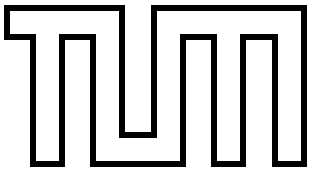


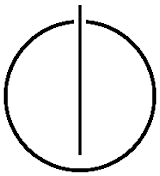
DEPARTMENT OF INFORMATICS

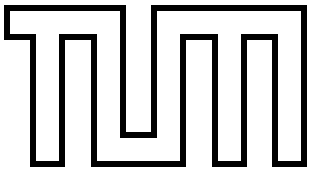
TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Parallelization of Efficient Tree Structures for Skyline Computation

Oleksii Kulikov





DEPARTMENT OF INFORMATICS

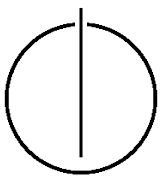
TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Parallelization of Efficient Tree Structures for Skyline Computation

Parallelisierung effizienter Baumstrukturen für Skyline Berechnung

Author:	Oleksii Kulikov
Supervisor:	Prof. Alfons Kemper, Ph.D.
Advisor:	Maximilian E. Schüle, M.Sc.
Date:	October 15, 2018



I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, October 8, 2018 Oleksii Kulikov

Abstract

The purpose of this work is to adapt and parallelize skyline computation for main memory database systems, as well as to adapt the ART tree for categorical skyline computation.

Skyline algorithms produce their results much faster when integrated into a database system, instead of running on top of it. Therefore, this work adjusts some of the most common skyline algorithms, so that they can be used as a building block in main memory databases. In addition to that, modern parallelization approaches are introduced and applied to the algorithms. The conducted performance tests show that the suggested parallelization approaches have been successful in reducing the computation time of the algorithms when running in parallelized environments.

Moreover, the ART tree is used to develop a novel skyline algorithm called SARTS. The algorithm is able to progressively output skyline tuples in online environments, while being fast and keeping its memory usage to a minimum. As shown in the evaluation, it can keep up with its predecessor ST-S in terms of computation time, while using up to 20 times less memory at runtime.

Zusammenfassung

Die vorliegende Arbeit befasst sich mit der Anpassung und Parallelisierung effizienter Skyline Algorithmen für moderne Hauptspeicher-Datenbanksysteme. Außerdem richtet sie den ART Baum für kategorische Skyline Berechnung ein.

Um Datenbank-Anfragen schneller bearbeiten zu können, werden einige der bekannten Skyline Algorithmen so angepasst, dass sie als Baustein direkt in eine Hauptspeicher-Datenbank aufgenommen werden können. Daraufhin werden moderne Parallelisierungstechniken vorgestellt und auf die Skyline Algorithmen angewendet. Die gemessenen Performanzergebnisse bezeugen, dass die vorgeschlagenen Parallelisierungsansätze die Laufzeit der Algorithmen in parallelisierter Umgebung verbessern konnten.

Darüber hinaus wurde der ART Baum dazu verwendet, den neuartigen Algorithmus SARTS zu entwerfen. Der Algorithmus eignet sich besonders gut zur progressiven Berechnung der Skyline in Online-Umgebungen. Im Rahmen der durchgeführten Tests zeigt der Algorithmus sehr solide Laufzeiten auch bei größeren Tupelmengen und verbraucht dabei bis zu 20-mal weniger Speicher als sein direkter Vorgänger ST-S.

Contents

1. Introduction

Assume that you have recently graduated from university and are looking for a new apartment to move in. As a young professional your requirements for the apartment are low rent and short distance to city center. This way, you can pay less at the beginning, and are well-positioned to attend job interviews all over the city. In order to narrow down your options, you want the original set of apartments to be reduced according to your priorities. Hence, you rely on the database, where the apartments are stored, to determine the options that are *interesting* for you. An apartment is defined as interesting when it is not worse than any other apartment in terms of rent and distance to city center. The resulting set of interesting apartments is called the *skyline*. The term skyline was originally chosen in resemblance to a skyline of buildings, meaning those that are not lower than the others in the same horizontal location. Having determined the apartments that are interesting according to your requirements of low rent and short distance to city center, you can now choose the one particular apartment that suits your personal preferences best.

1.1. Motivation

Various online services have seen a significant rise in popularity in the last several years. The application areas range from flight booking to apartment rental. At the roots of any such service there is a massive database, containing relevant information on every item of the underlying dataset. In order to extract the database entires that are required for a particular use case, different operators can be applied to the dataset. One of the most useful filtering operators is the skyline operator. While multiple algorithms have been developed to efficiently compute the skyline of a set of tuples, there are still some optimization aspects that have not yet been covered by extensive research.

This work in specific takes aim at reducing the computation time of some of the most prominent skyline algorithms by parallelizing them for modern CPU architectures. In addition to that, a novel skyline algorithm called SARTS is presented, which exploits the highly efficient memory usage of the ART tree [16]. The algorithm was developed specifically for datasets based on categorical attributes and makes use of some modern optimization approaches in this area.

1.2. Methods

The present work is organized as following. At first, the necessary preliminaries with focus on the skyline operator are given. A brief overview of related work and the existing skyline algorithms takes place, with special emphasis on Naive- and Block-Nested-Loops, Divide-and-Conquer and ST-S algorithms. Hereafter, this paper briefly reviews the main

advantages of the ART tree in the context of databases, and proposes a novel skyline algorithm called SARTS, which makes efficient use of the tree for dominance checks. The new algorithm and its implementation approaches are presented in greater detail. Afterwards, some of the modern parallelization techniques are first explained and then applied to the given skyline algorithms. At this point, two different query pipelining models are introduced: volcano and produce/consume. Then, an evaluation of this work's results takes place. For this purpose, various performance tests are conducted and their outcomes discussed. In the conclusion to this work, the main results and proposals of the paper are once again summarized, and an outlook to possible future work is given.

2. Skyline Computation

Various algorithms have been developed to compute the skyline of a given set of tuples. To date, no multipurpose skyline algorithm exists that would perform equally well in every environment and under any circumstances. Therefore, a number of algorithms have recently emerged that are each tailored to some very specific range of skyline problems. In this chapter, some of the most relevant up-to-date skyline algorithms are briefly introduced and the necessary theoretical foundation to the term skyline is given.

2.1. The Skyline Operator

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 [15]. While the problem has been extensively described and analyzed by Kung et al. as early as in 1975 [15], the term skyline in the context of databases was first introduced by D. Kossmann et al. in 2001 [5], along with the first efficient skyline algorithms.

The skyline of a set of tuples is defined as those tuples that are not dominated by any other tuple in the set. A tuple p dominates a tuple q if p is not worse than q in every given dimension and is better than q in at least one dimension. Or, equally,

$$p \succ q := \nexists i \in N_0^+ . p[A_i] > q[A_i] \wedge \exists j \in N_0^+ . p[A_j] < q[A_j]. \tag{2.1}$$

The definition assumes that the tuple attributes in each dimension have to be minimized. If they have to be maximized instead, then the $>$ and $<$ signs have to be flipped respectively. The notation $p[A_i]$ (resp. $q[A_i]$) stands for p's (resp. q's) i-th attribute value.

To integrate the operation of skyline computation into a database system was first proposed by Kossmann et al. [5] in 2001. The researchers suggested a skyline extension to SQL that would filter out the tuples from the underlying database that are not worse than any other tuple. The SQL syntax proposed to be incorporated into the structure of a query would be similar to:

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

Herein, d_1, ... , d_n are the dimensions of the skyline. The annotations MIN and MAX specify for each dimension whether it has to be minimized or maximized. The authors argue that implementing the functionality of the skyline operator on top of the database is both costly and inefficient, and thus propose the new operator to be integrated directly into the database.

2.2. Database Context

The implementation of the skyline operator inside a database system is influenced by several, sometimes contradictory, criteria. These are, among others, the type of the database, its estimated size, the prevalent data types used, the requirements regarding time and resource usage, as well as the particular use cases in mind at the time of database design.

One of the most common optimization approaches is to reduce the number of needed dominance checks, in order to compute the skyline with a lesser number of comparisons. This way, the CPU cost of the skyline operation can be significantly reduced. Therefore, this optimization technique is most sensible for database systems with generally limited computation resources. One of the algorithms taking this path of optimization is the ST-S algorithm [22], which will be discussed later in greater detail. The main drawback of such algorithms is generally higher memory usage, as the number of dominance checks is most easily reduced by using an extra data structure to "sift" the tuples through it. In most cases, this is some specially fitted form of tree which enables a significant reduction in tuple comparisons. This is why only database systems with a sufficient amount of main memory available at runtime are eligible for this type of optimization. For this reason, in-memory databases seem optimally fitted for such algorithms.

Another criterion to account for is the estimated size of the original dataset. While some algorithms show excellent running times on low numbers of tuples, their runtime grows exponentially with the rising size of the input. Other algorithms, however, are best suited for very large datasets, usually by enabling various parallelization approaches. Large multicore database systems, specially optimized for computation-intensive OLAP queries are the best prerequisite for such algorithms. Some of the approaches are based on the well-known divide-and-conquer principle to achieve this goal.

For database systems with frequent background storage access, a different optimization technique has been developed. In order to conduct the entire computation process within the (often limited) main memory, the tuples for the computation are always loaded block-wise, and the number of I/O accesses to the background storage is thus reduced. Block-Nested-Loops [5] is one the most basic, but nonetheless efficient algorithms for this purpose.

Depending on the data type of the attribute values stored in the tuples, some algorithms might prove not eligible for the particular scenario at all. This is for instance the case with tree-based algorithms where each attribute can only assume categorical values. The reason for this is that each node can maximally have as many child nodes as there are allowed attribute values. While this is usually not a problem for some bounded number of integer values, there are simply too many possible double values between any two range boundaries to create a new child node for each of them.

In the following and throughout this paper, a hybrid main-memory database system, such as HyPer [13], is assumed as the context of this work. It combines the advantages of both OLTP and OLAP databases in one, and thus enables most of the optimization techniques mentioned above. While optimization approaches such as bulk-loading of the tuples into main memory are no longer required, improvements like tree-based dominance tests and also parallelization become possible.

2.3. Related Work

While a lot of work has been done developing high-performing algorithms for skyline computation, many aspects of the field are still not entirely covered. Such aspects include memory efficiency for progressive skyline algorithms, as well as parallelization of originally sequential approaches. The following section gives an overview of some of the most prominent skyline algorithms to date.

2.3.1. Naive-Nested-Loops Algorithm

Arguably the simplest existing skyline algorithm is Naive-Nested-Loops. As mentioned by Kossmann et al, "[t]his is essentially what happens if a Skyline query is implemented on top of a database system"[5].

The main idea behind the algorithm is to compare each tuple of the dataset with every other tuple. This is accomplished by creating two loops, one of them nested in the other, and for each tuple to traverse the entire dataset from beginning to end, in order to check whether the tuple is dominated by any other. If it is not dominated, then it is added to the resulting skyline. While Naive-Nested-Loops is very versatile and can be applied to almost every given dataset, it tends to perform badly in comparison to some of the newer algorithms. A pseudo-code notation of the algorithm is shown in Algorithm 1.

Algorithm 1 Naive Nested-Loops Algorithm

```
1:  Input : Tuple List T
2:  Output : Skyline skyline
3:  for each tuple t ∈ T do
4:      is_not_dominated ←True
5:      for each tuple d ∈ T\{t} do
6:          if dominates(d, t) then
7:              is_not_dominated ←False
8:              Exit inner loop
9:      if is_not_dominated then
10:         Add t to skyline
```

2.3.2. Block-Nested-Loops Algorithm

One of the most basic skyline algorithms, and yet fairly efficient, is the Block-Nested-Loops [5] algorithm (in the following BNL). It takes the concept of Naive-Nested-Loops and extends it by making sure that incomparable tuples are kept in a persisting *window* in main memory. The main idea behind this is to reduce the number of necessary I/O operations when working with traditional DBMS, as disc storage has significantly longer access times than main memory [12].

Whenever a new tuple is inserted into the window, it is iteratively compared to all the other tuples that already find themselves in the window. At this point, one of the following happens:

- If the new tuple is dominated by some other tuple in the window, then it is eliminated and is no longer considered for the skyline.
- If the new tuple dominates one or more of the other tuples in the window, then these tuples are eliminated from the window and are no longer considered for the skyline. The new tuple gets inserted into the window as a new skyline candidate.
- If the new tuple is incomparable with all the other tuples in the window and there is enough space left in it, then it gets inserted. If there is not enough space in the window, then the new tuple is written to the temporary file on disc and will be considered again for further iterations of the algorithm. The last step with the temporary file is not necessary if a main memory database is used, because in this case all the incomparable tuples fit into the window.

All the tuples that are left in the window at the end of each iteration and that have been compared to all the tuples in the temporary file, can be output as part of the skyline. For the original algorithm, the researchers propose to accomplish this by assigning *timestamps* to the tuples [5]. This way, one can determine in which order the tuples have originally been read in. The tuples that have been output before the algorithm terminates are part of the skyline. At this point, all the other tuples from the original dataset have been eliminated.

The pseudo-code of the in-memory version of the BNL algorithm, which is used in this work, is shown in Algorithm 2. It only needs one iteration based on the original set of tuples, and does not require a temporary file because all skyline tuples fit into main memory.

Algorithm 2 Block-Nested-Loops Algorithm (in-memory)

```
1: Input : Tuple List T, Window window
2: Output : Skyline skyline
3: t_0 ← first element of T
4: Add t_0 to window
5: for each tuple t ∈ T\{t_0} do
6:     Add t to window
7:     for each tuple d ∈ window\{t} do
8:         if dominates(t, d) then
9:             Eliminate d from window
10:        if dominates(d, t) then
11:            Eliminate t from window
12:            Exit inner loop
13: Return window as skyline
```

BNL cannot output any of its results before the entire skyline has been computed [11]. This is because even the last of the remaining tuples can eliminate some of the candidate tuples that find themselves in the window. Moreover, while the I/O behavior of this algorithm is significantly superior to that of Naive-Nested-Loops, this advantage gets lost when working with an in-memory database system. The advantage of not having to compare every tuple with every other tuple, however, is still a major improvement. A more detailed description of the BNL algorithm can be found in the original paper [5].

2.3.3. Divide-and-Conquer Algorithm

The Divide-and-Conquer algorithm [5, 15] follows the well-known divide-and-conquer principle, when the original problem is first divided into smaller sub-problems, in order to minimize the workload to solve each of them individually. The algorithm recursively partitions the original dataset into smaller datasets, until each of the subsets only consists of one tuple. For the partitioning, the median m_d of the current dataset is determined for one of the dimensions, let us call it d. This paper assumes the version of the algorithm in which d is initialized as the last of the given dimensions. The first partition P_1 is then filled with tuples that have lesser attribute values in this dimension than the median. The second partition P_2 is filled with tuples that have greater or equal attribute values in this dimension than the median. In order to determine the median, it is sometimes sensible to presort the tuples of the original dataset according to dimension d. The partitioning process is visualized in Fig. 2.1.

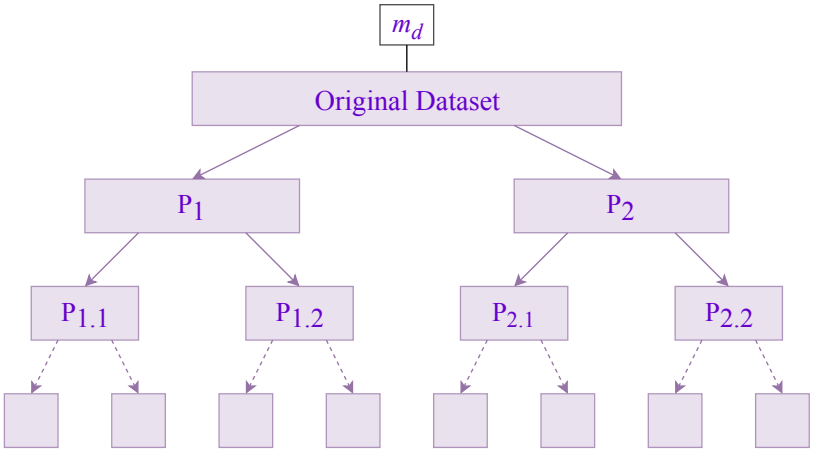


Figure 2.1.: DNC Partitioning Process

When the recursive partitioning is completed, the resulting subsets get merged starting with the smallest ones. Whenever two partitions P_1 and P_2 are merged, the tuples in P_2 that are dominated by any tuple in P_1 are eliminated. The tuples in P_1 cannot be dominated by tuples from P_2, because they are à-priori better than the tuples in P_2 in at least one dimension, which is d. This is due to the division of the original dataset according to the median m_d. Hence, only incomparable tuples are left when the merging is complete. The merge process is illustrated in Fig. 2.2.

Kossmann et al. [5] suggest to further improve the merging process by once again applying the divide-and-conquer principle. For this purpose, each of the two partitions P_1 and P_2 gets partitioned again; this time according to a different dimension ($d-1$ for instance). P_1 and P_2 are now split into $P_{1.1}$ and $P_{1.2}$, and $P_{2.1}$ and $P_{2.2}$, respectively. This has the advantage that not all tuples in P_1 and P_2 have to be compared to each other. Instead, now only the tuples in $P_{1.1}$ and $P_{2.1}$, in $P_{1.2}$ and $P_{2.2}$, and in $P_{1.1}$ and $P_{2.2}$ need to be compared. Thus, the step of comparing the tuples in $P_{1.2}$ to the tuples in $P_{2.1}$ can be omitted. This is because, due to the partitioning with the median m_{d-1}, the tuples in $P_{1.2}$ and in $P_{2.1}$ are incomparable. $P_{1.2}$ is better than $P_{2.1}$ in dimension d, and $P_{2.1}$ is better than $P_{1.2}$ in

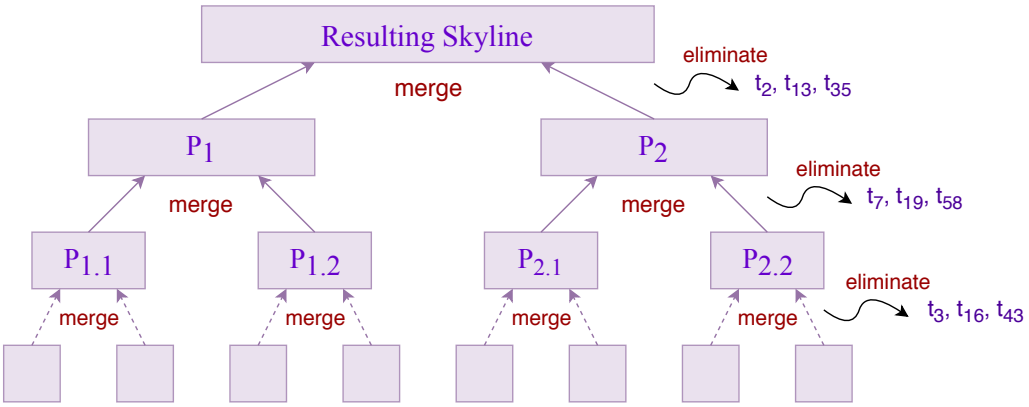


Figure 2.2.: DNC Merging Process

dimension $d-1$. The recursion of the merge function ends as soon as there are no unused dimensions left or the size of the respective subset is either 0 or 1. This advanced merging process is visualized in Fig. 2.3. After the last two subsets have been merged, the resulting set of tuples is the skyline.

The pseudo-code to the DNC algorithm is shown in Algorithm 3. The C++ code, including the helper functions $merge$ and $partition$ can be found in Appendix B to this work.

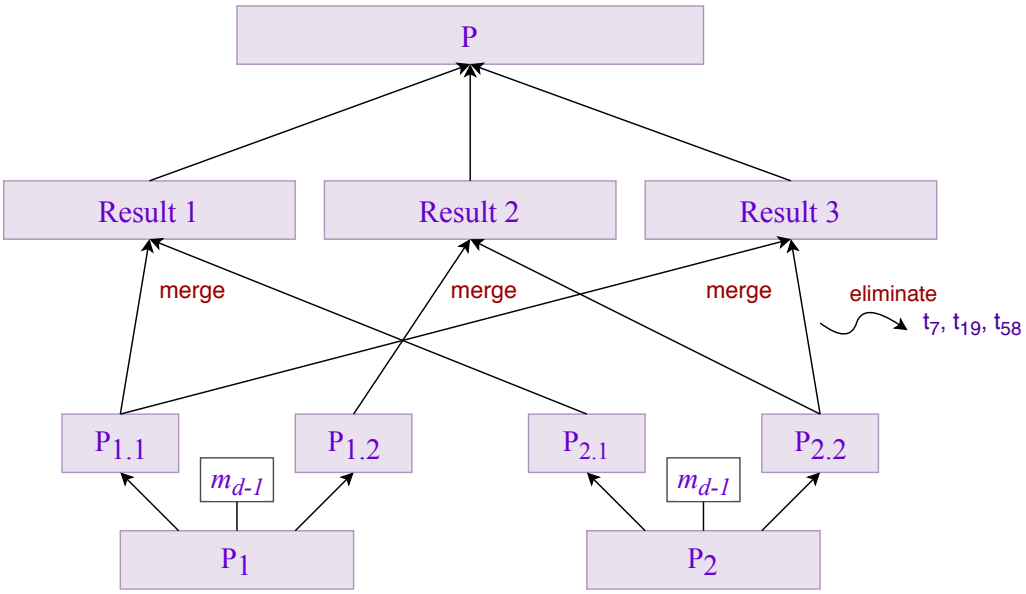


Figure 2.3.: DNC Advanced Merging

For databases with limited main memory capacities the authors further suggest a modification to the basic DNC algorithm called *M-way partitioning*. The main idea is to always partition the dataset in M subsets instead of only two. M should be chosen so that the size of the resulting partitions is small enough to fit into main memory. This approach has to be applied both during the partitioning phase as well as during the merging phase of the algorithm. This sort of modification, however, is not subject of this paper due to the

Algorithm 3 Divide-and-Conquer Algorithm

1: **Input :** Tuple List T, Dimension dim
2: **Output :** Skyline $skyline$
3: **if** $T.size$ = 1 **then return** T
4: $pivot \leftarrow median(T, dim)$
5: $(P_1,\ P_2) = partition(T, dim, pivot)$
6: $S_1 \leftarrow$ Skyline$(P_1,\ dim)$
7: $S_2 \leftarrow$ Skyline$(P_2,\ dim)$
8: $merge\_result = merge(S_1,\ S_2,\ dim)$
9: Add S_1 to $skyline$
10: Add $merge\_result$ to $skyline$
11: **return** $skyline$

context of in-memory databases. The M-way partitioning can be read up in more detail in [5].

Just like the BNL algorithm, Divide-and-Conquer cannot produce its results progressively, meaning that it can only output any of the skyline tuples, once the entire computation of the skyline is completed [11].

2.3.4. Other Algorithms

With BNL and DNC being two of the more basic algorithms, several of the later proposed approaches reuse and extend their ideas, while optimizing the computation in specific areas. In the following, some of the newer skyline algorithms will be briefly introduced.

Bitmap and Index Algorithms

Bitmap [23] and Index [23] are two different algorithms aimed at producing the skyline tuples progressively while keeping response times generally low. The main idea of the Bitmap algorithm is to encode the underlying dataset as a bitmap structure in order to speed up computation [11]. In particular, the fact that bitwise operations are generally fast is exploited. At the beginning of the algorithm every single tuple from the underlying dataset has its attributes mapped into a bit-vector. Later, during computation, all comparison operations are executed on bit-slices derived from the bit-vectors. This enables the algorithm to quickly determine which tuples are dominated by others and which are not.

The Index algorithm takes a different approach at progressive skyline computation and makes use of an index structure, namely a B^+-tree [23]. In addition to that, while it does produce the skyline tuples progressively, it does not do so one-by-one, but does it in blocks instead [11]. First, the algorithm implements a transformation mechanism which maps each tuple into one-dimensional space and stores it in the B^+-tree. During this process, tuples that are mapped to the same value are all stored in the same cluster. Due to the now existing sort order, the algorithm can decide which of the existing tuples are more likely to be part of the skyline, and therefore also behave more dominantly when eliminating other tuples. Also, tuples with common features that were previously mapped to land in

the same cluster, can be checked for being part of the skyline in burst-like batches, which speeds up the performance and enables the algorithm to work progressively.

NN and BBS Algorithms

Another algorithm which is more focused on the "big picture" instead of the entire skyline, is the NN algorithm [14]. It is based on nearest-neighbor-search, hence the name. All the tuples of a dataset can be represented as d-dimensional points in a d-dimensional coordinate system. Assuming that the tuple attributes need to be minimized, the algorithm starts by applying nearest neighbor search to find the point that is closest to the origin. This point is inserted into the skyline right away. At this stage of the algorithm, it partitions the d-dimensional space of the coordinate system into four different partitions. The first one is the partition between the first point and the origin. It does not contain any further skyline points, as there are — by definition of the nearest-neighbor-search — no other points between the first one found and the origin. The second partition cannot contain any skyline points either, because they all have greater attribute values in every dimension than the first point, and are therefore dominated by it. The two remaining partitions can still contain points that are part of the skyline. The NN algorithm is recursively applied to these partitions for as long as there are any that are not empty.

The nearest-neighbor-search component of the algorithm performs particularly well if the original dataset is indexed with a data structure such as the R*-tree. Just like the Bitmap and Index algorithms, the NN algorithm produces the skyline results progressively, i.e. without the need for the entire skyline to be computed before the first results are produced. While the first skyline tuples are output very fast, the rest of the skyline might need a much longer time to be computed [14].

A successor to the original NN algorithm, called BBS (Branch-and-Bound Skyline) [20], has been developed soon after. While it also makes use of the nearest-neighbor-search and the R*-tree, it has some significant improvements over the NN algorithm. One of these is that it only traverses the R*-tree once and thus reduces the number of redundant accesses. The algorithm organizes all of the skyline candidates in a heap and keeps them sorted according to their minimal distance from the origin [11]. There is more information on the BBS algorithm given in [20].

Sorting-Based Algorithms

There exists an entire range of sorting-based algorithms for skyline computation. The somewhat earlier developed SFS algorithm (Sort-Filter-Skyline) [7] reuses the idea of the BNL algorithm and adjusts it so that the tuples are no longer considered in arbitrary order, but are instead presorted first. Then the tuples that are more likely to be part of the skyline are read in first. This way, the skyline can be computed faster, as the tuples inserted into the skyline at the beginning are very likely to be efficient "tuple-killers" and to eliminate most of the dominated tuples very fast. In addition to that, eligible tuples can be output progressively, which was not originally the case with the BNL algorithm. The drawback of presorting the underlying dataset is usually well compensated by the significantly reduced number of comparisons.

A further improvement over the SFS algorithm is its successor SaLSa (Sort and Limit Skyline Algorithm) [2]. Just like SFS, it presorts the original dataset. However, this does not only happen for the reason of reducing the entire number of comparisons during computation. The key idea of SaLSa is to implement a threshold mechanism to stop the algorithm early as soon as all the tuples left are dominated by the tuples already in the skyline. In order to enable this technique, a slightly different sorting function is suggested, which works as following: First, all the tuples are presorted (ascending) according to their smallest attribute value among all dimensions. Then, if two or more tuples compete for the same position in the sorted set, the one with the smaller sum of attribute values across all dimensions wins the tie and gets the lesser position in the set.

Later, during each iteration of the algorithm, the threshold value is updated with respect to the newest tuple that was read in. As soon as the stopping condition is reached, all tuples that are left, are dominated by the threshold, and the algorithm terminates. The stopping mechanism will be discussed in more detail in chapter 2.3.5. Unfortunately, the threshold only reduces the number of tuples processed by the algorithm, but is not as efficient in scenarios with high dimensionality [11].

2.3.5. ST-S Algorithm

One of the newest skyline algorithms to date is the ST-S (Skyline using Tree Sorting-based) algorithm [22]. It is based on SaLSa, which, in its turn, is based on BNL and SFS. Thus, ST-S also belongs to the "family" of sorting-based algorithms. Its main addition to the concept of SaLSa is an algorithm-internal indexing structure, which is similar to a radix tree. This special tree structure is called *N-Tree* within this work. The main purpose of the N-Tree is to execute dominance checks much faster by reducing the overall amount of comparisons between tuples. There are two drawbacks to the tree-based dominance checks. The first one is that ST-S is only suitable for categorical attributes, because continuous attributes (e.g. *double* values) would imply an almost infinite amount of possible categories, and thus far too many children for each of the inner nodes of the tree. The second disadvantage is the somewhat higher memory usage during computation, as the tree structure takes up some of the available space. Using a tree structure to store skyline candidates however means that no separate *window*, like in the original BNL algorithm, is needed. The authors in [22] claim that ST-S significantly outperforms its predecessor SaLSa for categorical attributes in terms of computation time. In chapter 5 of this work, ST-S will be further compared to the Naive-Nested-Loops and SARTS algorithms.

While the authors of the original paper [22] work with binary attribute values (i.e. each attribute is either 0 or 1), they also propose using more categories if needed in a specific use case. This paper makes use of this proposition for generality purposes and implements the tree variant with multiple possible categories. The resulting tree structure used in ST-S is shown in Fig. 2.4. Every inner node, including the root, has an array which can hold as many children as there are possible attribute values. Hence, if all the values in the set $S := \{0,\ 1,\ 2,\ 3,\ 4\}$ can be taken, then each of the tree nodes has an array of size $|S| = 5$ with up to 5 different children. Each path taken from the root to a leaf represents the assignment of attribute values of some particular inserted tuple(s). The IDs of the tuples are always saved in form of arrays within the leaves. In addition to that, each of the inner nodes contains two different score values — a *minScore* and a *maxScore*. MinScore

represents the smallest possible score that can be achieved within leaves reachable from the current node. MaxScore stores the highest possible score, respectively. An illustration of the nodes is given in Fig. 2.5.

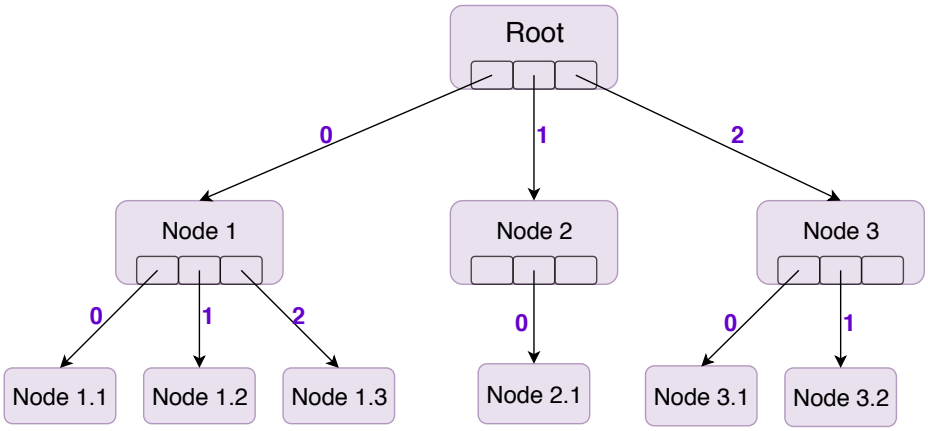


Figure 2.4.: N-Tree for ST-S Algorithm

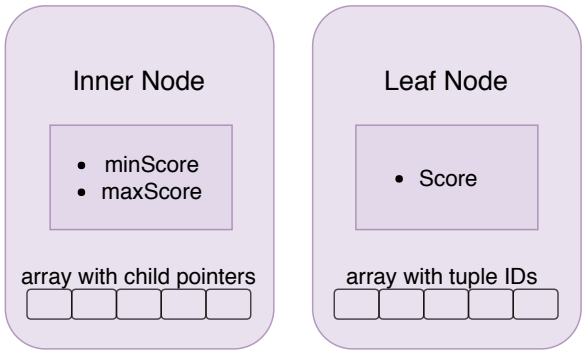


Figure 2.5.: Nodes of the N-Tree

The score of a particular tuple is determined by a scoring function, let us call it F. The only condition that F must fulfill is to not assign a higher score to a tuple that is dominated than to its dominator. The scoring function used in the following is

$$F(t) := \sum_{i=0}^{n-1} (2^{n-i} * t[A_i]), \tag{2.2}$$

with t being the tuple to receive a score, n the length of the tuple, and $t[A_i]$ the i-th attribute of the tuple.

At the beginning, the tuples are sorted with the monotonic function *minC()* (resp. *maxC()* if the attributes have to be maximized). *minC()* is defined as

$$minC(t) := (min_{A_i}\{t[A_i]\}, \sum_{A_i}(t[A_i])). \tag{2.3}$$

It consists of two components as mentioned in chapter 2.3.4: a main comparison attribute, which is the smallest of all attribute values of a tuple, and a tie-breaker being the sum over all attribute values of the same tuple. Suppose the dataset shown in Fig. 2.6 (a). By applying the sorting function *minC* to the dataset, the sorted list of tuples shown in Fig. 2.6 (b) emerges.

Unsorted Dataset				
Apartement	Rent [€]	Distance to Center [m]	min[Ai]	sum[Ai]
A1	700	1000	700	1700
A2	500	3000	500	3500
A3	850	500	500	1350
A4	350	5000	350	5350
A5	600	1500	600	2100

(a)

Sorted Dataset				
Apartement	Rent [€]	Distance to Center [m]	min[Ai]	sum[Ai]
A4	350	5000	350	5350
A3	850	500	500	1350
A2	500	3000	500	3500
A5	600	1500	600	2100
A1	700	1000	700	1700

(b)

Figure 2.6.: Example: Sorting with *minC()*

The pseudo-code to the ST-S algorithm is given in Algorithm 4. It works as following:

1. The tuples are presorted with *minC()* (line 3).
2. t_{stop} is the threshold, and is undefined at the beginning. It is later updated (lines 13-14) with knowledge of some tuples that have been proved to be part of the skyline.
3. The first tuple t_0 from the presorted dataset is always part of the skyline due to the choice of *minC()*. It gets inserted into the tree and is put out as part of the skyline (lines 5-7).
4. In the following loop every tuple t of the sorted dataset, except for the first one t_0, is checked for being dominated by any tuple already in the skyline (line 10). The checks are carried out with the help of the tree, which holds all the skyline tuples to date. The responsible operation *is_dominated* will be introduced in more detail later.
5. If a tuple t is dominated by some other tuple in the skyline, it is no longer considered for the skyline. If it is not dominated, it gets inserted into the tree (line 11), so that it is able to eliminate future tuples that are not part of the skyline.

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 14), and now holds the value of t, until the next update occurs.

7. At the beginning of each iteration of the loop, the stopping condition is checked (line 9), so that the algorithm can stop early if all the tuples left in the dataset are à-priori dominated by the threshold. This functionality has been inherited from the SaLSa algorithm (chapter 2.3.4). If the maximum attribute value of the threshold tuple t_{stop} is less or equal to the minimum attribute value of the current tuple t, and the two tuples are not the same, then thanks to presorting with *minC()* none of the remaining tuples can be part of the skyline. At this point, the algorithm can terminate and the tree with the tuples is no longer needed. If the tuples are the same, however, then the current tuple also has to be checked to be part of the skyline.

Algorithm 4 ST-S Algorithm

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

Both the *insert* and *is_dominated* operations have been slightly modified from the original variant, in order to incorporate multiple attribute categories instead of just two. They also make use of the *minScore* and *maxScore* functionality to speed up dominance checks within the tree.

The pseudo-code to the *insert* operation is shown in Algorithm 5. The proceedings are the following:

1. Depending on the current depth level within the tree, the *minScore* and *maxScore* attributes of the current node are updated (lines 2-7). On level 0 (root level) the smallest possible score is 0, and the highest possible score is calculated using the greatest-possible attribute value for each of the remaining levels. If the current node is not the root, the scores are updated by applying the scoring function in accordance with the current tree level and the attribute values which already occured.

2. If the current level is the deepest level possible, then it means that the algorithm reached a leaf node (line 8). At this point, if some other tuple has already been

inserted in this leaf node, the score of the leaf no longer needs to be calculated. If the current tuple is the first one to get stored in the leaf, then the node's score is updated. In any case, the *tupleID* of the tuple is attached to the *tupleIDs* list of the leaf node (lines 9-10).

3. If the last level has not yet been reached, then the tree has to be further traversed. For this purpose, the *insert* operation is called recursively on the child node at position $t[level]$ with the same tuple t and the *level* increased by one (line 15).

4. If the child at the position $t[level]$ does not yet exist, it is simply created as a new node (lines 13-14) before further traversing the tree.

Algorithm 5 INSERT Operation for ST-S

```
1:  Input : Tuple t, Node node, Level level, Attributes atts
2:  if level = 0 then
3:      node.minScore ← 0
4:      node.maxScore ← Σ_{i=0}^{t.size−1} (2^{t.size−i} * max(atts))
5:  else
6:      node.minScore ← Σ_{i=0}^{level−1} (2^{t.size−i} * t[i])
7:      node.maxScore ← node.minScore + Σ_{i=level}^{t.size−1} (2^{t.size−i} * max(atts))
8:  if level = t.size then
9:      if node.score is None then
10:         node.score ← score(t)
11:     Append t.tupleID to node.tupleIDs
12: else
13:     if node.child[t[level]] not exists then
14:         node.child[t[level]] ← New Node()
15:     insert(t, node.child[t[level]], level + 1)
```

The pseudo-code to the *is_dominated* operation can be seen in Algorithm 6. Starting from the root, the tree is traversed from top to bottom. The usage of the sorting and scoring functions enables the algorithm to only search in those branches of the tree that are not à-priori dominated.

1. In this work the wish to minimize the attribute values is assumed. Therefore, only those children of the current node that have a smaller position than $t[level]$ are further considered in the dominance test. These children are recursively traversed by the *is_dominated* function (lines 7-11).

2. If the current node is *None*, then it has not yet been initialized, and thus also does not have any children that the algorithm could traverse. Hence, there are no paths from the current node that would dominate the given tuple. *False* is returned (line 3).

3. Also, if the score of the given tuple is smaller than the smallest-possible score of the current node, then, due to our choice of the scoring function, the tuple cannot be dominated by any path through this node. *False* is returned (line 3).

4. If by traversing the tree the deepest possible *level* has been reached, the algorithm checks whether the score of the given tuple equals the score of the leaf node. If the scores are equal, then the tuple is obviously not dominated by the other tuples in this leaf node, and *False* is returned (line 5). Otherwise, *True* is returned and the tuple is no longer considered a skyline candidate (line 6).

Algorithm 6 IS_DOMINATED Operation for ST-S

1: **Input :** Tuple t, Node $node$, Level $level$, Score s
2: **Output :** True if t is dominated, otherwise False
3: **if** $node$ is $None$ or $s < node.minScore$ **then return** False
4: **if** $level = t.size$ and $score(t) \neq node.score$ **then return** True
5: **if** $level = t.size$ and $score(t) = node.score$ **then return** False
6: $weight \leftarrow 2^{t.size\ -\ level} * t[level]$
7: **for** each $i \in N_0,\ i < t[level]$ **do**
8: **if** is_dominated(t, $node.child[i]$, $level + 1$, $s + weight$) **then**
9: **return** True
10: **if** is_dominated(t, $node.child[t[level]]$, $level + 1$, s) **then**
11: **return** True
12: **return** False

2.3.6. Classification

The previously introduced skyline algorithms can be organized according to different criteria. A classification with criteria related to the topic of this work is given in Table 2.1.

Family	Algorithms	Based on	Progressive Output	Data Structures
Nested-Loops	Naive NL		yes	
	BNL	Naive NL	no	
Divide-and-Conquer	DNC	Maximum Vector Problem	no	
Sorting-based	SFS	BNL	yes	
	SaLSa	SFS	yes	
	ST-S	SaLSa	yes	N-Tree (radix-like)
Nearest Neighbor	NN	DNC, Nearest Neighbor Search	yes	R*-Tree
	BBS	NN	yes	R*-Tree
	Bitmap		yes	Bitmap
	Index		yes	B$^+$-Tree

Table 2.1.: Classification of Introduced Algorithms

2.3.7. Parallelization Approaches

The focus of this work lies within multicore parallelization with CPUs. In the following, some of the modern approaches to parallelization of skyline algorithms are briefly covered.

Parallelization Criteria

Chester et al. [6] (2015) propose in their work on skyline parallelism the main criteria for developing efficient multicore parallelization approaches. The key suggestions are:

1. When dividing the input data into subsets, the resulting data blocks should be independent, unordered, and about equal-sized. This way, the results of each working unit will not depend on the results of other units and the workload will be close to equally distributed.

2. Data structures that are shared between threads should be either in read-only mode, or only have a few write changes during parallelization. While the coordination on write accesses could be achieved by locking, this often results in a major bottleneck of the otherwise parallel execution. If the number of writes is high and no locks are used, race conditions between threads are almost certain to occur.

3. If synchronization points are used, their location in the code should be chosen very carefully. At these points, there is no parallel processing both during the time of waiting for the single threads to finish their current work, and during the time of sequential synchronization. If the synchronization points were chosen wrongly, the performance of the algorithm will most certainly take a significant hit.

Existing Parallel Algorithms

A range of new algorithms, specifically developed for parallel execution, has recently appeared. Some of the interesting ones to read up on include *pskyline* [21] and *APSkyline* [18] based on the divide-and-conquer principle, *Hybrid* [6] making use of prefiltering, partitioning, and sorting, *SkyAlign* [4], which is one of the few skyline algorithms specifically designed for GPU computation, as well as *MR-GPMRS* [19] designed for computation with the MapReduce paradigm. This paper will not discuss these algorithms in greater detail, as it is more aimed at parallelizing existing sequential algorithms, as well as the novel SARTS algorithm (chapter 3).

3. ART for Skyline

In the following, this work presents a novel skyline algorithm called SARTS, which beholds all of the major advantages of the ST-S algorithm, and further improves it by utilizing the more memory-efficient tree structure ART.

3.1. Adaptive Radix Tree

The Adaptive Radix Tree (ART) [16] is an efficient indexing structure, specifically designed for main-memory databases. In contrast to most traditional in-memory indexing structures, like binary search trees, ART is able to make optimal usage of modern CPU caches, thus enabling both quicker response times and lower space consumption. The main difference to traditional radix trees is the adaptive resizability of the tree's inner nodes, which results in much more compact tree design.

3.1.1. Characteristics

The main drawback of radix trees is wasteful space behavior. The reason for this is that all inner nodes in the radix tree have child arrays of exactly the same size, independently of how many of the child pointers are not null. Therefore, if some of the children do not exist, the excessive memory needed to fully allocate the array of pointers is simply wasted.

The ART tree solves this problem by introducing resizable nodes of four different types: *Node4*, *Node16*, *Node48* and *Node256*. The nodes can have up to 4, 16, 48 and 256 child pointers, respectively. Whenever an inner node has not enough space to insert a new child pointer, it grows to the next bigger type. Similarly, if one of the existing child nodes has been deleted from the tree, then its parent checks if the number of its children is small enough to shrink to the next smaller size.

Each of the nodes consists of two different arrays, which function similarly to a key-value store. The entries in the keys array point to the corresponding entries in the array with child pointers. The structure of the four node types is illustrated in Fig. 3.1 and described in the following:

- **Node4:** The first array is responsible for storing the key bytes in ascending order. The second array holds the pointers to the child nodes. Both arrays are of size 4.
- **Node16:** This node type is the same as Node4, except for that its arrays are of size 16.
- **Node48:** With 48 different entries, sequential search for the right entry in the keys array would take too much time. Therefore, Node48 implements a 256-element array for the key entries. Unlike Node4 and Node16, the key bytes now can be found directly in the index of the array, and the elements of the array are pointers to the entries in the children array.

- **Node256:** This is the largest possible and also the simplest node type in the tree, and can hold up to 256 different entries. It only consists of one array. The key bytes can be found in the index of the array and the entries are child pointers. This way, no indirection is needed in this node type.

In addition to that, each of the nodes possesses a header which holds attributes such as the node type and the number of its chidren. This list of attributes can be adjusted depending on the particular use case of the ART tree. Chapter 3.2 makes use of such adjustments.

Due to the complex structure of the ART nodes, special operations are needed to make use of them. These include *findChild*, *newNode* and *grow*. The operations are explained in greater detail in [16]; the C++ code to each of them can be found in Appendix B to this work.

3.1.2. Further Optimizations

In addition to the core functionality of ART, the authors propose two further optimizations. The first one is called *lazy expansion*. With this technique, inner nodes are only created if there is more than one path to the leaves passing through them. Otherwise, they are simply left out to save space and to reduce access time to the leaves. The second technique is called *path compression*. It results in removing all inner nodes that only have one child node. Both lazy expansion and path compression could be useful when trying to save even more space and to improve lookup times. More details on both techniques are provided in [16].

3.2. SARTS - A Novel Skyline Algorithm

In the following, SARTS (Skyline using ART Sorting-based), a novel skyline algorithm for categorical attributes is presented. It makes use of the core concepts of ST-S (chapter 2.3.5), and further improves it by implementing a more efficient indexing structure for dominance checks — the ART tree.

The interface of the ART tree has been kept similar to that of the N-Tree in ST-S. This enables for a very straightforward integration of the ART tree 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 on the inside, the *is_dominated* operation is almost identical to the one in ST-S. Therefore, for the pseudo-code of SARTS and *is_dominated* it can be referred to Algorithms 4 and 6.

The *insert* operation for SARTS differs from the ST-S variant in that 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 7.

The main difference within the *is_dominated* operation is, similarly to *insert*, the usage of the *findChild()* function any time the correct child for further traversing of the tree needs to be found.

In addition to that, just like the nodes of the N-Tree, the inner nodes of the ART tree have to be extended by a *minScore* and a *maxScore*, and the leaf nodes by the *score* attribute and

Algorithm 7 INSERT Operation for SARTS

Input : Tuple t, Node $parent$ Node $current$, Level $level$, Attributes $atts$
if $level$ = 0 **then**
    $node.minScore \leftarrow 0$
    $node.maxScore \leftarrow \sum_{i=0}^{t.size-1}(2^{t.size-i} * max(atts))$
else
    $node.minScore \leftarrow \sum_{i=0}^{level-1}(2^{t.size-i} * t[i])$
    $node.maxScore \leftarrow node.minScore + \sum_{i=level}^{t.size-1}(2^{t.size-i} * max(atts))$
if $level$ = $t.size$ **then**
    if $node.score$ is $None$ **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 $current.size$ = 4 **or** $current.size$ = 16 **or** $current.size$ = 48 **then**
            $grow(parent, current, t[level - 1])$
        $child \leftarrow newChild(current, t[level])$
    insert(t, $current$, $child$, $level + 1$)

an array of *tupleIDs*. This enables faster traversing of the tree during dominance checks by skipping tree regions that cannot dominate the current tuple.

The ART version used in this paper is the basic one for simplicity purposes. It does not incorporate the two optimization approaches *path compression* and *lazy expansion*.

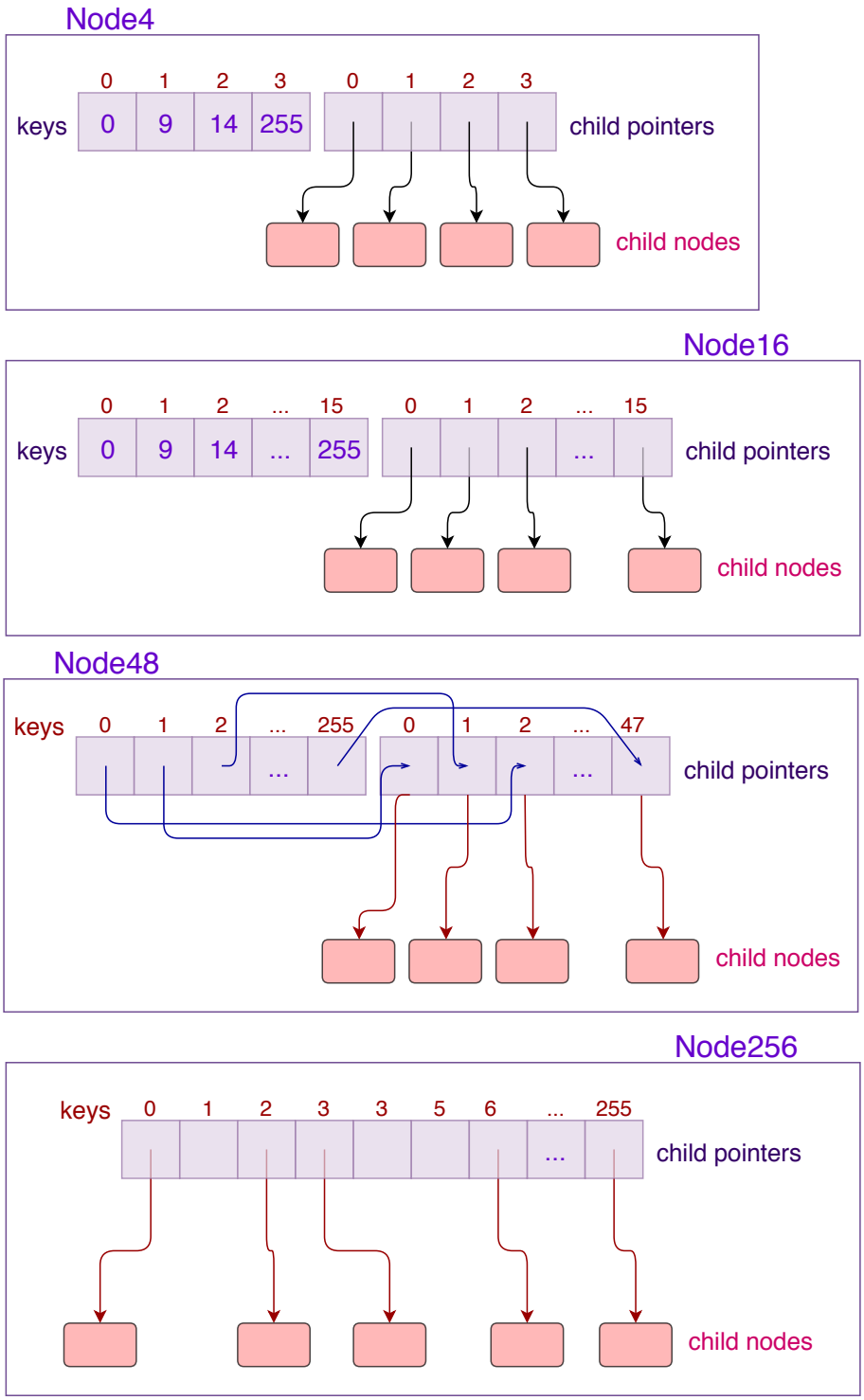


Figure 3.1.: Node Types of the ART Tree

4. Parallelization

The following chapter presents the parallelization approaches suggested in this work. All the algorithms were implemented and parallelized in C++, mainly due to its efficiency in terms of speed and memory management.

4.1. Methods and Frameworks

There are two different implementation models used for the algorithms. The Block-Nested-Loops algorithm is implemented in two versions: one of them makes use of the Volcano model, the other of the produce/consume concept. This was done in order to be able to compare the two models in terms of their performance. The rest of the algorithms were implemented with the produce/consume concept only. Both models are briefly covered in the following.

4.1.1. Volcano Model

The Volcano Model, sometimes also called Iterator Model, is an evaluation strategy of database queries [1]. A standard database query is evaluated by passing a stream of tuples through several database operators, with each of these operators doing their task (e.g. join, filter, skyline, etc.). Whenever an operator calls *next()* on its predecessor in the evaluation pipeline, the preceding operator has two options:

- If the next tuple is ready to be delivered, it is simply returned.
- If the next tuple has not yet been produced, then the predecessor either produces it itself, or, if necessary, first calls *next()* on its own preceding operator.

The query pipelining process for the Volcano Model is visualized in Fig. 4.1. In some cases, the response time to a *next()* call can be fairly long due to the requested tuple(s) not being ready for delivery yet.

4.1.2. Produce/Consume

The produce/consume concept is slightly different. Each of the database operators has one child operator and one parent operator. All operators implement the common *operator* interface, and thus all possess a child, a parent, as well as the two functions *produce* and *consume*. Whenever an operator needs its child to produce tuples, it calls *produce* on the child. This only happens once, and therefore prevents the operator from continuously calling *next* as it is the case in the Volcano model. After the child received the produce instruction, it first calls *produce* on its own child. As soon as all the tuples have been

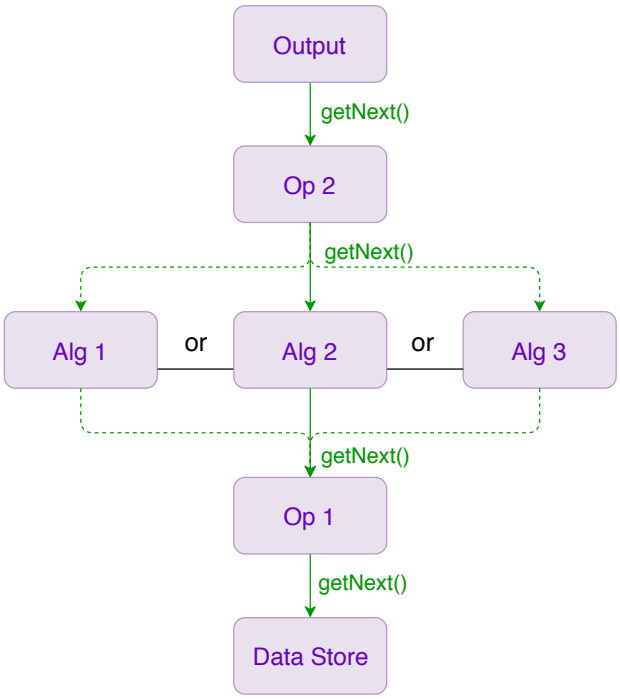


Figure 4.1.: Query Pipelining with Volcano Model

received and processed, the child repeatedly calls *consume* on its parent and "feeds" the tuples to it one by one. The entire process is shown in Fig. 4.2.

It is important to notice that the produce/consume concept used in this work does **not** include code generation. Instead, the concept is used "as-is". The code is normally compiled and still exists at runtime. A pseudo-code implementation of the produce/consume concept is shown in Algorithm 8. It features an integration of a sample skyline algorithm into a database system.

Algorithm 8 Integration of a Skyline Operator into a Database System

```
Input : Parent Operator parent, Child Operator child
T ← New List()
function CONSUME(Tuple t)
    Add t to T
function PRODUCE
    child.produce()
    computeSkyline()
function COMPUTESKYLINE
    for each tuple t ∈ T do
        if t is not dominated then
            parent.consume(t)
```

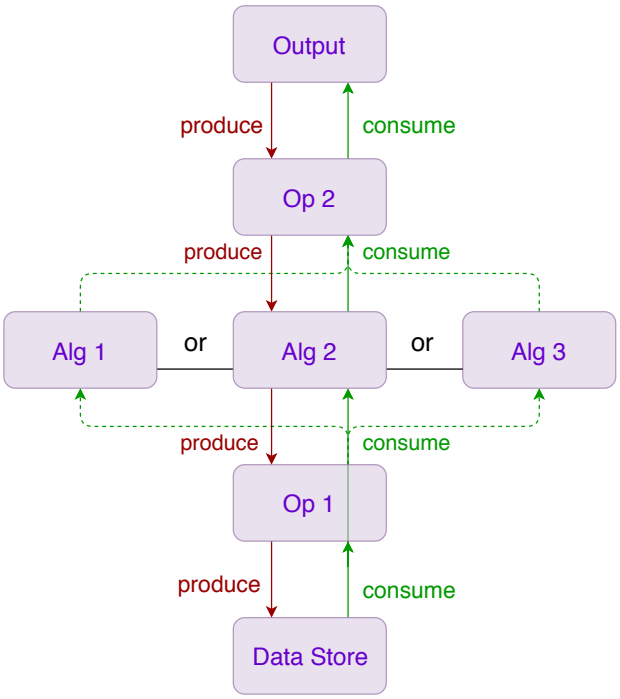


Figure 4.2.: Query Pipelining with Produce/Consume Model

4.1.3. Parallelization Frameworks

The two main parallelization frameworks utilized in this work are Intel TBB's *parallel_for* [8] construct for C++, as well as the C++ *std::future* library. While the OpenMP [3] API is frequently listed as the main alternative to parallel_for, the latter seemed slightly more suitable for the purposes of this paper.

4.2. Naive-Nested-Loops and Block-Nested-Loops

The main idea when parallelizing 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 would be running sequentially, thus reducing the benefit of parallelizing the code in the first place.

The pseudo-code notation of the parallelized version of Naive-Nested-Loops is given in Algorithm 9. As global variables and helper functions within the same class, such as $dominates()$ and T, cannot by default be "seen" from inside the parallelized loop, they are captured and handed over to *parallel_for* (line 3). The code within the outer loop works similarly to the sequential version, but with one difference. The command *Exit inner loop*[1], which was previously used in the sequential version of the algorithm (Algorithm 1), is no longer available within a *parallel_for* loop. This is because it can be problematic for all the threads to coordinate their actions to such an extent that they can all simultaneously break

[1] In C++ this is a ***break*** statement.

Algorithm 9 Parallelized Naive-Nested-Loops Algorithm

```
1: Input : Tuple List T
2: Output : Skyline skyline
3: parallel_for each tuple t ∈ T with captured T, skyline, dominates() do
4:     is_not_dominated ←True
5:     for each tuple d ∈ T\{t} and as long as is_not_dominated do
6:         if dominates(d, t) then
7:             is_not_dominated ←False
8:     if is_not_dominated then
9:         Add t to skyline
```

their execution without causing any problems. Therefore, *parallel_for* does not allow *break* statements. The command has been replaced with the *is_not_dominated* flag as a stopping condition in the inner loop (line 5). The algorithm stops as soon as all threads finished their work and eligible tuples find themselves in the $skyline$.

Several attempts were made to parallelize the Block-Nested-Loops algorithm in this work. The most promising one was to first modify Naive-Nested-Loops so that it would use an atomic bitmap shared between the threads. This bitmap would act similarly to a "gravestone" and would store the indexes of the tuples that are no longer eligible for the skyline. This way, each of the threads would skip any tuples that were already eliminated by other threads, which would significantly reduce its workload. Because this approach uses Naive-Nested-Loops at its base, it is quite easily parallelizable in contrast to the original BNL algorithm. Unfortunately though, it did not produce the expected improvements in running time in comparison to the sequential BNL algorithm. Therefore, no separate parallelized version of BNL is used in the Evaluation chapter (5) of this work. Instead, two different variants of BNL, one of them using the Volcano model, the other using the Produce/Consume concept will be part of the Evaluation.

4.3. Divide-and-Conquer

Two different parallelization techniques were applied to the Divide-and-Conquer algorithm. The first one is another construct from TBB's *parallel* "family" called *parallel_sort* [9]. 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 datasets. *parallel_sort* is applied in two places within the DNC algorithm:

1. It is used within a separate function whose task is to determine the median of a set of tuples. For this purpose, the tuples are first sorted with *parallel_sort* according to the given dimension. After that, the median of the dataset is taken as the element finding itself exactly in the middle of the sorted set.

2. It is used for determining the minimum of a subset of tuples. This functionality is applied when the tuples of the dataset only have two dimensions. In this case, the

skyline can be computed by finding the minimum of the first subset and comparing it to all elements of the second subset.

The second parallelization technique used in DNC is creating two asynchronous threads for the recursive calls of the *mergeBasic* function. As there are three recursive calls to *mergeBasic* from inside the function, one might be thinking of executing all three of them in parallel. Unfortunately, the third call of *mergeBasic* receives the result of the second one as parameter. Therefore, the third call can only be executed as soon as the second one has been completed. As the two parallelized threads run asynchronously, they are not by default waited for by the rest of the code. Because of this, the *future* library offers the method *get* that can be called on each of the parallel thread instances. The function *get* blocks the execution of the program until the result of the corresponding thread is available. As soon as the thread has finished, the result of its computation is returned by *get* and can be used for further operations.

4.4. SARTS and ST-S

Parallelizing the ST-S and SARTS algorithms results for both of them in almost identical implementation. This is because the main differences between the two algorithms are hidden within the respective tree structures. As the interfaces of both trees are technically the same, the algorithms were also parallelized with the same approach.

The main idea is to divide the original dataset into as many partitions as there are threads on the machine. For each of these partitions the skyline of its tuples is computed independently in a separate thread. Every thread receives its own tree structure to store the tuples that are part of the skyline and to perform efficient 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 skyline. The skylines of all partitions combined are in the utmost cases much smaller than the original dataset. Therefore, the final merge does not take as much time as computing the entire skyline from scratch.

In addition to the main parallelization approach, presorting the tuples before the actual algorithm begins also happens in parallel. For this purpose, the same *parallel_sort* construct is used as in the parallelized version of DNC. As the original dataset tends to be very large in real-world applications, sorting it in parallel leads to a very significant "efficiency boost".

The pseudo-code to the parallelized version of SARTS/ST-S is given in Algorithm 10.

1. At first, the tree structures $subtrees$ are initialized for each of the threads to run in parallel on their subset of tuples (lines 3-5).

2. Then, the subsets are filled with the respective range of tuples from the original dataset (lines 7-8). In the given pseudo-code, the dataset is simply partitioned into sequential ranges of equal size. There are exactly as many partitions as there are threads.

3. For each of the subsets, a new asynchronous thread is started and receives its thread ID, the corresponding subset and an empty tree, for instance ART, as parameters

Algorithm 10 Parallelized SARTS/ST-S Algorithm

1: **Input :** Number of Threads $n\_threads$, Tuple List T, Tree $tree$, Subset Results $sub\_results$
2: **Output :** Skyline $skyline$
3: $subtrees \leftarrow$ undefined
4: **for** each $i \in N_0, i < n\_threads$ **do**
5: $\quad subtrees[i] =$ **new** Tree()
6: $subsets \leftarrow$ undefined
7: **for** each $i \in N_0, i < n\_threads$ **do**
8: $\quad subsets[i] \leftarrow T[i * T.size \,/\, n\_threads]$ until $T[(i+1) * T.size \,/\, n\_threads - 1]$
9: $\quad$ Start **new** asynchronous Thread() for compute_skyline_subset(i, $subsets[i]$, $subtrees[i]$)
10: **for** each asynchronous Thread t **do**
11: $\quad$ Wait for t to finish
12: $skyline \leftarrow$ Skyline($sub\_results$, $tree$) // Compute Skyline either with SARTS or ST-S

(line 9).

4. After all the threads have been started, they compute their sub-skylines in parallel and store the resulting candidate tuples into a common array called $sub\_results$.

5. As soon as the threads have finished their work (lines 10-11), their results are merged into the final skyline. This is done by applying the sequential version of SARTS/ST-S as presented in chapter 3.2 (resp. 2.3.5 for ST-S) to $sub\_results$.

The pseudo-code to the *compute_skyline_subset* operation for parallelized SARTS/ST-S is given in Algorithm 11.

Algorithm 11 COMPUTE_SKYLINE_SUBSET for Parallelized SARTS/ST-S

1: **Input :** Thread Number tid, Number of Threads $n\_threads$, Tuple Subset T, Tree sub_tree, Subset Results $sub\_results$
2: Sort T in-place using a monotonic function *minC()*
3: $t_0 \leftarrow$ first element of T
4: $t_{stop} \leftarrow t_0$
5: insert(t_0, $sub\_tree.root$, 0)
6: Add t_0 to $sub\_results$ at position $tid * (sub\_results.size \,/\, n\_threads)$
7: **for** each tuple $t \in T \backslash \{t_0\}$ **do**
8: $\quad$ **if** $max(t_{stop}) \leq min(t)$ and $t_{stop} \neq t$ **then return**
9: $\quad$ **if not** is_dominated(t, $sub\_tree.root$, 0, $score(t)$) **then**
10: $\quad\quad$ insert(t, $sub\_tree.root$, 0)
11: $\quad\quad$ Add t to $sub\_results$ at position $tid * (sub\_results.size \,/\, n\_threads) + t.index$
12: $\quad\quad$ **if** $max(t) < max(t_{stop})$ **then**
13: $\quad\quad\quad t_{stop} \leftarrow t$

Instead of computing the skyline on the entire set of tuples, the function only takes a

subset of the original data as argument. It also receives the ID of the thread that it is assigned to, the overall number of threads running in parallel, as well as a reference to an array where the skyline results of all subsets will be stored.

The skyline is computed similarly to the non-parallelized version with one major difference. Whenever a tuple that is definitely part of the skyline is stored, it is not just appended to some list of skyline tuples. Instead, it is stored into the common $sub\_results$ array which all skyline threads share access to. In order to prevent any sort of race condition during *write* operations between the threads, each of the threads writes its results strictly into the part of the array assigned to this particular thread (lines 6, 11).

5. Evaluation

The following chapter first explains the setup behind the conducted tests. After that, the performance of the parallelized algorithms is measured and compared to their respective non-parallelized versions. The algorithms in test are Naive-Nested-Loops, Block-Nested-Loops, Divide-and-Conquer, ST-S and SARTS.

5.1. Setup

All the tests were conducted on a Lenovo E550 machine with the following technical specification:

- **CPU:** Dual core Intel Core i7-5500U with a 4096 KB cache
- **RAM:** 8 GB DDR3L 1600 MHz
- **Graphics:** Intel HD Graphics 5500
- **Operating System:** Linux Mint 18.2 Sonya 64-bit

Before testing, the machine was newly rebooted in order to achieve a completely "clean run" for the tests. Any wired and wireless networking (WiFi, Bluetooth, etc.) has been switched off. Only the programs necessary for the tests have been started, including a Linux terminal and the C++ program running and measuring the algorithms. Any processes not related to the basic operating system maintenance with noticeable CPU or main memory usage have been killed.

Except for the Volcano version of BNL, all the algorithms were tested utilizing the produce/consume concept (section 4.1). A sample structure of the testing program is shown in Fig. 5.1. At the beginning, the *main* class of the program initializes the test with user input, which is the number of tuples n and the number of dimensions dim. Then, also within *main*, the objects required for the program to run, are instantiated. These include a random tuple *generator*, the algorithms that need to be tested, as well as an *output* object for each of the algorithms, which receives and prints the respective skyline. The *output* object initializes an algorithm by calling *produce()* on it and waiting for it to produce results. The algorithm, in its turn, receives its input by calling *produce()* on the *generator*. In order to simulate non-recurring database queries, the *generator* creates n random tuples with dim different dimensions each, using a random number generator. The tuples are then "pushed" through the pipeline from the *generator*, bypassing and being filtered by the particular algorithm, and reaching the *output* object in the end. The *operator* interface implemented by every class in the program — except for the main — enables for an easy exchangeability of the algorithms.

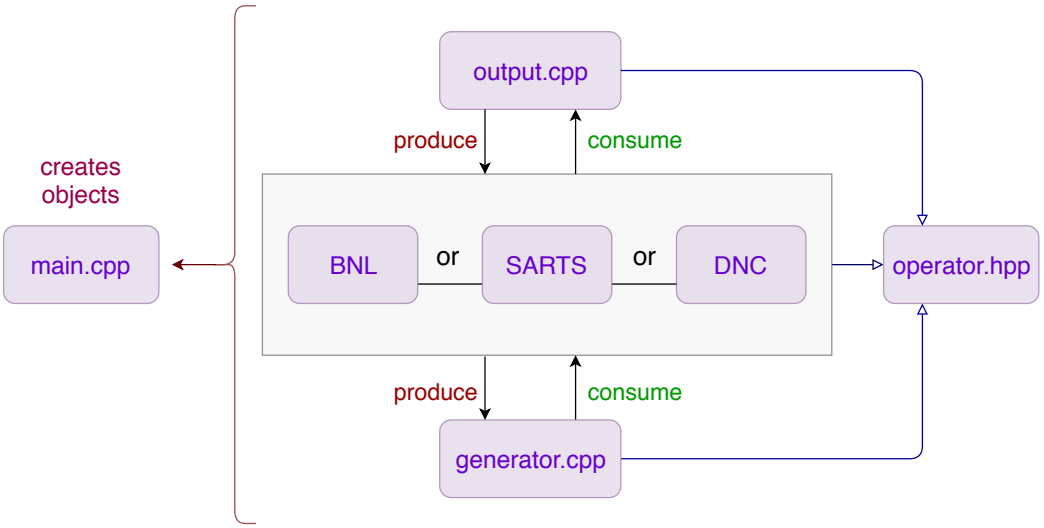


Figure 5.1.: Test Setup with Sample Algorithms

While tree-based skyline algorithms, such as ST-S and SARTS, can only realistically work with categorical attributes, other general-purpose algorithms are not restricted to categorical attribute domains. Therefore, all 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.

The memory usage of N-Tree and ART was captured with the tool Valgrind Massif. It is a heap profiler capable of determining how much heap memory the program is using at particular points in time.

5.2. Results

In the following, the results of the conducted measurements are presented.

5.2.1. Computation Time

Volcano and Produce/Consume

To begin with, the two query pipelining approaches Volcano and produce/consume were compared to each other. For this purpose, two different variants of the Block-Nested-Loops algorithm were implemented. Both versions can be found in the Appendix B to this work. The results of the comparison (Fig. 5.2) show that up to a threshold of about 1 million tuples the running time of both BNL Volcano and BNL p/c is virtually the same. With the number of tuples further rising, the Volcano version consistently outperforms the produce/consume version. Therefore, the conclusion can be drawn that while the performance of Volcano is only slightly better than that of produce/consume, Volcano model is still the way to go for query pipelining. However, one should keep in mind that the produce/consume concept used in this work is not the original one, which would

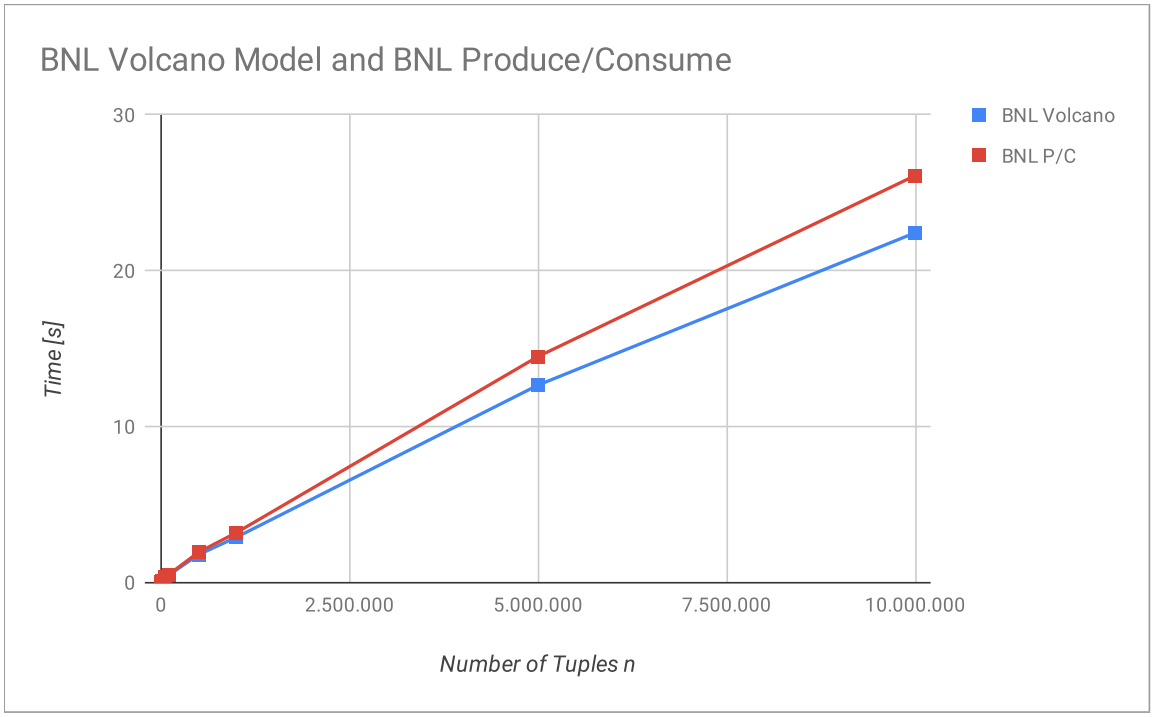


Figure 5.2.: Comparison of Volcano Model and Produce/Consume Concept by Running Time

also include code generation, as explained in section 4.1. The underlying numbers to this comparison are given in form of a table in Appendix A.

Non-Progressive Algorithms

Non-progressive skyline algorithms included in this work are Block-Nested-Loops and Divide-and-Conquer. In all three of the conducted tests, BNL proves that it scales significantly better than DNC. It shows overall better performance with rising numbers of tuples (Fig. 5.3), dimensions (Fig. 5.4), and also threads (Fig. 5.5). The same applies to the parallelized version of DNC. While it does perform better than the sequential version of DNC, it still cannot keep up with BNL in any of the tests.

Herewith, the results are similar to the ones produced in the original paper [5], where BNL and DNC were first introduced. In their evaluation, the authors showed that basic BNL significantly outperforms basic DNC for larger n. It is remarkable that the results measured within this work are significantly better than the results in [5] for both BNL and DNC. This is, however, not surprising, regarding the lack of I/O operations as well as a much more powerful testing machine this time.

On the large scale of comparison between BNL and DNC, the two versions of BNL, namely Volcano style and Produce/Consume perform virtually the same in all three tests. There is, however, a slight difference between the two versions, as previously shown in this section. The concrete numbers measured within the tests can be found in Appendix A.

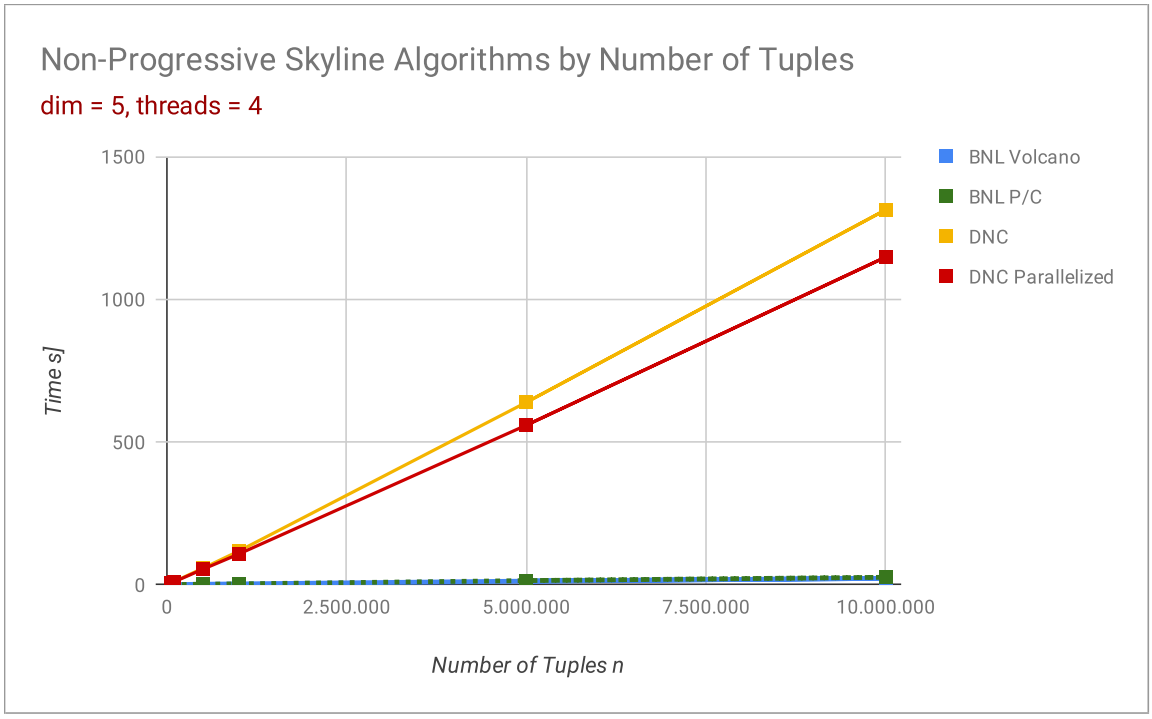


Figure 5.3.: Running Time of Non-Progressive Algorithms by Number of Tuples

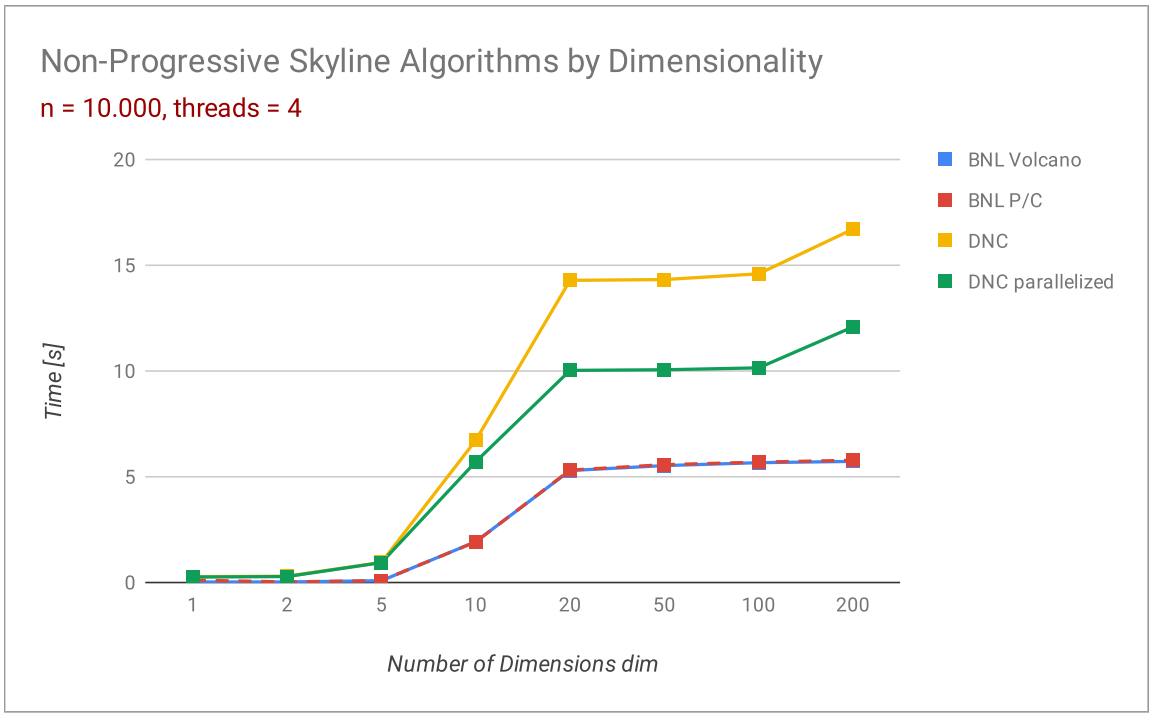


Figure 5.4.: Running Time of Non-Progressive Algorithms by Dimensionality

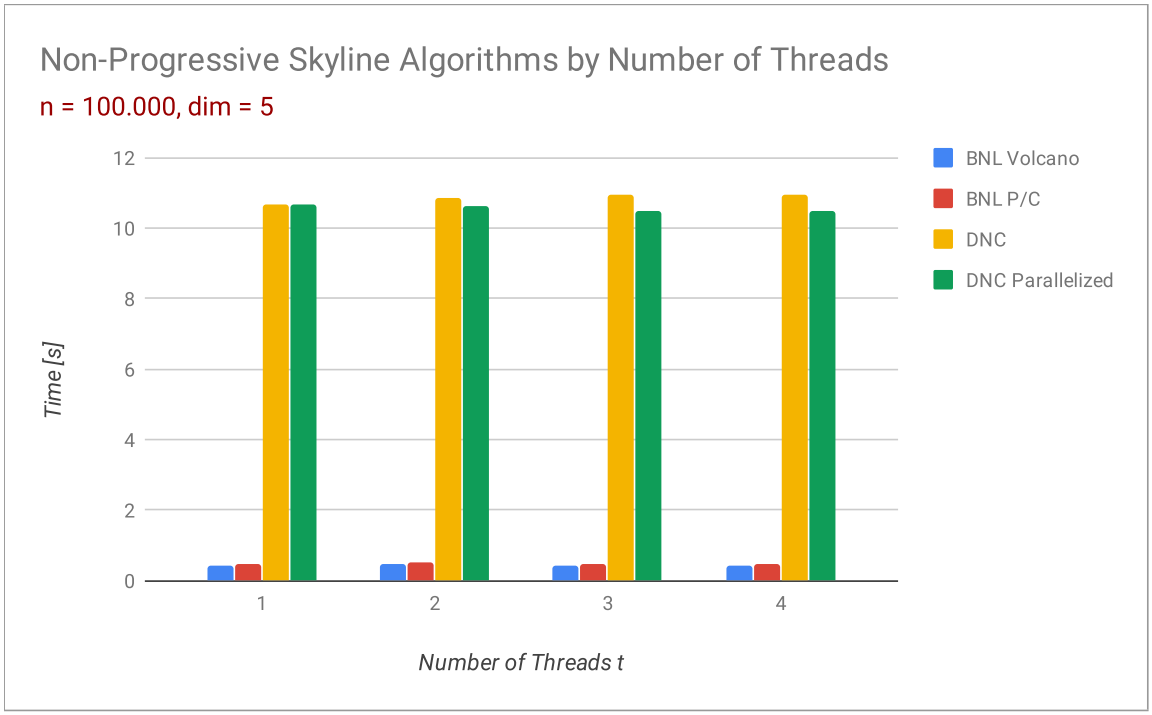


Figure 5.5.: Running Time of Non-Progressive Algorithms by Number of Threads

Progressive Algorithms

Despite being the arguably simplest skyline algorithm to date, Naive-Nested-Loops can be both progressive and parallelizable if implemented correctly. This is what this work makes use of in order to make it comparable with the two newer algorithms ST-S and SARTS.

As expected, for a rising number of tuples n both ST-S and SARTS perform extremely well. As shown in Fig. 5.6, they significantly outperform Naive-Nested-Loops for both middle-range and large n. This is not surprising, considering that ST-S and SARTS were both specifically developed for large categorical datasets. It is due to the efficient 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 dataset (section 3.1.1).

A more close-up comparison of ST-S and SARTS is shown in Fig. 5.7. It can be seen that the parallelized versions of both algorithms start outperforming their sequential counterparts at a dataset size of about 100,000 tuples. While the ST-S and SARTS algorithms generally go "hand-in-hand" for both sequential and parallelized versions, the ST-S algorithm is slightly better than SARTS for large n. This advantage, however, seems of lesser significance in comparison to the memory usage gains of the SARTS algorithm, which will be presented in the next section of this work.

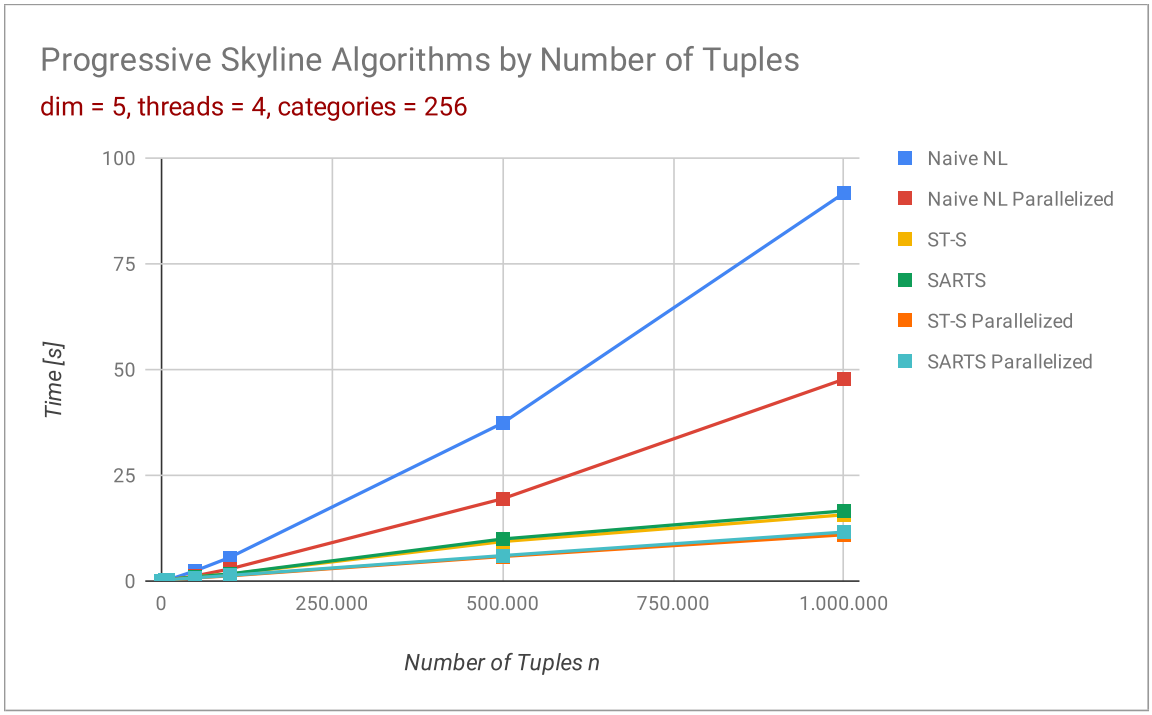


Figure 5.6.: Running Time of Progressive Algorithms by Number of Tuples

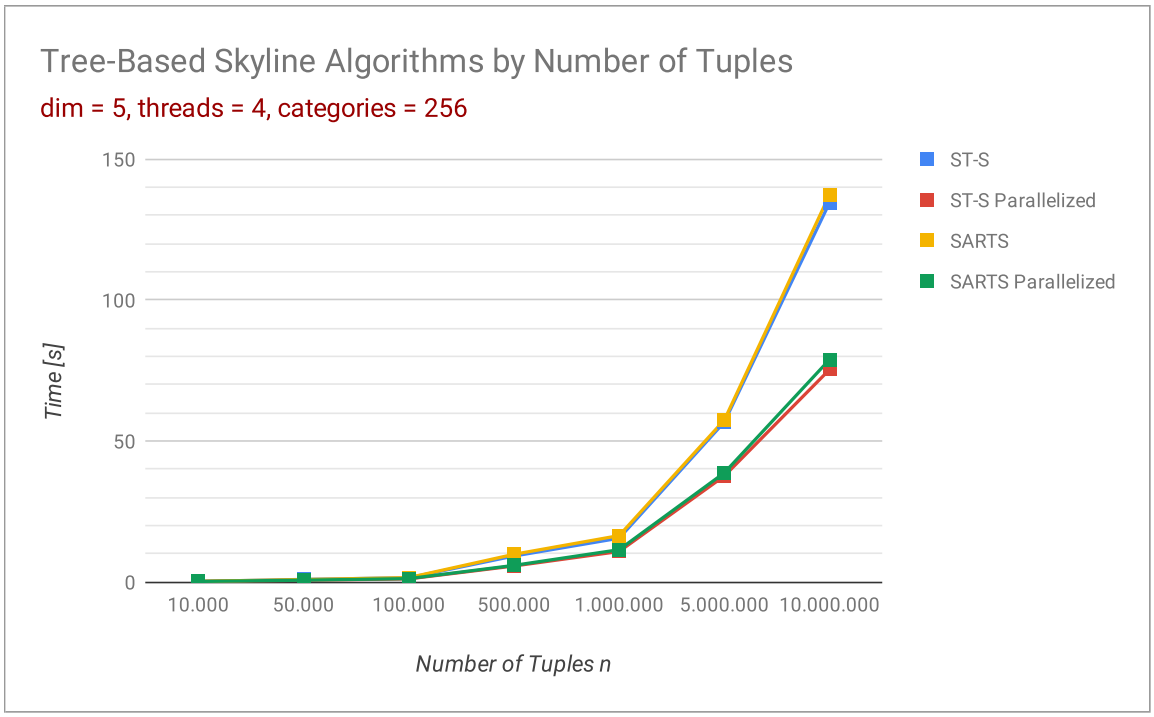


Figure 5.7.: Running Time of Tree-Based Progressive Algorithms by Number of Tuples

When looking at the results of scaling with dimensionality, a somewhat different picture emerges (Fig. 5.8). In this test, Naive-Nested-Loops significantly outperforms both parallelized and sequential versions of ST-S and SARTS. The reason for this is that radix-based tree structures — which N-Tree and ART both are — generally scale badly with longer keys. This is the trade-off they have to take for very efficient scaling with the number of elements inserted. The longer the keys of the dataset 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 dataset have, the less efficient tree-based dominance checks become. This is exactly what can be seen in Fig. 5.8. It also explains why parallelized versions of the algorithms perform worse than the sequential ones. The parallelization of ST-S and SARTS is based on multiple sub-trees used to compute subset skylines before merging them into one final skyline. Thus, the sub-trees also underlie the "curse of dimensionality".

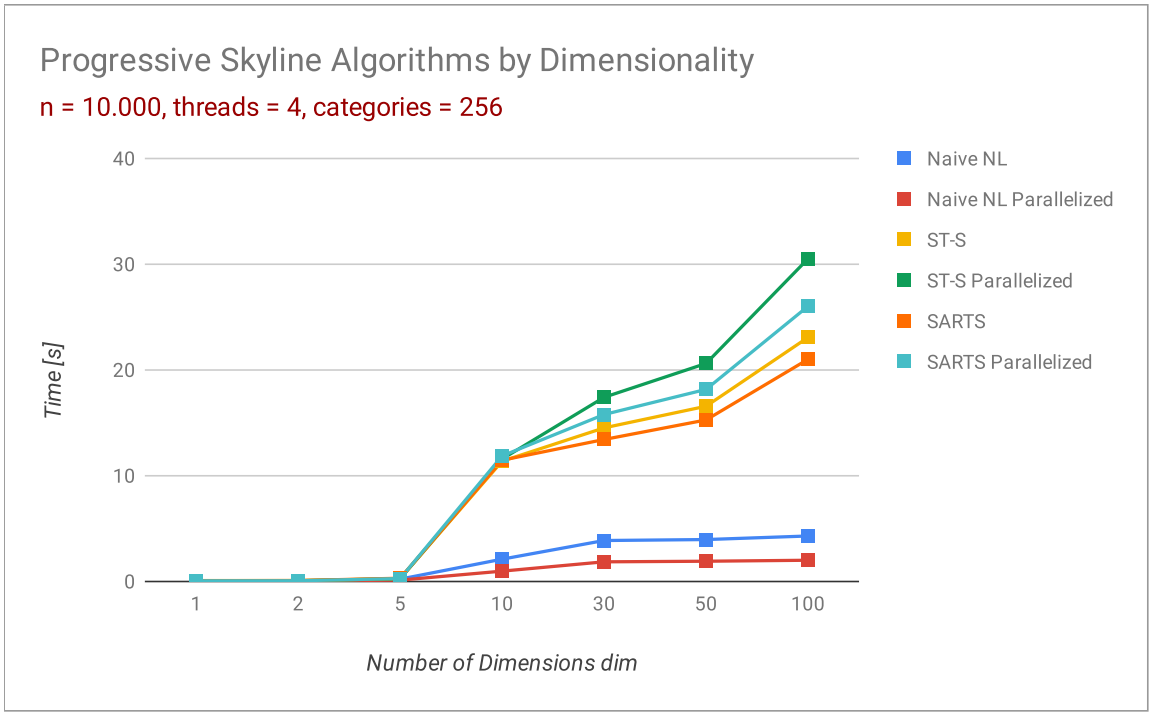


Figure 5.8.: Running Time of Progressive Algorithms by Dimensionality

The comparison of progressive algorithms depending on the number of threads available shows that ST-S and SARTS outperform Naive-Nested-Loops for both parallelized and non-parallelized versions. As expected, the time consumption of the parallelized algorithms gets less with a rising number of threads. This means that the parallelization approaches taken in chapter 4 produce the desirable performance improvements. While on 1 and 2 threads ST-S and SARTS are a bit faster in their sequential version, on 3 cores and more the parallelized versions show better performance. For the Naive-Nested-Loops algorithm this already happens from 2 threads and up. The slightly worse performance of

tree-based algorithms on 1 and 2 threads is likely due to the additional overhead required to partition the original dataset, to create a sub-tree for each of the threads, as well as to set up the threads themselves. The performance improvements are expected to become more noticeable if the algorithms run on more than 4 threads, like for instance 8 or 16.

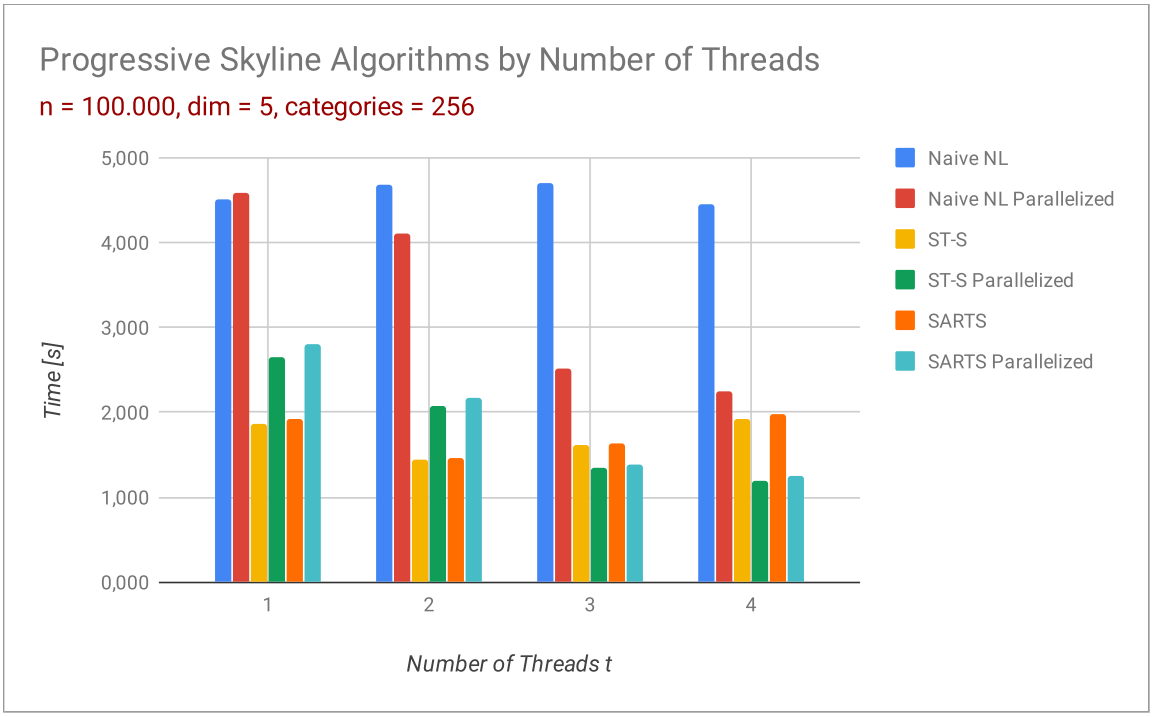


Figure 5.9.: Running Time of Progressive Algorithms by Number of Threads

5.2.2. Memory Usage

Within the last two tests, the main memory usage of the ART tree was compared to that of the N-Tree. Fig. 5.10 shows that the ART tree has significantly lower space consumption than the N-Tree for every n in test. While for n = 10 ART uses about 22-times less memory than N-Tree, even for large n, such as 10^6, it still uses appx. 20-times less space than N-Tree. The measured numbers can be found in Appendix A to this work.

A similar picture can be seen when looking at memory consumption scaling with dimensionality in Fig. 5.11. While for low numbers of dim, the ART tree already performs better than the N-Tree, it generally scales much more efficiently with high dimensionality. At the point of dim = 50 the ART uses only around $\frac{1}{38}$ of the memory that N-Tree does. Thus, it can be concluded that the SARTS algorithm is significantly more memory-efficient than ST-S due to the usage of ART.

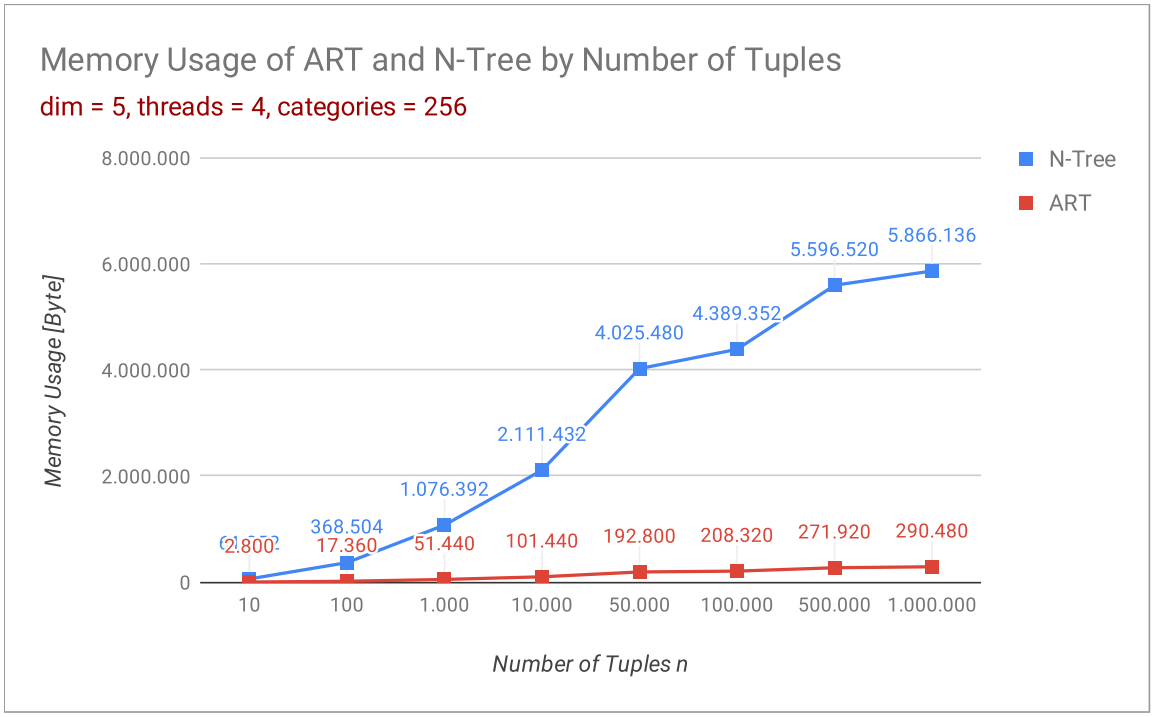


Figure 5.10.: Memory Usage of ART and N-Tree by Number of Tuples

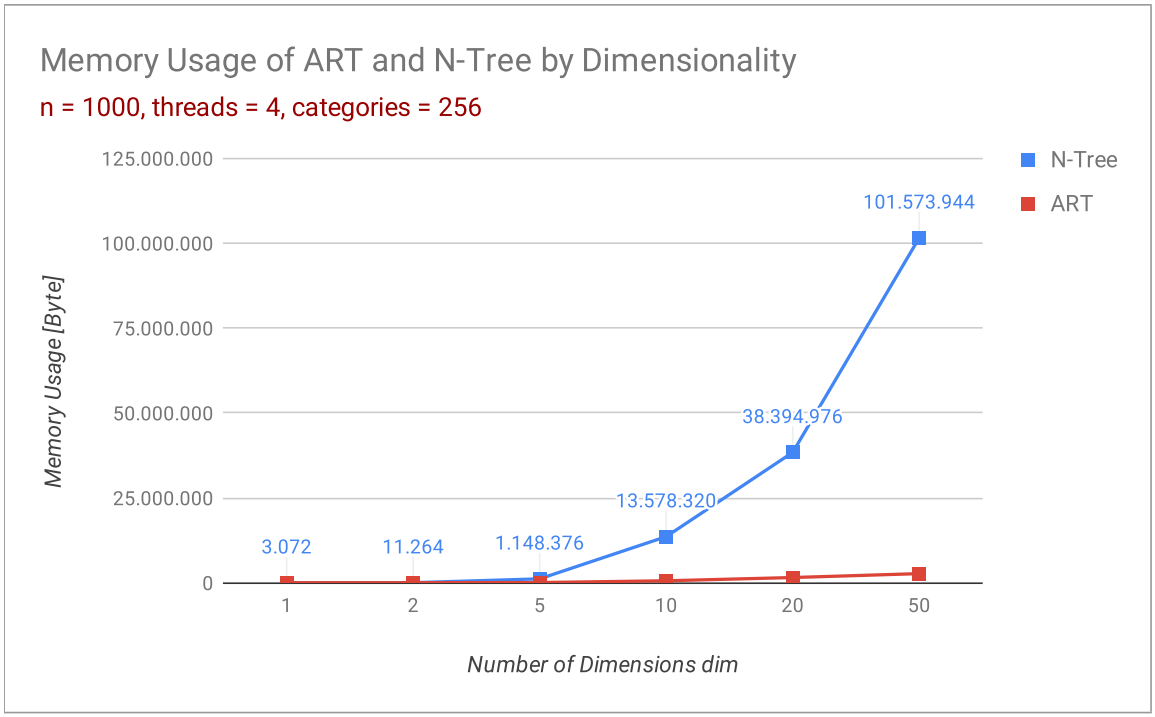


Figure 5.11.: Memory Usage of ART and N-Tree by Dimensionality

6. Conclusion

6.1. Summary

In this work, three of the basic skyline algorithms Naive-Nested-Loops, Block-Nested-Loops and Divide-and-Conquer, as well as the newer ST-S algorithm were introduced. The algorithms were placed into the "big picture" of the current state-of-the-art skyline algorithms and explained in detail. Thereafter, the novel skyline algorithm SARTS, which utilizes the highly efficient ART tree, was presented. The algorithm keeps all the advantages of ST-S, while being significantly more memory-efficient at the same time. The algorithms were parallelized using different approaches and frameworks, which were explained in greater detail. In the last chapter, an evaluation of the conducted tests was carried out and the outcomes analyzed.

With the results of this work, the following points should be considered when choosing the right skyline algorithm for the particular use case:

- When the application scenario assumes continuous attribute values and does not require progressive behavior, then Block-Nested-Loops seems to be a very potent "all-rounder" algorithm, well suited for both I/O-intensive as well as in-memory databases. At this point, some of the newer algorithms based on Block-Nested-Loops should be considered, such as SFS [7] and SaLSa [2]. The presorting and the threshold approaches showed that they can improve an algorithm such as ST-S, and thus can be recommended to be applied to BNL as well.

- In a scenario where progressiveness is important, an online-capable algorithm such as ST-S or SARTS should be chosen. Both algorithms perform excellently for medium-range to high n and provide good scalability in parallelized environments. While tree-based algorithms do not scale well with high dimensionality, most online services seem to have a high number of database entries with mostly low dimensionality nowadays[1].

- In environments that require efficient memory usage SARTS is highly recommended to be chosen over ST-S due to its significantly lower space consumption.

6.2. Outlook

While the parallelization approaches introduced in this work showed to improve the respective sequential versions of the algorithms, some of them have the potential to perform

[1] Consider a database holding around 1 million hotels with 5-10 different categorical attributes each for this purpose.

even better when running on more than 4 threads. This could be further tested, provided that a suiting machine is available.

The current version of the SARTS algorithm already keeps up with ST-S in terms of computation time, and overtakes it by far in terms of efficient memory usage. Nevertheless, it still can be improved by utilizing not the basic, but the full version of the ART tree, including both lazy expansion and path compression. These techniques enable faster insert operations and dominance checks, and also save even more memory by leaving out unnecessary nodes. It is expected that with these improvements the SARTS algorithm would bypass ST-S both in speed and in efficiency of memory usage. The only (known) successor to ST-S to date is the algorithm TA-SKY, also proposed by the same authors in [22]. Thus, it would be interesting to compare an improved version of SARTS to TA-SKY in terms of time and memory management.

SARTS was originally developed in the context of an in-memory database, such as HyPer [13]. Still, it also seems suitable for I/O-intensive DBMS due to BNL being one of its "ancestors". While performance tests in this work showed that SARTS can deal quite well with synthetic datasets, it is always sensible to test an algorithm on real-world datasets. For this reason, it can be recommended to integrate SARTS in a database system, in order to see how well it fares in this environment.

As SARTS should be primarily used as an online algorithm, it also appears sensible to test it within a database system which demands its algorithms to work progressively, i.e. to deliver first results before the entire skyline is computed. Some sort of interactive web application seems fit for this purpose.

Appendix

A. Performance Measurements

Table A.1.: Non-Progressive Algorithms by Number of Tuples

dim = 5, threads = 4

	Time in [s]			
n	BNL i/m	BNL p/c	DNC	DNC parallel
10	0,000	0,000	0,000	0,000
100	0,001	0,001	0,009	0,004
1.000	0,006	0,005	0,078	0,076
5.000	0,026	0,028	0,453	0,445
10.000	0,057	0,061	0,967	0,935
50.000	0,345	0,364	5,215	5,032
100.000	0,448	0,485	10,800	10,316
500.000	1,772	1,932	57,852	54,366
1.000.000	2,886	3,181	118,021	107,641
5.000.000	12,650	14,485	638,834	558,816
10.000.000	22,426	26,094	1.313,030	1.147,510

Table A.2.: Non-Progressive Algorithms by Dimensionality

n = 10.000, threads = 4

	Time in [s]			
dimension	BNL i/m	BNL p/c	DNC	DNC parallel
1	0,006	0,093	0,257	0,239
2	0,007	0,010	0,281	0,263
5	0,067	0,070	0,933	0,922
10	1,911	1,904	6,706	5,692
20	5,276	5,296	14,269	10,012
50	5,509	5,554	14,305	10,041
100	5,647	5,672	14,577	10,130
200	5,707	5,762	16,689	12,066

Table A.3.: Non-Progressive Algorithms by Number of Threads

n = 100.000, dim = 5

	Time in [s]			
threads	BNL i/m	BNL p/c	DNC	DNC parallel
1	0,433	0,470	10,685	10,689
2	0,488	0,516	10,836	10,646
3	0,448	0,485	10,937	10,480
4	0,432	0,467	10,947	10,475

Table A.4.: Progressive Algorithms by Number of Tuples

dim = 5, threads = 4, categories = 256

	Time in [s]					
n	NNL	NNL parallel	STS	STS parallel	SARTS	SARTS parallel
10	0,000	0,002	0,001	0,002	0,001	0,001
100	0,000	0,001	0,005	0,009	0,004	0,003
1.000	0,009	0,005	0,028	0,026	0,018	0,025
5.000	0,073	0,038	0,136	0,120	0,151	0,133
10.000	0,187	0,093	0,235	0,229	0,272	0,239
50.000	2,347	1,139	0,961	0,657	0,918	0,685
100.000	5,472	2,762	1,550	1,149	1,590	1,245
500.000	37,254	19,372	9,251	5,718	9,860	5,936
1.000.000	91,618	47,616	15,578	10,860	16,501	11,498
5.000.000	550,613	288,644	56,798	37,659	57,512	38,786
10.000.000	1.359,420	719,286	134,610	75,404	137,417	78,944

Table A.5.: Progressive Algorithms by Dimensionality

n = 10.000, threads = 4, categories = 256

	Time in [s]					
dimension	NNL	NNL parallel	STS	STS parallel	SARTS	SARTS parallel
1	0,012	0,0137	0,050	0,026	0,039	0,026
2	0,010	0,012	0,054	0,032	0,054	0,032
5	0,187	0,093	0,235	0,229	0,272	0,239
10	2,075	0,942	11,340	11,548	11,431	11,837
30	3,838	1,823	14,491	17,381	13,392	15,754
50	3,929	1,879	16,535	20,582	15,239	18,129
100	4,266	1,972	23,044	30,482	20,975	25,990

Table A.6.: Progressive Algorithms by Number of Threads

n = 100.000, dim = 5, categories = 256

	Time in [s]					
threads	NNL	NNL parallel	STS	STS parallel	SARTS	SARTS parallel
1	4,509	4,584	1,865	2,650	1,918	2,804
2	4,674	4,096	1,446	2,081	1,463	2,164
3	4,705	2,513	1,619	1,352	1,638	1,380
4	4,452	2,243	1,922	1,200	1,977	1,253

Table A.7.: Memory Usage by Number of Tuples

dim = 5, threads = 4, categories = 256

	Memory in [Byte]	
n	N-Tree	ART
10	61.352	2.800
100	368.504	17.360
1.000	1.076.392	51.440
10.000	2.111.432	101.440
50.000	4.025.480	192.800
100.000	4.389.352	208.320
500.000	5.596.520	271.920
1.000.000	5.866.136	290.480

Table A.8.: Memory Usage by Dimensionality

n = 1.000, threads = 4, categories = 256

	Memory in [Byte]	
dim	N-Tree	ART
1	3.072	1.024
2	11.264	1.664
5	1.148.376	54.800
10	13.578.320	572.720
20	38.394.976	1.530.560
50	101.573.944	2.682.400

B. C++ Code

B.1. Dominates Operation for NNL, BNL and DNC

```
/**
* Checks whether one tuple dominates the other and returns true|false
* @param dominator the tuple to check for dominating
* @param dominated the tuple to check for being dominated
*/
bool dominates(const std::vector<int> &dominator, const std::vector<int>
 ↪  &dominated){
   bool flag = true;
   for(std::vector<int>::size_type i = 0; i< dominated.size(); i++){
      if(dominator[i] > dominated[i]) return false;
      if(dominated[i] > dominator[i]) flag = false;
   }
   if(flag) return false;
   return true;
}
```

B.2. Naive-Nested-Loops

```
void computeSkylineProduce(){
   for(std::size_t i = 0; i < storage.size(); i++){
      bool not_dominated = true;
      for(std::size_t j = 0; j < storage.size(); j++){
         if(i != j){
            if(dominates(storage[j], storage[i])){
               not_dominated = false;
               break;
            }
         }
      }
      if(not_dominated){
         parent->consume(storage[i]);
      }
   }
}
```

B.3. Naive-Nested-Loops Parallelized

```
void computeSkylineProduceParallel(){
   const std::vector<std::vector<int>> storage = this->storage;
```

```
   CatOperator *parent = this->parent;
   parallel_for(std::size_t(0), storage.size(), [this, storage, parent](
   ↪  std::size_t i ) {
      bool not_dominated = true;
      bool flag = false;
      for(std::size_t j = 0; j < storage.size() && !flag; j++){
         if(i != j){
            if(dominates(storage[j], storage[i])){
               not_dominated = false;
               flag = true;
            }
         }
      }
      // The following mutex slows down the parallelization, but
      ↪  provides an easy way to avoid race conditions
      // In case no mutex is used, the programmer needs to make sure
      ↪  that no race condition occurs in the following if-block
      static spin_mutex mtx;
      spin_mutex::scoped_lock lock(mtx);
      if(not_dominated){
         parent->consume(storage[i]);
      }
   } );
}
```

B.4. Block-Nested-Loops Volcano Model

```
void computeSkyline(){
   // storage is the window here
   storage.push_back(child->getNext());
   while(true){
      std::vector<int> tuple = child->getNext();
      if(!tuple.empty()){
         storage.push_back(tuple);
         for(std::size_t j = 0; j < storage.size()-1; j++){
            if(dominates(storage.back(), storage[j])){
               storage.erase(storage.begin() + j);
               j--;
            }
            else if (dominates(storage[j], storage.back())){
               storage.erase(storage.begin() + storage.size()-1);
               break;
            }
         }
      }
      else break;
   }
}
```

B.5. Block-Nested-Loops Produce/Consume

```
void computeSkylineProduce(){
   // storage contains tuples produced by the generator
   std::vector<std::vector<int>> window;
   window.push_back(storage[0]);
   for(std::size_t i = 1; i < storage.size(); i++){
      std::vector<int> tuple = storage[i];
      window.push_back(tuple);
      for(std::size_t j = 0; j < window.size()-1; j++){
         if(dominates(window.back(), window[j])){
            window.erase(window.begin() + j);
            j--;
         }
         else if (dominates(window[j], window.back())){
            window.erase(window.begin() + window.size()-1);
            break;
         }
      }
   }
   for(std::size_t i = 0; i < window.size(); i++){
      parent->consume(window[i]);
   }
}
```

B.6. Divide-and-Conquer

Algorithm

```
std::vector<std::vector<double>> computeSkyline(const
↪  std::vector<std::vector<double>> &M, const int &dimension){
   if(M.size() == 1) return M;

   std::vector<double> pivot = median(M, dimension-1); // dimension-1
   ↪  because we need the last index
   std::pair<std::vector<std::vector<double>>,
   ↪  std::vector<std::vector<double>>> P = partition(M, dimension-1,
   ↪  pivot);

   std::vector<std::vector<double>> S_1, S_2;
   S_1 = computeSkyline(P.first, dimension);
   S_2 = computeSkyline(P.second, dimension);

   std::vector<std::vector<double>> result;
   std::vector<std::vector<double>> merge_result = mergeBasic(S_1, S_2,
   ↪  dimension);

   // Union S_1 and merge_result
   for(std::vector<std::vector<double>>::size_type i = 0; i <
   ↪  S_1.size(); i++){
      result.push_back(S_1[i]);
```

```
   }
   for(std::vector<std::vector<double>>::size_type i = 0; i <
   ↪  merge_result.size(); i++){
      result.push_back(merge_result[i]);
   }

   return result;
}
```

Partition Operation

```
std::pair<std::vector<std::vector<double>>,
↪  std::vector<std::vector<double>>> partition(const
↪  std::vector<std::vector<double>> &tuples, const int &dimension,
↪  const std::vector<double> &pivot){
   std::vector<std::vector<double>> P_1, P_2;
   std::pair<std::vector<std::vector<double>>,
   ↪  std::vector<std::vector<double>>> partitions;

   for(std::vector<std::vector<double>>::size_type i = 0; i <
   ↪  tuples.size(); i++){
      if(tuples[i][dimension] < pivot[dimension])
         P_1.push_back(tuples[i]);
      else
         P_2.push_back(tuples[i]);
   }

   partitions.first = P_1;
   partitions.second = P_2;

   return partitions;
}
```

Merge Operation

```
std::vector<std::vector<double>>
↪  mergeBasic(std::vector<std::vector<double>> S_1, const
↪  std::vector<std::vector<double>> &S_2, const int &dimension){
   std::vector<std::vector<double>> result;

   if(S_2.size() == 0) return result;

   if(S_1.size() == 1){ // trivial case - S_1 has only 1 tuple
      for(std::vector<std::vector<double>>::size_type i = 0; i <
      ↪  S_2.size(); i++){
         if(!dominates(S_1[0], S_2[i]))
            result.push_back(S_2[i]);
      }
   }
   else if(S_2.size() == 1){ // trivial case - S_2 has only 1 tuple
      result.push_back(S_2[0]);
```

```
        for(std::vector<std::vector<double>>::size_type i = 0; i <
        ↪  S_1.size(); i++){
            if(dominates(S_1[i], S_2[0])){
                result.erase(result.begin());
                break;
            }
        }
    }
    else if(S_1[0].size() == 2){ // low dimension
        // Min from S_1 according to dimension 1
        std::sort(S_1.begin(), S_1.end(), [](const std::vector<double> &a,
        ↪  const std::vector<double> &b){
            return a[0] < b[0];
        });
        std::vector<double> min = S_1[0];
        // Compare S_2 to Min according to dimension 1; in dimension 2 S_1
        ↪  is always better
        for(std::vector<std::vector<double>>::size_type i = 0; i <
        ↪  S_2.size(); i++){
            if(S_2[i][0] < min[0]) result.push_back(S_2[i]);
        }
    }
    else{ // general case
        std::vector<double> pivot = median(S_1, dimension-1-1);
        std::pair<std::vector<std::vector<double>>,
        ↪  std::vector<std::vector<double>>> partitions_dim_1 =
        ↪  partition(S_1, dimension-1-1, pivot);
        std::pair<std::vector<std::vector<double>>,
        ↪  std::vector<std::vector<double>>> partitions_dim_2 =
        ↪  partition(S_2, dimension-1-1, pivot);
        std::vector<std::vector<double>> result_1, result_2, result_3;

        result_1 = mergeBasic(partitions_dim_1.first,
        ↪  partitions_dim_2.first, dimension);
        result_2 = mergeBasic(partitions_dim_1.second,
        ↪  partitions_dim_2.second, dimension);
        result_3 = mergeBasic(partitions_dim_1.first, result_2,
        ↪  dimension-1);

        // Union result_1 and result_3
        for(std::vector<std::vector<double>>::size_type i = 0; i <
        ↪  result_1.size(); i++){
            result.push_back(result_1[i]);
        }
        for(std::vector<std::vector<double>>::size_type i = 0; i <
        ↪  result_3.size(); i++){
            result.push_back(result_3[i]);
        }
    }

    return result;
}
```

B.7. ST-S/ SARTS

```
void computeSkylineProduce(){
   // pre-sort tuples in place
   sort(storage);
   std::vector<int> t_stop = storage[0];
   tree.insert(storage[0], tree.root, 0);
   parent->consume(storage[0]);
   for(std::size_t i = 1; i < storage.size(); i++){
      // stop if all the tuples left are dominated Ã -priori
      if((max(t_stop) <= min(storage[i])) && (t_stop != storage[i])){
         return;
      }
      // check for dominance
      if(!tree.is_dominated(storage[i], tree.root, 0,
      ↪  tree.score(storage[i]))){
         parent->consume(storage[i]);
         tree.insert(storage[i], tree.root, 0);
         if(max(storage[i]) < max(t_stop)){
            t_stop = storage[i];
         }
      }
   }
}
```

B.8. ST-S/ SARTS Parallelized

Algorithm

```
void computeSkylineProduce(){
   const std::vector<std::vector<int>> storage = this->storage;
   const std::size_t number_of_threads = NUMBER_OF_THREADS;
   std::vector<Tree*> subtrees;
   for(std::size_t i = 0; i < NUMBER_OF_THREADS; i++){
      Tree* subtree = new Tree(tree.get_attributes());
      subtrees.push_back(subtree);
   }
   std::future<void> futures[NUMBER_OF_THREADS];
   // compute subquery skylines
   for(std::size_t i = 0; i < number_of_threads; i++){
      std::vector<std::vector<int>> subset;
      if(i == number_of_threads-1){
         subset.resize(storage.size() / number_of_threads +
         ↪  storage.size() % number_of_threads);
      }
      else{
         subset.resize(storage.size() / number_of_threads);
      }
      for(std::size_t j = 0; j < subset.size(); j++){
         subset[j] = storage[i*(storage.size()/number_of_threads) + j];
      }
```

```
      // Replace &ParallelSTS by &ParallelSARTS to receive SARTS
      futures[i] = std::async(std::launch::async,
      ↪  &ParallelSTS::computeSkylineSubset, this, i, subset,
      ↪  subtrees[i]);
   }
   for(std::size_t i = 0; i < NUMBER_OF_THREADS; i++){
      futures[i].get();
   }
   // compute final skyline
   std::vector<std::vector<int>> input;
   for(std::size_t i = 0; i < subset_results.size(); i++){
      if(!subset_results[i].empty()){
         input.push_back(subset_results[i]);
      }
   }
   sort(input);
   std::vector<int> t_stop = input[0];
   tree.insert(input[0], tree.root, 0);
   parent->consume(input[0]);
   for(std::size_t i = 1; i < input.size(); i++){
      if((max(t_stop) <= min(input[i])) && (t_stop != input[i])){
         return;
      }
      if(!ntree.is_dominated(input[i], tree.root, 0,
      ↪  tree.score(input[i]))){
         parent->consume(input[i]);
         tree.insert(input[i], tree.root, 0);
         if(max(input[i]) < max(t_stop)){
            t_stop = input[i];
         }
      }
   }
   // free memory
   for(std::size_t i = 0; i < subtrees.size(); i++){
      if(subtrees[i] != nullptr) delete subtrees[i];
   }
}
```

ComputeSkylineSubset Operation

```
void computeSkylineSubset(unsigned threadNumber,
↪  std::vector<std::vector<int>> tuples, Tree* tree){
   // pre-sort tuples in place
   sort(tuples);
   std::vector<int> t_stop = tuples[0];
   tree->insert(tuples[0], tree->root, 0);
   subset_results[threadNumber * (subset_results.size() /
   ↪  NUMBER_OF_THREADS) + 0] = tuples[0];
   for(std::size_t k = 1; k < tuples.size(); k++){
      // stop if all tuples left are dominated a-priori
      if((max(t_stop) <= min(tuples[k])) && (t_stop != tuples[k])){
```

```
            return;
         }
         // check for dominance
         if(!tree->is_dominated(tuples[k], tree->root, 0,
         ↪  tree->score(tuples[k]))){
            subset_results[threadNumber * (subset_results.size() /
            ↪  NUMBER_OF_THREADS) + k] = tuples[k];
            tree->insert(tuples[k], tree->root, 0);
            if(max(tuples[k]) < max(t_stop)){
               t_stop = tuples[k];
            }
         }
      }
   }
```

B.9. N-Tree

Insert Operation

```
void NTree::insert(const std::vector<int> &tuple, node* p, unsigned int
↪  level){
   if(level == 0){
      p->minScore = 0;
      p->maxScore = 0;
      for(std::size_t i = 0; i < tuple.size(); i++){
         p->maxScore += (int) (pow(2.0, (double) (tuple.size()-i)) *
         ↪  attributes[attributes.size()-1]);
      }
   }
   else{
      p->minScore = 0;
      for(std::size_t i = 0; i < level; i++){
         p->minScore += (int) (pow(2.0, (double) (tuple.size()-i)) *
         ↪  tuple[i]);
      }
      p->maxScore = p->minScore;
      for(std::size_t i = level; i < tuple.size(); i++){
         p->maxScore += (int) (pow(2.0, (double) (tuple.size()-i)) *
         ↪  attributes[attributes.size()-1]);
      }
   }
   if(level == tuple.size()){
      p->tupleIDs.push_back(tupleID++);
   }
   else{
      if(p->children.empty()){
         p->children.resize(attributes.size());
      }
      if(!p->children[tuple[level]]){
         p->children[tuple[level]]=new node();
      }
```

```
      insert(tuple, p->children[tuple[level]], level+1);
   }
}
```

Is_Dominated Operation

```
bool NTree::is_dominated(const std::vector<int> &tuple, node* p,
↪  unsigned int level, unsigned int currentScore){
   if(p==nullptr || (currentScore < p->minScore)){
      return false;
   }
   if((level == tuple.size()) && (score(tuple) != p->minScore)){
      return true;
   }
   if((level == tuple.size()) && (score(tuple) == p->minScore)){
      return false;
   }
   // search the subtrees from left to right
   unsigned int weight = (int) (pow(2.0, (double) (tuple.size()-level))
   ↪  * tuple[level]);
   for(int i = 0; i < tuple[level]; i++){
      if(is_dominated(tuple, p->children[i], level+1, currentScore +
      ↪  weight)){
         return true;
      }
   }
   if(is_dominated(tuple, p->children[tuple[level]], level+1,
   ↪  currentScore)){
      return true;
   }
   return false;
}
```

B.10. ART

Insert Operation

```
void ART::insert(const std::vector<int> &tuple, Node *&parent, Node
↪  *&current, unsigned int level){
   if(level == 0){
      current->minScore = 0;
      current->maxScore = 0;
      for(std::size_t i = 0; i < tuple.size(); i++){
         current->maxScore += (int) (pow(2.0, (double) (tuple.size()-i))
         ↪  * attributes[attributes.size()-1]);
      }
   }
   else{
      current->minScore = 0;
      for(std::size_t i = 0; i < level; i++){
```

```
            current->minScore += (int) (pow(2.0, (double) (tuple.size()-i))
            ↪  * tuple[i]);
         }
         current->maxScore = current->minScore;
         for(std::size_t i = level; i < tuple.size(); i++){
            current->maxScore += (int) (pow(2.0, (double) (tuple.size()-i))
            ↪  * attributes[attributes.size()-1]);
         }
      }
      if(level == tuple.size()){
         current->tupleIDs.push_back(tupleID++);
      }
      else{
         Node* child = findChild(current, tuple[level]);
         if(!child){
            switch(current->type){
               case NodeType4:
                  if(current->count == 4)
                     grow(parent, current, tuple[level-1]);
                  break;
               case NodeType16:
                  if(current->count == 16)
                     grow(parent, current, tuple[level-1]);
                  break;
               case NodeType48:
                  if(current->count == 48)
                     grow(parent, current, tuple[level-1]);
                  break;
               case NodeType256:
               default:
                  break;
            }
            child = newChild(current, tuple[level]);
         }
         insert(tuple, current, child, level+1);
      }
   }
```

Is_Dominated Operation

```
bool ART::is_dominated(const std::vector<int> &tuple, Node* p, unsigned
↪  int level, unsigned int currentScore){
   if(p==NULL || (currentScore < p->minScore)){
      return false;
   }
   if((level == tuple.size()) && (score(tuple) != p->minScore)){
      return true;
   }
   if((level == tuple.size()) && (score(tuple) == p->minScore)){
      return false;
   }
```

```
   // search the subtrees from left to right
   unsigned int weight = (int) (pow(2.0, (double) (tuple.size()-level))
   ↪  * tuple[level]);
   for(int i = 0; i < tuple[level]; i++){
      Node* child = findChild(p, i);
      if(child){ // if child not null
         if(is_dominated(tuple, child, level+1, currentScore + weight)){
            return true;
         }
      }
   }
   Node* child = findChild(p, tuple[level]);
   if(child){
      if(is_dominated(tuple, child, level+1, currentScore)){
         return true;
      }
   }
   return false;
}
```

Find_Child Operation

```
Node* ART::findChild(Node* parent, const int &attribute){
   switch(parent->type){
      case NodeType4: {
         Node4* node = static_cast<Node4*>(parent);
         for (unsigned i = 0; i < node->count; i++){
            if (node->key[i] == attribute){
               return node->children[i];
            }
         }
         return NULL;
      }
      case NodeType16: {
         Node16* node = static_cast<Node16*>(parent);
         for (unsigned i = 0; i < node->count; i++){
            if (node->key[i] == attribute){
               return node->children[i];
            }
         }
         return NULL;
      }
      case NodeType48: {
         Node48* node = static_cast<Node48*>(parent);
         if (node->childIndex[attribute] != emptyMarker){
            return node->children[node->childIndex[attribute]];
         }
         else
            return NULL;
      }
      case NodeType256: {
```

```
         Node256* node = static_cast<Node256*>(parent);
         return node->children[attribute];
      }
      default: {
         return NULL;
      }
   }
}
```

New_Child Operation

```
Node* ART::newChild(Node *&node, const int &attribute){
   Node4* child = new Node4();
   switch(node->type){
      case NodeType4: {
         Node4* parent = static_cast<Node4*>(node);
         unsigned pos;
         // make free space for the new child entry
         for (pos = 0; (pos < parent->count) && (parent->key[pos] <
          ↪  attribute); pos++);
         memmove(parent->key+pos+1, parent->key+pos, parent->count-pos);
         memmove(parent->children+pos+1, parent->children+pos,
          ↪  (parent->count-pos)*sizeof(Node*));
         parent->key[pos] = attribute;
         parent->children[pos] = child;
         parent->count++;
         break;
      }
      case NodeType16: {
         Node16* parent = static_cast<Node16*>(node);
         unsigned pos;
         // make free space for the new child entry
         for (pos = 0; (pos < parent->count) && (parent->key[pos] <
          ↪  attribute); pos++);
         memmove(parent->key+pos+1, parent->key+pos, parent->count-pos);
         memmove(parent->children+pos+1, parent->children+pos,
          ↪  (parent->count-pos)*sizeof(Node*));
         parent->key[pos] = attribute;
         parent->children[pos] = child;
         parent->count++;
         break;
      }
      case NodeType48: {
         Node48* parent = static_cast<Node48*>(node);
         unsigned pos = parent->count;
         // if there are empty slots inbetween, use them instead of
          ↪  appending the child pointer at the end
         if(parent->children[pos]){
            for(pos = 0; parent->children[pos] != NULL; pos++);
         }
         parent->children[pos] = child;
```

```
            parent->childIndex[attribute] = pos;
            parent->count++;
            break;
         }
         case NodeType256: {
            Node256* parent = static_cast<Node256*>(node);
            parent->children[attribute] = child;
            parent->count++;
            break;
         }
         default:
            break;
      }
      return child;
}
```

Grow Operation

```
void ART::grow(Node *&parent, Node *&node, const int &indexOfCurrent){
      Node* newNode;
      switch(node->type){
         case NodeType4:
            newNode = new Node16();
            newNode->count = 4;
            for(std::size_t i = 0; i < 4; i++){
               static_cast<Node16*>(newNode)->key[i] =
                ↪  static_cast<Node4*>(node)->key[i];
            }
            for(std::size_t i = 0; i < 4; i++){
               static_cast<Node16*>(newNode)->children[i] =
                ↪  static_cast<Node4*>(node)->children[i];
            }
            break;
         case NodeType16:
            newNode = new Node48();
            newNode->count = 16;
            for(std::size_t i = 0; i < 16; i++){
               static_cast<Node48*>(newNode)->children[i] =
                ↪  static_cast<Node16*>(node)->children[i];
            }
            for (unsigned i = 0; i < node->count; i++){
               static_cast<Node48*>(newNode)->childIndex
                ↪  [static_cast<Node16*>(node)->key[i]] = i;
            }
            break;
         case NodeType48:
            newNode = new Node256();
            newNode->count = 48;
            for (unsigned i = 0; i < 256; i++){
               if (static_cast<Node48*>(node)->childIndex[i] != 48){ //
                ↪  slot not empty
```

```
                static_cast<Node256*>(newNode)->children[i] =
                ↪  static_cast<Node48*>(node)->children
                ↪  [static_cast<Node48*>(node)->childIndex[i]];
            }
         }
         break;
      default:
         break;
   }
   newNode->minScore = node->minScore;
   newNode->maxScore = node->maxScore;
   for(std::size_t i = 0; i < node->tupleIDs.size(); i++){
      newNode->tupleIDs[i] = node->tupleIDs[i];
   }

   if(node != root){
      // Code to updateParent() is not given for space reasons. Contact
      ↪  the author if needed.
      updateParent(parent, newNode, indexOfCurrent);
   }
   delete node;
   node = newNode;
}
```

Bibliography

[1] *Volcano Model*. http://dbms-arch.wikia.com/wiki/Volcano_Model. Last Accessed on Sept. 25, 2018.

[2] I. BARTOLINI, P. CIACCIA, and M. PATELLA, *Efficient Sort-Based Skyline Evaluation*, 33(4), 11 2008.

[3] O. A. R. BOARD, *OpenMP application programming interface*. https://www.openmp.org/. Last Accessed on Sept. 25, 2018.

[4] K. S. BØGH, S. CHESTER, and I. ASSENT, *Work-Efficient Parallel Skyline Computation for the GPU*, vol. 8, 2015, pp. 962–973.

[5] S. BORZSONY, D. KOSSMANN, and K. STOCKER, *The Skyline operator*, in Proceedings 17th International Conference on Data Engineering, April 2001, pp. 421–430.

[6] S. CHESTER, D. ŠIDLAUSKAS, I. ASSENT, and K. S. BØGH, *Scalable Parallelization of Skyline Computation for Multi-core Processors*, in IEEE 31st International Conference on Data Engineering, Seoul, South Korea, April 2015, IEEE.

[7] J. CHOMICKI, P. GODFREY, J. GRYZ, and D. LIANG, *Skyline with presorting*, in Proceedings 19th International Conference on Data Engineering, March 2003, pp. 717–719.

[8] I. CORPORATION, *parallel_for construct, part of Threading Building Blocks*. https://software.intel.com/en-us/node/506057. Last Accessed on Sept. 25, 2018.

[9] I. CORPORATION, *parallel_sort Template Function, part of Threading Building Blocks*. https://software.intel.com/en-us/node/506167. Last Accessed on Sept. 25, 2018.

[10] G. GRAEFE, *Volcano — an extensible and parallel query evaluation system*, vol. 6, Feb 1994, pp. 120–135.

[11] C. KALYVAS and T. TZOURAMANIS, *A Survey of Skyline Query Processing*, April 2017.

[12] A. KEMPER and A. EICKLER, *Datenbanksysteme: Eine Einfuehrung*, De Gruyter Oldenbourg, 2015.

[13] A. KEMPER and T. NEUMANN, *HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots*, in IEEE 27th International Conference on Data Engineering, Hannover, Germany, April 2011, IEEE, pp. 195–206.

[14] D. KOSSMANN, F. RAMSAK, and S. ROST, *Shooting Stars in the Sky: An Online Algorithm for Skyline Queries*, in Proceedings of the 28th VLDB Conference, Hong Kong, China, August 2002, pp. 275–286.

[15] H. T. KUNG, F. LUCCIO, and F. P. PREPARATA, *On finding the Maxima of a Set of Vectors*, in Journal of the Association for Computing Machinery, vol. 22, October 1975, pp. 469–476.

[16] V. LEIS, A. KEMPER, and T. NEUMANN, *The Adaptive Radix Tree: ARTful Indexing for Main-memory Databases*, in Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013), ICDE '13, Washington, DC, USA, 2013, IEEE Computer Society, pp. 38–49.

[17] V. LEIS and T. NEUMANN, *Compiling Database Queries into Machine Code*, in Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, IEEE, 2014, pp. 3–11.

[18] S. LIKNES, A. VLACHOU, C. DOULKERIDIS, and K. NØRVÅG, *APSkyline: Improved Skyline Computation for Multicore Architectures*, vol. 8421 of Lecture Notes in Computer Science, Springer International Publishing, Switzerland, March 2014, pp. 312–326.

[19] K. MULLESGAARD, J. L. PEDERSEN, H. LU, and Y. ZHOU, *Efficient Skyline Computation in MapReduce*, in Proc. 17th International Conference on Extending Database Technology, Athens, Greece, March 2014, pp. 37–48.

[20] D. PAPADIAS, Y. TAO, G. FU, and B. SEEGER, *An Optimal and Progressive Algorithm for Skyline Queries*, in ACM Proceedings, San Diego, CA, USA, June 2003.

[21] S. PARK, T. KIM, J. PARK, J. KIM, and H. IM, *Parallel Skyline Computation on Multicore Architectures*, in 2009 IEEE 25th International Conference on Data Engineering, Shanghai, China, March 2009, pp. 760–771.

[22] M. F. RAHMAN, A. ASUDEH, N. KOUDAS, and G. DAS, *Efficient Computation of Subspace Skyline over Categorical Domains*, in ACM Proceedings, Singapur, November 2017, CIKM, pp. 407–416. Session 2E: Skyline Queries.

[23] K.-L. TAN, P.-K. ENG, and B. C. OOI, *Efficient Progressive Skyline Computation*, in Proc. 27th International Conference on Very Large Data Bases, Roma, Italy, September 2001.