Efficient Processing of Large-scale Spatio-temporal Data

Dissertation
zur Erlangung des akademischen Grades

Doktor–Ingenieur (Dr.–Ing.)

vorgelegt der

TECHNISCHEN UNIVERSITÄT ILMENAU
Fakultät für Informatik und Automatisierung

von

Dipl.-Inf. Stefan Hagedorn

Gutachter:

1. Prof. Dr.–Ing. habil. Kai-Uwe Sattler
2. Prof. Dr.–Ing. habil. Bernhard Seeger
3. Prof. Dr. rer. nat. Michael Gertz

Tag der Einreichung: 06. November 2019
Tag der wissenschaftlichen Aussprache: 06. Mai 2020

urn:nbn:de:gbv:ilm1-2020000132

Abstract

Millions of location-aware devices, such as mobile phones, cars, and environmental sensors constantly report their positions often in combination with a timestamp to a server for different kinds of analyses. While the location information of the devices and reported events is represented as points and polygons, raster data is another type of spatial data, which is for example produced by cameras and sensors. This Big spatio-temporal Data needs to be processed on scalable platforms, such as Hadoop and Apache Spark. However, these platforms are unaware of, e.g., spatial neighborhood, making them practically impossible to use for this kind of data.

Furthermore, the repeated executions of the programs during development and also by different users result in long execution times and potentially high costs in rented clusters. These costs can be reduced by reusing commonly computed intermediate results.

Within this thesis, we tackle the two challenges described above. First, we present the STARK framework for processing spatio-temporal vector and raster data on the Apache Spark stack. For operators, we identify several possible algorithms and study how they can benefit from the underlying platform's properties. We further investigate how indexes can be realized in the distributed and parallel architecture of Big Data processing engines and compare methods for data partitioning, which perform differently well with respect to data skew and data set size. Furthermore, an approach to reduce the amount of data to process at operator level is presented. In order to reduce the execution times, we introduce an approach to transparently recycle intermediate results of dataflow programs, based on operator costs. To compute the costs, we instrument the programs with profiling code to gather the execution time and result size of the operators.

In the evaluation we first compare the various implementation and configuration possibilities in STARK and identify scenarios when and how partitioning and indexing should be applied. We further compare STARK to related systems and show that we can achieve significantly better execution times, not only when exploiting existing partitioning information. In the second part of the evaluation we show that with the transparent cost-based materialization and recycling of intermediate results the execution times of programs can be reduced significantly.

Zusammenfassung

Millionen Geräte, wie z.B. Mobiltelefone, Autos und Umweltsensoren senden ihre Positionen zusammen mit einem Zeitstempel und weiteren Nutzdaten an einen Server zu verschiedenen Analysezwecken. Die Positionsinformationen und übertragenen Ereignisinformationen werden als Punkte oder Polygone dargestellt. Eine weitere Art räumlicher Daten sind Rasterdaten, die zum Beispiel von Kameras und Sensoren produziert werden. Diese großen räumlich-zeitlichen Datenmengen können nur auf skalierbaren Plattformen wie Hadoop und Apache Spark verarbeitet werden, die jedoch z.B. die Nachbarschaftsinformation nicht ausnutzen können – was die Ausführung bestimmter Anfragen praktisch unmöglich macht.

Die wiederholten Ausführungen der Analyseprogramme während ihrer Entwicklung und durch verschiedene Nutzer resultieren in langen Ausführungszeiten und hohen Kosten für gemietete Ressourcen, die durch die Wiederverwendung von Zwischenergebnissen reduziert werden können.

Diese Arbeit beschäftigt sich mit den beiden oben beschriebenen Herausforderungen. Wir präsentieren zunächst das STARK Framework für die Verarbeitung räumlich-zeitlicher Vektor- und Rasterdaten in Apache Spark. Wir identifizieren verschiedene Algorithmen für Operatoren und analysieren, wie diese von den Eigenschaften der zugrundeliegenden Plattform profitieren können. Weiterhin wird untersucht, wie Indexe in der verteilten und parallelen Umgebung realisiert werden können. Außerdem vergleichen wir Partitionierungsmethoden, die unterschiedlich gut mit ungleichmäßiger Datenverteilung und der Größe der Datenmenge umgehen können und präsentieren einen Ansatz um die auf Operatorebene zu verarbeitende Datenmenge frühzeitig zu reduzieren. Um die Ausführungszeit von Programmen zu verkürzen, stellen wir einen Ansatz zur transparenten Materialisierung von Zwischenergebnissen vor. Dieser Ansatz benutzt ein Entscheidungsmodell, welches auf den tatsächlichen Operatorkosten basiert.

In der Evaluierung vergleichen wir die verschiedenen Implementierungs- sowie Konfigurationsmöglichkeiten in STARK und identifizieren Szenarien wann Partitionierung und Indexierung eingesetzt werden sollten. Außerdem vergleichen wir STARK mit verwandten Systemen. Im zweiten Teil der Evaluierung zeigen wir, dass die transparente Wiederverwendung der materialisierten Zwischenergebnisse die Ausführungszeit der Programme signifikant verringern kann.

Danksagung

Es gibt viele Menschen bei denen ich mich an dieser Stelle für ihre Unterstützung während dieser Arbeit bedanken möchte.

Zuerst möchte ich meinem Betreuer Kai-Uwe Sattler danken. Nachdem ich nach dem Diplom die Uni zunächst verlassen hatte, gab er mir die Möglichkeit wieder zurück zu kommen und in verschiedenen spannenden Projekten zu arbeiten. So konnte ich im Laufe der Zeit wichtige Erfahrungen in vielen verschiedenen Bereichen sammeln. Letztendlich wäre diese Arbeit ohne seine Ratschläge, Motivierungen und vor allem Geduld sicherlich niemals möglich gewesen.

Außerdem möchte ich meinen Gutachtern Bernhard Seeger und Michael Gertz danken, die sich Zeit für Hinweise zur Arbeit sowie für die Erstellung der Gutachten genommen haben.

Natürlich möchte ich mich auch bei all meinen Kollegen bedanken, mit denen ich in den letzten Jahren zusammenarbeiten durfte. Dies sind vor allem Felix, Stephan und Philipp mit denen ich das Büro geteilt habe, aber auch all die anderen mit denen ich immer Probleme und Ideen diskutieren konnte und bei denen ich Unterstützung und Aufmunterung für die oftmals langen Nächte im Büro vor den Deadlines fand. Außerdem danke ich allen Studierenden, die mit ihren Bachelor-, Master- und anderen Studienarbeiten einen wichtigen Teil zum Gelingen der Arbeit beigetragen haben.

Ein ganz besonderer Dank gilt meiner Frau Manuela und meinen Kindern Henri und Theo. Ihr habt mir den nötigen Rückhalt gegeben, mich aufgemuntert und aufgebaut und Verständnis gehabt, wenn ich – vor allem in den Wochen vor der Fertigstellung der Arbeit – wenig Zeit für euch hatte und auch an den Wochenenden viel mehr Zeit im Büro als zu Hause verbracht habe.

Auch meinen Eltern möchte ich hier danken. Ihr habt mich und uns immer bedingungslos unterstützt und damit alles erst möglich gemacht. Dafür bin ich euch unendlich dankbar!

Contents

1 Introduction **11**
- 1.1 Motivation . . . 12
- 1.2 Problem Statement and Contribution . . . 13
- 1.3 Outline . . . 15

2 Use Cases Examples **17**
- 2.1 Earth Observation & Environmental Monitoring . . . 17
- 2.2 Event Correlation . . . 18
- 2.3 Requirement Analysis . . . 19
  - 2.3.1 Storage Level . . . 20
  - 2.3.2 Data Model . . . 21
  - 2.3.3 Query Model . . . 22

3 Foundations **25**
- 3.1 Spatial & Temporal Data Processing . . . 25
  - 3.1.1 Data Types and Operations . . . 25
  - 3.1.2 Geometric and Geodetic Computations . . . 31
  - 3.1.3 Indexing Spatio-temporal Data . . . 33
  - 3.1.4 Indexes for Raster Data . . . 36
- 3.2 Big Data Processing Platforms . . . 36
  - 3.2.1 Hadoop MapReduce . . . 37
  - 3.2.2 Apache Spark . . . 39
  - 3.2.3 Other Platforms . . . 42

4 Related Work **45**
- 4.1 Hadoop MapReduce-based systems . . . 46
- 4.2 Apache Spark-based systems . . . 47

5 Framework Design **51**
- 5.1 Storage Level . . . 51
  - 5.1.1 Storage Formats . . . 51
  - 5.1.2 Near Data Processing . . . 52
- 5.2 Data Model . . . 54
  - 5.2.1 Data Types . . . 54
  - 5.2.2 Spatio-temporal Partitioning . . . 56
  - 5.2.3 Indexing . . . 62
- 5.3 Query Model . . . 64

5.3.1 Operations for Spatio-temporal Vector and Raster Data 64
5.3.2 Spatio-temporal Vector and Raster Operators 66
5.3.3 Distance Functions 78
5.3.4 Language Integration 79
5.4 Conclusion 79

6 Prototype Implementation for Apache Spark 81
6.1 STARK Architecture in Detail 81
6.2 Adding Types for Vector & Raster Data 81
6.3 Basic Operations 83
6.4 Operations for Data Analytics 85
6.5 Spatial and Temporal Partitioning 86
6.6 Indexes 86
6.7 Spatio-temporal Spark Context 87

7 Declarative Language Support 89
7.1 SparkSQL Integration 89
7.1.1 Data Type Registration 89
7.1.2 User Defined Functions 89
7.2 Piglet 91
7.2.1 Architecture 91
7.2.2 Language Extensions 92
7.2.3 Spatio-temporal Types and Operations 93

8 Reusing Intermediate Results in Dataflow Programs 95
8.1 Opportunities for Reusing 96
8.2 Related Work 98
8.3 A Cost-based Decision Model 99
8.3.1 Foundations 99
8.3.2 Decision Model 103
8.4 Profiling Dataflow Programs in Piglet 106

9 Performance Evaluation 109
9.1 Impact of Spatio-temporal Partitioning and Indexing 110
9.1.1 Partitioning 110
9.1.2 Indexing 119
9.1.3 Conclusion 122
9.2 Comparing STARK with Related Systems 123
9.2.1 Spatial Range Queries 123
9.2.2 k Nearest Neighbor Search 124
9.2.3 Join Processing 126
9.2.4 Raster Data Processing 126
9.2.5 Conclusion 127
9.3 Reusing Intermediate Results 128

10 Conclusion & Open Research Questions **131**
10.1 Conclusion 131
10.2 Open Research Questions 133

Appendices **137**
A More k Nearest Neighbor Search Results 137
B More Examples of Piglet Scripts 140

Chapter 1
Introduction

Space agencies of many nations operate satellites in space that monitor the earth. Researchers can use their recordings to, e. g., investigate the impact of climate change, measure the global urban footprint, and monitor the extent of rain forests, deserts, and the rise of the sea level.

The DLR (Deutsches Zentrum für Luft- und Raumfahrt, engl.: German Aerospace Center) publishes images taken by the Sentinel 2 satellite mission (and many others). The satellite takes pictures of the earth using optical sensors. These images are available in raster data format and researchers use them for various tasks.

Archaeologists can use the high resolution images of global scale to find traces of ancient cities or ruins, even though they are buried underground. Using time series analysis methodologies, they can also use the raster images and recordings from other sensors to re-construct the cutback of the rain forests over time. In July 2018 the project ICARUS started its operational phase to track animals' movements from space using sensors mounted on the International Space Station with the goal to predict natural disasters such as volcano eruptions and earthquakes[1].

The NPMM200[2] at the TU Ilmenau can be used to take high resolution images of, e. g., wafers. Depending on camera resolution and magnification, data sets up to 17 TB per object are produced [10, 54], which cannot be handled by single node database systems anymore. The images are stored in raster data format and because of their large size, scalable processing operators are needed. Besides error detection, researchers use the high resolution images to find regions of interest to extract and inspect with other software. Objects examined with such setups include wafers, micro-mechanical objects, and optical assemblies, such as lenses. Often the goal is to find errors produced during the making process or to analyze the material's properties.

However, not only images and sensor values are important spatial data. Nowadays, almost every mobile device and modern car is equipped with a Global Positioning System (GPS) device that can also report its position to the manufacturer or some other company. Especially for car rental and car sharing companies the current position information as well as its course over time can be useful to understand where more cars are needed and how the fleet can be optimized. Another example are agricultural machines which also often include a GPS system and report their position. This information is used to automatically identify the borders of farmland and cultivated areas. The gained information can be utilized for better route planning and autonomous driving of such machines. Software platforms used to preserve and analyze this captivating but costly data must be able to handle especially the spatial and temporal information efficiently.

[1] `https://icarusinitiative.org/`

[2] Nano Positioning and Measurement Machine

Database systems have been around for decades and are heavily used in all kinds of software to store and retrieve the valuable information. With their advance and wide-spread acceptance since the 1970s, researchers have focused on developing new and optimizing existing storage schemes, access methods, and query operator implementations.
The spatial, temporal and, of course, their combination – spatio-temporal – data greatly benefit from being treated specifically. Database Management Systems (DBMSs) usually are implemented in a general fashion so that the internal processing is data type agnostic, i. e., a filter operator can be applied on any attribute and a join operator can combine relations on any join attribute type. However, while basic attribute types are being processed using predicates such as equal (=) or less than (<) spatial and temporal attributes need another set of functions and predicates. Typical predicates on spatial data are *intersects* or *contains*, expressing that one shape somehow intersects or completely contains another shape. For temporal data, *before* and *after* are common predicates. Evaluating these predicates can be expensive and DBMSs started to extend their portfolio of basic data types and operators to represent spatial and temporal data objects along with efficient algorithms for processing.

Besides implementing spatial operators in DBMSs, specialized programs were introduced to analyze spatial data. So called Geographic Information Systems (GISs) support various formats used to store spatial objects and provide users with a rich set of operators to inspect their data sets. Often, GIS applications visualize the processed data on maps and are able to add layers from other data sources, too. Researchers, management, and authorities so are able to visually identify patterns or are supported in their decision making process.

1.1 Motivation

In the late 2000s, a new trend arose among researchers and companies of all sizes: Big Data and data science. As the price for storage dropped, more and more data was collected and analyzed with the goal for companies to understand their customers or predict future trends and opportunities. The increased amount of data stored soon exceeded the processing capabilities of a single machine and distributed and parallel systems to process them in acceptable time were required. In [28], Google presented their implementation of the MapReduce paradigm that allowed batch processing of large document collections in parallel on clusters of commodity computers. The notion of MapReduce was adopted in several frameworks, like the open source Hadoop MapReduce[3]. Although Hadoop scales very well with the number of nodes in a cluster (cf. Section 3.2.1), it natively supports basic data types only (Strings, integer, ...). Thus, processing spatial or spatio-temporal data must be achieved with high programming efforts by the users, who have to implement the spatial predicates on their own. Indexes, a standard way to speed up execution in DBMS, are neither available in Hadoop, nor is any other form of optimization for spatial and temporal data characteristics. However, the native support for spatial

[3] `https://hadoop.apache.org/`

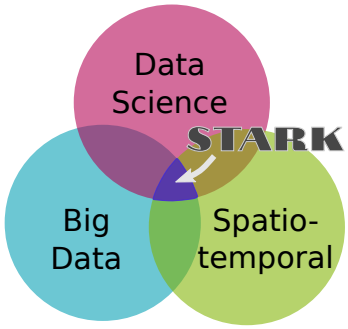


Figure 1.1: Classification of this work in context of research areas.

and temporal data in these Big Data platforms is required by many use cases, as will be described below. Even though the spatial or temporal data set itself may be small enough to be processed on a single machine, in some use cases it may be necessary to combine it with some other extremely large data set using e. g., a spatial join so that it has to be performed in a parallel setup. This may be the case if users need to find the name of a region, e. g., country for position information from where a Twitter post was sent: the data set containing the regions is rather small compared to the data set with posts from a long time range.

Furthermore, there are several scenarios where (parts of) the same program(s) are run repeatedly: for testing reasons during development, due to the incremental nature of data exploration, execution of the same program with different input parameters, or simply multiple users running the same program simultaneously. Because of these repetitions, many potentially expensive operators are executed over and over again, computing the same result every time and wasting valuable resources. Often, this development and data analysis happens in notebook systems like Jupyter[4] or Apache Zeppelin[5]. Here, users run their queries and programs interactively and expect instant results.

While a framework will offer the required operations with an efficient implementation, usability and low entrance barrier are important for solving the actual analyses tasks. Therefore, declarative languages that can be translated and optimized are needed.

1.2 Problem Statement and Contribution

In the previous section some use case scenarios were introduced that depict the necessity of spatial and spatio-temporal operators for processing large data sets in a cluster environment. Figure 1.1 shows a classification of the work in this thesis, whose result is a framework called *STARK* (Spatio-temporal Data Analytics on Spark), in the context of other research areas. With this work, we aim to create a framework that combines Data Science tasks with spatio-temporal data processing on large data sets using Big Data technologies.

Recently, a few frameworks were proposed that partially provide the required

[4] `https://jupyter.org/`
[5] `https://zeppelin.apache.org/`

operators and solve some of the identified problems. However, these systems mostly focus on spatial vector data only and do not include any extension (data types or operators) neither for spatio-temporal nor raster data. Furthermore, these systems include partitioning strategies that look at spatial features only so that data sets cannot be partitioned on their temporal feature.

Besides the lack of support for temporal data, these systems only provide basic operators to perform a spatial filter and spatial join – although many systems only support a very limited set of predicates to apply. However, to perform more sophisticated analyses on the spatio-temporal data sets, operations such as Skyline computation and clustering are needed.

Furthermore, most of these systems do not support raster data and especially not the combination with vector objects so that users can e. g., filter their raster data set for parts that are within a country or another polygonal region.

The main goal of this work is to analyze how spatial, spatio-temporal as well as raster data can be efficiently supported in cluster processing platforms. As the target platform for our prototype implementation we select Apache Spark. For this analysis, we build a framework for spatio-temporal data processing on Apache Spark with the goal to provide fast and scalable operators.

Thus, in the context of this analysis and framework, the main contributions of this work are:

1. **Data Types and Operators for Vector and Raster Processing.** We analyze the required types for data representation as well as operations thereon. The data types are used to model vector and raster data. Furthermore, we analyze different ways to design and implement standard operators such as filters and joins as well as analytical operators such as Skyline, k nearest neighbors, and clustering over vector data.

2. **Efficient Methods to Support Operations.** Especially the partitioning methods are crucial for efficient spatio-temporal data processing as they distribute the workload to the compute nodes and allow operators to utilize the spatial and temporal characteristics of the data. Additional indexing strategies, which are not present in current Big Data processing engines, allow to execute the spatial operators in reasonable time. We study and evaluate different partitioning schemes and analyze their impact along with indexes on query performance.

3. **Recycling Intermediate Results.** To improve the overall performance of the systems when the same operator occurs frequently, we present a cost-based decision model that identifies operators whose result should be materialized to persistent storage. The model includes three different strategies to base its decision on. When the same operator is encountered again in the same or another script, the materialized result can simply be loaded instead of executing the operator and all its predecessors in the program again. Using this cost model, execution times of scripts that share common operations can be reduced from minutes to a few seconds. To supply the decision model with the required information to compute the costs, we introduce an approach to profile operators executed in a data parallel cluster environment.

4. **Declarative Language Support.** Users are assisted in creating their analysis programs by extensions of higher level languages to support the operations included in our framework. These languages are for example the Structured Query Language (SQL) as well as Pig Latin, a dataflow scripting language originally designed to create MapReduce programs.

1.3 Outline

The rest of this thesis is structured as follows. We start by presenting two exemplary use cases of typical spatio-temporal data processing. From these use case examples, we extract requirements for data processing platforms. In the subsequent Chapter 3, we discuss the fundamentals of this work, namely the basics of spatial and temporal data processing as well as the core concepts of the Big Data processing platforms. Chapter 4 presents projects related to the work of this thesis.

In Chapter 5 we present and discuss design decisions for our spatio-temporal data processing framework from bottom up: Section 5.1 deals with storage formats and near data processing opportunities, Section 5.2 presents approaches for data types, spatial and spatio-temporal data partitioning as well as indexing, and in Section 5.3 we finally propose possible concepts for operators upon the data sets and show how they can exploit and benefit from the previous levels. After we have presented the conceptual design of the framework, Chapter 6 sketches important implementation details of the STARK framework. Its support in declarative languages is shown in Chapter 7.

As an improvement of resource utilization in the cluster, in Chapter 8 we present a decision model for recycling intermediate results in dataflow programs.

In Chapter 9 we present the results of our performance evaluation of both, the spatio-temporal data processing framework as well as the decision model. Finally, the thesis closes in Chapter 10 where we present a summary and conclusion of the previous chapters and give an outlook to open research questions.

Chapter 2
Use Cases Examples

Spatial and temporal data can be found in many areas where data is being stored and processed. In the following, we outline two use case examples that show the importance of the support for spatio-temporal vector and raster data processing on Big Data platforms.

2.1 Earth Observation & Environmental Monitoring

Our planet is surrounded by a multitude of satellites that have various purposes. Many of them observe properties of the planet and downstream them to a ground station where all incoming measurements are received, pre-processed, and archived. The improved technology for sensors that allows a fine resolution and the increasing number of satellites orbiting the earth results in a massive amount of data to be stored and processed. As of 2018, the German satellite data archive of the DLR preserves around 20 PB of raw and processed data and it is expected to reach the current capacity of the tape archive of 50 PB at the beginning of the 2020s.

Meta information about the payload files in the archive, such as the spatial and temporal coverage, sensor settings, mission, etc. is stored in a catalog. This catalog is queried to find the actual files in the archive that contain information to answer a current question. Such questions are for example: *Find all temperature recordings from 2001 to 2018 in Germany* or *Find images of vessels in the Baltic Sea on 2018-09-10*. *Germany* and *Baltic Sea* can either be given as polygon boundaries or the polygons need be fetched for the given names.

Since the meta information in the catalog is already large it may be worth processing the queries in parallel in a cluster, rather than on a single database server. The catalog system needs to process spatial, temporal as well as normal filter predicates to find the required files. In order to not having to scan through all entries in the catalog, indexes are required to efficiently check the spatial and temporal predicates from the queries, i. e., if a record is in *Germany* or the *Baltic Sea* or in the time range of *2001 to 2018* or on *2018-09-10*, respectively. A catalog system using data parallel execution in a cluster could reduce the time being spent scanning the index and furthermore open the possibilities to directly process the large data files in that cluster environment.

Besides the catalog meta data, earth observation produces even more spatial and temporal data. The DLR as well as NASA and other aerospace agencies run various web platforms where pre-processed data from satellites can be accessed and downloaded for free. Such data is usually provided as raster data, e. g., in Comma Separated Values (CSV), GeoTIFF or other file formats. For example, the CODE-DE[1] project provides access to images taken by the Sentinel 2 satellite mission. The web interface lets users search for regions based on spatial and temporal filters. In the background, from the user provided search constraints, a query is run against the underlying database. When the result is ready, users can download their result as raster images.

Often, this data has to be combined with other data sets which are potentially large and parallel cluster computing is required. With an efficient and scalable system that allows employees and end-users to directly run their analyses in a provided cluster, the amount of data that has to actually be transferred over the public internet could be reduced significantly.

There are numerous projects that use imaging and sensors, also mounted on satellites, to monitor the earth's flora, fauna, and environmental parameters. Typically, temperature, rainfall, but also optical image data is represented as raster data, where each pixel of a data set represents one measured value or a pixel in the image, respectively. Interesting questions are for example *What was the maximum amount of precipitation per year in the Sahara over the last decade?*, *List the names of countries with an average temperature above 25° C*, or *Find all ships in this image.* To answer them, the raster data sets containing the temperature and rainfall data must be filtered or joined with a vector object or data set, respectively. The tiles in the raster dataset cover a spatial region which has to be tested against a single query region to find rainfall values in the *Sahara* or be joined with a set of the border definitions (polygons). Not only is the combination of raster and vector data required, but also the efficient execution, as join operations are known to be expensive and therefore time consuming. To find ships in an image, the user extracts a ship from the image and wants to find all similar objects. The reference image is basically a pattern that must be matched against the data set.

2.2 Event Correlation

On the World Wide Web, a plethora of new content is created every second, ranging from short tweets or posts on social media platforms over personal blog posts to professional news articles. One can assume that a large majority of these articles and posts contain references to some kind of event. Such events may be historical incidents such as the beginning of a war or the coronation of a king. But also references to more current news like concerts or sport events are included in these texts. Besides the explicit mentioning of the time and location of occurrence of an event in the text, the published resources themselves often inherently contain spatial and temporal information. For tweets or posts on social media platforms the current position of the user as well as the timestamp when the post was created are stored.

[1] `https://code-de.org/`

Extracting the time, location, and actors from text resources and deciding which mentioning of, e. g., a place belongs to an event and which person took part in it is a complicated task, that requires sophisticated strategies and information retrieval algorithms, e. g., as proposed in [94]. However, once the event information is available in a structured format it can be further analyzed, e. g., with respect to their spatial and temporal correlations.

An important task in the context of business intelligence and document exploration applications is finding similar events to a given reference event or identifying groups of similar events in a given event data set. Events that are correlated in space and/or time to a reference event can be found by searching for the nearest neighbors, known as the *k nearest neighbors* or by applying a multidimensional optimization algorithm to create the *Skyline* for this reference event. In order to decide which events are correlated, a similarity metric is needed. This similarity can be expressed by the spatial and/or temporal distances of the objects The distance between two spatial objects can be calculated using the well known Euclidean distance and for two temporal objects, the minimum distance can be used. However, the Euclidean distance is only applicable in a Cartesian space, whereas calculations on a globe should use the great-circle distance, e. g., using the Haversine formula. Though, neither spatial nor temporal distance functions are included natively in the Big Data processing platforms.

Furthermore, the nearest neighbor and Skyline computations have a time complexity of $O(n^2)$ if implemented in a naïve way. A spatial index on the data could improve the nearest neighbor search significantly as it allows to discard large parts of a data set that cannot contain result objects. Spatial and temporal data partitioning also plays an important role in this scenario. For computing the Skyline, the spatial and temporal distance of every object has to be compared to every other object in the data set. With a partitioning strategy that is aware of its application, these comparisons can be performed on the partitions as a preliminary filter and hence, reduce the number of overall comparisons drastically.

2.3 Requirement Analysis

From a user perspective, the workflows in the use cases described above are always similar. For initial processing and analysis tasks, notebook systems such as Jupyter or Apache Zeppelin are used. In these scripts, the data, often present as plain text files in the file system, is referenced and represented by some appropriate data type. Using this data type and constructs of the chosen programming language, the researchers specify operations to apply in order to solve their current task. Hence, after an initial version of the script has been executed, its result is investigated and appropriate follow-up operations are added. Since the data sets are large and many researchers work in parallel on their individual problems, data set cannot necessarily be cached. Thus, with every execution of a script (or cell in the notebook system), the data file is loaded from storage.

In order to solve the example, questions outlined in the use cases, after the large data set has been loaded, it is, for example filtered by some spatial region and/or temporal interval or is joined with some other data set based on the spatial and/or

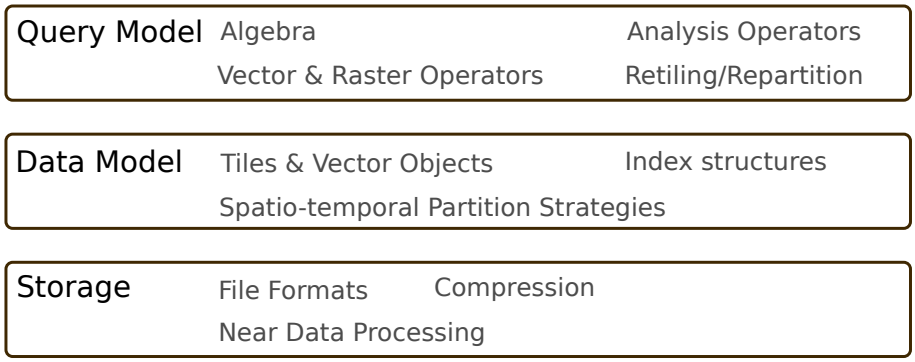


Figure 2.1: Three levels to integrate spatio-temporal data processing into.

temporal features. After this, the result is aggregated to, e. g., calculate the average temperature value, potentially filtered to receive values (not) exceeding a threshold parameter, and finally plotted on the screen. After the workflow has been finalized, it may be executed again and again with different threshold values to inspect data distribution and correlations.

Based on this, we can identify several requirements and user demands for spatio-temporal vector and spatial raster data processing. The requirements can be categorized into three levels, that built upon each other (cf. Fig. 2.1): The storage level is the lowest level and represents the file system or hardware level. On the next level, a data model for representation and exploitation of the respective characteristics is needed. The query model implements the algebra and operators to work on the data. In the following, we will discuss these requirements in more detail, starting at the lowest level.

2.3.1 Storage Level

On the storage level, especially for raster data, one goal is to optimize the space required to store data on disk and also to reduce the time to access all or specific elements in a file. As we will discuss in Section 3.1.1, there already exist several file formats for storing spatial vector and raster data. These formats are often generic in order to support a wide range of applications and use cases. However, especially for raster data some use cases might benefit from tailored formats that were designed for specific scenarios and their characteristics. In such cases, the formats could transparently apply compression on (parts of) the actual data.

Besides space efficiency, the time required to load the data sets is extremely important to allow interactive usage of a system. To achieve this, not only optimized file formats are needed, but also an intelligent storage layer that supports query operators by either executing (parts of) the query on specialized hardware or use-case aware software, like the file system. This is also known as Near Data Processing (NDP).

2.3.2 Data Model

As a basis for vector and raster data processing, native data types are needed to represent data. Only with a valid representation of, e. g., a position or raster tile, a system can store data efficiently and implement the operations required by users.

Vector data For vector data processing a data type is needed to represent at least the most common shapes, namely points, linestrings, and polygons. Other shapes, such as circles, may also be supported to even more enhance the usability. Along with the spatial feature, a temporal component is needed to represent e. g., the *valid time* of the spatial feature. The valid time expresses the time (period) when the associated object existed in the real world. Alternatively, the *transaction time* is used as the time the object was added to a data set. Hence, an instant type, describing a single point in time, and an interval, i. e., a time range with a start and end point in time, are needed. Especially for the valid time, an interval may be open-end, meaning the object will exist forever, or the end time is not known (yet). The SQL extension on bitemporal data storage defined in [60] includes statements to explicitly mark temporal columns to form the valid time and/or transaction time of tuples.

Raster data A raster data set contains of set of *tiles* where each tile contains pixel values. Typically, a tile has a rectangular shape and thus covers $n \times m$ pixels. In most cases the tiles may be aligned with the underlying coordinate system, i. e., the edges of a tile are parallel to the axis of the coordinate reference system. However, depending on the recording method, the tiles may also be skewed.

It must be possible to identify a tile in the global raster, e. g., its column and row index in the grid, and it must contain the values for all the pixels in the tile. The actual data types to be stored can differ from one data set to another: Byte, Integer, Double, or even custom types. It must be possible to represent these different types within tiles, while all pixels in a raster data set are typically of the same type.

However, depending on the application scenario, there might be multiple values for a pixel, e. g., when the sensors in the satellite from the earth Observation scenario in Section 2.1 measure values in different bands. In such a scenario, a tile must not only store the sequence of pixel values, but also a sequence of values per pixel, one for each band.

Spatio-temporal Partitioning & Indexes In a data parallel cluster, every node processes a portion of the complete data set, called a partition. Due to the generic data model of the processing platforms, the existing partitioning mechanisms do not consider spatial or spatio-temporal distribution of data.

A partitioner is responsible to chunk a data set into disjunctive partitions. Each partition is later processed by a single worker node. When data is loaded some generic partitioning scheme (round robin, hash) might be applied. As a result, a partition contains objects that are, from a spatio-temporal point of view, not similar to each other, i. e., they are not necessarily located near to each other.

Thus, new partitioning techniques are needed that quickly assign an element to a partition based on its location and/or time of occurrence. There are, however, some requirements for a partitioning algorithm:

Completeness: Every object of a data set must be assigned to (at least) one partition.

Spatio-temporal locality: Objects that are in spatial and/or temporal close proximity to each other should be assigned to the same partition.

Reasonable number of partitions: The number of partitions should be large enough to keep all worker nodes busy but small enough to not burden the execution engine with managing partitions and assigning them to workers as this incurs a significant overhead and negative impact on query performance.

Balanced partition size: The number of elements in each partition should be (almost) equal, so that every worker node has approximately the same amount of work. This is necessary to prevent one node doing all the work while others finish earlier and idle while waiting for the overloaded worker node.

Handle data skew: Skew can occur for example in a use case of position information of mobile devices: While one can expect the majority of the devices to be located in metropolitan areas, only very few can be found in rural areas. Thus, in terms of the covered area, a partitioning should create many smaller partitions in dense areas and larger ones in regions with few objects. Skew can occur for the temporal feature as well, of course. Skew handling is tightly coupled with the previous requirement of balanced partition sizes.

Such partitioners may be based on the spatial feature only, the temporal feature only, or both together. The spatio-temporal partitioning information then can be used during query execution to decide which partitions contribute to the query result.

Since tiles in a raster data set have a spatial extent as well, they can also be partitioned based on their position.

Usually, data is loaded from text files or some relational database, like Hive. After partitioning the data, all elements in the data set need to be tested for, e. g., a filter or join predicate, without the help of index structures. However, especially for the expensive spatial predicate checks appropriate index structures will reduce query execution time so that it becomes practicable at all. If the index can be persisted, subsequent executions will benefit from the potentially costly creation of the index structures.

2.3.3 Query Model

Relational database systems provide a set of operations on spatial data and many of them are also defined in the SQL/MM standard (Part 3)[59]. A platform for vector and raster data processing also needs to implement (at least some of) these functions to operate on the underlying data.

A loader function must support at least one of the established formats, like *well-known-text* (WKT), *well-known-binary* (WKB), GeoJSON, etc. for vector data and,

e. g., NetCDF for raster data. If loading data is not natively supported, transformation operations from other data structures are needed. Although when working with large data sets, queries and programs are run interactively and results are expected as fast as possible. Users and operations should not spend (too much) time with preparing data sets for specific algorithms or operator implementations.

Vector operations A spatial filter on vector data returns all objects from a collection that match a predicate with a given query object. In a spatio-temporal filter, both aspects, the location and time (instant or interval) of the objects in the data set must fulfill the given predicate. Like a traditional join in relational databases, a spatio-temporal join must find all pairs of objects that match the given predicate. The predicates define (or test) how two objects are related to each other. An exhaustive list of possible relations for spatial objects has been defined in the DE-9IM by Clementini and Di Felice in [24] and the same can be derived for temporal objects. For operations, Güting introduced the geo-relational algebra in [43] and Allen proposed the interval algebra in [6].

Besides these relational operations, analytical operations are needed to extract new information from the raw data. The k nearest neighbors (kNN) search is just one example to find objects that are similar to a given reference object. Similarly, the Skyline as a multidimensional optimization problem considers the two dimensions (distances to the reference object in space and time) separately. To find groups of similar events a (density based) clustering method such as DBSCAN [38] can be used. All three operations require a metric to define the similarity. Since we focus on spatio-temporal data processing, such a metric can be the distance between the objects, and thus, appropriate implementations are needed.

Depending on the use case and input data, different distance functions are required. They are needed for operators like Clustering, Skyline, kNN. The actual function implementation may vary according to the use case: While for two dimensional planar coordinate values the simple Euclidean distance function is enough, for GPS coordinates on a globe, more accurate functions are needed. Furthermore, there is e. g., the Hausdorff distance, which calculates the difference of sets are (in a metric space). Also not only are the distance functions needed for spatial vector (and raster) data, but also for temporal objects (point in time and intervals). To allow a flexible usage of the envisioned platform, different distance functions should be present and applicable for the operators. Furthermore, users should be able to implement and use their own distance functions within the platform.

Raster operations While vector data processing functions are defined in the SQL/MM standard Part 3 [59], there is no standard for raster data operations, and especially not for the combination of raster with vector data [97]. However, 2019 the SQL/MDA [70] standard was published to specify access and handling of multidimensional arrays in SQL. These multidimensional arrays can be used to model and query raster data using SQL.

For raster data, a filter typically defines a rectangular or polygonal area to find all tiles that lie within or intersect with that area which are then further processed. The filter operation is not limited to spatial features, but can also be used to find tiles with certain values or exceeding them in a given range. Often, a raster data set represents an image or values that can be drawn as an image (temperature values) and thus, users need to find objects in these images. Therefore, a filter should also be able to find all tiles containing the object, given as a (two-dimensional) sequence of pixel values as a pattern.

Joining two raster data sets is required to combine the pixel values in each tile, by adding them or calculating their difference. The latter could be used to combine two raster images taken at different points in time to generate a raster data set showing the change of temperature values or ground levels.

A combination of raster and vector data is crucial to find all tiles or pixels from a raster data set that intersect with a given vector polygon, as described in the use cases in Chapter 2. The combination of raster and vector data also means joining them, which requires efficient implementation to achieve reasonable response times.

In addition to the filter and join operations, aggregation operators are needed to process the values and actually compute some application dependent results. Such functions include, e. g., `area` for vector data, and `avg` for raster data. Naturally, these functions can also be used in filter and join conditions. Furthermore, operators and functions that work locally on single tiles, but also globally on a set of tiles are needed, e. g., to identify tiles with values exceeding a threshold or aggregate tiles to a single scalar value.

Algebra, Language & Usability Since data workers should not focus on programming tasks but rather on their current (data science) problem, a high-level language is needed to hide the underlying internals (partitioning, specific algorithms for operators, etc.). SQL is such a high-level declarative language. It is very wide spread and is the de facto standard for interacting with (relational) database systems. Furthermore, Apache Spark supports SQL queries using SparkSQL extension on Dataframes. Since there is also a need for SQL in the Big Data ecosystem, several database systems with an SQL frontend have been developed that operate in a data parallel cluster.

For supporting spatial vector and raster data operations, an SQL compiler must be extended to support the operations and predicates mentioned in Section 2.3.3. Additionally, the data types for representing vector objects and raster tiles must be supported.

Albeit SQL is well-suited for extracting data from a data set and perform basic calculations, it can hardly be used to prepare data from different sources, in particular when they are of different formats. Furthermore, long workflows are difficult to express in a single SQL statement. For such tasks, a script language like Pig Latin [75] is an alternative to SQL as it allows to easily create programs that can process almost any type of data in arbitrary complex programs.

Chapter 3
Foundations

This chapter discusses the fundamentals needed when working with spatial and temporal data. We begin with an introduction to spatial and temporal data and the typical file formats used to store this kind of data. After this, we briefly present spatial and temporal index structures and then explain the basic concepts of the Big Data processing platforms.

3.1 Spatial & Temporal Data Processing

In order to process spatial and temporal objects, data types representing real world objects and operations thereon are needed. Since spatial and temporal data processing is performed in many application that also interact with each other, standards for types and operations were defined.

This section gives a short introduction into the data types and also describes the operations needed as well as existing index structures. More information can be found in the following literature. In [42] Güting presents an introduction to spatial database systems including the required types and operations. Eldawy and Mokbel analyze which main features are required for spatial data processing systems in [34] and Jacox and Samet look deeper into spatial join techniques in [61]. In [15] Begum and Supreethi survey spatial indexing methods.

3.1.1 Data Types and Operations

Typical spatial objects that are contained in real world spatial data sets are [42, 43]:

Points: A point describes a position in an n-dimensional space. It does not have an extent in that space and can be seen as the most basic shape. Points can e. g., be used to represent the position of a sensor, the point of occurrence of an event or even the location of a city – if the area of the city is unimportant for the use case.

Linestring: A linestring is described by a sequence of connected points, where the connection between two points is called a segment. Linestrings are used to model roads or rivers, if their actual width is not of interest and only their route is important.

Polygon: A polygon is an object with a spatial extent. The boundary of a polygon is defined by a closed linestring whose starting point is also the ending point. However, whether the last point has to be explicitly given with the values of the first point depends on the used system. Polygons are used to model areas such as lakes, countries, or, as mentioned before, cities in their actual area or rivers and roads with their real width. A specialized form of a polygon

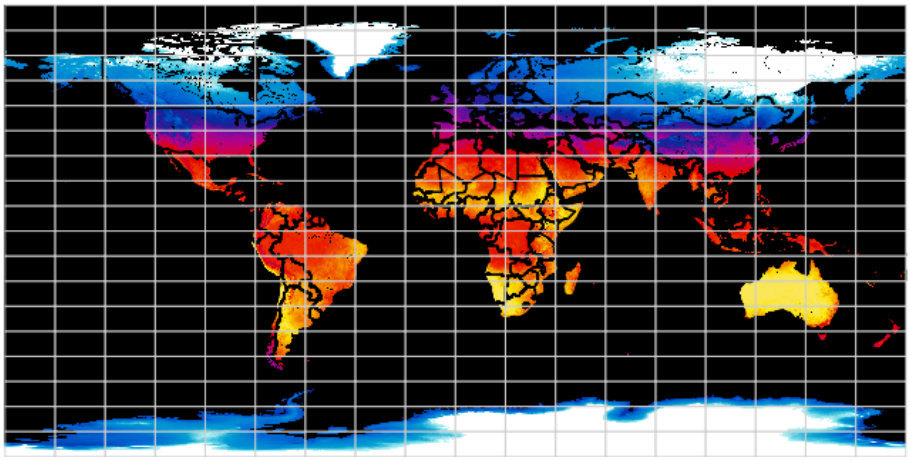


Figure 3.1: Visualization of tiles in a raster data set.

is a rectangle, for which it is enough to only specify the two corner points that can be connected by the rectangle's diagonal. A polygon has a *boundary*, and *interior* and *exterior*. In many systems, the interior is the smaller area bounded by the given sequence of points, while the exterior is the complement of the interior. However, some systems expect the polygon definition in clock-wise ordering and define the interior to be on the right hand side of the given sequence of points and the exterior on the left hand side. Furthermore, polygons may have holes, for example the official border of Italy has a hole for the Vatican City.

These types are used within a data set/relation for attributes so that a system can interpret them and use them within operations and optimization.

Besides the mentioned geometric shapes, one can easily think of further forms that may be required for a specific problem. Such forms are e. g., triangles or circles. They can, however, be modeled using polygons, although circles can only be approximated. Real triangles and perfect circles are usually not found in nature so that many systems do not define a specific data type for them. However, PostGIS for example allows to create circles by creating a `ST_BUFFER` object from a point and provide a radius. According to the PostGIS documentation[1], a buffer is a geometric object that covers all points within the given radius around the provided geometry.

In addition to vector data, another important type of spatial data processing is *raster* data processing. A raster data set divides the space into usually equally sized rectangular raster *tiles*. A tile itself consists of a set of pixel values, each pixel covering some area, as illustrated in Fig. 3.1. Raster data is typically created by some sensors, for instance camera sensors. The resolution of the sensor determines the pixel size. As described in the introduction, the imaging may be performed by high resolution cameras taking images from chips on wafers or (optical) sensors on satellites capturing the earth's temperature or cloud coverage.

Raster tiles cannot simply be modeled using the existing rectangle or polygon types. In raster data a pixel is contained as a value in a tile, while for vector data

[1] `http://postgis.net/docs/ST_Buffer.html`

the geometry itself is the actual (spatial) value. For this reason and because both types of data representation require their specific operations, individual data types are needed.

While spatial vector (and raster) data is usually two- or three-dimensional, there is only one dimension for temporal data. Analogous to points in spatial vector data, there exists a point in time, also referred to as an *instant*. An instant describes a certain value on a time axis. *Intervals* represent a period with an instant as its beginning, and an optional instant as its end. The SQL 2011 standard [60] contains definitions for such temporal data. This value however, depends on the required granularity and is additionally limited by capabilities of the sensor (clock), if any. Granularity means the precision of the measurement device or time type: minutes, seconds, milliseconds, etc.

The temporal units form a hierarchy, i. e., an hour consists of minutes, a minute consists of seconds, etc. This hierarchy could be used to model *uncertainties*: If an instant is not given in on the finest granularity level supported, it may be seen as imprecise, consequently making it an *interval*. Imprecise or uncertain definitions have an impact on the operators working on these types as well as e. g., distance functions. The handling of imprecise values, which is also possible for spatial data, is not part of this thesis.

An interval describes a range on a time axis with a start and end point, and therefore has a length. The start and end values are instants on the time axis. Often, but not necessarily always, intervals are defined as right-open, meaning that the given start value is included in the interval, but the end value is not.

Since spatial and temporal data has been collected and processed for a long time, all relevant systems included support for such data. The part three of the SQL Multimedia (SQL/MM) standard defined in [59] was created to ensure that all SQL systems offering spatial data support are compatible to each other [96]. It defines several basic data types as well as how users can store and retrieve spatial data in and from their relations. Additionally, functions to test relationships between two geometries and to convert between different geometry types are specified.

While SQL/MM standardizes how DBMS should handle spatial data and what functions are required for users to query and process the spatial objects, it does not specify how two geometries can be related to each other. The Dimensionally Extended nine-Intersection Model (DE-9IM) [25] is a standard that describes how two geometries are related to each other in the two-dimensional space and the topological model makes it invariant to rotation. The model is based on the boundary, interior, and exterior of the polygons and Fig. 3.2 shows a visualization of the result of the relation of two 2-dimensional polygons. From the DE-9IM, *spatial predicates* can be derived, which can be used to check the relations of two geometries to each other. Typical predicates often used in applications are:

contains One geometry completely contains the other one, so that no point of the contained geometry is outside of the containing geometry, i. e., in its exterior.

touches Two geometries share at least one common border point, but no interior point of one geometry lies within the other geometry.

intersects At least one point of one geometry is in the interior the other geometry.

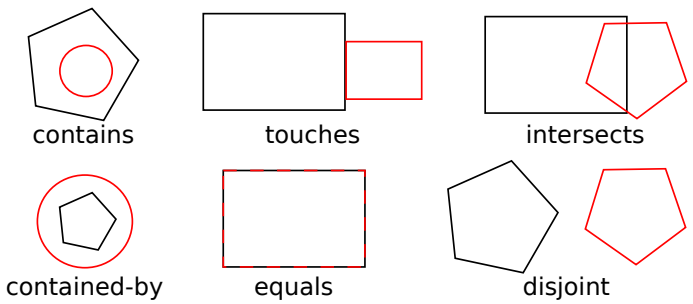


Figure 3.2: Visual representation of the possible relations of two vector geometries in 2D space.

contained-by This is the reverse operation of *contains*.

equals Both geometries are congruent and no point of one geometry is outside the other geometry.

disjoint Both geometries do not have any point in common.

Any application providing operations to process spatial and temporal data needs to also provide functions to write and read such data to/from storage. Often, data is shared between different applications: data is produced by some sensors and send over network to a receiving station, where it is written to storage and later loaded processed by some user application. DBMS with support for spatial data (see Chapter 4) define their own binary storage format. However, for data exchange and communication among various systems, some standard formats have been developed.

Well-Known-Text & Well-Known-Binary The Well-Known-Text (WKT) format is a textual format representing one geometry per entry. It is often used in CSV files, generated e. g., from database dumps. Here, however, quotation or a delimiter other than a comma has to be used, since WKT already uses commas within the textual representation.

According to the definition (see [76] p. 3-11 ff.) a typical WKT string starts with the type of the geometry, an opening parenthesis followed by the sequence of points describing the geometries border, and a closing parenthesis. For example, the text `POINT(50.68 10.93)` defines a two-dimensional point and `POLYGON((50.68 10.93, 52.81 10.93, 51.58 13.47, 50.68 10.93))` represents a triangular polygon by a sequence of points, forming the polygons border. A polygon may consist of multiple rings, given as linestrings, which are also supported as an own type in WKT. The previous example defined be polygon with only one ring, another one could be added to model a hole in that polygon. WKT is also able to model collections of geometries that are a logical unit, e. g., countries with their islands. This can be achieved using the `MULTPOINT`, `MULTILINESTRING`, or `MULTIPOLYGON` types.

In contrast to the string representation of WKT, the Well-Known-Binary (WKB) format models objects as a sequence of bytes so that applications can exchange data in binary form (see [76] pp. 3-24 – 3-28). In WKB geometry types are encoded using an integer value (point has code `1`, polygon code `3`, ...). Some header information, such as the geometry type, format, number of points, etc. is prepended to the actual payload data. Fig. 3.3 shows a sample representation of a polygon definition.

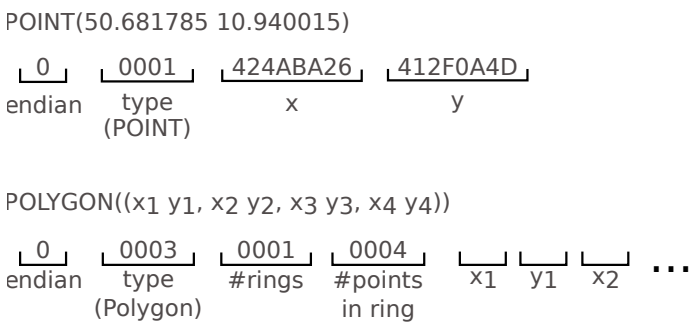


Figure 3.3: Depiction of order of values in WKB

GeoJSON WKT and WKB both only define how to represent a single geometry using textual or binary format, but they do not add any context, like Spatial Reference Identifier (SRID) or additional features. The JavaScript Object Notation (JSON) format was created in the context of web development to share JavaScript objects between different modules (such as front-end and backend) of a (web) application and different applications. Compared to Extensible Markup Language (XML), JSON is much more compact as it reduces the number of boilerplate characters making it suitable for exchanging structured data objects.

A JSON document models an object in the form of key-value pairs. For an event, keys can be for example `eventID` and `description` with appropriate values. The values may be primitive types, including strings, collections, or even nested JSON documents to model complex objects. For the location of the event, one could add a key `location` and assign it the WKT string of that location. That way, any application can use a JSON parser to unmarshal a received JSON document into an object and then use another library to parse the WKT string of the location field into a geometry object that can be used for further calculations.

An example of this approach is given in Listing 1. However, the JSON parser will treat the WKT string of the location as a normal string and an application requires an additional library and transformation step to parse that string into a geometry object. Since values may also be complex JSON objects, one could also set the location to an object that defines a shape type, the coordinate values and necessary values. As an attempt to standardize the modeling of spatial objects in JSON is the GeoJSON format described in RFC7946 [19].

GeoJSON can directly model a geometry using a JSON object with two keys: `type` and `coordinates`, where `type` defines the shape, e. g., point, linestring, polygon, etc. and `coordinates` specifies the coordinate or sequence of coordinates, for the particular geometry. Depending on the use case, it might be enough to just specify the coordinates for the shapes of a data set. If the use case requires to add additional information to the geometries, GeoJSON provides *feature objects*, that contains fields for additional meta data, such as a name, description, and other information, along with the `geometry` field. An example of a feature declaration is given in Listing 2 that models a single feature, representing a polygon (rectangle) and provides a name property as meta data.

There exist various libraries with parsers for JSON as well as GeoJSON to convert the text data into objects (or `structs`) of the respective programming language.

Listing 1 Example of a JSON document for one event.

```
{
  "eventID": 12345,
  "description": "First Metallica Concert",
  "date": "1982-03-14",
  "location": "POINT(33.83, -117.90)",
  "actors": [ {"name": "Metallica"} ]
}
```

For GeoJSON, the libraries will create a `Feature` instance with the fields for the geometry, as a `Polygon` instance and a field for the properties.

More File Formats Although already described in 1998 in [37], Shapefiles are still an often used format for exchanging geospatial data and supported by many Desktop GIS applications. A Shapefile is a collection of at least three files that all together comprise the complete data set: the main file (`.shp` extension) stores the geometries data as vector objects. An index file stores offsets to faster lookup records in the main file. Additional attributes and features can be stored in a third `dBASE` file.

The quite old format has some limitations that may make it unsuitable for large data sets: each of the files comprising the Shapefile may be 2 GB at maximum. There are other limitations such as record lengths, field name lengths, number of fields, or that data fields can contain either space or time, but not both.

For raster data, an often used format is GeoTIFF, an extension of the TIFF format with geospatial attributes. The format is popular as it allows a lossless data representation, is able to compress the stored data using various schemes, and can be augmented with application specific information [4]. GeoTIFF is a self-contained format and directly includes all necessary attributes. The attributes are used for e. g., the used coordinate system and the projection method [83].

NetCDF is a binary file format for storing and sharing any (hierarchical) scientific data. In the header, key–value pairs are used to describe the meta information about the file, but also the organization of the payload data [101]. The payload data is stored as an (multidimensional) array, which makes it suitable for raster images. However, unlike GeoTIFF, NetCDF does not include any compression techniques.

The aforementioned formats were designed when users worked on single machines. However, in the era of rented clusters where computing nodes are decoupled from storage, e. g., Amazon EC2 and S3, requirements have changed. In cloud computing, machines are added and removed from a configuration and nodes read only parts of files. The Meta Raster Format (MRF) [13] addresses these challenges. MRF consists of three files: an XML file containing meta information about the content (spatial projection, tiling, etc.), a data file with the actual data as raw binary values or even in image formats like JPG, PNG, and an index file with offset and spatial location information for each tile in the data file.

Listing 2 A sample GeoJSON document for a single feature object.

```
{
  "type": "feature",
  "geometry": {
    "type": "polygon",
    "coordinates": [
      [10 10], [20 10], [20 20], [10 20], [10 10]
    ],
    "properties": {
      "name": "a query rectangle"
    }
  }
}
```

Temporal Data Dealing with temporal data is not trivial and causes many difficulties during development and execution of software systems. Especially working with values from different time zones and representations requires special care. Currently, there exists no file format dedicated to temporal data. A common strategy to store date or date-time information is to transform it to a textual representation and parse this string when reading data. Most modern programming languages provide mechanisms to apply a given pattern string describing the expected format on a input string to convert it back to date/time object. The ISO-8601 standard defines a timestamp format for interchanging date and time values. It uses either the Coordinated Universal Time (UTC) or the local time of the application.

The SQL-92 was the first SQL standard to define how to handle date and time types. It includes a time-zone information table and additionally defines how to convert the various representation formats, such as the Unix Epoch and string formats, to and from the date and time types. In standard SQL, a duration can be expressed using the `INTERVAL` type. The types for date and time can be accessed using various methods and it is possible to perform comparisons using standard operators (<,>, ...) and calculations. The date and time representation to be used in Internet protocols and documents is discussed in RFC 3339[2].

3.1.2 Geometric and Geodetic Computations

All spatial or spatio-temporal objects are located at some position in space (and time) by definition. While for the temporal component of such objects it is clear that they lie on the time axis of our real world, it is not always clear what the actual space for the location of such objects is.

In many applications it is sufficient to interpret a point, e. g., `(33.83, -117.90)`, or any other geometry in a two-dimensional planar space. This planar space might be a representation of the earth, but does not necessarily have to. It can as well be seen as some abstract space with potentially infinite extent. However, using a

[2] `https://tools.ietf.org/html/rfc3339`

planar space for calculations on geometries that are supposed to be on the earth's surface has some significant drawbacks that need to be considered.

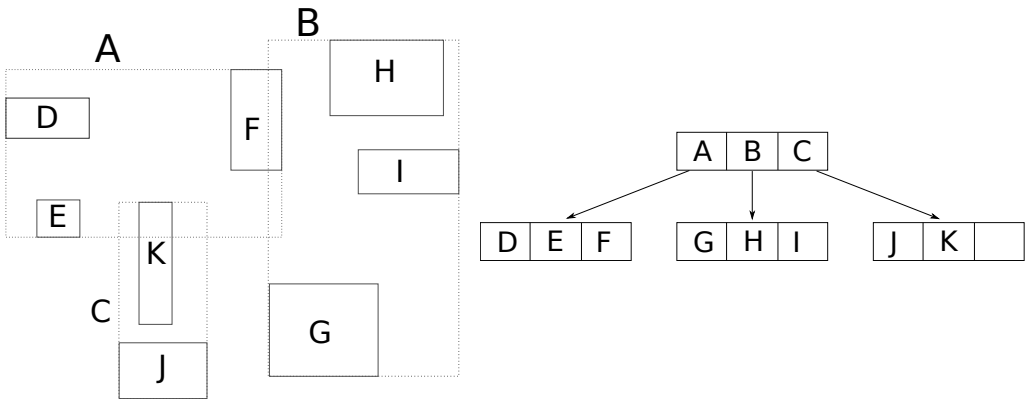
The earth is a three dimensional spheroid. If it is treated as a two dimensional planar space, this space is bounded in both dimensions although on a spheroid there is no "end". Another problem arises at the poles. With a two dimensional representation, the poles, which are actually single points each, stretch to infinity at the upper edge (the North Pole) or lower edge (the South Pole) of the two dimensional space. Thus, images that use such a projection, i. e., Mercator projection often do not show the poles [96]. This distortion towards the poles additionally has the effect that countries are shown larger than they actually are, giving a false impression of the earth. A projection without this distortion is e. g., the Authagraph projection[3], that retains shapes and sizes of the countries and continents as well as the relative position to each other.

A further problem is in the calculation itself: while on the planar space the relatively simple Euclidean distance can be used to compute the distance between two points, on a spheroid the distance formula becomes more complicated as it must respect the earth's spherical shape. Though, if only data from a rather small area, compared to the size of the earth, has to be processed, for many applications it is enough to use the two dimensional planar space, because the deviation errors are negligible.

However, applications used at, e. g., DLR process global data and require correct results and thus, for calculations the earth's shape needs to be modeled as correctly as possible. The earth is not a perfect sphere, but rather a spheroid flattened at the poles. The model of the earth is referred to as the Spatial Reference System (SRS). It defines the coordinate system in which a point or any geometry is given. Typically, libraries implementing spatial operations support different SRSs which can be set often per geometry individually by setting the SRID of the geometry. In fact, thousands of SRSs were already created each for a specific application scenario and region: The PostGIS extension that adds spatial data types and operations to the PostgreSQL DBMS includes over 5000 SRSs. An SRS tries to model the earth's surface as well as possible for a use case. The World Geodetic System 1984 (WGS84) is often used in European or North American applications such as GPS based navigation systems as it is a good general model of the earth in this area. However, data from certain regions such as the Highlands of Tibet will suffer from inaccuracies as the WGS84 does not model them well. There are other SRSs that fit the earth's surface better in this particular region [96]. In such geodetic coordinate systems a point is given in two or, depending on the use case, three coordinates: a latitude, longitude, and the optional altitude. In mathematical geometry the x value is given before the y. In GPS, the latitude, which compares to y, is usually stated first: `(latitude, longitude)` or `(latitude, longitude, altitude)`.

Note, considering the SRS for spatial objects is not in the focus of this work and just presented here for the sake of completeness. The WKT and WKB formats do not support setting the SRS. Therefore, it is (currently) left to the user to convert coordinates of different SRSs into a single desired one, or use the correct functions.

[3] `http://www.authagraph.com/top/?lang=en`



(a) Spatial organization of R-tree nodes (b) R-tree structure.

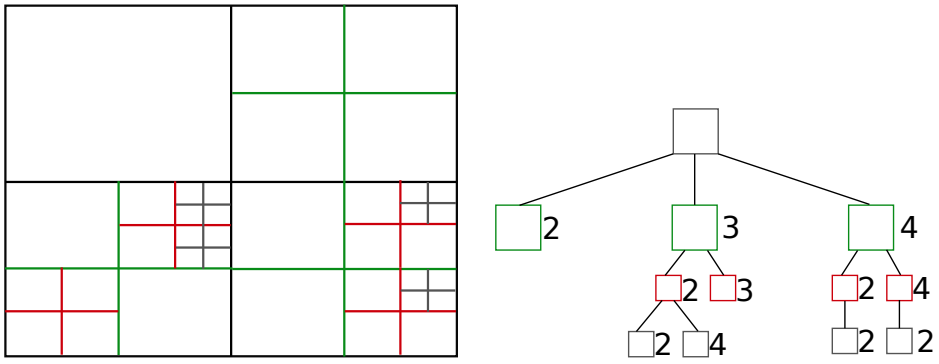
Figure 3.4: R-tree spatial organization and tree structure.

3.1.3 Indexing Spatio-temporal Data

DBMSs have to execute queries with expensive operations extremely fast to meet the users' requirements. To reduce the amount of data that actually has to be accessed during query evaluation, several index structures have been developed. Index structures typically are tree structures and during the traversal significant portions of the data set can be excluded without needing to evaluate predicates. The most prominent index structure used in relational DBMSs, presumably is the B^+-tree. In a B^+-tree the inner nodes contain the indexed values in sorted order as well as pointers to the children. The leaf nodes then contain the pointers to the actual data pages. The number of elements an inner or leaf node may hold is determined by an order value each, e. g., m and k. A node stores between m and $2m$ values. If during delete or insert operations a node would contain fewer or more values than m or $2m$, respectively, according re-balancing steps are performed.

R-tree The spatial index structure that is probably most often implemented in GIS and DBMS is the R-tree [45] (or one of its variants). It follows the same idea as the B^+-tree or its base variant, the B-tree. The notion of the R-tree is that an inner node represents a Minimum Bounding Box (MBB) of the elements it contains. Thus, the root node represents an MBB containing all elements in the index. On each level, the space is divided into regions/MBBs, that may be non-disjoint. The leaf nodes store the MBBs of the actual geometric objects. The R-tree is neighbor-preserving, i. e., geographically near objects are also near to each other in the tree. Inside a node, the objects are sorted for faster lookups. A sample R-tree is depicted in Fig. 3.4: Fig. 3.4(a) shows the two-dimensional objects as well as the nodes' MBBs and Fig. 3.4(b) depicts the resulting tree structure.

Like B^+-trees, the tree is balanced and insert and delete operations might cause re-balancing steps. Especially inserts can get problematic as the MBB of the respective node has to be extended. Rebalancing steps resulting from node overflows and splits are expensive to compute and also need to be propagated to the root node, so that large parts of the tree may be reorganized after a single insertion.



(a) Spatial organization of Quadtree nodes.

(b) Unbalanced Quadtree structure.

Figure 3.5: Quadtree spatial organization and tree structure.

Lookups start at the root level and the query object is compared to all MBBs inside a node. The traversal continues in the sub-tree that matches, i. e., intersects, with the query object until the leaf level is reached. Since in an R-tree the MBBs are non-disjoint, multiple children of a node might intersect with the query object so that all those paths need to be traversed. A query to the R-tree only yields result candidates, since only the MBBs of the geometries are stored. Therefore, it is possible that for, e. g., a point query the query object intersects with the MBB of a polygon, but not with the polygon itself. To prune this candidate result set, a secondary step is required to check the actual geometries of the candidates against the query object.

Following multiple paths during lookups can be a serious performance issue. To solve this, the R^+-tree [102] forces the MBBs of the nodes to be disjoint. It follows that objects spanning multiple MBBs need to be *clipped*. This clipping results in additional overhead when inserting objects, though.

There are more variants of the R-tree such as the R^*-tree [14] that tries to improve the insert performance or the Sort-Tile-R-tree (STR-tree) that uses a sort-tile bulk loading strategy, but is a normal R-tree otherwise.

The R-tree could be used to store temporal and spatio-temporal keys. For temporal data, the MBBs will be one-dimensional intervals. In order to store spatio-temporal data, the time can be added as the third dimension of the spatial feature. Furthermore, there are specialized variants for indexing spatio-temporal data, such as the R^{ST}-tree [86], the time parameterized R-tree (TPR-tree) [87], and the MV3R-tree [100] for interval queries.

Quadtree A Quadtree [88] can also be used to index two-dimensional objects (the extension for three-dimensional data is called Octree). The root node of the tree represents the Minimum Bounding Rectangle (MBR) (the two-dimensional form of the MBB) for all objects in the tree. A region is recursively divided into four sub-regions, called quadrants, if the number of objects inside that region exceeds a given threshold, typically defined by the block size on disk in traditional relational DBMS. The generated quadrants usually have quadratic or rectangular shapes, because as they offer cheap containment checks during lookups. The tree structure is given

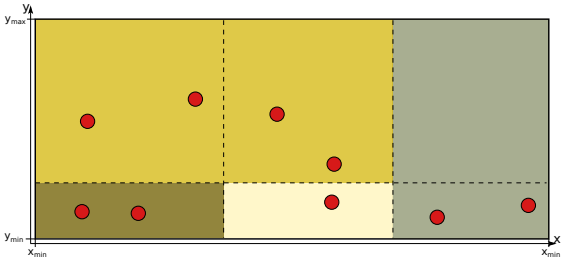


Figure 3.6: A Grid-File with cells (dashed lines) and regions (by color).

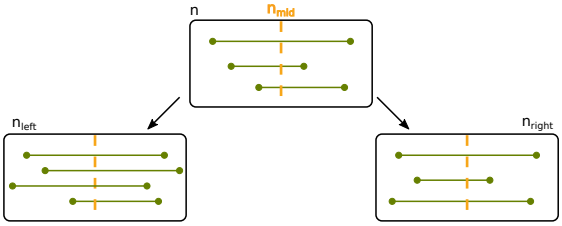


Figure 3.7: Representation of an interval tree.

with the recursive division of the quadrants. As shown in Fig. 3.5, a Quadtree is not balanced since at any level a generated quadrant may be empty while another one still contains elements and may be further split.

Grid File Grid-Files [72] are a multidimensional index structure and were developed to reduce the number of disk accesses during exact match lookups while still having a dynamic structure that is neighborhood-preserving.

As for Quadtrees, the minimum and maximum values in all dimensions need to be known a priori. The space is divided into cells by lines parallel to the axes and cells are combined into regions in a way that the size of a region matches the size of a disk block. Splits are performed during inserts when a cell overflows.

One can find many more interesting index structures in the literature that may be worth implementing or investigating in the context of this work, e. g., GeoHash or Binary-space-partitioning tree (BSP-tree). GeoHash[4] is a hierarchical structure that recursively divides the space into a number of equally-sized regions and enumerates them using Space-filling curve (SFC). The concatenation of the region numbers on each level then form the geo-hash string. The BSP-tree splits the space in always two regions. However, this split must not necessarily be parallel to the axes. The decision if a region needs to be split is cost based, where in traditional relational DBMS this cost is determined by the disk block size.

The R-tree (and its variants) have proven to be very fast and efficient and the Quadtree is easy to compute and will therefore be used in the following. However, future work should investigate if GeoHash and other data structures might be applicable in this work, too.

Temporal information can, but does not have to, be associated with a spatial feature. This results in two variants for indexing: (1) spatio-temporal indexing and (2) temporal indexing.

Indexing Temporal Data In spatio-temporal indexing, the spatial and the temporal information is used as a combined key to be used in the index. For this, multidimensional index structures, such as the ones explained above can be used and the two-dimensional spatial objects are augmented by the temporal dimension. These three dimensional objects could, e. g., be put into a variant of the R-tree.

[4] `http://geohash.org/`

However, especially when working with data of historical events, the temporal dimension might span over an extremely long period, which might have a negative impact on the performance of the index. When using only the temporal information for indexing, interval trees are very common. As shown in Fig. 3.7 the tree is organized as a binary tree. The left child of a node n stores all intervals that are completely *before* n_{mid}, whereas all intervals completely *after* n_{mid} are stored in the right child of n. The node n itself stores all intervals that intersect with n_{mid}. On the root level, n_{mid} represents the middle point of the complete range covered by all intervals in the data set. At every inner level, n_{mid} represents the middle point of all intervals in this sub-tree.

3.1.4 Indexes for Raster Data

Spatio-temporal vector objects are indexed to speed up queries that quickly need to find these objects based on their spatial and/or temporal features. For raster data, however, there are different use cases for indexing:

1. When one needs to find tiles that, e. g., intersect with tiles from another raster data set or that lie within a given region.

2. When searching for tiles with certain values, the index must be based on the pixel values inside the tiles.

In the first case, the index needs to be based on the spatial features of the raster tiles. As we will show later in Fig. 5.12, raster tiles can be represented as rectangles in the vector space. Hence, the spatial index structures described in this section can be used also for raster data.

In the second case, the index needs to be built over the actual pixel values. A simple, but certainly frequent use case is to find tiles that contain pixels of certain value, e. g., a value from which one can deduce an error in the photographed material. For this use case a global min-max index that stores the minimum and maximum pixel value for each tile is useful. For a query looking for tiles with value `x`, the index can be consulted to quickly find the tiles that have a minimum value less or equal to `x` and a maximum value greater or equal to `x`. Additionally, indexes can be built for pixel combinations. Such combinations could be a pattern of interest, e. g., a ship on the ocean in satellite images. In the index, all tiles containing this pattern can be stored.

3.2 Big Data Processing Platforms

Relational DBMSs have been extensively studied and improved over the decades since the release of System R [9] in the 1970s. Over time, the systems and the underlying algorithms have been adapted to handle the ever growing data sets. When companies and organizations started to store (almost) all information generated or received in their systems, the Big Data era began. Although DBMSs are capable of handling large data sets by running distributed and parallel instances in a cluster, the unstructured data generated in, e. g., log files and the manifold schemata from data sets from

various sources required new ways of processing. Additionally, SQL is not suitable for iterative algorithms and model creation to gain new insights.

In 2004 the MapReduce programming model was introduced by Google [28]. In 2008 Yahoo presented Hadoop MapReduce[5], an open source implementation of this programming model which was rapidly established as *the* platform for Big Data processing. Companies, such as MapR, contribute to Hadoop, but also run commercial spin-offs, e. g., for Amazon Elastic MapReduce.

3.2.1 Hadoop MapReduce

The Hadoop framework is based on a shared nothing architecture: a cluster of commodity computers which are connected over network. Hadoop employs data parallelism, meaning that all nodes in the cluster perform the same task independently from each other on their own chunk of the large data set. A task is for example the execution of the `map` or `reduce` function.

Execution Model The core notion of MapReduce is that programmers have to implement only the two functions: `map` and `reduce`. The `map` function is used to extract required information from a possibly unstructured input element and to transform the input into a format that can be aggregated by `reduce`. Those functions are well known from functional programming languages like Erlang or Haskell. In functional programming languages, `map` and `reduce` are defined as higher-order functions on lists. For `map` a programmer has to pass in a (lambda) function that accepts one element from the list and returns a value from the same or another type. With `reduce`, the values in a list can be aggregated into a single value and thus, the function passed as a parameter value to `reduce` must take two elements from the list as parameter. In functional programming, `reduce` is also known as `fold`, or both coexist with slight differences.

In Hadoop terminology, in the *map phase* a *mapper* task executes the `map` function provided by the programmer. This means that the programmer does not need to care about the parallelism, but just about the processing of one element of the data set to extract required fields or values. Accordingly, in the *reduce phase* the *reducer* task executes the provided `reduce` function. In between the map and reduce phases, another crucial phase exists: *shuffle & sort*. Fig. 3.8 shows the general flow and phases and visualizes the parallel execution of the provided `map` and `reduce` functions in the map and reduce phases, respectively.

MapReduce is built on key–value pairs. The map function gets a key and a value as input (from the framework), processes it, and outputs no, one, or many other key–value pairs. The types of the output keys and values may be different from the types in the input pair:

$$map(k_1, v_1) \rightarrow listof(k_2, v_2)$$

[5] `https://hadoop.apache.org/`

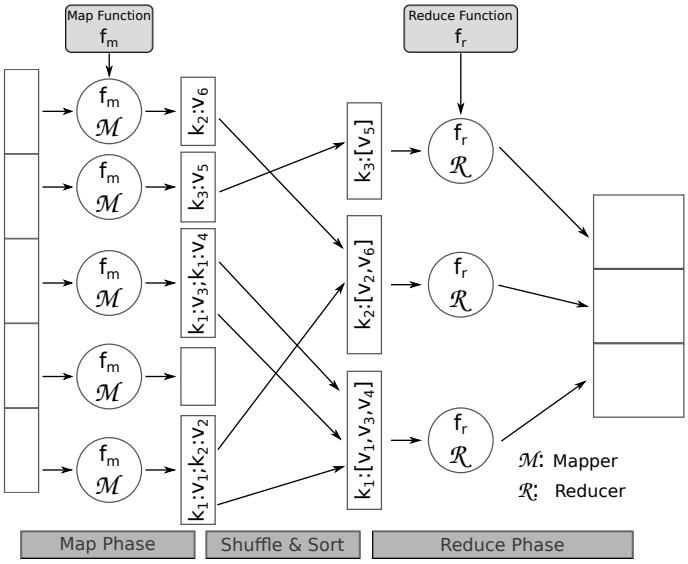


Figure 3.8: Phases in the MapReduce programming model.

As shown in Fig. 3.8, the output of the map phase is put into the shuffle & sort phase. In this phase, all result tuples from the map functions are grouped by their key and for each group, the list of corresponding values is created. This phase is named shuffle, because data from all nodes is sent and exchanged among all nodes, so that it is basically redistributed. After shuffling (and sorting the groups), for each group the group key and the according list of values is sent to a reduce task that executes the user supplied `reduce` function to aggregate the values. Therefore, the `reduce` function must follow the given signature:

$$reduce(k_2, listof(v_2)) \rightarrow listof(v3)$$

A MapReduce program that solves a complex problem will consist of many cycles of the map and reduce phase, where in each cycle another `map` and `reduce` function will be executed that will further process the output of the previous cycle.

Many large companies run a (Hadoop) MapReduce cluster with hundreds of machines that permanently move and process large data sets[6]. With that many commodity hardware involved, failures occur quite frequently (disk crashes, memory errors, memory limit exceeded, etc.). To achieve tolerance against such failures, Hadoop writes all intermediate results, i. e., the outputs of the map, shuffle, and reduce phases, to disk. Thus, if, e. g., a reduce task fails because the underlying hardware is defect, the framework will start this task again on another node and it will only have to read the intermediate result from disk instead of having to execute the map and shuffle again to obtain its input.

Fault tolerance is further accomplished by using the Hadoop Distributed File System (HDFS). HDFS connects all nodes in the Hadoop cluster into one logical file system. A file in HDFS is stored as a sequence of blocks and each block is replicated to several nodes in the cluster. By default a block is 128 MiB and replicated two times. These values can be configured when setting up the HDFS. A task of the

[6] `https://wiki.apache.org/hadoop/PoweredBy`

Hadoop program processes one block and when a node has failed and the task needs to be restarted, this is done on a node that stores a replica of the respective block.

A cluster should, of course, not only run a single program (commonly referred to as *job*) at a time, but rather share the resources among all submitted jobs so that multiple jobs can run in parallel to increase throughput. This requires that resources, such as nodes, CPUs on these nodes, RAM, etc. are assigned to a task. The management and assignment of resources in the cluster to jobs and tasks is done in the YARN component. YARN is responsible for the management part in Hadoop MapReduce, i. e., besides the allocation of resources this is the actual execution of the MapReduce programs as well as tracking the nodes' states.

Data Partitioning By default, Hadoop starts a reduce task for each distinct key in the output of the map phase. The assignment from the key, which can be of any type, to a reducer with a numeric ID, is done using, e. g., a hash partitioner. The framework calls the `getPartition` function of the configured partitioner for each key to know on which reducer it should be further processed. Users can configure Hadoop to use another, potentially self-implemented partitioner class to obtain more control over the execution.

3.2.2 Apache Spark

A big advantage of the MapReduce platform is the fault tolerant parallel processing of large data sets. However, as discussed in the previous section, fault tolerance is achieved by writing the results of the mappers to disk where it is loaded again by the subsequent reducers. As is known, writing to and reading from disk is rather slow, compared to processing data in-memory (RAM). The Apache Spark[7] platform tries to solve this issue with an in-memory execution model.

Resilient Distributed Dataset In its core, Spark uses a data structure called Resilient Distributed Dataset (RDD). An RDD is an immutable data structure and abstracts the distributed nature and processing. Thus, a developer only works with RDD objects and applies operations to them. Internally, RDDs are partitioned so that every node only processes its assigned partition. If more partitions than worker nodes exist, a node receives another partition when it has finished processing one partition.

Operations performed on an RDD are divided into *transformations* and *actions*. Transformations are operations that create a new RDD from an existing one, whereas actions compute a value from the content. The transformations in Spark are lazy: The actual computation of a transformation result is deferred until an action is executed and needs the result of that transformation. To achieve this, the RDDs store the function to apply to the data and also keep a reference to their input RDD.

This behavior helps to achieve fault tolerance. Since an RDD also stores its lineage, i. e., information about its parent RDD from which it was created, the result can be recomputed in case of a node failure. If a failure occurs and the result for a partition of an RDD is lost for whatever reason, the Spark engine will simply restart

[7] `https://spark.apache.org`

Listing 3 A Scala Spark program using Resilient Distributed Datasets.

```
1 case class Event(id: String, description: String, date: Date, type: EventType)
2 val sc: SparkContext = ...
3 val raw = sc.textFile("/data/events.csv")
4 val count = raw.map(line => line.split(";"))
5                .map(array => Event(array(0).toInt, array(1), new
                 ↪  Date(array(2)), EventType(array(3)))
6                .filter(_.type == EventType.SPORTS).count()
```

the computation for that partition, triggering the computation of the respective partitions in the parent RDDs as well.

Listing 3 shows a sample Scala program using Spark's RDD application programming interface (API). The program given in Listing 3 is called the *driver* program which starts the processing and will receive the final results from the actions. The Spark context variable `sc` is used to interact with the Spark engine, set and get configurations, and also to load (text) files from persistent storage, such as HDFS and create an RDD instance representing the file content. For text files, every line in that file will be one entry of type `string` in the RDD. On this RDD, three transformations are applied one after the other. First, the strings in the RDD are split by a delimiter character (here, a semicolon) using the `map` operation which accepts a function to apply to each element in the RDD, resulting in an RDD with each element being an array of strings. Next, from these arrays instances of the user defined class `Event` are created, again using `map`. After that, the events are filtered so that only *sport* events are left. Since these were only transformations, no computation will have been performed so far. Only when the `count` action is encountered, the computation is triggered and only now the file will be loaded from storage.

As stated before, a node processes only a partition of the data set. To reduce communication among the nodes, a node will try to apply as many transformations as possible to its partition. Only when it needs data from another partition, e. g., to compute a `group by` or a `join`, data will be shuffled over the network to other nodes.

By default, Spark uses a hash partitioner to create the partitions, e. g., by calling the `repartition` method. However, the RDD API allows developers to apply a self-implemented partitioner that for example is aware of certain data characteristics. This is particularly useful for spatial data processing since a partition might, e. g., be a region on the earth's surface or a temporal interval. We will investigate how spatial and temporal data partitioning can be utilized for spatio-temporal data processing in Sections 3.2.2 and 5.2.2.

Internals of the RDD model To understand the execution details of Spark programs, we need to look closer on the internal implementation of RDDs. An RDD is an abstract class with basically two fields: the Spark context for the program the RDD is used in and a list of dependencies. A `Dependency` in Spark encodes how a partition in the result of an RDD is derived from the partitions in the parent RDD. For example, a one-to-one dependency means that no shuffle is required to compute the result of this RDD from its parent partition.

Typically, in a one-to-one dependency, partition with index i is also derived from partition i in the parent. However, this is not a requirement and the partition may depend on any other partition. We will make use of this behavior for the partition pruning later.

The RDD class has two abstract methods, namely `compute` and `getPartitions`. The task of `getPartitions` is to tell the Spark engine about the partitions comprising the RDD. It will be called only once by the engine for planning purposes and returns an array with instances of `Partition`. A `Partition` is identified only by its index number. With this partition ID and the type of dependency (one-to-one or shuffle) the engine keeps track which partition is needed by which worker node. The actual computation of an RDD is performed in the `compute` method, which gets a partition as well as a `TaskContext` object as input parameters. Since the transformations in Spark are supposed to be pipelined, the internal execution needs to be pipelined. This pipelining is achieved using iterators, i. e., the RDD implementation follows the Volcano iterator model [40] used in many relational DBMSs. Thus, the `compute` method has to return an iterator over its result elements. The input data is fetched by getting the iterator from the parent, based on the given partition and task context. A concrete RDD implementation can perform its operation on this input iterator and return its result iterator. Only an action will start consuming the iterator and start the whole execution.

An example for a concrete RDD implementations is the class `MapPartitionsRDD` which is e. g., created for `map` and `filter` transformations. Both of these take a function as input. The `MapPartitionsRDD` class stores this function and in its `compute` method it will be applied to every element in the input iterator.

Data Partitioning When loading data from HDFS, Spark generates partitions so that each partition contains the same amount of data (records or bytes). Operations such as `reduceByKey` however, need to repartition the data in order to collect all values with the same key into a group. For this, Spark uses a hash partitioner that assigns objects to partitions by applying a hash function to the key element. Another partitioner used in Spark is the range partitioner that can be used to collect ranges of keys into the same partition.

The RDD API allows to create own partitioners by extending the abstract base `Partitioner` class. The partitioners can be applied only to RDDs with a key–value schema, since the decision to which partition an object belongs to is always based on its key. A partitioner must implement the `getPartition` method that accepts one argument – the key. As a result this method returns an integer indicating the index number of the partition that object is assigned to.

The partitioning is performed during a shuffle, represented by `ShuffleRDD`s. Thus, partitioning is not performed by a single, but in parallel on the nodes during execution of the program. If the partitioner needs to maintain a state during execution, the state is present on every involved worker node and must therefore be synchronized (merged).

In this work we will make use of the partitioner API to implement the discussed partitioning strategies. Note, an RDD carries an attribute with a reference to the partitioner that was used to create that RDD.

Spark SQL and Dataframes RDDs can be used with Scala, Java, and Python. This allows a wide range of developers to use the platform and lets them create complex programs to solve their data processing tasks. However, with the increased prevalence of Big Data processing in all kinds of areas, support of traditional SQL queries was needed: Data has simply been moved onto this platform and existing queries should be re-used.

SparkSQL was introduced as part of the Spark engine to execute SQL queries on data stored as files in HDFS. SparkSQL makes use of `Dataset`s and `Dataframe`s as layers above the core RDDs, that provide the engine with more information about the types and operations. An RDD only has one generic type parameter, that may be a single Integer or an abstract data type, that cannot be exploited by the execution engine for optimizations. `Dataframe`s, however, have typed columns similar to a relation and operations on these columns. This gives the execution engine information about the data (types) and the operations executed so that it can optimize them. Operations on `Dataframe`s resemble SQL operations, e. g., `where`, `select`, etc., but SQL strings can also be processed, of course.

Internally, operations on `Dataset`s and `Dataframe`s are translated into operations on RDDs. The `Dataset` and `Dataframe` APIs, however, are of higher level than the RDD API so that developers have less possibilities to influence execution, albeit SparkSQL allows to add own operations, rewriting strategies, and data types.

Spark further allows to process data streams using so called *DStreams* that consist of a sequence of micro batch RDDs or the higher level *structured streaming* on basis of `Dataset` and `Dataframe`.

3.2.3 Other Platforms

Besides Hadoop MapReduce and Apache Spark, several other platforms exist to process large data sets.

Apache Flink[8], which has its origins in the Stratosphere project [5], is often compared to Spark as it also employs an in-memory execution model. Like Spark, it can be used in YARN (and Mesos) clusters to execute distributed parallel tasks. Flink is primarily designed to process streams of data, but it can also be used for batch processing. Besides reading from static files from various sources, Flink has connectors for Kafka, RabbitMQ, and Twitter streams. The core data structure in the streaming component are `DataStreams` on which operations are applied (similar to Spark's RDDs).

Flink can also apply a partitioning, which however directly influences the parallelism of the query. One can only create as much partitions as there are cores/executors in the system.

Spark and Flink typically load their data as files from HDFS and with SparkSQL user can run SQL queries on this data. However, with the need for SQL support on the Hadoop platform, other (database) systems emerged. HBase[9] is a NoSQL engine on the Hadoop platform. It is designed after Google's Bigtable [21] and provides

[8] `https://flink.apache.org/`
[9] `https://hbase.apache.org/`

low latency data access and updates. The Apache Phoenix[10] project creates a SQL interface for HBase including ACID transaction properties. Hive[11] is a warehouse on the Hadoop platform also with SQL support. It is often used in write-once, read-often scenarios but also provides ACID transactions.

The Apache Impala[12] project is an SQL engine that uses the Hadoop infrastructure to execute the query in parallel. It is written in C++ and consists of three main parts: the core daemon that runs on every data node and executes the query as well as writes to the files, the StateStore that collects and monitors the health status of each daemon, and the Catalog Service that provides meta information. Another SQL-on-Hadoop engine is *Actian Vector in Hadoop* (VectorH) [26] that uses the standalone Vector technology in single nodes for parallel vectorized processing and tightly integrates with HDFS to find nodes with replicas of the required blocks. This way they can guarantee local I/O operations.

[10] `https://phoenix.apache.org/`
[11] `https://hive.apache.org/`
[12] `https://impala.apache.org/`

Chapter 4
Related Work

Spatial and spatio-temporal data processing has been of high interest since computer programs were used to store, manage, and query information. All major relational DBMSs have built-in support (Microsoft SQLServer, MySQL/MariaDB) or extensions (Oracle Spatial, IBM Db2 Spatial Extender, PostGIS for PostgreSQL) for managing spatial data. These database systems and extensions also provide raster data types and operations. Besides these all-purpose DBMS with support for spatial data, there are a few systems specifically designed for multidimensional array data (raster data) processing . The most prominent systems are RasDaMan [11] by Baumann et al. and SciDB [98] by Stonebraker et al. Like SciDB, TileDB [80] is a storage manager for scientific data supporting dense and sparse arrays.

Since the goal of this work is to add efficient support for spatio-temporal vector and raster data processing in data parallel Big Data engines, we will limit the overview of related approaches to

1. frameworks with support for at least spatial vector data or raster data and
2. that are designed for Big Data engines (Hadoop and Spark).

Initially, for the Big Data platforms such as Hadoop MapReduce, Spark, and Flink, there was no support for spatial data. To close this gap, several research projects were initiated to address the challenges and implement parallel algorithms and spatial data processing frameworks. In general, these works can be categorized based on the underlying execution platform, Hadoop or Spark (to the best of our knowledge there is no such research work based on Flink) and on how they integrate with that platform.

We will introduce those projects in the following and compare them on the basis of the following characteristics:

Data Types: Support for spatio-temporal vector data; support for raster data

Operators: Can vector and raster data be combined? Which operations are supported in general (filter, join, analytical operators)?

Partitioning & Indexing: What kind of spatial and/or temporal partitioning schemes are available? Can data be indexed for performance improvement?

Language/Interface: How can users work with the framework?

4.1 Hadoop MapReduce-based systems

SpatialHadoop The first approach to implement spatial operations as an extension for Hadoop MapReduce is SpatialHadoop [33, 31]. The framework provides spatial types for typical vector data as well as operators for range queries, k-nearest neighbors, and joins (spatial join, distance join). However, there is no support or temporal data. Furthermore, convex hulls as well as Skylines over the input data set can be computed. SpatialHadoop employs two index levels: on a global level an index partitions data across all nodes while a second index organizes data inside each partition. These indexes are used during read to eliminate records that do not contribute to the final result. As index structures, SpatialHadoop supports grid files, R-tree, and R+-trees.

HadoopGIS Another approach that extends the plain Hadoop MapReduce framework with spatial operators is HadoopGIS [2]. Similarly to SpatialHadoop, it utilizes a two level indexing: a global partition indexing and an optional local spatial indexing. The query processing engine, called RESQUE, uses these indexes to identify partitions to load and to speed up processing the required partitions. The RESQUE engine provides spatial operators like *intersects*, *contains*, *distance*, etc. The HadoopGIS system is integrated into Hive to provide a declarative SQL-like query language as user interface, namely HiveQL with spatial extensions. It supports typical operations such as range query, spatial join as well as kNN on the typical vector data types. It does not support temporal data.

GeoMesa & GeoWave GeoMesa [39] uses the key-value store Apache Accumolo as its storage backend. In GeoMesa the keys are created as a combination of the temporal value and the GeoHash[1] representation of the spatial component. It is primarily designed for point data and non-point data has to be decomposed into multiple disjoint geo-hashes, resulting in duplicated entries in the index. It seems that data always has to have a spatial and a temporal component. When querying data, only those data items intersecting with the query region are considered – based on the computed geo-hashes. GeoWave[2] is a geo-spatial index that is also based on Accumulo or HBase. Like GeoMesa, it uses space filling curves to represent multidimensional objects as 1-dimensional keys. In these systems, no explicit partitioning or spatial indexing is performed as they rely on the GeoHash and space filling curves. As language, the Contextual Query Language can be used.

Sphinx Sphinx [35] extends the Hadoop database system Impala[3] with spatial vector data processing functionality and thus, provides an SQL interface to the user. For that it adds typical spatial vector types (point, polygon, ...) and the spatial range query and filter operators to the query planner and reuses the partitioning and indexing techniques introduced in SpatialHadoop. The planner uses the statistics provided in Impala to decide whether an index should be used - if available - for execution. At runtime, Sphinx generates C++ code for the given query.

[1] `https://en.wikipedia.org/wiki/Geohash`
[2] `https://ngageoint.github.io/geowave/`
[3] `http://impala.apache.org/`

Parallel SECONDO Parallel SECONDO [67] is a distributed version of the SECONDO [44] database. In parallel SECONDO, every node maintains its own SECONDO database (*Data Server*). The actual work is distributed to these nodes using the Hadoop framework and each node runs its task in the SECONDO database. For data exchange, Parallel SECONDO uses its own file system PSFS (Parallel SECONDO File System) and uses HDFS only to store meta data about PSFS. SECONDO supports spatial networks of static or moving objects and spatial (vector) objects. These types and the operations in SECONDO can be used to answer questions like: *"Which part of the network can be reached within 50 km distance from a given network position?"* or *"Return the part of the network that lies within forest X"* [44].

Spatial Index Framework In [105] Whitmann et al. present a framework to index spatial data for the Hadoop platform that uses Quadtrees to support spatial queries. Input data is read from HDFS and used to create a partial Quadtree on each worker node. Then, using a custom partitioner, the entries of the partial trees are shuffled to be combined to a sub-tree of a complete index. On each node, a partial tree is created which is then shuffled to other nodes and combined to a sub-tree of a complete index. There is no declarative language to be used and queries seemingly have to be written as Hadoop programs. As operators, the system supports range queries and kNN queries.

4.2 Apache Spark-based systems

GeoSpark GeoSpark [112, 111] is a Java implementation to process only spatial vector data without special treatment of a temporal dimension. The project consists of three parts: GeoSpark Core, SQL, and Viz. The GeoSpark core project implements all computational functionality while GeoSpark SQL contains the necessary classes to integrate into Spark SQL. GeoSpark Viz contains some classes to visualize the content of the data set in an image. Thus, the following presents the capabilities of GeoSpark Core.

There are four different specialized RDDs: `PointRDD`, `RectangleRDD`, `CircleRDD`, and `PolygonRDD`. These special RDDs internally maintain a plain Spark RDD that contains elements of the respective type, i.e. points, rectangles, polygons, and circles. Only the generic base `SpatialRDD` can hold geometries of various types. GeoSpark supports k nearest neighbor queries, range queries, and join queries with *contains* or *intersects* predicates only and each of these queries can be executed with or without using an index. For joins, the *within distance* predicate can additionally be chosen.

As described in [112], GeoSpark supports R-trees and Quadtrees to create an ad hoc index the RDDs. A persistent index does not seem to be possible since there is no index load functionality. For partitioning the data sets, several strategies are available: R-tree, Quadtree, Hilbert-curves, as well as fixed grid partitioning. Internally, GeoSpark uses the JTS[4] library to represent the spatial objects and it uses the provided R-tree implementation for indexing and partitioning.

[4] `https://github.com/locationtech/jts`

While in earlier versions it was not possible to associate a spatial object with some payload information, the current version 1.2.0 stores the payload information serialized into a string using a fixed delimiter, which is then stored within each geometry object. In the Java API, RDDs are not created by transformations or actions, but by creating new instances of the spatial RDDs and pass in values, such as the path of the file to load and the offset of the spatial attribute within that file. Operations like joins are not implemented as methods/functions on these RDDs, but as extra classes, that accept one or more spatial RDDs as input.

SpatialSpark The goal of the SpatialSpark approach described in [110] is to provide a parallel join technique for large spatial vector data sets with the main focus on parallel hardware like multi-core CPUs and GPUs. To compute a join, the complete right relation is indexed using an R-tree broadcasted to all workers. Then, all items of the left relation are probed against that R-tree to find join partners. If the right relation does not fit into memory, SpatialSpark provides Fixed Grid Partitioning, Binary Space Partitioning, and Sort Tile Partitioning with and without using an R-tree as index [110]. As filter operation, SpatialSpark supports range queries with the predicates *contains*, *within* (*containedBy*), and *withinDistance*. The partitioning techniques from above, however, can not be used for filter operations. When querying a persistent index for these range queries the *intersects* predicate is compulsorily used. While SpatialSpark provides spatial operations for Spark, the core work of the authors is to integrate spatial operations into Impala. Internally, they expect RDDs with an ID and a geometry object, which are processed when calling the specific query object (like `RangeQuery` or `BroadcastSpatialJoin`).

LocationSpark LocationSpark [99] supports various query types (range queries, kNN search, joins, ...) via a Scala API. LocationSpark includes a query scheduler that analyzes a sample of the input data and uses the collected statistics for adaptive workload-aware partitioning [99]. That is, the a cost model decides if a partition contains too many objects and initiates a repartitioning step if needed. However, it supports only point and rectangle types and thus, is not usable for many application scenarios. The spatial data is represented in a `SpatialRDD` consisting of tuples with a geometry and a payload field. Data can be partitioned using a grid or Quadtree. As index structures, R-trees, Quadtrees, IR-trees are supported.

GeoTrellis & RasterFrames The only platform that is capable of processing raster data is GeoTrellis. GeoTrellis first provided distributed and parallel raster data processing by leveraging its own engine using Akka to communicate between worker nodes. In the current release, they moved from Akka to Spark and provide a `RasterRDD` type to represent a raster data set consisting of tiles. It is possible to perform various operations on `RasterRDD`s, such as joins, filters, or visualization. The joins, however, are not using spatial predicates but use a key based on the row and column index of a tile, which represents a normal Spark join. The RasterFrames project adds support for raster data to the Spark Dataframes API. With RasterFrames it is possible to apply raster operations as well as filters using vector objects. To the best of our knowledge, for GeoTrellis and RasterFrames no (scientific) publications exist.

Table 4.1: Feature comparison of Spark-based spatial data processing platforms. *V: vector data, R: raster data*

Feature	STARK	GeoSpark	Spatial Spark	Location Spark	Simba	Raster Frames
Raster	✓	–	–	–	–	✓
Temporal	✓	–	–	–	–	–
Filter	V–V R–V	V–V	V–V	V–V	V–V	R-V
Joins	V–V R–R R–V	V–V	V–V	V–V	V–V	R–R
Analytics	kNN Skyline DBSCAN	kNN	–	kNN k-Means	kNN	
Partitioner	fixed grid, cost-based grid, R-tree, Quadtree	R-tree, Quadtree, Voronoi	fixed grid, binary split, Sort-Tile	grid, region R-tree	STR	
Indexes	R-tree, Quadtree	R-tree, Quadtree	R-tree	R-tree, Quadtree	R-tree	–
API/Lang.	Scala, SQL, Pig Latin	Java,SQL	Java	Scala	SQL	SQL

Further projects Besides these, there are some other projects, that all process spatial vector data: Simba [107, 108] allows to create spatial data processing programs using SparkSQL or via Spark Dataframe API and optimizes these queries employing a cost-based optimization module as an extension to SparkSQL's Catalyst optimizer.

Another project is Magellan[5], for which the Git repository claims that it "deeply leverages modern database techniques like efficient data layout, code generation and query optimization in order to optimize geospatial queries". Magellan supports various geometry types, such as points, linestrings and polygons. As operations, spatial filters and joins are supported. A Z-order curve is used to partition the data. Magellan extends SparkSQL and provides its functionality as user-defined functions (UDFs) to the system. To the best of our knowledge, besides the Git repository and blog posts, no scientific publication exists for this system.

Table 4.1 shows a feature comparison of the previously mentioned projects and our STARK. In this table, `V` stands for vector data, `R` for raster data. For filter and join operations, the table lists which combination of these two types are allowed in the respective operators.

This literature research showed that the demand for spatial and spatio-temporal data processing platforms has led to a number of projects tackling this problem.

[5] `https://github.com/harsha2010/magellan`

Besides processing spatial vector data only, users also need to process their large raster data sets using Big Data technologies, resulting in the GeoTrellis and RasterFrames projects. Although many real world data sets containing spatial data often also include temporal values, none of the described systems has explicit support for this data type.

All of these systems include at least spatial filter and join operation. However, besides these basic operations, more advanced analytical operations are needed as well, such as the kNN search or clustering. Within this work, we build a spatial data processing engine that is also aware of the time-related values. Furthermore, we close the gap between vector and raster data processing engines and aim to allow the combination of these two types of spatial data.

Chapter 5
Framework Design

In this chapter we will outline and discuss the general concepts as a basis of a framework processing raster and vector data sets at large scale. The framework will address the requirements outlined in the previous chapters. The target platform for this framework is Apache Spark using HDFS. However, the concepts of operators, partitioning, and indexing can as well be applied to other data parallel systems.

5.1 Storage Level

In this section we discuss the possibilities for supporting vector and raster data on the storage/file system level. It is the lowest level discussed in this thesis and layers above will benefit from the features it provides.

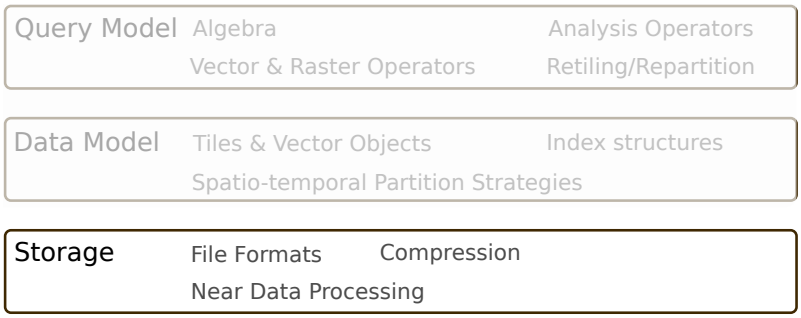


Figure 5.1: The storage level is the lowest level discussed in this thesis.

5.1.1 Storage Formats

Vector Data Vector data objects are typically stored as part of some data set containing many different attributes. As an example, in the GDELT[1] data set events from all around the world are recorded. Here, every record has attributes describing the involved actors, country codes as strings, the type of the event, the time of occurrence, the location and many more. A file format or storage layout can be optimized for different types of queries. In the context of this work, data on disk/HDFS can be organized respecting the spatial and/or temporal neighborhood of the objects: objects that are located near to each other should be stored in a way so that they can be loaded together. This is tightly coupled with the partitioning the data parallel systems perform. Thus, one approach is to materialize the spatial and spatio-temporal partitioning meta information (cf. Section 5.2.2) and exploit this for additional speed-up, as discussed below.

[1] `https://www.gdeltproject.org/`

$$\begin{bmatrix} [z_{11}^{t_1}, z_{12}^{t_1}, z_{13}^{t_1}, \ldots, z_{kn}^{t_1}], \\ [z_{11}^{t_2}, z_{12}^{t_2}, z_{13}^{t_2}, \ldots, z_{kn}^{t_2}] \\ \ldots \\ [z_{11}^{t_j}, z_{12}^{t_j}, z_{13}^{t_j}, \ldots, z_{kn}^{t_j}] \end{bmatrix}$$

Figure 5.2: Storing one image after another for fast access to full images.

$$\begin{bmatrix} [z_{11}^{t_1}, z_{11}^{t_2}, z_{11}^{t_3}, \ldots, z_{11}^{t_j}], \\ [z_{12}^{t_1}, z_{12}^{t_2}, z_{12}^{t_3}, \ldots, z_{12}^{t_j}] \\ \ldots \\ [z_{kn}^{t_1}, z_{kn}^{t_2}, z_{kn}^{t_3}, \ldots, z_{kn}^{t_j}] \end{bmatrix}$$

Figure 5.3: Pixel-wise storage: for every pixel, store all values.

Raster Data As described in Chapter 2, an application of raster data is time series analysis where for the same region, several images are taken over time. Here, depending on the actual analysis task, different internal file organization strategies can be employed:

1. store one image after another
2. store information pixel-wise, i. e., first all values for the first pixel in all images, then for the second, and so on.

The two options are depicted in Figs. 5.2 and 5.3, respectively. While in the first organization strategy loading one image completely, e. g., for visualization, is faster than in the second one, the second strategy is suitable for analysis tasks that need to perform calculations over all values existing for a pixel.

Using simple compression formats, such as run-length encoding, is often not possible, because there simply is no long run of the same `double` value. Except for the pixel-wise storage format: Since in many scenarios not every pixel will change over time, compression might yield some benefit. Advanced compression techniques are e. g., LERC [13], LZW [104], or Deflate [29]. Though, we will not consider compression in the context of this thesis and leave it for future work.

5.1.2 Near Data Processing

The storage formats utilize data as well as query characteristics to speed up the execution. However, the data still needs to be read completely from storage into the worker nodes and is processed there. To further speed up query-response time and reduce overall resource utilization, it is desirable to reduce the amount of data loaded into memory.

Near data processing means to process data near the storage hardware, i. e., disk (more common: SSD). Recently, several approaches, like [106, 12] have been published that use specialized hardware, such as Field Programmable Gate Arrays (FPGAs), to execute database operators on data objects while they are transferred from disk into RAM – hence, before they reach the CPU.

Though it would be possible to add additional hardware to a cluster and use it for this kind of work, we can also perform *logical* near data processing. As discussed above, the storage format employs the locality of the data, e. g., by writing partitioning information to disk. Thus, when a program or query loads data from disk

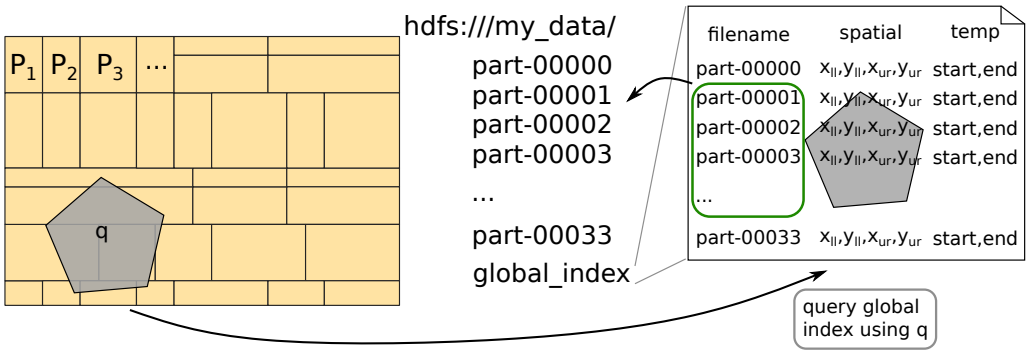


Figure 5.4: Logical Near Data Processing in HDFS: loading only partitions that intersect with a query region.

in order to perform some (spatio-temporal) filter operation later on, this filter can be pushed down into the load function. The function will then decide which parts of the data set match with the given value (range) according to the filter predicate and return only these partitions. In this way, it is possible to reduce the amount of resources (RAM, worker nodes) needed to processes the current query and leave more resources to other concurrently running queries.

Let $P = \{P_1, P_2, \ldots, P_n\}$ be the set of all partitions of a data set D and q a spatio-temporal region used as a filter range. Then, the partitions that contain result objects for that filter operation regarding region q and a predicate Φ can be given as:

$$\sigma_\Phi = \{P_i | P_i \in P \wedge \Phi(P_i, q)\}$$

The idea is depicted in Fig. 5.4: for every partition, a separate file is created in the HDFS. The meta information about the spatio-temporal partition bounds are written to an additional `global_index`. This global index can be organized as a spatio-temporal index structure, e. g., an R-tree variant, or as a plain file. In the latter case, a linear search has to be performed. This, however, is justifiable in most cases since the number of partitions will be rather small, compared to the number of data objects. When a filter on the data set is performed, it can be pushed down into the loader function responsible to read the partitions and return them to the processing engine. This function first reads the global index and identifies those partitions that intersect with the query region q. Subsequently, the file names of those partitions are forwarded to the native load function of the execution engine in order to load only those and not the complete data set. In this way, the load on the cluster can be reduced drastically as fewer worker nodes will be needed for processing.

A similar approach can be implemented for values of raster data sets. Here, in addition to the spatial (and temporal) components, we can also consider the actual values and create an index storing Small Materialized Aggregates (SMA), such as the minimum and maximum pixel value in each partition. This information can later be used to speed up according filter and join operations.

5.2 Data Model

This section deals with the data types needed representing the spatio-temporal vector and raster data. We will further discuss the partitioning as well as indexing strategies with the goal to speed up the query execution time.

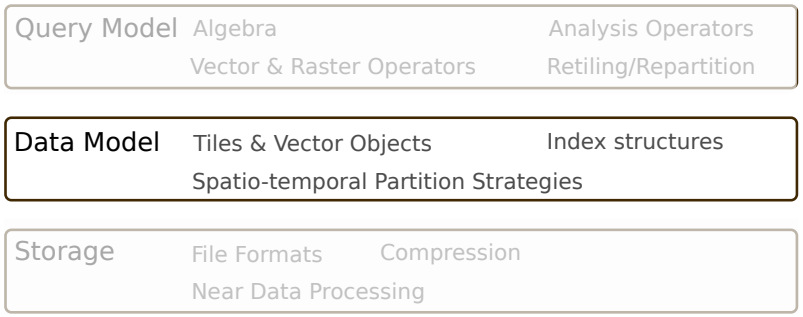


Figure 5.5: The data model contains definitions to represent vector and raster data as well as strategies to partition and index them.

5.2.1 Data Types

Vector Data Spatio-temporal vector data needs to be represented as an object with a location and a time. In this framework, we consider the spatio-temporal feature as a single property of a record. Thus, the spatial and temporal information is represented by a single object. Since our main focus is on spatial data, the time information is optional and may be left empty. In that case, the object does not have an associated time information, although a time may be present in some other attribute in the schema, of course.

The data type which contains the spatial as well as temporal information is named `stobject`. The type itself as well as its fields can be given as follows in EBNF:

```
stobject = geo, [time] ;

geo = POINT | LINESTRING | POLYGON |
      MULTIPOINT | MULTILINESTRING | MULTIPOLYGON ;

time = instant | interval ;

instant = LONG | DATE | TIME | DATETIME;
interval = instant, [instant] ;
```

This means an `stobject` is an abstract data type with two fields: a geometry object that represents a vector geometry and an optional time. The `geo` field must allow to hold all known vector data types (points, polygons, etc.) and therefore, a base type for all geometries is needed. These geometry types require their own data structures to be represented, which can be taken from some external library. The `time` field

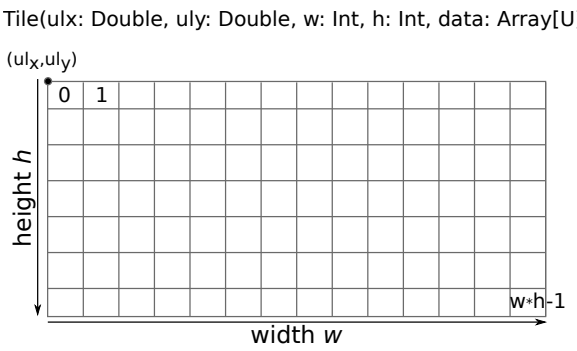


Figure 5.6: Tile representation with position information.

must allow to hold instances as well as intervals. Intervals always have a known start time, which is expressed as an `instant`, but their end may be undefined to represent open-ended intervals. An instant can be created from a number representing seconds elapsed since the Unix epoch or an explicit date. Such types are available in (almost) all programming languages.

Raster Data Representing raster data requires to

1. model the raster tiles as container types with a set of cells,
2. allow various data types to be used for cell values,
3. include meta information about a tile in the tile itself, and
4. ensure tiles can be used together with vector objects in spatial predicates.

Raster data is stored using a `tile` type. An instance of this type represents a single tile in the raster. A tile is always an rectangle and for each tile we store its position information, i. e., the coordinate values of the upper left corner as well its width and height (in terms of number of pixels), as shown in Fig. 5.6. Storing the size information for each tile is important as it enables us to implement operators, such as filters, that might match only a part of a tile which results in a set of irregular sized tiles. Since computations are performed in parallel on different worker nodes, meta information would have otherwise be made available to all nodes and be updated when this information changes and tiles are created/deleted. Along with the position and extent information, a tile stores the contained pixel values. All values are of one certain type and usually all tiles in a raster data set store the same type. However, raster data sets can be of any type: sensors may record `double` values, images from cameras create RGB or integer values, and synthetically created data sets may consist of user defined abstract types.

Similar to `NULL` in relational databases and programming languages, a special value to indicate a missing value is needed. We rely on existing special values of built-in types of programming languages to mark such a missing value. For floating point values, this can be the `not a number` constant, or negative (or positive) infinity. For other types, the `NULL` reference of the programming language is used.

Multi-band tiles can be supported using the fact that a tile can hold user defined types. For multi-band data, this type can be an array of values, where every entry in this array is a value in one band.

Collections of Vector and Raster Objects Data sets may contain a plethora of vector and raster objects, respectively. Thus, the objects need to be collected in a container structure which can be processed in parallel in a cluster environment. In Apache Spark, the platform used for the prototype implementation (cf. Chapter 6), this structure is an RDD. In order to provide operations on this collection of objects specialized collections, i. e., RDDs, are needed. The specialized RDDs must provide the same operations as traditional RDDs, but additionally implement the operations for spatio-temporal vector objects and raster tiles, respectively. Thus, we extend the original RDD class of Apache Spark and create specialized classes that implement the respective operations (filter and join). The partitioning and indexing to apply will be configurable. The operations are discussed below in Section 5.3.2.

5.2.2 Spatio-temporal Partitioning

Data parallel systems need to partition the data according to some strategy to assign parts of the input data set to worker nodes. Strategies such as hash partitioning or round robin are well known and widely used within such systems. However, these strategies do not maintain the locality of the data, meaning that spatial objects in the input data set that are near to each other will likely be assigned to different partitions. Thus, the area spanned by the elements inside a partition will overlap with many if not all other (areas of the other) partitions. While this itself is no problem for query execution in general, it inhibits the exploitation of the partition area to optimize query execution.

A spatial (and temporal) partitioning method assigns an object to a partition based on its position in space and/or time so that nearby elements will likely be assigned to the same partition. We can distinguish three partitioning categories:

Spatial: In spatial-only partitioning only the spatial feature of the `stobject`s are considered. This can also be applied to partition raster data sets.

Temporal: Analogously to spatial partitioning, in temporal-only partitioning only the temporal feature is used to assign objects to partitions. Since in our design the temporal information is optional, objects without a temporal part cannot be assigned to any partition and should either be added to a meta partition or an error is generated.

Spatio-temporal: Spatio-temporal partitioning applies both strategies in a single partitioning step: first, the objects are partitioned using a spatial partitioning approach and then, every generated spatial partition is again partitioned using a temporal partitioner. It is, of course, possible to first apply a temporal partitioner and after that the spatial partitioning. While the order in which the partitioners are applied will most likely not have an impact on the load balance among the nodes, it is very likely to have an impact on the resulting query performance.

Partition Pruning A spatial or temporal partition is defined by an area or range it covers, rather than, e. g., a hash bucket ID.

When the bounds of a partition are known, this information can be used during query execution to decide which partitions, or in the case of spatial joins, combination of partitions can contribute to the final result. This is very similar to the logical near data processing discussed in Section 5.1.2. Though, in this case we apply the pruning on the partitions in memory. This is useful when the partition information is not present on disk, e. g., when partitioning was applied only after data was loaded. The pruning can be used for filter, kNN search as well as for joins. Assume two input data sets R and S, each partitioned into m and n partitions respectively. If there was no information about the spatial or temporal bounds of the partitions and a spatio-temporal join is performed, $m \times n$ combinations of partitions would have to be created and all objects in the partitions would have to be tested to match the join condition. However, if the spatial or temporal bounds of the partitions are known, only those partitions that intersect with each other need to be combined.

Partitioning Strategies We can distinguish two kinds of partitioning methods: with duplicate generation and without. A partitioning with duplicate generation means that an element is assigned into multiple partitions. This may be the case for geometries that cover a region (polygons) or time intervals (cf. Section 5.2.1). Here, a respective object is replicated into all partitions it intersects with. Duplicates may also arise when the boundaries of the partitions overlap, i. e., the areas or ranges covered by the partitions are not disjunct and objects are assigned to all partitions. However, for spatial and temporal data processing there is no reason to generate non-disjunct partitions and thus, in the following we assume that partition bounds are always calculated so that disjunct partitions are created. Since this replication might cause duplicate query result records, e. g., for spatial join operations, a de-duplication step is required afterwards. For identifying and removing such duplicates, the reference point method proposed in [30] can be used.

To avoid the additional de-duplication step, a partitioning without duplicate generation is required. Since we assume disjunct partitions, points and instants are always contained in exactly one partition. Other types, i. e., linestrings, polygons, and intervals, need to be assigned to partitions based on a single (characteristic) point.

For geometries, this may be the centroid point or any interior point or point of its boundary/MBR, whereas for time intervals the start, end (if existent), or the middle could be used.

However, as discussed before, the partition bounds information is supposed to be used to prune partitions that, e. g., do not intersect with a query range. As shown in Fig. 5.7, the polygons and intervals may exceed the partition's boundary. In the figure, the three objects are each assigned to different partitions (marked by `X`). Consider a query that wants to find all polygons that contain the query point Q. If only the actual partitions' boundaries are checked, we find that Q lies within P4, which contains no objects in this example and thus, the result would be empty, because the other partitions were excluded from evaluation. Therefore, in addition to the calculated boundary, we also have to keep the *extent* of a partition. The extent

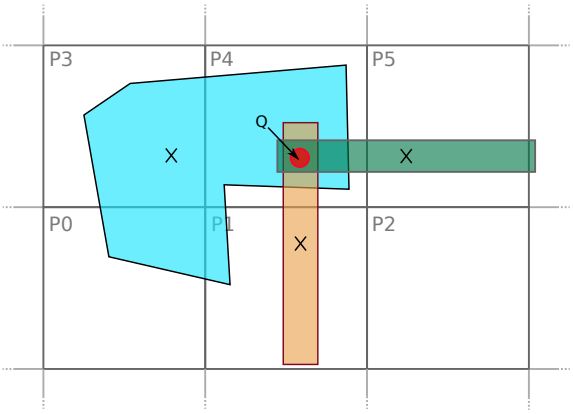


Figure 5.7: Example of objects in different partitions as results for point query (red circle).

is the MBR of the convex hull of all elements inside that respective partition and is used instead of the actual calculated partition boundary to perform the intersection check with the query range or partition from another data set.

In Section 2.3.2 we outlined four requirements for partitioning schemes: locality, reasonable number of partitions, partition size, and skew handling. In the following, we will discuss several approaches that meet these requirements differently well.

While in the one-dimensional space of temporal data only intervals are possible as shape for the partitions, in the two- or higher-dimensional space of the spatial data, several shapes can be used. However, when creating partitioning strategies one has to keep in mind that the partition bounds are computed only once, while for every object in the input data set this list of computed partitions is searched to return the containing one. If the partitions are of a polygonal shape, this check will be a point-in-polygon check which is much more expensive than a point-in-rectangle check. In the remainder of this section, different possible partitioning strategies are discussed.

Interval Partitioning To partition the data based on the temporal information only, an interval based partitioning strategy is required. In a straightforward strategy, the complete space, from the earliest (start-)time to the latest (end-)time, is divided into a number of partitions of the same length.

The fixed-length partitions might contradict the requirement of equally sized partitions and skew handling. Thus, with statistics about data distribution which can be collected over a sample of the input data set, partitions with different interval lengths but (almost) equal number of contained objects can be created.

To assign the objects (instants or intervals) to partitions (intervals), we have to decide to which partition an object belongs. However, we cannot assume that the intervals representing the temporal objects are completely contained in any partition. To achieve a definite mapping anyway, we reduce the intervals of objects to a single point (instant). Theoretically, there are at least three points that could represent an interval: its start, its end, or the middle point. Though, the end might be undefined for infinite intervals, making it also impossible to calculate a middle point. Thus, neither of them can be used. Therefore, in the interval partitioner of our framework, objects are assigned to partitions based on their starting point.

Grid Based Partitioning Similar to the fixed-length intervals approach described above, the spatial data space can be divided using a grid of equally sized partitions, represented by rectangles. For this, only the minimum and maximum values in each dimension must be known.

The number of partitions per dimension (*ppd*) is given as input parameter. The side length of the rectangle (partition) in dimension i is then calculated as:

$$length_i = \frac{|max_i - min_i|}{ppd} \tag{5.1}$$

To compute the partition a given point p belongs to, the partitioner simply has to calculate the partition ID $partitionId$. For a two dimensional scenario the formula is given as:

$$\begin{aligned} x &= \lfloor \lfloor \mid p_1 - min_1 \mid \rfloor \; / \; length_1 \rfloor \\ y &= \lfloor \lfloor \mid p_2 - min_2 \mid \rfloor \; / \; length_2 \rfloor \\ partitionId &= y * ppd + x \end{aligned} \tag{5.2}$$

This strategy can easily be adapted to a higher number of dimensions. The fixed grid partitioning is cheap to compute as it does not consider at the actual data distribution besides the minimum and maximum values. However, especially real world spatial data sets do not have a uniform distribution in most cases. They rather contain some dense regions, e. g., in cities and sparse areas (oceans, deserts) – depending on the application scenario. In such cases the fixed grid partitioning results in a few partitions containing the majority of the objects while all other partitions contain no or only some elements. The result is that some worker nodes are suffering heavy load while the others that were assigned the empty partitions idle.

Thus, in order to generate a balanced partitioning where all partitions contain the same amount of objects, a strategy that considers the actual data distribution is needed. In [55] He et al. present an approach that allows to set a maximum cost for partitions. The goal of the partitioning strategy, developed in the context of a MapReduce based DBSCAN implementation, is to generate partitions that all have (almost) the same costs. He et al. define a cost formula that takes disk accesses and loading times into account.

Based on this idea, we developed our own Binary Space Partitioning (BSP) to create balanced spatial partitions. The general idea is as follows:

1. First, the data space is divided into small quadratic cells of a fixed side length.

2. For every dimension, create a candidate partitioning by performing a binary split of the partition in the current dimension.

3. The candidate partitioning with smallest cost difference between both generated partitions is applied.

4. For both generated partitions recursively apply step 2 if:

Figure 5.8: Visualization of an example Binary Space Partitioning (solid lines) with cells (dashed lines) and data objects (points).

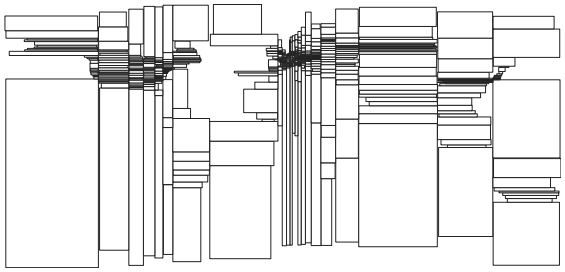


Figure 5.9: Visualization of an R-tree partitioning.

- the generated partition is longer than one cell length in at least one dimension and
- its cost is greater than the given maximum cost.

Applying this partitioning approach will result in partitions of almost equal cost, if the cell size is chosen reasonably according to the data. Figure 5.8 shows the result of a sample partitioning. To compute the costs for each candidate partitioning, the costs for each cell must be known in advance. Thus, in an additional step before the actual partitioning starts, the cell statistics are generated with an additional pass over the data. These statistics are represented in a histogram for the cells with the number of elements in each cell.

When using only a sample of the input data to compute the statistics and partitioning, it may happen during the assignment of data objects to partitions that an object is not contained in any partition. To decide to which partition this particular object belongs to, two options exist:

1. An additional *overflow* partition is introduced which will hold all elements that are not contained in any *real* partition. This overflow relation will very likely span across the complete data space, since those not contained objects spread over the complete data space and thus intersect with a large number of the real partitions. The smaller the sample of the data used to compute the partition is, the higher the probability that an object is not contained in any partition and will be assigned to the overflow partition.

2. For an object o that is not contained in any partition, the nearest partition is determined and the object is assigned to it. Additionally, the extent of that partition is updated to include the given object.

Tree-based Partitioning Spatial index structures divide the data space into (disjunct) regions and organize them in e. g., a tree structure. The R-tree and Quadtree are such tree based structures. One can utilize the bounds of the tree's nodes as partitions.

Typically, an R-tree has a capacity parameter specifying the maximum number of elements stored in a node. The capacity of the tree nodes has to be chosen

according to the desired number of partitions and number of elements per partition. This way, we can find the partition bounds in the first level, which is already the leaf level, of the tree. Therefore, the capacity c is calculated as:

$$c = \frac{input\ size}{max\ cost\ per\ partition}$$

The MBRs of the leaf nodes in the tree represent the partitions. Figure 5.9 visualizes the partitions generated by this strategy. Maintaining a tree during many inserts is time consuming and may produce nodes that do not optimally utilize the space. However, bulk loading techniques such as the Sort-Tile-Recursive (STR) packing [64] can be used especially when the tree is read-only after loading.

In a Quadtree a node always has four children, if they are not empty. Similarly to the described approach for R-trees, the bounds of the nodes on the leaf level in the Quadtree can be used as partitions. Note that since the trees are not used for searching, it does not matter that the Quadtree is not balanced. We are not interested in traversing the tree, but rather use the bounds of the generated leaf nodes as partitions.

The advantage of the tree partitioning strategies over the previously discussed grid strategies is that the tree is a dynamic data structure that adapts to the data. While the fixed grid approach did not consider the data distribution at all, the BSP needs a pass over (a sample of) the data to compute the histogram and then create the actual partitioning based on the histogram. The R-tree partitioning approach adapts to the data distribution during the build phase using the efficient STR method.

R-trees and Quadtrees are common spatial index structures and are available in commonly used libraries. However, other index structures for higher dimensional data ($d \geq 2$), such as kd-trees or grid files, can also be used for partitioning.

Like for grid partitioning the tree does not need to be built over the complete data set, which would require to load the whole data into memory or use a disk based tree which would incur additional load time because of the disk accesses.

Other Partitioning Strategies The grid- and tree-based strategies create rectangular partitions for spatial data. However, a rectangular shape is not necessarily required and alternatives might also achieve good results.

Voronoi Partitioning Voronoi diagrams divide a two (or higher) dimensional space into regions. Each region is determined by a center point which is usually – but not necessarily – member of the input data. For all points from the input data set the distance to all centers is calculated and the object is assigned to that region it has the smallest distance to. However, this strategy has the disadvantage of choosing good center points as it greatly influences the resulting partitions. If the centers are too close to each other and are not distributed over the data space, it is very likely that some partitions contain only few elements while other partitions stretch over a large area and contain a large number of objects. For the assignment of points to regions/partitions the distance needs to be calculated, which might be more expensive than a point-in-rectangle check where only double values have to be compared.

Angular Partitioning The described approaches create a general partitioning that is independent from the actual query. However, Chen, Hwang, and Wu proposed a angular partitioning in [22] especially for the application in Skyline query computing (see below) on MapReduce.

The idea is to, starting from a fix point, divide the data space into sectors around that fix point. Each sector covers an angle around the fix point. For this, the Cartesian coordinates are translated into polar coordinates, represented by a radius (distance to the fix point) and one angle Φ in the two dimensional case. The authors showed that this approach achieves good results for the Skyline query processing. However, it could also be applied in general to the input data. In this case the fix point could be, e. g., the lower left point or the mid point of the MBR of the universe (the input data). The idea of this partitioning scheme can be grasped from Fig. 5.17(a) on page 74.

Combined Spatio-temporal Partitioning The introduced strategies are either designed for temporal or spatial vector/raster data only. However, since we are dealing with spatio-temporal (vector) data, partitioning should consider all dimensions of an object.

One approach is to add the time to the spatial feature as an additional dimension. Since the grid-based and tree-based strategies can theoretically be implemented for a higher number of dimensions, they can easily deal with three-dimensional (2 spatial + 1 temporal) or four-dimensional (3 spatial + 1 temporal) objects. Treating the time as an additional dimension for the spatial objects will work, but might not achieve best results. Thus, another option is to first partition on one dimension (spatial or temporal) and after that, partition each of the resulting intermediate partitions with respect the respective other dimension.

Raster Data Partitioning The tiles in a raster data set cover a spatial region of rectangular shape and each tile carries the information about its spatial extent. Thus, the same spatial partitioning strategies as for vector data can be applied on a raster data set as well. For sparse raster data sets, i. e., where a few tiles are scattered over a large area, a cost- or tree-based approach might be useful. For regular raster data set where tiles cover the complete space, we can assume a simple grid partitioning is best.

5.2.3 Indexing

In the context of this framework, indexing can be done on two levels:

1. **Global Indexing:** An index is created globally over the complete data set. It is used to decide which partitions contribute to the query result.

2. **Local Indexing:** For every partition, a partition-local index is created over the content of that partition. It is used to speedup processing that particular partition.

The logical near data processing as discussed in Section 5.1.2 is a form of global indexing. The meta information about the partitions spatial and temporal extent can be used as a form of index. We therefore will limit the following to local indexing of partitions.

Usually, a partition is given to a worker in form of an iterator from which all elements are pulled and processed according to the query, e. g., compared with a given filter predicate. Since all data items within a partition are candidates for the current spatio-temporal predicate , they all have to be evaluated. This is done on all partitions (returned by the global index) and all records from the input, similar to a full table scan in a DBMS. If the data set is large and the operation to apply for each record is complex, execution of a query will take a lot of time, making interactive applications and ad hoc queries impractical. Thus, theoretically any in-memory spatial, temporal, or spatio-temporal index structure can be used to speed up execution.

The worker nodes have to load the index into memory and thus, it must fit into memory. This requires to partition the data in a way that the partitions do not contain too many objects in the first place. The number of objects a partition should have at maximum depends on the available RAM on the nodes and the record size.

An input data set which is, e. g., given as a CSV file does not have an index by default. However, once an index was created it might be needed in later queries as well. Therefore, two indexing modes exist: online index generation and persistent indexing.

Online Index Generation A worker node iterates over all elements in its current partition and adds the elements to an in-memory index. After evaluating the query (filter, join, ...) the index is discarded, i. e., it exists only during processing the partition. Note, that this mode is transparent to the user and other operators as the index is not exposed to any other component in the system.

Persistent Indexing Often users and groups of users work with the same data sets and use it for various purposes. Thus, discarding the index after evaluating one query is not always wanted and it should rather be materialized to storage. This mode changes the schema of the input from a list of fields to a single `Index` field. The indexed partitions can be written to storage using byte serialization and also be loaded using a platform's native methods.

Online indexing will probably have no effect for filter operations as all objects in a partition have to be loaded and put into the index before the index structure is queried for matching objects. Since all objects have to be touched anyway, the predicate could directly be evaluated without the memory overhead of an index. Nonetheless, for joins and other operations such as kNN search, the index might bring some benefit.

In persistent indexing mode the overhead of building the index before using it is not present and even filter queries can already benefit. Though, the index has to be serialized as a binary object. In Java, serialized binary objects are known to be extremely large. Thus, loading the index requires to read large binary objects,

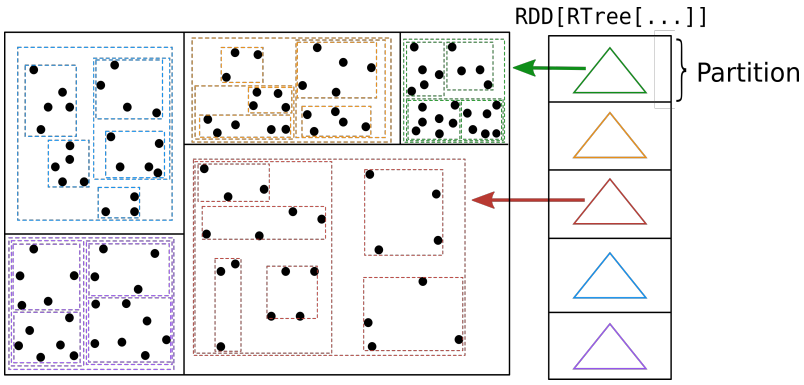


Figure 5.10: Indexing the partitions' content

which potentially has too much overhead and wipes any benefit achieved by querying the index. We will analyze this in the evaluation in Section 9.1.2. In Fig. 5.10 the indexing of the partitions' content is depicted as well as the effect on the schema. Partitioning and partition pruning described in Section 5.2.2 is orthogonal to indexing and helps reducing the data before actual computation.

5.3 Query Model

Using the previously defined data types, partitioning schemes, and indexing strategies, in this section we discuss possible operators to query the data sets.

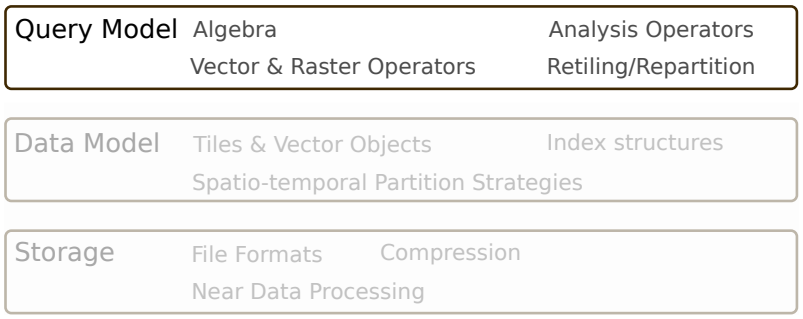


Figure 5.11: The query model provides various operations to query the data sets.

5.3.1 Operations for Spatio-temporal Vector and Raster Data

In order to implement operations on spatio-temporal vector and raster data, the definition of an algebra is needed. In [43], Güting proposed a geo-relational algebra as language to work on spatial vector objects. In combination with Allen's interval algebra [6], this algebra can be adapted for the spatio-temporal data objects in our scenario. Thus, a set of spatio-temporal objects `stobject*`, the following operations can be defined:

```
stobject* x stobject       -> stobject*  // spatio-temporal filter
                                         // (contains, intersects,
                                         // Skyline, ...)

stobject* x stobject x NUM -> stobject*  // k nearest neighbors
stobject* x stobject*      -> stobject*  // spatio-temporal join

stobject*                  -> stobject   // convex hull, largest,
                                         // smallest, ...
```

Naturally, non-spatial operations such as `stobject* -> NUM` for a `count` operation can be used as defined in [43].

The following operations are defined for single instances of `stobject`:

```
stobject x stobject        -> stobject   // extend, convex hull,
                                         // intersection
stobject x stobject        -> BOOL       // predicates: contains,
                                         // intersects, smaller
```

Functions that check the relationships (intersects, contains, greater, smaller, ...) of two such objects with each other need to consider both, their location and time information. Thus, a check of these relationships is only true, iff:

1. the check yields true for the spatial component, and
2. both temporal components are *not* defined or
3. both temporal components are defined and they also return true for the respective check.

Given as a formal expression: for two spatio-temporal objects o and p and a predicate Φ:

$$\begin{aligned}\Phi(o,p) \;\Leftrightarrow\; & \Phi_s(s(o),s(p)) \wedge (\\ & (t(o) = \bot \wedge t(p) = \bot) \vee \\ & (t(o) \neq \bot \wedge t(p) \neq \bot \wedge \Phi_t(t(o),t(p))))\end{aligned}$$

Where $s(x)$ denotes the spatial component of x, $t(x)$ the temporal component of x, Φ_s and Φ_t denote predicates that check spatial or temporal objects, respectively, and $\bot$ stands for `undefined` or `NULL`.

In addition to the spatio-temporal operators above, we also need an algebra and semantics to work with raster data sets and combine them with vector objects. In the following, `tile` describes a single tile and `tile*` a set of tiles, i. e., the raster data set. On such a set of tiles, the following operations are needed, where `U` is the type of the values in `tile`:

```
tile* x tile*        -> tile*    // raster-raster join
tile* x stobject*    -> tile*    // raster-vector join
tile* x stobject     -> tile*    // raster-vector filter

tile* x U            -> NUM      // count values
tile* x U            -> BOOL     // hasValue
tile* x U            -> tile*    // tiles with value
```

The above operations apply either a set of tiles, one or more vector objects, or a scalar value on a set of tiles. To do so, the individual tiles need to be processed as well. The required operations on single tiles are described below.

```
tile x tile          -> tile     // intersection of two tiles
tile x tile          -> BOOL     // intersects, contains, ...

tile x U             -> NUM      // count values
tile x U             -> BOOL     // hasValue
tile x U             -> tile     // arithmetic expressions
                                 // (+,-,*,/)

tile x POINT         -> U        // get value at spatial position
tile x NUM           -> U        // i-th pixel value

tile x (U -> V)      -> tile     // transform from U into type V

tile                 -> U        // min, max, avg, median
```

5.3.2 Spatio-temporal Vector and Raster Operators

In this section, we discuss the operators required for big spatio-temporal data analytics and analyze how they can benefit from parallelism and the discussed partitioning and index strategies.

Filter As described above, a filter operation on a set V of `stobject`s requires a query object q also of type `stobject` and returns all objects in the set, that match with the query object according to a given predicate Φ:

$$\sigma_{\Phi}^{VV} = \{STO | STO \in V : \Phi(STO, q)\}$$

This predicate Φ could either be given as a value from a predefined set of predicate constants or as a UDF. The implementation of the filter then tests the elements in the data set (partition) against the given query object using the predicate function. The implementation can make use of available indexes.

Partition pruning can be applied on existing partitions either by selecting the partitions to load from storage or, if the dataset has already been loaded, by processing only those partitions P that match the query predicate regarding q: $\Phi(P, q)$.

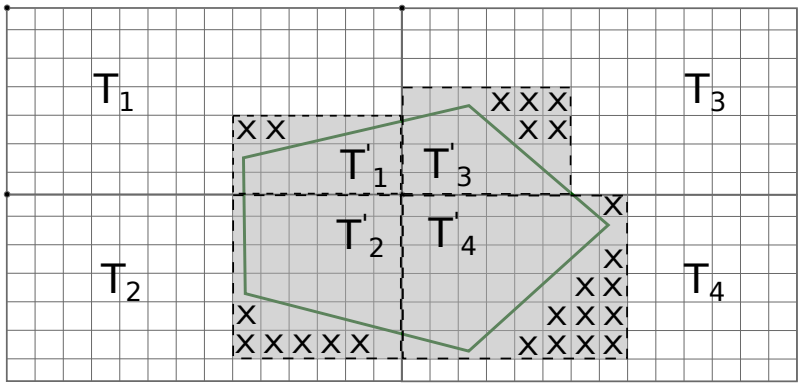


Figure 5.12: Filtering raster data with a vector polygon. The result contains new tiles with different dimensions. Pixels marked with `X` mean `NULL` or `NODATA`.

Corollar 1. *For a spatio-temporal predicate Φ, the result of a spatio-temporal filter on input V and query object q can be constructed only from partitions P_V of V, where $\Phi(P_V, q)$ holds.*

Since a tile T in the raster data set can be vectorized as a rectangle T^{vec} a filter on a raster data set R can be realized the same way. Thus, a raster data set can be filtered using any spatial vector query object (polygon, rectangle, etc.) q. The filter identifies all tiles that match q according to the filter predicate Φ and returns them as a new raster data set. This is done by converting the tile into a vector object, namely a rectangle, using its spatial meta information. The filter predicate Φ is then applied on the rectangle representing the tile and q.

$$\begin{aligned} \sigma_{\Phi}^{RV} &= \{T' | T \in R : \Phi(T^{vec}, q)\} \\ T'^{vec} &= MBR(T^{vec} \cap s(q)) \end{aligned} \tag{5.3}$$

A query region may only intersect with a part of a tile. In this case, the result of the filter returns a new tile that represents the intersection and thus, has different extent than the original tile. As shown in Fig. 5.12, the new tile is generated from the MBR of the intersection of the input tile and q. In this new tile, the pixel values which are not part of the intersection are set to a user defined default value representing a missing value. For images, an alternative approach is to set this to a specific color, e. g., black or white.

Pattern Search In the following, we describe the efficient pattern search in raster data sets [82]. The pattern search should return all tiles within a data set containing the pattern. If the pattern spans across multiple tiles, all of them should be part of the result. A pattern is defined as a two-dimensional array of values of the same type as used in the raster data set. Defining the exact pixel values to search for is difficult or even useless in some scenarios. In the use case described in Section 2.1, the user searches for ships in satellite images. The ships are likely to have a color between white and gray in the raster image. Each pixel of the ship probably has a different gray value. Thus, the pattern search should also have a tolerance range to also match pixels that differ from the pattern value up to a defined percentage.

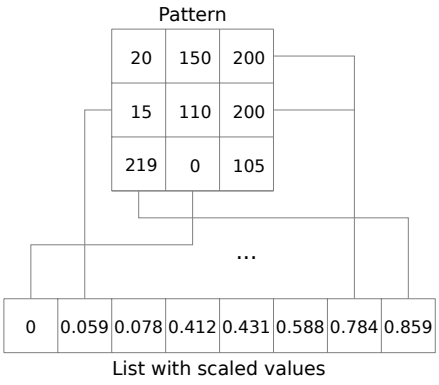


Figure 5.13: Scaled pattern values in an ordered list.

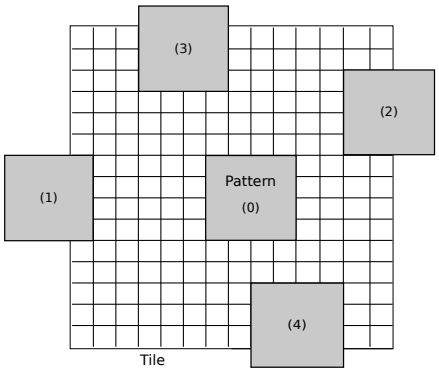


Figure 5.14: Possible positions of a pattern inside a tile.

If the pattern is completely recognized within a tile, this tile can be added to the result. However, if the pattern is large, it may be often the case that only parts of it are found within a tile and the other part is in one or more neighboring tiles. Therefore, the pattern search consists of two main steps:

1. Apply the pattern locally in tiles to produce candidate tiles and

2. merge neighboring candidate tiles to form the final result.

In order to allow a tolerance range for the search, we first scale the pattern values to [0,1]. This can be done by dividing the pattern value by the maximum value of the domain, e. g., 255 for type `Byte`. These values are then sorted in ascending order in a list L. (cf. Fig. 5.13). The list is then replicated to each processing node that will start to search for matches in its local tiles.

Tile-local Search Within a single tile, the values of the pattern need to be compared with all values of the tile. The matching is performed using the scaled values described above. A pixel value in the tile is converted to the scaled value and from this the tolerance interval is created. For a pixel value of 200 the scaled value is 0.784. If the tolerance is 10%, the tolerance interval is calculated as $[0.9 * 0.784, 1.1 * 0.784] = [0.7905, 0.8627]$. We then search for all pattern values in L that lie within this interval. For every pixel value in a tile, this results in a list of possible matches in the result. Now, for each pixel for which a match in the pattern was found, we need to further check if the next values in the row also match their corresponding pattern values.

Comparing every pixel with every value in the pattern would incur a massive overhead which could render the whole pattern search unusable. Thus, the pattern is processed line-wise: every line in the pattern is matched against the tile individually. For every row in the pattern, we compute the position of the last value within the tile. Figure 5.14 shows five scenarios of a pattern overlapping a single tile. Class 1 – 4 can be combined to form additional variants. Based on these classes, we calculate the last possible position of the pattern row inside the tile and then try to match this value with its corresponding pixel value in the tile. Only if the match is positive, the current row is checked further.

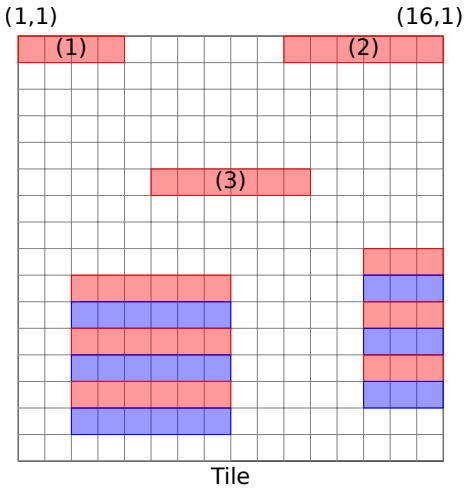


Figure 5.15: Matching rows in a raster tile.

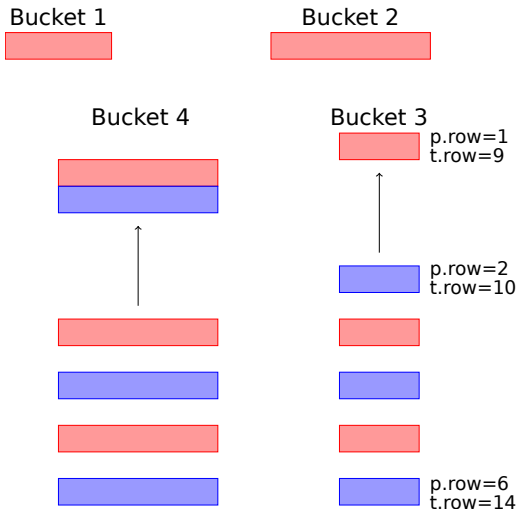


Figure 5.16: Adding matches to buckets.

If this last possible value in a row does not match the pattern, it does not matter if the pixels between the start and the end value match and we can discard the current candidate.

Note, the previous check uses each matched pixel in the pattern as a starting point and is performed for every matching pixel independently. This produces a row describing the match, as shown in Fig. 5.15. Every match is identified by a position key consisting of the position within the tile (row and column number) denoted as t as well as the position p within the pattern at which it starts. To merge the individual identified rows, we collect them into buckets. The bucket number is calculated by the $t.row$, $t.col$, as well as $p.row$ values. For every row where $p.row = 1$, i. e., it was recognized in the first row of the pattern, a new bucket is started. There is a special case to consider here: the actual full match might start in the tile above (i. e., north of) the current tile. The match found in the current tile will have $p.row > 1$. To be able to combine the partial matches later, for matches that touch the upper edge of the current tile we also start a new bucket (number (1) and (2) in Fig. 5.15).

All other matches are then merged into their according bucket. The matches are stored in ascending order (by their row number in the tile) and a bucket only accepts new rows with a $p.row$ and $t.row$ being greater than the corresponding values of the previously inserted row. This means that the bucket 3 starts with a match identified by $p.row = 1$ and $t.row = 9$ and only accepts a match with $p.row = 2$ and $t.row = 10$ (see Fig. 5.16).

Merging Local Results The local results on the nodes need to be merged to form the global result. The results of two nodes are merged by combining the rows of the respective buckets. To reduce the amount of data exchanged between nodes, only the start and end point of the rows in the bucket need to be sent. All other information can be reconstructed on the other node.

Based on the $t.row$, $t.col$, $p.row$, $p.col$ values and the information about where the partial match ends (left, right, bottom, top edge of tile) of the partial matches in one tile, we can calculate the partial matches in another tile. Similar to the

two phases of a hash join, we insert all partial matches of the first tile into a `Map` structure with the key consisting of the five mentioned values. The map is probed with the partial matches of the second tile. We continue combining two tiles this way, until all tiles have been processed and matches have been combined.

Join A spatio-temporal vector or raster join operation combines two vector data sets, two raster data sets, or a raster and a vector data set using some spatial or spatio-temporal predicate Φ to apply as join condition.

There are two strategies to perform a join operation in parallel: dynamic replication and dynamic partitioning. In dynamic replication, the smaller of both input relations is replicated to the nodes maintaining partitions of the other relation. Dynamic partitioning means that both input relations are partitioned using the same partitioning scheme and a node joins the content of two corresponding partitions. In a data parallel shared nothing cluster, the dynamic replication can be implemented using a `broadcast` to make the respective relation available on all worker nodes. This approach should be used for small inputs only, which fit into memory of the worker nodes.

When replicating one complete relation is not possible, the framework has to determine which partitions to join.

Regardless of vector or raster data, the join can benefit from the existing spatial or spatio-temporal partitioning. There are three scenarios:

1. If one or both inputs were not partitioned using a spatial/spatio-temporal partitioner, the join partners are distributed over all partitions of the input data sets R and S. Assume R consists of n and S of m partitions, then $n \times m$ join partitions have to be created and for each the local join result is computed. Alternatively, both inputs can be repartitioned dynamically, preferably using the same partitioner algorithm.

2. The two inputs are both partitioned using a spatial/spatio-temporal partitioner, but both with a different one (or with different parameters). In this case, we can apply the partition pruning and identify intersecting partitions and join them.

3. If both inputs were partitioned using the same partitioner, the created partitions are congruent and partition i of input R has to be joined with partition i from input S.

Corollar 2. *For a spatio-temporal predicate Φ, the result of a join on input R and S, with R_P and S_P being the set of partitions for each input, can be constructed only from the combinations of those partitions R_{P_i} from R_P and S_{P_i} from S_P, where $\Phi(P_{R_i}, P_{S_i})$ holds.*

In the second case, finding the intersecting partitions can be performed with a simple nested loop if the number of partitions is small. For a large number of partitions, the partitions of one relation can be indexed. This index can be queried using the partitions from the other relation and the result is the list of partitions that intersect with the query partition.

In the third case, if the two inputs were partitioned using the same partitioner (and parameters), it is known that partition R_{P_0} has to be combined with S_{P_0}, R_{P_1} with S_{P_1}, and so on. Thus, the lists of partitions of R and S can be *zipped*. There are two important facts to consider here: (1) The number of partitions is not enough to perform this zipping. If R contains objects in Germany and S objects in the USA, then both data sets may be partitioned into p partitions, but $R_{P_0} \cap S_{P_1} = \emptyset$. (2) Furthermore, since objects may exceed the partition boundaries, the identification of partitions to zip must be based on the partitions' extent. Objects that intersect multiple partitions are replicated into all of them. This requires to perform an additional de-duplication step.

For the actual join, the worker nodes have to find the join partners in the content of the two partitions. If an index exists on any of the two inputs, it can of course be used to improve the join.

In the literature, several approaches to compute the spatial join (also in parallel) exist. In [8] Arge et al. present a sweep-line based approach if neither input data set is indexed. Further, [69] presents a spatial merge join and [66] adapts the hash join principle to spatial joins. However, these approaches are in the context of spatial database systems and not designed for shared nothing cluster environments, though the basic idea can also be ported to these platforms. Brinkhoff, Kriegel, and Seeger presented a parallel spatial join technique using R-trees where the tree, which must exist on both inputs, is traversed in parallel [18]. The "Spatial Join with MapReduce" in [113] presented the first spatial join on MapReduce and showed that it can compete with the parallel version of the spatial merge join.

In general, there are three cases:

Case 1 No index exists on either input. In this case a nested loop or sweep-line approach can be used. Alternatively, an index can be created on-the-fly on one or both inputs and then Case 2 or 3 can be applied, respectively.

Case 2 An index exists on one of the inputs. Here, we iterate over the input without index and probe the existing index to find the join partners. Alternatively, the non-indexed input can also be indexed and Case 3 is applied.

Case 3 If both inputs are indexed, the join strategy presented in [18] can be used for R-trees or [77] for Quadtrees.

While the general approach of identifying join partitions is the same for vector and raster data, the resulting schema that they produce differs. In the following, we will discuss how the actual join result is constructed for possible combinations of vector and raster data joins.

Vector Data A join requires the definition of the join attribute in the two input data sets. The Big Data engines typically assume the join attribute to be the *key* in the schema. Hence, we also assume the schema in the input data sets to be a key value pair of `(STO , Payload)`, where `STO` is an instance of the spatio-temporal data type defined in Section 2.3.2 and `Payload` is some associated payload data.

The vector data join returns all tuples from both inputs `R` and `S` that satisfy the join condition Φ as $(\texttt{Payload}_R, \texttt{Payload}_S)$:

$$\bowtie_{\Phi}^{VV} = \{(P_R, P_S) | (STO_R, P_R) \in R, (STO_S, P_S) \in S : \Phi(STO_R, STO_S)\} \quad (5.4)$$

The payload itself may be a nested structure containing the associated `STO` again. This way, it can be extracted from payload objects in the join result so that another spatio-temporal operation (filter, clustering, etc.) can be applied subsequently.

Raster Data A join of a raster data set with a vector data set works the same way as joining two vector data sets. After identifying intersecting partitions, their contents are checked pair-wise to find all tiles that match with vector objects in the vector data set. Since each tile stores the coordinates of its upper left point as well as the width and the height, the transformation into a vector rectangle, that is tested against the query polygon, incurs no significant overhead.

The resulting schema of the join of a raster data set R and a vector data set V and predicate Φ is a pair of a tile and the tuple of V that matched with that tile: `(Tile, (STO,Payload))`, where `Payload` is the payload from the vector data set which is always of schema `(STO, Payload)`:

$$\begin{gathered} \bowtie_{\Phi}^{RV} = \{(T', (STO, P)) | T \in R, (STO, P) \in V : \Phi(T^{vec}, STO)\} \\ T'^{vec} = MBR(T^{vec} \cap STO) \end{gathered} \tag{5.5}$$

Similar to the raster-vector filter, T'^{vec} is the minimum bounding rectangle of the intersection $\cap$ of T^{vec} and the vector object STO from V. T' then contains all pixels of T that are contained in T'^{vec}.

Joining two raster data sets R_1 and R_2 will find (parts of) tiles that match a given predicate Φ. The user additionally specifies a `combine` function f that will be applied on the pixel values of the parts of the two matching tiles to compute the pixel value in the resulting tile.

$$\bowtie_{\Phi,f}^{RR} = \{T'_1 \oplus_f T'_2 | T_1 \in R_1, T_2 \in R_2 : \Phi(T_1^{vec}, T_2^{vec})\} \tag{5.6}$$

As in the previous cases, T'_1 and T'_2 contain those pixels in T_1 and T_2, respectively, that lie in the intersection of these two tiles. The resulting tiles are generated by combining two matching tiles pixel-wise and compute the pixel value in the result tile using f. Thus, f is a function of $f : U \times V \to W$, where U and V are the data types used in R_1 and R_2, respectively, and W is the generated result value. U, V, and W can all be the same type, of course.

Skyline The Skyline, or Pareto-optimum, is a multidimensional optimization problem, with the goal to find those objects (or solutions) from a data set (or problem/-solution space) so that each object of the result is the best option in at least one dimension. In [17], Börzsöny, Kossmann, and Stocker proposed this operation as an operator for relational database systems. The Skyline of a data set R for a given query point q is defined as:

$$S_q = \{p_i | p_i \in R \wedge \neg \exists p_j \in R : p_j \neq p_i \wedge p_j \succ_q p_i\} \tag{5.7}$$

Where $\succ_q$ denotes a dominance relationship with $p_j \succ_q p_i$ meaning p_j is *better* than p_i according to the query point q. Here, *better* means a minimum or maximum criteria, depending on the application.

In contrast to [91] that adapted the idea to spatial-only data, the dimensions that we consider here are the spatial and temporal distance to the reference object.

Following this, the dominance relationship must be defined on the distance of the object in R to q in the spatial and temporal dimension, as shown in Eq. (5.8).

$$p_j \succ_q p_i \Leftrightarrow dist_{st}(p_j, q) \leq dist_{st}(p_i, q) \tag{5.8}$$

Since the distance values depend on the query point q, they cannot be pre-computed in general, but are determined ad hoc during execution. Thus, the computation of the Skyline for an input data set R includes two major steps:

1. Calculate the distance $dist_{st}(p, q)$ of an object p to q in spatial and temporal dimension and

2. decide if p is dominated by any other point in the result Skyline.

The Skyline is an interesting operator for finding similar objects in a data set to a given reference object and several implementations have been proposed.

A naïve nested loop approach compares every object in R with all other objects, resulting in a complexity of $O(|R|^2)$. Furthermore, a full nested loop implementation in a data parallel system requires to shuffle all partitions to all nodes. This means, massive amounts of data being replicated and transferred over network.

Adaptions to the characteristics of a cluster environment usually follow the same general approach: compute partition-local Skylines individually in parallel and merge them later to form the global Skyline.

A Skyline operator can greatly benefit from a spatial partitioning: when using a partitioner on the set of distance values, the partitions have well-known bounds (in terms of distance values / coordinates). Thus, there might be partitions which cannot contain Skyline objects simply because they are dominated by another (non-empty) partition.

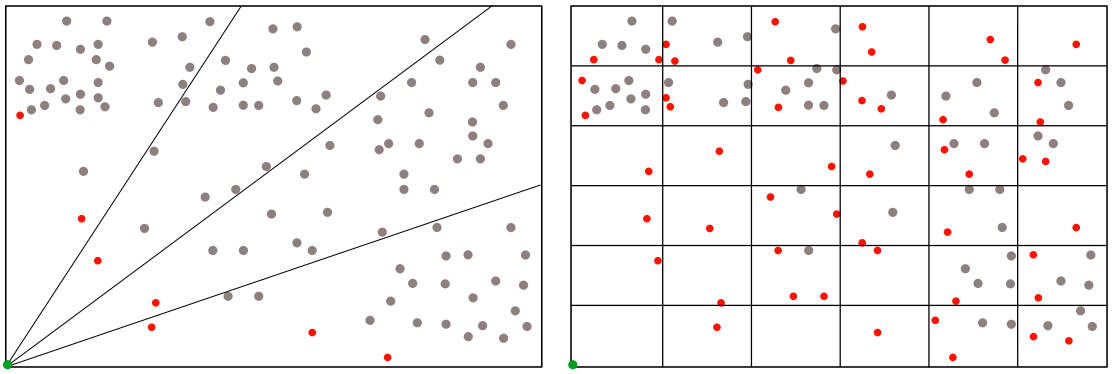
Mullesgaard et al. presented an approach to compute Skyline in MapReduce in [71]. In this approach they exploit the grid partitioning information and partitions dominated by others are excluded from further processing. They use bitstrings with positions set to `1` if the corresponding partition is not empty. For every non-empty partition, the positions of the dominated partitions in the bitstring are set to `0`. After this step, only partitions whose index in the bitstring is still `1` are considered in the further Skyline computation.

In [22], an angular partitioning that creates partitions (segments) of fixed angles around the given query point is proposed. The advantage is that with this partitioning strategy, no additional partition pruning is necessary and the local Skylines already dominate many points so that merging them requires fewer redundant dominance checks.

Figure 5.17(a) sketches this angular partitioning and highlights the resulting Skyline points. For comparison, in Fig. 5.17(b) partitioning is shown.

We now discuss for possible implementations for the Skyline computation, that (partially) adapt the just described approaches.

1. `GridPart`: First, for every element in the input its spatial and temporal distance to `q` is computed, resulting in an intermediate data set of distance values. The *distance data set* is partitioned using a grid based partitioner. Then, only



(a) Angular partitioning and local Skylines. (b) Local Skylines on Grid-based partitioning

Figure 5.17: Local Skyline computation with Angular vs. Grid-based partitioning. Red points mark partition-local Skyline points that need to be merged to the final result.

for those partitions that are not dominated by another one the local Skylines are determined. The local Skylines are finally merged into a global Skyline result [71]. The difference to the previous approach is that the input data set is transformed into distance data set which is explicitly partitioned using a grid. Alternatively to a bitstring, the dominance can be computed from the partition number in the grid. Chen et al. achieved much higher dominance ratio inside the partitions compared to grid partitioning, so that fewer points from the local Skylines need to be merged.

2. `Angular`: We first compute the *distance data set* as in `GridPart`, but use the angular partitioning introduced in [22]. For each angular partition (segment) the local Skylines are computed which are finally merged into the final global Skyline.

3. `AngularNoPart`: Repartitioning the whole distance data set is very expensive. Thus, in the third approach we try to avoid shuffling the complete input data. The code is given in Algorithm 1 as pseudocode. It starts with a non-repartitioned data set `R` and for every partition a mapping from an ID to a `Skyline` object is created. Then, for every element `obj` in partition `P`, the angular partitioner of [22] is used to determine the partition ID the object would belong to (but data is not repartitioned), based on its angle to the reference object `q` and finds the Skyline objects in the map based on this ID. Thus, for each physical partition created by the underlying platform, we generate multiple local Skylines, based on a logical angular partitioning. Since we already maintain local Skylines, elements that are dominated by some objects are pruned. This way, total data size to shuffle in order to build the global Skyline is reduced. In the merging step, Skylines from different physical partitions with the same angular partition ID are merged. The `Skyline` structure maintains a list of Skyline points `L`. A newly offered point `o` is added to `L`, only if it is not dominated by any point already in `L`. When `o` was added to `L`, all points previously in `L` that are dominated by `o` are removed.

Algorithm 1 AngularNoPart method for partition R and reference point q

```
1: procedure SkylineAngularNoPart(R, q)
2:     skylines ← map<int → Skyline>
3:     for each P ∈ partitions(R) do          ▷ partitions are processed in parallel
4:         localSkyline(P, q)
5:     end for
       return merge(skylines)                          ▷ merge all local Skylines
6: end procedure
7: procedure localSkyline(P,q)
8:     partitioner ← new AngularPartitioner
9:     for each obj ∈ P do
10:        stDist ← dist(obj, q)            ▷ pair of spatial and temporal distance
11:        partId ← partitioner.getPartition(stDist)
12:        skylines[partiId].offer(stDist)
13:    end for
14: end procedure
```

4. `Aggregate`: This approach also makes use of the `Skyline` structure and is supposed to benefit from the plain parallel processing without any additional partitioning information. For every partition, the local Skyline is computed. These local Skylines are then merged into a global Skyline as final result. This approach avoids an explicit repartitioning step to save execution time, but computes the Skyline over all objects in the input data set. Note, the partitions that this algorithm works on are not spatial partitions, but e. g., the normal partitions generated by Spark when loading the data. We dubbed this approach `aggregate`, because the Skyline is computed as an aggregate over the input data set.

While in MapReduce intermediate results are written to disk anyway, repartitioning the data might not be a significant overhead. For in-memory architectures like e. g., Spark and Flink however, the shuffling caused by partitioning incurs a significant overhead compared to the plain execution time of queries. Thus, we expect that approaches that forgo an explicit repartitioning to achieve very good results in the evaluation in Section 9.2.2.

Nearest Neighbors Search For the nearest neighbors search operator a similar strategy as for Skyline computation can be applied: compute the partition-local solutions and merge them into a global final result.

Additionally, the kNN search can be supported by existing (or to be created) indexes, such as R-trees. Here we discuss three different ways to compute the nearest neighbors for a given reference point.

Naïve In a straightforward implementation, if no index exists, we first compute the k nearest neighbors for each partition individually, by calculating the distance of each data object to the reference point using a provided distance function.

Algorithm 2 Partition-local kNN computation using kNN struct.

```
1: procedure kNearestNeighbors(partition, k, q)
2:     knn ← [], max ← -1, m ← -1
3:     for each p ∈ partition do
4:         pos ← m + 1
5:         if pos < k then
6:             knn[pos] = p
7:             update_max(pos)
8:             m++
9:         else if dist(knn[max],q) < dist(p,q) then
10:            knn[max] = p
11:            update_max()
12:        end if
13:    end for
14:    return knn
15: end procedure
```

Then, the data objects are sorted in ascending order by their respective distance values and only the first k items are returned. If an index exists, or is supposed to be created during execution, this index is queried for the k nearest neighbors in that particular partition. In both cases, this results in n lists of k elements, if the data set consists of n partitions. We then apply a global sort of these n × k elements and return only the first k items as the final result.

LocalKnn Sorting all elements in a partition may be time consuming and also requires to collect them in memory. Thus, we additionally implement a `KNN` data structure that maintains k elements in an array as well as the positions of the element with the maximum distance values in this array. As for Skylines, the structure is used to aggregate the elements in the input data set, as shown in Algorithm 2. For each partition a local `KNN` instance is generated and every element in the partition is inserted. If fewer than k elements are in the `KNN` array, the element is added and the positions for min and max are updated if necessary. If the structure already has k elements, the new element is inserted only if it has a smaller distance value than the current maximum. In that case, the maximum is replaced with the new element and the max pointer is updated. This strategy avoids an ordering of the elements and also removes objects as soon as possible to reduce load on the workers. The local kNN results are finally merged to find the global result.

BoundedKNN In the previous strategy, still for all elements in the data set their distances to the reference point have to be computed and tested against the current result objects. To reduce the number of elements being processed, the kNN search can be bounded by the maximum distance of the kNNs in the partition containing `q` [99]. It is given in Algorithm 3. This approach first finds the partition containing `q` and finds the k nearest neighbors in it. Then, an area around the query point `q` based on the distance of the k-th element to

Algorithm 3 Bounded kNN Search

```
1: procedure BOUNDEDKNNSEARCH(rdd, k, q)
2:     P ← getPartitionContaining(q)
3:     kNN_P ← kNearestNeighbors(P, k, q)
4:     if kNN_P== ∅ OR (dist(q, kNN_P[k-1])== 0 AND kNN_P.length<k) then
5:         return LocalKnn(rdd, k, q)      ▷ P empty or less than k duplicates of q
6:     end if
7:     r ← buildRangeAround(q, kNN_P[k-1])
8:     I ← rdd.rangeQuery(r)
9:     kNN_I ← kNearestNeighbors(I, k, q)
10:    if kNN_I.length == k then
11:        return kNN_I
12:    else if kNN_I.length > k then
13:        return kNN_I.takeOrdered(k)
14:    else if kNN_I.length < k then
15:        return LocalKnn(rdd, k, q)      ▷ too few elements in r, restart global
   search
16:    end if
17: end procedure
```

`q` is built and used to query to complete data set. This range query can make use of the partition pruning mechanism outlined in Section 5.2.2. If the data set contains duplicates, it may be that the maximum distance found in the partition-local kNN search in line 3 is 0. In this case, the constructed query range would be the point `q` itself instead of a real range. Thus, appropriate steps must be taken, e. g., restart with one of the strategies explained above.

Clustering Clustering is an often applied operation to find groups of objects in the input according to some similarity measure. For the spatio-temporal data considered in this work, the similarity between two objects is their distance in the spatial and temporal dimension.

Over the years, many clustering algorithms have been proposed in the literature, such as k-Means [65], DBSCAN [38], OPTICS [7], and CURE [41]. Especially the DBSCAN algorithms is often preferred over, e. g., k-Means as it makes no assumptions about the number of clusters (k) or the shape of the clusters. With the advance of the Big Data platforms, people were required to find clusters in their large data sets, too, and hence, many of the algorithms originally defined for single computers were modified to exploit the parallel execution model of Hadoop. For DBSCAN several adaptions for MapReduce, such as [27, 73, 55, 56], have been proposed. Only [16] implements a spatio-temporal clustering, also based on DBSCAN.

The clustering operator is inspired by [55] and exploits the data parallelism and leverages a spatial partitioning: the input data is partitioned so that preferably all partitions contain the same number of elements using e. g., the BSP partitioner. The partitions are extended in each dimension by the value of the ϵ parameter of DBSCAN to overlap with their neighboring partitions. Thus, after this step, some

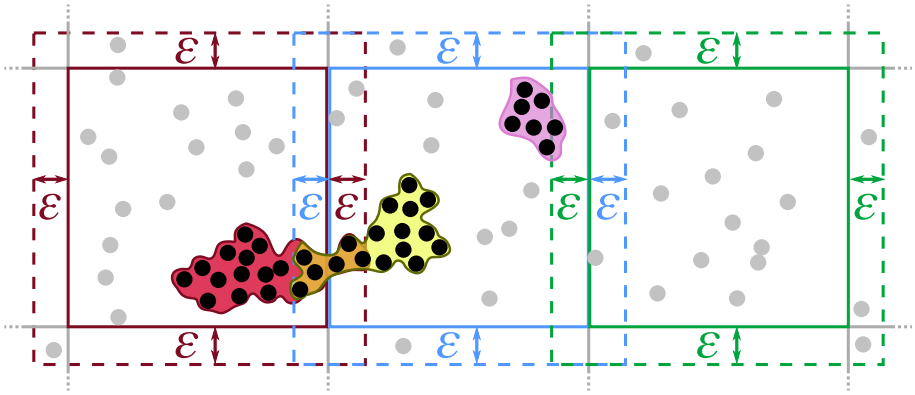


Figure 5.18: Merge clusters via points in ϵ-overlap.

objects may be assigned to multiple partitions. We then compute a local DBSCAN in each partition in parallel using the traditional DBSCAN algorithm.

The results are partition-local clusters which are merged in an additional step where data objects from the overlap regions are used to merge clusters, as shown in Fig. 5.18. This is achieved by creating a graph of cluster pairs where in the graph nodes represent local clusters and edges denote inter-partition relationships between clusters that can be merged. The algorithm is sketched in Algorithm 4

5.3.3 Distance Functions

Some of the above described operators require distance functions to compute their result, such as the k nearest neighbor search, Skyline, and the clustering. Instead of hard coding a single function, e. g., the Euclidean distance measure, the desired distance function should be passed as a parameter by the user. In that way, applications that use geometric data, but also scenarios where geodetic calculations are necessary can be supported by the framework. While the implementation of the actual operator logic is the same, the distance function to correctly calculate the distance between two objects on the earth's surface differs from the calculation in a two-dimensional planar space.

Furthermore, this concept allows to implement different strategies for distance calculation for polygons and intervals, where the distance between such objects could be the smallest distance, the distance of their respective center points, or their the minimum *and* maximum distance. Thus, besides a scalar distance that contains a single distance measure value, an interval distance which can be used to express distance measures with a minimum and maximum value is required. This also allows to process e. g., imprecise events that we proposed in [49] and will further discuss for future work in Section 10.2.

Since the distance functions are applied on spatio-temporal objects, they either return one value as result or a pair of `(spatial_dist, temporal_dist)`. On such a distance pair, users can apply weights to merge them into a single value and express their preference of one dimension over the other.

Algorithm 4 STDBSCAN

```
1: procedure DBSCAN(rdd, eps, minpts)
2:     partitions ← computePartitions(rdd)
3:     extend(partitions, eps)                    ▷ make partitions overlap by eps
4:     partedRDD ← assignToPartitions(rdd, partitions)
5:     localClusters ← ∅
6:     for each p ∈ partedRDD.partitions do
7:         clusters ← localDBSCAN(p, eps, minpts)  ▷ in parallel in each partition
8:         localClusters ++= clusters
9:     end for
10:    globalResult ← mergeClusters(localClusters)
11:    return globalResult
12: end procedure
```

5.3.4 Language Integration

The discussed operations are based on a collection of spatio-temporal vector and raster objects and we outlined how they can be implemented in a data parallel cluster setup. As mentioned before, the framework will be implemented for Apache Spark using the RDDs. Using RDDs however, has the disadvantage of having to create Scala, Java, or Python programs that create and modify them using the respective API in a program.

Programming, compiling, and deploying to the cluster is an overhead that might be unacceptable for data scientists and other researchers that quickly need their results. Thus, integration into declarative languages is needed where users simply enter their query and the interpreting system is responsible for compilation, deployment, and execution.

To achieve this, the framework with its domain specific language (DSL) can be integrated into SparkSQL by offering filter, join, as well as other operators as functions that can be used in SQL queries. Since programs often have to perform more tasks than just querying, like cleaning, reformatting, sanitation, etc., a more powerful declarative language is useful and important. Thus, the STARK operations are also integrated into the Pig Latin language. We will discuss the integration into SparkSQL in Section 7.1 and the Pig Latin integration in Section 7.2.

5.4 Conclusion

In this section we discussed several aspects for designing and implementing a spatio-temporal data processing engine on Apache Spark. For the discussed kNN and Skyline operators, we proposed different implementation variants that can make use of existing (or to be created) indexes as well as partitioning strategies.

As such partitioning strategies, we considered three variants. The FixedGrid as well as the cost-based BSP and R-tree partitioners. In our evaluation we will show how these partitioning strategies behave under different parameters. To further improve query execution, indexes can be used. Since they are not available by

default in the Big Data processing engines, we introduced two basic variants for creating them: the online creation as well as the persistent indexes. Especially the approach of the persistent indexes is supposed to improve query operators as it avoids the potentially costly index creation at query execution time.

Besides the fundamental partitioning and indexing strategies, we further discussed the possibility of a logical near data processing. With this, query ranges or reference points should be used to identify result candidate partitions.

In our evaluation in Chapter 9 we will analyze the impact and necessity of these strategies on the overall query performance. Especially the early identification of result partitions is promising as one can expect a drastic reduction of data to load and process at the worker nodes.

Chapter 6
Prototype Implementation for Apache Spark

In the previous Chapter 5 we discussed several strategies to work with spatio-temporal vector and raster data on data parallel platforms. To show their validity and examine potential differences, we chose to implement the operators as well as the partitioning and indexing strategies in a framework for Apache Spark. The result is a Scala-based framework, called STARK (Spatio-Temporal Data Analytics on Spark), available as Open Source on GitHub[1].

In this chapter we briefly describe the internal implementation of its core. The general architecture of STARK is shown in Fig. 6.1.

6.1 STARK Architecture in Detail

The STARK framework is designed to integrate with the Spark Core API based on RDDs. Thus, its classes extend the basic RDD structure of Spark and implement the operators. Additional wrappers integrate STARK into the SparkSQL engine.

6.2 Adding Types for Vector & Raster Data

The discussed functions and strategies require data types that can be used to model the vector and raster data sets. As discussed in Section 5.2.1, a vector object needs to have a spatial as well as an temporal component and since not all spatial objects also have a temporal value, this time component is optional.

Thus, to represent vector data, we introduce the basic datatype called `STObject`. The class provides only two fields: (1) `geo` for storing the spatial component and (2) `time` to hold the temporal component of an object.

```
STObject(geo: Geometry, time: Option[TemporalExpression])
```

In Scala, the `time` field is an `Option`, meaning that it either has a value or it is `None` (instead of `null`).

Like many other Java based open source projects that deal with spatial data, STARK uses the JTS library. This library provides basic data types for geometries, such as points, line strings, and polygons as well as operations thereon. Additionally, the library includes an R-tree implementation that will be used for indexing.

The dependencies to the JTS library are kept to a minimum so that the framework can be ported to any library if desired. The Google S2[2] is very promising as it supports geodetic operations, while JTS performs geometric operations only.

[1] `https://github.com/dbis-ilm/stark/`
[2] `https://s2geometry.io/`

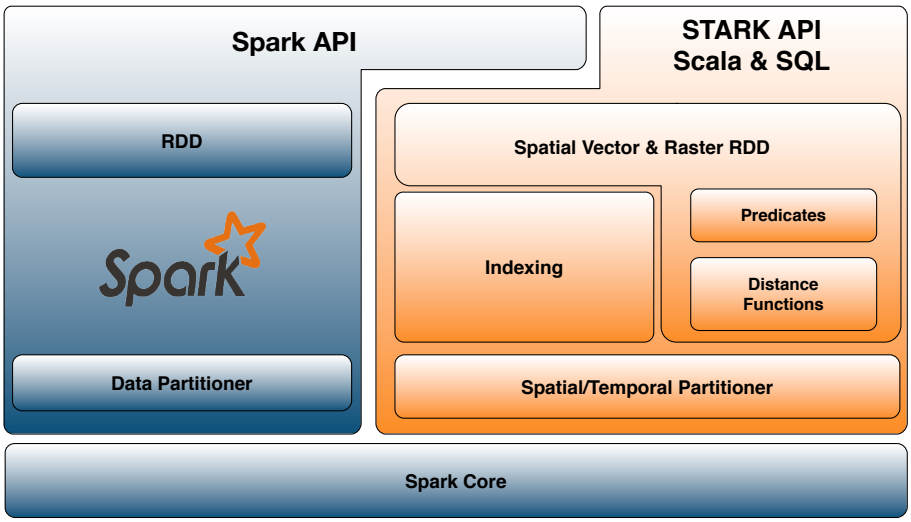


Figure 6.1: General architecture of STARK and its integration into Apache Spark

However, S2 is written in C++ and its Java port is not available, yet. The `STObject` type and its methods are agnostic to the used coordinate system so that they can easily be adapted to other libraries.

The `Geometry` type used in `STObject` is the base class of all geometric objects in JTS. In STARK we additionally implement a `Rectangle` class that extends the existing `Polygon` class to improve performance when dealing with such data. For the temporal components, STARK brings its own types: the `Instant` class is used for points in time, while the `Interval` class represents right open intervals. Additionally, the end of an interval is optional to also model infinite intervals or intervals whose end is unknown. `Instant` and `Interval` are both derived from a `TemporalExpression` trait and implement functions such as `intersect`, `contains`, `before`, `after`, etc.

To represent the tiles in a raster data set, a `Tile` class is used. The pixel data of a tile is stored in an generic array so that tiles can be used for any pixel value type `U`.

```
Tile(ulx: Double, uly: Double, width: Int, height: Int,
     pixels: Array[U], w: Double)
```

As identified in the previous design chapter, a tile must encode its global position in the data space. Thus, the `Tile` class stores the coordinates of its upper left (ul) corner as meta information along with the width and height in number of pixels. To recreate the rectangle in the vector space covered by the tile, the resolution must be also be known. This is encoded in the width `w` of a single pixel in the vector space.

It is enough to represent a rectangle with only two points: one point with the lowest values in all dimensions and the other one with the highest values in all dimensions. In the two-dimensional space this is the lower left (ll) and upper right (ur) point.

The rectangle T^{vec} of a tile T is then created as a rectangle with these two points:

$$\begin{aligned} T^{vec}_{ll} &= (T_{ul_x}, T_{ul_y} - w * height) \\ T^{vec}_{ur} &= (T_{ul_x} + w * width, T_{ul_y}) \end{aligned} \tag{6.1}$$

6.3 Basic Operations

All data processing pipelines start with loading data from some source. STARK relies on Spark's native methods to load data from sources such as HDFS. Loading a text file in Spark results in an RDD of strings where each string represents one line in the input file. By applying transformations via `map`, users can extract the spatial and temporal or raster values from the string and generate an `STObject` or `Tile` instance. `STObject`s can be created by providing the coordinate values for points or by passing a WKT string of the geometry. This allows to mix several geometric types in a single data set. In both cases the temporal component can optionally be given as a `Long` value.

Most operations are implemented in the `SpatialRDDFunction` class that wraps a traditional RDD. This follows the concept that Spark implements for special operators on Pair-RDDs, like a join. A join can only be executed if the input RDDs are Pair-RDDs, i.e. they contain 2-tuples `(k,v)`, where `k` is used as join attribute. Spark automatically creates a `PairRDDFunction` object with the input RDD as parameter, using Scala's *implicit conversion*[3]. The `PairRDDFunction` class then implements the join method.

STARK also provides such implicit conversions that create `SpatialRDDFunction` objects for Pair-RDDs, where the key `k` is of type `STObject`[4]. The value `v` of that pair can be of any type and is maintained during all operations. The class `SpatialRDDFunction` implements all spatial operations supported by STARK: filtering, join, kNN, clustering, as well as index creation.

The implicit conversion is transparent to the users and creates a seamless integration into any Spark program. Users do not have to explicitly create an instance of any of STARK's classes (except `STObject`) to use the spatial operators.

Consider an example where we have a dataset given as a CSV file that contains a list of events from various categories. The schema of that file might be:

```
(id: Int, description: String, category: String, wkt: String, time: Long)
```

After loading, pre-processing, and transforming, we get an RDD of exactly that type: `RDD[(Int, String, String, String, Long)]`. We then create an instance of `STObject` representing the location and time of occurrence from the WKT string and time field, respectively, of each entry:

```
val events = rdd.map { case (id, desc, category, wkt, time) =>
                    ( STObject(wkt, time), (id, desc, category) ) }
```

The resulting RDD is of type `RDD[(STObject, (Int, String, String))]`. We can now simply use this RDD to call the functions to perform a spatio-temporal range query.

```
val qryTime = 1481287522
val qry = STObject("POLYGON((....))", qryTime)  // create query object
val contain = events.containedBy(qry)  // contained by the query region
val intersect = events.intersect(qry)
```

[3] `https://docs.scala-lang.org/tour/implicit-conversions.html`

[4] In the following we will refer to such an `RDD[(STObject,V)]` as `SpatialRDD` and exclude the implicit conversion.

The integration of raster operations follows the same approach: for an `RDD[Tile]`, an implicit conversion exists that creates a `RasterRDD` on which the raster operations are defined.

Filter Operation Filters on `SpatialRDD` and `RasterRDD` are implemented in their own RDD classes.

The `FilterRDD` has the input RDD as well as the query object, as instance of `STObject`, as input parameters. Since `RasterRDD` can be filtered with vector objects, too, the filter implementation for `RasterRDD` is analog. Additional parameters are the filter predicate, expected as a value from predefined constants or as a UDF as well as an index configuration. The index configuration is optional and needed only if the input RDD is not already indexed and is used to create an index on the current partition online during execution.

As described above in Section 3.2.2, the Spark framework requests the partitions from an RDD using its `getPartitions` method. In the `FilterRDD` the `getPartitions` method checks if a spatial or temporal partitioner was applied to create the input RDD. If so, the bounds and extents of all partitions are fetched from the partitioner. These values are then tested against the query object using an intersects predicate. Only if a partition' extent intersects with the query object, this particular partition is added to the result of the method. Non-intersecting partitions are pruned so that their contents are not further evaluated. Note that this partition pruning is only applied if the filter predicate was given as a value from the predefined constants and not as a UDF.

The `compute` method is executed in parallel on the worker nodes to process the partitions returned by `getPartitions`. If the index configuration is set to use no index, STARK iterates over the content of the current partition and tests every element against the query object. With online indexing, the index is built according to the provided settings (index type, order of the tree, etc.) and then it is queried with the query object.

Joins A spatial or spatio-temporal join on `SpatialRDD`s and `RasterRDD`s is implemented in the `JoinRDD`. The `getPartitions` method of this `JoinRDD` class contains the partition pruning step.

This step is implemented by adding the partitions of one input to an in-memory R-tree and query that index using the partition bounds of the other input. The result of this method is a set of join partitions. For every partition of an RDD Spark creates a task that has to be processed by the platform. If partition l_1 is found to intersect with $r_1, r_2, \ldots, r_k$, then k join partitions and therefore also tasks will be created. All these tasks read the same data for their left input (l_1). We implemented this kind of join partition as `OneToOnePartition`, that references exactly one partition from the left input RDD and one from the right input.

To prevent reading the same data in every task, another option is to reference all partitions from the right input that intersect with the currently considered partition from the left input. With this `OneToManyPartition` we can reduce the repeated reads of the left partition and reduce the number of total tasks in the system.

As mentioned earlier, when both data set are partitioned using the same partitioner, the join can be implemented by zipping the corresponding partitions from both data sets. However, since objects may exceed the spatial and/or temporal bounds of a partition, they might intersect with multiple other partitions. Thus, in order to implement this join variant, STARK computes with which partitions an object intersects, and assigns it to all those partitions. However, since the object has been replicated into multiple partitions, the join result contains duplicates that need to be removed. Here, we assume that in the payload of a tuple a unique identifier is present which is used to identify and remove duplicates.

Since joining two data sets (or the partitions' contents) requires lots of pairwise comparisons, a plain nested loop or sweep line approach is only applicable for small inputs. Implementations using indexes yield much better results. Indexes, as described in Section 5.2.3, can be created online and be discarded after use, or be created permanently.

The actual join implementation for raster data with a vector data or with another raster data set is the same as joining two `SpatialRDD`s. The only difference is that the raster tiles are vectorized into rectangles which are used for the actual check in the join predicate. Furthermore, the resulting tiles are created from the intersection of the tiles or the tile and the vector polygon, respectively.

6.4 Operations for Data Analytics

Skyline In Section 5.3.2 four possible Skyline computation strategies were introduced. STARK implements all four of these in order to be able to compare them and decide which one achieves best results in terms of execution time.

The `Aggregate` strategy can directly be realized using Spark's `aggregate(zero, seq, combine)` method. This method aggregates the content of an RDD into a single value. It starts with the `zero` value for each partition and sequentially adds all elements from the partition using the provided `seq` function. The per-partition aggregated values are then combined to a global aggregated value. In STARK, the `zero` value is the already described `Skyline` class with an initially empty list and values are added to this structure in the `seq` function. The Skylines generated for each partition are then merged in the `combine` function by iterating over all elements from one Skyline and inserting them into the other one – this will also check the dominance relationship and discard dominated points.

In the `GridPart` strategy the fixed grid partitioner is used and dominated partitions are pruned before their local Skylines are computed. STARK also implements the angular partitioner for the strategies `Angular` and `AngularNoPart`. Their implementation follows the description given in Section 5.3.2.

Since the Skylines are collected in RAM on the worker nodes, they must be small enough to fit into the available memory. To avoid that worker nodes fail with too large Skylines, which may happen for large objects, the many Skylines per partition generated in `AngularNoPart` are stored in an external map that automatically spills entries to disk if the workers run out of memory.

k Nearest Neighbors As for Skylines, the implementation of the kNN search follows the description in Section 5.3.2. Like the `Skyline` structure, for kNN we built a `KNN` structure that tracks the k nearest objects and prunes objects when newly added elements are nearer than existing ones. The algorithms make use of an existing index, if available, to search for the partition-local results the a branch-and-bound tree traversal provided by the JTS library is used.

Clustering The clustering operator uses the cost based BSP to divide the work among the executors. Local clusters are merged in an additional step. The partitions are extended in each dimension to overlap with their neighboring partitions so that clusters that span across multiple partitions can be merged. Objects within this overlap region are replicated to multiple partitions and thus, these replicas need to be recognized during execution. Therefore, the `cluster` method in STARK expects a function to extract key candidate from a tuple which is used to identify replicated tuples.

6.5 Spatial and Temporal Partitioning

The spatial and temporal partitioners implemented in STARK are supposed to integrate with Spark as much as possible. Thus, to group the partitioners we defined the `SpatialPartitioner` as well as `TemporalPartitioner` traits. These base traits are implemented by the concrete partitioning strategy implementations. The grid based partitioners (fixed grid, BSP) share another abstract class `GridPartitioner` that provides methods to determine the minimum and maximum values in each dimension as well as to compute the bounds of the grid cells and the histogram over the input RDD.

The computation of these statistics is performed over the input RDD before the partitioning happens. The actual partitioning is performed in parallel during a shuffle where copies of the partitioner are used on the worker nodes to decide to which partition an object belongs.

6.6 Indexes

STARK uses the index structures available in the JTS library, namely STR-tree and QuadTree. However, only the R-tree can be used for kNN computations. All functions in STARK are designed to be independent from the actual underlying index – if possible. Thus, wrapper classes around the JTS implementations exist, that implement, e. g., the `Index`, or `KNNIndex` traits. This way, the kNN functions that are implemented based on a index can require an index structure that supports this type of query, while the filter or join expect only an index that implements the general `Index` trait.

The original design of Spark is that operators receive an iterator over a partition's content so that only one element of that partition has to be kept in memory. Building an index for a partition, however, requires to collect its complete content in memory. Querying the index in the filter operator traverses the tree and collects the results in

MBR	time	file
0,0 – 10,10	Interval(1234,5678)	part-00000
10,0 – 20,10	Interval(1234,5678)	part-00001
...	...	
170,80 – 180,90	Interval(9932,9999)	part-00999

Figure 6.2: Global index created in STARK with partition information.

another list. As the construction of this list consumes time and allocates additional memory, the R-tree implementation of JTS was extended in STARK to return an iterator for the query result. This iterator is not an iterator over the result list, but is populated with the next result element only if its `next` method is called, i. e., the result is actually consumed. After retrieving a result, the traversal of the tree is paused until the next element is requested. This implementation lines up with Spark's lazy evaluation and saves time and memory.

6.7 Spatio-temporal Spark Context

Every Spark program uses the `SparkContext` (or `SparkSession` for SparkSQL) to initially load data. The context is created in the driver and servers as an entry point for the distributed processing, but also provides access to information about the execution environment. Like the rest of Spark, the `SparkContext`, used to load data into RDDs, is unaware of any spatial or temporal features. STARK implements its own `STSparkContext` that extends `SparkContext`. In STARK, if `SpatialRDD`s or `RasterRDD`s are written to disk that were partitioned using one of STARK's partitioners, the partitioning information is also stored to disk in the global index file. The file contains the extent information for each partition along with the file name were it was written to (cf. Fig. 6.2).

Using this global index, the `STSparkContext` implements the logical near data processing and is able to prune the partitions to load, when provided a query object. The identified partitions are loaded using Spark's native functions. This context class also implements methods to execute kNN search and load matching partitions for joins.

While in the low level Scala API developers have to do this by hand, query planners for higher level declarative languages, like SQL or Pig Latin can make use this feature without the user's interaction.

Chapter 7
Declarative Language Support

The Scala API of STARK can be used by skilled programmers to implement the processing pipelines and manually optimize operations. However, the rather low level implementation of programs requires programming knowledge that may be not present for many data scientists. In the field of data analytics declarative languages are more wide spread, because they let people focus on their problem instead of implementation and optimization details. Thus, to support *data workers*, STARK integrates into SparkSQL and our own Piglet system. In this chapter, we will briefly describe this integration.

7.1 SparkSQL Integration

The SparkSQL architecture allows external libraries to register user-defined types (UDTs) and UDFs which can then be used for representing own data and calling specialized functions.

7.1.1 Data Type Registration

The UDT is used as data type for a column in the schema, whereas a UDF can be used in various places, e. g., in the `SELECT` or `WHERE` clause.

STARK uses this mechanism to register UDTs for the `STObject` and `Tile` classes: `STObjectUDT` and `TileUDT`, respectively. This way, like in any relational DBMS, a table schema can contain one or more columns that hold the spatial or spatio-temporal feature of a row. For raster data sets, each row in the table has a (single) column that holds the tile. In Spark, UDTs can be registered conveniently using the `UDTRegistration` class. All STARK UDTs register themselves upon initialization.

The UDTs make spatio-temporal vector and raster data first class citizens for SparkSQL. Often, in Spark data is loaded into RDDs or `Dataset`s which are then registered as tables or views to that they are known to SparkSQL. To create instances of these types, constructor functions are needed. The `STObjectUDT` can be created from WKT strings using the `st_geomfromtext` function, known from the Open Geospatial Consortium (OGC) standard. A similar `tile` function can be used to create a `TileUDT` instance by providing the location of the upper left corner point, width, height, pixel width as well as the array of value (e. g., `double`s).

7.1.2 User Defined Functions

The spatial vector and raster functions are also added as UDFs. STARK sticks to the names used in the OGC standard. Hence, vector functions, as described in Section 5.3.2, carry the `st_` prefix (`st_intersects`, `st_contains`, etc.).

The UDFs are registered using Spark's internal methods to register temporary functions via the Spark `sessionState` when the `STARKSession` is created. The `STARKSession` encapsulates a `SparkSession` which is used for SparkSQL features.

An example query joining two vector data sets by an intersects predicate can be formulated as:

```
SELECT l.name, r.name
FROM left l, right r
WHERE st_intersects(st_geomfromtext(l.wkt), r.geo)
```

Assuming a schema of `left` with attributes `name` and `wkt` whereas relation `right` has a `geo` attribute containing already a `STObjectUDT`.

The raster data processing functions can be called in a similar way:

```
SELECT countValues(tile, 1)
FROM myraster
WHERE r_contains( st_geomfromwkt("POLYGON(...)"), tile)
```

As discussed in the previous chapter, STARK allows to combine raster tiles with vector data objects and in the query above, the `r_contains` functions finds all tiles of a raster data set that are completely contained in the provided polygon.

While spatio-temporal filter operations can simply be expressed using the appropriate UDF in the `WHERE` clause that then calls the underlying filter function on the RDD, joins require some extra handling.

As shown above, the join predicate is also expressed in the `WHERE` clause. The Apache SparkSQL optimizer Catalyst allows to inject own rules to transform logical into physical operators. The rules are tested for every logical operator in the plan. Hence, a rule gets a logical operator as input and if it can handle this logical operator, it returns an according physical operator. If the rule does not handle this operator, it returns an empty result. Catalyst privileges user defined rules and will not produce alternative physical operators based on other rules when a user defined rule successfully created a physical operator. Similar to UDFs, the rules are registered when the `STARKSession` is created.

STARK implements such a rule for joins. Spark automatically detects the relations in the `FROM` clause and the respective expression in the `WHERE` clause and creates a corresponding logical join operator. STARK's join rule tests if the current operator is a join with a spatial predicate and only then emits a physical join node.

The physical join operator creates the RDDs from the referenced names and creates a `SpatialJoinRDD` – the RDD class that implements the join in STARK.

STARK implements an additional rule to load only the partitions intersecting with the query range, just as described in Section 5.1.2. Spark automatically makes filter operations available in to the load operator. However, the default operator cannot handle the spatio-temporal predicate and would just ignore it. STARK's rewriting rule checks if the filter has a spatio-temporal predicate and if so, uses the spatio-temporal query object to identify the intersecting partitions using the `STSparkContext`. With the list of file names of these partitions a new physical load operator is created. That way, the logical NDP can be transparently utilized in SparkSQL.

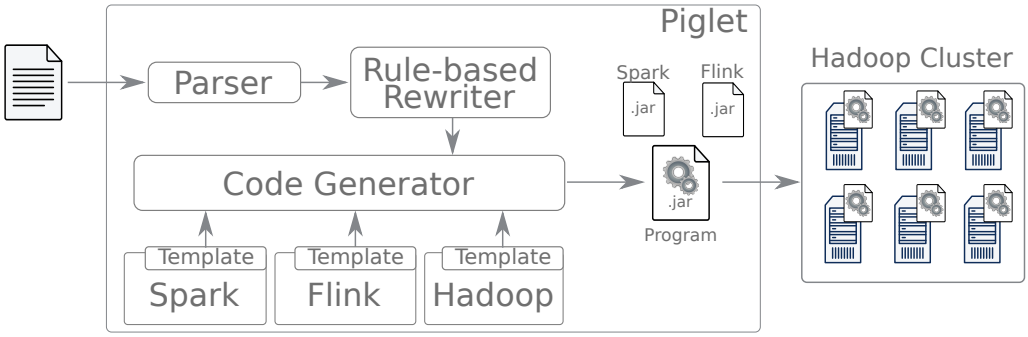


Figure 7.1: Overview of Piglet's internal architecture.

7.2 Piglet

SQL has been around for several decades and was adopted in many areas as the language of choice to query data sets. Its popularity has led to projects that use the declarative language to query not only databases, but even raw data files, such as NoDB/PostgresRaw [3] or `textql`[1].

Exploring large and often unknown data sets is one of the most challenging tasks in unlocking, preparing, and analyzing data to support decisions and derive models in various domains. Data exploration often implies the incremental analysis of data sets, starting with data cleaning and removing invalid entries and then finding a way to the information of interest. For those data cleaning and iterative algorithms SQL is often not flexible enough as it is suitable only for structured data in a relational form, but cannot be used to process unstructured data (e. g., log files).

In the following we describe our Piglet transpiler, which we proposed in [48].

7.2.1 Architecture

The Pig Latin language is very promising because of its flexibility and easy-to-remember syntax. With Piglet we adapt and extend the Pig Latin language and generate programs for various target platforms, instead of only Hadoop MapReduce as the original Pig Latin does. The overall architecture is shown in Fig. 7.1. The main goal of Piglet is to provide a declarative language and generate programs for various target platforms.

As shown in Fig. 7.1, Piglet accepts input scripts as files, but also provides an interactive shell, similar to the Spark shell or Pig's grunt shell. The scripts are parsed into a dataflow plan, which is a directed acyclic graph (DAG), as intermediate representation. Based on this DAG, various analysis and rewriting steps are performed similar to the logical rule-based optimization in a relational DBMS. These optimization rules include push-down of filters in the DAG, combination of filters, removing of duplicate filters, etc.

[1] `https://github.com/dinedal/textql`

After optimization, code for the selected target platform is emitted. This is done by traversing the `DataflowPlan`, the instance of the DAG, from the source operators to the sinks in topological sort order and append the Scala code for each operator to the program. For each operator, an emitter class exists that stores the template for it. Since the code for some operators is the same for all platforms, it is shared. For those operators whose code is not the same (syntactically, input parameters) special emitters are present for each target platform.

After the code generation phase has finished, the generated program is compiled into a jar archive and automatically sent to the cluster (or local host) for execution.

7.2.2 Language Extensions

As we discussed in the previous part of this work, Hadoop has no concept of spatio-temporal data. Hence, the original Pig Latin language does not have any spatial data types or functions. Therefore, Piglet extends the Pig Latin script language by the necessary types and functions to represent and process spatio-temporal data. Besides the extension for spatio-temporal data, Piglet includes even more extensions for various use cases that will be described briefly below. Note, since Piglet generates code for different execution platforms, not all extensions are supported by all platforms as they may require special operators. In Appendix B, more complex example scripts can be found that we also used during evaluation in Section 9.3.

RDF/SPARQL The Linked Open Data principle aims at providing information in a structured, machine readable format. For this, Resource Description Framework (RDF) has been established as a framework for representing objects with their attributes in *triples* of `subject` – `predicate` – `object`. The query language for RDF data sets in SPARQL whose main part are basic graph patterns (BGPs) that express the structure and pattern of result objects. Piglet implements a `BGP_FILTER` operator which lets users express such SPARQL-like queries inside a Pig Latin script. This way, RDF data can be integrated with e. g., CSV or other formats. An example usage of this `BGP_FILTER` is given below:

```
rdf = RDFLOAD('cdcollection.nt');
filtered = BGP_FILTER rdf BY {
 ?artist <produced> ?record .
 ?artist <country> ?country .
 ?record <release> "2018" .
};
aggr = GROUP filtered BY country;
```

This query finds all artists with their original country that produced a record in 2018. The result will have three attributes: artist, record, and country that can be referenced in subsequent operations.

Internally, the `BGP_FILTER` is rewritten by the optimizer into a sequence of filter and join operations. In [47] we defined the necessary rewriting rules as well as a tuplified format to improve processing of the RDF triples on Hadoop and Spark.

Streaming & Complex Event Processing Often, data is generated continuously and at such a high velocity, that it is not possible (or not desired) to store it permanently. Thus, queries are executed on a stream of data rather than on a static input file. Piglet contains extensions to connect to different sources for data streams, such as a TCP or ZeroMQ socket, which we also introduced in [51]. An important operation on streams is the window function that holds a snapshot of the data stream in order to compute aggregates. With Piglet's `WINDOW` operator, count-based or time-based windows can be expressed.

The following snipped shows how to connect to a TCP stream, create a tumbling window of 5 seconds and group the window content on the first attribute to count the number of occurrences of that value inside the window:

```
a = SOCKET_READ 'tcp://127.0.0.1:8889';
w = WINDOW a RANGE 5 seconds SLIDE RANGE 5 seconds;
grpd = GROUP w BY $0;
cntd = FOREACH grpd GENERATE group, COUNT(w);
```

Besides the plain stream processing operations, a frequent task is to find recurring patterns, such as complex events. Complex events are defined as sequences and combination of single events using logical operators (conjunction, disjunction, negation) as well as temporal relations. For complex event processing, Piglet implements a `MATCH_EVENT` operator that can be used to express such complex events. As a combination of the Linked Data principle and stream query processing, we discussed the complex event processing on Linked Stream Data in [85].

UDF While the provided operators implement functionality needed in many programs, often more sophisticated algorithms or mathematical calculations are needed. Such functionality is often present in external libraries. In the original Pig Latin, users can include *jar* files using a `REGISTER` command. This will make all classes and methods available as functions to the Pig Latin operators.

If the desired functionality has not been implemented in a library yet, or only a single function is needed, users have to create a library in order to include that in their Pig Latin program. To bypass this overhead, in Piglet users can directly embed Scala code inside a `<% ... %>` block. The embedded functions can be used in any Piglet operator, e. g., projection, filter, and join. In addition to embedded code, Piglet also supports macros using `DEFINE` as a shortcut for recurring sequences of operators.

7.2.3 Spatio-temporal Types and Operations

In the context of this work, we extended the Pig Latin language by spatio-temporal types and operations. Internally, Piglet uses the STARK library and thus, these extensions are only available for Spark as a target platform.

Data Types Spatial functions have been standardized in the SQL/MM standard. The Pigeon language, proposed in [32], adapts these operations into the Pig Latin environment. Pigeon is a library that is registered as UDF in a Pig Latin script and provides converters and into a spatial type as well as basic and analysis functions.

Pigeon uses the default column data type `bytearray` for the spatial feature column. When spatial functions are called on such a column it is automatically converted into a spatial object. Pigeon uses SpatialHadoop [33] as underlying execution engine on Hadoop. Piglet follows a different approach, which we will describe below. In the Pig Latin language, data sets are loaded by e. g., the `PigStorage` function that reads text files (mostly CSV). In such formats, every column is interpreted as a string or byte array and is later casted into the actual data type (integer, double, bool, etc.). To create spatio-temporal objects, Piglet has a constructor function to parse a string into a spatio-temporal object. To comply with the underlying STARK library, this constructor accepts a WKT string as well as a temporal expression, such as a single `Long` value (instant) or a pair of them (interval). Internally, in the logical plan in Piglet this operator is represented as a expression, namely `ConstructSTGeometryExpression`. Only during code generation phase, when this operator is encountered, it will emit Scala code that uses the `STObject` class from STARK and its constructors to initialize an instance. Similarly, with a `Tile` type it would be possible to load raster data sets: A textual or binary representation of a tiles is loaded and for every tile an element is created.

Operations Piglet provides various basic and also advanced analysis functions. Unlike in Pigeon, besides filtering and joining data sets, Piglet is also able to control the partitioning and indexing. A `ST_FILTER` operator allows to provide a spatio-temporal object as query range on the input data set. Optionally, the filter can be configured to use the online indexing mode of STARK (cf. Section 5.2.3). Similarly, the `ST_JOIN` operation used to perform a join with a spatio-temporal predicate can also be configured to apply a partitioning and/or indexing before the actual operation. In STARK, all operations are available only if the corresponding RDD contains tuples of `(STObject, Payload)`. Thus, the code emitters in Piglet for these operators

1. transform the data so that the `STObject` element is the key, then
2. execute the STARK operator, and finally
3. perform a re-transformation to achieve the initial schema, if necessary.

This way, users do not have to worry about how data has to be organized for execution – keeping the simplicity in declarative languages. The schema transformations are implemented using Spark's `keyBy` and `map` operations which are cheap transformations that do not incur any significant overhead.

The following example shows how data is loaded, converted to the geometry type and can be filtered using a spatial filter.

```
a = LOAD 'events.csv' USING PigStorage(',') AS (name: chararray, lat: double,
↪  lon: chararray);
b = FOREACH a GENERATE  name, geometry("POINT("+lat+" "+lon+")") as loc;
c = ST_FILTER b BY containedby(loc, geometry("POINT(50.1 10.2)"));
DUMP c;
```

In addition to query operations, we extended the language to include a `PARTITION BY` as well as an `INDEX` operation. As the names suggest, the operators allow to control the partitioning and indexing within a Pig Latin script.

Chapter 8
Reusing Intermediate Results in Dataflow Programs

As we described in the uses cases in Chapter 2, the creation of processing pipelines is often an iterative process. Data scientists need to understand the content of the input data set, perform some data cleaning and sanitation, and eventually find out how it needs to be integrated in order to solve their current task. Typically, this follows an incremental and explorative approach where the dataflow jobs are specified and executed step by step to inspect and validate results, test different parameters, and decide about subsequent steps to add further necessary operators.

As an example, researchers analyze the events stored in the GDELT data set. The CSV export contains more than 50 columns with many `NULL` values. Researchers working with this data set first need to find out which columns to use. Then, in the next version of the program invalid rows are removed and the result is inspected. After that, the events from all over the world are reduced to events in, e. g., Europe using a spatial filter. Finally, the actual algorithm, e. g., clustering can be applied.

After the successful creation of such programs, they are stored for later additional executions. Often, these programs also take input parameters, such as filter thresholds, columns to project to, or input paths and perform the sequence of operations to produce some result. Such a generic program can be used by a group of researchers and be executed by every researcher individually in the cluster.

The repeated execution of operators and whole parts of a query during data exploration as well as when many users work with the same data, leads to a situation where the same operations are executed again and again. Not only does this take unnecessary time to complete the query, it also occupies cluster resources that could be used by other programs, or in case of rented clusters, need to be payed. For example, an Amazon EC2 cluster has a per-hour pricing of around $ 1.8 (c4.8xlarge, Region Frankfurt) but storage is only $ 0.024 per GB (S3, 50TB, Region Frankfurt)[1]. If the execution times of the jobs running on such clusters can be reduced significantly, users could save a considerable amount of money.

Thus, to speed up execution and save resources, intermediate results, i. e., results of operators within a program that are not the final desired result, can be materialized to (persistent) storage. Repeated executions then can benefit from the existing materialized result and load this one instead of computing it again.

Storing every intermediate result is not feasible, as this will likely exceed the storage capacity of any cluster at some point in time. Therefore, only those intermediate results which are likely to be used again should be materialized. However, if the time to load materialized data takes longer than computing it again, which is the case for a large result set that was computed e. g., by a cheap filter with very low selectivity, reusing it will prolongate the execution.

[1] Prices from October 2019

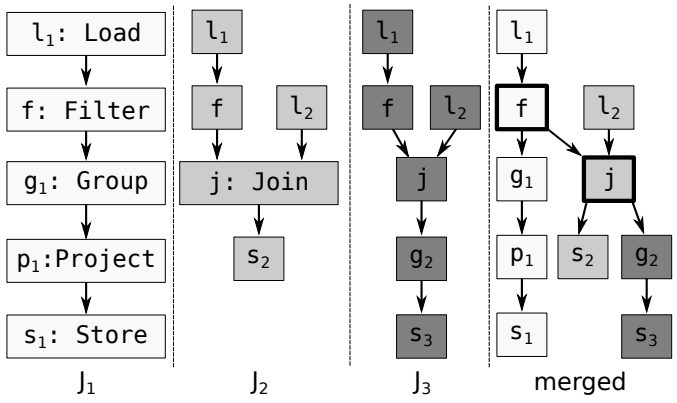


Figure 8.1: Merging multiple DAGs into a single DAG

Hence, we can identify three requirements for a system that allows to reuse intermediate results:

Selection Strategy: A selection strategy to decide which intermediate results of a query should be materialized is needed.

Cost Model: The selection strategy can consider the costs (in terms of execution time) of operators in the plan and the time saved when reusing the result of this operator during another execution.

Cache: When storage is limited, the total available space can be treated as a cache. When the cache capacity for materialized results is reached, a strategy to replace old result objects by new ones is needed.

In the following, solutions for the above mentioned requirements are proposed. We first start by identifying how intermediate results can be shared between queries, then define the basic structures for the decision model that implements the above mentioned requirements.

8.1 Opportunities for Reusing

Merge Strategy Over time, one or more users (data scientists, . . .) of the cluster create their set of scripts that load the input data set, perform transformations and calculations etc. If the input data set changes, e. g., because a new sensor took a new picture, a satellite mission transferred new data to the ground station, or a crawler extracted new values from web pages, the already existing scripts may need to be executed again to compute new result products over the updated information. In this case, the scripts may be sent in one batch to the execution engine.

The engine will parse the scripts in the batch, create a DAG for each of them and execute them – in the trivial case one after the other. However, since the scripts in the batch load the same input, they are likely to also perform the same pre-processing of the raw input data. To optimize resource usage in the cluster, the DAGs in the batch can be merged into a single DAG.

Algorithm 5 Merging dataflow plans

```
1: procedure MERGE(plans)
2:     resultPlan ← plans[0]
3:     for i ← 1 to plans.size − 1 do
4:         for all op in topDownDFS(plans[i]) do
5:             if resultPlan.contains(op.lineageID) then
6:                 continue
7:             else
8:                 resultPlan.addAndConnect(op)
9:             end if
10:        end for
11:    end for
12: end procedure
```

The idea is shown in Fig. 8.1: Operators that occur in multiple jobs/DAGs are present only once in the DAG of the result job. The algorithm to merge a list of dataflow plans into one plan is shown in Algorithm 5. The first plan is used as a basis for merging and will be copied completely to the result plan. After that, we iterate over all remaining plans to process them individually: Each plan is traversed using depth-first-search starting at the source (`LOAD`) operators and going down to the sink operators. For each operator that we find by traversing the plan, we check if we can find the operator in the merged plan by searching for its lineage ID (see Section 8.3.1). If the operator is found, it means that this operator and its predecessors are already part of the result plan, because they were present in one of the already processed plans, and it is therefore skipped and not added again. If the current operator is not found in the result plan, it has to be added and needs to be connected to the respective input and output operators. For connecting the operators, the information about input and output operators of the original operator is used, i.e. the respective operators can be found by searching for their lineage IDs. It may be the case that not all input or output operators are already part of the result plan. In this case the respective connections will be left unset as they will be established as soon as the according operator is added in one of the following cycles of the inner loop.

In Fig. 8.1, initial job J_1 is used as final merged plan for job J_m. When job J_2 is processed, we start at operator l_1 and find its lineage ID in J_m and therefore it is skipped. The same holds for the next inspected operator f. Then, j is processed and its lineage ID is not yet found in J_m. Thus, it is added to the merged plan. After all operators of J_2 have been checked, the algorithm starts the same process with job J_3. Here, the operators l_1, f, j, and l_2 are found in J_m and only g_2 and s_3 need to be added to J_m.

Materialization The *merge* strategy is only applicable for a batch of scripts. If a group of researchers access the data in an ad hoc fashion, they run their queries independently from each other and do not submit them in a batch. However, the scripts may also access the same data sets and hence, share the same operators.

In general, there might be a set of scripts which is not executed in a single batch, but in a transactional style.

To optimize response times and resource utilization, intermediate results, i. e., results of operators within the program, can be saved automatically or explicitly by a user in order to make them available for other scripts. This is similar to materialized views in relational DBMSs. The materialized results can then be loaded by other programs to avoid a duplicated execution of these operators. Fig. 8.2 on page 99 shows the general idea of the materialization strategy.

8.2 Related Work

Caching and reusing intermediate query results has been extensively studied for relational databases and data warehouses. Selecting partial results, or views respectively, for materialization [53, 23] as well as rewriting queries using these views [52] are two closely related problems that are used in database systems to improve query response time.

Early works on reusing materialized views (or derived relations) were done by Larson and Yang in [63, 109]. Other research results on the view-matching problems for SQL queries were published in [1] and [95]. In [81], the Hawc architecture is introduced that extends the logical optimizer of an SQL system and considers the query history in order to decide which intermediate result may be worth materializing to speed up further executions – even if this would create a more expensive plan which, however, is executed only once. A related problem is automatic index tuning which has been studied extensively [58, 89, 90, 102]. Here, recommenders analyze the given workload and underlying data and recommend to or autonomously create and drop indexes.

For Hadoop MapReduce the MRShare framework [74] merges a batch of jobs into a new batch so that groups of jobs can share scans over input files and the Map output. This is similar to our merging strategy described earlier. Other projects such as ReStore [36], PigReuse [20], or [103] are similar to MRShare in the sense that they all merge a batch of scripts into a single plan or share the intermediate results after a map phase. In PigReuse, the optimization goal is to minimize the number of operators and the number of generated MapReduce jobs - but they do not analyze the total cost of the generated plans.

Several additional frameworks were created for Apache Spark to support data analysts with their tasks. KeystoneML [93] is able to identify expensive operations in machine learning pipelines on Big Data platforms like Apache Spark. They employ a cost model using cluster costs (such as network bandwidth, CPU speed, etc.) and operator costs to estimate total execution costs. Using this, physical operators for a logical plan are chosen and materialization points are determined. RDDShare [57] is also based on Spark and simply identifies common operators in a batch of Spark programs and merges them into a single program.

Our work differs from the approaches mentioned in a way that they either only focus on merging a batch of scripts or they do not use a cost model for their algorithms. Furthermore, most related work is based on Hadoop MapReduce, except KeystoneML and RDDShare, which differs significantly from Spark's characteristics.

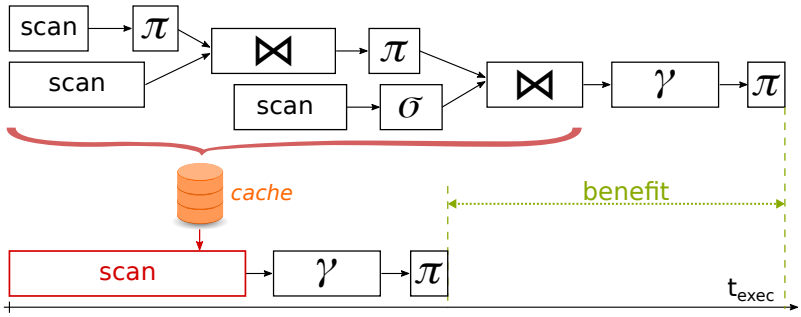


Figure 8.2: Runtime difference for loading materialized result.

8.3 A Cost-based Decision Model

This section describes the approach for a cost-based decision model which can be used to identify frequently used operator results worth materializing. The goal of our cost model is to identify those operators in the DAG where materializing their intermediate results speeds up subsequent executions most and is based on the execution time and result cardinality of operators. Fig. 8.2 shows a DAG where the width of a node's box represents its processing time. If, e.g., the result of the second join operator is materialized, subsequent executions of dataflow programs that also contain this part in their respective DAG will benefit by only having to load the already present result from disk.

This leads to two basic questions that need to be answered:

1. If multiple materialized results are present and applicable for reuse in a job, which of them should be loaded?

2. The intermediate result of which operators in the current job are worth materializing so that a subsequent execution will benefit most?

In order to support these decisions, our model introduces materialization points, for which the *benefits* are calculated. In the following, we will introduce and define these terms.

8.3.1 Foundations

Equivalence of Operators To be able to recycle intermediate results in a dataflow program, the system, or more specifically the optimizer of that system, needs to be able to decide if the result of an operator o is contained in the result of another operator p from a different job[2].

The decision if the result of one operator is contained in another one is not trivial and hard to decide. Furthermore, as discussed before, dataflow languages are extended by UDFs (or embedded code) that implement specialized algorithms. Such UDFs have to be treated as black boxes which make the containment check impossible.

[2]These two jobs may be the same script executed at two different times.

To overcome this problem, instead of testing for containment, one can test if two operators are *identical* and leave the containment check for future work. Two operators are identical, if their input operators are identical and they produce the same output. This check can be realized using unique lineage identifiers (LIDs). The LID is the hash code of the *lineage string*, which consists of the operator name (`LOAD`, `FILTER`, `GROUPBY`, ...) and the parameter values, such as input file name or filter predicates, combined with the lineage string of the direct input operator(s). We denote the lineage string of an operator o_i as $\mathsf{lid}(o_i)$. The lineage string of the filter operator with a simple predicate $x < 7$ (for some attribute x in the schema):

```
FILTER:x<7%LOAD:input.csv:1544956894
```

For the load operator the lineage string additionally contains the last modification date of that file (retrieved via the corresponding file system API) to be consistent if the underlying file was modified by another program.

Materialization Point A materialization point M marks a position in a DAG for the decision model for which the intermediate result is either to be materialized or existing materialized values can be loaded from storage. Thus, a materialization point is defined for an operator, but represents its result. For any operator o_i the corresponding materialization point is denoted as M_i, meaning that M_i is an alias for the output of operator o_i in the optimizer component.

We have to distinguish between *candidate materialization points* and *materialization points*. Candidate materialization points are those potential places in a DAG, where the intermediate result should either be materialized or could be loaded from storage. In general, any non-sink operator has the potential to be materialized and therefore can be regarded as a candidate materialization point. Besides the sink operators, source operators are also excluded for materialization, as the result of a source operator is the actual input data set and materialization would mean to store the input data set again. Obviously, this will not bring any additional value.

A materialization point is then a candidate materialization point which was selected for materialization or for loading the materialized data. The materialized data is treated as a cache and the set of all materialization points $M_1, M_2, \ldots, M_n$ that are currently kept in the cache is the *materialization configuration* $\mathcal{M} = \{M_1, M_2, \ldots, M_n\}$.

Benefit The goal of recycling intermediate results is to achieve some significant speedup when the same sequence of operators is executed repeatedly. In this context, the speedup is the saved execution time of the job. We call this amount of time saved the *benefit*. It is the task of the decision model to maximize the benefit and hence minimize the execution times of submitted jobs. The benefit is calculated as the amount of time saved when intermediate results are loaded instead of executing the complete job, as depicted in Fig. 8.2. Alternatively, one can also regard the benefit as the amount of money saved by needing to rent fewer machines in a public cloud.

The benefit and the decision model are both based on the actual costs of operators. While data management systems (such as DBMSs) maintain and update

statistics of the relations, such as number of blocks, tuple size, etc., such information is unavailable in the cluster processing frameworks like Spark and Hadoop, because they operate on plain text files. Therefore, the information required by the cost model for its decisions must be acquired during the execution of a job. In order to calculate the benefit for an operator o_i our model relies on the following statistics:

- **cardinality $\mathsf{card}(o_i)$:** The number of result tuples of o_i.
- **tuple width $\mathsf{width}(o_i)$:** The average number of bytes per result tuple of o_i.
- **execution time $t_{\mathsf{exec}}(o_i)$:** The duration it takes the operator to completely process its input data.

Since these statistics have to be collected at runtime during job execution, an operator with $\mathsf{lid}(o_i)$ has to be executed at least three times to benefit from materialization:

1. In the first run, no statistics are present and profiling code will be injected into the program to collect the needed statistics.
2. When o_i is encountered a second time, the optimizer can determine from the now existing statistics, if o_i is expensive and whether a speedup can be achieved for subsequent runs if the result of this particular operator is materialized. The rewriter, as part of the optimizer, then is instructed to insert a materialization operator that will write the results of o_i to persistent storage along with a mapping of $\mathsf{lid}(o_i)$ to the produced file.
3. In a third run, the materialized result for $\mathsf{lid}(o_i)$ is found and considered to be loaded instead of executing the operator and all its predecessors. In this case, the rewriter will insert a source operator that reads the materialized file and removes o_i as well as all its predecessors recursively from the DAG. A predecessor o_p of o_i can, of course, only be removed if no other operator o_j in the DAG depends on o_p.

With the available statistics, the benefit t_{benefit} of a materialization point M_i can be expressed as in Eq. (8.1).

$$t_{\mathsf{benefit}}(M_i) = t_{\mathsf{total}}(M_i) - t_{\mathsf{read}}(M_i) \tag{8.1}$$

$$t_{\mathsf{total}}(M_i) = \sum_{o \in \mathsf{prefix}(M_i)} t_{\mathsf{exec}}(o) \tag{8.2}$$

Here, $t_{\mathsf{total}}(M_i)$ denotes the cumulative execution time of operators in the prefix of o_i from the source to M_i and $t_{\mathsf{read}}(M_i)$ is the time required to read the materialized data of M_i from storage.

The calculation of $t_{\mathsf{total}}(M_i)$ depends on the prefix of M_i. If the prefix of M_i does not contain a join (or similar) operator, $t_{\mathsf{total}}(M_i)$ can be calculated as in Eq. (8.2). If the prefix of M_i does contain a join (or similar) operator j, with $k_1(j), \ldots, k_n(j)$ as the direct inputs to j, only the longest (concerning execution time) of those branches is considered:

$$t_{\mathsf{total}}(M_i) = \max\{t_{\mathsf{total}}(k_1(j)), \ldots, t_{\mathsf{total}}(k_n(j))\} + \sum_{\substack{o \in \mathsf{prefix}(M_i) \\ \wedge o \notin \mathsf{prefix}(j)}} t_{\mathsf{exec}}(o) \tag{8.3}$$

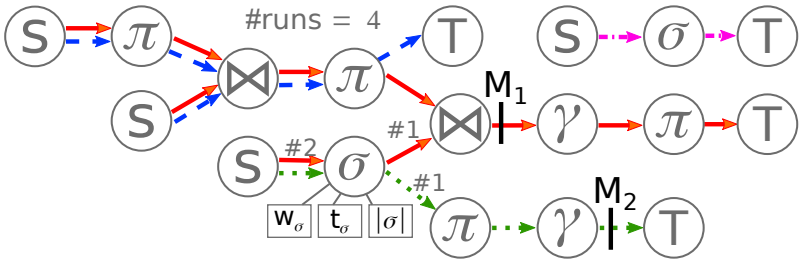


Figure 8.3: The global operator graph for four jobs. Stored information on nodes and edges is exemplary shown for a single operator node.

This means to take the maximum time of all input branches of the join operator and add the execution times of all other operators in the prefix of M_i which are not part of the join input, i. e., are located between j and o_i. Since the input branches of the currently considered join operator may contain other join operators as well, this approach has to be applied recursively to calculate the costs.

$t_{\mathsf{read}}(M_i)$, the time required to read the materialized result of M_i, depends on the number of bytes that can be read per second (*bps*) in the cluster the application is running on. In centralized systems the read performance only depends on the possible throughput of the storage device. HDFS, however, stores replicas for each block of a file, and thus, the performance also depends on the distribution of the blocks in the network. Hadoop contains tools to test the IO performance of the cluster installation.

$$t_{\mathsf{read}}(M_i) = \frac{\mathsf{card}(o_i) \cdot \mathsf{width}(o_i)}{bps} \tag{8.4}$$

Global Operator Graph The idea of materializing frequently used intermediate results requires to store the history of executed jobs and their operators. We call this history the *global operator graph*; a DAG that was created by merging the DAGs of all ever submitted jobs. The global operator graph is used to calculate the benefit of candidate materialization points during the rewriting phase and thus also stores the collected statistics on the operator nodes. In order to make this history information available across multiple jobs, the graph is stored persistently.

The collected runtime statistics are stored on the nodes, representing operators which are identified by their respective lids. The frequency how often an operator followed another one in all executed jobs is stored in the edges between these operators. Besides these operator and execution statistics, the global operator graph also serves as a meta catalog for materialization points, i. e., already materialized results. Fig. 8.3 shows an example of such a global operator graph for four jobs J_1 (solid red edges), J_2 (dashed blue), J_3 (dotted green), and J_4 (dashed purple). In reality, in the model there is only one edge between the nodes. The multiple edges are just for illustrating the different jobs.

Initially, the graph is empty and the DAG of the first job to be executed is used as the global operator graph. For all other jobs, their DAGs are traversed in topological sort order operator by operator, starting at the sources. If the current operator is not present in the graph, yet, it will be added and connected to its parent operator(s). The new connection is an edge in the graph with an initial value

of one. Using topological sort order guarantees that the parent operator(s) have already been added to the graph. If the current operator is already present in the graph, this means that a program with the same prefix was executed before. In such a case only the respective edge value is incremented by one and then the next operator in the current job's DAG is being processed.

After the execution of a job has finished and all statistics (runtimes and result sizes of the operators) have been collected (cf. Section 8.4), they are added to the respective nodes in the graph. If the operator is executed for the first time, no statistics are present for this operator and the collected values are simply added to the node in the graph. If the operator was already executed before as part of another job, present statistics are merged with the newly collected ones by averaging them.

The graph serves as input for the decision model and is used to calculate the benefit based on the statistics and materialization points.

8.3.2 Decision Model

Like in traditional DBMS, the decision model is part of an optimizer component in the system. It is used to answer the two questions posed in the beginning of this section.

Which Materialization Point to Load? Answering the first question is straightforward. From the list of candidate materialization points for a given job, only those are selected for which materialized results are present. Then, from these candidates, the one materialization point that will result in the highest benefit is chosen to achieve the greatest speedup. If the job contains multiple paths, which are then combined in a join (or similar), selecting multiple materialization points, one for each path, is possible.

Which Materialization Point to Store? For the selection of a candidate materialization point to materialize to persistent storage, the decision model has to consider three orthogonal dimensions:

1. Which candidate materialization points are meaningful at all?
2. How to rank the resulting candidate materialization points from (1) to decide which are materialized?
3. How to handle cache overflows and decide what to remove from cache?

1. Candidate Materialization Point Selection Obviously, a sink operator is not a candidate for materialization as the result is either written to persistent storage anyway or printed to screen. In the latter case, the materialization point before that sink operator would be one whose result could be re-used by another job. A source operator will only read data from storage and pass it to the next operator without modification. Hence, materializing the output of a source operator will write the same data back to disk and no benefit can be gained from this. Therefore, subsequent operations only need to consider candidate materialization points that do not belong to a source or sink operator.

2. Strategies for Ranking Materialization Points To decide which candidate materialization point to really materialize, different strategies exist, which might be suitable for different use cases:

Last A naïve but intuitive strategy for selecting a materialization point is to always choose the *latest* possible one. This is the one materialization point that is closest to a sink operator. If a job contains n sinks, the materialization point before each sink is selected, which means to write n intermediate results. This is a simple caching strategy and might work well during the incremental development of scripts, described earlier. However, this bears the "risk" that the materialized result will not be needed again, e.g., if subsequently executed scripts are not the next step of the incremental development, but branch off at another operator so that an earlier materialization point would have been a better choice. Furthermore, the last materialization point may only bring a small (or even no) benefit for reusing.

MaxBenefit This strategy chooses that one materialization point with the highest benefit. Compared to the previous strategy, where the last one is selected, it is guaranteed to bring the best possible benefit when the result is needed again. Like in the previous strategy, if the result is not needed again, materialization was pointless.

Markov The selection of the materialization point(s) should be considered as an optimization problem, regarding the required space of the respective intermediate result on disk and the possible benefit: Since available space on disk is limited, only those materialization points with high chances to be needed again should be selected. Thus, selection of the materialization points should consider the probability for reuse – for which a Markov chain can be applied.

In fact, the result of the Markov strategy should be regarded as a two dimensional (benefits and probabilities) optimization problem to maximize the benefit as well as the probability for reuse of the selected materialization points. The result of this optimization problem are all points where there exists no other point with both a higher benefit and higher probability. All points in the Pareto front mark materialization points with either a high probability and/or high benefit, thus being worth materializing. If only one materialization point should be selected, it has to be chosen from the Pareto front. For this, the probability of re-occurrence of a materialization point can be considered as the weight for the benefit, so that the materialization point with the highest product of probability and benefit should be selected:

$$\begin{aligned}\{M_i \in \mathcal{M} \mid \nexists M_j \in \mathcal{M}, i \neq j : \\ P_{total}(M_j) * t_{benefit}(M_j) > P_{total}(M_i) * t_{benefit}(M_i)\}\end{aligned} \tag{8.5}$$

If this set contains multiple elements, one can be chosen arbitrarily or user specified weights can be applied to express a favor of one dimension over the other.

In Eq. (8.5), $P_{total}(o_i)$ denotes the minimum probability found on the path in the DAG from the source operators to o_i:

$$P_{total}(o_i) = min\{P_{o_k,o_l} | o_k, o_l \in prefix(o_i), o_k \rightarrow o_l\} \tag{8.6}$$

where $o_k \rightarrow o_l$ means that o_k has an edge o_l in the DAG. P_{o_k,o_l} describes the probability that o_k will be followed by operator o_l and can be calculated in different ways. One approach is to put the frequency into relation of the total number of executions, with respect to some time window $\mathcal{W}$. The probability P_{o_k,o_l} then would be as in Eq. (8.7).

A second approach is to put the frequency in relation to the number of operators that follow o_k, as shown in Eq. (8.8).

$$P_{o_k,o_l} = \frac{f^{\mathcal{W}}_{o_k,o_l}}{min(\mathcal{W}, runs)} \tag{8.7}$$

$$P_{o_k,o_l} = \frac{f^{\mathcal{W}}_{o_k,o_l}}{deg^{+}_{\mathcal{W}}(o_k)} \tag{8.8}$$

Here, $f^{\mathcal{W}}_{o_k,o_l}$ is the plain frequency count for the transition stored on the edges, that lie within the considered window $\mathcal{W}$, $runs$ is the total number of jobs that are executed by the system, and $deg^{+}_{\mathcal{W}}(o_k)$ is the outdegree of a node o_k.

3. Cache management. Materializing the intermediate results requires sufficient storage capacity. However, as for any cache, the available space is limited to some extent and not everything can be kept forever. When a new item is supposed to be added to the cache, but there is not enough space available, stragegies to decide which items are kept and which are removed from the cache are needed. Here, the new entry is the materialization point selected by the cost model to materialize. While for database buffers the new element must be added, possibly replacing an existing one, for our materialization scenario a decision may be that the new entry must not evict an existing one. Hence, a materialization point selected by the cost model may eventually not be materialized, because there is not enough space available in the cache and no existing entry can be removed from the cache.

Straightforward cache replacement strategies are for example least recently used (LRU) and least frequently used (LFU). The LRU will track the access times of the cache entries, i. e., the timestamp when the respective materialized data of the materialization point was loaded by a script. To make space for new data, this strategy will remove those entries from the cache that have the oldest timestamp, and thus have not been loaded recently. The LFU will count the number accesses (reads) of the materialized data and remove those with the lowest access count values.

The previous two strategies only consider the accesses to the materialization points, but neither their benefit for the jobs nor their occupied space on disk. In buffer pools of DBMSs, the entries are always of the same size (the page size). In the materialization scenario, however, the cache entries typically have different sizes. A replacement strategy must ensure to maximize the cumulative benefit of the contained materialization points. Since a materialization point has a *value* (the benefit) and a weight (storage space), this can be treated as the well-known Knapsack problem, which can be solved, e. g., with dynamic programming or with a greedy method over the benefit–weight fraction.

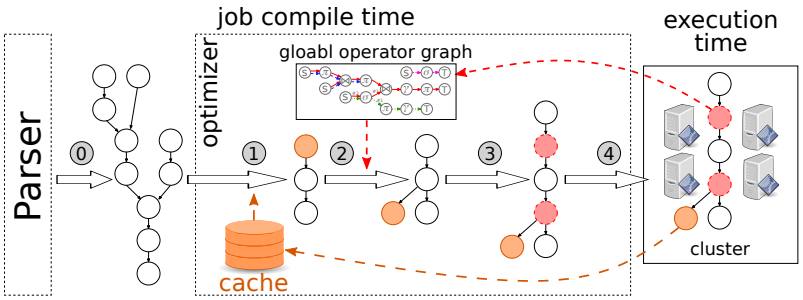


Figure 8.4: Architecture overview: (0) Transform script to DAG, (1) Insert `LOAD` for existing data, (2) Insert `STORE` (based on statistics), (3) code instrumentation for profiling, (4) execute as Spark job.

While the Knapsack helps keeping only those materialization points with high benefits or benefit–weight relations, it does not include the actual number of reuses of a materialization point.

Therefore, we also incorporate the access frequency into the value of an item in the bag. The value v of a materialization point M_i is calculated as

$$v(M_i) = f * t_{\text{benefit}}(M_i) \tag{8.9}$$

where f is the number of reads of M_i.

8.4 Profiling Dataflow Programs in Piglet

Fig. 8.4 shows the general architecture overview of our approach. In step 0, the script is parsed into an intermediate representation, the `DataflowPlan` representing the DAG. Then, the optimizer component receives the DAG for the current job and after applying general rule-based optimizations, the DAG will be modified for recycling. We employ a cache that will store the materialized results. Ideally, the cache should have access to HDFS for persistent storage. If materialized data exists for a part of the current DAG, that part will be replaced by a `LOAD` operator that reads the cached data. In step 2, the global operator graph that stores profiling information is checked to determine if operators of the current DAG should be materialized and respective `STORE` operators are inserted. After that in step 3, profiling operators are inserted to collect runtime statistics of the operators in the graph. During execution on a cluster (step 4), those operators will send information to the optimizer which will update its statistics. If the optimizer decided in step 2 to materialize intermediate results, those will be written to the cache by the inserted `STORE` operators.

We implemented the described cost-based decision model and profiler into our Piglet project. However, we would like to emphasize that the model described in this work is neither restricted to Piglet nor Spark and could be adopted into other platforms. The details of the decision model that uses the information in the global operator graph as well as the cache will be explained in more detail in the next sections.

In order to gain the desired statistics information, there are two options: (1) The optimizer is started separately to analyze the input file and create a profile. Before

executing a job, the optimizer then tries to come to a decision based on the statistics and selectivity estimations of the involved operators. (2) The job is instrumented with code that collects the necessary statistics during execution and the runtime or execution platform has to be extended by such an optimizer that manages and utilizes the collected data. The first approach is more or less what traditional DBMS do and is currently also being implemented in Apache Spark for SparkSQL.

However, the second approach has the advantage that execution time is measured as well as the result size, instead of relying on estimations that are based on assumptions. In our evaluation we will show that the instrumentation does not incur in any significant overhead.

Result Size To determine the total number of bytes in the result of an operator and thus the size of the materialization point, we use Java's built-in byte serialization capabilities to create a sequence of bytes for an object. The number of bytes produced by this serialization is then assumed to be the result on disk. However, experiments showed that there still is a discrepancy between the calculated and actual size on disk. Nevertheless, these calculated value are still a good estimate and better than the estimates produced by Spark's `SizeEstimator`.

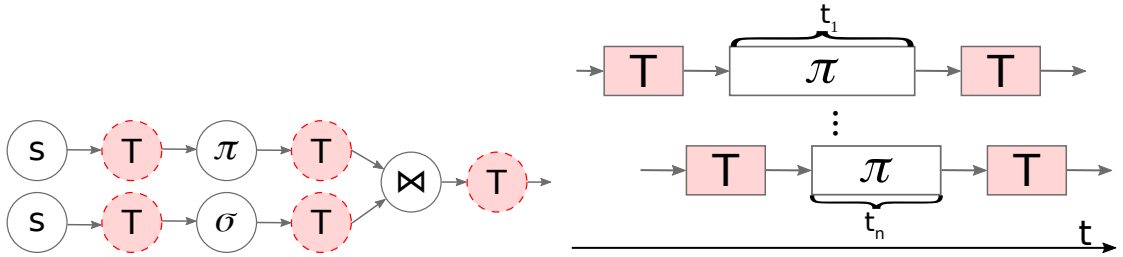
We sample the result of the operator and pass each result tuple individually into the our estimator. The information for each partition is accumulated using Spark's accumulator mechanism and sent to the optimizer. From the received information the optimizer can calculate the average tuple size as well as the total number of tuples in the result.

Execution Time per Operator. A more difficult task is to measure the execution time of an operator. Spark comes with a `SparkListener` interface that provides information about the status of the current execution including start and completion time of the tasks and stages that form the job. However, relying on the execution times of the stages is too coarse for our goal as possible materialization points for recycling would be after a stage only. Thus, there would not be many of such materialization points and, more importantly, we would lose most operators that are shared among different jobs, because they are hidden inside a stage and thereby reducing the usefulness of the idea.

We therefore implemented our own approach to measure the execution duration of an operator per partition based on code instrumentation. On the logical level, *timing* operators are inserted between all other operators in the plan, as depicted in Fig. 8.5(a). The task of the timing operators is to send a message to the profiling manager component of the optimizer with the current system time, when they are executed[3]. The profiling manager will receive a timestamp and the lid of the operator for each partition and calculates the average execution time of the operators based on this information.

The realization of this concept needs to deal with Spark's lazy evaluation as well as the data parallelism. Thus, for each timing operator we inject code to report the current time when a partition is processed (using `mapPartitions`). The partitions

[3]This requires that the clocks on all nodes are synchronized, of course. For example via NTP.



(a) Conceptual *timing* operators in the DAG. (b) Parallel instances of *timing* operators during execution.

Figure 8.5: Measuring operator execution time in a shared nothing cluster.

that an operator instance in each task processes can be of different sizes, although the platforms try to keep them balanced to avoid skewed workload on the nodes. Thus, the operator instances require different times to process their input. For n partitions, it results in n different execution time information, from which we have to derive an overall execution time for an operator (cf. Fig. 8.5(b)). In our approach we use the average of these n collected times. Other strategies (min, max, median), however, are also possible and *min* or *max* could be used to implement an optimistic or pessimistic behavior. Since an RDD has information about its parents, it is enough to only insert this code after the operator and let the profiling manager calculate the execution duration, based on the received times of the respective parent operator.

The values produced by this profiling are sent to the optimizer which collects and aggregates this information and eventually uses it for its decisions. We will evaluate the effectiveness of the decision model in the evaluation in Section 9.3.

Chapter 9
Performance Evaluation

In this evaluation we will analyze the performance of the spatial and spatio-temporal vector and raster data processing operators as well as the impact of partitioning, indexing, and recycling intermediate results. We will compare the different aspects and alternative approaches for the discussed operators in STARK, but also compare them with other frameworks designed for this type of data processing.

In the following experiments, we first concentrate on the STARK framework and the related projects and perform a benchmark test to analyze their performance. After that, we demonstrate the impact of the transparent and automatic intermediate result materialization on the general query performance.

Although Apache Spark is used for processing large amounts of data, users often run interactive ad hoc queries and want to see the results immediately. Therefore, the goal of this benchmark is to study how well the here considered frameworks perform in such a workload setting. For benchmarking, we use the experiences from our proposal in [46] as well as queries and data from Pandey et al. in [79]. Unless otherwise stated, time measurements include the complete processing pipeline consisting of loading data from HDFS until the last tuple has been consumed.

As Sidlauskas and Jensen pointed out in [92], comparing frameworks is complicated, because differences in implementation details of the same algorithm in different systems might have a significant impact on the execution performance. Thus, in the following, when we compare STARK with other systems, we compare the implementations as well as the algorithms and approaches of them.

After the performance evaluation of the spatial data processing frameworks, we perform tests that show how the execution time of dataflow programs is reduced by recycling materialized intermediate results. Here, we inspect the overhead that the profiling and rewriting cause and also analyze the impact of the three different selection strategies to decide which intermediate result to materialize.

The experiments were executed on our Hadoop cluster of 15 nodes, where each node has an Intel Core i5 processor (3 GHz), 16GB RAM, and a 1 TB HDD. The nodes run on Ubuntu 14.04 with Hadoop 2.7, Scala 2.11, and Java 1.8.0u102. The jobs for the experiments were submitted to the cluster with `master` set to YARN and deploy mode `client`. The latter causes the driver to run in the local machine, which is a desktop PC with 16 GB RAM and an Intel Core i7-2600 CPU with 3.40 GHz. All machines are connected via a 1 GiB/s LAN. In the experiments, we use the data sets briefly described in Table 9.1. Note, the `weather`, `taxi`, and `block` dataset do contain spatial and temporal information. Though, we did not use them for spatio-temporal query processing and therefore mark them with "–" in the overview.

Table 9.1: Datasets used in the evaluation.

Name	geometry type	temporal values	No. objects	Filesize
point	points	no	200 mio.	4 GB
linestring	linestrings	no	72 mio.	17.5 GB
rectangle	rectangles	no	114 mio.	13.3 GB
polygon	polygons	no	114 mio	18.2 GB
sentinel	points & polygons	yes	2 mio.	5.3 GB
gdelt	points	yes	127 mio.	48 GB
weather	–	–	180 mio.	34.7 GB
taxi	–	–	550 mio.	90 GB
block	–	–	38000	18 MB

9.1 Impact of Spatio-temporal Partitioning and Indexing

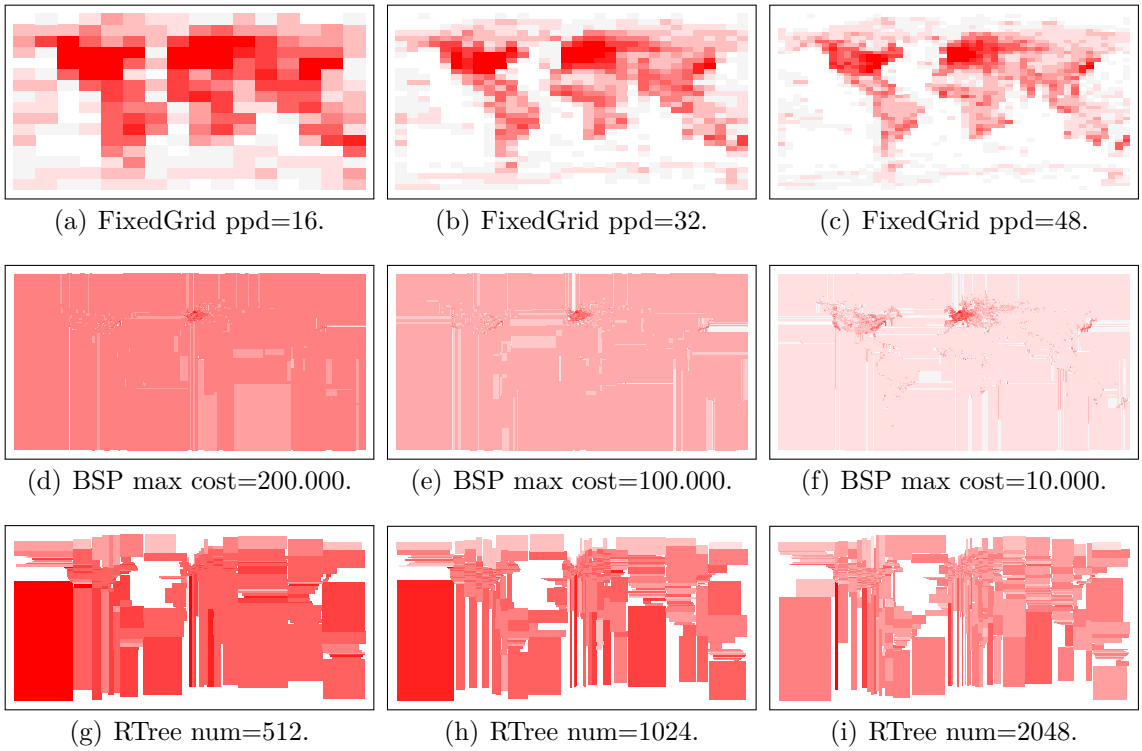
In this section we analyze the impact of the partitioners and indexes on the query performance. The goal is to identify cases when partitioning and/or indexing must be applied and when the overhead of their creation becomes too heavy.

9.1.1 Partitioning

Strategies In Section 5.2.2 we discussed three different partitioner strategies: fixed grid, cost-based, and R-tree partitioning. The partitions they create as well as the number of elements inside each partition greatly varies among the strategies, but also for different parameter values. The generated partitions, in turn, have an impact on the execution time of operators.

We first analyze the partitions generated by STARK's partitioning strategies in Fig. 9.1. For each strategy, three different parameter values are used and we count the number of objects assigned to each partition. As input data, we exemplary used the `point` data set. The more elements a partition contains, the more intense/saturated its color is displayed. White areas are empty and no partition covers them. As we discussed earlier, the goal of partitioning is to create partitions with (almost) equal number of elements. Therefore, for a good and balanced partitioning, the generated partitions should all have the same color.

For the FixedGrid partitioner, we divided each dimension into 16, 32, and 48 segments, respectively. For the BSP, we used a cell size of 1 and max cost of 200.000, 100.000, and 10.000. The R-tree partitioning was set to create 512, 1024, 2048 partitions.



(a) FixedGrid ppd=16. (b) FixedGrid ppd=32. (c) FixedGrid ppd=48.

(d) BSP max cost=200.000. (e) BSP max cost=100.000. (f) BSP max cost=10.000.

(g) RTree num=512. (h) RTree num=1024. (i) RTree num=2048.

Figure 9.1: Partitioning results for STARK's partitioners with three different parameters each. Number of generated partitions increases from left to right.

Figures 9.1(a) to 9.1(c) show the results of the FixedGrid partitioning. One can clearly see that with the increased number of partitions, the more accurately the actual data distribution is modeled. Though, that is not a quality criterion. There are many partitions with light-red color (containing few elements) and also lots of dark-red partitions with a large number of elements. As a consequence, worker nodes might be overloaded when they need to collect all elements into memory, e. g., to build an index, as we encountered for the kNN search below.

For BSP, the number of elements per partition is influenced by the *max cost* parameter. The larger this value is set, the fewer partitions are generated. In Figure 9.1(d) all partitions have almost equal number of elements and only in Europe, some small partitions with more elements than the others were created. However, with the decrease of *max cost*, the number of partitions increases, because the algorithm needs to further split partitions exceeding this value. This results in equally-sized partitions in sparse areas, but partitions with more elements in dense areas. These high-load partitions exist because their extent equals to a cell and cannot be further split. To achieve an even more balanced partitioning, the cell size would need to be decreased. This, however, results in longer times needed to compute the histogram and the partitions bounds.

For the R-tree partitioning, all partitions are also of almost equal size, even in the dense areas. However, note that this partitioning was computed on a sample of the input data, while the fixed grid and BSP were created on the full input data set. The reason is that the tree is constructed in the driver and therefore, all objects have

Table 9.2: Partitioning characteristics. StdDev is the standard deviation over the number of objects per partition, t_{comp} is the time to compute partitions bounds, t_{repart} is the to repartition the data set.

Partitioner	Parameter	No. Partitions	Avg. No. Objects	StdDev	t_{comp}	t_{repart}
FixedGrid	ppd = 16	224	892857	4173843	0.03	42
	ppd = 32	695	287770	1311427	0.03	38
	ppd = 48	1343	148920	709907	0.03	42
BSP	max = 200.000	1147	174368	132401	28	90
	max = 100.000	1954	102354	116735	23	133
	max = 10.000	6628	30175	75128	47	504
R-Tree	num = 512	529	378072	214997	0.7	70
	num = 1024	1056	189394	116375	0.6	112
	num = 2048	2070	96618	60273	0.7	317

to be transferred to this node. This causes long data transfers, computation time, and huge memory consumption on this node. A dedicated R-tree for partitioning could just store the extent of the leaf nodes without the actual objects to save space.

Table 9.2 shows some characteristics of the generated partitionings. The fixed grid partitioner removes empty partitions, so that *No. Partitions* does not necessarily match ppd^2. This can also be seen in Fig. 9.1 where the white areas are not covered by any partition. Furthermore, the standard deviation of the number of elements per partitions ranges from 700.000 to more than 4 million. Thus, there are extremely large partitions which causes problems when the content of a partitions needs to be completely fetched into the executors memory e. g., to build an index. For BSP, the average number of objects per partition matches the desired maximum number quite well. Discrepancies arose due to extremely dense areas that cannot be further split (due to the selected cell size). This is also one reason for the standard deviation of the number of objects per partition. The R-tree partitioning also created the desired number of partitions with an acceptable variance in the number of objects per partition.

The time t_{comp} in the table represents the time required to compute the partition bounds from a cached data set, but not to repartition it. For the FixedGrid strategy, the time is independent of the number of partitions, since the grid is statically generated and only the bounds have to be calculated. The R-tree partitioner works on a sample of the data. To create the sample, we instructed Spark to randomly collect a sample with a fraction of 0.001. Thus, the sample contains around 200000 elements in a local array, making the very fast computation times possible. Only the BSP completely scans the input data to create the histogram and compute the partitions based on this histogram.

Although the BSP needs some more time than the R-tree partitioner, it does not have to sample the data and therefore eases the computation of the partitions extent information. Furthermore, as already mentioned, the R-tree is created locally in the driver. Depending on the sample size, the required resources might exceed

the drivers capacities and result in errors (which cause restarts and even extend the execution time). We therefore use, unless stated otherwise, the BSP in the following experiments.

Hypothesis 1. *The overhead incurred by repartitioning data according to the spatial or spatio-temporal key can be amortized when expensive operations benefit from the partitioning information. Though, this overhead cannot be amortized when the partitioning is used only for a single query.*

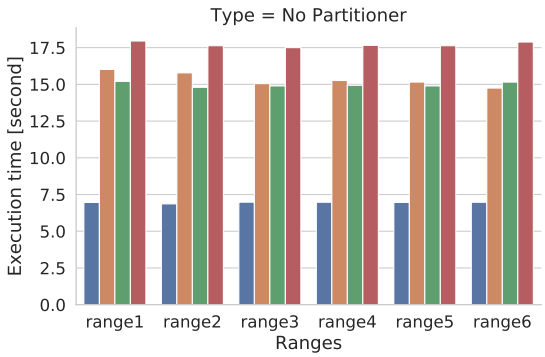
To show that the costs for repartitioning can be amortized, we can consider the example of the spatial or spatio-temporal join. Without a spatial or spatio-temporal partitioning, all native partitions generated by Spark would have to be checked against each other, resulting in, regarding the partitions, a Cartesian product. We experimentally ran this join on a reduced point dataset (10 million points) with the polygon dataset and had to terminate it after 90 minutes. As we will show later, a join on partitioned (and indexed) data sets finishes within a few seconds.

However, there are cases where the partitioning incurs so much overhead, that it cannot be amortized during query evaluation. The spatio-temporal filter is an example for a quite cheap operation that can easily be executed without a spatial partitioning.

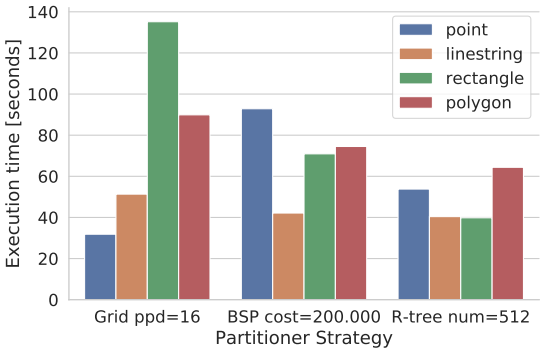
Range queries In a typical use case, users load a spatio-temporal dataset from HDFS and apply a filter with a query range to find all objects of the dataset within this range. In Fig. 9.2(a), the execution times of six spatial ranges on the four input data sets are shown. The time includes reading the dataset from storage and apply the filter without any spatial partitioning. The selectivities of the query ranges are as follows: range1: 0.001%, range2: 0.01%, range3: 1%, range4: 10%, range5: 50%, range6: 100%. Range 1 has the highest selectivity (smallest region) and Range 6 selects the complete data set. One can easily see that the runtimes in this figure are much shorter than the times required alone for partitioning the datasets shown in Fig. 9.2(b), regardless of the used strategy.

Hypothesis 2. *While the overhead for partitioning is too large for a single spatial range query, materializing the partitioning information and using it to reduce data to load significantly improves the query execution time and throughput.*

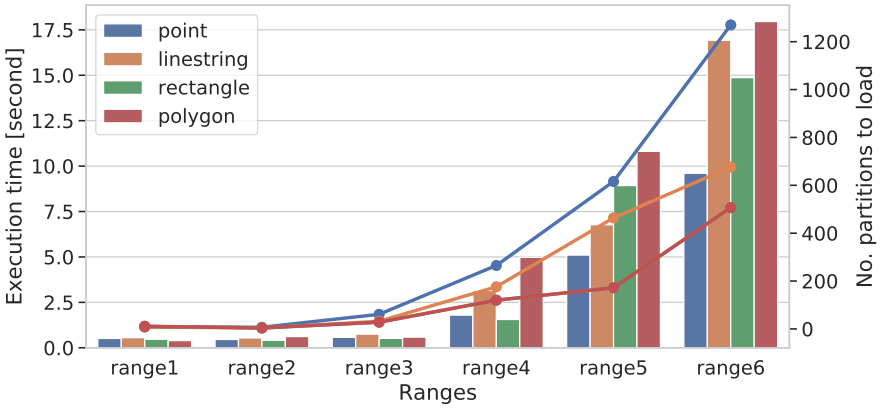
The current implementation in STARK for using materialized partitioning information is the logical near data processing as discussed in Sections 5.1.2 and 6.7. Here, we first load the global index and apply the filter predicate to identify the partitions containing result candidates. After that, only those partitions are loaded and processed. This two step approach requires two accesses to HDFS. However, loading only a few partitions has a significant impact as shown in Fig. 9.2(c). For ranges 1 – 3 the execution times are around 1 second or even below and, therefore, only a fraction of the values shown in Fig. 9.2(a). However, with the increased range size and the decreased selectivity, more partitions with result candidates are identified and need to be loaded. Naturally, the benefit of this near data processing approach disappears when all partitions need to be loaded.



(a) Range queries without spatial partitioning.



(b) Runtimes of spatial partitioners for the four datasets ($t = t_{comp} + t_{repart}$.)

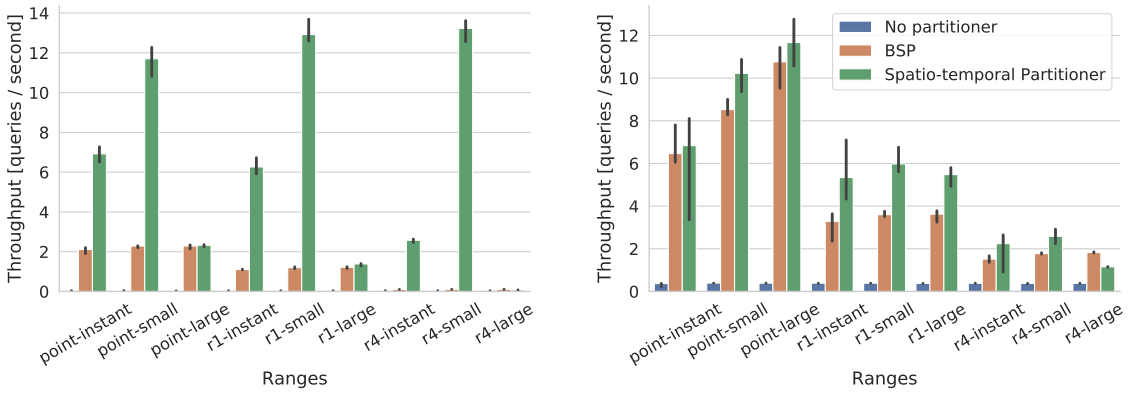


(c) Range queries with logical near data processing in load function.

Figure 9.2: Comparison of execution times for (a) range queries without spatial partitioning, (b) raw runtimes for partitioner, (c) runtimes for range queries with pushing down filter predicate into load function.

STARK is not only able to handle spatial, but also spatio-temporal data. We therefore evaluate the logical near data processing of spatio-temporal range queries, too. Since the datasets from the previous tests do not contain any temporal information, we used two other real world data sets: `sentinel` is from the meta data catalog of a satellite mission containing the areas covered by the measurements as 20 million polygons. The `gdelt` dataset contains 127 million events from 2013 to 2015. The values used in the query ranges are as follows:

- `point`: (39.961727, 84.47858) – randomly selected
- `r1`, `r4`: query ranges `range1` and `range4`, from previous experiment
- `instant`: Sat., 10 August 1985, 9:53:47 AM GMT (timestamp: 492515627)
- `small`: one day interval: Sunday, 23 Oct. 2016, 7:16:51 PM to Mon., 24 October 2016, 7:17:05 PM GMT
- `large`: interval from `instant` to Mon-, 24 Oct. 2016, 7:17:05 PM GMT



(a) Filter on `gdelt` dataset.

(b) Filter on `sentinel` dataset.

Figure 9.3: Spatio-temporal filter operation with logical near data processing for `gdelt` and `sentinel` data sets.

Figure 9.3 shows the throughput per second for the nine query ranges on the two data sets. We executed this experiment in three ways: without applying any partitioning (i. e., loading and filtering the data directly), with a spatial partitioning only (using BSP with max cost 200000), and with a spatio-temporal partitioner that internally also used the BSP as well as a fixed range temporal partitioner (numpartitions = 10). Prior to execution, the partitioning information was stored in the global index for each data set. As before, one can see that the logical near data processing in general has a great advantage over the variant without partition pruning (*No Partitioner* in the figure). Furthermore, for the larger `gdelt` dataset, the spatio-temporal partitioning achieves a much higher throughput than the other variants in many cases. Only when the temporal range is large, not many partitions can be excluded from being loaded based on their temporal characteristics. This leads to the (almost) same number of partitions to load as in the variant with the BSP partitioner. For `r4-large`, much more partitions than for the other query ranges had to be loaded so that the difference does not become visible in this figure. Without any partitioning, this query took around 30 seconds while with partitioning, execution time was around 9 seconds for BSP and 12 seconds for spatio-temporal partitioning. Note, the result size in this case did not play an important role as it was only 0 or a few thousand at maximum.

For the `sentinel` dataset the results are similar. Though, due to the different data distribution, the difference between spatial-only and spatio-temporal partitioning is not as high as for the `gdelt` dataset. For the `r4-large` query, the BSP variant performed a bit better, for both, the `gdelt` and `sentinel` datasets, because it created fewer partitions in general that needed scanned and probably because only a spatial comparison had to be performed. Although, the latter only has only a marginal influence on the execution time.

k Nearest Neighbor Search & Skylines Next, we compare the performance of the discussed kNN search approaches. We chose three spatial positions as reference points: a randomly selected point $(39.961727, 84.47858)$ denoted as `random`, a point

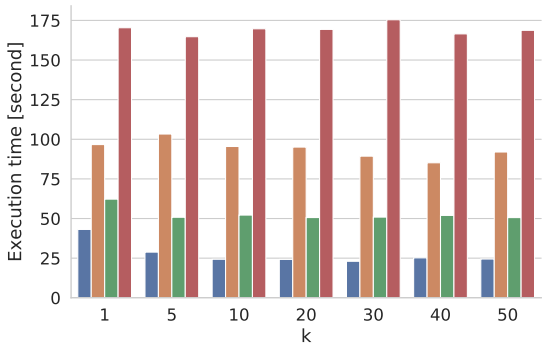
Figure 9.4: Throughput of the three discussed kNN implementations in STARK for reference point `ilm`. The input dataset was cached in memory.

in Ilmenau (50.681815, 10.939779) denoted as `ilm`, and `00` located at (0, 0). We loaded the input data sets, partitioned them using a spatial partitioner if applicable, created partition-local indexes, and cached them in memory. Therefore, the values presented in Fig. 9.4 show the raw performance of the kNN search.

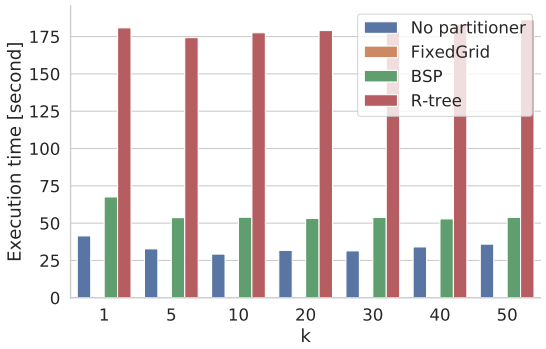
The figure exemplary shows the throughput for the reference point `ilm`. For the other two points the results are similar, with *BoundedKNN* achieving a significantly higher throughput than the other two variants. The high throughput is achieved by reducing the number of partitions to scan: If the reference point lies within a dense region, its k nearest neighbors are likely to also lie within the same partition and the result can be constructed by only processing one partition. This is important if there exist much more partitions than executors, so that workers need to process more than one partition. The *Local* variant is not much better than the *Naïve* implementation. Managing the kNN struct as well as the frequent (de-)serialization required during the merge of the local results incur too much overhead.

We additionally executed the experiment without caching the dataset first. The results are similar to those in the previously described scenario, but with increased execution times, of course.

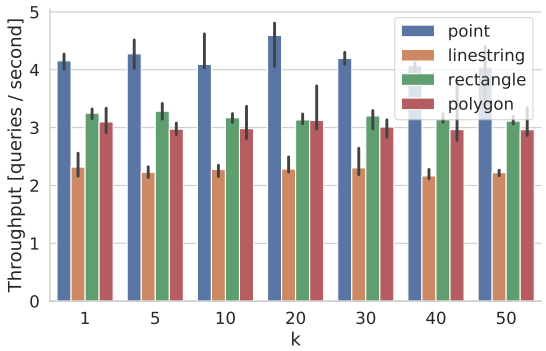
Figures 9.5(a) and 9.5(b) show the execution time of the *BoundedKNN* using different partitioners, without caching the dataset first. Here, we encounter the drawback of the fast *FixedGrid* partitioner: It was not possible to run this experiment for the polygon and rectangle dataset with the *FixedGrid* partitioner (ppd=32), because some partitions contained so many elements that building the index in memory caused the executors to fail as they ran out of memory. Partitioning using the R-



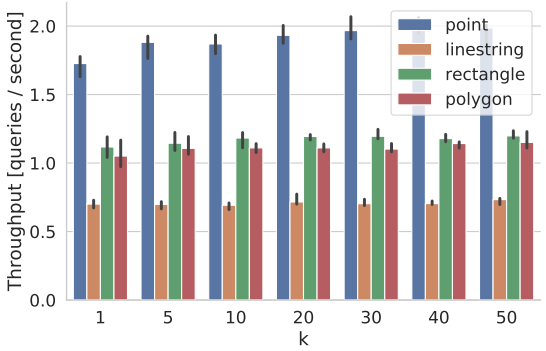
(a) `Linestring` dataset.



(b) `Polygon` dataset.



(c) Throughput for kNN with NDP for `ilm`.



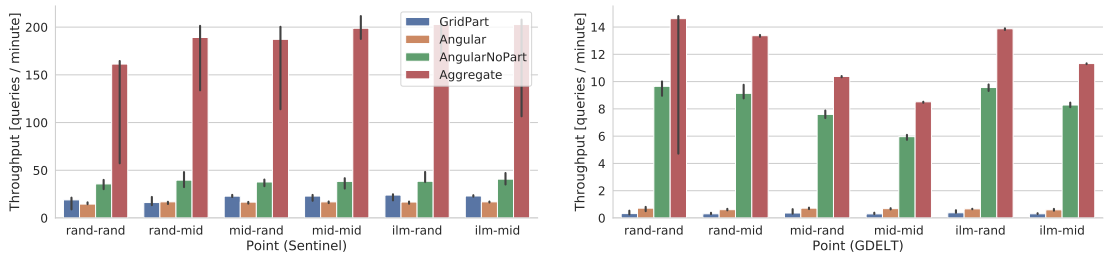
(d) Throughput for kNN with NDP for `random`.

Figure 9.5: Execution time of kNN search using different partitioners on (a) linestring and (b) polygon datasets with reference point `ilm` as well as throughputs for NDP in (c) and (d).

tree partitioner (num=2048) took very long time, because of the partitions being created over a sample of the data. During the assignment of elements to partitions, objects that do not lie within a partition need to be assigned to their nearest partitions, causing extra computational overhead. Additionally, it happened that the *BoundedKNN* did not find the result objects in the current partition and therefore had to scan more partitions than when BSP was used. Overall, the kNN search performs good without a spatial partitioning. The best results with partitioning where achieved by the BSP, which, however, was still two times slower than the kNN search without additional spatial partitioning.

For kNN search, the near data processing can also be applied. A comparison of Figs. 9.5(a) and 9.5(b) with Fig. 9.5(c) shows that even when no partitioning was performed, the kNN search takes at least 25 seconds, while when the logical NDP approach is used, multiple queries per second are possible. For the reference point `random` in Fig. 9.5(d), the kNN search took longer than for `ilm`. The reason might be that the *BoundedKNN* algorithm needed to load more partitions than for point `ilm` in order to construct the final result.

Next, we analyze the performance of the different Skyline algorithms introduced in Section 5.3.2 on the two input data sets we also used for the spatio-temporal range



(a) Skyline results for `sentinel` data set.

(b) Skyline results for `gdelt` data set.

Figure 9.6: Runtime comparison of variants for Skyline computation.

queries above (`gdelt` and `sentinel`). Figure 9.6 shows the runtimes for computing the Skylines for different reference points. Here, we used the three locations from the previous kNN experiment and combined it with two temporal instants to form six reference points. One instant is an arbitrary time[1] called `random` and the other one is the calculated middle point of the `sentinel` data set[2]. These time values do not have a special meaning for this experiment. As the two graphs show, the actual reference points do not have an impact on the execution times. For the Sentinel data, the Skyline computation finishes in maximal 4 seconds, due to the relatively small data set size. Though, the *GridPart* and *Angular* variants are much slower than *Aggregate* and *AngularNoPart*, because the latter two do not require to reshuffle the data. The *Aggregate* method is 1 second faster than *AngularNoPart*. The reason is that we simply exploit the full parallelism without too much complex logic. In *AngularNoPart*, for every element in the current partition it is decided to which logical angular partition it belongs to. Thus, multiple `Skyline` structures need to be kept in memory, making the merging process more expensive. For the larger `gdelt` data set, the difference between *Aggregate* and *AngularNoPart* is still present, but smaller than for `sentinel`. With the increased data size, the drawbacks of repartitioning the data becomes even more visible. While the *Aggregate* method always finishes within 10 seconds, *Angular* is factor 10 slower and requires 100 seconds – even with their optimized pruning.

Join Processing Partitioning has shown to be crucial for joins. Without a spatio-temporal partitioning, the combinations of all partitions from the left and right input have to be created. For each of these combinations, Spark spawns a task which has to be managed. If the data sets to be joined contain 1000 partitions each, this results in a million tasks, which showed to have too much overhead to proceed with the actual work. The join processing in general is very expensive and time consuming, so that for the cluster used for the experiments on we had to reduce the original data sets as follows: 10 million points, 35 million linestrings, 57 million rectangles, and 57 million polygons.

The experiments have shown that a join without spatial or spatio-temporal partitioning results in unfeasible execution times and those runs were canceled after

[1]Saturday, 10 August 1985, 9:53:47 AM GMT; Unix Epoch: 492515627
[2]Sunday, 23 October 2016, 7:16:51 PM GMT; Unix Epoch: 1477250211

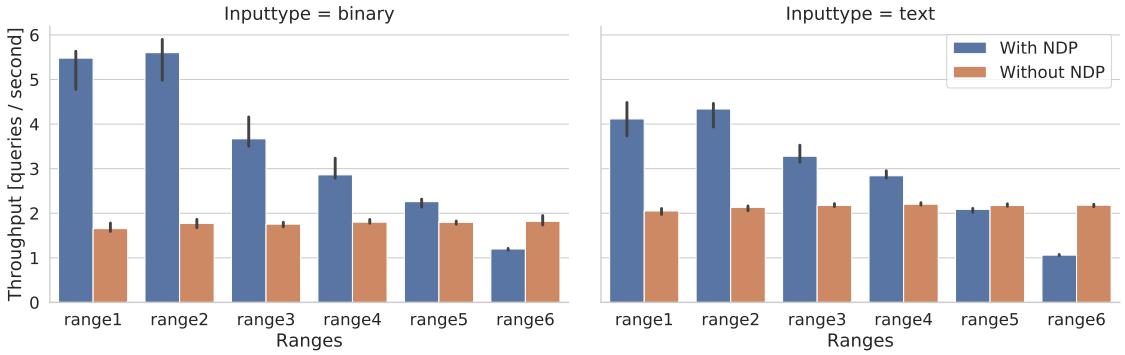


Figure 9.7: Throughput or raster range queries on two input data types.

90 minutes without any progress. Simply reducing the number of partitions is also often not an option as this would increase the number of objects per partition, which might cause problems during index creation.

Raster Data Processing STARK's tile format can be either in plain text or as byte serialized tiles. In Fig. 9.7, the influence of these two formats on the execution times are shown. The figure furthermore shows the influence of pushing down the filter range into the load operation. The dataset was queried with the six query ranges that were also used for the range queries on vector data sets before. It can be seen that without the logical NDP, the input format does not have a significant impact on the throughput. Neither does the query range in this scenario, as we have to load the complete file from storage and do not apply a partitioning which could be used for pruning. Same as for vector data sets, pruning the partitions in the load operator has an enormous effect on the execution time and throughput. Only when the query range is so large that (almost) all partitions need to be loaded, the overhead of identifying matching partitions cannot be amortized anymore. For `range6`, the NDP approach was even slower than the traditional variant. This suggests that an optimizer of system supporting declarative language could estimate a query range's selectivity and use this to decide how to load the data.

With NDP, the binary format showed to achieve a better throughput than when using the textual tile representation, although the binary format occupied twice as much space as the textual format (1.3 GB vs 652 MB). The reason is that the long strings of the text format must be split and parsed to form the tiles, whereas the with the binary format the tiles can directly be deserialized.

9.1.2 Indexing

Similar to repartitioning the data during query evaluation, indexes can be created for a single query as well. Creating the partition-local indexes means to "touch" every element and insert it into the index structure. For filter operations, this is an extra step: instead of directly evaluating the filter predicate, the objects are first inserted into the index, which is then queried.

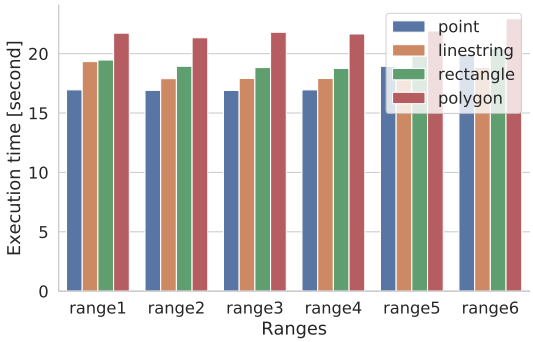


Figure 9.8: Range queries with online index generation (without partitioning).

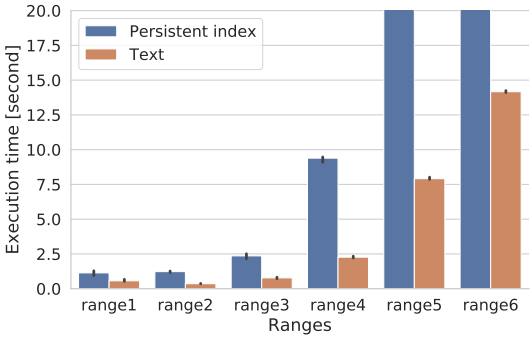


Figure 9.9: Impact of persistent indexes for range queries on point dataset with NDP (in AWS).

Hypothesis 3. *Although the online generation of indexes incurs some overhead, the partition-local indexes significantly improve query execution time. Unlike joins, spatio-temporal filter operations will not benefit from online index generation.*

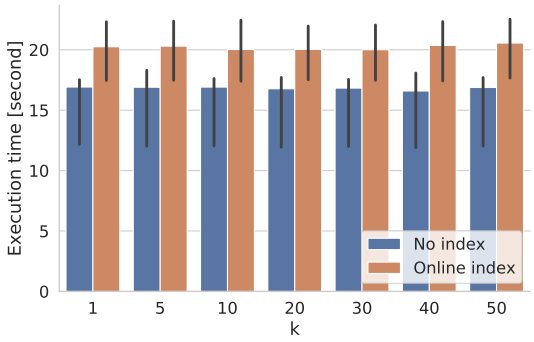
Creating the index online is an overhead which cannot be amortized during evaluating the filter predicate, as Fig. 9.8 shows. Compared to the execution times in Fig. 9.2(a), the execution time for the point data sets was doubled and for other data sets it also increased by a few seconds. As we expected, this means that creating the index online for filtering has a large negative impact and is impractical.

However, if the index is materialized with additional meta information, operators can not only identify partitions to load, but also have an index available instantly. Though, the Java object serialization creates very large objects, even with specialized serialization frameworks, such as the Kryo serializer. Albeit STARK implements specialized serializers for this framework, the created objects are still so large that the time required to read them from HDFS when a hard disk is used exceeds every other runtime: The `point` data set used in the experiments uses only 4 GB to store the 200 million points, whereas the binary representation of the same dataset with partition-local indexes occupies 43 GB. Reading these large data sets caused extremely long execution times in our HDD-based cluster.

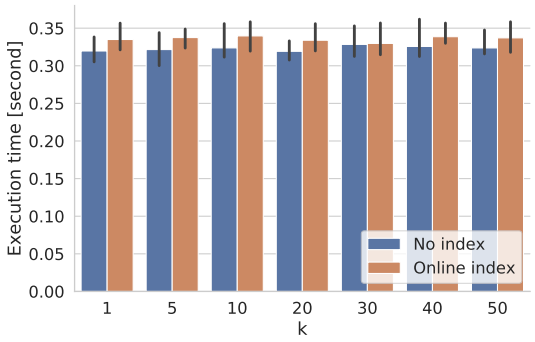
Thus, we repeated the experiment in a Hadoop cluster with SSDs attached to the nodes. For this, we created an Amazon EC2 Elastic MapReduce cluster (emr-5.27.0 with Amazon Hadoop 2.8.5 and Apache Spark 2.4.4). This cluster consists of 5 worker nodes (`m5.2xlarge`), each with a 100 GB general purpose SSD and one master node (`m5.xlarge`). The original datasets were loaded from S3, partitioned and indexed, and finally materialized into the cluster-local HDFS.

We executed the range queries on these data sets and report their performance in Fig. 9.9. For `range1`, the persistent index variant is already more than 0.5 seconds slower than the plain text variant without indexes. The more partitions are loaded, the more visible becomes the impact of the large binary objects of the indexes. For `range5` and `range6` execution took around 60 and 80 seconds, respectively.

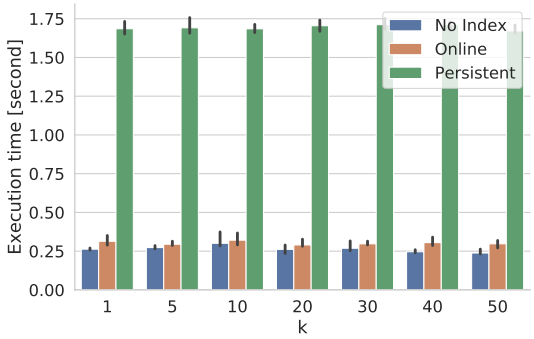
A similar behavior can be observed for the kNN search. The benefit that the index brings is annihilated by the cost for online index creation. The experiments without a spatial partitioning have shown that when an index was created and used,



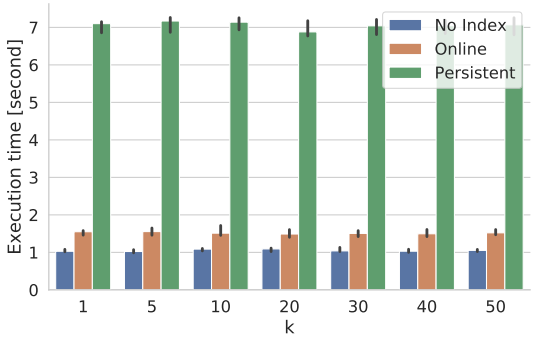
(a) Overhead of indexing for kNN search without prior spatial partitioning.



(b) Overhead of indexing for kNN search with logical NDP.



(c) kNN search on points (in AWS).



(d) kNN search on rectangles (in AWS).

Figure 9.10: Overhead of indexes on kNN search.

it was around 5 seconds slower than when no index was used (Fig. 9.10(a), average over all datasets for point `ilm`).

When the kNN operation was pushed into the load function and an index was created online, it was minimally slower than when no index was used (Fig. 9.10(b), average over all datasets for point `ilm`). Thus, in this setting the overhead for index creation was not amortized during query evaluation. However, in clusters with more powerful machines, larger partitions are possible. With the larger partitions, more items would have to be scanned when no index is used. Hence, in such settings, online index creation might be beneficial.

Hypothesis 4. *Since the kNN search is more complex than plain filters and only very few partitions need to be loaded with the logical NDP approach, using an already created and persistent index results in better performance than using no index or computing it online.*

In order to test this hypothesis, we additionally ran the kNN search in the same AWS cluster as for the range queries. The results for the `point` and `rectangle` datasets and reference point `ilm` are shown in Figs. 9.10(c) and 9.10(d). The results clearly refute this hypothesis. The large byte serialized indexes still require too much time to read and deserialize, making this variant significantly slower than a kNN search with online index generation. In all four cases in Fig. 9.10 computing the kNN is fastest when no index is used.

Hypothesis 5. *The expensive spatial/spatio-temporal join operation must be performed with partition-local indexes on large data sets in order to achieve reasonable execution times.*

In contrast to kNN and filters, in joins the index is not only queried once, but for all m elements from the second relation. Thus, instead of performing a nested loop join in $O(n \times m)$, the join can be performed in $O(m \times log_M(n))$ with n elements in the left (indexed) input and M being the maximum capacity of a tree node. To show the impact, we exemplary performed a self join of a data set with 10 million points. Before the join, the partitioned dataset was cached into memory. With online index creation, the join took ca. 5.1 seconds, whereas without using any index, the join timed out after our 30 minutes limit.

We also investigated the impact of the persistent indexing on join query performance on the example of the `point`–`point` and `point`–`linestring` combinations using the AWS cluster.

When no index was available, we loaded the raw text files and created the index online. In the variant with persistent index, we loaded the indexed data for one relation and used the textual data for the other one. For the `point`–`point` combination, the join takes around 12 seconds in both cases. Only when the non-indexed relation was loaded as binary data, the execution time slightly increased to ca. 16 seconds. For the `point`–`linestring` combination the index exists on the `linestring`s and we found a clear advantage of the peristent index: In the variant with online index generation, the join takes around 68 seconds, while with the persistent index, the time was decreased to 44 seconds (or 48 when the points where loaded as binary objects). Thus, there is a benefit gained from the index showing that the persistent index should be used in joins.

9.1.3 Conclusion

Spatial and spatio-temporal partitioning can be extremely important to achieve reasonable query response times. Especially for interactive applications, for example in notebook systems like Jupyter, fast response times are mandatory. Often, the input files are just present as raw text files that are loaded and processed again and again. For spatial filters, repartitioning the data is not an option due to the comparably high shuffling costs in Apache Spark. Also, creating indexes online during query evaluation incurs too much overhead for the quite cheap evaluation of a spatial range query and kNN search.

In order to further speed up query execution, the partitioning and indexing can be computed once and information about the partitions can be materialized as meta data. When this available information is used to decide which partitions of the data set actually contain result candidates, the query execution time can be reduced by a factor up to 10 – if the number of partitions to load is significantly smaller than the overall number of partitions.

For joins, however, spatial (or spatio-temporal) partitioning must be applied prior to the operation as otherwise all partitions combinations must be evaluated.

Furthermore, an index must be available in the join as the partition-local join evaluation would otherwise be computed using a nested loop, which results in unac-

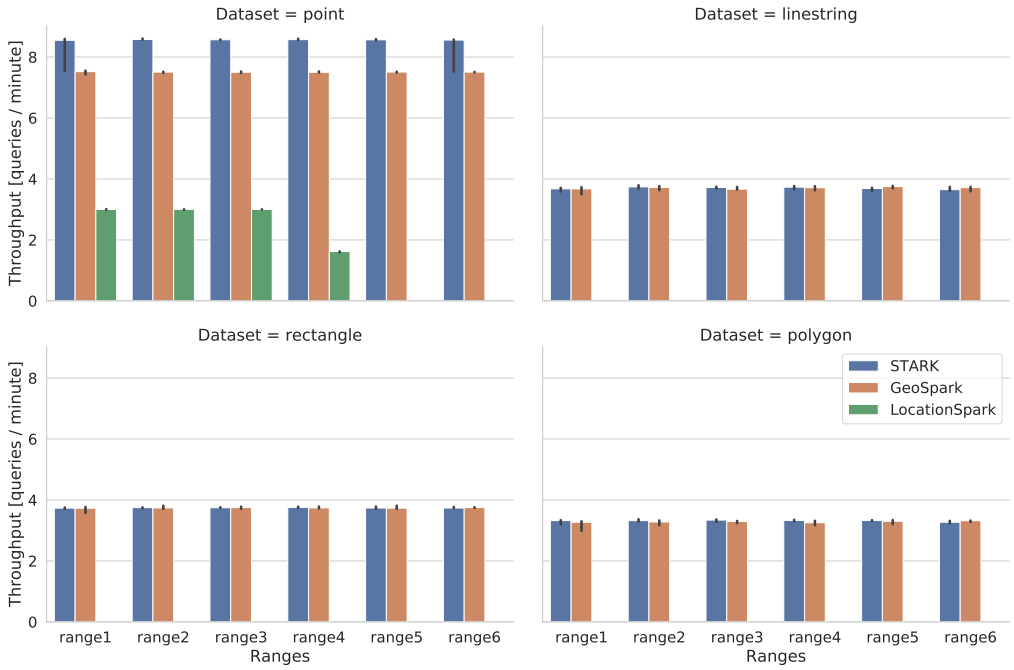


Figure 9.11: Throughput for range query with six query ranges.

ceptable runtimes (experiments were set to time out after 30 minutes in this case). With increased data size, i. e., number of objects, it showed that persistently stored indexes help to improve the join performance. However, the large binary Java objects have a clear overhead for reading, compared to textual data.

9.2 Comparing STARK with Related Systems

In this section we compare STARK with three other Spark-based spatial data processing frameworks. We chose GeoSpark[3] (version 1.2.0) and LocationSpark (latest version on GitHub[4]) as they both showed best results in the benchmark performed in [79]. Additionally, we compare the performance for raster–vector operations of STARK with RasterFrames[5] (version 0.7.0).

We show that STARK can outperform all three systems by exploiting its global and local indexes.

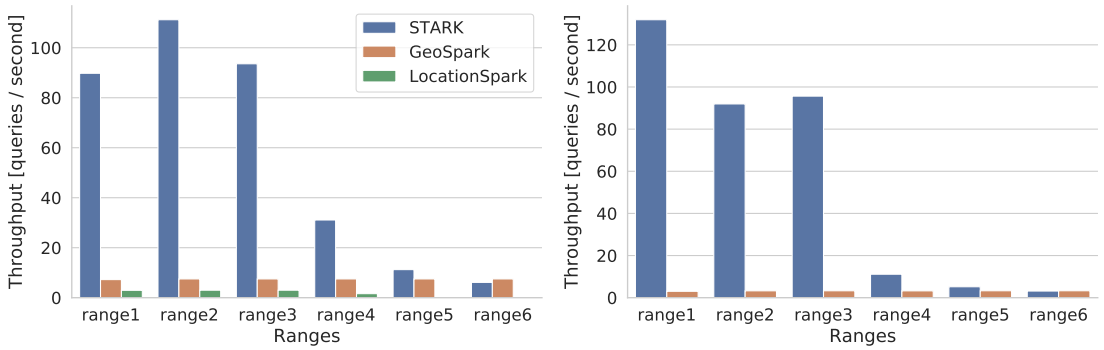
9.2.1 Spatial Range Queries

In Fig. 9.11 we report the throughput for range queries using the six ranges from the previous experiments on the four data sets. For this setting we did not use a partitioner and no indexing method. It can be seen that for points STARK and GeoSpark perform much better than LocationSpark, which always repartitions data

[3] `https://github.com/DataSystemsLab/GeoSpark`

[4] `https://github.com/purduedb/LocationSpark` – Latest commit ee90d4b from Jan 6, 2017; had to be compiled with Scala 2.10 and run with Spark 1.6

[5] `https://github.com/locationtech/rasterframes`



(a) Range queries with NDP for `points`.

(b) Range queries with NDP for `polygons`.

Figure 9.12: Throughput for range queries with six query ranges and STARK pushing down filter predicate into load function.

and builds local indexes, which in this case causes too much overhead. STARK constantly performs a bit better than GeoSpark and achieves around one query more per minute. For the other data sets, both frameworks perform equally well. For every query range, the runtimes are almost equal, indicating that the actual filter step is neglectable compared to the time needed to load (and prepare) the input data. Both, STARK and GeoSpark internally use the JTS library and thus, the actual spatial computation is performed by the same library in both systems, resulting in the similar execution times.

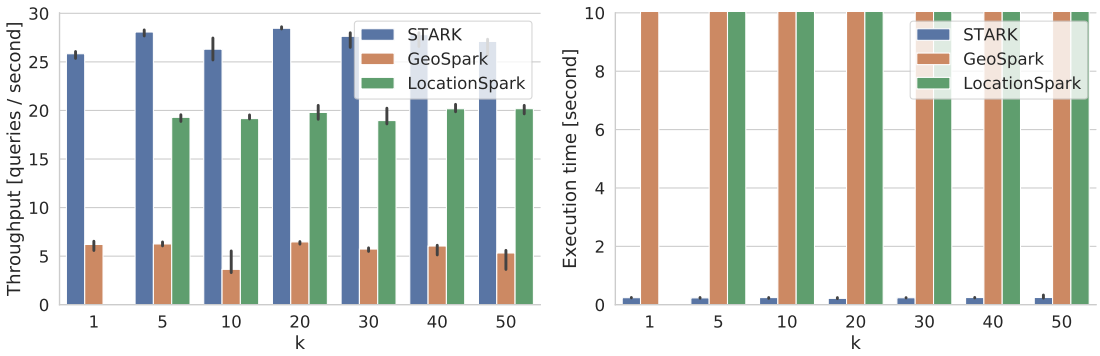
LocationSpark was not able to answer the query for ranges 5, and 6: we constantly received `GC overhead limit exceeded` errors after a few repetitions. This means that there are so many objects created and need to be released by Java's garbage collector to make space for new objects. However, if so many objects are created, the JVM spends more time in the garbage collector than in the actual program logic and, therefore, the process is automatically terminated by Spark. A reason might be that the operator returns the result in a local data structure on the driver instead of keeping it in an RDD. This is a problem as the result size is too large to fit into the memory of the driver. Furthermore, LocationSpark is only able to handle range queries on point data sets.

Since filtering is a common task for spatial data analytics, we additionally compare the performance of the logical NDP in STARK with the other systems. The results in Fig. 9.12 clearly show that STARK outperforms the other systems in this scenario. The execution time is decreased to only 0.5 – 0.8 seconds for ranges 1 – 3, so that the throughput increases to multiple queries per second[6]. This is an improvement of factor 10 compared to the results in Fig. 9.11.

9.2.2 k Nearest Neighbor Search

Next, we compare the results of STARK with the other platforms in Fig. 9.13 for kNN search. The first thing to note is that LocationSpark only supports kNN search on a point dataset. Therefore, here we only show the kNN search results over points.

[6]Recall that data is not cached in memory, but has to be loaded from storage.



(a) Throughput for kNN search on cached data. (b) Execution time for kNN search reading from disk (NDP in STARK).

Figure 9.13: Throughput for cached `point` data set (a) and execution time with logical NDP (b).

STARK and LocationSpark achieve similar results and differ only in 0.1 – 0.2 seconds as they both implement the same approach (*BoundedKNN*). However, for the reference point `ilm`, STARK performs better than LocationSpark. As shown in Appendix A, for the other points LocationSpark is a bit faster, though. Since both systems implement the same approach, this difference results either from external influences (another process disturbing on one of the nodes, network latency during communication between nodes) or minor implementation details. For example, STARK performs some additional checks to test if it has already found the final result to avoid errors like we encountered for LocationSpark. Furthermore, the partition size containing the reference point plays an important role. The more elements it contains, the longer the total execution time of the kNN operator is. Even a this small execution time difference results in a few queries more per second. In the case of the reference point `ilm` shown in Fig. 9.13(a), the datasets were partitioned and cached in memory. Here, STARK is faster than LocationSpark, resulting in more queries per second.

In Fig. 9.13(b) we present the results for the kNN search, when the data is not cached, but has to be loaded from storage. As for the range queries, with the logical NDP approach much fewer partitions, than the data set actually is comprised of, had to be loaded and processed. In Fig. 9.13(b), the y scale ends at 10 seconds, to make the runtime of 0.3 seconds in STARK visible. In these cases, GeoSpark takes around 30 seconds and LocationSpark around 40 seconds.

LocationSpark always assumes that it can construct a query range to find the k nearest neighbors in. In our experiments we encountered a Java exception for k = 1 when LocationSpark tried to pull an element from an empty iterator.

For the sake of completeness, we show the results for both, the STARK internal comparison as well as the comparison with the other platforms for the two other reference points in Appendix A.

Table 9.3: Comparison of execution times when loading data from storage.

Combination	Platform	Execution time [second]
`point–point`	STARK w/ Online Index	12.9
	STARK Persistent Index	12.0
	GeoSpark	36.0
`point–linestring`	STARK w/ Online Index	68.9
	STARK Persistent Index	44.6
	GeoSpark	94.5

9.2.3 Join Processing

The results of spatial joins are shown in Fig. 9.14. Here, we used the same reduced data sets as previously when we studied the impact of partitioning and indexing (10 mio. points, 35 mio. linestrings, and 57 mio. polygons and rectangles.) For this experiment, data was partitioned and cached. The reported times include the time for indexing creation and the actual join processing. Except for the polygon–polygon case, where STARK is slightly slower than GeoSpark, STARK performs significantly better than the other two systems in most cases. As for range queries, LocationSpark solely supports point-rectangle join.

We additionally compare the execution times of the systems when reading data is included. Again, these experiments were executed in the AWS cluster. Since LocationSpark is quite old already, we had to setup a new cluster for this framework with an appropriate Spark Version. We created an emr-4.9.3 cluster with Amazon Hadoop 2.7.3, Spark 1.6.3 using `m4.xlarge` machines[7]. Though, we were not able to produce results for LocationSpark as the process repeatedly crashed with *Slave lost* messages.

Therefore, we can only compare STARK with GeoSpark here and exemplary use the `point–point` and `point–linestring` combinations that we already also used in Section 9.1.2. As before, the index was built on the `linestring` dataset. The results are listed in Table 9.3. Again, STARK is clearly faster than GeoSpark – not only when the index can be loaded from storage.

From a usability point of view, LocationSpark is highly limited as it only supports the *contained-by* predicate on point data sets. GeoSpark is also quite limited as it supports *contains* and *intersects* as predicates only. STARK allows any predicate and is even able to apply user-defined functions as predicates.

9.2.4 Raster Data Processing

For raster data processing, we compare STARK with RasterFrames. For filtering, we used a raster data set with temperature values from the earth in TIFF format for RaterFrames and in our own tiled format for STARK. In Fig. 9.15 the raster–vector filter in STARK and RasterFrames are compared. It shows that with this approach, we can achieve a throughput up to three times as high as in RasterFrames.

[7]The `m5` machines where not available in this configuration.

Figure 9.14: Throughput for spatial joins using *contained-by* predicate.

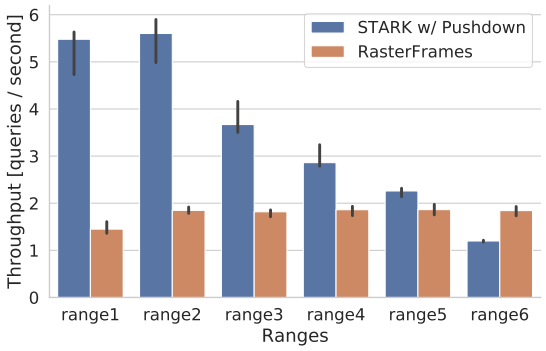


Figure 9.15: Comparison raster–vector filter in STARK and RasterFrames.

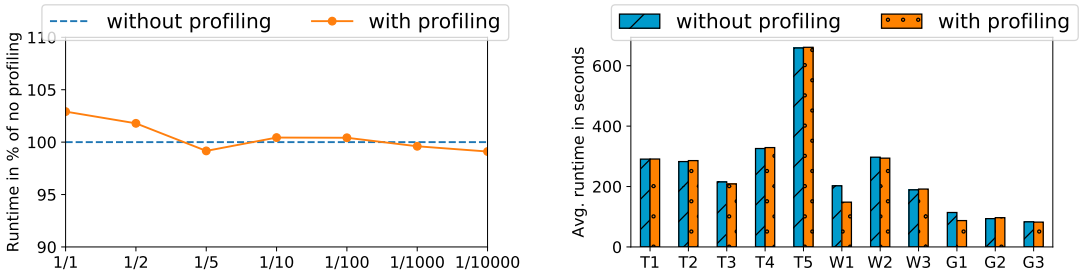
Joining a raster data set with a vector data set is not supported in RasterFrames or GeoTrellis, i. e., it cannot even be expressed. Thus, we can only report the measurements for STARK here: we joined the temperature data set with a data set containing the borders of 224 countries (polygons) in the world. Because of the partitioning and indexing opportunities in STARK, on average this join was completed in only 24 seconds, whereas without the spatial partitioning it degenerates to a Cartesian product and we had to terminate this experiment after several minutes.

9.2.5 Conclusion

As one result of this thesis we developed the STARK framework to efficiently process spatial and spatio-temporal vector as well as raster data. In this part of the evaluation, we have shown that STARK can outperform other platforms by making use of an existing global index. This index is used in combination with query ranges or, in case of kNN, reference points to reduce the amount of data to read from storage and process at the worker nodes. However, STARK performs better than the competing systems not only when using this NDP approach. The efficient implementation of operators, indexes, and the possibility of persistent partition-local indexes make STARK superior in nearly all executed experiments.

LocationSpark implements a fast indexing and partitioning strategy which works well especially when data is already cached in memory. However, when data is loaded for filter or kNN search, the mandatory repartitioning and indexing incurs an unnecessary overhead. GeoSpark performs well for range queries and joins, but implements only a naïve kNN search strategy.

Currently, the logical near data processing works only partially for joins, as it would require to load a file, that represents a spatio-temporal partition in STARK, also as exactly one partition in Apache Spark. This is not possible yet and Spark automatically decides to merge (parts of) files into one partition during load. The resulting partitions, however, violate the spatio-temporal bounds stored in the global index. Thus, in order to also apply this approach for joins, the native load functions of Apache Spark and Hadoop would need to be modified. This is left as optimizations in future work on this framework.



(a) Varying sample rates.

(b) Overhead per script

Figure 9.16: Overhead of code instrumentation.

9.3 Reusing Intermediate Results

In this part of the evaluation we demonstrate the applicability and impact of the transparent materialization approach. We use real world data from three use case scenarios: `weather` contains sensor data from the SRBench [114] benchmark for hurricane Katrina, `taxi` stores taxi trips in New York data[8] (Yellow Cabs, 2013 – 2016) and is used in combination with New York `block` data[9]. We also use the `gdelt` data from 2013 to 2016. We created several scripts for each use case scenario: `T1` – `T5` for `taxi` scenario, `W1` – `W3` for `weather` scenario, and `G1` – `G3` for `gdelt`[10].

Hypothesis 6. *The code is instrumented with profiling code to gather operator statistics. The injected profiling code only introduces a negligible overhead and does not influence the actual operator execution.*

In Fig. 9.16(a), the influence of the sample fraction is shown. It can be seen that profiling incurs only a small overhead for a sample rate of 1/1 and 1/2 (meaning 100% or 50% respectively are selected) and runtimes only differ in less than 10 seconds or 5%. Thus, for our other experiments we selected a sample rate of 10%. Figure 9.16(b) shows the execution time without profiling as well as with code instrumentation for profiling for each of our test scripts. There are almost no runtime differences in execution time.

The decision model is implemented as part of the optimizer. The optimizer as well as the decision model cannot guarantee to always achieve the optimal result, due to changes in the current overall cluster workload and other parameter changes. Thus, it is important that the model and optimizer do not make bad decisions and materialize wrong intermediate results where recycling them would increase the execution time.

Hypothesis 7. *The materialization decision always improves query execution time and avoids bad decisions.*

Fig. 9.17 shows runtimes for each script for three executions. Prior to the first execution no statistics were available and thus are collected during this execution

[8] `http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml`

[9] `http://www1.nyc.gov/site/planning/data-maps/open-data.page`

[10] The scripts can be found in Piglet's GitHub repository.

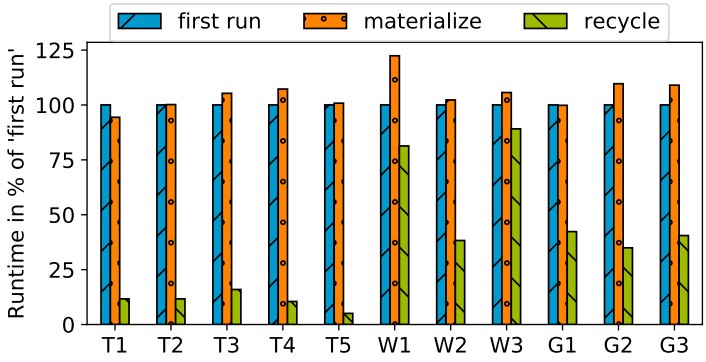


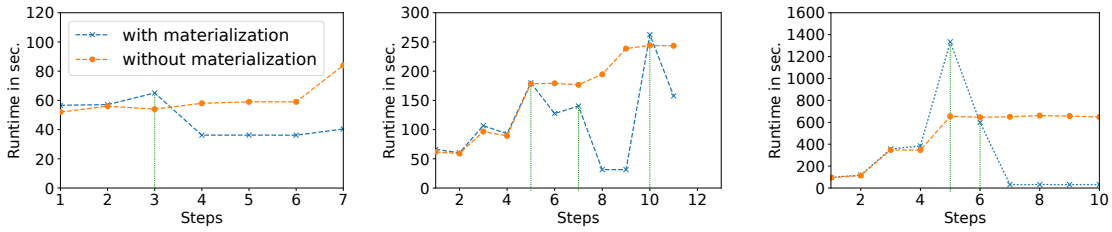
Figure 9.17: Three execution times for each script showing the benefit of loading materialized results. Runtime in percent of the first execution.

(blue bars). In the second execution, previously generated statistics were used to decide which operator to materialize (orange bars). Hence, this execution includes also the writing of the intermediate result. For the last execution, the materialized results were loaded (green bars) and thereby reduced the overall execution time of the job.

As we argued in previous sections, scripts are often developed incrementally. In this experiment we show the runtime differences for one script of each scenario for incremental execution.[11]. (Fig. 9.18). We first ran the according script without materialization support (dashed orange lines) and compared the runtimes to an execution with materialization enabled (dotted blue lines). On the x-axis, step 0 is the initial execution with a `LOAD` and a first `FILTER` operator and subsequent steps add one or more operators. For the first few steps, both execution times are equal except for some minimal discrepancies. At some point however, the optimizer recognizes a materialization point for which a benefit will be achieved and writes the respective result to disk (indicated by vertical lines). In this step, the execution time for *with materialization* rises above the reference time *without materialization*. However, since this is executed only once, the additional costs (clearly visible in Fig. 9.18(c)) will easily be amortized in subsequent executions that benefit from loading the materialized data. In fact, materialization reduced the cumulative execution time for `G1` from 7 to 5 minutes, for `W2` from 30 to 20 minutes, and for `T5` from 80 to 30 minutes.

In our experiments we saw that the estimated benefits often were close to real measured speedups, but sometimes also deviated to some extent. We observed that in almost all cases when a benefit could be achieved, we underestimated it – meaning that a subsequent execution was even shorter than calculated. In our tests we never encountered the situation that the optimizer calculated a benefit for a candidate materialization point which actually did not bring any benefit during execution. From this we conclude that our cost model calculates executions costs well enough to select an appropriate materialization point and avoid candidate materialization points

[11]During the initial development of the scripts that we used in our evaluation, we already greatly benefited from the materialization and saved much time by reusing the stored results.



(a) Script G1 (b) Script W2 (c) Script T5

Figure 9.18: Incremental execution of one script for each scenario (different y scales).

that would cause longer execution times. However, if the parent partitions could not be determined precisely the computation of the respective operator's execution time may assume a shorter or longer time. Writing always the last materialization point will only bring benefits reliably in scenarios similar to the incremental development of scripts.

Hypothesis 8. *Scripts are created and developed and executed in various scenarios. The best strategy for selecting a materialization point for writing depends on the workload setting.*

To test the impact of the selected strategy on the performance, we looked at two cases: First, we used some additional scripts that all share the first six operators and then diverge into their individual paths that all contain another five operations. Strategy *last* did not materialize data as the last candidate materialization point of a job will not be repeated and thus no job benefited from recycling and execution of all scripts took around 420 seconds (7 minutes). For strategies *maxbenefit* and *markov* intermediate results were recycled and execution time was around 160 secs (2:40 minutes) for both when results were loaded. In the second case we disassembled the jobs from our three use case scenarios into a total of 132 jobs, executed them one after the other in a random sort order.For strategy *last* the overall execution time was 3:47 hours, while for *maxbenefit* and *markov* the total time was 2:52 hours and 2:48 hours, respectively. This shows that a selection strategy that takes the costs of operators into account achieves good results. In this setting, *maxbenefit* and *markov* created similar results, but *markov* strategy performed slightly better in this last experiment as it sometimes chose other materialization point than *maxbenefit*, which more subsequent jobs could recycle.

Chapter 10
Conclusion & Open Research Questions

10.1 Conclusion

In the era of Big Data, all kind of information is gathered and stored in order to analyze and use it to create new knowledge with the ultimate goal to solve research questions as well as to improve service quality and business strategies. Often, particularly spatial and spatio-temporal data is valuable and is generated, collected, and analyzed in various application scenarios. One example are location-based services retrieving location data from millions of users and provide service information such as points of interest or nearby friends and events. Besides location data with and without temporal information, raster data is another type of spatial data that requires special handling in data management systems. In order to analyze the large amounts of collected data, the Hadoop MapReduce and later the Apache Spark platform have been established. The inherent data parallel execution of programs allows to easily process terabytes of data. However, these platforms do not have native support for spatial data types and operations, resulting in inefficient jobs where, e. g., joins degrade to cross products. Furthermore, when a group of scientists wants to analyze the data sets, they often all run the same programs, or at least parts of them. This results in the same operations being executed over and over again in the cluster, wasting valuable computing resources that could be used otherwise, or in the case of rented clusters, could be switched off to save money.

In this work, we analyzed the requirements for efficient processing of spatial and spatio-temporal vector and raster data in data parallel cluster environments. We developed the STARK framework that tries to solve the identified challenges and discussed its main components in this thesis. STARK is a framework that integrates into Apache Spark without modifying its source code. Figure 10.1 shows the main contributions of this work categorized in different components.

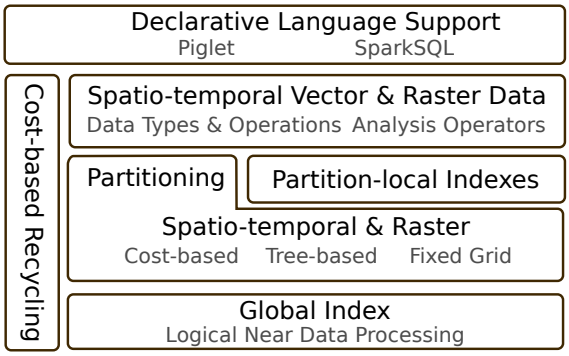


Figure 10.1: Main components and contributions of this thesis.

Declarative Languages: As interface, we provide a Scala-based DSL as well as basic integration into declarative languages. The supported languages currently are SparkSQL and our Pig Latin dialect Piglet. The integration into SparkSQL is achieved by using user-defined types and functions as well as custom rewriter strategies injected into the Catalyst optimizer. For Piglet, types as well as operations thereon are natively added to the language and transpiler.

Raster & Vector Data Types: The spatio-temporal vector and spatial raster objects are represented using dedicated types. To accommodate for the distributed processing in a cluster, for raster data the tile type carries meta information about the spatial location of the tile in addition to the actual pixel values. This way, no centralized catalog is needed. We furthermore augment the tiles with *small materialized aggregates* used during value-based query processing to quickly decide if a tile contains result candidates.

Spatio-temporal Operations: We discussed several possible strategies for kNN search and Skyline computation in the context of data parallel execution and analyzed their performance in the evaluation in Chapter 9. We further investigated how these operators can benefit from the available partitioning and indexing information. STARK implements several operators for processing vector and raster data. Using filter operations, users can find subsets of the input data set matching a spatial or spatio-temporal range according to a given predicate. Besides filtering a vector data set with another vector object, users can additionally use vector objects to filter for certain parts of a raster data set. Joins are supported between two sets of vector objects, two raster data sets as well as between a raster and a vector data set. In addition to the filter and joins, STARK supports three other types of operators: a kNN filter, a Skyline operator as well as a DBSCAN operator for spatio-temporal clustering.

Partitioning & Indexes: To distribute work to executor nodes, Apache Spark employs partitioners that partition the input data set based on certain criteria. We developed spatial as well as spatio-temporal partitioners to support query execution: The operators exploit the spatial and temporal extent of the generated partitions to decide which partitions can contain filter results or join partners, respectively. The extent information is additionally used to find the starting point for the kNN search or dominance regions for Skyline computation. In STARK, the partition information can be materialized and used as a global index. This global index is used by the load function to identify partitions with result candidates for a query. Besides the global index, partition-local indexes can be created to speedup the processing of a partition on a worker node. We investigated when and how operators benefit from the available partitioning and indexing information and found that especially the logical near data processing based on the global index has a huge positive effect on query performance. Furthermore, we analyzed how persistent indexes can be used to speed up processing. However, except for joins where indexing has shown to be mandatory, the relatively large binary index objects have a negative impact on execution time.

Cost-based Recycling: Typically, multiple users work in a cluster accessing the same files or run programs repeatedly. In order to save valuable cluster resources, in this work we developed a cost-based decision model to materialize intermediate results of jobs in the cluster. The model is based on the execution time of operators as well as their result size. The goal of the decision model is to maximize the benefit of the materialization. In this context, the benefit is the time saved when reusing a previously materialized intermediate result compared to the time needed to compute that result again. We implemented the decision model into our Piglet project and instrumented the executed jobs to gather the required parameters during execution. In Section 9.3 we have shown that the model works and helps reduce job execution time in various scenarios.

10.2 Open Research Questions

The STARK framework as well as the decision model for reusing intermediate results in a job can already be useful in many scenarios. However, we believe that this thesis can be used as a basis for further research. In the following, we will discuss four possible directions that extend the contributions made in this thesis.

Near Data Processing and Hardware Support The near data processing as discussed and implemented for now in STARK is performed in the load operator itself. When a filter condition is provided to the load function and the global index for the spatio-temporal partitioning is present, first the index is loaded and candidate partitions are identified. After that, the identified partitions are loaded and returned to the processing pipeline.

While this has shown good results for filter and kNN operations, this processing could still be pushed deeper. On the one hand, since every partition is stored in a separate file in the Hadoop platform, the global index for the partition could be added on file system level. Here, file and directory attributes could be exploited to store the meta information (MBR and enclosing interval, SMAs of tiles in partition). During read, the candidate partitions can be identified reading the file attributes.

On the other hand, while the processing on file system level might result in some performance gain compared to the current solution, a more interesting question is the support of specialized hardware. Currently, in Hadoop and Spark a cluster is assumed to consist only of commodity hardware, i. e., a CPU, disk, and some RAM. However, [106] and [78] already have shown that specialized hardware, such as FPGAs, can be used to reduce the amount of data significantly before it is transferred to the CPU. Furthermore, a system could utilize the parallel computing power of GPUs to perform spatial calculations and comparisons.

To be able to utilize such hardware in a transparent way, the queries need to be analyzed and it has to be decided which parts of the query should be executed on which device. Not only does this decision have to deal with the general capabilities of the devices, but also with the costs for transferring the query as well as the data objects to the GPU.

Handling Uncertain & Imprecise Data Spatial and temporal data is produced mainly in two fashions: by sensors like GPS devices that calculate the current position or as mentions of location names and date/time periods in texts. Sensors may not always function 100% accurately and often include some measurement errors. The accuracy for GPS devices depends on the number of reachable satellites and is usually a few meters for non-military usage.

Not only sensors produce inaccurate values. Information extracted from texts like news articles or historical scripts is often imprecise, too. Mentions of locations and dates often depend on the context of that article and therefore not always refer to an exact location or time. The report *"Metallica played in Germany in 2018"* might be sufficient for the purpose of some general information about a tour of the band Metallica. However, the description of the location is imprecise as it does not state in which city (actually, cities) and at which time they played. Such imprecise information can often be found in articles about historic events as the exact location and time is simply not known.

Algorithms to extract event information with the location and time from texts extract these imprecise definitions and convert them into structured information, e. g., the polygon definition of Germany and an interval from January 1st, 2018 to December 31st, 2018. For technical devices, their inaccuracy is often known or can be estimated from the surrounding conditions.

The imprecise definitions lead to uncertainties in the computations based on this data. Thus, there is not a single location or instant where an object is located or an event took place, but many of them. Additionally, an object/event occurred at these possible locations and instant with some probability.

A research question is how to handle the uncertainties and imprecise information as this impacts the operators and distance functions used within the operators. In preliminary work [50], we discussed the data representation for imprecise events. In future work, new distance functions as well as interpretations for, e. g., Skylines can be developed that account for the imprecise and uncertain data. Here, the challenge is to define the semantics of the dominance relationship. If it is not known where exactly a point is located, how can we decide if it dominates another point, whose location is also not known precisely.

Query Containment Currently, reusing intermediate results works only for exactly the same operators, i. e., they must have the same filter conditions constants, the same projected columns, etc. While this works well if the same script (or parts of it) are executed repeatedly, it reduces the number of times intermediate results can be reused by other scripts even if only slight changes are made.

To increase the applicability of materialized results for reuse in other scripts, the equivalence check should be replaced by a containment check: If the result of an operator o in the current script is a subset of an already materialized result m on disk, it might be beneficial to read m from storage and use it as input for o.

As an example, consider two scripts/queries that load the same input file, e. g., position information sent by mobile devices and that are executed one after the other. Both scripts may perform some pre-processing: the first script only needs to process data with a timestamp before 13.09.2018, while the second script needs data

only before 01.06.2018. After this filter, both scripts perform the same complex and long running aggregation steps.

Since both scripts have the same operations until this timestamp-based filtering, the result of the second script is a subset of the result of the first script. Yet, the second script has to perform all the expensive operations again on the large input dataset. Due to the fact that the lineage ID is based on the lineage ID of the parent operator(s), the lineage IDs of the expensive operations differ in both scripts. However, the result of the filter in the second script is a subset of the result of the timestamp filter in the first script. Therefore, the materialized results of the first script can be treated as candidates for the second script to load as well. As a prerequisite, it must be decidable that the result of o is indeed a subset of the materialized data on disk.

In addition to the containment check of filter results, projection plays an important role: usually optimizers will try to execute projections as early as possible to reduce the amount of data. Unfortunately, this also has a negative impact on the possibilities of reuse since the according column might have been removed. Thus, when implementing a containment check component, a trade-off has to be made between early execution of projections and opportunities for reuse.

In general, a system using containment checks rather than exact equality greatly increases the potential of a materialized result being reusable by another script and therefore, improves the query response times of more jobs.

Cache strategy The transparent materialization of intermediate results bears great potential for shared data centers executing thousands of queries [62, 84].

In Section 8.3.1 we discussed three questions needed to be answered to decide, which candidate materialization point to actually store. One question was regarding the cache size and which previously materialized result should be removed in order to add the new one. As an answer, we proposed to use the well known LRU or LFU algorithms or, to account also for the benefit, a Knapsack strategy.

As an additional improvement, new replacement strategies can be created that account for both, the number of accesses (reuses) to a materialized result as well as the benefit. Here, we envision a strategy inspired by ARC [68], maintaining two lists: one that contains all not yet reused materialized objects managed by LRU and a second list with materialization points that were accessed at least, e. g., one time. The lengths of these lists as well as the movement from one list to another can follow the general idea of the original ARC.

Furthermore, the cost model as well as the cache replacement strategies could directly include the monetary cost of operators and occupied storage.

Appendices

A More k Nearest Neighbor Search Results

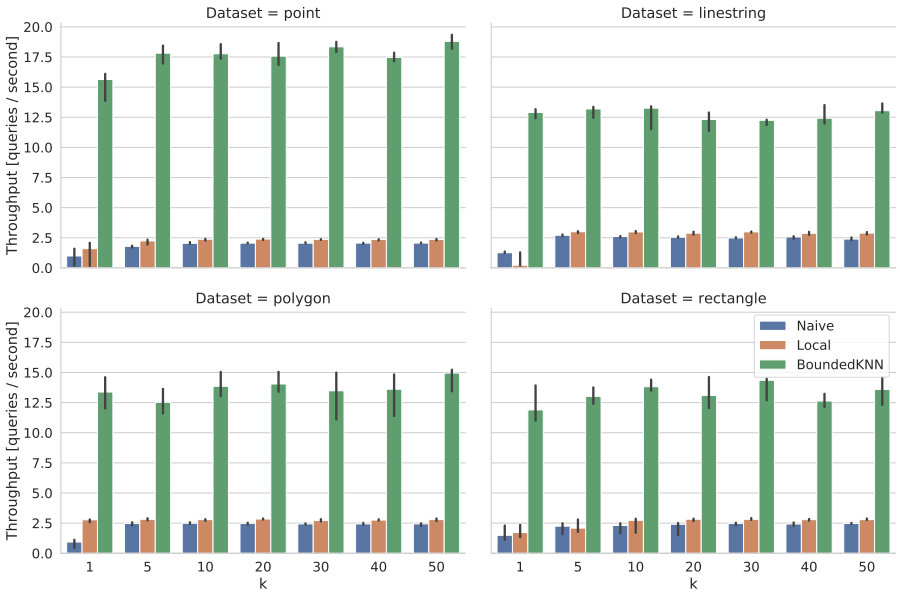
In Section 9.2.2 we evaluated STARK's different kNN implementations and also compared them two the other selected platforms using three different reference points. In the following, the results for the two reference points `random` and `00` are presented.

First, we show the results of the STARK internal comparison in Fig. 2.

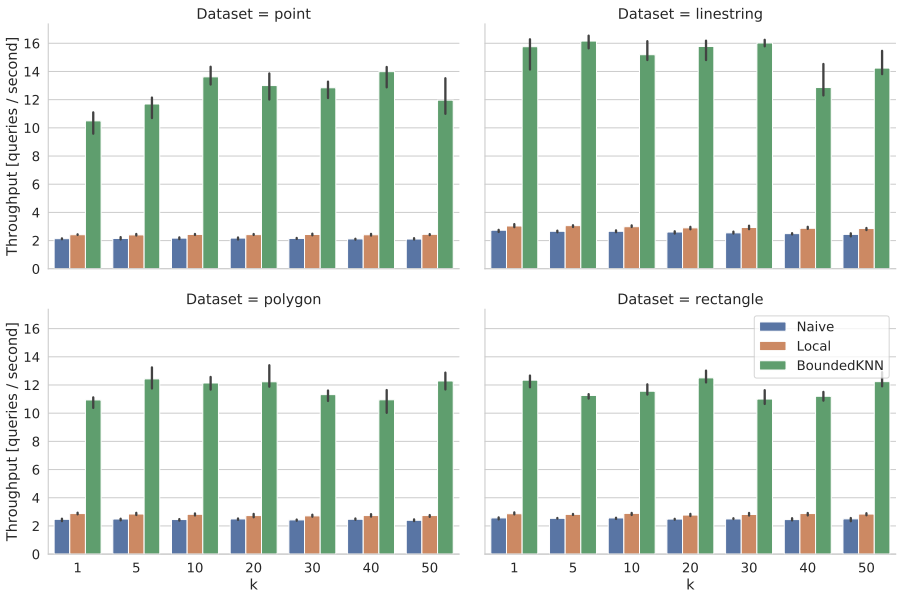
It can be seen that the throughput, and therefore, also the execution time of the operator, is independent from the value k in this case. Although the experiments were repeated several times, still some fluctuations in the measured times were encountered that reveal unexpected execution times in the figures. However, note that these times differences as small as 0.1 second can have a visible impact in these figures.

As we have already shown in the evaluation, the *BoundedKNN* achieves significantly higher throughput than the other algorithms.

Next, we also present the results of the comparison of the selected platforms for the two reference points in Fig. 3 For all three tested reference points, LocationSpark was not able to to find results with $k = 1$. However, unlike for reference point `ilm`, for these two points LocationSpark was faster. As we already mentioned in Section 9.2.2, the reason are most likely implementation differences and the size of the partitions containing the reference point.

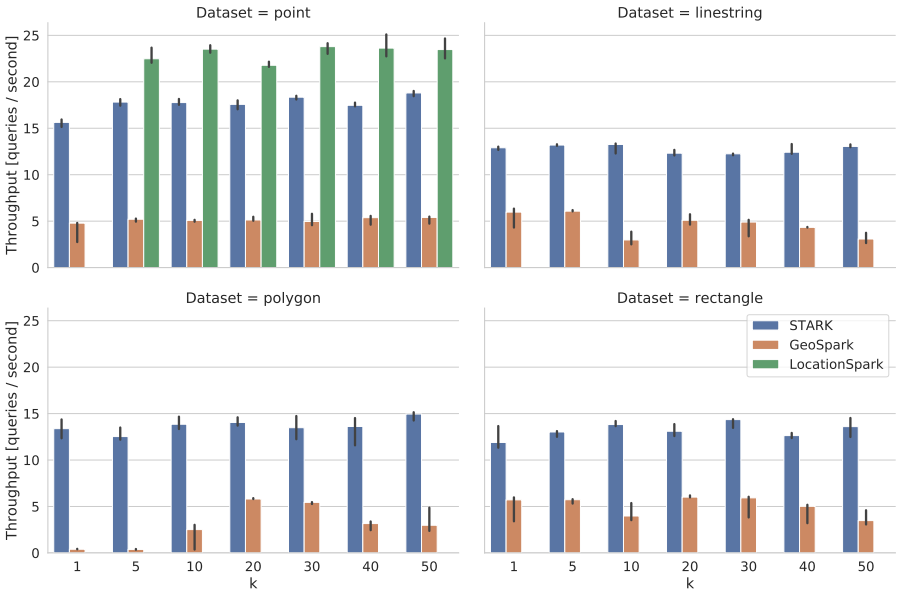


(a) Throughput of the three discussed kNN implementations in STARK for reference point `random`.

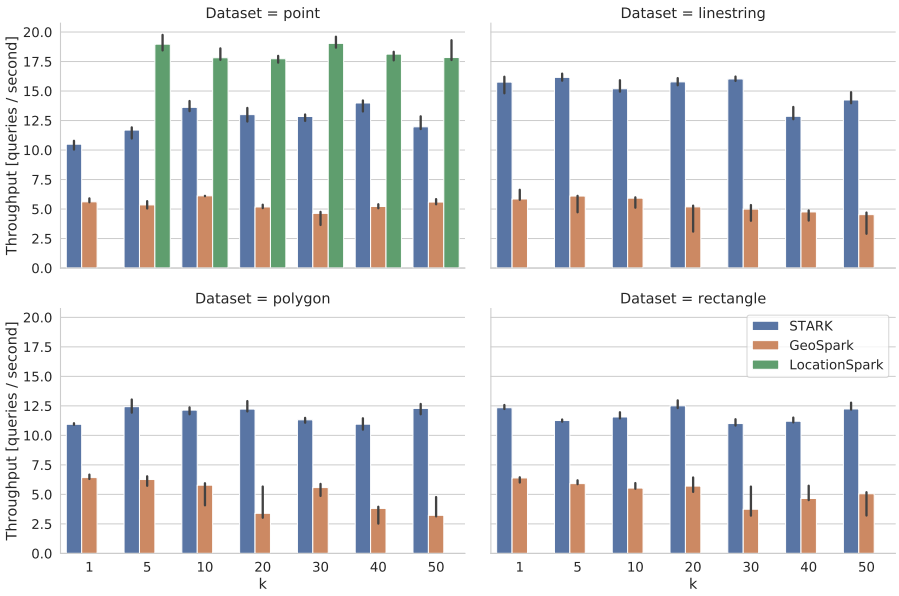


(b) Throughput of the three discussed kNN implementations in STARK for reference point `00`.

Figure 2: STARK internal comparison of kNN variants for reference points `rand` and `00`.



(a) Throughput of the tested systems for reference point `random`.



(b) Throughput of the tested systems for reference point `OO`.

Figure 3: Comparison of the selected platforms for reference points `random` and `OO`.

B More Examples of Piglet Scripts

Listing 4 Script G3

```
<%
def extractDomain(url: String): String = {
  if(!url.startsWith("http"))
    url
  else {
    val startPos = url.indexOf("//")+2
    val endPos = if(url.indexOf("/",startPos) < 0) { url.size } else {
    ↪  url.indexOf("/",startPos) }
    url.substring(startPos, endPos)
  }
}
def diff(d1: Double, d2: Double): Double = {
  math.abs(d1 - d2)
}
def isnum(s: String): Boolean = {
  scala.util.Try {
      s.toDouble
    }.map(_ => true).getOrElse(false)
}
%>
gdelt = LOAD '$gdelt' using PigStorage();
fields = FOREACH gdelt GENERATE $26 as eventcode, (double)$34 as avgtone, $57 as
↪  url;
withURL = FILTER fields BY nonempty(eventcode) and isnum(eventcode) and
↪  nonempty(url)
domain = FOREACH withURL GENERATE extractDomain(url) as site, (int)eventcode as
↪  ecode, avgtone;
grp = GROUP domain BY (site, ecode);
avgtones1 = FOREACH grp GENERATE group as siteecode, avg(domain.avgtone) as
↪  avgtone
avgtones = FILTER avgtones1 BY avgtone != 0
f = FOREACH avgtones GENERATE  siteecode.site as site,siteecode.ecode as code,
↪  avgtone
ordered = ORDER f BY site, code
dump ordered mute;
```

Listing 5 Script T4

```
<%
def dateToMonth(date: String): Int = {
  val formatter = java.time.format.DateTimeFormatter.ofPattern("yyyy-MM-dd
  ↪  HH:mm:ss")
  java.time.LocalDate.parse(date,formatter).getMonthValue()
}
%>

raw = load '$taxi' using PigStorage(',',skipEmpty=true) as (vendor_id:chararray,
↪  pickup_datetime:chararray, dropoff_datetime:chararray,
↪  passenger_count:chararray, trip_distance:chararray,
↪  pickup_longitude:chararray, pickup_latitude:chararray, rate_code:chararray,
↪  store_and_fwd_flag:chararray, dropoff_longitude:chararray,
↪  dropoff_latitude:chararray, payment_type:chararray, fare_amount:chararray,
↪  surcharge:chararray, mta_tax:chararray, tip_amount:chararray,
↪  tolls_amount:chararray,total_amount:chararray);

noHeader = filter raw by not STARTSWITH(lower(vendor_id),"vendor");
month_tip = FOREACH noHeader GENERATE dateToMonth(pickup_datetime) as month:int,
↪  (double)tip_amount as tip

grp = GROUP month_tip by month;
avg = FOREACH grp GENERATE group, AVG(month_tip.tip);
dump avg mute;
```

Listing 6 Script W3

```
<%
def getHourPerDay(s: String): String = {
  val t = s.replaceAll("\"","").split('^')(0)
  val dt = t.substring(0, t.lastIndexOf("-"))
  val ldt = java.time.LocalDateTime.parse(dt)
  s"${ldt.getYear}-${ldt.getMonthValue}-${ldt.getDayOfMonth}-${ldt.getHour}"
}
def getFloatValue(s: String): Double = {
  val t = s.split('^')(0)
  t.replaceAll("\"","").toDouble
}
def hash(s: String): Int = { s.hashCode() }
%>

triples = RDFLOAD('$rdffile')
windObs = BGP_FILTER triples BY {
  ?obs "<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#procedure>"
  ↪ ?sensor.
  ?obs "<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl
  ↪ #observedProperty>"
  ↪ "<http://knoesis.wright.edu/ssw/ont/weather.owl#_WindSpeed>".
  ?obs "<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#result>"
  ↪ ?resultSubj.
  ?obs "<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl #samplingTime>"
  ↪ ?timeSubj
}

obsSensor = FOREACH windObs GENERATE sensor, resultSubj , timeSubj;

triples2 = RDFLOAD('$rdffile')
resultsRaw = FILTER triples2 BY predicate ==
↪ "<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#floatValue>"
resultsWindAll = FOREACH resultsRaw GENERATE subject, getFloatValue(object) as
↪ windspeed:double
results = FILTER resultsWindAll BY windspeed >= 74

triples3 = RDFLOAD('$rdffile')
timesRaw = FILTER triples3 BY predicate ==
↪ "<http://www.w3.org/2006/time#inXSDDateTime>"
times = FOREACH timesRaw GENERATE subject, getHourPerDay(object) as
↪ hour:chararray

resultValues = join obsSensor by resultSubj, results by subject
r1 = FOREACH resultValues GENERATE sensor, timeSubj, windspeed;

timeResults = join r1 by timeSubj, times by subject;
fields = FOREACH timeResults GENERATE hash(CONCAT(sensor,hour)) as sensorHour,
↪ windspeed;
grp = GROUP fields BY sensorHour;
avgWind = FOREACH grp GENERATE group, AVG(fields.windspeed) as speed;
cnt = accumulate avgWind GENERATE COUNT(speed);
cntStr = FOREACH cnt GENERATE "results" as n1, $0;
dump cntStr
```

Bibliography

[1] Serge Abiteboul and Oliver M. Duschka. “Complexity of Answering Queries Using Materialized Views”. In: *SIGMOD*. ACM Press, 1998, pp. 254–263.

[2] Ablimit Aji et al. “Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce”. In: *Proc. VLDB Endow.* 6.11 (2013), pp. 1009–1020.

[3] Ioannis Alagiannis et al. “NoDB: efficient query execution on raw data files”. In: *Commun. ACM* 58.12 (2015), pp. 112–121.

[4] Aldus. *TIFF - Revision 6.0*. Tech. rep. 206. Aldus Corporation, 1992.

[5] Alexander Alexandrov et al. “The Stratosphere platform for big data analytics”. In: *VLDB J.* 23.6 (2014), pp. 939–964.

[6] James F. Allen. “Maintaining Knowledge about Temporal Intervals”. In: *Commun. ACM* 26.11 (1983), pp. 832–843.

[7] Mihael Ankerst et al. “OPTICS: Ordering Points To Identify the Clustering Structure”. In: *SIGMOD Conference*. ACM Press, 1999, pp. 49–60.

[8] Lars Arge et al. “Scalable Sweeping-Based Spatial Join”. In: *VLDB*. 1998, pp. 570–581.

[9] Morton M. Astrahan et al. “System R: Relational Approach to Database Management”. In: *ACM Trans. Database Syst.* 1.2 (1976), pp. 97–137.

[10] Felix Gerhard Balzer. “Entwicklung und Untersuchungen zur 3-D-Nano-positioniertechnik in großen Bewegungsbereichen”. PhD thesis. TU Ilmenau, 2014.

[11] Peter Baumann et al. “The Multidimensional Database System RasDaMan”. In: *SIGMOD Conference*. ACM Press, 1998, pp. 575–577.

[12] Andreas Becher et al. “Integration of FPGAs in Database Management Systems: Challenges and Opportunities”. In: *Datenbank-Spektrum* 18.3 (2018), pp. 145–156.

[13] P. Becker, L. Plesea, and T. Maurer. “Cloud optimized image format and compression”. In: *ISPRS Archives* 40.7W3 (2015), pp. 613–615.

[14] Norbert Beckmann et al. “The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles”. In: *SIGMOD Conference*. ACM Press, 1990, pp. 322–331.

[15] Shaik Abdul Nusrath Begum and K. P. Supreethi. “A Survey on Spatial Indexing”. In: *Journal of Web Development and Web Designing* 3.1 (2018), pp. 1–25.

[16] Derya Birant and Alp Kut. “ST-DBSCAN: An algorithm for clustering spatial-temporal data”. In: *Data Knowl. Eng.* 60.1 (2007), pp. 208–221.

[17] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. “The Skyline Operator”. In: *ICDE*. 2001, pp. 421–430.

[18] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. “Efficient Processing of Spatial Joins Using R-Trees”. In: *SIGMOD*. 1993, pp. 237–246.

[19] Howard Butler et al. “The GeoJSON Format”. In: *RFC* 7946 (2016), pp. 1–28.

[20] Jesús Camacho-Rodríguez et al. *PigReuse: A Reuse-based Optimizer for Pig Latin*. Tech. rep. Inria Saclay, 2016.

[21] Fay Chang et al. “Bigtable: A Distributed Storage System for Structured Data”. In: *ACM Trans. Comput. Syst.* 26.2 (2008), 4:1–4:26.

[22] Liang Chen, Kai Hwang, and Jian Wu. “MapReduce Skyline Query Processing with a New Angular Partitioning Approach”. In: *IPDPS Workshops & PhD Forum*. IEEE Computer Society, 2012, pp. 2262–2270.

[23] Rada Chirkova, Alon Y. Halevy, and Dan Suciu. “A formal perspective on the view selection problem”. In: *VLDB J.* 11.3 (2002), pp. 216–237.

[24] Eliseo Clementini and Paolino Di Felice. “A Model for Representing Topological Relationships between Complex Geometric Features in Spatial Databases”. In: *Inf. Sci.* 90.1-4 (1996), pp. 121–136.

[25] Eliseo Clementini, Jayant Sharma, and Max J. Egenhofer. “Modelling topological spatial relations: Strategies for query processing”. In: *Comput. Graph.* 18.6 (1994), pp. 815–822.

[26] Andrei Costea et al. “VectorH: Taking SQL-on-Hadoop to the Next Level”. In: *SIGMOD Conference*. ACM, 2016, pp. 1105–1117.

[27] Bi-Ru Dai and I-Chang Lin. “Efficient Map/Reduce-Based DBSCAN Algorithm with Optimized Data Partition”. In: *IEEE CLOUD*. IEEE Computer Society, 2012, pp. 59–66.

[28] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters”. In: *Commun. ACM* 51.1 (2008), pp. 107–113.

[29] Peter Deutsch. “DEFLATE Compressed Data Format Specification version 1.3”. In: *RFC* 1951 (1996), pp. 1–17.

[30] Jens-Peter Dittrich and Bernhard Seeger. “Data Redundancy and Duplicate Detection in Spatial Join Processing”. In: *ICDE*. IEEE Computer Society, 2000, pp. 535–546.

[31] Ahmed Eldawy and Mohamed F. Mokbel. “A Demonstration of SpatialHadoop: An Efficient MapReduce Framework for Spatial Data”. In: *Proc. VLDB Endow.* 6.12 (2013), pp. 1230–1233.

[32] Ahmed Eldawy and Mohamed F. Mokbel. “Pigeon: A spatial MapReduce language”. In: *ICDE*. IEEE Computer Society, 2014, pp. 1242–1245.

[33] Ahmed Eldawy and Mohamed F. Mokbel. “SpatialHadoop: A MapReduce framework for spatial data”. In: *ICDE*. IEEE Computer Society, 2015, pp. 1352–1363.

[34] Ahmed Eldawy and Mohamed F. Mokbel. “The Era of Big Spatial Data”. In: *Proc. VLDB Endow.* 10.12 (2017), pp. 1992–1995.

[35] Ahmed Eldawy et al. “Sphinx: Empowering Impala for Efficient Execution of SQL Queries on Big Spatial Data”. In: *SSTD*. Vol. 10411. Lecture Notes in Computer Science. Springer, 2017, pp. 65–83.

[36] Iman Elghandour and Ashraf Aboulnaga. “ReStore: Reusing Results of MapReduce Jobs”. In: *Proc. VLDB Endow.* 5.6 (2012), pp. 586–597.

[37] ESRI. *ESRI Shapefile Technical Description*. Tech. rep. 1998, pp. 370–371.

[38] Martin Ester et al. “A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise”. In: *KDD*. AAAI Press, 1996, pp. 226–231.

[39] Anthony D. Fox et al. “Spatio-temporal indexing in non-relational distributed databases”. In: *BigData*. IEEE Computer Society, 2013, pp. 291–299.

[40] Goetz Graefe. “Volcano - An Extensible and Parallel Query Evaluation System”. In: *IEEE Trans. Knowl. Data Eng.* 6.1 (1994), pp. 120–135.

[41] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. “Cure: An Efficient Clustering Algorithm for Large Databases”. In: *Inf. Syst.* 26.1 (2001), pp. 35–58.

[42] Ralf Hartmut Güting. “An Introduction to Spatial Database Systems”. In: *VLDB J.* 3.4 (1994), pp. 357–399.

[43] Ralf Hartmut Güting. “Geo-Relational Algebra: A Model and Query Language for Geometric Database Systems”. In: *EDBT*. Vol. 303. Lecture Notes in Computer Science. Springer, 1988, pp. 506–527.

[44] Ralf Hartmut Güting, Victor Teixeira de Almeida, and Zhiming Ding. “Modeling and querying moving objects in networks”. In: *VLDB J.* 15.2 (2006), pp. 165–190.

[45] Antonin Guttman. “R-Trees: A Dynamic Index Structure for Spatial Searching”. In: *SIGMOD*. ACM Press, 1984, pp. 47–57.

[46] Stefan Hagedorn, Philipp Götze, and Kai-Uwe Sattler. “Big Spatial Data Processing Frameworks: Feature and Performance Evaluation”. In: *EDBT*. OpenProceedings.org, 2017, pp. 490–493.

[47] Stefan Hagedorn, Katja Hose, and Kai-Uwe Sattler. “SPARQling Pig - Processing Linked Data with Pig Latin”. In: *BTW*. Vol. P-241. LNI. 2015, pp. 279–298.

[48] Stefan Hagedorn and Kai-Uwe Sattler. “Piglet: Interactive and Platform Transparent Analytics for RDF & Dynamic Data”. In: *WWW (Companion Volume)*. ACM, 2016, pp. 187–190.

[49] Stefan Hagedorn, Kai-Uwe Sattler, and Michael Gertz. “A Framework for Scalable Correlation of Spatio-temporal Event Data”. In: *BICOD*. Vol. 9147. Lecture Notes in Computer Science. Springer, 2015, pp. 9–15.

[50] Stefan Hagedorn, Kai-Uwe Sattler, and Michael Gertz. “Large-scale Analysis of Event Data”. In: *GvDB*. 2015, pp. 90–95.

[51] Stefan Hagedorn et al. “Stream processing platforms for analyzing big dynamic data”. In: *it Inf. Technol.* 58.4 (2016), pp. 195–205.

[52] Alon Y. Halevy. “Answering queries using views: A survey”. In: *VLDB J.* 10.4 (2001), pp. 270–294.

[53] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. “Implementing Data Cubes Efficiently”. In: *SIGMOD Conference*. ACM Press, 1996, pp. 205–216.

[54] Tino Hausotte. “Nanopositionier- und Nanomessmaschine”. PhD thesis. TU Ilmenau, 2002.

[55] Yaobin He et al. “MR-DBSCAN: a scalable MapReduce-based DBSCAN algorithm for heavily skewed data”. In: *Frontiers Comput. Sci.* 8.1 (2014), pp. 83–99.

[56] Xiaojuan Hu et al. “A MapReduce-based improvement algorithm for DBSCAN”. In: *Journal of Algorithms & Computational Technology* 12.1 (2018), pp. 53–61.

[57] Chao-Qiang Huang et al. “RDDShare: Reusing Results of Spark RDD”. In: *DSC*. IEEE Computer Society, 2016, pp. 370–375.

[58] Stratos Idreos et al. “Merging What’s Cracked, Cracking What’s Merged: Adaptive Indexing in Main-Memory Column-Stores”. In: *Proc. VLDB Endow.* 4.9 (2011), pp. 585–597.

[59] *Information Technology – Database Languages – SQL Multimedia and Application Packages – Part 3: Spatial*. Standard. International Organization for Standardization, 2003.

[60] *ISO/IEC 9075-1:2011 Information technology — Database languages — SQL — Part 2: SQL/Foundation*. Tech. rep. 2011.

[61] Edwin H. Jacox and Hanan Samet. “Spatial join techniques”. In: *ACM Trans. Database Syst.* 32.1 (2007), p. 7.

[62] Alekh Jindal et al. “Selecting Subexpressions to Materialize at Datacenter Scale”. In: *Proc. VLDB Endow.* 11.7 (2018), pp. 800–812.

[63] Per-Åke Larson and H. Z. Yang. “Computing Queries from Derived Relations”. In: *PVLDB*. Ed. by Alain Pirotte and Yannis Vassiliou. 1985, pp. 259–269.

[64] Scott T. Leutenegger, J. M. Edgington, and Mario Alberto López. “STR: A Simple and Efficient Algorithm for R-Tree Packing”. In: *ICDE*. Ed. by W. A. Gray and Per-Åke Larson. IEEE Computer Society, 1997, pp. 497–506.

[65] Stuart P. Lloyd. “Least squares quantization in PCM”. In: *IEEE Trans. Inf. Theory* 28.2 (1982), pp. 129–136.

[66] Ming-Ling Lo and Chinya V. Ravishankar. “Spatial Hash-Joins”. In: *SIGMOD Conference*. ACM Press, 1996, pp. 247–258.

[67] Jiamin Lu and Ralf Hartmut Güting. “Parallel SECONDO: A practical system for large-scale processing of moving objects”. In: *ICDE*. IEEE Computer Society, 2014, pp. 1190–1193.

[68] Nimrod Megiddo and Dharmendra S. Modha. “ARC: A Self-Tuning, Low Overhead Replacement Cache”. In: *FAST*. USENIX, 2003.

[69] Sara Migliorini and Alberto Belussi. “A Balanced Solution for the Partition-based Spatial Merge Join in MapReduce”. In: *EDBT Workshops*. Vol. 2578. CEUR Workshop Proceedings. 2020.

[70] Dimităr Mišev and Peter Baumann. *SQL Support for Multidimensional Arrays*. Information Resource Center der Jacobs University Bremen, 2017.

[71] Kasper Mullesgaard et al. “Efficient Skyline Computation in MapReduce”. In: *EDBT*. 2014, pp. 37–48.

[72] Jürg Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. “The Grid File: An Adaptable, Symmetric Multikey File Structure”. In: *ACM Trans. Database Syst.* 9.1 (1984), pp. 38–71.

[73] Maitry Noticewala and Dinesh Vaghela. “MR-IDBSCAN: Efficient Parallel Incremental DBSCAN Algorithm using MapReduce.” In: *International Journal of Computer Applications* 93.4 (2014), pp. 13–17.

[74] Tomasz Nykiel et al. “MRShare: Sharing Across Multiple Queries in MapReduce”. In: *Proc. VLDB Endow.* 3.1 (2010), pp. 494–505.

[75] Christopher Olston et al. “Pig Latin: A Not-So-Foreign Language for Data Processing”. In: *SIGMOD Conference*. ACM, 2008, pp. 1099–1110.

[76] OpenGIS Consortium Inc. *OpenGIS Simple Feature Access Specification For SQL*. Tech. rep. 1999.

[77] Jack A. Orenstein. “Spatial Query Processing in an Object-Oriented Database System”. In: *SIGMOD Conference*. ACM Press, 1986, pp. 326–336.

[78] Muhsen Owaida et al. “Lowering the Latency of Data Processing Pipelines Through FPGA based Hardware Acceleration”. In: *Proc. VLDB Endow.* 13.1 (2019), pp. 71–85.

[79] Varun Pandey et al. “How Good Are Modern Spatial Analytics Systems?” In: *Proc. VLDB Endow.* 11.11 (2018), pp. 1661–1673.

[80] Stavros Papadopoulos et al. “The TileDB Array Data Storage Manager”. In: *Proc. VLDB Endow.* 10.4 (2016), pp. 349–360.

[81] Luis Leopoldo Perez and Christopher M. Jermaine. “History-aware query optimization with materialized intermediate views”. In: *ICDE*. IEEE Computer Society, 2014, pp. 520–531.

[82] Timo Räth. “Efficient Raster Operations in Apache Spark”. Master’s thesis. TU Ilmenau, 2019.

[83] Niles Ritter and Mike Ruth. *GeoTIFF Format Specification*. Tech. rep. 1995.

[84] Abhishek Roy et al. “SparkCruise: Handsfree Computation Reuse in Spark”. In: *Proc. VLDB Endow.* 12.12 (2019), pp. 1850–1853.

[85] Omran Saleh, Stefan Hagedorn, and Kai-Uwe Sattler. “Complex Event Processing on Linked Stream Data”. In: *Datenbank-Spektrum* 15.2 (2015), pp. 119–129.

[86] Simonas Šaltenis and Christian S. Jensen. “R-tree based indexing of general spatio-temporal data”. In: *A TIMECENTER Technical Report* (1999).

[87] Simonas Saltenis et al. “Indexing the Positions of Continuously Moving Objects”. In: *SIGMOD Conference*. ACM, 2000, pp. 331–342.

[88] Hanan Samet. “The Quadtree and Related Hierarchical Data Structures”. In: *ACM Comput. Surv.* 16.2 (1984), pp. 187–260.

[89] Kai-Uwe Sattler, Ingolf Geist, and Eike Schallehn. “QUIET: Continuous Query-driven Index Tuning”. In: *VLDB*. Morgan Kaufmann, 2003, pp. 1129–1132.

[90] Karl Schnaitter et al. “COLT: continuous on-line tuning”. In: *SIGMOD Conference*. ACM, 2006, pp. 793–795.

[91] Mehdi Sharifzadeh and Cyrus Shahabi. “The Spatial Skyline Queries”. In: *Proc. VLDB Endow.* ACM, 2006, pp. 751–762.

[92] Darius Sidlauskas and Christian S. Jensen. “Spatial Joins in Main Memory: Implementation Matters!” In: *Proc. VLDB Endow.* 8.1 (2014), pp. 97–100.

[93] Evan R. Sparks et al. “KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics”. In: *ICDE*. IEEE Computer Society, 2017, pp. 535–546.

[94] Andreas Spitz et al. “Refining imprecise spatio-temporal events: a network-based approach”. In: *GIR*. ACM, 2016, 5:1–5:10.

[95] Divesh Srivastava et al. “Answering Queries with Aggregation Using Views”. In: *VLDB*. Morgan Kaufmann, 1996, pp. 318–329.

[96] Knut Stolze. “Integration of spatial vector data in enterprise relational database environments”. PhD thesis. University of Jena, Germany, 2006.

[97] Knut Stolze. “SQL/MM Spatial - The Standard to Manage Spatial Data in a Relational Database System”. In: *BTW*. Vol. P-26. LNI. GI, 2003, pp. 247–264.

[98] Michael Stonebraker et al. “The Architecture of SciDB”. In: *SSDBM*. Vol. 6809. Lecture Notes in Computer Science. Springer, 2011, pp. 1–16.

[99] MingJie Tang et al. “LocationSpark: A Distributed In-Memory Data Management System for Big Spatial Data”. In: *Proc. VLDB Endow.* 9.13 (2016), pp. 1565–1568.

[100] Yufei Tao and Dimitris Papadias. “MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries”. In: *Proc. VLDB Endow.* Morgan Kaufmann, 2001, pp. 431–440.

[101] Unidata UCAR. *NetCDF*. 2019.

[102] Gary Valentin et al. “DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes”. In: *ICDE*. IEEE Computer Society, 2000, pp. 101–110.

[103] Guoping Wang and Chee-Yong Chan. “Multi-Query Optimization in MapReduce Framework”. In: *Proc. VLDB Endow.* 7.3 (2013), pp. 145–156.

[104] Terry A. Welch. “A Technique for High-Performance Data Compression”. In: *IEEE Computer* 17.6 (1984), pp. 8–19.

[105] Randall T. Whitman et al. “Spatial Indexing and Analytics on Hadoop”. In: *SIGSPATIAL/GIS*. ACM, 2014, pp. 73–82.

[106] Louis Woods, Zsolt István, and Gustavo Alonso. “Ibex - An Intelligent Storage Engine with Support for Advanced SQL Off-loading”. In: *Proc. VLDB Endow.* 7.11 (2014), pp. 963–974.

[107] Dong Xie et al. “Simba: Efficient In-Memory Spatial Analytics”. In: *SIGMOD Conference*. ACM, 2016, pp. 1071–1085.

[108] Dong Xie et al. “Simba: spatial in-memory big data analysis”. In: *SIGSPATIAL/GIS*. ACM, 2016, 86:1–86:4.

[109] H. Z. Yang and Per-Åke Larson. “Query Transformation for PSJ-Queries”. In: *Proc. VLDB Endow.* Morgan Kaufmann, 1987, pp. 245–254.

[110] Simin You, Jianting Zhang, and Le Gruenwald. “Large-scale spatial join query processing in Cloud”. In: *ICDE Workshops*. IEEE Computer Society, 2015, pp. 34–41.

[111] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. “A demonstration of GeoSpark: A cluster computing framework for processing big spatial data”. In: *ICDE*. IEEE Computer Society, 2016, pp. 1410–1413.

[112] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. “GeoSpark: a cluster computing framework for processing large-scale spatial data”. In: *SIGSPATIAL/GIS*. ACM, 2015, 70:1–70:4.

[113] Shubin Zhang et al. “SJMR: Parallelizing spatial join with MapReduce on clusters”. In: *CLUSTER*. IEEE Computer Society, 2009, pp. 1–8.

[114] Ying Zhang et al. “SRBench: A Streaming RDF/SPARQL Benchmark”. In: *ISWC*. Vol. 7649. Lecture Notes in Computer Science. Springer, 2012, pp. 641–657.