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 blackbox 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

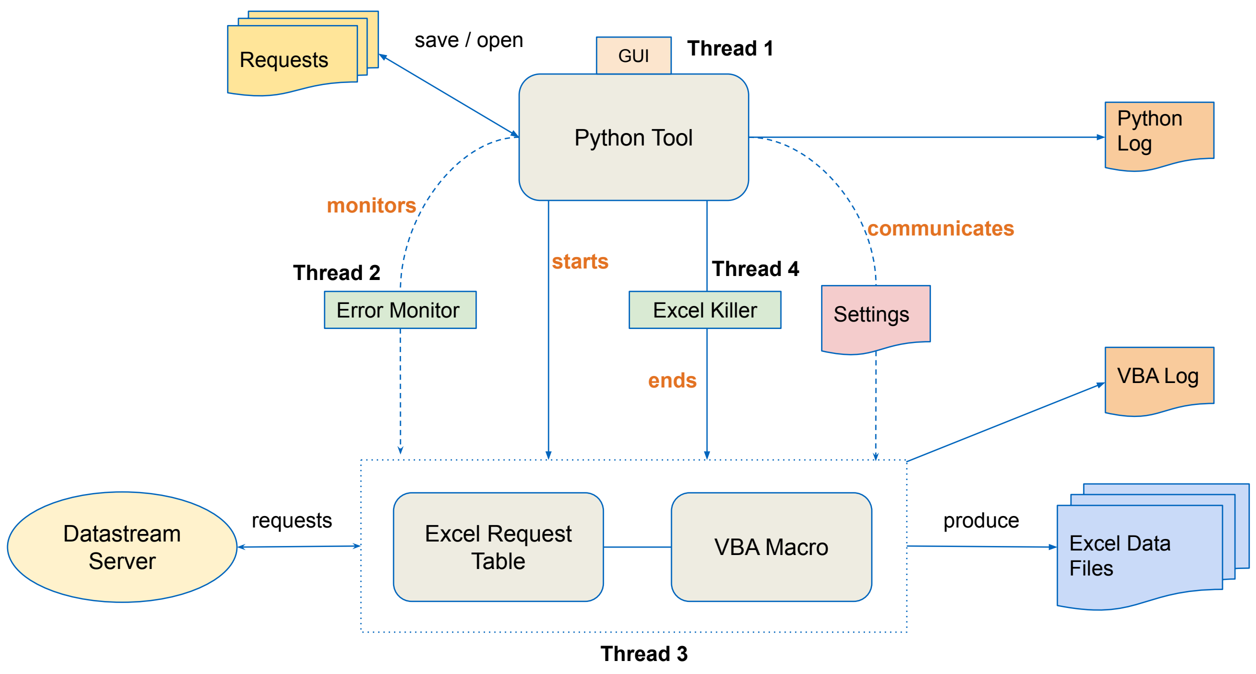


Figure 2.1.: Architecture of the Datastream Extraction Tool

- 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`

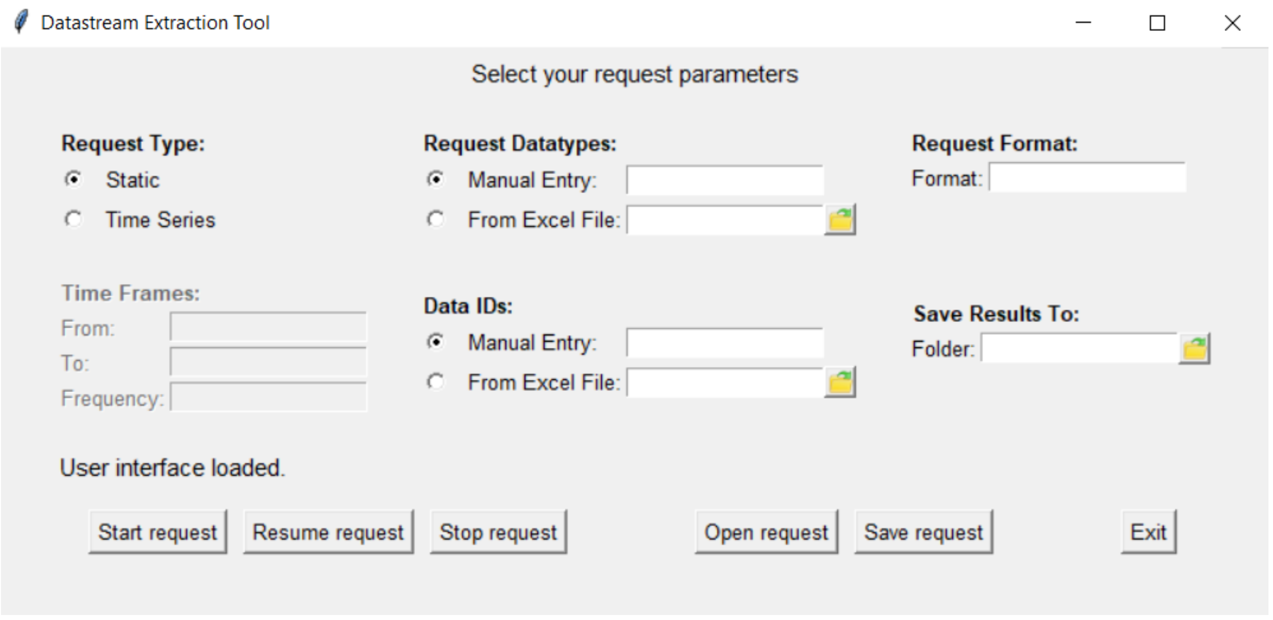


Figure 2.2.: Graphical User Interface of the Datastream Extraction Tool

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.

The entire Datastream Extraction Tool consisting of the VBA and Python parts, and including the required folder structure, is currently hosted on Google Drive under `https://drive.google.com/drive/u/0/folders/1YEgJ-PsgIuMResgep9CxUnMGHyDT0Rlm`.

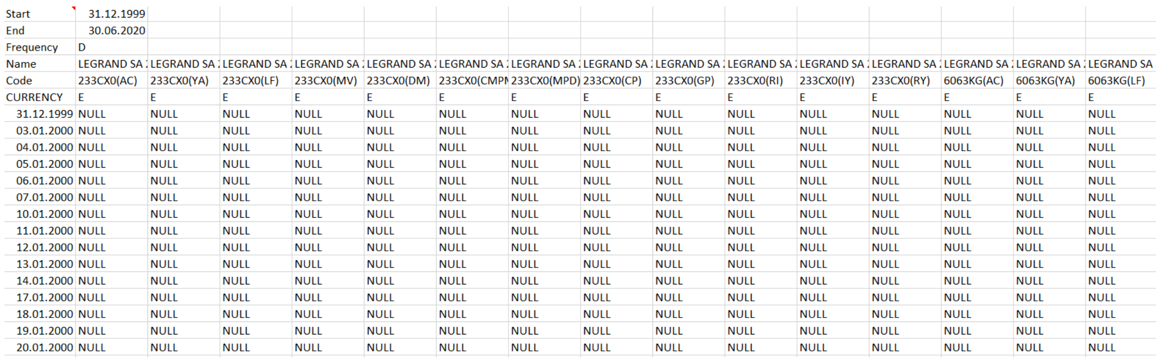
3. Data Preparation

As it is often the case with data science projects, the data preparation part is rather tedious. After the static and time series bond data has been extracted from Datastream – which will likely take several days – it is available in form of multiple data parts in Excel format. Since it is more convenient to perform further statistical analysis in a dedicated statistical environment, such as Stata or MatLab, the data needs to be brought into 'long' format, and additionally to be cleaned from null entries and outliers. The undertaken procedures are described in the following.

3.1. Data Formatting

For the static bond data, there is not much to be done in terms of formatting. Its original format, as downloaded from Datastream, is mostly suitable for further analysis and can be directly imported into Stata. Stata is the statistical software utilized in the scope of this projects and fits well for most of the analysis tasks.

The downloaded time series data is initially in 'wide' format and has multiple bonds in one row. It looks like shown in Fig. 3.1.

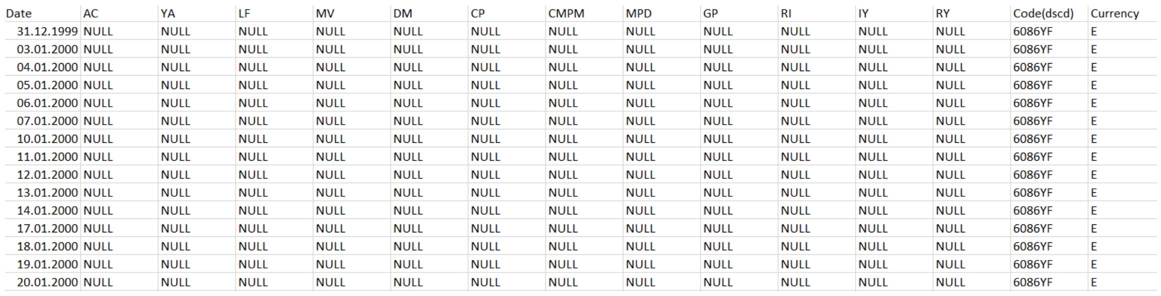


Start	31.12.1999														
End	30.06.2020														
Frequency	D														
Name	LEGRAND SA	LEGRAND SA	LEGRAND SA	LEGRAND SA	LEGRAND SA	LEGRAND SA	LEGRAND SA	LEGRAND SA	LEGRAND SA	LEGRAND SA	LEGRAND SA	LEGRAND SA	LEGRAND SA	LEGRAND SA	LEGRAND SA
Code	233CX0(AC)	233CX0(YA)	233CX0(LF)	233CX0(MV)	233CX0(DM)	233CX0(CMPI	233CX0(MPD)	233CX0(CP)	233CX0(GP)	233CX0(RI)	233CX0(IY)	233CX0(RY)	6063KG(AC)	6063KG(YA)	6063KG(LF)
CURRENCY	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
31.12.1999	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
03.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
04.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
05.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
06.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
07.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
10.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
11.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
12.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
13.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
14.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
17.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
18.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
19.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
20.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Figure 3.1.: Sample of downloaded raw time series bond data

The goal is to transform this time series data into 'long' format, as can be seen in Fig. 3.2, by saving the bonds one below the other. Additionally, the header has to be removed, and the *dscd* identifier of each bond as well as its currency have to be entered as a separate column for each date instead of being at the top.

I wrote a VBA macro with a function called *ToLongFormat()* which accomplishes the described task. While Stata might have also been able to do the formatting, I decided to work with VBA at this point, since it is native to the MS Excel environment. While the entire code can be found in the appendix, the main procedure is as follows:



Date	AC	YA	LF	MV	DM	CP	CMPM	MPD	GP	RI	IY	RY	Code(dscd)	Currency
31.12.1999	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	6086YF	E
03.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	6086YF	E
04.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	6086YF	E
05.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	6086YF	E
06.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	6086YF	E
07.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	6086YF	E
10.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	6086YF	E
11.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	6086YF	E
12.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	6086YF	E
13.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	6086YF	E
14.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	6086YF	E
17.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	6086YF	E
18.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	6086YF	E
19.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	6086YF	E
20.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	6086YF	E

Figure 3.2.: Sample of downloaded time series bond data in 'long' format

1. Define data layout constants depending on the initial format: header height, number of time stamps, number of datatypes, bonds per block, and number of blocks. A block is defined as all time stamps for multiple bonds which are located row-wise next to each other. An example can be seen in Fig. 3.1 where two bonds from the same firm, but with different *dscd* codes, are next to each other. There can be multiple such blocks one below the other in one Excel file, depending on how the data was downloaded.

2. For all data files (as there will be multiple for larger requests) and for all blocks within each file, remove the header rows, place bonds one below the other, and create columns for *dscd* and currency.

3. Add a newly created header row once at the beginning of the file. This header row can later be used to define variable names in statistical software.

Note that if the original Excel files were very large, i.e. with a high amount of securities or dates, Excel might reach its sheet length limit when running this macro. If you notice such behavior, there is another function shipped with this macro, called *SplitInSubfiles()*. You can use this function on your initial downloaded Excel data to reshape it into smaller-sized files, before transforming it to 'long' format.

3.2. Data Cleaning

As soon as the data has been formatted, it can be cleaned conveniently within statistical software. Since I am working with Stata, the Excel files with the reshaped time series data can be simply imported to Stata with the *import excel* Stata command. It is possible to save frequently used Stata scripts in form of *.do* files to reuse them later. You can find all the do-files in the appendix attached to this project. The file do_import_excel can be found in .

4. Matching

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

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