ARTful Skyline Computation for In-Memory Database Systems

Maximilian E. Schüle, Alex Kulikov, Alfons Kemper, and Thomas Neumann

Technical University of Munich
{m.schuele, alex.kulikov, alfons.kemper, thomas.neumann}@tum.de

Abstract. Skyline operators compute the Pareto-optimum on multi-dimensional data inside disk-based database systems. With the arising trend of main-memory database systems, pipelines process tuples in parallel and in-memory index structures, such as the adaptive radix tree, reduce the space consumption and accelerate query execution.
We argue that modern database systems are well suited to progressive skyline operators. In addition, space-efficient index structures together with tree-based skyline algorithms improve the overall performance on categorical input data. In this work, we parallelise skyline algorithms, reduce their memory consumption and allow their integration into the main-memory database system HyPer. In our evaluation, we show that our parallelisation techniques scale linearly with every additional worker, and that the adaptive radix tree reduces memory consumption in comparison to existing tree-based approaches for skyline computation.

Keywords: Skyline Operator · In-Memory DBMS · Adaptive Radix Tree.

1 Introduction

The skyline algorithm finds interesting tuples within multi-dimensional data sets. Specifically, these tuples form the Pareto-optimal set that contains only the best tuples regarding all criteria. Formally, the output set is defined as all tuples that are not dominated by any other tuple of the input set.

From a theoretical point of view, computing the skyline of a set of tuples corresponds to the mathematical problem of finding the maxima of a set of vectors [8]. A vector $p \in \mathbb{R}^n$ dominates another vector $q \in \mathbb{R}^n$ if p is at least as good as q in every dimension, and superior in at least one:

$$p \succ q \Leftrightarrow \forall i \in [n].p[i] \succeq q[i] \wedge \exists j \in [n].p[j] \succ q[j]. \quad (1)$$

Börzsönyi et. al. [2] provided the first skyline implementations and an SQL extension (see Listing 1.1). Their work compared the algorithm to a skyline formed out of skyscrapers: only those are visible which are either closer (regarding the distance) or higher than any other to a specific viewing point. Even though SQL-92 is capable of expressing skyline queries (see Listing 1.2), the

authors proposed an integration as an operator that executes optimised skyline algorithms within the database system.

```
SELECT * FROM inputtable i WHERE ... GROUP BY ... HAVING ...
SKYLINE OF [DISTINCT] d1 [MIN | MAX], ... , dn [MIN | MAX]
ORDER BY ...
```

Listing 1.1. Skyline extension of SQL: d_1, ... , d_n are the dimensions; MIN and MAX specify whether each dimension has to be minimised or maximised.

```
SELECT * FROM inputtable q WHERE NOT EXISTS (
  SELECT * FROM inputtable p WHERE p.d1 <= q.d1 AND ... AND p.dn<=q.dn
    AND (p.d1 < q.d1 OR ... OR p.dn<q.dn ))
```

Listing 1.2. Skyline query in SQL on a table *inputtable* with attributes d_1, ... , d_n.

We argue that integrated skyline operators can benefit from modern database systems that offer in-memory index structures as well as pipelined tuple processing. In the following, we integrate the naive-nested-loops skyline algorithm as an operator into the main-memory database system HyPer [6]. As a native operator, it supports code-generation according to the producer-consumer concept, that pushes tuples towards the parent operator in parallel pipelines. This enables the parallelisation of progressive skyline algorithms that continuously produce output. Furthermore, we optimise the space requirements for trie-based skyline algorithms and parallelise all introduced implementations for multi-threaded execution. In summary, this work's contributions are:

- the integration of a skyline operator into the main-memory database system HyPer following the producer-consumer model,
- a memory reduction for trie-based skyline computation on categorical data due to the usage of the adaptive radix tree,
- the parallelisation of naive-nested-loops as well as tree-based skyline algorithms within the context of database systems
- and an evaluation in terms of run time, memory usage and scalability that compares naive-nested-loops to tree-based skyline algorithms.

This work is organised as follows: First, we give an overview of the underlying main-memory database system with its adaptive radix tree and existing skyline algorithms. Hereafter, this paper proposes a novel skyline algorithm called SARTS, which uses the adaptive radix tree for dominance checks. Afterwards, parallelisation techniques are first explained and then applied to the given skyline algorithms. For the evaluation, we vary the number of input tuples as well as the number of available threads.

2 Related Work

As this work combines main-memory database systems with skyline algorithms, this section introduces the underlying operator concepts within modern database systems and common skyline algorithms.

2.1 Main-Memory Database Systems

HyPer [6,13,14] is an in-memory database system that introduced code-generation according to the producer-consumer model. Instead of traditional Volcano-style query execution [4], where the topmost operator iteratively demands the underlying ones to return tuples, operators in HyPer push tuples towards the parent operator. Two functions, `produce()` and `consume()`, generate the corresponding code using the LLVM compiler framework. During code-generation, `produce()` is called recursively from top to bottom, then each call evokes a `consume()` call on the parent node. This generates the code for processing tuples in parallel pipelines. We later integrate skyline as an operator that initiates parallel pipelines.

The adaptive radix tree (ART) [9] is the in-memory index-structure used in HyPer to retrieve tuples by their identifier. In contrast to a radix tree, the node's size is adaptive in order to reduce the memory consumption and improve the caching performance. The ART offers four different node types for either four, 16, 48 or 256 keys and can replace various tries such as prefix-trees represented by radix trees.

2.2 Skyline Algorithms

The naive-nested-loops (NNL) algorithm from the original paper [2] forms the basis for our in-database implementation A nested loop compares each tuple to each other one whether it is not dominated and therefore forms part of the return set. To reduce the number of disk accesses, the block-nested-loops (BNL) algorithm maintains a window in main-memory of all tuples considered for the skyline to that point. The divide-and-conquer (DNC) algorithm partitions [16] the tuples recursively and performs dominance checks when merging partitions.

Since its invention, skyline algorithms have been based on different data structures and hardware [5]. This work mainly incorporates research on the parallelisation of skyline algorithms [7,10,15,17] that produce progressive output [11]. This facilitates the integration into database systems according to the producer-consumer model.

Sorting-based algorithms such as Sort-Filter-Skyline (SFS) [3] or SaLSa [1] pre-sort the input first before computing the skyline. Pre-sorted input allows elements to be pruned which are worse. The skyline-using-tree-sorting (ST-S) algorithm [12] is tuned for binary attribute values, as it stores tuples in a radix tree called *N-tree* to perform dominance checks. In this work, we extend the algorithm to support categorical data and replace the N-tree with the ART.

3 SARTS

This section presents SARTS (Skyline using ART Sorting-based), a novel skyline algorithm for categorical attributes. It improves the core concepts of ST-S by implementing a more efficient indexing structure for dominance checks—the

ART. As our proposed SARTS algorithm is based on the ST-S algorithm, we first explain the extension of ST-S for categorical attributes, before we proceed with the integration of the adaptive radix tree.

3.1 ST-S for Categorical Attributes

Every inner node of the N-tree in the ST-S algorithm, including the root, has an array, which can hold as many children as there are possible attribute values. Each path taken from the root to a leaf represents a tuple, which is assigned a score. The score of a particular tuple t with n attributes is determined by a scoring function with $t[i]$ as the i-th attribute of the tuple:

$$score(t) := \sum_{i=0}^{i<n} 2^{n-i} \cdot t[i]. \tag{2}$$

In each inner node, `minScore` and `maxScore` mark the boundaries for the tuple's possible score within branches descending from this node.
At the beginning, a monotonic function `minC()` or `maxC()` defines an order:

$$minC(t) := \big(\min_{0 \le i < n} (t[i]), \sum_{i=0}^{i<n} t[i] \big). \tag{3}$$

It consists of two components: a main comparison attribute, which is the smallest value of all tuple's attributes, and a tie-breaker that is the sum of all the tuple's attribute values. The ST-S algorithm (Algorithm 1) works as follows:

1. The tuples are presorted with `minC()` (line 1).
2. The threshold tuple t_{stop}, undefined at the beginning, is later updated (lines 11-12) with knowledge of tuples that are part of the skyline.
3. The first tuple t_0 from the presorted data set is always part of the skyline. It gets inserted into the tree and is put out as part of the skyline (lines 3-5).
4. The following loop checks for every input tuple t whether it is dominated by any tuple already in the skyline (line 8). The checks are carried out with the help of the tree, which holds all the skyline tuples to date.
5. If a tuple t is dominated by some other tuple in the skyline, it is no longer considered (line 8). Otherwise, it is inserted into the tree (line 9), so that it is able to eliminate future, dominated tuples.
6. If the maximum attribute value of the new skyline tuple t is smaller than the maximum attribute value of the threshold t_{stop}, then the threshold is updated (line 12), and now holds the value of t, until the next update occurs.
7. The algorithm stops as soon as all tuples left in the data set are a priori dominated by the threshold (line 7).
8. If the maximum attribute value of the threshold tuple t_{stop} is less than or equal to the minimum attribute value of the current tuple t, then none of the remaining, sorted tuples are part of the skyline.

Both the `insert()` and `is_dominated()` operations have been slightly modified from the original paper to deal with categorical attributes rather than binary ones.

Algorithm 1 ST-S Algorithm

Input: Tuple List T, Tree *tree*
Output: Skyline *skyline*

```
1: Sort T in-place using a monotonic function minC()
2: t_0 ← first element of T
3: t_stop ← t_0
4: insert(t_0, tree.root, 0)
5: Add t_0 to skyline // t_0 always part of skyline due to presorting
6: for each tuple t ∈ T\{t_0} do
7:     if max(t_stop) ≤ min(t) and t_stop ≠ t then return
8:     if not is_dominated(t, tree.root, 0, score(t)) then
9:         insert(t, tree.root, 0)
10:        Add t to skyline
11:        if max(t) < max(t_stop) then
12:            t_stop ← t
```

3.2 ART for Skyline

The interface of the ART has been kept similar to that of the N-Tree in ST-S. This enables the very straightforward integration of the ART into the algorithm, because the `insert()` and `is_dominated()` operations still have the same signature as in ST-S. While `insert()` is slightly different from the original variant, the `is_dominated()` operation is almost identical to the one in ST-S. The `insert()` operation for SARTS differs from the ST-S variant in this both finding the correct child to the current node and creating a new child are outsourced into two separate functions: `findChild()` and `newChild()`. In addition to that, before a new child can be created, the current node might first need to `grow()` to the next-bigger type, in order to create space for the new child. The pseudo-code to the `insert()` operation is given in Algorithm 2.

The main difference within the `is_dominated()` operation is, similarly to `insert()`, that it uses `findChild()` to determine the correct child for further traversal.

In addition to that, just like the nodes of the N-Tree, the inner nodes of the ART have to be extended by a `minScore` and a `maxScore`, and the leaf nodes by the `score` attribute and an array of `tupleIDs`. This enables the faster traversing of the tree during dominance checks, by skipping tree regions that cannot dominate the current tuple.

4 Parallelisation

The following section presents the parallelisation approaches for traditional skyline algorithms, such as NNL and DNC [1], as well as for two of the newer algorithms: ST-S and SARTS[2]. The corresponding source-code is publicly available.

[1] `https://gitlab.db.in.tum.de/alex_kulikov/skyline-computation`
[2] `https://gitlab.db.in.tum.de/alex_kulikov/skyline-categorical`

Algorithm 2 INSERT Operation for SARTS

Input: Tuple t, Node *parent* Node *current*, Level *level*, Attributes *atts*

if $level = 0$ **then**
  $node.minScore \leftarrow 0$
  $node.maxScore \leftarrow \sum_{i=0}^{t.size-1}(2^{t.size-i} \cdot max(atts))$
else if $level \mathrel{!}= t.size$ **then**
  $node.minScore \leftarrow \sum_{i=0}^{level-1}(2^{t.size-i} \cdot t[i])$
  $node.maxScore \leftarrow node.minScore + \sum_{i=level}^{t.size-1}(2^{t.size-i} \cdot max(atts))$
if $level = t.size$ **then**
  $node.score \leftarrow score(t)$
  Append $t.tupleID$ to $node.tupleIDs$
else
  $child \leftarrow findChild(current, t[level])$
  if *child* is *None* **then**
    if $current.size \mathrel{!}= 256$ **then**
      $grow(parent, current, t[level - 1])$
    $child \leftarrow newChild(current, t[level])$
  insert(t, *current*, *child*, $level + 1$)

4.1 Naive-/Block-Nested-Loops

The main idea when parallelising the naive-nested-loops algorithm is to use the `parallel_for` construct for the outer loop of the algorithm. The inner loop could also be taken for this purpose, but then the code, which finds itself in the outer loop but not in the inner one would be running sequentially, thus reducing the benefit of parallelising the code in the first place. The pseudo-code notation of the parallelised version of naive-nested-loops is given in Algorithm 3.

Algorithm 3 Parallel NNL

Input: Tuple List T
Output: Skyline *skyline*

parallel_for each tuple $t \in T$ **do**
  $is_not_dominated \leftarrow$True
  for each tuple $d \in T \backslash \{t\}$ **do**
    if dominates(d, t) **then**
      $is_not_dominated \leftarrow$False
      break
  if $is\_not\_dominated$ **then**
    Add t to *skyline*

4.2 Divide-and-Conquer

Two different parallelisation techniques were applied to the divide-and-conquer algorithm. Instead of applying a sequential sorting algorithm, `parallel_sort` sorts the elements using several worker threads simultaneously, and thus produces the result significantly faster than sequential functions for large data sets. `parallel_sort` is applied in two places within the DNC algorithm:

1. *Finding the median.* After sorting the tuples, the median of the data set is taken to be the element located exactly in the middle of the sorted set.

2. *Determining the minimum for two dimensions.* The skyline can be computed by finding the minimum of the first subset and comparing it to all elements of the second subset.

4.3 SARTS and ST-S

The parallelising of the ST-S and SARTS algorithms results in almost identical implementations. As the interfaces of both trees are technically the same, the algorithms were also parallelised via the same approach.

The main idea is to divide the original data set into as many partitions as there are threads on the machine. One thread for each partition computes the skyline of its tuples. Every thread receives its own tree structure to store the tuples that are part of the skyline and to perform dominance checks. In other words, the sequential version of SARTS (resp. ST-S) is simultaneously applied to each of the partitions. As soon as the skyline of every partition has been computed, the resulting skylines are merged to produce the final one. The skylines of all partitions combined are much smaller than the original data set. Therefore, the final merge does not take as much time as computing the entire skyline from scratch.

In addition to the main parallelisation approach, presorting the tuples also happens in parallel before the actual algorithm begins. As the original data set tends to be very large in real-world applications, sorting it in parallel leads to a very significant efficiency boost.

The skyline is computed similarly to the non-parallelised version, with one major difference. Whenever a tuple that is definitely part of the skyline is stored, it is not merely appended to some list of skyline tuples. Instead, it is stored into a common `sub_results` array, to which all skyline threads share access.

5 Evaluation

This section discusses the evaluation of the following algorithms: naive-nested-loops (NNL), block-nested-loops (BNL), divide-and-conquer (DNC), ST-S and SARTS. All the tests were conducted on a Linux Mint 18.2 machine offering an Intel Core i7-5500U CPU with a 4096 KB cache and 8 GB DDR3L of main-memory. As the tree-based skyline algorithms, such as ST-S and SARTS, are restricted to categorical attributes, tests that include the algorithms ST-S and SARTS were conducted using a limited set of integers as categories, ranging from 0 to 255. All other tests were performed with continuous attributes, represented as `double` values.

5.1 Non-Progressive Algorithms

The non-progressive skyline types included in this work are the block-nested-loops and the divide-and-conquer algorithms. In all three of the conducted tests, BNL scales significantly better than DNC. It shows overall better performance

with an increasing number of tuples, dimensions and also threads (Figure 1). Herewith, the results are similar to the ones produced in the original paper [2], which introduced BNL and DNC.

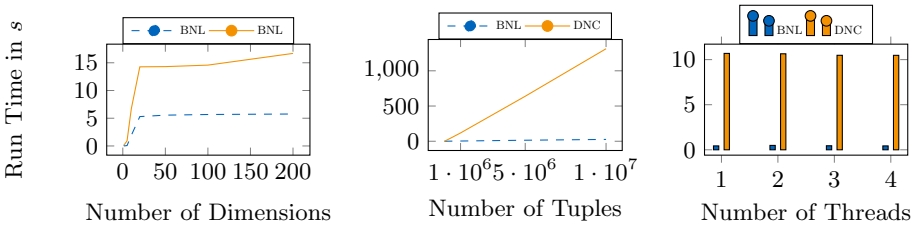

Fig. 1. Run time of non-progressive algorithms by number of tuples (default: 5 dimensions, 256 categories, 4 threads and 10,000 input tuples).

5.2 Progressive Algorithms

Naive-nested-loops can be both progressive and parallelisable and therefore compared to the two newer algorithms ST-S and SARTS. As expected, for a rising number of tuples, both ST-S and SARTS perform extremely well. As shown in Figure 2, they significantly outperform naive-nested-loops with larger input. This is not surprising, as ST-S and SARTS were specifically developed for large categorical data sets. It is due to the efficient nature of the tree structures used that dominance checks can be conducted very efficiently, and depend less on the number of tuples than on the dimensionality of the data set.

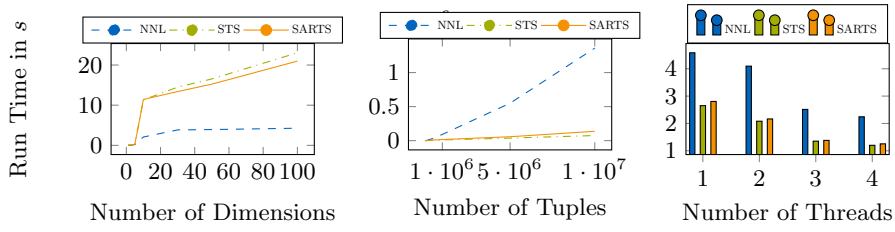

Fig. 2. Run time of progressive algorithms by number of tuples (default: 5 dimensions, 256 categories, 4 threads and 10,000 input tuples).

When looking at the results of scaling with dimensionality, the naive-nested-loops algorithm significantly outperforms both parallelised and sequential versions of ST-S and SARTS. The reason for this is that radix-based tree structures—N-Tree and ART—generally scale badly with longer keys. This is the trade-off

they have to accept for very efficient scaling with the number of inserted elements. The longer the keys of the data set are, the higher the tree gets, and the longer it takes to traverse the tree from top to bottom. In the application area of skyline computation, the length of a key corresponds to the dimensionality of a tuple. Hence, the more dimensions the tuples of a data set have, the less efficient tree-based dominance checks become.

A comparison of progressive algorithms depending on the number of threads available shows that both ST-S as well as SARTS outperform naive-nested-loops. As expected, all parallelised algorithms scale with the number of available threads.

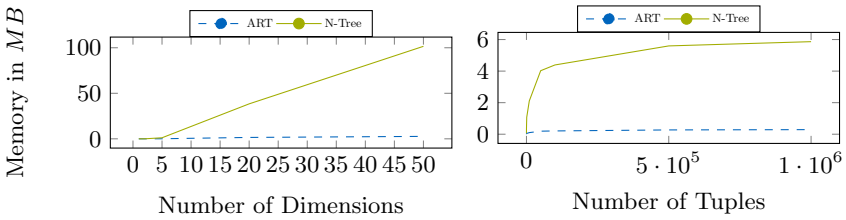

Fig. 3. Memory usage of ART and N-Tree by dimensionality and tuples (256 categories, 4 threads; left: 1000 input tuples, right: 5 dimensions).

Memory Usage The last two tests compare the main-memory usage of the ART to that of the N-Tree. Figure 3 shows that the ART significantly consumes less space than the N-Tree. The memory consumption is similar when using multiple dimensions: While the ART already performs better than the N-Tree for low number of dimensions, it generally scales much more efficiently with high dimensionality. Thus, it can be concluded that the SARTS algorithm is significantly more memory-efficient than ST-S due to the usage of the ART.

6 Conclusion

This work has integrated skyline algorithms as an operator inside the main-memory database system HyPer according to the producer-consumer model. As in-memory index structures improve look-up performance in main-memory database systems, we replaced traditional radix trees by the adaptive radix tree for fast skyline computation on categorical data. This called SARTS algorithm displayed the same lookup performance as its ancestor algorithm, ST-S, but was superior with regard to space consumption, due to adaptive nodes. We successfully parallelised naive-nested-loops, divide-and-conquer and the tree-based algorithms to allow scaling to multiple cores.

References

1. Bartolini, I., Ciaccia, P., Patella, M.: Efficient sort-based skyline evaluation. ACM Trans. Database Syst. **33**(4) (2008). https://doi.org/10.1145/1412331.1412343
2. Börzsönyi, S., Kossmann, D., Stocker, K.: The skyline operator. In: ICDE, April 2-6, 2001, Heidelberg, Germany. IEEE Computer Society (2001). https://doi.org/10.1109/ICDE.2001.914855
3. Chomicki, J., Godfrey, P., Gryz, J., Liang, D.: Skyline with presorting. In: ICDE, March 5-8, 2003, Bangalore, India. IEEE Computer Society (2003). https://doi.org/10.1109/ICDE.2003.1260846
4. Graefe, G.: Encapsulation of parallelism in the volcano query processing system. In: SIGMOD, Atlantic City, NJ, USA, May 23-25, 1990. ACM Press (1990). https://doi.org/10.1145/93597.98720
5. Hose, K., Vlachou, A.: A survey of skyline processing in highly distributed environments. VLDB J. **21**(3) (2012). https://doi.org/10.1007/s00778-011-0246-6
6. Hubig, N., Passing, L., Schüle, M.E., Vorona, D., Kemper, A., Neumann, T.: Hyperinsight: Data exploration deep inside hyper. In: CIKM, Singapore, November 06 - 10, 2017 (2017). https://doi.org/10.1145/3132847.3133167
7. Köhler, H., Yang, J., Zhou, X.: Efficient parallel skyline processing using hyperplane projections. In: SIGMOD, Athens, Greece, June 12-16, 2011. ACM (2011). https://doi.org/10.1145/1989323.1989333
8. Kung, H.T., Luccio, F., Preparata, F.P.: On finding the maxima of a set of vectors. J. ACM **22**(4) (1975). https://doi.org/10.1145/321906.321910
9. Leis, V., Kemper, A., Neumann, T.: The adaptive radix tree: Artful indexing for main-memory databases. In: ICDE, Brisbane, Australia, April 8-12, 2013. IEEE Computer Society (2013). https://doi.org/10.1109/ICDE.2013.6544812
10. Liknes, S., Vlachou, A., Doulkeridis, C., Nørvåg, K.: Apskyline: Improved skyline computation for multicore architectures. In: DASFAA, Bali, Indonesia, April 21-24, 2014. (2014). https://doi.org/10.1007/978-3-319-05810-8_21
11. Papadias, D., Tao, Y., Fu, G., Seeger, B.: An optimal and progressive algorithm for skyline queries. In: SIGMOD, San Diego, California, USA, June 9-12, 2003. ACM (2003). https://doi.org/10.1145/872757.872814
12. Rahman, M.F., Asudeh, A., Koudas, N., Das, G.: Efficient computation of subspace skyline over categorical domains. In: CIKM, Singapore, November 06 - 10, 2017. ACM (2017). https://doi.org/10.1145/3132847.3133012
13. Schüle, M., Bungeroth, M., Vorona, D., Kemper, A., Günnemann, S., Neumann, T.: ML2SQL - compiling a declarative machine learning language to SQL and python. In: EDBT, Lisbon, Portugal, March 26-29, 2019 (2019). https://doi.org/10.5441/002/edbt.2019.56
14. Schüle, M., Vorona, D., Passing, L., Lang, H., Kemper, A., Günnemann, S., Neumann, T.: The power of SQL lambda functions. In: EDBT, Lisbon, Portugal, March 26-29, 2019 (2019). https://doi.org/10.5441/002/edbt.2019.49
15. Tang, M., Yu, Y., Aref, W.G., Malluhi, Q.M., Ouzzani, M.: Efficient parallel skyline query processing for high-dimensional data. In: ICDE, Macao, China, April 8-11, 2019. IEEE (2019). https://doi.org/10.1109/ICDE.2019.00251
16. Vlachou, A., Doulkeridis, C., Kotidis, Y.: Angle-based space partitioning for efficient parallel skyline computation. In: SIGMOD, Vancouver, BC, Canada, June 10-12, 2008. ACM (2008). https://doi.org/10.1145/1376616.1376642
17. Zois, V., Gupta, D., Tsotras, V.J., Najjar, W.A., Roy, J.: Massively parallel skyline computation for processing-in-memory architectures. In: PACT, Limassol, Cyprus, November 01-04, 2018. ACM (2018). https://doi.org/10.1145/3243176.3243187