Real-Time Skyline Computation on Data Streams with SLS: Implementation and Experiences

L. Rudenko, M. Endres

Report 2018-01
July 31, 2018
Real-Time Skyline Computation on Data Streams with SLS: Implementation and Experiences

Lena Rudenko, Markus Endres

Institut for Computer Science, University of Augsburg, D-86135 Augsburg, Germany
{lена.rudenko,markus.endres}@informatik.uni-augsburg.de

Abstract. Skyline processing has received considerable attention in the last decade, in particular when filtering the most preferred objects from a multi-dimensional set on contradictory criteria. Most of the work on Skyline computation focus on conventional databases, but stream data analysis becomes a high relevant topic in various academic and business fields. Nowadays, an enormous number of applications require the analysis of time evolving data and therefore the study of continuous query processing has recently attracted the interest of researchers all over the world. In this paper, we propose a novel algorithm called SLS for evaluating Skyline queries over continuous settings, and empirically demonstrate the advantage of this algorithm on artificial and real data. Our algorithm continuously monitors the incoming data and therefore is able to maintain the Skyline incrementally. For this, SLS utilizes the lattice structure a Skyline query constructs and analyzes the Skyline in linear time.

Keywords Streams, Skyline, Preferences, Realtime

1 Introduction

Today, data processed by humans as well as computers is very large, rapidly increasing and often in form of data streams. Users want to analyze this data to extract personalized and customized information in order to learn from this ever-growing amount of data, cp. e.g., [123]. Many modern applications such as network monitoring, financial analysis, infrastructure manufacturing, sensor networks, meteorological observations, or social networks require query processing over data streams, cp. for example [456]. Therefore stream data processing is a highly relevant topic today.

A stream is a continuous unbounded flow of data objects made available over time. Due to the continuous and potentially unlimited character of stream data, it needs to be processed sequentially and incrementally. However, queries on streams run continuously over a period of time and return different results as new data arrive. That means, looking on stream data twice is not possible. Hence, analyzing streams can be considered as a difficult and complex task which is in
the focus of current research. Many scientists all over the world try to process and to analyze streams to extract important information from such continuous data flows. Babu and Widom [7], e.g., focus on the problem, how to define and evaluate continuous queries over data streams.

On the other hand, preference queries [8,10,11,3] have received considerable attention in the past, due to their use in selecting the most preferred items, especially when the filtering criteria are contradictory. In particular, Skyline queries [12,13,14], a subset of preference queries, have been thoroughly studied by the database community to filter high relevant information from a dataset. A Skyline query, also known as Pareto preference query, selects those objects from a dataset $D$ that are not dominated by any others. An object $p$ having $d$ attributes (dimensions) dominates an object $q$, if $p$ is better than $q$ in at least one dimension and not worse than $q$ in all other dimensions, for a defined comparison function. This dominance criterion defines a partial order and therefore transitivity holds. The Skyline is the set of points which are not dominated by any other point of $D$. Without loss of generality, the Skyline with the $\min$ function for all attributes is used in this paper.

Example 1. Figure 1 presents the Skyline of tweets – short messages from Twitter. Each tweet is represented as a point in the two-dimensional space of activity status of a user and the hashtag. The first dimension (activity status) is represented by the status values {active, non-active, unknown} which are mapped to the scores 0, 1, and 2. The second dimension (hashtag) is an element of 

{$\#$pyeongchang2018, $\#$olympics, $\#$olympia2018, $\#$teamgermany, $\#$others} and is represented with the values 0, ..., 4.

Now, the objective is to find all tweets, which are Pareto optimal w.r.t. the activity status and the hashtag, since we assume that very active users using the hashtag $\#$pyeongchang2018 post the most interesting information on the Olympic winter games in Pyeongchang. Of course, there are also less active users using the same hashtag, or users using, e.g., $\#$olympia2018 for their tweets.

Of interest are all tweets that are not worse than any other tweet in both dimensions w.r.t. to our search preference. Tweet $t_4$ is dominated by the tweets $t_1$ and $t_2$, $t_5$ by $t_1$, $t_2$ and $t_3$. The tweets $t_1$, $t_2$ and $t_3$, on the other hand, are not dominated by any other tweets and build the Skyline.

Algorithms proposed for traditional database Skyline computation, e.g., [15,12,10,14], are not appropriate for continuous data and therefore new techniques should be developed to fulfill the requirements posed by the data stream model. The most important property of data streams is that new objects are continuously appended, and therefore, efficient storage and processing techniques are required to cope with high update rates. Hence, a stream-oriented algorithm should satisfy the following requirements (cp. [2]): 1) fast response time, 2) incremental evaluation, 3) limited number of data access, and 4) in memory storage to avoid expensive disk accesses.

1 Twitter: https://twitter.com/
This paper is an extended version of [17,18] and presents implementation details and comprehensive experiments on the SLS algorithm for real-time Skyline processing on data streams. The proposed algorithm is based on the lattice structure representing the better-than relationships that must be built only once for efficient Skyline computation on continuous data. Our algorithm satisfies all four requirements on stream-oriented algorithms as mentioned before.

The remainder of this paper is organized as follows: Section 2 recapitulates essential concepts of Skyline queries. In Section 3 we introduce the Stream Lattice Skyline algorithm. Our results on comprehensive experiments are shown in Section 4. Section 5 contains a summary and outlook.

2 Background

2.1 Skyline Queries

The aim of a Skyline query or Pareto preference [3] is to find the best objects in a dataset, denoted by Sky(D). More formally:

**Definition 1 (Dominance and Indifference.)** Assume a set of vectors $D \subseteq \mathbb{R}^d$. Given $p = (p_1, ..., p_n), q = (q_1, ..., q_d) \in D$, $p$ dominates $q$ on $D$, denotes as $p \prec q$, if the following holds:

$$p \prec q \iff \forall i \in \{1, ..., d\} : p_i \leq q_i \land \exists j \in \{1, ..., d\} : p_j < q_j$$

(1)

Two objects $p$ and $q$ are called indifferent on $D$, denoted as $p \sim q$, if and only if $p \not\prec q$ and $q \not\prec p$.

Note that following Definition 1 we consider subset of $\mathbb{R}^d$ in that we search for Skylines w.r.t. the natural order $\leq$ in each dimension.

**Definition 2 (Skyline Sky(D).)** The Skyline Sky(D) of $D$ is defined by the maxima in $D$ according to the ordering $\prec$, or explicitly by the set

$$Sky(D) := \{p \in D \mid \nexists q \in D : q \prec p\}$$

(2)
In this sense, the minimal values in each domain are preferred and we write \( p \prec q \) if \( p \) is better than \( q \).

Skylines are not restricted to numerical domains. For any universe \( \Omega \) and orderings \( \prec_i \in (\Omega \times \Omega)(i \in \{1, ..., d\}) \) the Skyline w.r.t. \( \prec_i \) can be computed, if there exist scoring functions \( g_i : \Omega \rightarrow \mathbb{R} \) for all \( i \in \{1, ..., d\} \) such that \( p \prec_i q \Leftrightarrow g_i(p) < g_i(q) \). Then the Skyline of a set \( M \subseteq \Omega \) w.r.t. \( (\prec_i)_{i=1,\ldots,d} \) is equivalent to the Skyline of \( \{(g(p_1), \ldots, g(p_d)) | p_i \in M\} \). That means, categorical domains like activity status or hashtag as in Example 1 can easily be mapped to a numerical domain.

Note that there is also the concept of top-k Skyline queries where the aim is to find the \( k \) best objects [19][20].

2.2 Skyline Queries on Data Streams

Skyline processing on data streams require modified algorithms, since a stream is a continuous dataflow and there is no “final” result after some data of the stream is processed. The result must be calculated and adjusted as soon as new data arrive, since new stream objects received later can build a new Skyline compared with objects already recognized in previously computed (temporary) Skylines. To the best of our knowledge, only Block-Nested-Loop (BNL) style algorithms (cp. [12][21][22][23] or [24]) can be adapted to Skyline evaluation on continuous data. These algorithms follow an object-to-object comparison approach, an expensive operation with a worst-case runtime of \( O(n^2) \), where \( n \) is the number of objects.

For analyzing a continuous, unbounded data stream, it is necessary to divide it into a series of (non-overlapping) chunks \( c_1, c_2, \ldots \). A BNL-style algorithm would evaluate the Skyline on the first chunk, i.e., \( Sky(c_1) \), cp. Definition 2. Since \( c_2 \) could contain better objects w.r.t. the dominance criterion in Definition 1, one also has to compare the new objects from \( c_2 \) to the current Skyline, i.e., compute

\[
Sky(Sky(c_1) \cup c_2),
\]

and so on. However, this leads to a computational overhead if \( c_2 \) is large.

3 The Stream Lattice Skyline Algorithm

Our Stream Lattice Skyline (SLS) algorithm was developed for efficient real-time Skyline computation on unbounded streams. It does not depend on object comparisons as BNL algorithms do, but on the lattice structure constructed by a Skyline query over low-cardinality domains (either inherently small, such as activity status of a user – or mapped to low-cardinality domains, such as number of followers in Twitter).
3.1 The Idea of SLS

Before we describe our SLS algorithm for real-time stream processing, we revisit the basics of Hexagon [25] and Lattice Skyline [26] our algorithm is based on.

A Skyline query over discrete domains constructs a lattice, a structure where two objects \( o_1 \) and \( o_2 \) of a dataset have a least upper bound and a greatest lower bound. Visualization of such lattices is often done using Better-Than-Graphs (BTG) (similar to Hasse diagrams). An example of a BTG is shown in Figure 2.

The nodes in the BTG represent equivalence classes. The idea is to map objects from the stream to these equivalence classes using some kind of feature function. All values in the same class are considered substitutable.

![Figure 2: Stream processing with SLS.](image)

We write \([2,4]\) to describe a two-dimensional domain as well as the maximal possible values of the feature vector representing objects. For example, the BTG in Figure 2 could present a Skyline on the tweets activity status of a user (\{activ, non-active, unknown\}) (values 0, 1, and 2) and the hashtag which might be an element of \{#pyeongchang2018, #olympics, #olympia2018, #teamdeutschland, #others\} (values 0, . . . , 4). The arrows in the BTG show dominance relationships between nodes. The node \((0, 0)\) presents the best node, whereas \((2, 4)\) is the worst node. The bold numbers next to each node are unique identifiers (ID) for each node in the BTG. Nodes having the same level are indifferent. That means for example, that neither the objects in the node \((0, 4)\) are better than the objects in \((2, 2)\) nor vice versa. They have the same overall level 4. A dataset does not necessarily contain representatives for each BTG node. All gray nodes are occupied with an element of the data and therefore non-empty, white nodes are empty.

The elements of the data that compose the Skyline are those in the BTG that have no path leading to them from another non-empty node. In Figure 2 these are...
the nodes (0,1) and (2,0). All other nodes have direct or transitive edges from these both nodes, and therefore are dominated.

Our approach in general works as follows: After constructing the BTG, which only must be done once, all objects of a chunk are mapped to the corresponding nodes in a consecutive way, e.g., object A is mapped to (2, 0), object B to (2, 2), and so on. Assume all gray nodes in Figure 2 are occupied with data from the first chunk. Afterwards, a breadth-first traversal (BFT) runs to find the non-empty nodes (blue dashed line in Figure 2). For the first non-empty node (here (0, 1)) we start a depth-first traversal (DFT) (red arrows) to mark all transitive dominated nodes as dominated. If the DFT reaches the bottom node (2, 4) (or an already dominated node) it will recursively follow all other edges. Thereafter, the BFT continues with node (1, 0), which is empty. The next non-empty node is (1, 1), but dominated. Continue with (2, 0). Since all other nodes are marked as dominated, the remaining nodes, (0, 1) and (2, 0), present the Skyline.

Note that lattice-based algorithms are developed for Skyline computation over low-cardinality domains. An attribute domain \( \text{dom} (S) \) is said to be low-cardinality if its value is drawn from a set \( S = s_1, ..., s_m \), such that the set cardinality \( m \) is small. For a low-cardinality domain and \( s_i \in \mathbb{R} \), a one-to-one mapping function \( f : \text{dom} (\mathbb{R}) \rightarrow \mathbb{N}_0, f(s_i) = i - 1, \) can be defined to get discrete values as required in our algorithm.

3.2 The SLS Algorithm

In this section we describe our SLS algorithm in detail. Thereby, we follow the idea depicted in Section 3.1. SLS is based on a series of finite chunks as described in Section 2.2. We divide SLS into three phases:

1. The Construction Phase (see pseudo code Phase 1) initializes the BTG which depends on the Skyline query (see [23, 20] for details). This has to be done only once for the first chunk (line 1). For evaluation of the following chunks, the existing BTG will be reused.

   *The BTG is represented by an array (line 2) of Nodes in main memory. A Node is a data structure representing an equivalence class in the BTG, which may contain objects from the stream. Nodes are identified by their IDs (cp. [23]), which correspond to their position in the BTG array. A Node also contains its status empty (initial status), non-empty, or dominated.*

   **Phase 1 Construction Phase**
   1: if algorithmFirstRun then
   2: \( \text{BTG} := \text{array of empty Nodes (equivalence classes)} \)
   3: end if

2. In the Adding Phase (see pseudo code Phase 2) we process the input data:
   a. Read the next chunk \( c_i \) from the data stream \( S \).
b. Iterate through the objects $o_j$ of chunk $c_i$ (line 3). Each object will be mapped to one node in the BTG. For this we compute the ID of the current object $o_j$ (line 4) and store it in the BTG array, if the node is not dominated (line 5 and 6).

### Phase 2 Adding Phase

1. **Input:** Next chunk $c_i$ from data stream $S$
2. $\triangleright$ Add objects $o_j$ from chunk $c_i$ to the BTG
3. for $o_j$ in $c_i$ do
4. ID := compute ID for $o_j$
5. if $\neg$BTG[ID].isDominated() then $\triangleright$ Node is not dominated
6. BTG[ID].add($o_j$)
7. BTG[ID].setStatus = non-empty
8. end if
9. end for

3. **Removal Phase** (see pseudo code Phase 3): After all objects in the chunk have been processed, the nodes of the BTG that are marked as non-empty and are not reachable by the transitive dominance relationship from any other non-empty node of the BTG represent the (temporary) Skyline. From an algorithmic point of view this is done by a combination of breadth-first traversal (BFT) and depth-first traversal (DFT).

   We start a BFT at the top of BTG (line 1) and search for the first non-empty node, which is not dominated. From this node on, we start a DFT to mark dominated nodes (line 4 and procedure DFT in line 9) recursively. When processing objects from the next chunk, this ensures that we do not need to add objects to already dominated nodes in the BTG. This reduces memory requirements and enhances performance. After processing all nodes in the DFT, we continue with the BFT until all nodes are visited. The remaining nodes contain the temporary Skyline set and can be presented to the user.

4. Since there is a continuous data stream, Phase 2 and Phase 3 have to be repeated for the next chunks.

The Skyline computation in Phase 3 can be done after an arbitrary number of processed chunks or after a pre-defined time. Therefore, our algorithm can be used for real-time Skyline evaluation. It is also possible to parallelize this approach in the sense of [27,28]: Parallelize the adding of the objects in the chunk, and, after adding an object, directly start a DFT to mark nodes as dominated.

Since SLS follows the idea of Hexagon and Lattice Skyline, the linear runtime complexity of $O(dn + dV)$ remains for our algorithm. Thereby, $n$ is the number of input objects, $d$ number of dimensions, and $V$ the size of the lattice, i.e., the product of the cardinalities of the $d$ low-cardinality domains from which the attributes are drawn.
Phase 3 Removal Phase

1: Start a BFT beginning at the top of the BTG
2: for each ID in BFT do
3:   if \(\neg BTG[ID].isDominated() \land \neg BTG[ID].isEmpty()\) then
4:     DFT(ID) \(\triangleright\) Start depth-first traversal to mark dominated nodes
5:   end if
6: end for
7: \(\triangleright\) Start DFT
8: DFT(ID) =
9: for each successor sID of ID do
10:   if \(BTG[sID].isDominated()\) then return \(\triangleright\) Node is already dominated
11: end if
12: \(\triangleright\) Otherwise remove dominated nodes and continue DFT
13: \(BTG[sID].setStatus =\) dominated
14: \(BTG[sID].clear()\) \(\triangleright\) Set objects to null
15: DFT(sID) \(\triangleright\) Recursion, continue with successor of sID
16: end for

4 Experiments

In this section we present comprehensive experiments on our SLS algorithm.

4.1 Benchmark Framework

In our benchmarks we wanted to explore the behavior of SLS on synthetic and real-world data, depending on the data size, chunk size, and different domain size. For runtime evaluation we compared SLS to the stream variant of BNL, cp. Section 2.2, because to the best of our knowledge, this is the only other stream-based Skyline algorithm.

For our experiments on artificial data we generated correlated (corr), anti-correlated (anti), and independent (ind) data streams as described in [12] and varied three parameters: (1) the data cardinality (n), (2) the data dimensionality (d), and (3) the number of distinct values for each attribute domain.

For real data, we used Twitter records collected over a specific period of time. These objects (tweets) include various attributes such as name, description, created_at, followers_count, status_count, lang, and many more. We mapped all attribute values of these short messages to a numerical domain according to a mapping function described in Section 3.1.

For analyzing a data stream, it is necessary to divide it into a series of chunks \(c_t\) as described in Section 2.2. For this we used Apache Flink\(^2\), an open source platform for scalable stream and batch data processing, which is also able to process real-world data like Twitter. For more details on the implementation of our personalized stream processing framework we refer to [29,30,31].

\(^2\) Apache Flink: https://flink.apache.org/
Our algorithms have been implemented using Java 8. All experiments are performed on a single node running Debian Linux 7.1. The machine is equipped with two Intel Xeon 2.53 GHz quad-core processors.

4.2 Influence of the Chunk Size

In our first experiment we varied the chunk size to find out the optimal number of objects per chunk. We also compared BNL to SLS w.r.t. their runtime depending on the chunk size. We used datasets with 100K and 500K objects and considered the algorithms behaviour for anti-correlated, independent and correlated data distribution.

For a more reliable result we considered different domains: [1, 2, 2, 3], [1, 5, 10], [1, 2, 2, 2, 2, 2, 2, 3], [2, 3, 7, 8, 4, 10], [1, 5, 2560] and [13, 35, 70]. Remember, each number corresponds to the maximal possible values of the single domains.

Figure 3 shows our results for anti-correlated data. In all experiments SLS performs significant better than BNL independent of the chunk size. For small chunks (up to 500 objects) and for very large chunks (more than 50K objects), BNL is substantial worse than SLS. For small chunk sizes this can be explained by a higher number of unions which has to be carried out after each chunk evaluation, cp. Equation 3. For larger chunks the object comparison process takes more time.

Fig. 3: Influence of the chunk size. SLS vs BNL, anti-correlated data distribution.
Real-Time Skyline Computation on Data Streams with SLS

Fig. 3: Influence of the chunk size. SLS vs BNL, anti-correlated data distribution.
SLS seems to be nearly constant w.r.t. the runtime. However, a closer look to the runtimes of SLS spawns that SLS is slower for small chunks (up to 200 objects) than for chunks with more than 200 objects, cp. Figures 6a to 6d on page 16. This can be explained by the frequent repeating of the BFT and DFT in SLS, which have to be carried out for each chunk. For the chunk size over 20K objects the runtime of SLS increases again, because the adding of new objects to the BTG (Phase 2) in SLS is more expensive. In summary, we claim that the optimal chunk size for the best runtime with anti-correlated data distribution is between 200 and 20K objects.

Figure 4 (pages 12 – 13) presents the comparison result of SLS and BNL for independent data distribution. We can see, that SLS has better runtime for small and medium chunks (up to 10K – 20K depending on the domains), but for large chunks both algorithms have very similar runtime. For some domains, e.g. [1,5,2560] and [13,35,70] (cp. Figures 4i to 4l) BNL outperforms SLS. In the case of these two domains we are dealing with the high cardinality domains, which produce deep BTGs in the sense of the height. The required time for the depth search (DFT) in the deep BTG (cp. Removing Phase 3 in Section 3.2) for such domains is significant longer than for low cardinality domains. This enables better runtime of BNL compared to SLS for chunk sizes from 1K objects for domains with high cardinality.

![Figure 4: Influence of the chunk size. SLS vs BNL, independent data distribution.](image-url)
Fig. 4: Influence of the chunk size. SLS vs BNL, independent data distribution.
For a better estimation of the optimal chunk size for independent data, we take a closer look to the runtime of SLS. Figures 6e to 6h on page 16 demonstrate the best results for the chunk size between 1K and 10K objects.

In Figure 5 (pages 14–15) we can observe the runtime comparison of BNL and SLS for correlated data. For a chunk size up to 2K objects SLS is much better than BNL. For larger chunks both algorithms show very similar runtime.

Fig. 5: Influence of the chunk size. SLS vs BNL, correlated data distribution.
As we can see in Figures 6i to 6l on page 17, the SLS achieves its best runtime results for correlated data where the chunk size is between 200 and 20K objects.

4.3 Influence of Different Domains

In this experiment, we explored the influence of different domains on SLS. We used data with 10K and 500K objects.

The results on anti-correlated data are shown in Figure 7 (page 18). In Figures 7a and 7b we varied the number of attributes from 4 to 9, while the domain values remain within the low-cardinality range \{0, ..., 10\}. The runtime
Fig. 6: Influence of the chunk size on SLS.
behavior is similar for all domains, because low-cardinality domains produce flat BTGs and therefore the runtime for the DFT search in the BTG is nearly constant. Note that the \([2,3,7,8,4,10]\) domain has some 47.5K nodes, \([1,2,2,2,2,2,2,2,3]\) produces a BTG with 17.5K nodes and \([1,2,2,3]\) has only 72 nodes.

In Figures 7c, 7d and 7e we compared domains with the same number of attributes, but varied strongly the number of distinct values for each attribute and generated the high-cardinality domains. These domains produce deeper BTGs in the sense of the height, but observe a similar behavior as in Figure 7a and 7b. There are some 36K nodes in the largest BTG and only 132 nodes in the smallest.

In Figure 7f we compared two domains ([1,2,2,3] and [2,1,2,3,2,1,10,2,2]) producing flat BTGs and two domains building deep BTGs ([13, 35, 70] and [1,5,2560]). As we can see in this figure, SLS needs more time for deep BTGs, because the required time for depth search (DFT) (c.p. Removing Phase 3 in Section 3.2) for high-cardinality domains is significant longer than for low cardinality domains.

In summary, the runtimes of SLS are nearly independent from the number of attributes and the size of the domain, as long as we have low-cardinality domains. The best chunk size for anti-correlated data distribution is between 200 and 20K objects for all tested domains.
The influence of different domains on SLS for independent data is shown in Figure 8 (page 19). Similar to the experiments on anti-correlated data, we compared low-cardinality domains producing flat BTGs in Figures 8a and 8b, high-cardinality domains creating deep BTGs in Figures 8c, 8d, and 8e, and mixed these domains in Figure 8f.

The SLS behaviour is very similar for different low-cardinality domains and for various high-cardinality domains. Furthermore, the runtime of SLS is higher for deep BTGs, due to the higher runtime of the DFT.

The best runtime results SLS shows for the chunk sizes up to 10K objects. Compared to anti-correlated data, the results for independent data have no troubles with small chunks. The very large chunks (more than 10K objects)
require more runtime because the adding of new objects to the BTG (Phase 2 Section 3.2) in SLS is more expensive.

![Graphs showing runtime vs chunk size for different domains.](image)

Fig. 8: Influence of different domains, n=500K, independent data distribution.

Figure 9 (on page 20) shows the experimental results for correlated data. We used the same domains as for anti-correlated and independent data and compared flat and deep BTGs. These results also confirm our assumption: runtimes of SLS are nearly independent from the number of attributes and the size of the domain, as long as we have low-cardinality domains. Deep BTGs produced by high-cardinality domains are processed by SLS more slowly. The best runtimes were shown for chunk sizes between 200 and 20K objects.
4.4 Influence of the Data Distribution

In this experiment we wanted to investigate the impact of different data distributions on SLS. We used independent (ind), correlated (cor), and anti-correlated (anti) data. We varied the size of the dataset (100K and 500K objects), and the domains. In Figure 10 (page 21) we compared low-cardinality domains producing flat BTGs and in Figure 11 on page 22 we demonstrated the SLS behaviour for different data distributions with high-cardinality domains generating deep BTGs.
SLS relies on the lattice structure a Skyline query constructs and does not depend on any object-to-object comparisons. Therefore, we expected that the runtime of SLS is nearly the same for any kind of data distribution.

This expectation was completely fulfilled for low-cardinality domains as confirmed by Figure 10. For high-cardinality domains, SLS takes more time for the processing of very small (up to 200 objects) and very large (over 20K objects) chunks. We observed the best runtime for the chunks with [200; 20K] objects, as already shown in Section 4.2.
4.5 Runtime Comparison of SLS, Hexagon and BNL

In this section we describe experiments in which we compared SLS with BNL and Hexagon. Since Hexagon is not able to process streams, we analysed the runtimes of Hexagon and compared them to the runtimes of SLS and the stream-based version of BNL with only one big chunk which is equal to the dataset size. We varied the data cardinality, data distribution and domains.

Figure 12 (page 23) presents the result for anti-correlated data. In this case SLS clearly outperforms BNL: The larger the dataset, the greater the difference.

The results for independent data are presented in Figure 13 (page 24). SLS shows often shows the best runtime, but sometimes BNL is better (for example with 50K objects for [1,1,2,2,2,2,2,2,2,3] domain or with 500K objects for [2856,1,5] domain).

For correlated data SLS demonstrates better results for all chunk sizes except 1000K objects. For this size BNL is clearly faster (see Figure 14 on page 25).

We tested also Hexagon in order to find out if our SLS implementation (based on Hexagon) shows very similar runtime results. And indeed, the two algorithms show very similar runtimes how we can see in Figures 12, 13 and 14.
Fig. 12: SLS vs Hexagon vs BNL, anti-correlated data distribution.
Fig. 13: SLS vs Hexagon vs BNL, independent data distribution.
Fig. 14: SLS vs Hexagon vs BNL, correlated data distribution.
4.6 Real-World Data

For real data experiments we used tweets collected from Twitter over a specific period of time (for repeatable experiments). We used (disjunct) datasets of 100K and 500K objects. We mapped all attributes to a numerical domain according to a mapping function as described in Section 3.1. For example, we mapped status_count (number of posted tweets), followers_number and hashtag to the numerical domain [2856,5,1].

In our first experiment we compared the runtime of SLS and BNL on different data sizes, but the same domain, cp. Figure 15.
Again, for BNL the worst runtime is for small chunks due to the frequent repetition of tuple-comparisons in each single chunk. However, for larger chunks, BNL becomes better. We assume that there are some killer objects (objects better than most of the other objects in the dataset), which can be accessed earlier by BNL through the larger chunk sizes and therefore speed-ups performance. Nevertheless, SLS is still better than BNL, in particular for the larger dataset.

In our second experiment we explored the runtime of SLS for real Twitter data in comparison to generated independent data having the same domains ([4,126,77], [4,77,2,10] and [2856,5,1]). We found it interesting to compare real data to independent generated data, since real data is often assumed to be independent distributed. Figure 16 presents our results.
Since real data is not always perfectly independent distributed and there are some killer objects the results show the expected behavior: SLS performs significant better on real data than on artificial data. In addition, we see that also for real data chunk sizes between 200 and 20K objects are a good choice.

In summary, we conclude that SLS has an optimal runtime for a chunk size between 200 and 20K objects. For this range, SLS outperforms BNL for different domains, dataset sizes, and data distributions for real and synthetic data.

5 Conclusion

There is significant interest of the research community towards continuous query processing. And one information filtering approach on data streams is to compute the Skyline set. In this paper, we presented our novel algorithm SLS to find the Skyline on a data stream. Exploiting the lattice, SLS does not rely on object-to-object comparisons like BNL-style approaches, is independent of any data partitioning, and has a linear runtime complexity. In addition, SLS fulfills all requirements on modern stream algorithms: 1) fast runtime as seen in our experiments, 2) incremental evaluation since new chunks can easily be added to the BTG, 3) limited number of data access, because only non-dominated objects are added to the BTG, and 4) in-memory storage since the BTG is hold in RAM to avoid expensive I/O accesses. We also presented comprehensive results on characteristic experiments to confirm that SLS is currently the most advanced real-time Skyline stream processing algorithm.

Nevertheless, there are still open issues which must be addressed in the future. For example, SLS can be parallelized as described in [27,28]. Also we want to enhance our algorithm to handle unrestricted high-cardinality domains as depicted in [32]. However, this could be a challenging task.

Also, maybe index structures could be used for stream data preference processing, cp. [33].

References


