Layered Processing of Skyline-Window-Join (SWJ) Queries using Iteration-Fabric

Mithila Nagendra [#1], K. Selçuk Candan [#2]

[#]*School of Computing, Informatics, and Decision Systems Engineering*
Arizona State University, Tempe, AZ 85287-8809, USA
{[1]mnagendra, [2]candan}@asu.edu

Abstract*— The problem of finding *interesting* tuples in a data set, more commonly known as the *skyline* problem, has been extensively studied in scenarios where the data is static. More recently, skyline research has moved towards data streaming environments, where tuples arrive/expire in a continuous manner. Several algorithms have been developed to track skyline changes over *sliding windows*; however, existing methods focus on skyline analysis in which all required skyline attributes belong to a single incoming data stream. This constraint renders current algorithms unsuitable for applications that require a real-time "join" operation to be carried out between multiple incoming data streams, arriving from different sources, before the skyline query can be answered. Based on this motivation, in this paper, we address the problem of computing *skyline-window-join* (SWJ) queries over pairs of data streams, considering sliding windows that take into account only the most recent tuples. In particular, we propose a Layered Skyline-window-Join (LSJ) operator that (a) partitions the overall process into processing layers and (b) maintains skyline-join results in an incremental manner by continuously monitoring the changes in all layers of the process. We combine the advantages of existing skyline methods (including those that efficiently maintain skyline results over a single stream, and those that compute the skyline of pairs of static data sets) to develop a novel *iteration-fabric* skyline-window-join processing structure. Using the *iteration-fabric*, LSJ eliminates redundant work across consecutive windows by leveraging shared data across all iteration layers of the windowed skyline-join processing. To the best of our knowledge, this is the first paper that addresses join-based skyline queries over sliding windows. Extensive experimental evaluations over real and simulated data show that LSJ provides large gains over naive extensions of existing schemes which are not designed to eliminate redundant work across multiple processing layers.

I. Introduction

Given a set, D, of data points in a feature space, the *skyline* of D consists of the points that are not dominated[1] by any other data point in D [1]. Intuitively, the skyline is a set of *interesting* points that help paint the "bigger picture" of the data in question, providing insight into the diversity of the data across different data features. Searching for non-dominated data is valuable in many applications that involve multi-criteria decision making [2]. For example, a stock investor might find the skyline of stock market transactions useful in making trade decisions. The number of shares (volume) and price per share are two attributes that are typically used to characterize stock transactions. In Figure 1, the points that are connected represent stock transactions that are part of the skyline; this includes transactions that are low-priced or/and have a large trade volume at a given point in time. The skyline in this example represents transactions that are more *interesting* than the rest of the transactions with respect to one or both criteria. Other transactions are not in the skyline because they are dominated in terms of price per share and/or volume by at least one transaction that is in the skyline. The shaded area in Figure 1 is the dominance region of stock transaction *b*: for any transaction in the region shown, *b* is either cheaper (per share) and/or has a higher volume; therefore *b* can be said to be more *interesting* than all transactions it dominates.

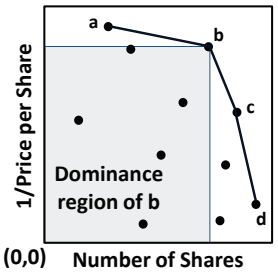


Fig. 1. Skyline of stock transactions

Due to its ability to summarize large data sets into small but *interesting* subsets, efficient processing of skyline queries has received considerable attention. Most early works focused on cases where the data is static [1], [3]. However, the advent of data stream applications has motivated the development of techniques to address the unique requirements of skyline query processing over data streams, where rather than computing a single skyline, the system continuously tracks skyline changes in an incremental manner. Several algorithms [4], [5], [6], [7], [8] that consider the special characteristics of streams, such as fast data updates and strict limits for response time, have been proposed. These techniques support on-line computation of skylines over rapid data streams and are able to efficiently monitor skyline changes as tuples arrive/expire continuously.

A. Skylines over Multiple Data Streams

Existing data stream skyline algorithms, nevertheless, focus on single-stream skyline processing in which all required

This work is supported by an NSF grant (#1116394 – "RanKloud: Data Partitioning and Resource Allocation Strategies for Scalable Multimedia and Social Media Analysis") and a KRF grant ("A Framework for Real-time Context Monitoring in Sensor-rich Personal Mobile Environments").

[1]A point dominates another point if it is as good or better in all dimensions, and better in at least one dimension.

skyline attributes are present in the same stream. However, there are many applications that require integration of the data from different sources (e.g., data originating from different sensors or from different sources in a distributed publish/subscribe architecture) and, in such scenarios, the data stream skyline query may involve attributes belonging to different data streams, making the join operation an integral part of the overall process. For instance, going back to our earlier example (Figure 1), a stock investor may also consider the risk of investing in a certain stock and the corresponding commission cost to be additional factors in his/her decision-making process. If this information is available as part of a second data stream, then the two streams would need to be "joined" before the skyline of stock transactions with respect to the price, volume, risk and commission cost can be found. Other join-based skyline applications can involve tracking objects through a network of sensors, or recording changes in large (but slow evolving) web-based data sets. Examples include generating click information about the number of visits to multiple web sites, or even monitoring the progress of cars through highway toll-booths based on specific attributes.

While the need to support skyline join queries over data streams is increasingly being recognized by the community [9], as of today, there are no algorithms that integrate skyline computation as a part of window-based join processing. As discussed earlier, existing techniques mainly focus on skyline processing over single-streams. These techniques can be used in conjunction with stream-join algorithms (by incrementally maintaining the join data using stream-join algorithms and then searching for skylines over the joined data stream), but in this paper we note that this approach will introduce significant amounts of waste as, it has been shown for static data [3], [10], performing a skyline search *after* a join operation is almost always less efficient than integrating the skyline search with the join processing.

B. Contributions of this Paper

This paper studies skyline-window-join (SWJ) query processing over multiple data streams. We propose a Layered Skyline-window-Join (LSJ) operator that maintains skyline-join results in a layered, incremental manner by continuously monitoring the changes in the data streams, and eliminates redundant work between consecutive windows by leveraging shared skyline objects across all iteration layers of skyline-join processing. Our contributions are as follows:

- Firstly, we formally define the *skyline-window-join* (SWJ) operation over multiple data streams.
- Next, we develop a framework called the *iteration-fabric* that forms the backbone of the proposed layered, incremental skyline-join algorithms over data streams.
- The *iteration-fabric* helps us combine the advantages of two existing skyline methods, *StabSky* [6] and *Iterative* [3], in developing a Layered Skyline-window-Join algorithm (LSJ) that maintains skyline-join results in an incremental manner by continuously monitoring the changes in the input streams and leveraging any overlaps that exist between the data considered at individual processing layers of consecutive sliding windows.
- We test the efficiency and performance of the proposed algorithms. Extensive experimental evaluations over real and simulated data show that LSJ provides significant gains, especially on data with correlated skyline attributes, over alternative schemes which are not designed to eliminate redundant work across skyline-join layers, especially in scenarios with large data volumes and with considerable overlaps between consecutive windows.

To the best of our knowledge, this is the first paper to address join-based skyline queries over pairs of data streams. The rest of the paper is structured as follows: in Section II, we give an overview of the existing work in the fields of data streams and skyline query processing. Section III presents the preliminaries and states the problem tackled in this paper. In Section IV, we discuss of a naive implementation of the SWJ operation, which highlights the underlying challenges and points out to the opportunities we leverage when developing the novel algorithms proposed in this paper. In Section V, we discuss the proposed algorithms in detail. Section VI presents an extensive experimental evaluation of the proposed approaches. We conclude the paper in Section VII.

II. Related Work

In this section, we provide an overview of the existing work in the fields of data streams and skyline query processing.

A. Data Streams

Over the last decade, we have witnessed a growth in the number of applications in which data arrives in a streaming manner and at high speeds [11]. These include financial applications that process streams of stock market or credit card transactions, telephone call monitoring applications that process streams of call-detail records, network traffic monitoring and sensor network applications that analyse environmental data gathered by sensors [12], [13]. These applications often require long-running, continuous queries as opposed to the traditional one-time queries. Thus, the advent of a wide array of stream-based applications has necessitated a push towards the development of algorithms that take into consideration the constant changes in stream environments.

B. Join Processing and Top-k Queries over Data Streams

Streaming algorithms for join processing are relevant to our research. [14] presents a symmetric hash join method that is optimized for in-memory performance. Following this, a plethora of techniques have been developed for efficiently processing join queries over data streams [15], [16], [17], [18], [19], [20]. Many of these works focus on eliminating, in the most efficient and effective manner, redundancy in join processing across consecutive time windows to maximize the output rate [15], [16]. Others focus on memory; they present join processing and load shedding techniques that minimize loss in accuracy when the memory is insufficient [17], [18].

[19] develops a novel *state-slice* sharing paradigm for window join queries. In particular, it proposes a method to slice window states into fine-grained slices that form a chain of sliced window joins to reduce the number of joins from quadratic to linear. [20] proposes an algorithm called *SCUBA* that utilizes dynamic clustering to optimize the execution of multiple continuous queries on spatio-temporal data streams.

Recent works on data streams also include investigation of richer query semantics such as skyline and top-k. As skyline is more related to the contributions of this paper, we present a detailed discussion of the literature on streaming skyline techniques in Section II-C.

Since the mid 2000s, there has been a plethora of work on processing streaming top-k queries. In an early work, Mouratidis *et al.* address the problem of answering continuous top-k queries over a single stream [21]. The authors propose two algorithms: (a) the *TMA* algorithm, which computes new answers to a query every time some of the current top-k results expire, and (b) the *SMA* algorithm that reduces top-k queries to k-skyband queries in order to avoid complete re-computation when some results expire. A very recent work in this area includes [22] which presents a framework called *MTopS* that handles multiple continuous top-k queries executed simultaneously against a common data stream. Other works on data streams examine top-k join processing [23], [24]. These primarily focus on maintaining the top-k join results and candidate lists incrementally, as the data streams and their scores evolve over time.

C. Skyline Processing over Data Streams

In the conventional setting of static data, there is a large body of work for both single-table skyline processing [1], [25], [26] and multi-table skyline-join processing [3], [10], [27]. These methods assume that the data is unchanging at the time of query execution and focus on computing a single skyline rather than continuously tracking skyline changes.

Recently, several algorithms have been developed to track skyline changes in stream environments. These methods are able to continuously monitor the changes in the skyline according to the arrival of new tuples and expiration of old ones. Sun *et al.* in [4] address skyline queries over distributed data streams, where the streams are derived from multiple horizontally split data sources. The authors develop an algorithm called *BOCS* that consists of an efficient centralized algorithm, *GridSky*, and an associated communication protocol to compute skylines in an incremental manner.

In [5], Das Sarma *et al.* propose a set of multi-pass data streaming algorithms that compute the skyline of a massive database with strong worst-case performance guarantees. The data stream in this context refers to the data objects in a database (residing on disk) that is read into and processed through the main memory in a streaming manner. The key contribution of this paper is a randomized multi-pass streaming algorithm called *RAND*. The paper shows that single pass algorithms under the sliding window model like [6], [7], [8] are too restrictive and proves that it is impossible to design an efficient skyline algorithm that reads each point exactly once.

Data stream skyline processing under the sliding window model is addressed in [6] and [7]. As stated earlier, in this environment, the skyline tends to keep changing with objects arriving and expiring while time passes. An important issue that needs to be addressed here is the expiration of skyline objects, i.e. how to replace expired skyline objects with their proper successor(s) without having to compute from scratch among objects that are exclusively dominated by the expired ones. To handle this problem, Tao *et al.* present the *Eager* algorithm [7] that employs an event list, while Lin *et al.* propose a method (*StabSky*) that leverages *dominance graphs* [6]. Both these methods are derived by memorizing the relationship between a current skyline object and its successor(s). Once skyline objects expire, their successor(s) can be presented as the updated skyline without any added computation.

Zhang *et al.* propose another technique under the sliding window model [8]. The authors develop an incremental method to address continuous, probabilistic skyline queries over sliding windows on uncertain data objects, which have probability thresholds. In [28], Jiang *et al.* look at a novel type of query analysis on time series data called interval skyline queries. These queries return a set of time series that is not dominated by any other time series in a given time interval. Lastly, Park *et al.* propose a computation framework called *TI-Sky* [29] that evaluates skyline queries over continuous time-interval streams. The time-interval model is more general than the sliding window model; here, unlike the sliding window model, each object in the stream has its own expiration time.

The above-mentioned approaches focus on skyline analysis in which the skyline attributes belong to a single stream, thus rendering them inapplicable to the problem being addressed in this paper. Very recently, Catania *et al.* proposed an algorithm to process so called *relaxed queries* over data streams [9]. The paper studies a specific case of skyline queries called *relaxation skyline* (r-skyline) queries and extends them to window-based join over data streams. Preliminary experimental results reported in [9] show that there is a need to design more efficient algorithms for r-skyline computation. In particular, mirroring our motivation in this paper, the authors conclude that there is a need for algorithms that consider skyline computation hand-in-hand with window-based join processing.

In this paper, we make a first attempt to solve the more general problem of answering *skyline-window-join* (SWJ) queries over data streams. The following section provides an introduction to join-based skyline queries over sliding windows.

III. Preliminaries

In this section, we formally introduce the *skyline-window-join* (SWJ) model of processing continuous, sliding window skyline-join queries over data streams. Table I summarizes the notations used throughout the paper.

A. Sliding Windows over Data Streams

This paper focuses on skyline-joins over pairs of data streams bound by time-based sliding windows.

TABLE I

NOTATIONS USED IN THIS PAPER

Notation	Description
S_i	stream $i \in \{1, 2\}$
$\mathcal{A}$	set of data attributes
$p.A$	value of attribute A of tuple p
Θ	join condition
$\mathcal{A}_\Theta$	set of join attributes
$\hat{A}$	set of skyline attributes
$p.start$	generation timestamp of tuple p
$p.end$	expiration timestamp of tuple p
ω	size of a sliding window
σ	sliding window shift length
W_j	sliding window j
$W_{i,j}$	data for stream $i \in \{1, 2\}$ in sliding window j
$W_{i,l,j}$	data for stream $i \in \{1, 2\}$ at layer l in window j
$W^+_{i,l,j}$	set of tuples added at layer l for stream $i \in \{1, 2\}$ in window j
$W^-_{i,l,j}$	tuple set removed at layer l for stream $i \in \{1, 2\}$ in window j
$\Lambda_{i,l,j}$	local skyline set of layer l for stream $i \in \{1, 2\}$ in window j
$\mathcal{N}_{i,l,j}$	non-redundant tuple set in layer l, window j, stream $i \in \{1, 2\}$
$\mathcal{T}_{i,l,j}$	tree-structured dominance graph built on $\mathcal{N}_{i,l,j}$
$\mathcal{G}_{l,j}$	global skyline set of layer l, window j
$\mathcal{G}_j$	global skyline set of window j

Sliding windows allow unbounded data streams to be limited to a certain size and finite number of states. As described in [11], the size of a window can be stated in terms of "logical" or "physical" units. *Time-based* sliding windows fall under the category of windows described using logical units. Each tuple, p, is *alive* in the system for a specific length of time; this is termed as the tuple's *lifespan* and is equal to $[p.start, p.end)$. Here, $p.start$ is the tuple's generation timestamp – the time at which the tuple was generated at the remote source. The expiry time, $p.end$, is the time when the tuple is removed from the window – this occurs when $p.start$ is no longer covered by the sliding window. The length/size of the sliding window is described by a parameter ω: each window W_j spans a time period from $[W_j.start, W_j.end)$, where $W_j.end = W_j.start + \omega$. In the *count-based* sliding window model, on the other hand, a tuple expires after ω subsequent tuples have been received, regardless of their generation timestamps. In other words, in this model, only the tuples with ω most recent timestamps are considered. A second parameter used in defining sliding windows over data streams is the "shift" or σ: In time-based sliding windows, for example, the shift constraints the start of the window to shift σ units; i.e., $\forall_j W_{j+1}.start = W_j.start + \sigma$. In count-based sliding windows, the shift defines how many tuples are skipped from one window to the next.

Note that count-based sliding windows can be treated as time-based windows by associating each tuple p with an artificial generation time, $p.start$, that equals its relative position in the stream (i.e., the first tuple arrived has position 1, the second 2, and so on). Therefore, in the rest of the paper, without loss of generality, we assume that the sliding windows are *time-based*, as opposed to *count-based*.

B. Continuous Joins over Sliding Windows

Given (a) two streams, $S_1(A_1, \ldots, A_{d1}, TS)$ and $S_2(B_1, \ldots, B_{d2}, TS)$, where $\mathcal{A} = \{A_1, \ldots, A_{d1}\} \cup \{B_1, \ldots, B_{d2}\}$ is the set of data attributes of the two streams and TS is the timestamp attribute, (b) sequences of windows, $\ldots W_{1,j} \ldots$ and $\ldots W_{2,j} \ldots$, on the two streams defined by[2] ω and σ, and (c) a join condition, Θ, on the join attribute set $\mathcal{A}_\Theta$, a continuous join-query over data streams seeks to maintain the set of join results $R_j = W_{1,j} \bowtie_\Theta W_{2,j}$.

C. Skyline-Window-Joins (SWJ) over Sliding Windows

We refer to the skyline-join operations over sliding windows as *skyline-window-join* (SWJ). A *skyline-window-join* operation over the data streams S_1 and S_2 seeks to maintain, for each window W_j, the subset of tuples in R_j consisting of tuples not *dominated* by any other tuple(s) in R_j.

Given two tuples p and q, we say that p *dominates* q *in the skyline attribute set* $\hat{A} \subseteq \mathcal{A}$ *(denoted as*[3] $p \rhd_{\hat{A}} q$*)*, if

$$\forall X \in \hat{A}, (p.X \succeq q.X) \wedge (\exists Y \in \hat{A} \mid p.Y \succ q.Y).$$

In other words, p dominates q if p is better than or equal to ($\succeq$) q in all dimensions and better than ($\succ$) q in at least one dimension of the skyline attribute set $\hat{A}$.

Definition 1 (Skyline-Window-Join, $\overset{sw}{\bowtie}$)**.** *Given (a) two streams S_1 and S_2, (b) sequences of windows, $\ldots W_{1,j} \ldots$ and $\ldots W_{2,j} \ldots$, on the two streams defined by ω and σ, (c) a join condition, Θ, and (d) a set of skyline attributes, $\hat{A}$, a tuple, p, is in the j^{th} skyline-join window, $S_1 \overset{sw}{\bowtie}_{\Theta,j,\hat{A}} S_2$, iff (a) $p \in R_j = W_{1,j} \bowtie_\Theta W_{2,j}$ and (b) $\nexists q \in R_j - \{p\}$ s.t. $q \rhd_{\hat{A}} p$.*

The tuple, p, consists of the concatenation of two tuples p_1 and p_2, where p_1 corresponds to a tuple in stream S_1 and p_2 corresponds to a tuple in S_2. Each skyline point, p, is alive *as long as p_1 and p_2 are* alive *in their respective streams.* $\diamond$

The following is an example of a skyline-window-join query:

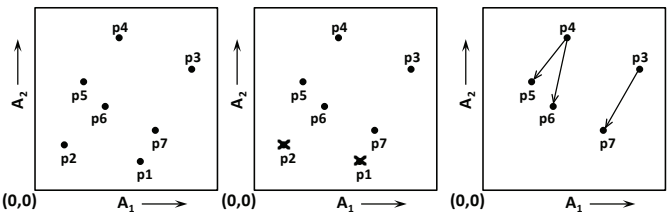
Example 1 (SWJ Query)**.** *Given two stock market transaction streams,* `Investment(stockID, price, volume, timestamp)` *and* `Risk(stockID, risk, cost, timestamp)`*, that contain information about stock transactions, the query:*

```
Skyline = SWJ * FROM Investment I, Risk R,
                WHERE I.stockID = R.stockID,
                I.timestamp within last 24h,
                R.timestamp within last 24h,
                1/price MAX, volume MAX,
                1/risk MAX, 1/cost MAX,
```

equi-joins the pair of streams on the join attributes A_Θ = {`Investment.stockID, Risk.stockID`}, within the window constraints of $\omega = 24$ hours, and returns results that are not dominated by any other results based on the skyline attributes $\hat{A}$ = {`price, volume, risk, cost`}. $\diamond$

[2]Note that, while in the more general case the two streams can have different ω and σ, for clarity, we limit the discussion for the case where the two streams have the same window characteristics.

[3]In the rest of the paper, whenever it is clear from the context, we omit references to $\hat{A}$ and use $p \rhd q$ to denote that p *dominates* q in the skyline attribute set $\hat{A}$; also $p \not\rhd q$ indicates that p *does not dominate* q.



(a) Data stream (b) Redundant tuples ($p1, p2$); Non-redundant tuples ($p3, \ldots, p7$) (c) Dominance graph

Fig. 2. (a) Tuples in the stream arrive the order $p1, p2, \ldots, p7$; (b, c) Data structures used by *StabSky* to process skyline queries over the given stream

In the above query, the underlying skyline preference function is MAX. Other possible annotations include MIN, where the dimensions are minimized, and DIFF, which denotes that two records with different values in a particular dimension may both be part of the skyline [1]. While any of these monotonic functions are acceptable as preference functions, in the rest of the paper, without loss of generality, we assume that MAX is specified as the preference function.

IV. A First Attempt to Processing SWJ Queries

In this section, we first explain *StabSky* [6] and *Iterative* [3] that together form a core part of the novel *iteration-fabric* framework. We then discuss a naive implementation of SWJ, highlight the underlying challenges, and point out the key observations leveraged to develop the `LSJ` algorithm.

A. The StabSky Algorithm

As mentioned in Section II-C, *StabSky* [6] is an algorithm that addresses skyline processing over a single data stream under the sliding window model. This algorithm is based on (a) minimizing the number of tuples kept in memory, and (b) effectively characterizing and encoding "critical" dominance relationships among the tuples in the data stream in order to precisely answer all n-of-m skyline queries. Here, m is the number of most recent tuples seen in a data stream and n ($n \leq m$) is any most recent tuples seen so far.

[6] proves that it is not necessary to maintain all possible dominance relationships among tuples in a data stream to precisely answer n-of-m skyline queries. Thus, it proposes a *dominance graph* – a *forest* structure whose edge set consists of only the *critical* dominance relationships among the tuples in the non-redundant set $\mathcal{N}$. A tuple p is said to be *redundant* with respect to the most recent m elements in the stream if p has expired (i.e. p is outside the most recent m elements) or is dominated by a younger tuple q (i.e. q arrives later than p).

Figure 2 illustrates the key features of *StabSky*. Figure 2(a) shows a stream of tuples that arrive in the order $p1, p2, \ldots$. In Figure 2(b), tuple $p1$ becomes *redundant* since it is dominated by a younger tuple $p7$. The solid black points shown in the figure belong to the set of *non-redundant* tuples, $\mathcal{N}$, and are the only tuples that need to be retained for skyline computation; the remaining tuples ($p1$, $p2$) are considered to be *redundant*.

A dominance relation $q \rightarrow p$ is defined to be *critical* if and only if q is the youngest tuple (but older than p) in $\mathcal{N}$ that dominates p. Figure 2(c) visualizes the *dominance graph*, $\mathcal{T}$, built on the $\mathcal{N}$ shown in Figure 2(b). Here, edge $p4 \rightarrow p6$ is *critical* and the only dominance relation maintained on $p6$ even though it is dominated by both $p3$ and $p4$.

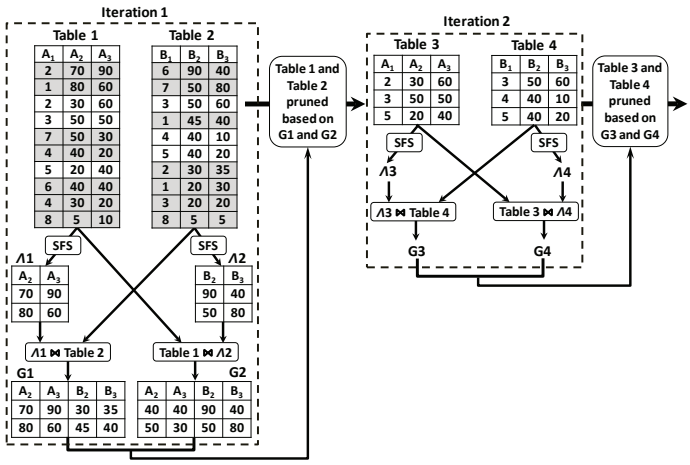


Fig. 3. Iterative process underlying the existing *skyline-join* operator

The *StabSky* algorithm uses the above-mentioned data structures to effectively process n-of-m skyline queries. [6] states that for a given n ($n \leq m$), a tuple p in a data stream is a skyline point for the n-of-m query if and only if (a) p is a root node in the current dominance graph, $\mathcal{T}$, or (b) there is a critical dominance edge $q \rightarrow p$ in $\mathcal{T}$ such that q arrives earlier than the n^{th} most recent tuple in the stream.

B. The Iterative Skyline-Join Algorithm

As discussed earlier, there is a large body of research in the area of static, multi-table skyline-join processing [3], [27], [10]. Among these, Sun *et al.* introduce a new operator called `skyline-join` [3] and two algorithms to support skyline-join queries. The first one extends the *Sort and Limit Skyline (SaLSa)* algorithm [26] to cope with multiple relations. The second algorithm called *Iterative* finds skyline results by pruning the search space iteratively. Overall, the *Iterative* algorithm is shown to perform well.

Figure 3 illustrates the *Iterative* algorithm. Here, Table 1 and Table 2 are the input tables, A_2, A_3, B_2, B_3 represent the skyline attributes, and the join is carried out on A_1 and B_1. *Iterative* first computes the local skyline ($\Lambda 1$, $\Lambda 2$) of the input tables. It then generates skyline results ($\mathcal{G}1$, $\mathcal{G}2$) of the skyline-join using $\Lambda 1$ and $\Lambda 2$. These results are used for pruning the input tables to obtain Table 3 and Table 4; pruned tuples are highlighted in Figure 3. This completes one iteration. The process is then repeated with Table 3 and Table 4 being used as inputs to the next iteration. This process continues until at least one of the input tables is completely eliminated.

C. Key Insights: Layers and Overlaps

It is easy to see that a simple way to execute skyline-window-join operations is to apply the iterative skyline-join algorithm for each window. Figure 4(a) visualizes the process. More importantly, however, we observe, that the consecutive

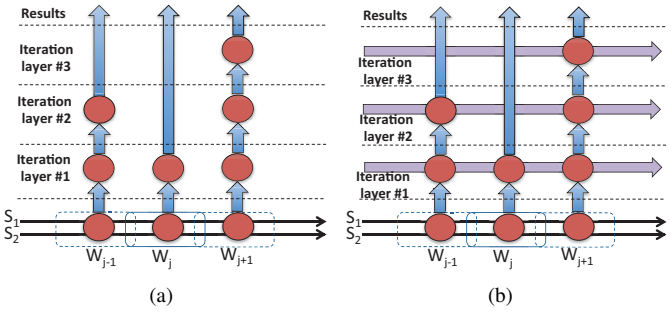


Fig. 4. (a) Executing skyline-window-join queries by applying the iterative skyline-join algorithm for each window; (b) Viewing layers as separate "virtual streams" that feed the upper layers of iteration

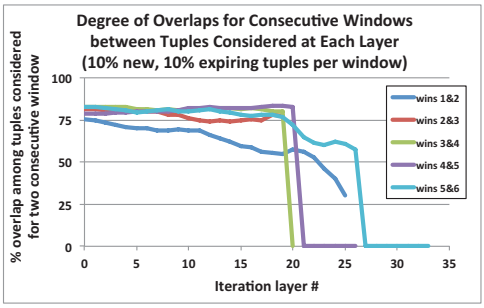


Fig. 5. Sample SWJ execution for 5 consecutive windows (10% new and 10% expiring tuples per window): the plots show that the skyline-join process iterate somewhere between 20 to 35 times for different windows and the overlaps (among consecutive windows) of tuples considered at different layers of iterations remain high across layers of iteration

iterations of the algorithm, spanning multiple windows, can be viewed as separate *iteration layers* (Figure 4(b)).

> Our key insight in this paper is that *overlaps exist not only at the lowest data layer (across consecutive data windows), but also at the individual iteration layers, where the tuples processed can be considered as "virtual streams" that evolve from one window to the next (see Figure 5 for a sample execution).*

Therefore, we argue that if we naively execute the skyline-window-join operation by applying the iterative skyline join algorithm separately for each window, we can end up with significant amount of redundant work. We further argue that if we can quickly identify and eliminate these per-layer overlaps, we can achieve significant savings in processing time.

Based on these insights, in the next section, we present a novel Layered Skyline-window-Join (LSJ) operator which avoids redundancies in skyline-join query processing by weaving together the consecutive windows and consecutive iteration layers into an *iteration-fabric* processing structure.

V. Layered Skyline-Window-Join (LSJ) Operator

In this section, we present an efficient Layered Skyline-window-Join (LSJ) operator for computing skyline-joins over sliding windows. LSJ leverages a novel *iteration-fabric* processing structure to identify and eliminate per-layer overlaps across consecutive windows.

A. Iteration-Fabric Processing Structure

Figure 6 gives an overview of the proposed *iteration-fabric* framework. As shown in the figure, for each window, LSJ applies an iterative skyline-join process: the data is passed to the lowest layer-module (or $\mathcal{L}$-module), where each $\mathcal{L}$-module

1) computes the local skylines of the input windows,
2) generates partial skyline-join results,
3) prunes the tuples in the input windows using the partial skyline-join result set, and
4) passes the pruned data to the $\mathcal{L}$-module at the next layer.

It is important, however, to note that the $\mathcal{L}$-modules are not only vertically connected across different layers of the same window, but also horizontally connected across consecutive windows of the same layer. This enables the iteration-fabric structure to identify and eliminate processing redundancies across all layers of the skyline-window-join processing[4].

As shown in Figure 6, each $\mathcal{L}$-module consists of two sub-modules, `M-LocalSky` and `M-Iterative`.

1) The `M-LocalSky` *Sub-Module:* At layer, l, of the j^{th} window, the `M-LocalSky` sub-module produces the local skyline sets, $\Lambda_{1,l,j}$ and $\Lambda_{2,l,j}$, of the tuples corresponding to the data sets $W_{1,l,j}$ and $W_{2,l,j}$, corresponding to the layer l of the j^{th} window, obtained from the $\mathcal{L}$-module at the $(l-1)^{th}$ layer of the j^{th} window.

As described in Section IV-B, each iteration of the *Iterative* algorithm [3] starts with finding the local skyline points of the input data to that iteration. The authors use the well-known *Sort-Filter-Skyline (SFS)* algorithm [25] to compute the local skyline points at each iteration. SFS, however, is applicable to scenarios in which the data is static. Therefore, while we can use SFS to compute local skylines only based on inputs in a given window, this would not enable us to leverage the overlaps in the data at the same layer in consecutive windows.

Instead, `M-LocalSky` generates the local skyline points, $\Lambda_{1,l,j}$ and $\Lambda_{2,l,j}$, not only by considering $W_{1,l,j}$ and $W_{2,l,j}$ passed from a lower layer, but also $W_{1,l,j-1}$ and $W_{2,l,j-1}$ from the previous window. Intuitively, tuples in $\Delta^{-}_{i,l,j} = W_{i,l,j-1} \backslash W_{i,l,j}$ are considered as the expiring tuples at the window j of layer l and $\Delta^{+}_{i,l,j} = W_{i,l,j} \backslash W_{i,l,j-1}$ are treated as the new tuples at the window j of layer l; here "\" is the set difference operator. Given these, we compute, $\Lambda_{1,l,j}$ and $\Lambda_{2,l,j}$, not using using a static skyline technique like SFS, but relying on a modified version of the on-line skyline techniques (*StabSky*) presented in [6]. Specifically, we adapt the methods proposed for processing n-of-m skyline queries over count-based sliding windows (see Section IV-A). In essence, the data $W_{1,l,*}$ and $W_{2,l,*}$ at layer l are treated as streams that evolve over time, and their local skyline sets at the j^{th} window (i.e, $\Lambda_{1,l,j}$ and $\Lambda_{2,l,j}$) are computed by taking into account the overlaps between the same layer of the *previous* and the current window being analyzed. The details of `M-LocalSky` are presented in Section V-B.

[4]This both horizontally and vertically connected iteration-fabric structure also provides opportunities for highly-parallel executions where different $\mathcal{L}$-modules are associated to different processing units (e.g., cores). We leave the investigation of this to future work.

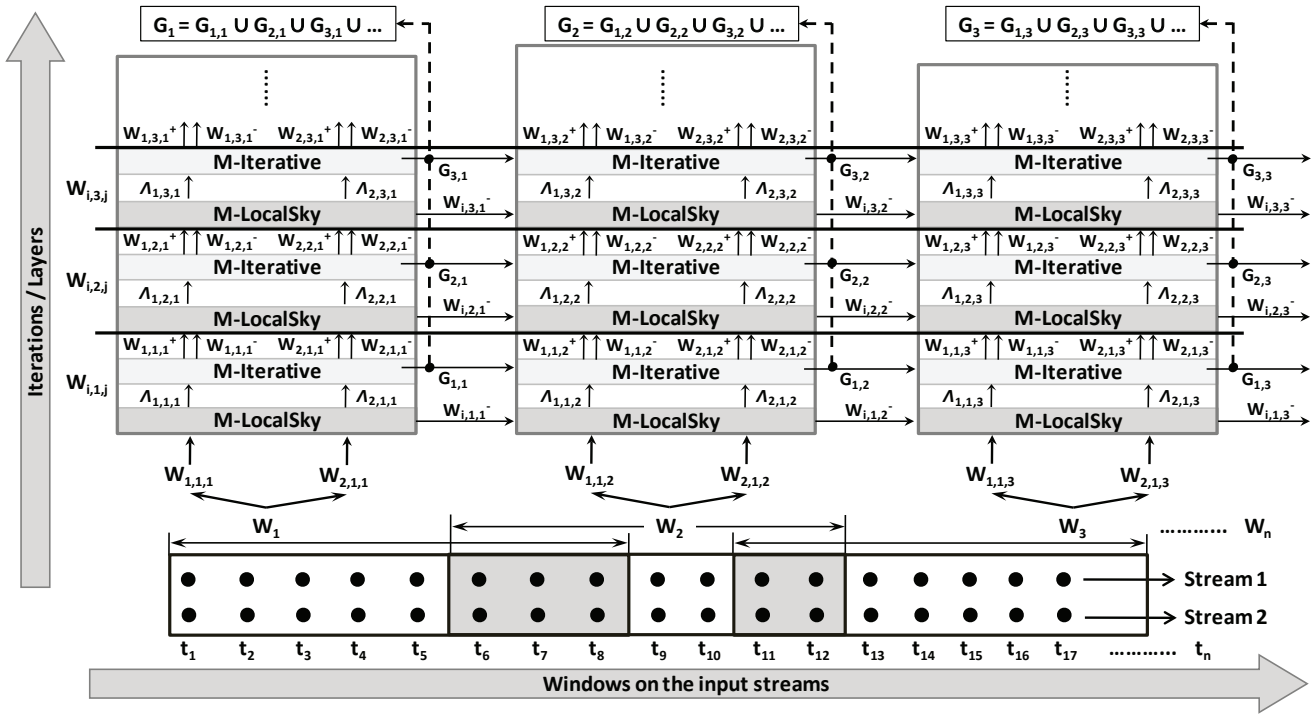


Fig. 6. Overview of the *iteration-fabric*, which weaves together layer-modules (or $\mathcal{L}$-modules, each consisting of two sub-modules, `M-LocalSky` and `M-Iterative`) into a grid across iterations and windows

2) The `M-Iterative` *Sub-Module:* The local skyline sets produced by `M-LocalSky` are pushed into the `M-Iterative` sub-module that uses an adapted version of *Iterative* [3] to produce the global skyline set, $\mathcal{G}_{l,j}$, of the layer l in the current window j. Once $\mathcal{G}_{l,j}$ is identified, `M-Iterative` then prunes the input data sets $W_{1,l,j}$ and $W_{2,l,j}$ and pushes the tuples that qualify to the next layer, $l+1$, as $W_{1,l+1,j}$ and $W_{2,l+1,j}$. The overall global skyline set, $\mathcal{G}_j$, of the skyline-join query over the current sliding window j is given by the union of the global skyline sets produced at each layer of the window. We discuss the details of the `M-Iterative` sub-module in Section V-C.

B. Local Skyline Computation Module (`M-LocalSky`)

We build the `M-LocalSky` component of the *iteration-fabric* based on a modified version of the *StabSky* algorithm [6], which maintains skylines of data streams that evolve over time. The proposed `M-LocalSky` module (Figure 7) computes the local skyline sets ($\Lambda_{1,l,j}$ and $\Lambda_{2,l,j}$) by taking into account the overlaps between the same layer of the *previous* and the current window being analyzed.

`M-LocalSky` implements a time-based sliding window version of the *StabSky* algorithm. This is achieved by replacing a tuple p's position label by its generation timestamp $p.start$. `M-LocalSky` maintains *non-redundant* tuples ($\mathcal{N}_{i,l,j}$) for each layer l of stream $i \in \{1,2\}$. It also maintains the corresponding *dominance graph* ($\mathcal{T}_{i,l,j}$) for each layer l built on the non-redundant tuples in $\mathcal{N}_{i,l,j}$. The dominance graphs are encoded into *intervals* and stored in *interval trees* using the scheme described in [6]. For example, the dominance graph in Figure 2(c) built on the tuples $p1, p2, p3, \ldots$ with generation timestamps $1, 2, 3, \ldots$, respectively, can be encoded as the intervals $(0,3]$, $(0,4]$, $(3,7]$, $(4,5]$, and $(4,6]$.

As shown in Figure 7, given the set of expiring ($\Delta^{-}_{i,l,j}$) and new tuples ($\Delta^{+}_{i,l,j}$) of window j at layer l, `M-LocalSky` makes modifications to the *non-redundant* tuple set ($\mathcal{N}_{i,l,j}$) and the *dominance graph* ($\mathcal{T}_{i,l,j}$) of each layer l in stream $i \in \{1,2\}$ by applying a slightly different version of the techniques developed in [6]. In the original *StabSky* algorithm, when a new tuple, p_{new}, arrives in the data stream, p_{new} is added to the set of *non-redundant* tuples, $\mathcal{N}$, and the oldest element, p_{old}, in $\mathcal{N}$ is removed only if it has expired. The `M-LocalSky` module, on the other hand, makes additions and deletions of tuples to the *non-redundant* sets and *dominance graphs* based on $\Delta^{-}_{i,l,j}$ and $\Delta^{+}_{i,l,j}$.

At the end of each call, `M-LocalSky` returns the local skyline points ($\Lambda_{i,l,j}$) for each stream $i \in \{1,2\}$. The local skyline sets contain only the root nodes of the corresponding dominance graphs since we consider the sliding window model. The sliding window model is a special case of the n-of-m query model described in [6]; in sliding windows $n = m$.

C. Iteration Module (`M-Iterative`)

The proposed `M-Iterative` module is described in Figure 8. At each call of `M-Iterative`, the local skyline sets ($\Lambda_{1,l,j}$ and $\Lambda_{2,l,j}$) generated by the `M-LocalSky` module are processed to obtain the global skyline set ($\mathcal{G}_{l,j}$) of layer l in window j. In the modified version of the original *Iterative* algorithm [3], the `M-Iterative` module performs window-based joins using the symmetric hash join method [14].

Algorithm 1: M-LocalSky

Input:
$\Delta^{-}_{i,l,j}$: Expiring tuples of window j at layer l for stream $i \in \{1,2\}$
$\Delta^{+}_{i,l,j}$: New tuples of window j at layer l for stream $i \in \{1,2\}$
$\hat{A}$: Skyline attribute set of stream $i \in \{1,2\}$

Output:
$\Lambda_{i,l,j}$: Local skyline set of layer l for stream $i \in \{1,2\}$ in window j

Procedure: M-LocalSky($\Delta^{-}_{i,l,j}$, $\Delta^{+}_{i,l,j}$, $\hat{A}$)
for each expired tuple p_{old} in $\Delta^{-}_{i,l,j}$ **do**
$\mathcal{N}_{i,l,j} := \mathcal{N}_{i,l,j} - \{p_{old}\}$
remove interval $(a.start, p_{old}.start]$ with $p_{old}.start$ as the right end from $\mathcal{T}_{i,l,j}$
for each old critical edge $p_{old} \rightarrow p$ **do**
find the new critical edge $a \rightarrow p$
update interval $(p_{old}.start, p.start]$ in $\mathcal{T}_{i,l,j}$ to $(a.start, p.start]$ (or $(0, p.start)$])
end for
end for
for each new tuple p_{new} in $\Delta^{+}_{i,l,j}$ **do**
find $D_{p_{new}} \subseteq \mathcal{N}_{i,l,j}$ dominated by p_{new}
for each tuple $p \in D_{p_{new}}$ **do**
remove the intervals in $\mathcal{T}_{i,l,j}$ with $p.start$ as an end
end for
$\mathcal{N}_{i,l,j} := \mathcal{N}_{i,l,j} - D_{p_{new}} + \{p_{new}\}$
find the critical edge $p \rightarrow p_{new}$
add $(p.start, p_{new}.start]$ (or $(0, p_{new}.start)$]) to $\mathcal{T}_{i,l,j}$
end for
add the root nodes of $\mathcal{T}_{i,l,j}$ to $\Lambda_{i,l,j}$
return $\Lambda_{i,l,j}$

Fig. 7. The M-LocalSky module

Algorithm 2: M-Iterative

Input:
$W_{i,l,j}$: Data for stream $i \in \{1,2\}$ at layer l in window j
$\Lambda_{i,l,j}$: Local skyline set of layer l for stream $i \in \{1,2\}$ in window j
$\hat{A}$: Skyline attribute set of stream $i \in \{1,2\}$

Output:
$\mathcal{G}_{l,j}$: Global skyline set of layer l in window j

Procedure: M-Iterative($W_{i,l,j}$, $\Lambda_{i,l,j}$, $\hat{A}$)
$P_1 = \Lambda_{1,l,j} \overset{sw}{\bowtie} W_{2,l,j}$
$P_2 = W_{1,l,j} \overset{sw}{\bowtie} \Lambda_{2,l,j}$
$\mathcal{G}_{l,j} = \mathcal{G}_{l,j} + P_1 + P_2$
$\mathcal{G}_j = \mathcal{G}_j + \mathcal{G}_{l,j}$
Prune $W_{i,l,j}$ using *outsiderPrune* [3]
Push unpruned tuples in $W_{i,l,j}$ to iteration layer $W_{i,l+1,j}$
return $\mathcal{G}_{l,j}$

Fig. 8. The M-Iterative module

Once $\mathcal{G}_{l,j}$ is computed, M-Iterative prunes the input windows, $W_{1,l,j}$ and $W_{2,l,j}$, using *outsiderPrune*[5] [3]. Finally, M-Iterative completes execution by pushing the unpruned tuples to the next layer, $l+1$, as $W_{1,l+1,j}$ and $W_{2,l+1,j}$. The overall global skyline, $\mathcal{G}_j$, of the SWJ query over the current sliding window j is incrementally maintained by M-Iterative; $\mathcal{G}_j$ is obtained by the union of the global skyline sets produced at each layer l of window j.

[5]The *APDominatePrune* optimization in [3] adds significant run-time execution cost and hence is not suitable for streaming scenarios; thus the methods described in this paper do not use this optimization.

D. Truncated Layered Skyline-Window-Join

Let us consider Figure 5 which shows a sample execution of a SWJ query for 5 consecutive windows, with 10% new tuples arriving and 10% tuples expiring per window. It is easy to see from the figure that the per-layer overlaps tend to drop as the iteration count increases: there are often larger number of overlaps in the earlier iterations, whereas the number of overlaps gets almost monotonically lower as the iterations progress. Since the benefits of the *iteration-fabric* depend on the degree of per-layer overlap, it may be advantageous to stop checking for overlaps against the previous window once the degree of overlap drops below a preset threshold at an iteration level or when the gains achieved through overlap analysis falls below the time needed to maintain the data structures. We refer to this as *truncated* LSJ processing.

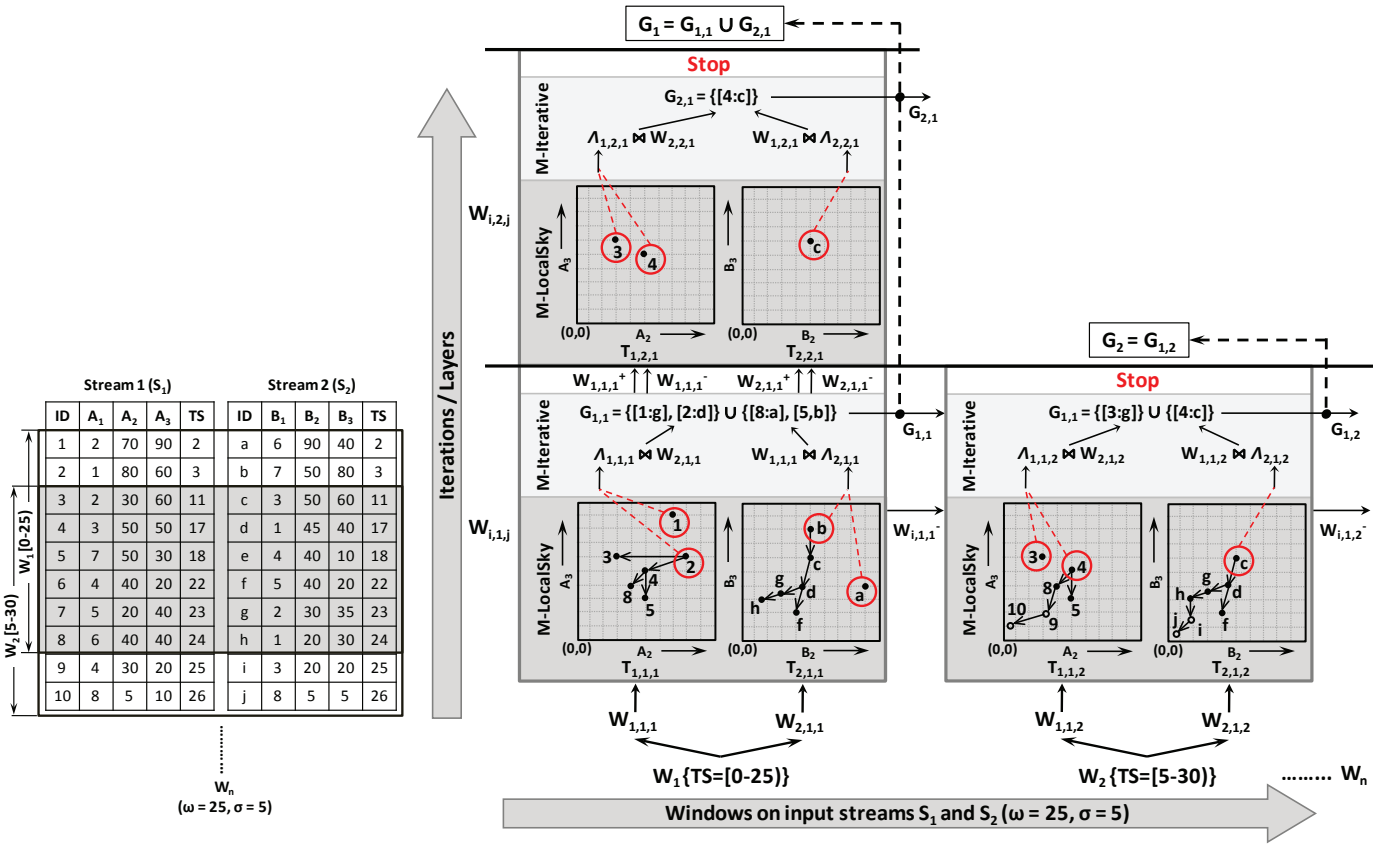
E. Example Execution of Layered Skyline-Window-Joins

Figure 9 illustrates the proposed LSJ operator. The incoming streams are bound by sliding windows (Figure 9(a)) and skyline-window-joins are executed over these sliding windows using the novel *iteration-fabric* framework (Figure 9(b)).

In the given example, S_1 (containing tuples with IDs $1, 2, \ldots$) and S_2 (containing tuples with IDs $a, b, \ldots$) are the input streams, $\hat{\mathcal{A}} = \{A_2, A_3, B_2, B_3\}$ is the set of skyline attributes, TS is the timestamp attribute, and an equi-join is carried out on the join attribute set $\mathcal{A}_\Theta = \{A_1, B_1\}$. The input streams are bound by a sliding window of size $\omega = 25$. Each window W_j spans a time period of $[W_j.start, W_j.end)$, where $W_j.end = W_j.start + \omega$; for instance, the time period of $W_1 = [0, 25)$. The sliding window is constrained by $\sigma = 5$, i.e. the start of the window shifts 5 units at every window update; for example, $W_2.start = W_1.start + \sigma = 5$.

LSJ executes SWJ queries by pushing the tuples contained in a sliding window through the M-LocalSky and M-Iterative modules. As shown in Figure 9(b), LSJ starts with window W_1 and sends the tuples of each of the streams in layer 1 ($W_{1,1,1}$, $W_{2,1,1}$) to the M-LocalSky module in order to obtain the local skyline sets $\Lambda_{1,1,1}$ and $\Lambda_{2,1,1}$. Since this is the very first window being executed, M-LocalSky does not have to look at the layers of a previous window, and therefore, no overlap analysis is performed at this stage. M-LocalSky builds the dominance graphs $\mathcal{T}_{1,1,1}$ and $\mathcal{T}_{2,1,1}$ on the non-redundant tuple sets $\mathcal{N}_{1,1,1} = (1,2,3,4,5,8)$ and $\mathcal{N}_{2,1,1} = (a,b,c,d,f,g,h)$, respectively. The roots of $\mathcal{T}_{1,1,1}$ and $\mathcal{T}_{2,1,1}$ are the local skyline points of layer 1 in W_1, i.e. $\Lambda_{1,1,1} = (1,2)$ and $\Lambda_{2,1,1} = (a,b)$. These local skyline sets are then pushed to the M-Iterative module.

M-Iterative generates the global skyline of layer 1 in W_1 ($\mathcal{G}_{1,1}$) using the local skyline sets obtained from



(a) Input streams bound by sliding windows

(b) Execution of skyline-window-joins using the *iteration-fabric* framework

Fig. 9. An example Layered Skyline-window-Join (LSJ) operation

M-LocalSky. $\mathcal{G}_{1,1}$ is then used for pruning $W_{1,1,1}$ and $W_{2,1,1}$ to obtain the tuples that qualify to be in layer 2 of W_1 ($W_{1,2,1}$, $W_{2,2,1}$). As seen in Figure 9(b), the dominance graphs $\mathcal{T}_{1,2,1}$ and $\mathcal{T}_{2,2,1}$ of layer 2 in W_1 reflect the tuples pruned. LSJ repeats its calls to M-LocalSky and M-Iterative as before; it eventually comes to a stop when all tuples in the current window are pruned and no new layers exist. The global skyline set of W_1 is given by the global skyline sets obtained at every layer, i.e. $\mathcal{G}_1 = \mathcal{G}_{1,1} + \mathcal{G}_{2,1}$.

After analysing W_1, LSJ continues execution on window W_2 (obtained based on the parameters $\omega = 25$ and $\sigma = 5$). Once again, LSJ sends the tuples in W_2 to the M-LocalSky and M-Iterative modules to obtain the global skyline of this window. The key difference here is that the M-LocalSky module now analysis the overlap between the layers of the current and previous windows. Figure 9(a) shows that there is a significant overlap between windows W_1 and W_2. Therefore, when the tuples of each of the streams in a particular layer of W_2 are pushed to the M-LocalSky module, it makes insertions and deletions to the dominance graphs only based on the set of expiring and new tuples – tuples that overlap are not reprocessed. For instance, in Figure 9(b), the overlap set between layers $W_{1,1,1}$ and $W_{1,1,2}$ contains tuples with IDs $3, 4, 5, 8$ (indicated by the solid dots in dominance graph $\mathcal{T}_{1,1,2}$), so the set of new tuples contains $9, 10$ and the set of expiring tuples has $1, 2$. Thus, based on these sets of expiring and new tuples, the dominance graph of Stream 1 in layer 1 of W_2 ($\mathcal{T}_{1,1,2}$) is obtained by making insertions and deletions to the previous dominance graph of Stream 1 in layer 1 of W_1 ($\mathcal{T}_{1,1,1}$). If a previous layer doesn't exist, then the M-LocalSky module processes the current layer similar to how it processes layers in window W_1 (described earlier).

Note that, in this simplified example, sharing across windows is leveraged only at the first layer. However, in practice there exists opportunities for sharing at more than one layer (see Figure 5), and as evaluated in the next section, LSJ leverages these overlaps for improved performance.

VI. Performance Evaluation

In this section, we evaluate the performance of LSJ on real and simulated data sets by varying the parameters involved.

A. Experimental Setup

This is the first paper that we are aware of which targets skyline-join queries on data streams. Therefore, for comparison purposes, we have implemented alternative schemes, with varying degrees of *intelligence*:

- A Naive method which first performs a window-based join of the input streams using the symmetric hash join method [14]. The join results are then processed using SFS [25] to obtain the final skyline set of each window.

TABLE II
INTEL BERKELEY RESEARCH LAB DATA SET

date:yyyy-mm-dd	time:hh:mm:ss.xxx	epoch:int	moteid:int
temperature:real	humidity:real	light:real	voltage:real

- An *iterative-skyline-join* (ISJ) scheme that operates by applying SFS at each iteration (Section IV-B).
- In addition, we have varied the *truncation layer*, l, of the *iteration-fabric*. Here, LSJ ($l = *$) means that the first $*$ layers of a particular sliding window are processed using the *iteration-fabric* processing structure (Section V-A), while the remaining layers in the window are executed using the *iterative-skyline-join* (ISJ) scheme (Section IV-B). LSJ ($l = 1$) corresponds to the case where the overlaps only at the data-entry level are leveraged using *StabSky* [6], whereas LSJ and LSJ (all) correspond to the case where overlaps are identified and leveraged at all levels of the *iteration-fabric*.

• **Evaluation Platform.** The above algorithms were all implemented in Java. The Interval tree used for maintaining dominance graphs in M-LocalSky was adapted from www.gephi.org[6]. All experiments were conducted on a machine with an Intel Core i5 3.10GHz processor, 8GB RAM (1GB of which is available to the Java machine[7]) and Windows 7 operating system. Each experiment is run three times and the execution times reported are the averages of the three runs.

• **Datasets.** The evaluations were carried out on both synthetic and real data. Synthetic streams were generated based on correlated, independent and anti-correlated distributions[8] as described in [1]. Since the tuples generated by [1] have no timestamps, we borrowed the *epoch* values from the *Intel Berkeley Research lab* data set[9] and used them as timestamps.

We also ran experiments on the above-mentioned *Intel Berkeley Research (IBR) lab* sensor data stream. This data contains readings collected from 54 sensors and has about 2.3 million readings. The complete schema of the data set is shown in Table II. In this, *moteid* is a unique integer assigned to each sensor and *epoch* corresponds to a monotonically increasing sequence number obtained from the sensors every 30 seconds. The data set also gives the x and y coordinates of the sensors We use this information to calculate each sensor's distance from the point $(0, 0)$ and utilize this as an additional attribute.

• **Evaluation Parameters.** As is common in assessing skyline algorithms, we use execution time as the major evaluation metric. Execution time of a SWJ query is the duration from the time an algorithm starts to the time it returns the entire skyline set. It also includes the time taken to maintain the various index structures that are built on-the-fly. The streams were analyzed over 30 sliding windows and the execution time gains relative to competitor methods are reported. Time gain (%) is calculated as $(1 - a/b) \times 100$, where a, b represent the total time taken to execute skyline-joins over 30 windows by the two algorithms being compared (i.e. a vs. b).

The analysis was carried out over windows of different sizes (ω) and window shift lengths (σ). We also analyzed the effects of the join rate (r) between the streams and the dimensionality (d) of the skyline attribute set per input stream.

[6]Source code available at https://github.com/gephi/gephi-launchpad-branches/tree/master/AttributesAPI/src/org/gephi/data/attributes/type.

[7]In these experiments, the memory was not a bottleneck.

[8]http://randdataset.projects.postgresql.org/.

[9]http://db.csail.mit.edu/labdata/labdata.html.

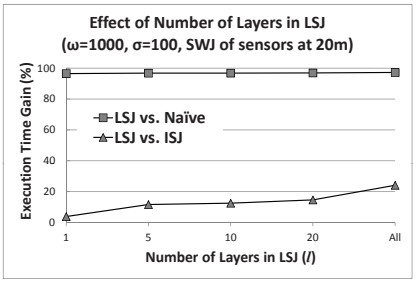


Fig. 10. Effect of number of layers (l) in LSJ (Since ISJ does not consider layer overlaps, l has no impact on its execution time)

B. Evaluation over Real Streams

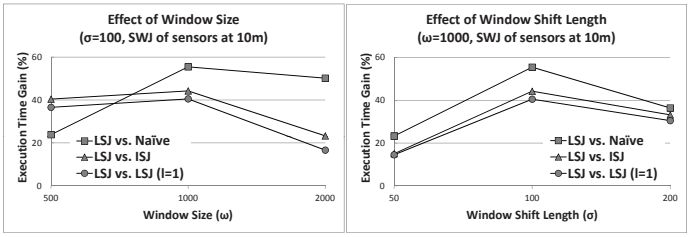
This section presents evaluation results of the proposed approaches on real data. We consider a scenario where the even-numbered *IBR lab* sensors produce readings only related to *temperature* and *voltage*, while the odd-numbered sensors give readings of *humidity* and *light*. This results in two input streams, *intel-even (moteid, temperature, voltage, epoch, distance)* and *intel-odd (moteid, humidity, light, epoch, distance)*. Given these, we search for the skyline-join over the set of sensors that are distance, δ, from the origin $(0, 0)$ of the room for the attributes *temperature, voltage, humidity, light*. Informally, this query returns a set of *interesting* readings produced by sensors that belong to a particular region of the room. The sliding window is defined using the *epoch* attribute.

• **Impact of the Number of Layers (l) on LSJ.** Figure 10 illustrates the behavior of LSJ as the number of layers (l) to which the *iteration-fabric* is applied changes. Here, the SWJ operation is carried out among sensors that are at a distance of 20 meters from the corner of the room.

The figure shows that the execution time gain achieved by LSJ increases as l increases, with the case where *iteration-fabric* is applied to all layers (LSJ ($l = all$)) performing the best overall. Figure 10 shows that, for the given configuration, LSJ has significant ($> 95\%$) gains over Naive. More importantly, however, the performance of LSJ increases with the number layers included in the *iteration-fabric* and it provides $\sim 25\%$ gain over ISJ, when all the layers are included.

As we see in the rest of the experiments, this gain is a function of the various parameters, including amount of data and the data distribution and in our experiments with this real data set, the gains over ISJ varied between $\sim 25\%$ and $\sim 40\%$. Gains up to $\sim 80\%$ are observed in correlated data streams as discussed in Section VI-C.

• **Effect of the Window Size.** The experiment reported in Figure 11(a) studies the effect of the size of the sliding



(a) Effect of window size (ω) (b) Effect of window shift length (σ)

Fig. 11. Evaluation over real streams

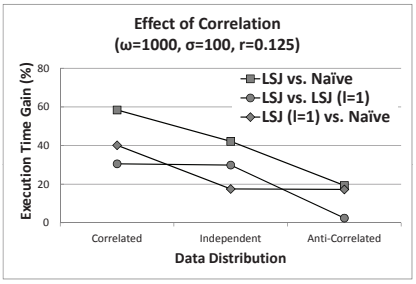


Fig. 12. Effect of data correlation

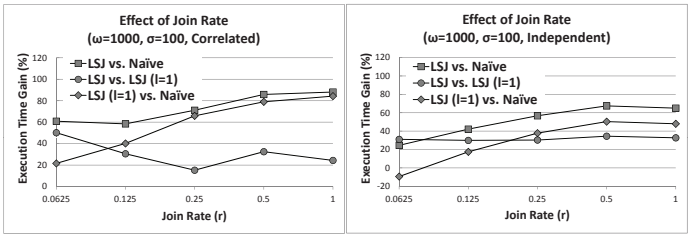
windows (ω) – for a given window shift. In this setup, the SWJ operation is carried out among sensors that are at a distance of 10 meters from the corner of the room. Figure 11(a) shows that LSJ performs better than Naive, ISJ, and LSJ ($l = 1$) for varying window sizes. The performance of LSJ is especially high ($\sim 45\%$ gain over ISJ) when the window size is $10\times$ the shift length. The time gains relative to ISJ and LSJ ($l = 1$) drop when the window size is much larger. This means that, for this data, the overlaps at higher iteration layers are not large enough to compensate for the overhead of maintaining the necessary data structures. Thus, these data streams would benefit from a truncated execution of LSJ (described in Section V-D).

• **Effect of Window Shift Length.** Figure 11(b) examines the execution time gains achieved by LSJ as the window shift length (σ) is changed – for a fixed window size. A larger window shift length implies a bigger change in the layers of the *iteration-fabric*, with lesser overlaps being present between the layers of consecutive windows. As shown in Figure 11(b), LSJ successfully leverages any overlaps that may exist and executes SWJ queries faster, more efficiently than the Naive, ISJ, and LSJ ($l = 1$) methods and provides high ($\sim 45\%$) gains over ISJ when the window size is $10\times$ the shift length. Note that there is a drop in performance gain when the shift length is very small ($\sigma = 50$). As we later experimentally show in Section VI-C, this occurs when the skyline attributes are relatively independent. Once again, in such a scenario a truncated execution of LSJ would be beneficial.

C. Evaluation over Synthetic Streams

In this section, we carry out a more detailed analysis of LSJ on synthetically generated data streams, where we also vary the join selectivity and consider correlated, independent and anti-correlated distributions. The default dimensionality (d) of the skyline attribute set per data stream is set to 2, which gives a total of 4 skyline attributes for SWJ operations.

• **Effect of Correlation.** Figure 12 illustrates that LSJ provides the largest gains on correlated data distributions and provides the lowest gains in anti-correlated distributions. In fact, LSJ's (i.e., LSJ (all)'s) performance gains against LSJ ($l = 1$) is close to 0%. This implies that in anti-correlated data sets, the process does not generate sufficiently large overlaps in high-numbered iteration layers – most of the overlaps are identified and eliminated at the data layer itself; therefore, LSJ ($l = 1$) itself is sufficient. In correlated and independent data sets, however, eliminating overlaps at the higher numbered iteration layers provide $\sim 30\%$ execution time gains over LSJ ($l = 1$).



(a) Correlated (b) Independent

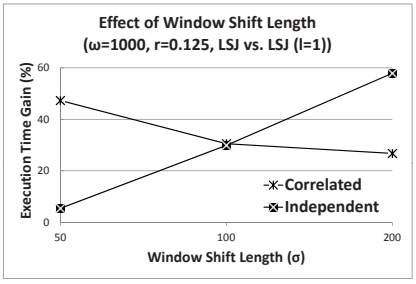
Fig. 13. Effect of join rate (r)

Note that both LSJ and LSJ ($l = 1$) perform better for correlated data than for anti-correlated data streams. This is expected as the *Iterative* technique that forms the basis of LSJ family of algorithms is known to perform less efficiently on anti-correlated data distributions [3].

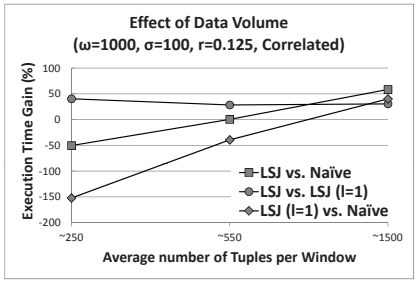
• **Effect of Join Rate.** Figure 13 shows the effect of the join rate (r) between pairs of sliding windows. We can observe that as the join work increases with the join rate, LSJ's embedding of the join computation into the skyline process provides larger gains during SWJ operations.

However, it is interesting that, as can be seen in Figure 13(a), on correlated data sets, LSJ's gains over LSJ ($l = 1$) drops as the join rate increases – in contrast, on independent data sets (Figure 13(b)), the gain relative to LSJ ($l = 1$) stays constant. This implies that on correlated data sets most of the overlaps are identified and removed early in the iteration process, whereas in independent data sets, this is not the case.

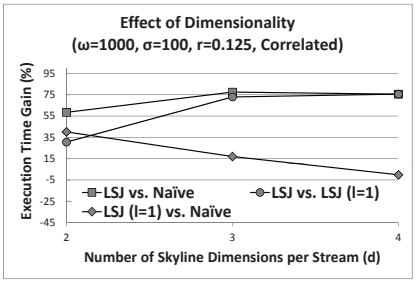
• **Effect of Window Shift Length.** Figure 14(a) examines the time gains achieved by LSJ as the window shift length (σ) is changed. As mentioned before, a larger window shift length implies a bigger change in the layers of the *iteration-fabric*, with lesser overlaps being present between the layers of consecutive windows. As seen in Figure 14(a), the performance also depends on data distribution. For correlated data, as expected, higher overlaps (smaller shift lengths) provide larger gains. For independent data, however, the behavior is the opposite. This implies that in skyline-window-joins with independent skyline attributes most of the overlaps are eliminated in the small-numbered (i.e. earlier) iteration layers. This highlights that a truncated execution of LSJ would be useful in cases where the data streams are not highly correlated.



(a) Effect of window shift length (σ)



(b) Effect of data volume



(c) Effect of dimensionality (d) of skyline attribute set per input stream

Fig. 14. Evaluation over synthetic streams

• **Effect of the Data Volume/Window Size.** This experiment (Figure 14(b)) examines the effect of the changes in the volume of data contained in each sliding window. As can be observed, when the the data volume is low, the overhead of processing data through the *iteration-fabric* is not worthwhile. However, as the volume of the data increases, the benefits of the LSJ approach becomes more and more apparent.

• **Effect of Skyline Dimensionality.** Figure 14(c) shows that LSJ scales particularly well to SWJ operations on high dimensional skyline queries. As can be observed, the gains of LSJ increases both against Naive and LSJ ($l = 1$) as the number of skyline dimensions increases. Hence, we can conclude that as the number of skyline dimensions increases, less overlaps are removed at earlier iterations and thus applying the *iteration-fabric* to all layers of SWJ brings better performance.

VII. Conclusions

In this paper, we introduced and studied the problem of computing skyline-window-join (SWJ) queries over pairs of data streams. Recognizing that overlaps exist not only at the consecutive windows of the input data steams, but also consecutive windows of the individual iteration layers of skyline-computation, we first proposed a novel *iteration-fabric* processing structure to identify and eliminate per-layer overlaps across consecutive windows and then presented a Layered Skyline-window-Join (LSJ) operator that (a) partitions the overall process into processing layers and (b) maintains skyline-join results in an incremental manner by continuously monitoring the changes in all layers of the process. Extensive experimental evaluations over real and simulated data sets showed that the proposed approach provides significant gains over processing schemes that are not designed to eliminate redundant work across multiple processing layers.

References

[1] S. Börzsönyi, D. Kossmann, and K. Stocker, "The skyline operator," in *ICDE*, 2001, pp. 421–430.

[2] R. E. Steuer, *Multiple Criteria Optimization: Theory, Computation and Application*. John Wiley, New York, 546 pp, 1986.

[3] D. Sun, S. Wu, J. Li, and A. K. H. Tung, "Skyline-join in distributed databases," in *ICDE Workshops*, 2008.

[4] S. Sun, Z. Huang, H. Zhong, D. Dai, H. Liu, and J. Li, "Efficient monitoring of skyline queries over distributed data streams," *Knowl. Inf. Syst.*, vol. 25, pp. 575–606, 2010.

[5] A. Das Sarma, A. Lall, D. Nanongkai, and J. Xu, "Randomized multi-pass streaming skyline algorithms," *VLDB*, vol. 2, pp. 85–96, 2009.

[6] X. Lin, Y. Yuan, W. Wang, and H. Lu, "Stabbing the sky: Efficient skyline computation over sliding windows," in *ICDE*, 2005.

[7] Y. Tao and D. Papadias, "Maintaining sliding window skylines on data streams," *IEEE TKDE*, vol. 18, pp. 377–391, 2006.

[8] W. Zhang, X. Lin, Y. Zhang, W. Wang, and J. X. Yu, "Probabilistic skyline operator over sliding windows," in *ICDE*, 2009, pp. 1060–1071.

[9] B. Catania, G. Guerrini, M. T. Pinto, and P. Podesta, "Relaxed queries over data streams," in *SEBD*, 2012.

[10] M. Nagendra and K. S. Candan, "Skyline-sensitive joins with LR-pruning," in *EDBT*, 2012, pp. 252–263.

[11] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *PODS*, 2002, pp. 1–16.

[12] C. Cranor, Y. Gao, T. Johnson, V. Shkapenyuk, and O. Spatscheck, "Gigascope: High performance network monitoring with an sql interface," in *ACM SIGMOD*, 2002.

[13] S. Madden and M. Franklin, "Fjording the stream: An architecture for queries over streaming sensor data," in *ICDE*, 2002.

[14] A. N. Wilschut and P. M. G. Apers, "Dataflow query execution in a parallel main-memory environment," in *PDIS*, 1991, pp. 68–77.

[15] S. D. Viglas, J. F. Naughton, and J. Burger, "Maximizing the output rate of multi-way join queries over streaming information sources," in *VLDB*, 2003, pp. 285–296.

[16] H.-G. Li, S. Chen, J. Tatemura, D. Agrawal, K. S. Candan, and W.-P. Hsiung, "Safety guarantee of continuous join queries over punctuated data streams," in *VLDB*, 2006, pp. 19–30.

[17] U. Srivastava and J. Widom, "Memory-limited execution of windowed stream joins," in *VLDB*, 2004, pp. 324–335.

[18] N. Tatbul and S. Zdonik, "Window-aware load shedding for aggregation queries over data streams," in *VLDB*, 2006, pp. 799–810.

[19] S. Wang, E. A. Rundensteiner, S. Ganguly, and S. Bhatnagar, "State-slice: New paradigm of multi-query optimization of window-based stream queries," in *VLDB*, 2006, pp. 619–630.

[20] R. V. Nehme and E. A. Rundensteiner, "Scuba: Scalable cluster-based algorithm for evaluating continuous spatio-temporal queries on moving objects," in *EDBT*, 2006, pp. 1001–1019.

[21] K. Mouratidis, S. Bakiras, and D. Papadias, "Continuous monitoring of top-k queries over sliding windows," in *SIGMOD*, 2006, pp. 635–646.

[22] A. Shastri, D. Yang, E. A. Rundensteiner, and M. O. Ward, "Mtops: Scalable processing of continuous top-k multi-query workloads," in *CIKM*, 2011, pp. 1107–1116.

[23] L. Peng, R. Yu, K. S. Candan, and X. Wang, "Object and combination shedding schemes for adaptive media workflow execution," *IEEE TKDE*, vol. 22, no. 1, pp. 105–119, 2010.

[24] X. Wang, K. S. Candan, and J. Song, "Complex pattern ranking (CPR): Evaluating top-k pattern queries over event streams," in *DEBS*, 2011.

[25] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, "Skyline with presorting," in *ICDE*, 2003.

[26] I. Bartolini, P. Ciaccia, and M. Patella, "Salsa: computing the skyline without scanning the whole sky," in *CIKM*, 2006.

[27] A. Vlachou, C. Doulkeridis, and N. Polyzotis, "Skyline query processing over joins," in *SIGMOD*, 2011.

[28] B. Jiang and J. Pei, "Online interval skyline queries on time series," in *ICDE*, 2009, pp. 1036–1047.

[29] N. H. Park, V. Raghavan, and E. A. Rundensteiner, "Supporting multi-criteria decision support queries over time-interval data streams," in *DEXA: Part I*, 2010, pp. 281–289.