confirming the above observations. Unlike Fig. 18, however, *LL-Lazy* also needs to buffer a large number of tuples at the end of each period since its PM becomes more expensive due to the increased skyline size. Furthermore, *LL-Eager* requires a larger amount of memory, which is consistent with the findings in Fig. 16b.

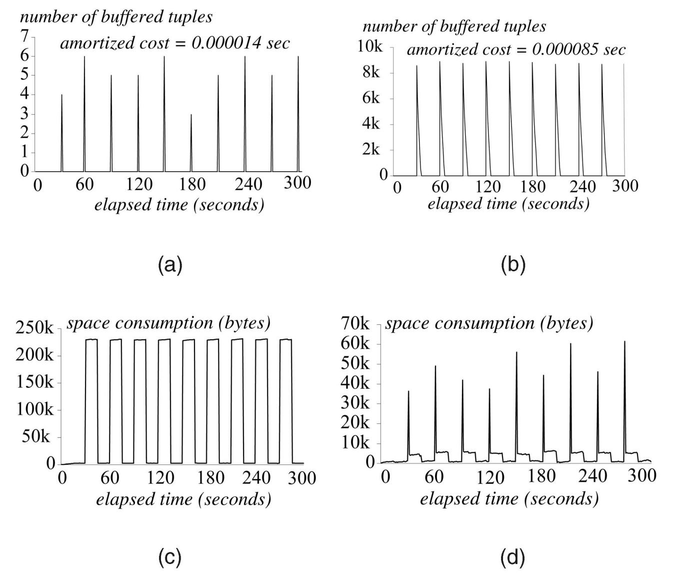

Fig. 18. Space overhead with time (*independent*, LL-implementations). (a) Buffer size (*LL-Lazy*). (b) Buffer size (*LL-Eager*). (c) Memory consumption (*LL-Lazy*). (d) Memory consumption (*LL-Eager*).

5. We set W to half of the time difference between two consecutive spiky arrivals so that the number of simultaneously live tuples varies significantly in the same period. Specifically, during the first 15 seconds of a period, there are more than 500k live points, while the number drops to around 150 after the 15th second.

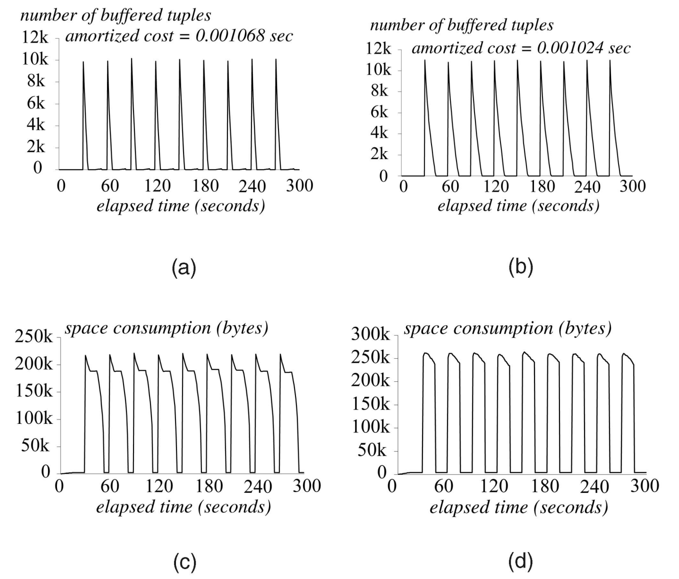

Fig. 19. Space overhead with time (*anticorrelated*, LL-implementations). (a) Buffer size (*LL-Lazy*). (b) Buffer size (*LL-Eager*). (c) Memory consumption (*LL-Lazy*). (d) Memory consumption (*LL-Eager*).

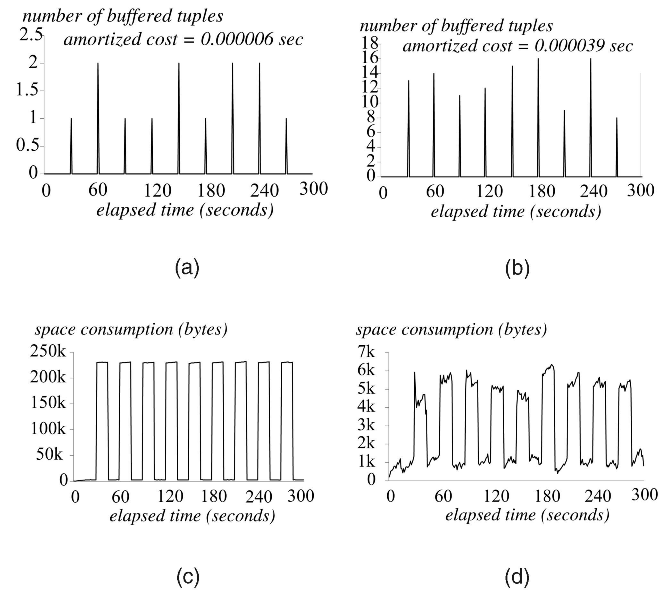

Fig. 20. Space overhead with time (*independent*, I-implementations). (a) Buffer size (*I-Lazy*). (b) Buffer size (*I-Eager*). (c) Memory consumption (*I-Lazy*). (d) Memory consumption (*I-Eager*).

Next, we repeat the same experiments using the indexed versions of the two frameworks. Figs. 20 and 21 demonstrate the results for *independent* and *anticorrelated* distributions, respectively. The phenomena are almost identical to those in Figs. 18 and 19, except that the buffer sizes of both algorithms are smaller. For *independent*, the space overhead of *I-Eager* does not "surge" at the end of each period as in Fig. 18b because here the buffer consumes a negligible amount of memory.

In summary, *Lazy* requires less CPU computation than *Eager* (in terms of both indexed and nonindexed implementations). *Eager*, however, achieves balanced performance in the sense that it incurs small processing cost for every tuple (as opposed to the spikes of *Lazy* in Fig. 13). Furthermore, this framework requires much smaller space for *independent* data, and 2D streams of both distributions. The indexed versions of both frameworks have extremely low amortized overhead (below 5×10^{-4} seconds/tuple) and, hence, they can support very fast streams. We show that with around 250k bytes memory, all solutions are able to handle highly spiky traffic where 10^5 tuples are received in one second.

7 Conclusions

In this paper, we proposed two algorithmic frameworks for continuously monitoring skyline changes over stream data,

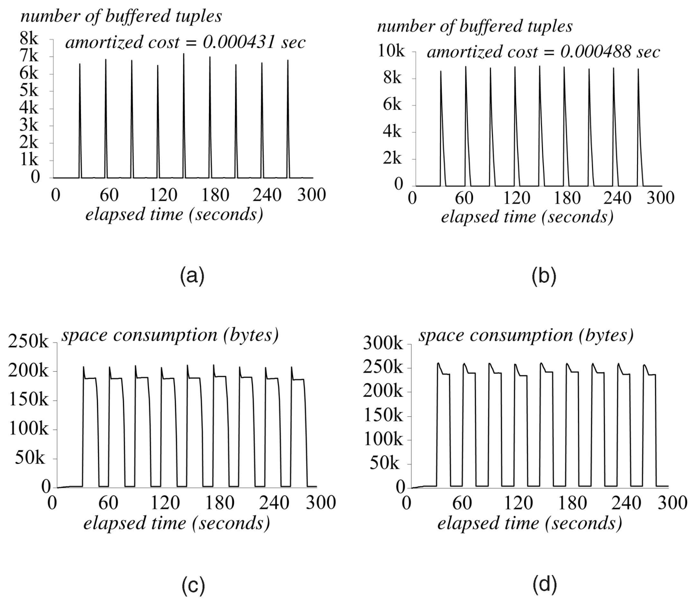

Fig. 21. Space overhead with time (*anticorrelated*, I-implementations). (a) Buffer size (*I-Lazy*). (b) Buffer size (*I-Eager*). (c) Memory consumption (*I-Lazy*). (d) Memory consumption (*I-Eager*).

based on several interesting characteristics of the problem. We accompanied our algorithms with in-depth performance analysis that reveals valuable insight into their respective behavior. A possible direction for future work concerns extending our methods to other forms of skyline retrieval as discussed in [23]. For example, a "top-k skyline" extracts only the k skyline tuples maximizing a user's preference function and, therefore, its computation (on streams) would require even less data in memory than our current solutions (for conventional skylines).

It would be interesting to incorporate the skyline operator into an integrated system, which involves operator scheduling, resource allocation, and load shedding. In this work, we focused on append-only "sliding-window" streams, while skyline maintenance on other types of streams constitutes an interesting open problem. Finally, we plan to investigate load shedding techniques for extremely fast streams. This is a particularly demanding problem because, as the proposed algorithms can handle very spiky traffic (up to 10^5 tuples per second as shown in the experiments), the shedding method must be very fast in order to be meaningful in practice (thus, excluding techniques based on complex distance computations, sorting, etc).

Acknowledgments

Yufei Tao was jointly supported by a grant (project NO: 9360088) and another SRG grant (project NO: 7001843) from the City University of Hong Kong, and RGC Grant CityU 1163/04E from HKSAR Government. Dimitris Papadias was supported by RGC Grant HKUST 6178/04E from the HKSAR Government.

References

[1] P.K. Agarwal and J. Erickson, "Geometric Range Searching and Its Relatives," *Contemporary Math.*, vol. 223, pp. 1-56, 1999.

[2] R. Ananthakrishna, A. Das, J. Gehrke, F. Korn, S. Muthukrishnan, and D. Srivastava, "Efficient Approximation of Correlated Sums on Data Streams," *IEEE Trans. Knowledge and Data Eng.*, vol. 15, no. 3, pp. 569-572, 2003.

[3] W.-T. Balke, U. Guntzer, and J.X. Zheng, "Efficient Distributed Skylining for Web Information Systems," *Proc. Int'l Conf. Extending Database Technology*, pp. 256-273, 2004.

[4] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles," *Proc. ACM SIGMOD*, pp. 322-331 1990.

[5] M. Berg, M. Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications.* Springer, 2000.

[6] S. Borzsonyi, D. Kossmann, and K. Stocker, "The Skyline Operator," *Proc. Int'l Conf. Data Eng.*, pp. 421-430, 2001.

[7] D. Carney, U. Cetintemel, A. Rasin, S.B. Zdonik, M. Cherniack, and M. Stonebraker, "Operator Scheduling in a Data Stream Manager," *Proc. Int'l Conf. Very Large Databases (VLDB)*, pp. 838-849, 2003.

[8] B. Chazelle, "A Functional Approach to Data Structures and Its Use in Multidimensional Searching," *SIAM J. Computing*, vol. 17, no. 3, pp. 427-462, 1988.

[9] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, "Skyline with Presorting," *Proc. Int'l Conf. Data Eng.*, pp. 717-719, 2003.

[10] A. Das, J. Gehrke, and M. Riedewald, "Approximate Join Processing over Data Streams," *Proc. SIGMOD*, pp. 40-51, 2003.

[11] R. Fagin, A. Lotem, and M. Naor, "Optimal Aggregation Algorithms for Middleware," *Proc. ACM Symp. Principles of Database Systems*, 2001.

[12] S. Ganguly, M.N. Garofalakis, and R. Rastogi, "Processing Set Expressions over Continuous Update Streams," *Proc. ACM SIGMOD*, pp. 265-276, 2003.

[13] J. Gehrke, F. Korn, and D. Srivastava, "On Computing Correlated Aggregates over Continual Data Streams," *Proc. ACM SIGMOD*, pp. 13-24, 2001.

[14] P. Godfrey, "Skyline Cardinality for Relational Processing," *Proc. Third Int'l Symp. Foundations of Information and Knowledge Systems (FoIKS)*, pp. 78-97, 2004.

[15] L. Golab and M.T. Ozsu, "Processing Sliding Window Multijoins in Continuous Queries over Data Streams," *Proc. Int'l Conf. Very Large Databases (VLDB)*, pp. 500-511, 2003.

[16] G. Graefe and P.-A. Larson, "B-Tree Indexes and CPU Caches," *Proc. Int'l Conf. Data Eng.*, pp. 349-358, 2001.

[17] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. ACM SIGMOD*, pp. 47-57, 1984.

[18] G.R. Hjaltason and H. Samet, "Distance Browsing in Spatial Databases," *Proc. ACM Trans. Database Systems*, vol. 24, no. 2, pp. 265-318, 1999.

[19] K.V.R. Kanth and A.K. Singh, "Optimal Dynamic Range Searching in Nonreplicating Index Structures," *Proc. Int'l Conf. Database Theory*, pp. 257-276, 1999.

[20] K. Kim, S.K. Cha, and K. Kwon, "Optimizing Multidimensional Index Trees for Main Memory Access," *Proc. ACM SIGMOD*, 2001.

[21] D. Kossmann, F. Ramsak, and S. Rost, "Shooting Stars in the Sky: An Online Algorithm for Skyline Queries," *Proc. Int'l Conf. Very Large Databases (VLDB)*, pp. 275-286, 2002.

[22] H.T. Kung, F. Luccio, and F.P. Preparata, "On Finding the Maxima of a Set of Vectors," *J. ACM*, vol. 22, no. 4, pp. 469-476, 1975.

[23] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "An Optimal and Progressive Algorithm for Skyline Queries," *Proc. ACM SIGMOD*, pp. 467-478, 2003.

[24] U. Srivastava and J. Widom, "Memory-Limited Execution of Windowed Stream Joins," *Proc. Int'l Conf. Very Large Databases (VLDB)*, pp. 324-335, 2004.

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

[26] K.-L. Tan, P.-K. Eng, and B.C. Ooi, "Efficient Progressive Skyline Computation," *Proc. Int'l Conf. Very Large Databases (VLDB)*, pp. 301-310, 2001.

[27] Y. Theodoridis and T.K. Sellis, "A Model for the Prediction of R-Tree Performance," *Proc. ACM Symp. Principles of Database Systems*, pp. 161-171, 1996.

[28] S. Viglas and J.F. Naughton, "Rate-Based Query Optimization for Streaming Information Sources," *Proc. SIGMOD*, pp. 37-48, 2002.

[29] D. Zhang and V.J. Tsotras, "Improving Min/Max Aggregation over Spatial Objects," *Proc. Ninth ACM Int'l Symp. Advances in Geographic Information Systems*, pp. 88-93, 2001.

Yufei Tao received the diploma from the South China University of Technology in August 1999 and the PhD degree from the Hong Kong University of Science and Technology in July 2002, both in computer science. After that, he spent a year as a visiting scientist at the Carnegie Mellon University. Since September 2003, he has been an assistant professor in the Department of Computer Science, the City University of Hong Kong. He is the winner of the Hong Kong Young Scientist Award 2002 conferred by the Hong Kong Institution of Science.

Dimitris Papadias is an associate professor in the Department of Computer Science at the Hong Kong University of Science and Technology (HKUST). Before joining HKUST, he worked at various places including, the Data and Knowledge Base Systems Laboratory at the National Technical University of Athens, Greece, the Department of Geoinformation at the Technical University of Vienna, Austria, the Department of Computer Science and Engineering at the University of California at San Diego, the National Center for Geographic Information and Analysis at the University of Maine, and the Artificial Intelligence Research Division at the German National Research Center for Information Technology (GMD).