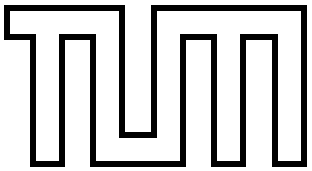


DEPARTMENT OF FINANCE

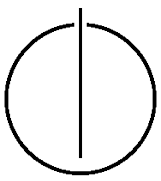
TECHNICAL UNIVERSITY OF MUNICH

Interdisciplinary Project in Finance and Informatics

Leads and Lags of Corporate Bonds and Stocks

Leads und Lags von Unternehmensanleihen und Aktien

Author:	Alex Kulikov
Supervisor:	Prof. Dr. Sebastian Müller
Advisor:	Zihan Gong
Date:	March 31, 2021



Abstract

Contents

1. Introduction

In the scope of an Interdisciplinary Project (*IDP* in the following) at the Technical University of Munich, the lead and lag relationship between corporate bond and stock returns had to be analyzed. With the needed stock data already provided, the first step of the IDP was to develop a tool which would be able to automatically extract static and time series data from the Thomson Reuters Datastream financial database. In the second step of the IDP, the extracted bond data had to be cleaned and prepared for further analysis by various transformation techniques. Additionally, a matching approach to join the stock and bond datasets by issuing company had to be developed. In the third and final step of the IDP, the resulting bond-stock database had to be checked for any sort of lead and/or lag relationships within bond and stock return pairs.

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 [15]. 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. Datastream Extraction Tool

In order to draw any conclusions regarding the relationship of bond and stock returns, the respective static and time series data needs to be acquired first. Since equity data is already available from the beginning of the IDP, the bond data is the only one which had to be acquired. For bond data extraction, the financial database product Datastream, provided by Thomson Reuters, can be used, since it is licensed for usage by TUM students and employees.

2.1. Download Solution

As Thomson Reuters has a wide range of products which can be used for different types of data, the first thing that needed to be done, was to determine the most suitable product to download both static and time series data for corporate bonds. After some time spent reading up and gathering information on the Thomson Reuters product portfolio, it became apparent that some of the products, such as the TR Python API, are only suitable for equity data download, and not for corporate bonds, and only have a very limited number of parameters available for download. On the other hand, it was found that other Thomson Reuters products, which would normally be suitable for automated download of bond data, such as e.g. DataScope Select (DSS) or Thomson Reuters Tick History (TRTH), are not included in the existing academic license. Other products – noticeably the Datastream Web Service (DSWS) API, which is most suited for such requests – are generally not available for academic clients.

These findings were a significant setback for the bond data extraction, since the only option left to acquire large amounts of corporate bond data, was over the Datastream Add-In for Microsoft Excel. While this add-in is rather convenient for small-scale manual requests with the help of so-called request tables, it is not optimized for large data extraction queries. It does not provide an API for customizable requests. Instead, communication with the Datastream server is handled over a single API call available in VBA. This one and only callable function is implemented in C++, and can only be invoked in a black-box manner, since the provider does not give out its implementation. This leads to only one possible solution to automatically extract corporate bond data from Datastream. It can be described with the following steps:

1. Acquire Datastream codes / identifiers for all financial instruments which need to be downloaded.

2. Split these identifiers into batches small enough to be processed in a single Datastream request.

3. Fill a request table with as many requests as needed to include all the batches.

4. Launch the Datastream requests for all the batches one after the other.

5. Monitor the download process to ensure that the data is being consistently downloaded.

The programmatic development of the download tool will be based exactly on these five steps. The last step (monitoring the execution) is especially crucial and complex to implement. The reason for this is, as previously mentioned, that the Datastream Excel Add-In is not well-fit for large data downloads. Therefore, the following problems continuously arise during the download process:

- Datastream add-in eventually signs out for no obvious reason.
- Data download hangs, without any notification stating the reason or the hanging fact itself.
- Excel suspends the add-in and places it into a blacklist for repeated faulty behavior.

Since VBA is single threaded and cannot detect or react to erroneous behavior when the download is running, it is impossible to do the monitoring in the VBA/Excel environment. For this purpose, a Python wrapper was developed as will be explained in .

2.2. Bond Identifiers Acquisition

Since for both static and time series requests Datastream requires unique financial instrument codes to be provided, it is first necessary to obtain a list of identifiers for the securities for which the data needs to be downloaded. The most commonly used unique security identifier in Datastream is the so-called Datastream Code (short *dscd*). In the scope of the project two different approaches have been developed for this task, and will be introduced in the following.

2.2.1. Programmatic Identifier Extraction

For the purpose of this work, we are interested in corporate bonds from all possible jurisdictions, and with any possible coupon an currency parameters. The only restriction is that we only concentrate on the issue date range between Dec 31, 1999 and June 30, 2020 (date of extraction).

Datastream allows to filter its financial security dataset by these parameters, e.g. when clicking on *Find Series* in a request table. After the securities have been filtered for the desired corporate bonds, these can be selected by repeatedly checking the box to select all bonds on the current page, and then clicking on *Next* to switch to next page. This is due to Datastream not providing an option to select all filtered securities at once if there are more than 4,000. Hence, if there are for instance 60,000 corporate bonds in the database in total, one would not be able to select them all at once. Instead, one would have to select the 15 bonds on the current page and then switch to next page $60,000/15 = 4,000$ times. This is of course very cumbersome for the user, and the repeated clicking sounds like a good process to automate programmatically.

There are multiple tools and scripting languages which enable fast and easy click automation. Specifically for this project I decided to go with Python 3 for this purpose, since it was already part of the environment. One of the packages which enable GUI automation in Python is *pyautogui*. With build-in methods like $click()$ and $hotkey()$ it enables the user to simulate mouse clicks on the computer screen by giving the functions the screen coordinates of the buttons. Placing the commands into a loop in the right order makes it possible to simulate the entire process of selecting corporate bonds in Datastream. For a possible Python implementation see Note that the screen coordinates can significantly differ depending on the screen resolution and window settings.

While the described approach solves the problem of selecting all the needed corporate bonds from Datastream, there are two downsides to it. The first one is that it is cumbersome for the developer to determine and to enter the screen coordinates of all the buttons involved. The second is that, even when fully automated, the tool needs a lot of time to select and return all of the chosen securities if there are many of them. At this point, the second approach, even though it is manual, is both faster and easier to apply.

2.2.2. Manual Identifier Extraction

To extract the needed corporate bond identifiers manually, we can make use of the fact that Datastream allows to select all filtered securities at once when there are less than 4,000. Because of this, we can simply split our entire data into multiple chunks that are all smaller than 4,000 bonds in total. This can be done by selecting one or more parameters (the number depends on the size of the dataset) according to which the bonds can be filtered even further. For example, an entire bond dataset with 60,000 bonds in total can be split by coupon size first, and then additionally by currency to produce bond batches of maximum 4,000 bonds each. For a visualization of this approach, see Fig. If the split has been done properly, it will include all of the desired securities, since each bond belongs to one particular coupon size as well as currency group. In most cases, there will be only few groups that need to be extracted from the interface. In the case of 60,000 bonds, one would only have $60,000/4,000 = 15$ groups in total. In reality, for this concrete use case, 19 different bond groups had to be created. This is because not all splits are perfect, and some of them just consist of 3,500 bonds instead of 4,000 for example. After the splitting work has been done, the resulting bond groups can be extracted manually with just a few clicks.

2.3. Automating the Request Table

After the bond identifiers in form of *dscd* codes have been extracted, they can be used to retrieve both static and time series data from Datastream. For this purpose, I wrote a VBA program which fully automates the download process. The only input needed from the user is the *dscd* identifiers, the desired variable codes (such as price, issued volume, etc.) as well as the desired time frames in the case of a time series request. Since the program code is rather complex, I will only cover the main approach briefly. A more in-depth description can be found in the appendix to this work.

At the beginning, the VBA tool retrieves the user-provided identifiers and datatypes

from the respective Excel files. Then, based on the input size, multiple calculations take place. For static information requests not much needs to be done, since these are usually relatively small and only depend on the number of securities for which the data is requested. For time series requests though, the tool estimates how large the entire request will get, depending on dates window, frequency of time points (e.g. daily or quarterly), the number of datatypes, and the number of identifiers. If the request is too large to be processed by Datastream in one run, it gets split into multiple smaller requests of equal size. Since there is no particular metric to estimate in advance whether a particular request will be executed by Datastream, or whether it is too large for that, the tool only computes an approximation based on an empirically measured *Bytes per Field* metric. The single requests then get entered into the request table one below the other, and each receive an own Excel file as destination to store the data.

As soon as the request table has been filled (which does not take long), the command to process the first request is issued to Datastream. This happens via a call to the single available function, which tells Datastream to process the current request table in a black-box manner. After the request ends, the tool checks whether the requested data has arrived to the destination file. If not, it checks the connection of the Datastream add-in and issues a warning to the user if the add-in unexpectedly disconnected. In both cases, the result of the request is logged, in order for the user to be able to read up on the proceedings later. To prevent the computer from sleeping or going in idle mode, the tool moves the computer mouse pointer after each request with a dedicated VBA function. When the first request of the request table has been processed, the other ones get executed in the same manner one after the other. Note that while it is possible to submit the execution for all requests at once, it is not advisable, since Datastream might issue an error due to the data being too large, or might otherwise simply hang during execution. This is exactly the reason why we had to split up the original request in multiple parts in the first place.

While the entire Datastream Extraction Tool is much more complex than what has been described here, the given explanation covers the most crucial parts of the download process. At this point, note that the error monitoring step, which was previously mentioned as essential, cannot be completed in VBA due to its single-threaded execution engine. Section 2.6 will cover the required workaround for this functionality.

2.4. User Interface

In order to provide a graphical user interface as well as an error monitoring capacity (section 2.6) for the created VBA tool, a Python 3 wrapper program has been created. It's architecture can be seen in Fig. 2.1. At this point, note that a detailed usage manual for the Datastream Extraction Tool – including its user interface – can be found in the appendix to this work.

As shown in the visualization, the Python program has a *GUI* component, which runs on its main thread (Thread 1). The layout of the user interface is depicted in Fig. 2.2.

Its features include:

- Request type selection (static or time series)
- Time frames and frequency for time series requests

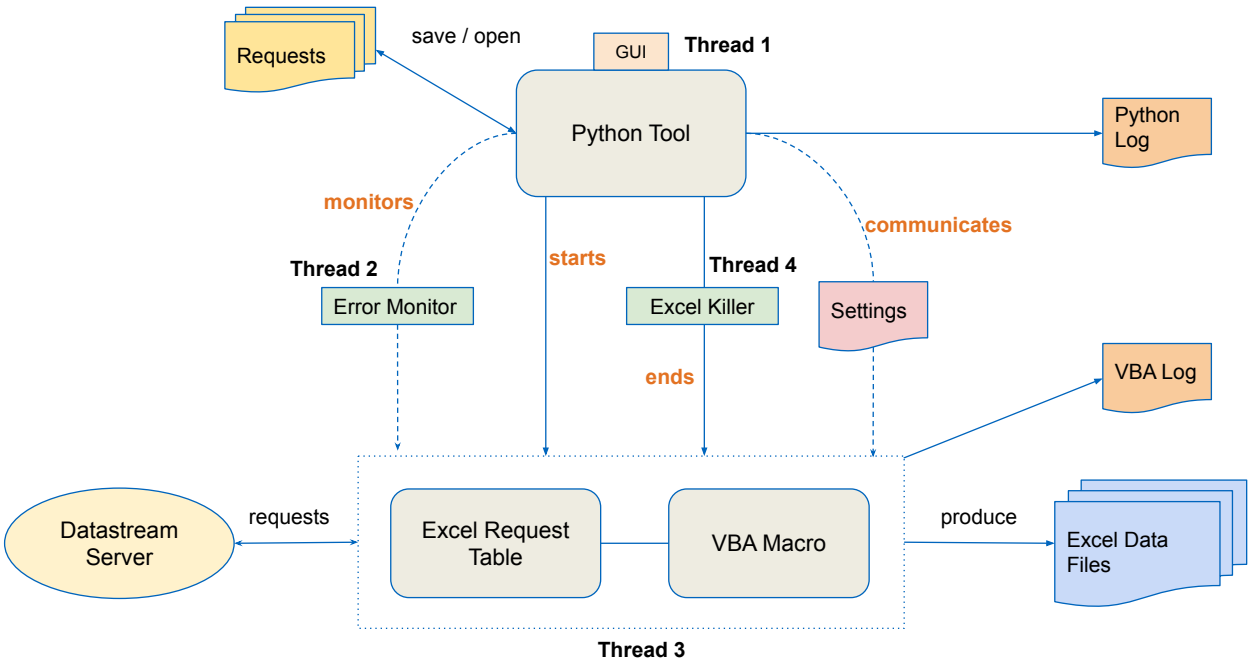


Figure 2.1.: Architecture of the Datastream Extraction Tool

- Request datatypes, which can be entered either in a manual list or via an Excel file
- Data identifiers, which can also be entered either manually or via Excel file
- Request format with one or more Datastream request options (e.g. header row, currency, etc.)
- Destination choice for downloaded data
- Saving and reusing frequently entered requests

The user interface has been programmed using the *Tkinter*[1] package, which enables rudimentary container-based gui creation.

2.5. Other Functionality

Besides the user interface, the Python wrapper offers a logging facility for the actions taken, as well as a functionality to exchange settings and messages with the VBA component. For the latter purpose, a *settings.txt* file is provided which encompasses user-provided request details to forward the request to the VBA program. The entries are made in a key-value manner, which enables a fast and easy information exchange between the Python and the VBA parts. Besides request forwarding, the settings file is used by the Python tool to receive regular updates from the VBA on the current download status. This

[1] `https://docs.python.org/3/library/tkinter.html`

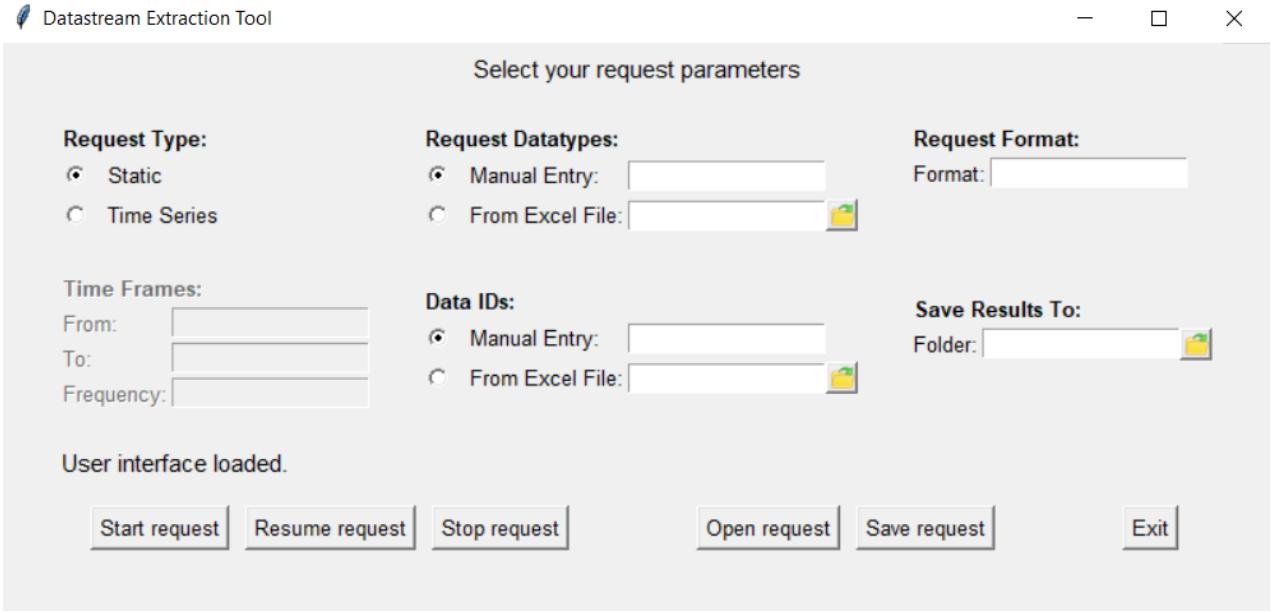


Figure 2.2.: Graphical User Interface of the Datastream Extraction Tool

information is used for both status updates within the gui as well as for the error monitoring functionality which will be explained in section 2.6.

2.6. Error Monitor

As shown in the visualization, the *Error Monitor* module runs in a separate thread (Thread 2), since Python allows execution on multiple threads simultaneously. It is started from the main thread, which controls the graphical user interface, whenever a new download request (Thread 3) has been issued to Datastream. The module maintains an internal counter which signifies the waiting time for the current request in Datastream to process. The counter gets increased each time the Python tool notices that the currently processed request in VBA has not yet changed to the next one. VBA, in its turn, keeps posting updates on the number of the request that is currently being downloaded to the settings file. The ping frequency of the error monitor can be adjusted to the user needs, and is currently set to 1 minute cycle time.

Whenever the counter of the error monitor reaches a programmer-defined threshold (currently 25 minutes), another component of the Python wrapper, the *Excel Killer* (Thread 4) comes into play. It issues an operating-system-level command to (violently) terminate the currently running Excel process. The reason why this has to be done violently is that Excel becomes entirely non-responsive whenever the Datastream download is hanging. Therefore, asking Excel to exit nicely does not result in Excel shutting down. The terminate request needs to run on a separate thread so as to not interfere with the Python gui and status updater.

After the current Excel process and thus also the running Datastream download have been shut down, the Python tool restarts the download request from the same point where

it finished its download before it started hanging. In other words, the download request gets automatically resumed.

Another functionality which the error monitor provides is the ability to detect when all needed data has been downloaded. This happens in a manner similar to the error monitoring itself. The tool simply reads in the number of the last executed download request from the settings file. Based on this information and on the total number of requests to be executed, it determines when the last data batch has been downloaded and ends the download. A corresponding status message is delivered to the user interface to notify the user that the download has finished.

3. Data Cleaning and Fomatting

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) [15] 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 [15]; 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 [15].

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 **??**), 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 **??** and **??**.

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 1.

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 1 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*.

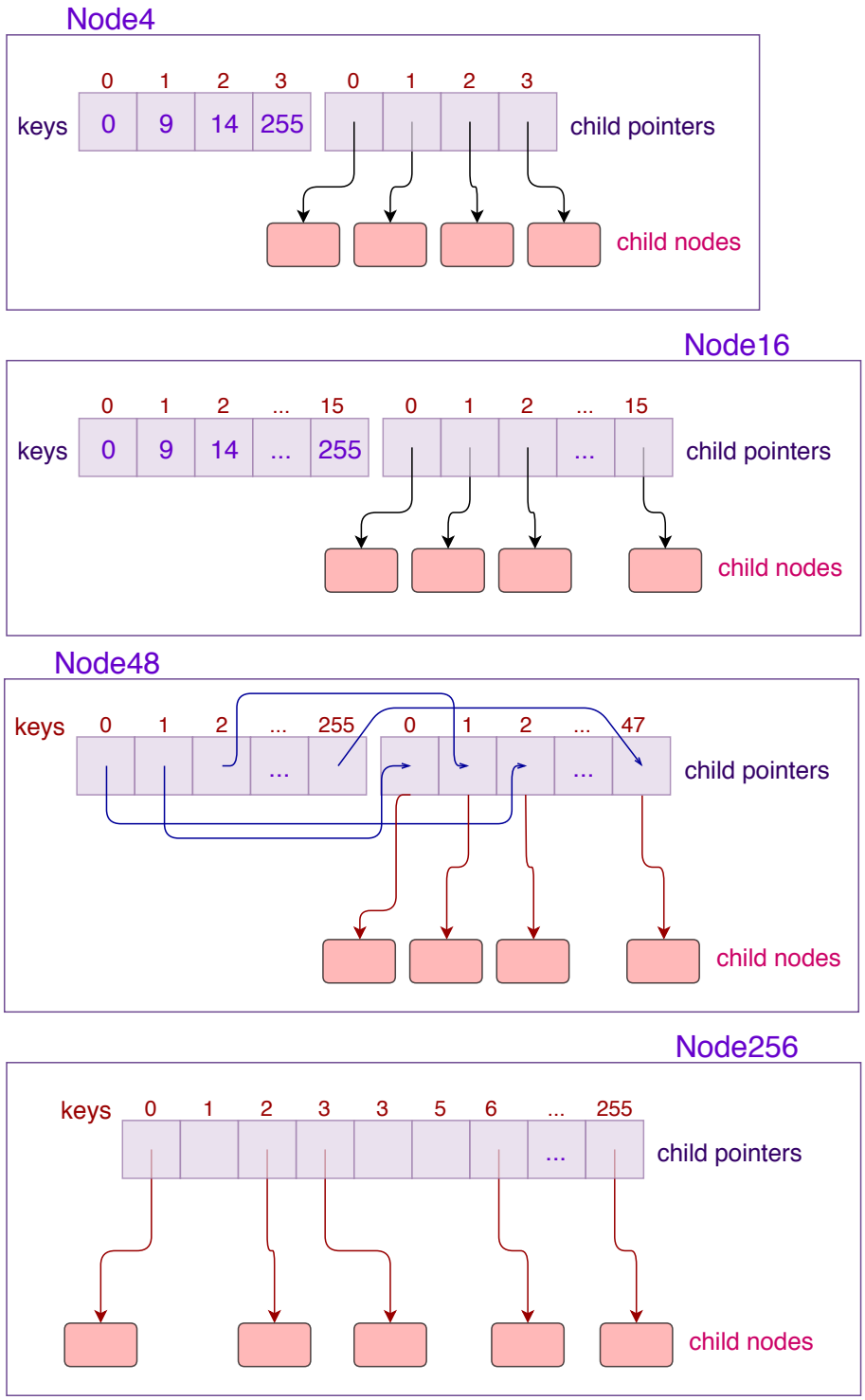


Figure 3.1.: Node Types of the ART Tree

4. Matching

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

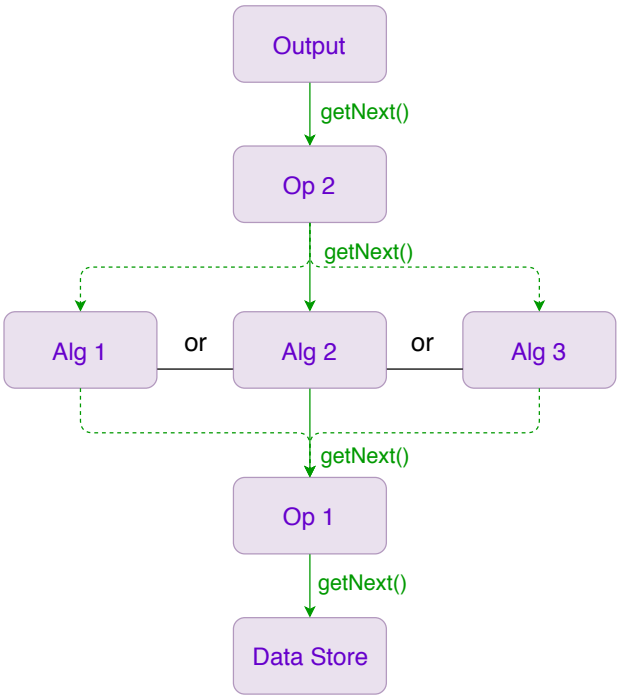


Figure 4.1.: Query Pipelining with Volcano Model

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 2. It features an integration of a sample skyline algorithm into a database system.

Algorithm 2 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)
```

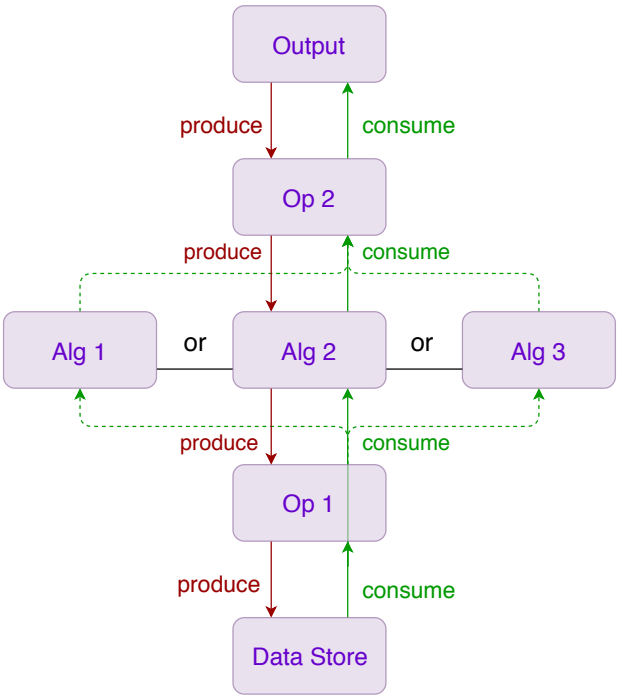


Figure 4.2.: Query Pipelining with Produce/Consume Model

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 3. 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 **??**), 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 3 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 (**??**) 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 4.

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 4 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. **??** for ST-S) to $sub\_results$.

The pseudo-code to the *compute_skyline_subset* operation for parallelized SARTS/ST-S is given in Algorithm 5.

Algorithm 5 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. Statistical Analysis

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

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 [20]. 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 [12]. 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] Ilaria Bartolini, Paolo Ciaccia, and Marco Patella. *Efficient Sort-Based Skyline Evaluation*. 33(4), 11 2008.

[3] OpenMP Architecture Review Board. *OpenMP application programming interface*. `https://www.openmp.org/`. Last Accessed on Sept. 25, 2018.

[4] Kenneth S. Bøgh, Sean Chester, and Ira Assent. *Work-Efficient Parallel Skyline Computation for the GPU*. volume 8, pages 962–973, 2015.

[5] S. Borzsony, D. Kossmann, and K. Stocker. *The Skyline operator*. In *Proceedings 17th International Conference on Data Engineering*, pages 421–430, April 2001.

[6] Sean Chester, Darius Šidlauskas, Ira Assent, and Kenneth 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*, pages 717–719, March 2003.

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

[9] Intel 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] Christos Kalyvas and Theodoros Tzouramanis. *A Survey of Skyline Query Processing*. April 2017.

[11] Alfons Kemper and André Eickler. *Datenbanksysteme: Eine Einfuehrung*. De Gruyter Oldenbourg, 2015.

[12] Alfons Kemper and Thomas Neumann. *HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots*. In *IEEE 27th International Conference on Data Engineering*, pages 195–206, Hannover, Germany, April 2011. IEEE.

[13] Donald Kossmann, Frank Ramsak, and Steffen Rost. *Shooting Stars in the Sky: An Online Algorithm for Skyline Queries*. In *Proceedings of the 28th VLDB Conference*, pages 275–286, Hong Kong, China, August 2002.

[14] 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*, volume 22, pages 469–476, October 1975.

[15] Viktor Leis, Alfons Kemper, and Thomas 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, pages 38–49, Washington, DC, USA, 2013. IEEE Computer Society.

[16] Stian Liknes, Akrivi Vlachou, Christos Doulkeridis, and Kjetil Nørvåg. *APSkyline: Improved Skyline Computation for Multicore Architectures*, volume 8421 of *Lecture Notes in Computer Science*, pages 312–326. Springer International Publishing, Switzerland, March 2014.

[17] Kasper Mullesgaard, Jens Laurits Pedersen, Hua Lu, and Yongluan Zhou. *Efficient Skyline Computation in MapReduce*. In *Proc. 17th International Conference on Extending Database Technology*, pages 37–48, Athens, Greece, March 2014.

[18] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. *An Optimal and Progressive Algorithm for Skyline Queries*. In *ACM Proceedings*, San Diego, CA, USA, June 2003.

[19] Sungwoo Park, Taekyung Kim, Jonghyun Park, Jinha Kim, and Hyeonseung Im. *Parallel Skyline Computation on Multicore Architectures*. In *2009 IEEE 25th International Conference on Data Engineering*, pages 760–771, Shanghai, China, March 2009.

[20] Md Farhadur Rahman, Abolfazl Asudeh, Nick Koudas, and Gautam Das. *Efficient Computation of Subspace Skyline over Categorical Domains*. In *ACM Proceedings*, pages 407–416, Singapur, November 2017. CIKM. Session 2E: Skyline Queries.

[21] Kian-Lee Tan, Pin-Kwang Eng, and Beng Chin Ooi. *Efficient Progressive Skyline Computation*. In *Proc. 27th International Conference on Very Large Data Bases*, Roma, Italy, September 2001.