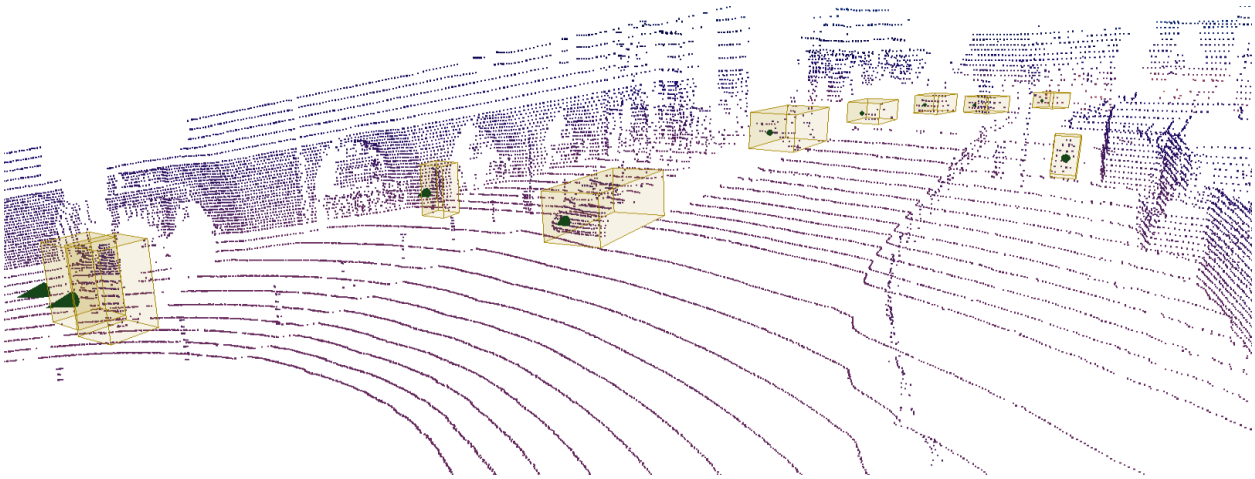
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Evaluating the Benefits of Spatio-Temporal Relational Operations for Validating LiDAR Perception Systems

Applied to Spatio-Temporal Joins in Apache Spark SQL

Master's thesis in Computer Science and Engineering

JAKOB HOLMGREN
PHILIP NORD

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

Evaluating the Benefits of Spatio-Temporal Relational Operations for Validating LiDAR Perception Systems

Applied to Spatio-Temporal Joins in Apache Spark SQL

JAKOB HOLMGREN

PHILIP NORD

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Evaluating the Benefits of Spatio-Temporal Relational Operations for Validating LiDAR Perception Systems
Applied to Spatio-Temporal Joins in Apache Spark SQL
JAKOB HOLMGREN
PHILIP NORD

Supervisor: Vincenzo Massimiliano Gulisano, CSE
Advisors: Isak Hjortgren and Michel Edkrantz, Annotell
Examiner: David Sands, CSE

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Visualization of an Annotated Point Cloud

Typeset in LaTeX
Gothenburg, Sweden 2021

Evaluating the Benefits of Spatio-Temporal Relational Operations for Validating LiDAR Perception Systems
Applied to Spatio-Temporal Joins in Apache Spark SQL
JAKOB HOLMGREN
PHILIP NORD

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

On the road to fully autonomous vehicles, advancements in sensor technology have enabled Advanced Driver-Assistance Systems (ADAS) in assisting the driver with tasks such as lane-keeping and collision avoidance. ADAS systems are enabled by perception systems which, given sensor input, outputs a perception model. Recently proposed perception models utilize data from the Light Detection And Ranging (LiDAR) sensor, that measures nearby occlusions with high definition and frequency. This enables perception models to detect objects and identify environmental segments relevant to ADAS tasks.

The critical safety requirements of perception models have created a demand for a framework that can efficiently assess the performance of the model on large amounts of labeled sensor data. Scalable analysis frameworks such as Apache Spark allow developers to perform large-scale data analysis using abstract interfaces while exploiting data-level parallelism in a cluster of nodes.

As Spark is a general platform designed for business intelligence and machine learning, there is a trade-off in both *expressive power* and *efficiency* for specialized data types. This trade-off is relevant for processing LiDAR data and by extension validating perception models. This lack of specialization impacts model validation such that only expert users have access to efficient implementations. Non-expert users are forced to write complex custom functions for their data which impact efficiency negatively. This increases the risk of errors in the implementation as custom functions are more difficult to interpret.

As the query compiler of Spark is inherently extensible, we investigate how its query language, optimizer, and engine have been extended with support for specialized operations on spatial and spatio-temporal data. Furthermore, we investigate how these extensions perform when applied to validating LiDAR sensor perception systems. In our pre-study, we identified spatial joins between volumes and points to be useful unsupported data operations. By implementing these operations with different approaches in Spark, we evaluate the expressive power and efficiency they provide. Finally, we perform a scalability analysis of the query response time of each implementation. We find that using spatial libraries does not improve the efficiency for the specific spatio-temporal join. Furthermore, we find that implementing the query and predicate is not easier with the specialized libraries. However, we find that accounting for different types of shapes of the volumes may impact efficiency on larger datasets.
Keywords: LiDAR, Apache Spark, Spatial, Spatiotemporal, 3D, Point Cloud

Acknowledgements

We want to thank our supervisor Vincenzo for guiding us with focus, our advisors Isak and Michel for providing insight, and Annotell for providing data, resources and for trusting us with the freedom to shape this project together.

Jakob Holmgren and Philip Nord, Gothenburg, June 2021

Contents

List of Figures

List of Tables

List of Algorithms

1

Introduction

This chapter introduces the main topic of this thesis, the motivation behind the topic, and the contribution of the thesis. Firstly, Section 1.1 introduces autonomous sensors and objectives in the automotive domain as well as how they relate to this thesis. Secondly, the problem definition is presented in Section 1.2 where scope, delimitations, and research questions are defined based on the objectives and problems in the domain. Finally, the main contributions, an outline of the thesis, and how they relate to the research questions are presented in Section 1.3.

1.1 Background

A recent goal in the automotive industry is to implement autonomous systems that can take driving decisions to safely maneuver a vehicle in different environments. To enable the development of such a system, it needs to understand its surroundings. For this purpose, a perception system ingests sensor data and outputs a model of the perceived physical environment captured by the sensor which can be fed to the autonomous system. A sensor that is widely used in the autonomous domain is the Light Detection And Ranging (LiDAR) sensor [32, 24, 29, 28]. The perception system takes measurements from the LiDAR sensor as input and outputs a model of the perceived environment. An example of such an output model is a description of all pedestrians detected by the perception system for the input data.

Given the safety requirements in the automotive domain, having high confidence in the performance of the autonomous system is vital [13, 36]. To confidently reason about the safety of an autonomous system, the performance of the perception system needs to be validated. That is, quality metrics of the model produced by the perception system needs to be computed as the model is what the autonomous system acts on. In order to be statistically confident about the abilities of the perception system, it needs to be validated against large amounts of data. The scale of the data is further emphasized by the requirement that the data needs to be collected from all environments the perception system is expected to operate in. For example, one can not be statistically confident in the abilities of the perception system when operating in the rain if it has only been validated on data gathered when it did not rain.

Input data magnitude and variation are two aspects of performance evaluations for perception systems, another is the implementation of the evaluation. When specifying metrics for the performance of a perception system, the implementation that

evaluate these metrics need to be simple in order to safely reason about the semantics of the code. Furthermore, it should be possible to leverage the efficiency and ease of implementation for domain experts and data analysts who are not necessarily computer scientists and experts at the inner workings of the computing environment. We observe two requirements in our computing environment to enable the evaluation of perception systems:

1. The computing environment needs to be efficient in evaluating the performance of perception systems on large amounts of data.
2. The computing environment needs support for high-level operations such that an evaluation is easy to implement and reason about.

Numerous data processing frameworks utilize the MapReduce paradigm to achieve data-level parallelism with two general data operations, map and reduce. Map operations produce one output item per input item independently, which means that it can be distributed across a cluster of workers to be run on different parts of a distributed dataset in parallel [12]. Drawing inspiration from the breakthroughs and to remedy the limitations of MapReduce, Apache Spark was implemented using an abstraction of distributed datasets called Resilient Distributed Datasets (RDDs) [57]. This allowed many different Map- and Reduce-type operations to be supported while the abstraction made Spark state-of-the-art in large-scale data processing due to the in-memory model and fault-tolerance enabled by RDDs. This ability to process large amounts of data makes Spark an interesting framework based on the first requirement.

For the second requirement, different projects such as Pig [33] and Shark [54] aim to support data manipulation languages for general large-scale data processing frameworks. Expanding on the ideas of Shark [54], Spark SQL was introduced as a package in the Spark project that implemented a rule-based query compiler named *Catalyst* [4]. Catalyst supports several high-level languages for manipulating large amounts of data in RDDs. For special operations, Spark SQL does not only support User Defined Functions (UDFs) written in high-level languages, but the inherent extensibility of the rule-based model of Catalyst allows the development of libraries with optimized UDFs that can harness the RDD primitives to provide efficient implementations of special operations.

Related works that propose frameworks for processing *spatial*, *geospatial*, and *spatiotemporal* data in Spark have been surveyed in [3, 16, 21, 37]. While proposed frameworks implement various specialized operations, these recent surveys show a lack of support for three-dimensional spatiotemporal data types, and by extension, LiDAR data operations.

1.2 Problem Statement

The lack of support for three-dimensional data operations in large-scale processing frameworks is a problem both in terms of computational efficiency and ease of implementation when evaluating LiDAR perception systems. It should be noted though, that Spark is a promising candidate for a framework that could satisfy these two

requirements for LiDAR data processing. This is mainly because the inherent extensibility of Catalyst's design has enabled a straightforward methodology to implement libraries that add specialized operations to Spark SQL. The main objective of this thesis is to investigate how well-suited Spark is for processing LiDAR data, what might be missing to improve such processing, and to survey existing techniques in the Spark ecosystem to find a trade-off between ease of implementation and performance. In order to better understand the requirements for LiDAR data processing, this thesis project is conducted in collaboration with Annotell, a Gothenburg-based company that operates in analytics and annotation of autonomous vehicle perception systems [25].

First, the requirements outlined in the previous section for determining whether Spark is well-suited or not need to be clarified:

- Efficiency - The computation time for the data processing needs to scale well with the input data size.
- Expressive Power - The validation needs to be simple to express and understand.

To expand on the main objective and these requirements, we define the following research questions which the thesis aims to answer.

RQ1 - What are the requirements of a framework for developing large-scale LiDAR data processing pipelines?

RQ2 - What useful properties that could speed up the processing of LiDAR data are not utilized?

RQ3 - Can we utilize the properties identified in the previous question to reduce the computation time for a LiDAR data processing problem defined by Annotell?

These questions and our thesis are scoped for evaluating the efficiency and expressive power of high-level operations for LiDAR data processing. The thesis is not concerned with how to best utilize these operations or how to best define a LiDAR classification evaluation. While efficiency is considered, this thesis will not delve into trade-offs between accuracy and computational efficiency for LiDAR classification evaluations.

1.3 Contribution and Outline of Thesis

The main contribution of this thesis is an evaluation of the state of the art for LiDAR data processing in Apache Spark. To evaluate the contribution in the context of this thesis and to understand the results, some supporting chapters are needed before the conclusion and results are presented. In Chapter 2 we present preliminaries necessary to understand the evaluation of a perception model end-to-end, from sensor data collection to output evaluation metrics. To identify what techniques are available, gaps in current research, and the contribution space, Chapter 3 introduces related work focused on specialized, efficient data analysis in Apache Spark. Not only does this help shape the methods for designing and evaluating contributions, presented in Chapter 4, but also provides insight towards answering the research questions.

These methods are applied to different approaches to create and implement solutions in Chapters 5, 6, and 7. In Chapter 8 these implementations are evaluated and compared in terms of both efficiency and expressive power in order to relate them to the research questions. Furthermore, the implementation strategies, execution plans, and results are discussed to give an understanding of the findings. Finally, a conclusion relating the results to the purpose of the thesis is presented in Chapter 9.

2

Preliminaries

This chapter presents preliminary information needed to understand the setting of the thesis as well as what a LiDAR perception model evaluation consists of. First, the LiDAR sensor and the annotation process are presented. This is followed by an explanation of how evaluations of performance models are enabled by relating perception model data to annotation data. After this, Resilient Distributed Datasets, the core concept behind Apache Spark are explained. Then, query processing and relational operations in traditional databases are introduced. Finally, we tie together the Apache Spark concepts with query processing by expanding on how Spark SQL bring these together.

2.1 LiDAR Sensor Data Annotation

A popular type of sensor in distributed cyber-physical systems is the Light Detection And Ranging (LiDAR) sensor [32]. The LiDAR sensor continuously scans its surroundings with lasers, measuring the relative distance to objects and surfaces by measuring the time it takes for the light to reflect back. This repeatedly creates detailed descriptions of the surrounding environment, called point clouds. This data is used as input for perception tasks, whether they are performed by a system or a human. The perception task at hand specifies a task objective such as "detect all pedestrians", or "mark all road signs". As there is a wide variety of pedestrians and road signs, writing step-by-step algorithms is not exhaustive enough. This is one of the reasons to why proposed perception systems rely on machine learning models to efficiently mimic the judgements of a human. These judgements make up the *ground truth* data set of the perception task, which the perception system learns from by optimizing an objective function over the examples. There are major differences in how this task is performed by the perception system during training, validation, and production. During development and validation, the classification function is improved and evaluated interchangeably batch-wise, whereas in production, perception needs to be performed continuously on the arriving sensor stream. In order to evaluate performance on a perception task, an evaluation data set is needed for which two perception models are compared. This is usually done with experimental output from a trained perception system and another ground truth data set.

To produce a ground truth data set, the sensor data is displayed to a human which perceives the data and performs manual annotation according to the perception task. The effort required for each example depends on the granularity of the task and

the richness of the data. In semantic segmentation, each individual measurement such as a pixel in an image or a point in a point cloud is classified [20]. This is usually very expensive as the size of the perception model is as large as the number of measurements. Detection tasks, where the objective is to detect and classify objects, are less fine-grained. As objects identified in an image consist of multiple measurements, a detection task encapsulates multiple points at once. For the sake of cost-effectiveness, rather than manually labeling each point individually, simpler shapes such as bounding boxes is commonly used [7]. A figure showing the visualization of a point cloud image together with annotated boxes around vehicles can be seen in Figure 2.1. One thing to note is that this shows how the annotator receives a reference view to enrich the perception of the scene.

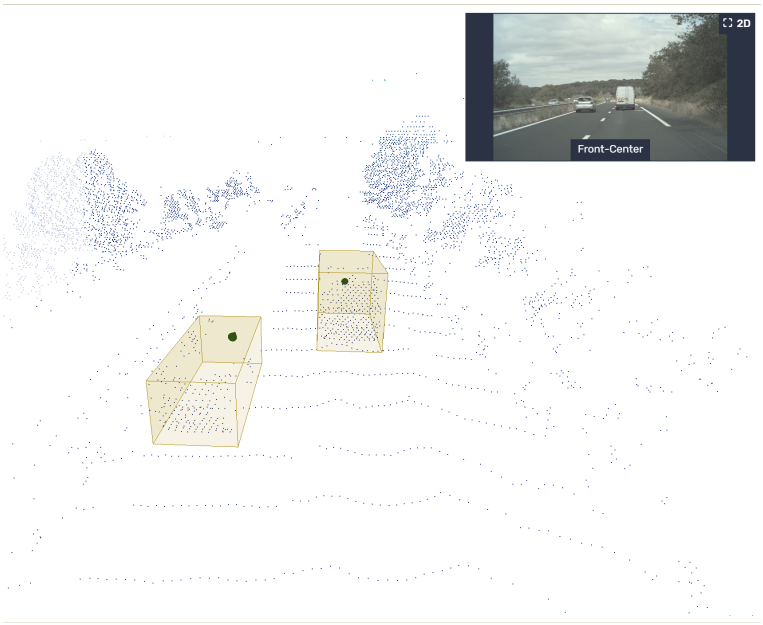


Figure 2.1: A perception task of drawing boxes around vehicles where two vehicles are identified and accordingly surrounded by bounding boxes.

It is important to note that while the annotation data set is referred to as ground truth it does not necessarily model reality perfectly. For example, in Figure 2.1 the vehicles are annotated as cuboids which is a simplification of their actual shapes. Allowing for annotations to be approximate increases the annotation speed significantly which is relevant given the scale of the data that needs to be evaluated and by extension annotated. Furthermore, the annotations only need to be accurate for the parts of the data that are relevant for autonomous driving tasks. For example, a cuboid annotation of a car still correctly models the space that is unsafe to enter as a driving action as visualized in Figure 2.2. For other combinations of tasks and objects, this approximation might not be acceptable. An example of this is an overpass where a cuboid annotation could model the overpass as collision-relevant when it is not as illustrated in Figure 2.3. To conclude, annotations do not always model reality perfectly, have various precision as they take either a cuboid or concave shape, and the desired precision is highly contextual.

Figure 2.2: A cuboid annotation of a car highlighted in red. Despite the lack of precision in cuboid annotations the car is correctly encapsulated and models that the area is unsafe to enter.

(a)

(b)

Figure 2.3: The same overpass annotated as a cuboid in (a) and as a concave shape in (b) where the cuboid annotation incorrectly models that it can not be driven under and the concave shape correctly models that it is can be driven under.

2.2 Point Cloud Segmentation Evaluation

The evaluation process analyzes the confusion between the two perception models, usually the classifications produced by the perception system and by human annotations. To enable the evaluation process, the same perception task has to be performed both by the perception system and the annotator. The four types of outcomes when analyzing the confusion between the two perception models for a task are *True Positive* (TP), *False Positive* (FP), *False Negative* (FN), and *True Negative* (TN) as shown in Table 2.1. Evaluating an outcome is highly dependant on the perception task. For example, for the detection task of identifying pedestrians, a true positive would be defined by the manual annotation agreeing with the perception system on the position of a pedestrian. A false positive means that the automated system identifies a pedestrian where the manual annotation does not, this can be due to disagreeing on the position, classification, or both. An example of this could be a mannequin in a store window which the perception system falsely identified as

a pedestrian whereas the manual annotation does not specify any pedestrian in the same location.

	Annotated Positive	Annotated Negative
Classified Positive	True Positive	False Positive
Classified Negative	False Negative	True Negative

Table 2.1: A confusion matrix for manual annotations and automated classifications produced by the perception system.

The example with the pedestrian is a *detection* task, where the output model is a set of detections. Comparing the two sets of detections gives subsets TP, FP, and FN. As TN is the complement of the prior three, and the universe is usually infinite, TN is also infinite and therefore not meaningful for detection metrics. Therefore, confusion metrics such as precision and recall, which are based on TP, FP and FN, are often used to evaluate detection tasks [48].

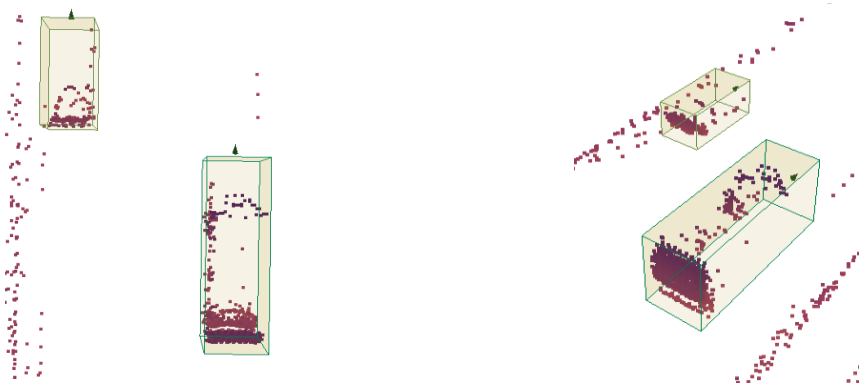
To validate a semantic segmentation perception system, various statistical confusion metrics have to be calculated on diverse test data. The first step of this process is to create the validation data set. As there are several ways to annotate semantic segmentation [20], this may require some preprocessing. When performing semantic segmentation through volumetric annotation, as described in the previous section, the actual classifications of each point in the point cloud is retrieved by cross-matching the point cloud data with the volumetric labelling. This models the classification function

$$semseg(point) \in \{class | (volume, class) \in Volumes, contains(volume, point)\}$$

where the $contains(volume, point)$ predicate holds if the *point* is in the *volume* in both the spatial and temporal dimensions. As observed in the previous section with cars volumetrically annotated as cuboids, the precision of these volumes may vary. Usually, this is acceptable in semantic segmentation as points contained within the cuboid are likely part of the car so other potential objects in the bounding space are disregarded. An example of how points are distributed in two cuboids, exemplifying how volumetric annotations can be suitable for semantic segmentation evaluations, can be seen in Figure 2.4.

When the perception models are both in the same point-wise space, various metrics have been proposed to evaluate the confusion of the models [48, 39, 20]. Apart from these, various other Key Performance Indicators (KPIs) are used in industry to communicate the quality of annotated data sets and perception systems [25].

As there are numerous possibly interesting confusion metrics and KPIs they all have to be performed on large validation datasets to ensure statistical safety [40]. Since these are required to contains many hours of recorded data, efficiency in computations as well as supporting data-level parallelism is critical to enable quick query-response time for short development cycles.



(a) (b)

Figure 2.4: Cuboid annotations of two vehicles as seen from above in (a) and from the back-right in (b). The cuboids are largely empty as the LiDAR can not see through the vehicles.

2.3 Resilient Distributed Datasets

The core of the Apache Spark framework is built on the *Resilient Distributed Dataset* (RDD) model [57] which enables in-memory execution of MapReduce [12] jobs. An RDD is described as an immutable, partitioned collection of records that can be defined in two ways. Either by reading some data from a distributed storage solution or by applying some *transformation* operation to an existing RDD. An example of a transformation is the map function. By applying a map function on each record of an existing RDD one can create a MappedRDD. These transformation operations allow the creation of RDDs. The other supported type of operations are *actions*. Actions compute some output based on the contents of the RDD and the action performed. An example of an action is *count* which calculates the number of records in the RDD. An important detail for performance reasons is that RDDs are not materialized until an action is performed on them. This is possible since RDDs track their lineage as chains of transformations, e.g. a MappedRDD knows that in order to materialize it will need to apply the provided mapping function to each record of another RDD. Not only does this mean that operations on RDDs can be evaluated lazily but it also means that fault tolerance can be achieved by recomputing missing RDDs as each RDD specifies how it can be materialized.

The RDD model builds upon the MapReduce paradigm and while MapReduce programs do not need to employ the RDD abstraction, doing so has some clear advantages. An example of this is how MapReduce lacks abstractions for in-memory computations. This leads to depending heavily on writing to and reading from disk, especially when performing iterative computations. This is bad from a performance standpoint but is a necessity in how the popular MapReduce frameworks achieve fault tolerance. This can be compared to RDDs which instead, using their lineage, recompute missing results, minimizing expensive disk operations.

The execution model enabled by RDDs allow developers to write a driver program defining RDDs, transformations, and actions. This abstracts away the underlying

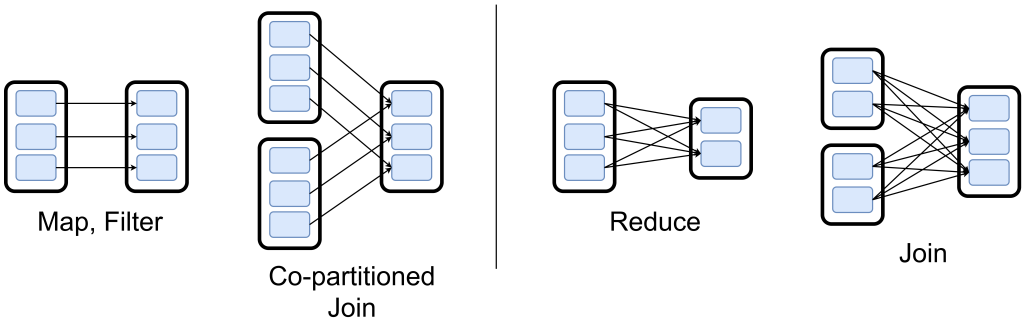


Figure 2.5: Visualization of partition dependencies of four RDD transformations. Two with narrow dependencies and two with wide dependencies.

distributed execution model efficiently, hiding details like which data is on which node and what data needs to be shuffled between nodes. However, some knowledge of what transformations are cheap and expensive is still required. This is mainly communicated through the concept of *narrow* and *wide* dependencies, depending on whether each input partition is dependant on one or more output partitions. For narrow dependencies, less data shuffling is required as the new partition can be created from where the old is located. For wide transformations such as joins, which requires the comparison of each combination of records in the input RDDs, data shuffling may becomes very expensive. These dependencies are visualized for four operations in Figure 2.5.

Since its launch new APIs and libraries have been added to the Spark ecosystem, all of these are built on top of the RDD core of Spark.

2.4 Query Processing and Operations in Relational Databases

The computational model of Spark made a big leap away from traditional database systems in order to support very large data sets. While Spark efficiently abstracts away some complexities of the distributed MapReduce model, the RDD API is still far from declarative. Traditional database systems utilize the relational data model proposed by Codd [8], to enable users to write declarative queries, explaining what result they want rather than what steps are needed to take in order to generate it.

The relational data model is based on set-theory, where a relation R is a set of tuples $(t_1, t_2, ..., t_n)$ and each element t_i is of type T_i. A relational database is then a collection of relations. The tuple $(T_1, T_2, ..., T_n)$ is referred to as the schema of the relation. A relational database defines two sets of relations, the *stored* set and the *expressible* set. The relations in the stored set are usually referred to as tables and are materialized in some arbitrary fashion. The expressible set of relations contain all relations which can be expressed using queries written in the Data Query Language (DQL) supported by the relational database.

2.4.1 Relational Operators

While it is possible to specify and support a DQL in an infinite number of ways, Codd proposed a generalized algebra of relational operators which transform one or more input relations to an output relation.

The relational algebra introduces three major operands: Projections (Π), Selections (σ) and Joins ($\bowtie$). These operands are semantically similar to the `map`, `filter` and `join` functions of RDDs respectively. A common extension to the algebra introduced by Codd is to include Aggregations [34]. As Aggregations are semantically similar to the `reduce` function in RDDs, they are a core feature of Spark and the MapReduce paradigm, where aggregations of large amounts of data is one of the key benefits of the system.

Projection (Π)

Projections simply transform each tuple in the input relation R to a tuple in the output relation S given a projection function $\pi : R \to S$.

$$\Pi_\pi(R) = \{s | r \in R \wedge s \in S \wedge \pi(r) = s\}$$

Selection (σ)

Selections filter the input relation based on a predicate relation θ.

$$\sigma_{a\theta}(R) = \{r | r \in R \wedge r.a \in \theta\}$$

Selection is unary as it takes one relation R with the domain T, one subset of the domain $a \subseteq T$, and one relation θ in the same domain as a. The output of a selection operation then gives all tuples in R for which $r.a$ is contained in θ. For instance, if one has the relation *points* with the domain $(x, y, z, time)$, and wants to select all point at time t for some visualization purposes, the selection operation would be defined as:

$$\sigma_{time=t}(points) = \{p | p \in points \wedge p.time \in \{t\}\}$$

Notice that "= t" is shorthand for the relation θ containing only the element t. If one would want to query all succeeding points of t, the relation θ would be $\theta = \{s \in time | s > t\}$

Aggregation (G)

Aggregate functions calculate output based on all values in a column. Examples include averages, minimum, maximum, and number of values. Combined with aggregate functions is the concept of grouping. When combining an aggregate function with a grouping specification, the aggregation is performed on the values of each group, resulting in one aggregate value for each group.

Join ($\bowtie$)

Joins are binary operands which means they have two input relations R and S. Similarly to Selection, they require a predicate relation θ which has the domain $T_\theta \in T_R \times T_S$. It is therefore a subset of all possible combinations of some attribute in R and S, indicating that the result are the combinations of tuples in R and S which the join attribute is in θ.

$$R \bowtie_{a\theta b} S = \{r \cup s | r \in R \wedge s \in S \wedge (r.a, s.b) \in \theta\}$$

The output of a θ-Join thus produces the output relation P of the union domain which contains all tuples in the input relations connected by θ. This is useful for connecting disparate relations. For instance, if one wants to translate points based on some recorded vehicle trajectory, the following join and projection is all that's required.

$$\Pi_{time,point+translation}(points \bowtie_{time\theta time} translations)$$

where

$$\theta = \{(a, b) | a \in Time \wedge b \in Time \wedge a = b\}$$

These different definitions of θ predicates are what allow many different variations of selection and join operations to be created.

One subset of predicate relations are spatial relations. We can identify different spatial relations in natural language such as `inside`, `intersects`, `touch`, `disjoint`, `covered` etc [14]. These help describe what relation entities of spatial type have. Spatial types have been implemented in various relational databases according to the Simple Features standard, formalized by the International Standardization Organization (ISO) [26] and Open GIS Consortium (OGC) [9]. These define spatial types, functions, and predicate relations including those mentioned above. An `ST_` prefix is applied to make explicit that these are applied to spatial types e.g. `ST_Contains`.

2.4.2 Query Processing

The purpose of the relational model is to support an expressive and declarative DQL. The declarative language style expresses how an output relation is derived from other relations, as opposed to the execution steps required in order to produce the output relation. This allows the user to not have to worry about how the data is physically stored or partitioned when manipulating it. It is instead up to the designer of the system to employ different database techniques to strike balance between optimizing specific queries while supporting other queries the system may be asked to support in the future.

In order to support a declarative DQL, traditional computer language techniques found in interpreters are utilized. The processing of a query is described in the database field as a series of steps from query text to a result [17]. Figure 2.6 shows the three major steps. (1) *Parsing* translates the query into a logical query plan,

represented as an abstract syntax tree. (2) *Optimization* transforms the plan into a physical execution plan, consisting of the primitive operations of the execution engine, by using different optimization strategies and metadata such as the schema and table sizes obtained from the catalog. (3) Finally, *Execution* runs the physical execution plan in the execution engine against the stored data, outputting the query result.

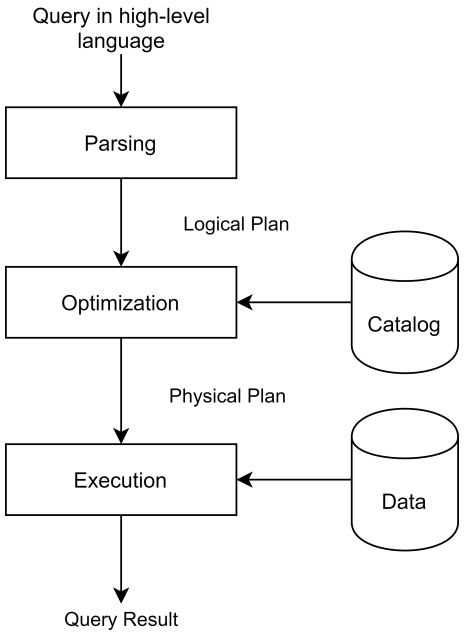


Figure 2.6: Query processing steps.

Similar to a type of language compilers, called interpreters, the three query processing steps (1) creates, (2) manipulates, and (3) consumes an internal abstract syntax tree [45]. Optimizations that ensure that an efficient physical execution plan is generated in (2) are critical. Heuristics of available tricks and shortcuts, are encoded in the query optimizer which manipulates the abstract syntax tree in a way that produces the same result in an optimal way. which means that for specific workloads such as spatial workloads, a specific set of heuristics exists which needs to be implemented to accelerate these workloads [46].

2.5 Spark SQL

In order to make the Spark framework more accessible, the Spark SQL module introduced relational processing on top of RDDs [4]. Spark SQL implements the ideas of Shark [54], with a query optimizer called *Catalyst*. Catalyst allows users to write queries using two declarative APIs: *DataFrames* and *SQL*. Catalyst is able to generate optimized execution plans for the Spark Runtime given code written using these APIs.

2.5.1 Catalyst

The compilation process in Catalyst takes a query and produces a physical Spark plan in four steps, *parsing*, *analyzing*, *optimizing*, and *physical planning*. Through these steps, Catalyst turns the query into an optimized physical execution plan, based on RDD primitives, that can be executed by the Spark engine.

Spark SQL provides a high-level DataFrame API that allows users to scale their evaluation to larger data sets without having to worry about the underlying physical execution plan. For many use-cases, using declarative languages allows Catalyst to generate a more efficient execution plan compared to writing it manually [4]. The query processing in catalyst takes a query and produces a physical Spark plan of RDD primitives using the general query processing steps described in Section 2.4.2. Internally, catalyst divides the optimization step into *analyzing*, where names referring to tables and columns are resolved using the catalogue of schemas, and *optimizing* where the logical plan is translated into an optimized physical plan. The physical plan is different from the logical plan in that the nodes of the syntax tree are RDDs. This means that operations are map- or reduce-type functions or specific patterns of data-shuffling. An overview of this can be seen in Figure 2.7.

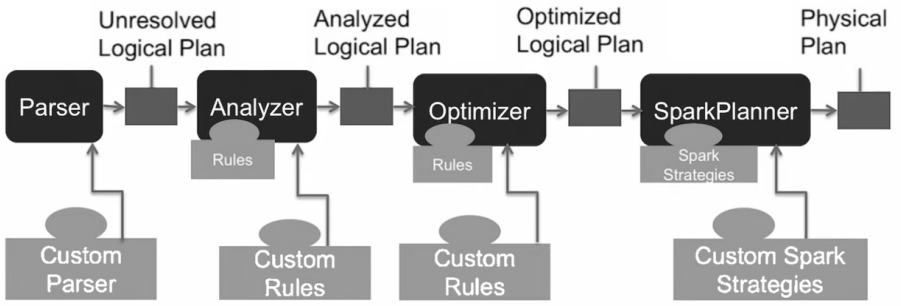


Figure 2.7: The catalyst compilation process from query to physical plan with custom extension points [27].

As seen in Figure 2.7, all of the steps are defined using rules and strategies. This allows general use cases to harness general optimizations and strategies, while also allowing external optimizations and strategies to extend Spark to specialize it for certain problem domains.

One useful strategy called predicate pushdown tries to apply filters as early as possible. This reduces the overall amount of data that is processed and for operations such as joins where the number of comparisons is at worst quadratic, this strategy has a big impact on the query complexity. This strategy is encoded into Catalyst as the `PushPredicateThroughNonJoin` and `PushDownLeftSemiAntiJoin` rules, which move filter-predicates before operations such as Projections and Joins. For a syntax tree, produced by query parsing, which first performs a join between two relations and then filters the result, the following Logical Plans demonstrate how the predicate pushdown strategy is applied. Note the order in which `Filter` and *Join* is applied.

Analyzed Logical Plan

```
== Parsed Logical Plan ==
Deduplicate [Annotation.timestamp]
+- Project [Annotation.timestamp]
   +- Filter (Annotation.timestamp = 123456)
      +- Project [...]
         +- Join Inner, (Annotation.timestamp = LiDAR.timestamp)
            :- Relation[Annotation] parquet
            +- Relation[LiDAR] parquet
```

Optimized Logical Plan

```
== Optimized Logical Plan ==
Aggregate [Annotation.timestamp], [Annotation.timestamp]
+- Project [Annotation.timestamp]
   +- Join Inner, (Annotation.timestamp = LiDAR.timestamp)
      :- Project [Annotation.timestamp]
      :  +- Filter (isnotnull(Annotation.timestamp) AND
      :             (Annotation.timestamp = 123456))
      :     +- Relation[Annotation] parquet
      +- Project [LiDAR.timestamp]
         +- Filter ((LiDAR.timestamp = 123456) AND
            :       isnotnull(LiDAR.timestamp))
            +- Relation[LiDAR] parquet
```

Multiple optimization strategies allow Spark SQL to support various declarative and imperative-style APIs for defining queries.

2.5.2 DataFrame

Spark SQL includes an API for DataFrames, a concept for which implementations already exist in the R language and the Python data science library Pandas [2]. Similar to those APIs, a DataFrame in Spark is analogous to a table in the relational data model in the sense that it is a collection of rows with a schema. Under the hood, DataFrames in Spark SQL share the same underlying data model as RDDs, where both are abstractions of collections of distributed data items. DataFrames also share other aspects to RDDs, such as lazyness where no execution occurs until an action is performed. Unlike RDDs, actions on DataFrames call Catalyst to perform query optimization which transforms the DataFrame to an equivalent RDD to be materialized. This enables one of the most important distinctions from DataFrames in Pandas and R in that DataFrame operations in Spark SQL are optimized before any evaluation starts.

With the addition of a schema, DataFrames allow different relational operations such as *select*, *where*, and *join* via its functional-style API. To illustrate the expressive power of the DataFrame API when compared to the RDD API we observe the two

code snippets below for which we implement a solution to getting the number of points in a point cloud for each timestamp using the two APIs.

RDD

```
parsedRDD
    .map { case(_, _, _, timestamp) => (timestamp, 1) }
    .reduceByKey(_+_)
```

DataFrame

```
parsedDataFrame
    .groupBy('timestamp).count()
```

2.5.3 Spark SQL Dialect

While DataFrames largely support the same set of operations as SQL, having support for writing SQL directly can be useful. This is especially true for users transitioning from a DBMS to Spark for data exploration as well as for computing multiple aggregates. For this purpose, a DataFrame can be registered as a temporary table on which one can perform SQL queries. It is important to remember that the DataFrame is still registered in the catalog as a non-materialized view which enables Catalyst to optimize the SQL queries and DataFrame expressions. An example of counting the number of points in a single point cloud instant using the SQL API can be seen below.

```
    df.registerTempTable("pointcloud")
    spark.sql("""
        SELECT count(*)
        FROM pointcloud
        WHERE timestamp = 123456789
    """)
```

It is also possible to modify the DataFrame before registering it as a table which can be quite useful, an example of this for the same problem can be seen below.

```
    df.filter('timestamp === 123456789)
      .registerTempTable("filtered_pointcloud")
    spark.sql("""
        SELECT count(*)
        FROM filtered_pointcloud
    """)
```

From the above example, we can also see that the SQL code and DataFrame API are quite similar. For example, *filter* for a DataFrame corresponds to *where* in SQL.

2.5.4 Partitioning Strategies

Both the RDD and DataFrame API in Spark expose two types of partitioning strategies: hash partitioning, and range partitioning. These make sure that data is distributed evenly between partitions, both physically, so partitions contain an even amount of data, and logically, so that partitions contain all data that will be reduced together. Logical co-partitioning enables otherwise expensive transformations with wide dependencies to be evaluated as a cheaper transformation using narrow dependencies.

Hash partitioning distributes tuples based on the values of hashed keys. Each partition is assigned a evenly sized range covering the entire range of the hash function. Hashing keys yields a deterministic, uniform distribution of hash values, making this strategy useful for reducing tuples on equal keys. This is due to the uniform distribution alleviating data skew, while ensuring that all tuples with equal keys end up in the same partition, making reduction on keys a transformation with narrow dependencies.

Not all reductions are performed on equal keys. For some operations that aggregate similar items, such as bin joins [52] and similarity joins [41], strict equality reduction is not sufficient. To partition similar tuples together the range partitioning strategy comes into play. Range partitioning assigns a low and high bound for each partition such that items with values within that range are only stored in that partition. To determine these low and high bounds in a way that prohibits skew, the data is first sampled. This partitioning strategy enables partition pruning, where partitions that are disjoint to the query can be safely ignored. For example, if all points measured in a time span are to be selected, using the high and low bound, we can safely determine whether each partition overlaps the time span or not and only process overlapping partitions.

3

Related Work

As shown in the previous chapter, Apache Spark is an interesting candidate for evaluating LiDAR perception models on large amounts of data. This chapter highlights previous work that extends the language and runtime of Apache Spark with specialized operations and optimizations. We are specifically focused on exploring work that brings support for spatial and spatio-temporal data types to Apache Spark. This serves two secondary purposes, by studying the current state-of-the-art we learn how Spark can be extended, as well as what types of operations and optimizations extensions choose to bring. In Table 3.1, the synthesized findings of related surveys are presented to create an overview of related contributions and what meaningful aspects these have. We also break down these aspects and highlight their usefulness for our purposes.

While there are several other related works that propose system-level architectures for storage [15], management [35], indexing [30], perception [42] and evaluation [31] of large amounts of LiDAR data with Apache Spark, the system-level perspective goes beyond the scope of this thesis.

3.1 Spatio-Temporal Libraries

Multiple Spark libraries improve working with specific types of spatio-temporal data. In recent studies, meaningful differences of Spark-based systems are compared [3, 16, 21, 37]. Specifically, these differences are found in supported data types, partitioning strategies, indexing strategies, query language, and queries. While these include both internal and external aspects, external aspects, such as supported language, data types, and queries, are most critical with regard to expressive power. Internal aspects may provide major performance gains, however, utilizing these is not viable without an understanding of internal low-level operations.

While there is ongoing work on RDD-level libraries supporting spatial analysis such as Spark3D [38], SparkGIS [5], LocationSpark [47] and SpatialSpark [55], the scope of this thesis focuses on libraries that support the Spark SQL query languages. These systems, denoted with "Spark SQL" in the API column of Table 3.1, integrate directly with Spark SQL to enable optimized spatial queries through the SQL and DataFrame interfaces.

Libraries achieve this by implementing custom *functions* such as projections, selection predicates, and join predicates as well as *strategies* to optimize queries using

partitioning and indexing techniques. This makes the evaluation of queries using the custom functions more efficient than using regular User-Defined Functions (UDFs).

Name and Active Years	Spark Version	API	Data Types	Partitioning	Indexing	Queries
Sedona [56] (2015-2021)	3.0.1, 2.4.7	Spark SQL, RDD	2D Point, LineString, Polygon, Rectangle	Uniform-Grid, Voronoi, R-Tree, Quad-Tree, KDB-Tree, Hilbert	R-Tree Quad-Tree	Range Selection, kNN Selection, Spatial Join, Distance Join
STARK [22] (2016-2020)	2.4.0	Spark SQL, RDD	STObject	Fixed-Grid, Binary-Space	R-Tree	kNN Selection, Spatio-Temporal Join
SpatialSpark [55] (2015-2017)	2.0.2	RDD	2D Point, LineString, Polygon, Rectangle	Unifrom, Binary-Split, STR	R-Tree	Range Selection, Spatial Join
Elcano [18] (2018)	Unknown	Spark SQL, RDD	2D Point, LineString, Polygon	N/A	GeoHash, R-Tree	Spatial Join
Simba [53] (2016-2018)	2.1.0	Spark SQL, RDD	2D Point	STR	R-Tree	Range Selection, kNN Selection, Distance Join, kNN Join
LocationSpark [47] (2015-2017)	1.6.2	RDD	2D Point, LineString, Polygon, Rectangle	Uniform-Grid, R-Tree, Quad-Tree	R-Tree, Quad-Tree, IR-Tree	Range Selection, kNN Selection, Spatial Join, kNN Join
Magellan [43] (2015-2018)	2.3.1	Spark SQL, RDD	2D Point, LineString, Polygon, MultiPoint, MultiPolygon	Z-Curve	N/A	Range Selection Spatial Join
SparkGIS [5] (2015-2016)	2.1.0	RDD	2D Point, LineString, Polygon	Fixed-Grid, Binary-Space, Quad-Tree, Strip-based, Hilbert-Curve, STR	R*-Tree	Range Selection, kNN Selection, Spatial Join

Table 3.1: Comparison of Spatial Libraries in Spark.

3.1.1 Data Types and Operations

For custom functions to be useful as spatial operations they need to be able to manipulate spatial datatypes. In Spark, these are registered by the library as User-Defined Types (UDTs). Many libraries implement UDTs which either borrow from or comply with the *Simple Features* specifications for two-dimensional geometrical data types and functions by ISO and the Open GIS Consortium (OGC) [9]. While the latest version of this specification by OGC specifies 3D data types, no libraries comply at this level. Instead, Elcano [18] distinguishes itself from other libraries by being strictly compliant at a lower compliance level while other libraries do not qualify for compliance at a basic level.

Geometries specified in Simple Features include `point`, `line`, and `polygon`. Spatial functions included in the standard either provides details about a shape, compare two shapes or create a new shape from another shape. Comparisons include predicates such as `intersects`, `touches`, and `contains` [44]. A large portion of libraries employ the Java Topology Suite (JTS) [11] to implement the UDTs specified by Simple Features. JTS offers wide support for two-dimensional spatial data types and functions, enabling spatial libraries to work as a middle layer between Spark and

the Simple Features specification. While some work specializes in spatial trajectory data and such data is available from the annotation process, these data types are outside the scope of this thesis.

3.1.2 Partitioning and Indexing

For the manipulation of UDTs to be efficient, partitioning and indexing strategies that support these UDTs need to be implemented to perform informed decisions on how to distribute data between and inside partitions respectively.

Multiple scalar dimensions are supported in Spark's native range partitioning which means that 3D points can be partitioned with regards to both X, Y, and Z. However, axis-aligned ranges do not sufficiently express arbitrary shapes such as lines, planes, and volumes. This prohibits expressivity as analysts need to define redundant columns for axis-aligned bounding boxes in order to utilize partition pruning. This is something that spatial libraries provide by extending Spark with custom partitioning and indexing strategies [52, 49]. Apache Sedona has the most diverse range of partition techniques as seen in Table 3.1. It provides various Grid- and Tree-based partitioning strategies as well as the Hilbert Curve, which is a type of Space-Filling Curve (SFC) that projects a space onto a line. Another SFC is the Z-Curve which Magellan employs as a partitioning strategy [43].

Spatial indexing organizes data such that spatial queries can be answered more efficiently. In some of the literature, partitioning is referred to as *global indexing* as it distributes data between partitions to achieve this. Furthermore, indexing strategies, or *local indexing*, speed up intra-partition processing using indexes such as spatial index trees like R-trees and Quad-trees. By first building and querying this tree, an expensive spatial predicate can be evaluated on a smaller dataset. Some libraries also support persistent indices, that store a global or local index to accelerate consecutive queries on the same spatial dataset.

Apache Sedona [56] (formerly GeoSpark) is one of the most active spatial libraries for Spark. It provides custom operations together with both custom optimization strategies to Catalyst and an extended RDD. The authors of Sedona also outline the limitations of Spark-based systems Simba, Magellan, SpatialSpark, and GeoMesa.

STARK is a library for working with spatio-temporal data in Spark [22]. It both extends the physical plan model with SpatialRDD, enabling efficient spatio-temporal analysis algorithms using partitioning and indexing strategies, providing Spark SQL with integrations for a declarative interface. STARK distinguishes the temporal dimension and predicates are conjunctions of spatial and temporal predicates.

There are more libraries proposed for similar purposes. While many of these are open-source, none have attracted the development community such as Apache Sedona. This has lead to a lack of maintenance and only Apache Sedona fully supports Spark version 3. While this means that most performance enhancements that come with the latest major version of Spark are not available when using STARK, STARK is still an interesting library to consider as it considers the temporal dimension.

4

Problem Specification

In the previous chapter, related work that extends Apache Spark with Spatial and Spatiotemporal operations was presented. As the surveys of Geospatial and Spatiotemporal libraries show, there is no literature or libraries, that optimize the processing of LiDAR sensor data for Apache Spark. This indicates room for optimizing analysis of LiDAR perception systems with Spark SQL, in terms of both efficiency and expressive power. To build and ultimately evaluate different solutions in this space, a clearly scoped problem is defined. This enables a concrete and contrastive evaluation of different approaches, presented in Chapters 5-7.

This chapter introduces the problem definition, the motivation behind it, and the evaluation metrics for a solution.

4.1 Problem Definition and Motivation

The problem definition will be explained as a spatial relation defined as the count aggregation of a spatiotemporal join of two input data sets, which counts the number of points that are contained by volumes.

$$G_{count}(annotation \bowtie_{contains} lidar)$$

In order to understand this output, its dependencies, and by extension the problem definition, first we need to understand the input data.

4.1.1 Input 1 - Point Cloud Data

Section 2.1 briefly explained how the LiDAR sensor outputs point clouds. In this section, the structure of point clouds is formalized since it is one of the inputs for our problem definition. A point cloud consists of multiple points, each of which has a classification as given by the perception model as well as a spatial and a temporal location. The spatial location consists of the points coordinates in the X, Y, and Z dimensions. The temporal location consists of a timestamp denoting when the point was measured. The relevant data types with a brief description are summarized in Table 4.1.

Point Cloud Data	
Data	*Description*
Classification	From perception model
Timestamp	Temporal Location
X	Point X-position
Y	Point Y-position
Z	Point Z-position

Table 4.1: Input data types and descriptions for point cloud data.

4.1.2 Input 2 - Volumetric Annotated Data

In Section 2.1, the annotation process was introduced as an option to create ground truth data to which the perception model classes can be compared. Here the output of the annotation process is formalized, since it is one of the inputs used in the problem definition. An annotated object has a classification as given by the annotator as well as a spatial and a temporal location. The temporal component is a timestamp relating it to a point cloud with the same timestamp. For annotations, the spatial component is more complex than for points in point clouds, as annotations can consist of various shapes to represent objects in the real world. For an annotation to represent different shapes there are two spatial components:

- Vertices - Each vertex has a unique identifier and X, Y, Z coordinates
- Faces - A face combines three vertices to create a triangle in 3D space, a shape is made up of a set of faces.

(a) **(b)**

Figure 4.1: A shape represented as just vertices in (a) and as faces denoting triplets of those vertices in (b).

Figure 4.1 visualizes how vertices and faces combine to create a cuboid shape, however, more complex shapes could be created similarly by having more vertices, faces and by combining the vertices into faces differently. The relevant data types for annotations with a brief description are summarized in Table 4.2.

Volumetric Annotated Data	
Data	*Description*
Classification	From annotation
Timestamp	Temporal location
AnnotationId	Identifier of this annotation
FaceID	Identifier of this face
Vertice1	The first VertexID of this face
Vertice2	The second VertexID of this face
Vertice3	The third VertexID of this face
VertexID	The identifier of this vertex
Vertex_X	The X coordinate of this vertex
Vertex_Y	The Y coordinate of this vertex
Vertex_Z	The Z coordinate of this vertex

Table 4.2: Input data types and descriptions for volumetric annotation data.

Note that both face- and vertex-attributes share the same table. This means that a row has `null`-values for the vertex-attributes if its a face and vice-versa.

4.1.3 Output - Spatially Relating of Inputs

The output is defined by the outcome of a spatial relation on the two inputs. The output should count the number of points in a point cloud that are *contained* within at least one of the volumetric annotations. For a point to be contained within a volumetric annotation it needs to be spatially located inside of an annotated volume and exist at the same timestamp as that annotated volume. A visualization of which points qualify and should be counted is given in Figure 4.2.

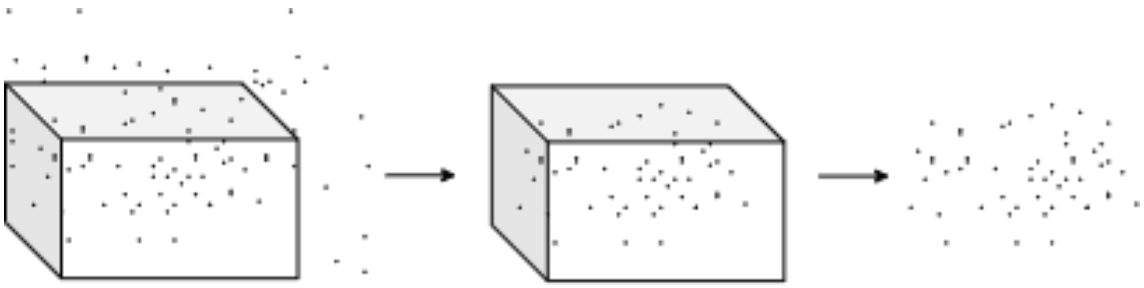


Figure 4.2: A visualization of the *contains* predicate in three steps. Firstly the annotation is projected on the point cloud. Secondly, points not contained by the annotation are removed. Thirdly, the projection of the annotation is removed and the resulting point cloud remains.

This output and by extension problem definition is chosen for three reasons. Firstly, without specialized 3D support, it is difficult to implement both in terms of efficiency and readability. Secondly, it varies in difficulty with the different annotation shapes, for example, axis-aligned cuboids are easier than concave shapes. Lastly, it is an interesting problem for a lot of different analyses within the space. For example, the accuracy of the perception model for a specific class could be measured by: (i)

filtering out all the volumetric annotations that are of that class; (ii) dividing the set of points into two sets, points that are contained and points that are not contained in the filtered annotations using the contains predicate, and finally, (iii) counting the number of contained points with a different class than the annotations and the number of points not contained with the same class as the annotations.

4.2 Evaluation Metrics

This section will define evaluation metrics that relate the problem definition from the previous section to the research questions introduced in 1.2 and formalize how proposed solutions are evaluated.

The research questions are focused on two things, computational efficiency, and expressive power. For a proposed solution to the problem defined in the previous section, computational efficiency is evaluated by implementing the idea, running it ten times, and computing the average time as well as the deviations for various cluster sizes. The evaluation in terms of computation time will also be related to whether the implementation supports all 3D shapes or some subset. For example, an efficient solution that can only handle axis-aligned cuboids is not very interesting if there exists a solution that is equally efficient but can handle concave shapes.

When evaluating expressive power of an entire domain, there is a large risk of the evaluation being opinionated. By clearly defining the input and output of the query, there are fewer possibilities for subjectivity to have an impact. Highly specialized operations are desirable and can be achieved either by implementing the operation as a UDF or by utilizing a Spark library that extends Spark SQL with the operation. For imported operations some data conversion is needed for the input data to fit the data types supported by the operation. In terms of expressive power, the data translation needed to enable usage of the imported operations needs to be easy to implement and understand. Providing data translation as UDFs in order to access specialized operations is likely to be more convenient and efficient than implementing the specialized operation on your own. This is due to libraries utilizing the internals of Spark and extending it to achieve optimized operations. However, the data types added by the library need to make sense and be easy to utilize. It is worth noting that expressive power is not valuable if the code is inefficient, as such these traits have to be related to the performance of the code. When discussing expressive power, it will be done from the perspective of someone with little to no Apache Spark experience and knowledge.

5

Approach 1 - Utilizing User-Defined Functions

This chapter describes how the approach of utilizing UDFs to encode a spatial predicate can be used to solve the task introduced in the previous chapter. As UDFs are very flexible, two variants of encoding the spatial predicate are developed and implemented. Both solutions use the same spatial filtering and an identical query. In order to understand and motivate the implementations and design choices, first the landscape and possibilities when utilizing UDFs in SparkSQL are presented and applied to the problem. This is coupled with the query used by both implementations. Secondly, the designs of the spatial filtering and the two versions of the spatial predicate are disclosed and justified. Finally, the implementations of the spatial filtering and the two versions of the spatial predicates are given.

5.1 Application

The query for this approach can be written with DataFrames in two ways. Either by performing operations supported by the relational, domain-specific language (DSL) on them or by registering them as tables to use in a SQL query. As both options perform equally and are optimized by Catalyst [4], the choice comes down to preference. For this implementation, SQL is chosen as we find it easier to explain the solution using SQL queries in the thesis.

SQL Query that counts all points inside any annotation

Prior to executing the query, the input data sets `lidar` and `annotation` are registered as DataFrames such that they can be accessed from the SQL context. The query below returns all points from the `lidar` DataFrame that are contained both spatially and temporally by at least one annotation from the `annotation` DataFrame

and includes extra clauses for spatial filtering:

```
SELECT count(*)
FROM lidar, annotation
WHERE lidar.time == annotation.time AND
      annotation.min_x < lidar.pointX AND
      lidar.pointX < annotation.max_x AND
      annotation.min_y < lidar.pointY AND
      lidar.pointY < annotation.max_y AND
      annotation.min_z < lidar.pointZ AND
      lidar.pointZ < annotation.max_z AND
      contains(annotation.volume, lidar.point)
```

For Spark to execute the query successfully, an implementation of the `contains` predicate is needed. By utilizing the UDF API, which allows users to define custom functions that Spark can utilize, `contains` can be implemented as a function. As the UDF API is available both in Spark and PySpark, UDFs provide access to external, non-spark libraries written in Java/Scala or Python. Furthermore, UDFs provide the user with the flexibility of defining the input and output parameters to fit their data types. These properties of UDFs, which make them easy to work with, are achieved by Spark treating UDFs as black boxes. However, this means Catalyst can not optimize UDFs during query optimization impacting performance negatively. As the UDFs are associated with performance constraints, minimizing the number of function calls to them can boost performance.

5.1.1 Axis-Aligned Spatial Filtering

The objective of the spatial filtering is not to solve the contains problem for all input combinations, but rather to remove combinations of points and annotations where the point is clearly not contained by the annotation. As such, one could argue that the filtering should be part of an optimized implementation of the contains predicate. However, by separating the spatial filtering to be performed outside of the UDF, Catalyst is able to consider the filtering when it performs query optimization.

To give an example of a point cloud with corresponding annotations for which there are points that are not spatially close to any annotation, see Figure 5.1. In the Figure, we see how a majority of the points are not in close proximity to either of the two annotations. Without spatial filtering, all combinations of points and annotations would have to be tried by the contains predicate to perform the evaluation whereas with spatial filtering the number of combinations can be reduced drastically.

For the spatial filtering, the extreme points of the annotation in the *X, Y,* and *Z* dimensions are used to create an axis-aligned cuboid spanning the extreme points of the annotation. A visualization of this for an axis-oriented cuboid annotation can be seen in Figure 5.2. The axis-aligned cuboid is then used for spatial filtering as any point not contained by it can not be contained by the annotation.

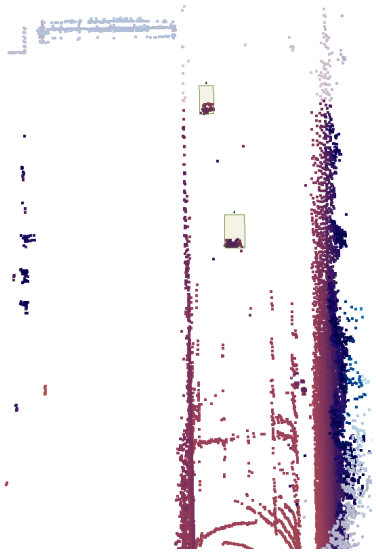
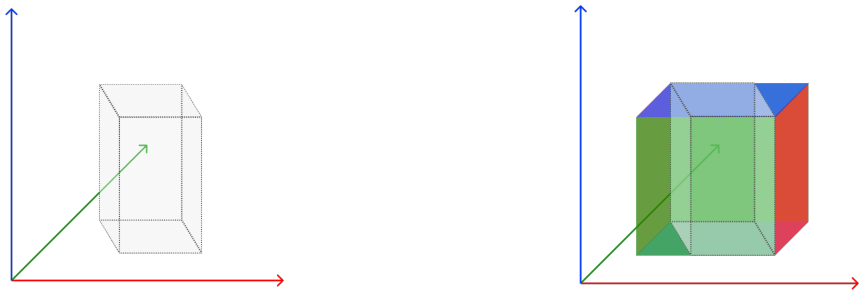


Figure 5.1: A cropped point cloud with two vehicle annotations.



(a) **(b)**

Figure 5.2: In (a) we see an axis-oriented cuboid annotation. In (b), the annotation from (a) is encapsulated by an axis-aligned cuboid that spans the extreme points of the annotation.

5.1.2 Point in Polyhedron Predicate

One way to implement the `contains` predicate for arbitrary polyhedrons and points is with the *even-odd* method [23]. The idea behind this is to create a ray originating in the point and to cast the ray in any direction. By determining whether the number of intersections between the ray and the surfaces of the polyhedron is even or odd, we know if the point is inside the volume or not. For the point to be inside the volume, the number of intersections must be odd. To understand why this is, assume that the point is inside the volume and that the volume is finite in size. Draw an infinite line starting at the point in any direction, eventually, the line exits the volume, when this happens the line has intersected the surface of the volume once. At this point, if the line intersects with the surface again the line re-enters the volume and the number of intersections is even. As the volume is finite, by continuing in the same direction, the line will eventually exit the volume again, resulting in an odd number of intersections. A 2D Visualization of this can be seen in Figure 5.3.

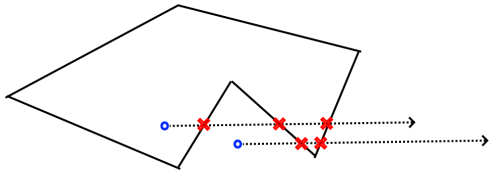


Figure 5.3: The rays and intersections of points inside and outside a concave polygon.

5.1.3 Point in Oriented Cuboid Predicate

For volumes represented as simpler shapes, there are other strategies for evaluating the predicate. One such approach can determine if a point is within an oriented cuboid.

To determine if the point is inside the cuboid, the point is projected onto an edge of the cuboid. If this projection is on the line between the end vertices of that edge the point is considered inside in that dimension. This property has to be tested true for each orthogonal dimension to be considered inside the entire cuboid.

First, the orthogonal edges $u = p_1 - p_2, v = p_1 - p_4, w = p_1 - p_5$ are retrieved. Checking whether the projection of a point x fall on these edges is done by calculating three scalar values. The origin: $u \cdot p_1$, the side length of u: $u \cdot p_2$ and the position of x on the edge u. If the position of x on the edge u is between the origin and the side length of u, the point x is inside the cube in that dimension. If this is true for u, v, and w, the point is is inside the box. A visualization of this for a point P and a point Q where P is contained by the annotation and Q is not can be seen in Figure 5.4.

Figure 5.4: In (a) we see a cuboid annotation with two points P and Q. In (b) we see how P and Q are located when projected on the lines u, v, and w.

5.2 Implementation

When making the choice between Python, Scala, and Java for the implementation of the UDF, two things are important to consider. Firstly, whether a library exists in one of these languages that will aid the developer in implementing a UDF. Secondly, in terms of performance whether the developer can afford the overhead costs that

come with PySpark. These overhead costs have previously been major [50], where input and output have been piped to the python environment via Unix pipes from the JVM and back again. The overhead cost of this process is relative to the amount of input data as well as the complexity of the function. For a function that is highly complex and only operates on small amounts of data, the cost might be negligible. However, if there are many small pieces of input data and a simple function, serialization is going to be a large portion of the cost of the UDF.

Later versions of Apache Spark have opted for utilizing Apache Arrow [1] as a language and environment-agnostic implementation to store and manipulate columnar data in memory. This Spark API is referred to as PandasUDFs or vectorized UDFs, as operations can be made on batches of items in a column instead of rows as is the case with regular UDFs. Performance-wise, vectorized UDFs suffer less from I/O overhead and are therefore more efficient compared to the earlier Python UDF API.

5.2.1 Axis-Aligned Spatial Filtering

The annotations in the input data are represented spatially as vertices and faces. As faces only combine vertices, the spatial x, y, and z values are contained in the vertices. As such the vertices are used to find the extreme points of the annotation using the `min` and `max` functions which given a field finds the minimum/maximum value for that field. The minimum and maximum values need to be found for each individual annotation and for all three spatial dimensions. This is achieved with the following SQL query which creates new columns to hold the extreme values:

Annotation Preprocessing

```
SELECT collect_list(
             struct(annotation.VertexID
                    annotation.Vertex_X,
                    annotation.Vertex_Y,
                    annotation.Vertex_z)) as Vertices,
       collect_list(
             struct(annotation.Vertex1,
                    annotation.Vertex2,
                    annotation.Vertex3)) as Faces,
       min(annotation.Vertex_X) as min_x,
       max(annotation.Vertex_X) as max_x,
       min(annotation.Vertex_Y) as min_y,
       max(annotation.Vertex_Y) as max_y,
       min(annotation.Vertex_Z) as min_z,
       max(annotation.Vertex_Z) as max_z,
FROM annotation
GROUP BY timestamp, id
```

This is how the `min_{x,y,z}` and `max_{x,y,z}` columns used in the query in Section 5.1 are created.

5.2.2 Point in Polyhedron Predicate

In Scala, there is a 3D geometry library, Apache Commons Geometry, which provides data types and functions for working with geometries and that supports 3D. This library also provides raycasting functionality which can be used as an efficient solution to determine whether a point is contained by a concave 3D geometry using the even-odd method [23]. The pseudocode for this algorithm is given in Algorithm 1.

Algorithm 1: Point in concave shape - Even-odd Test

Input: Faces: fs, Point: x
Output: Boolean
define ray with origin x and arbitrary direction;
foreach *face in fs* **do**
  if *ray intersects face* **then**
    count intersection;
end
if *the sum of intersections is odd* **then**
  return *true*;
else
  return *false*;
end

The weight of this approach lies on the *intersects* predicate which is evaluated on all faces of the polyhedron to determine whether the number of intersections is even or odd.

5.2.3 Point in Oriented Cuboid Predicate

For the implementation of the oriented cuboid predicate, the library from the previous section can be used as it also brings support for vector types and linear algebra operations. The pseudocode for this algorithm is given in Algorithm 2. It is worth noting that for the input data used in this thesis, the vertices in the annotation are

ordered such that the orthogonal vertices always appear at the same indices.

Algorithm 2: Point in cuboid - Side Projection Test

Input: Vertices: vs, Point: x
Output: Boolean
$p_1 \leftarrow$ arbitrary vertex in vs;
$(p_2, p_4, p_5) \leftarrow$ get orthogonal vertices of p_1;
$u \leftarrow p_1 - p_2$;
$v \leftarrow p_1 - p_4$;
$w \leftarrow p_1 - p_5$;
if $u \cdot x$ *is between* $u \cdot p_1$ *and* $u \cdot p_2$ **then**
  if $v \cdot x$ *is between* $v \cdot p_1$ *and* $v \cdot p_4$ **then**
    if $w \cdot x$ *is between* $w \cdot p_1$ *and* $w \cdot p_5$ **then**
      return *true*
end
return *false*

6

Approach 2 - Utilizing Sedona

This chapter describes a solution to the task introduced in Chapter 4 using the Sedona library. First, the features from the library which are utilized in the implementation are presented together with their strengths and weaknesses. Then, the implementation choices with respect to the features are disclosed and justified. This is coupled with the final implementation of the query and the surrounding code needed to enable it.

6.1 Application

While Sedona provides the desired predicate, `ST_Contains`, it only supports 2D geometries and operations whereas the problem from Chapter 4 requires a 3D contains solution. Furthermore, Sedona is a spatial library and does not support spatio-temporal objects. As such the `ST_Contains` predicate does not verify that the two geometries co-exist temporally. This section explains a methodology for solving the 3D spatio-temporal contains problem using the 2D spatial `ST_Contains` operation. The methodology only supports cuboid shapes which are rotated in one axis, as devising a methodology for solving the 3D contains problem for concave shapes using a 2D contains operation is not feasible.

Given a 3D annotation and a 3D point, the 3D contains operation can be replaced by three 2D contains operations. For the annotation, create three 2D planes XY, XZ, and YZ, and verify that the point lies on these planes, disregarding the dimension that is orthogonal to the plane. A visualization of this can be seen in Figure 6.1.

This idea needs to be combined with a temporal check in order to solve the problem presented in Chapter 4, resulting in the following SQL code:

SQL Query that counts all points inside any annotation

```
SELECT count(*)
FROM annotation, lidar
WHERE annotation.timestamp == lidar.timestamp
    and ST_Contains(annotation.XY, ST_Point(lidar.X, lidar.Y))
    and ST_Contains(annotation.XZ, ST_Point(lidar.X, lidar.Z))
    and ST_Contains(annotation.YZ, ST_Point(lidar.Y, lidar.Z))
```

For this query, the predicate is already provided by Sedona and does not need to

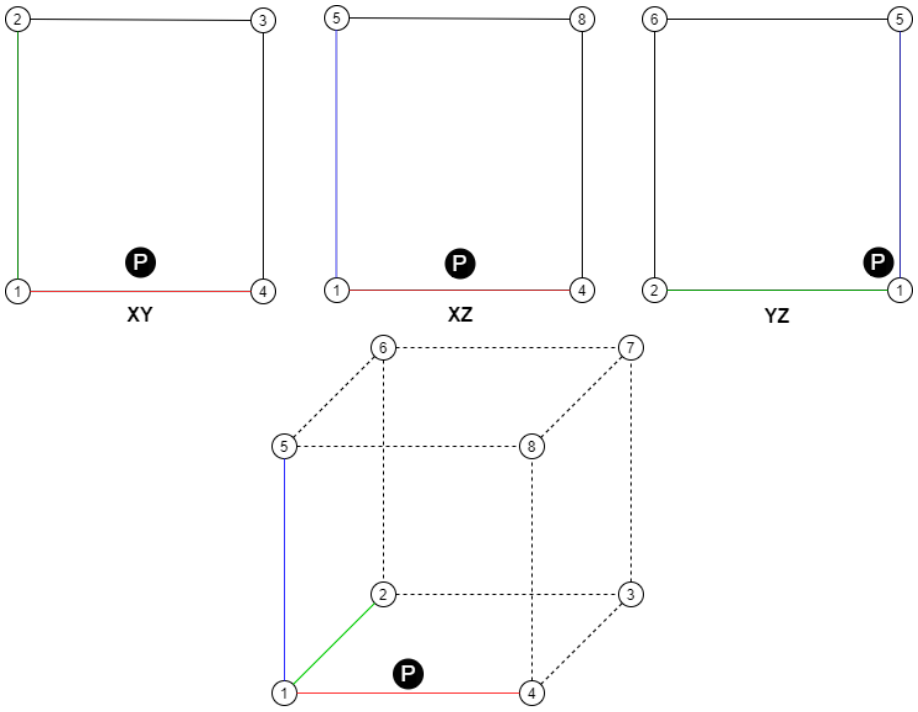


Figure 6.1: The three different 2D planes (XY, XZ, and YZ) of the same 3D cuboid, all containing the point p. As such, the cuboid contains p.

be implemented by the developer as opposed to in the previous Chapter. Not only is this convenient but more importantly, Sedona extends Catalyst with new rules enabling Catalyst to optimize all parts of the query.

In order to enable the query some data preprocessing and type conversions are needed. The `ST_Contains` predicate takes two input *geometries*. As such, to utilize the predicate the input data needs to be converted to a geometry type. As can be seen in the query, this is straightforward for the points in the point cloud with the `ST_Point` constructor. However, the three 2D planes of the annotations need to be represented as geometries. In order to reason about the overhead costs of the conversion and how difficult it is to translate the input data to data types compatible with Sedona, the next section will detail the implementation and data pre-processing needed to enable the query.

6.2 Implementation

To utilize the spatial predicates and functions provided by Sedona, the input data is converted to geometries. A polygon, a subclass of geometry, can be created from a string and a delimiter with the `ST_PolygonFromText` constructor. In order to use the `ST_PolygonFromText` constructor we need a function which outputs a text representation of a polygon given 2D vertices representing a plane of an annotation. For this a User-Defined Aggregate Function (UDAF) is defined as: `toWKT(v_dim1, v_dim2): String`. Using this UDAF the following SQL code creates three text

representations `text_XY`, `text_XZ`, and `text_YZ` of polygons, one for each plane:

```
SELECT toWKT(
        annotation.Vertex_X, annotation.Vertex_Y) as text_XY,
    toWKT(
        annotation.Vertex_X, annotation.Vertex_Z) as text_XZ,
    toWKT(
        annotation.Vertex_Y, annotation.Vertex_Z) as text_YZ
FROM annotation
GROUP BY timestamp, id
```

With the text representations of polygons for all planes, the polygon objects can be created using the `ST_PolygonFromText` constructor. It should be noted that the constructor inserts edges between the vertices in the order they appear in the text and the input annotation data is ordered differently. This is remedied by using the `ST_ConvexHull` function which given a geometry returns the convex hull of that geometry. While this would not correctly represent concave annotations, it correctly represents both cuboid and convex annotations which is what this implementation supports. As such, the following SQL code prepares the three polygons `XY`, `XZ`, and `YZ` used in the final query, presented in the previous section:

```
SELECT *,
    ST_ConvexHull(ST_PolygonFromText(text_XY, ",")) as XY,
    ST_ConvexHull(ST_PolygonFromText(text_XZ, ",")) as XZ,
    ST_ConvexHull(ST_PolygonFromText(text_YZ, ",")) as YZ
FROM annotation
```

7

Approach 3 - Utilizing STARK

This chapter describes a solution to the task introduced in Chapter 4 using the STARK library, following the same structure as the previous. First, the features from the library which are utilized in the implementation are presented together with their strengths and weaknesses. Then, the implementation choices with respect to the features available in STARK are disclosed and justified. This is coupled with the final implementation of the query and the surrounding code needed to enable it.

7.1 Application

STARK, similarly to Sedona provides an implementation of the contains predicate, `ST_Contains`. While neither library supports 3D, which is the problem space, STARK is a spatio-temporal library and as such the predicate verifies that the object is contained both spatially and temporally. In Section 6.1 we presented a methodology for solving the 3D contains problem using three 2D contains operations, a similar methodology will be applied here. However, here the temporal check is included in the predicate, resulting in the following SQL code:

SQL Query that counts all points inside any annotation

```
SELECT count(*)
FROM annotation, lidar
WHERE ST_Contains(annotation.ST_Geom_XY,
        toSTObject(lidar.X, lidar.Y, lidar.timestamp))
    and ST_Contains(annotation.ST_Geom_XZ,
        toSTObject(lidar.X, lidar.Z, lidar.timestamp))
    and ST_Contains(annotation.ST_Geom_YZ,
        toSTObject(lidar.Y, lidar.Z, lidar.timestamp))
```

One of STARK's strengths is that it extends RDDs with spatio-temporal partitioning and provides spatio-temporal indexing techniques for the contents of the partitions. While STARK does extend Catalyst, it does not extend Catalyst with rules to dynamically use spatial partitioning or indexing. As this thesis focuses on ease of use and queries as opposed to manually applying operations directly on RDDs, the implementation in this Chapter will not include any spatial partitioning or indexing. While the performance of the `ST_Contains` predicate still benefits from the additions

to Catalyst the potential benefits of partitioning and indexing will not be evaluated. It is important to note that STARK does not support versions 3.x of Spark. As such, this implementation will not have access to *Adaptive Query Execution* (AQE), a feature added in Spark 3.0 which allows for continuous optimization during query execution [51].

The query is enabled by type conversions and preprocessing of the input data. The `ST_Contains` predicate takes two input `ST_Object` (**S**patio-**T**emporal Object). For points, these are created using the `toSTObject` UDF we defined as seen in the query. This UDF uses the `STObject(double, double, long)` constructor. For annotations, similarly to the Sedona implementation, it is not as straightforward and the next section will outline implementation details, such as type conversions, enabling a better understanding of the overhead costs.

7.2 Implementation

An `ST_Object` can also be created using the `ST_Object(string, long)` constructor which takes a string representing a geometry and a long representing a timestamp. The string should have the Well-Known Text (WKT) format, standardised by the OGC [10]. To create WKT representations of 2D polygons for the annotations, a UDAF is defined as: `toWKT(v_dim1, v_dim2): String`. Using this UDAF the following SQL code creates three WKT representations `Poly_WKT_XY`, `Poly_WKT_XZ`, and `Poly_WKT_YZ` of polygons, one for each plane:

```
SELECT toWKT(
        annotation.Vertex_X, annotation.Vertex_Y) as Poly_WKT_XY,
    toWKT(
        annotation.Vertex_X, annotation.Vertex_Z) as Poly_WKT_XZ,
    toWKT(
        annotation.Vertex_Y, annotation.Vertex_Z) as Poly_WKT_YZ
FROM annotation
GROUP BY timestamp, id
```

The WKT strings representing polygons are now combined with a timestamp to create an `ST_Object` via a `fromWKTandTimestamp` UDF - which utilizes the `ST_Object(string, long)` constructor - as can be seen in the following SQL:

```
SELECT fromWKTandTimestamp(
        annotation.Poly_WKT_XY, annotation.timestamp) as ST_Geom_XY,
    fromWKTandTimestamp(
        annotation.Poly_WKT_XZ, annotation.timestamp) as ST_Geom_XZ,
    fromWKTandTimestamp(
        annotation.Poly_WKT_YZ, annotation.timestamp) as ST_Geom_YZ
FROM annotation
GROUP BY timestamp, id
```

With this SQL code, all of the objects needed to enable the query from the previous section are created finalizing the implementation for this Chapter.

8

Evaluation

In Chapter 4 our problem was defined together with metrics for which solutions to the problem should be evaluated. After that four implementations which solve the problem using three different strategies were presented. This chapter first introduces the evaluation environment. Then the optimized physical execution plans output by Catalyst are presented and briefly analyzed for all implementations, providing context for the results. After that, the four implementations are evaluated on computation clusters of different sizes. A cluster consists of a master and a number of worker nodes and by evaluating the computation time for different numbers of worker nodes we find how well the different implementations scale with the available computing resources. After the evaluations, the results will be discussed. This discussion relates the results to the scope of the thesis, the problem definition, and the execution plans to understand and contextualize the results.

8.1 Evaluation setup

To create the different clusters needed for the evaluations we use Dataproc in Google Cloud Platform. Both the master and worker nodes have the *n1-standard-4* machine type[19] which has 15GB of RAM and 4 CPU cores. As for the number of worker nodes, six different cluster sizes are used; 0, 2, 4, 8, 16, and 32 workers. When there are 0 workers the master node takes the role of a worker. The choice of both machine type and cluster size is important in a scalability test. For example, compare having access to two machines with 128GB of RAM and 32 CPU cores each to having access to sixteen machines with 16GB of RAM and 4 CPU cores each. In total, the available computing power is the same but running the implementations on two machines is likely to be cheaper as a result of performing I/O operations locally. However, the sixteen less powerful machines are cheaper than the two powerful machines. Furthermore, from a scalability analysis, one could find that having sixteen machines is redundant as the runtime is similar for twelve machines. As such, the scalability analysis helps enable the trade-off between cost (number of machines and machine type) and results (computation time or other metrics). In that sense, our scalability analysis is to some extent, incomplete as it only considers one machine type.

Experiments will be presented using the size of the cluster and the name for the implementation, as shown in Table 8.1. For example, an evaluation result for the experiment where the cuboid implementation from Chapter 5 runs on a cluster with

4 workers and spatial filtering enabled will be presented as *cuboid-filter-4*. Evaluation results for the experiment where the ray casting implementation runs on a cluster with 32 workers and without spatial filtering are presented as *raycast-32* and so forth.

Approach	**Shape**	**Filtering**	**Name**
1	Concave	No	raycast
1	Concave	Yes	raycast-filter
1	Cuboid	No	cuboid
1	Cuboid	Yes	cuboid-filter
2	Convex	N/A	sedona
3	Convex	N/A	stark

Table 8.1: List of Experiments.

The master node and the worker nodes run the same image. For the evaluations of the implementations in Chapter 5 and Chapter 6 the *2.0.9-debian10* image is used. For the evaluations of the implementation in Chapter 7 image *1.5.35-debian10* is used as STARK does not support Spark 3.x. The versions of relevant software for the images are listed in Table 8.2.

Name	1.5.35-debian10	2.0.9-debian10
Debian	v10	v10
Spark	v2.4.7	v3.1
Scala	v2.12.10	v2.12.13

Table 8.2: Software versions for images 1.5.35-debian10 and 2.0.9-debian10.

The input data files are stored in Google Cloud Storage (GCS) in a bucket that is located in the same region as the clusters. The input data for the evaluations consist of 5 minutes and 19 seconds long LiDAR recording and corresponding annotations. The input data consists of 362,164,330 LiDAR points and 23008 cuboid annotations, which are measured and created over 12179 timestamps. All implementations are tested on cuboid annotations to ensure consistency in the input for the different implementations and evaluations.

All the implementations are written in Scala, built as JAR files, and uploaded to GCS.

8.1.1 Query Execution Plans

While the individual Spark SQL queries are presented in Chapters 5-7, the optimized physical plans created by Catalyst are presented here.

Query Plan for Approach 1: UDF without Spatial Filtering

```
== Optimized Physical Plan ==
+- HashAggregate keys={} function=count()
   +- Exchange SinglePartition
      +- HashAggregate keys={} function=partial_count()
         +- Project
            +- BroadcastHashJoin keys={timestamp}, BuildRight UDF()
               :- Project
               :  +- Filter not_null(timestamp)
               :     +- FileScan LIDAR*.parquet
               +- BroadcastExchange HashedRelationBroadcastMode
                  +- Filter not_null(timestamp)
                     +- ObjectHashAggregate
                            keys={timestamp, MeshID}
                            function=collect_list(faces, vertices)
                        +- Exchange
                           +- ObjectHashAggregate
                                keys={timestamp, MeshID}
                                function=partial_collect_list(faces,
                                                              vertices)
                              +- Filter not_null(vertex)
                                 +- FileScan ANNOTATION*.parquet
```

This plan consists of five phases with three shuffles, called `exchanges` in the physical plan. The first exchange from the bottom is part of the aggregation of mesh faces and vertices. The aggregation begins with local aggregation before an exchange is made based on the aggregation key and is finished by aggregation at the beginning of the second phase. This produces the left relation of the join which is performed physically with a broadcast join. This join strategy is applied if the physical size of the input relations is below the default configuration value of 10MB. The smaller relation is broadcast to all workers and is applied as a map function, referred to as a map-side join. This specific broadcast join is a `BroadcastHashJoin`, where the hashed value of the timestamp is used to join tuples before any consecutive predicate, in this case, the UDF, is evaluated.

Query Plan for Approach 1: UDF with Spatial Filtering

```
== Optimized Physical Plan ==
+- HashAggregate count()
   +- Exchange SinglePartition
      +- HashAggregate partial_count()
         +- Project
            +- BroadcastHashJoin on timestamp, BuildRight
                    min_x < lidar.pointX &
                    max_x > lidar.pointX &
                    min_y < lidar.pointY &
                    max_y > lidar.pointY &
                    min_z < lidar.pointZ &
                    max_z > lidar.pointZ &
                    UDF()
               :- Project
               :  +- Filter not_null(timestamp)
               :     +- FileScan LIDAR*.parquet
               +- BroadcastExchange HashedRelationBroadcastMode
                  +- Filter not_null(timestamp)
                     +- ObjectHashAggregate collect_list(mesh)
                        +- Exchange
                           +- ObjectHashAggregate collect_list(mesh)
                                 +- FileScan ANNOTATION*.parquet
```

While more predicates are added for the axis-aligned spatial filtering, this has no effect on the plan other than that the predicates are included. The additional predicates are appended to the `BroadcastHashJoin` to be evaluated before the UDF, reducing the number of UDF calls but not the number of hash comparisons.

Query Plan for Approach 2: Apache Sedona

```
== Optimized Physical Plan ==
+- HashAggregate keys={} function=count()
   +- Exchange SinglePartition
      +- HashAggregate keys={} function=partial_count()
         +- Project
            +- BroadcastHashJoin keys={timestamp}, BuildLeft
               :        ST_Contains AND ST_Contains AND ST_Contains
               :- BroadcastExchange HashedRelationBroadcastMode
               :  +- ObjectHashAggregate keys={timestamp, MeshID}
               :     :                    function=toWKT()
               :     +- Exchange hashpartitioning
               :        :                 keys={timestamp, MeshID}
               :        +- ObjectHashAggregate
               :           :              keys={timestamp, MeshID}
               :           :              function=partial_toWKT()
               :           +- Filter isnotnull(timestamp)
               :              +- FileScan ANNOTATION*.parquet
               +- Project
                  +- Filter not_null(timestamp)
                     +- FileScan parquet LiDAR*.parquet
```

While Sedona implements strategies for spatial joins, the optimized execution plan above shows how the implemented query is optimized with the same join strategy, `BroadcastHashJoin`, as with Approach 1.

Query Plan for Approach 3: STARK

```
== Optimized Physical Plan ==
+- HashAggregate keys={}, function=count()
   +- Exchange SinglePartition
      +- HashAggregate keys={}, function=partial_count()
         +- Project
            +- BroadcastNestedLoopJoin BuildLeft, Inner,
               :        ((stcontains(Polygon_XY, xy) AND
               :         stcontains(Polygon_XZ, xz)) AND
               :         stcontains(Polygon_YZ, yz))
               :- BroadcastExchange IdentityBroadcastMode
               :  +- SortAggregate keys={timestamp, MeshID},
               :     :             function=polygonaggregator()
               :     +- Sort [timestamp ASC, MeshID ASC]
               :        +- Exchange hashpartitioning
               :           +- SortAggregate keys={timestamp, MeshID}
               :              :         function=polygonaggregator()
               :              +- Sort [timestamp ASC, MeshID ASC]
               :                 +- FileScan parquet [Annotations]
               +- Project
                  +- FileScan parquet [LiDAR]
```

While STARK implements many indexing and partitioning techniques, none of these are utilized in the execution plan. The execution plan includes the same five phases as previous queries but instead of a `BroadcastHashJoin`, the less efficient join strategy, `BroadcastNestedLoopJoin`, is used. This is due to the `ST_Contains` predicate containing both the spatial and temporal predicates, leaving no equality condition outside the UDF for Catalyst to optimize the join with.

8.2 Scalability analysis

The scalability analysis was performed by measuring the runtime of jobs submitted to the various clusters. The jobs referenced JAR files in GCS containing the various implementations and ten jobs were submitted for each combination of experiment and cluster size. The runtimes of the jobs are aggregated and plotted as boxplots. Furthermore, for each implementation, we will present how many seconds of recorded data it processes per second, on average, on a cluster with eight workers.

To test the implementations for correctness, the Sedona implementation is used as a baseline for correct output. This is because the Sedona library includes tests for the 2D contains predicate. By comparing the output of the other implementations to the output of the implementation that uses Sedona, we can determine whether the implementations are correct. When comparing the output between the implementations, the cuboid UDF and the ray casting UDF, they find the exact same set of 99137 point-annotation pairs. The Sedona implementation finds 99379 pairs and overlaps the pairs found by the UDF implementations entirely. We sampled and

visualized the additional pairs found by the Sedona library and verified that it was impossible to tell whether the point was contained by the annotation or not. The differences between the implementations come down to floating-point precision for edge cases.

8.2.1 Summary

To provide an overview of how the different experiments relate to each other in terms of performance, the results of the experiments running on a cluster with eight workers are presented in Figure 8.1. The STARK implementation is not included in the graph as it was unable to complete within an hour. The cuboid implementation when spatial filtering is enabled is the fastest implementation, however, it can only handle cuboid annotations. The ray cast implementation with spatial filtering enabled performs similarly and supports all annotation shapes. While the Sedona library is faster than both UDF implementations when they have spatial filtering disabled, it is outperformed by both UDF implementations when they have spatial filtering enabled.

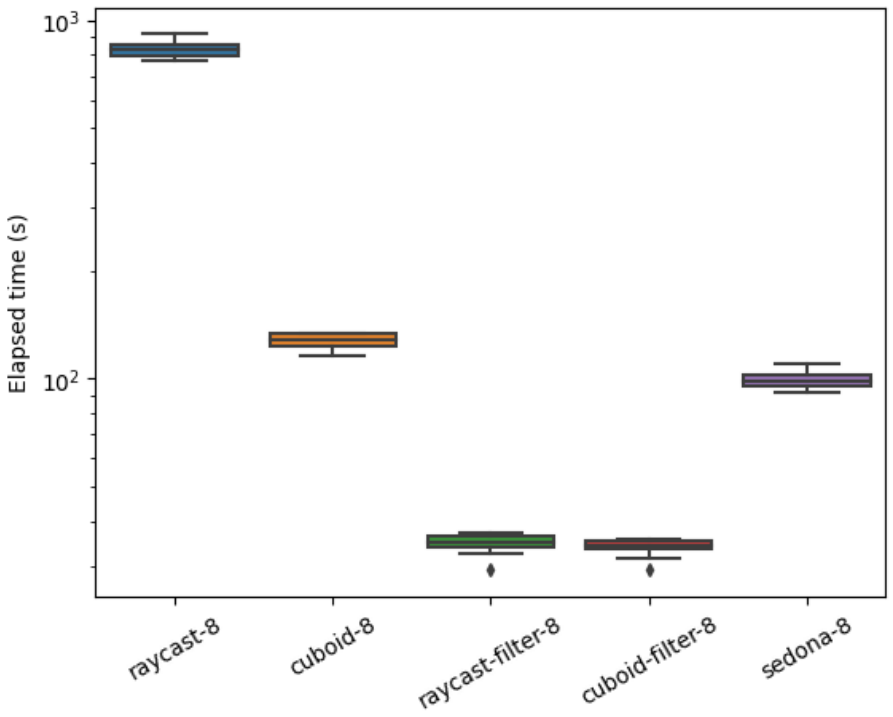


Figure 8.1: The runtime results for all different experiments on a cluster with eight workers. Note that the vertical axis is logarithmic.

As will be shown in the coming sections, none of the experiments achieved significant speedups as the cluster size increased beyond 8 workers. The lack of scalability observed in the experiments will be discussed and elaborated on in Section 8.3 whereas the coming sections will detail the evaluation results and scalability of each experiment.

8.2.2 Ray Casting UDF Implementation

The ray casting implementation as detailed in Chapter 5 is the only implementation that supports concave annotation shapes. For this implementation, there are two different experiments, one where the spatial filtering described in Section 5.1.1 is disabled and one where it is enabled.

Spatial filtering disabled

The evaluation results for this implementation with filtering disabled can be seen in Figure 8.2. On average, this implementation processed 0.4 seconds of the recorded input data per second with 8 worker nodes. The speedup from 0 workers to 2 workers is close to 2x. Furthermore, the speedup between the cluster with 2 workers and the cluster with 4 workers is 1.95x which is very efficient given that the computing resources are doubled. However this speedup reduces drastically for the remaining clusters as the speedup from 4 to 8 workers is 1.40x, 8 to 16 is 1.12x and 16 to 32 is 1.02x. The final speedup result suggests that simply increasing the cluster size further will not significantly improve the runtime.

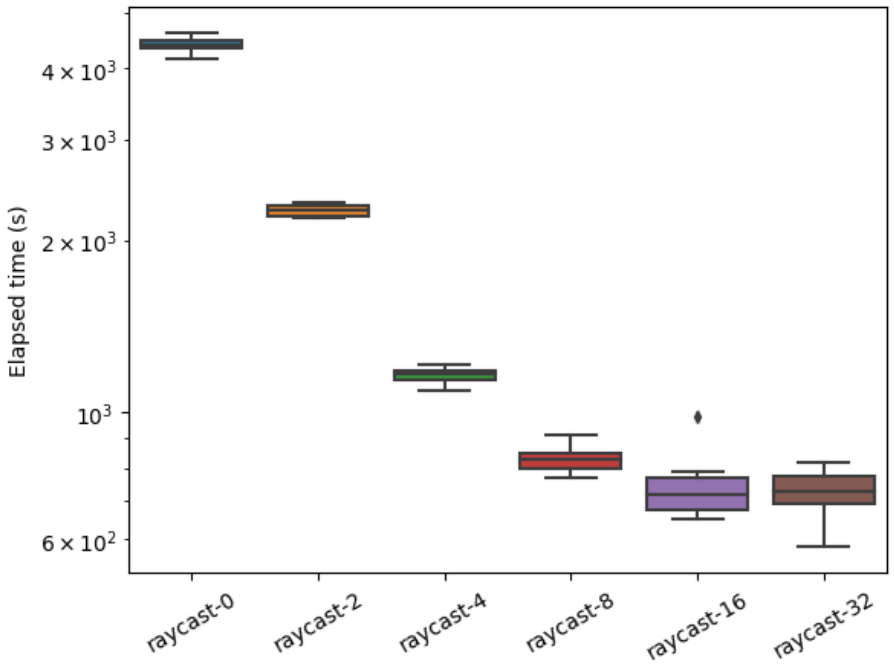


Figure 8.2: Scalability test for the raycast implementation on the six different cluster sizes with spatial filtering disabled. Note that the vertical axis is logarithmic.

Spatial filtering enabled

The results with filtering enabled can be seen in Figure 8.3. Most noticeably, enabling the spatial filtering results in a 78.6x speedup for the cluster with 0 workers and more than a 20x speedup across all cluster sizes. This implementation, on average, processed 9.2 seconds of the recorded input data per second for 8 workers. Unlike when spatial filtering is disabled, this implementation does not achieve a

significant speedup between any consecutive cluster sizes. The largest speedup observed is between 0 and 2 workers and it is only 1.34x. Furthermore, the speedup drastically decreases as it is 1.11x between 2 and 4 workers and 1.09x between 4 and 8 workers. This suggests that simply increasing the cluster size does not significantly improve the runtime.

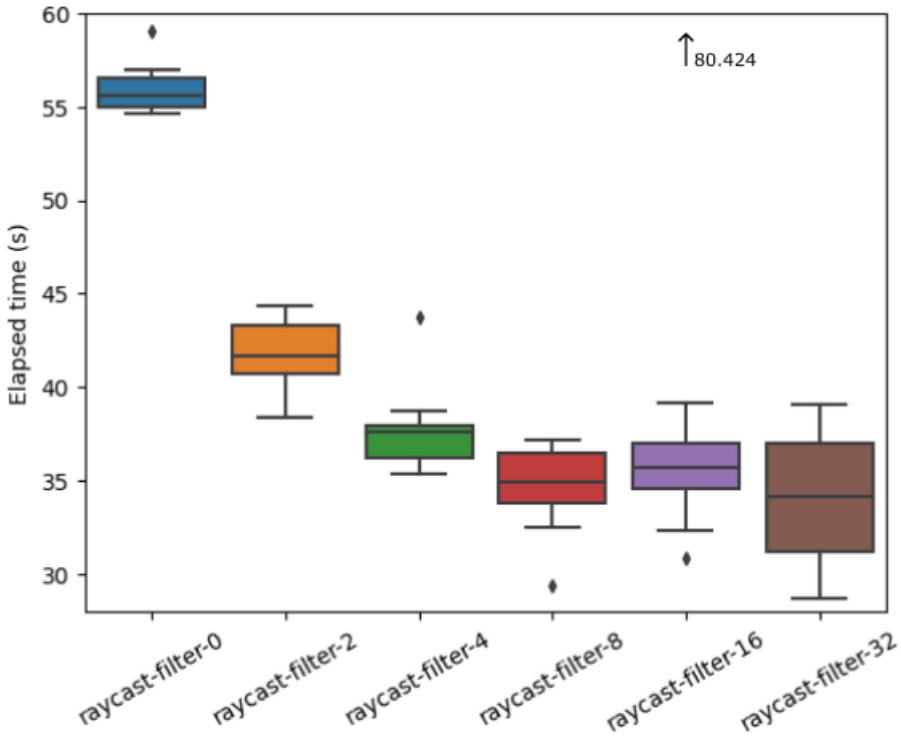


Figure 8.3: Scalability test for the raycast implementation on the six different cluster sizes with spatial filtering enabled. Note the outlier for the medium sized cluster denoted with an arrow

8.2.3 Cuboid UDF Implementation

This implementation uses linear algebra and UDFs to support cuboid shapes as detailed in Chapter 5. Similar to the previous implementation, there are two different experiments, one where spatial filtering is enabled and one where it is disabled.

Spatial filtering disabled

The evaluation results for this implementation with filtering disabled can be seen in Figure 8.4. This implementation, on average, processed 2.5 seconds of the recorded input data per second when running on the cluster with 8 workers. Similar to the ray casting implementation, the speedup is significant between 2 and 4 workers at 1.76x. This implementation scales well for the next cluster size as well as the speedup between the cluster with 4 and the cluster with 8 workers is 2.67x. However, the drop-off is significant after this as the speedup is 1.08x between 8 and 16 workers,

and 1.03x between 16 and 32 workers. The final speedup result suggests that simply increasing the cluster size further will not significantly improve the runtime.

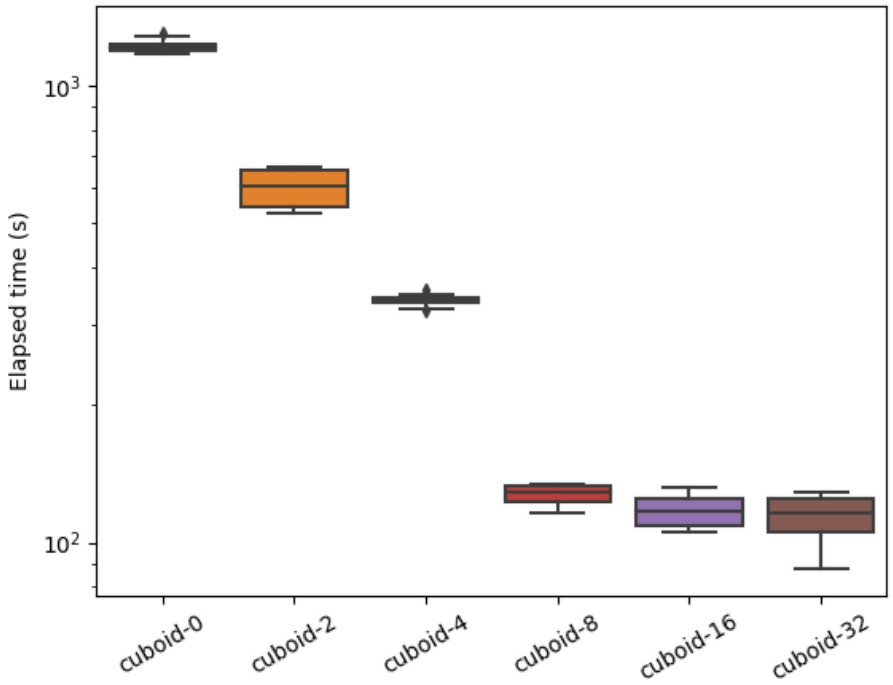


Figure 8.4: Scalability test for the cuboid implementation on the six different cluster sizes with spatial filtering disabled. Note that the vertical axis is logarithmic.

Spatial filtering enabled

The results with filtering enabled can be seen in Figure 8.5. Enabling the spatial filtering results in more than a 24x speedup for the cluster with 0 workers and more than a 3x speedup across all cluster sizes. On average, this implementation processed 9.4 seconds of recorded data per second using 8 workers with the spatial filter enabled. Similar to the experiment for the ray casting implementation with filtering enabled, this implementation does not scale well with the cluster size. The speedup between 0 and 2 workers is noticeable at 1.32x. However, the best speedup between two consecutive cluster sizes beyond this is only 1.08x. This suggests that simply increasing the cluster size does not significantly improve the runtime.

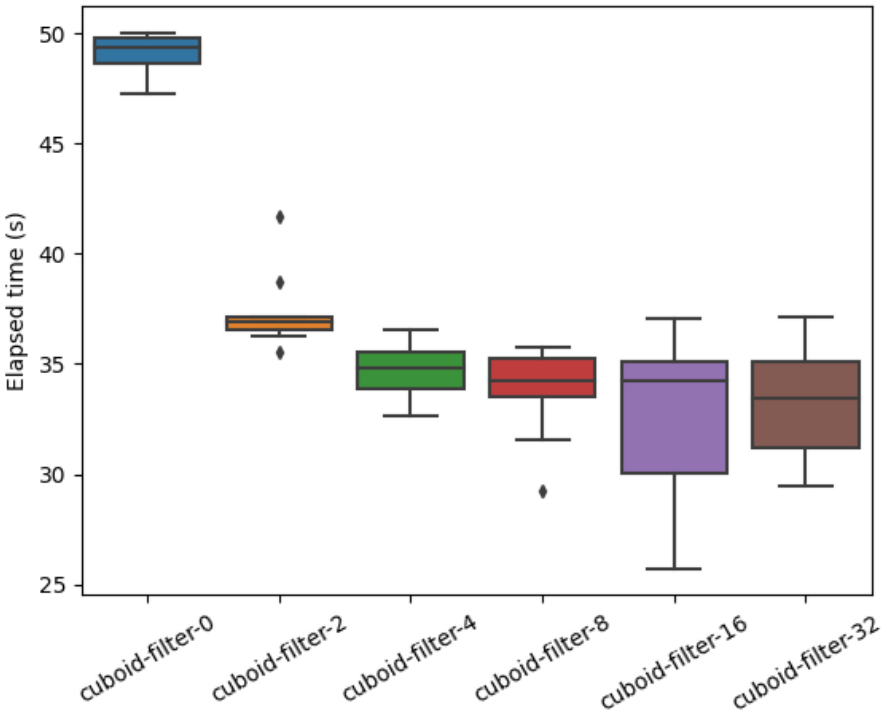


Figure 8.5: Scalability test for the cuboid implementation on the six different cluster sizes with spatial filtering enabled.

8.2.4 Sedona Implementation

The implementation using the Sedona library, presented in Chapter 6, supports cuboid and convex shapes. The results for this implementation can be seen in Figure 8.6. This implementation processed 3.2 seconds of the recorded input data per second when running on the cluster with 8 workers. The speedup is above 1.8x both between 0 and 2 as well as 2 and 4 workers. The speedup is noticeable between 4 and 8 as well at 1.37x. However, for larger cluster sizes it is not significant. Between 8 and 16 it is 1.09x and between 16 and 32 it is 1.04x. The final speedup result suggests that simply increasing the cluster size further will not significantly improve the runtime.

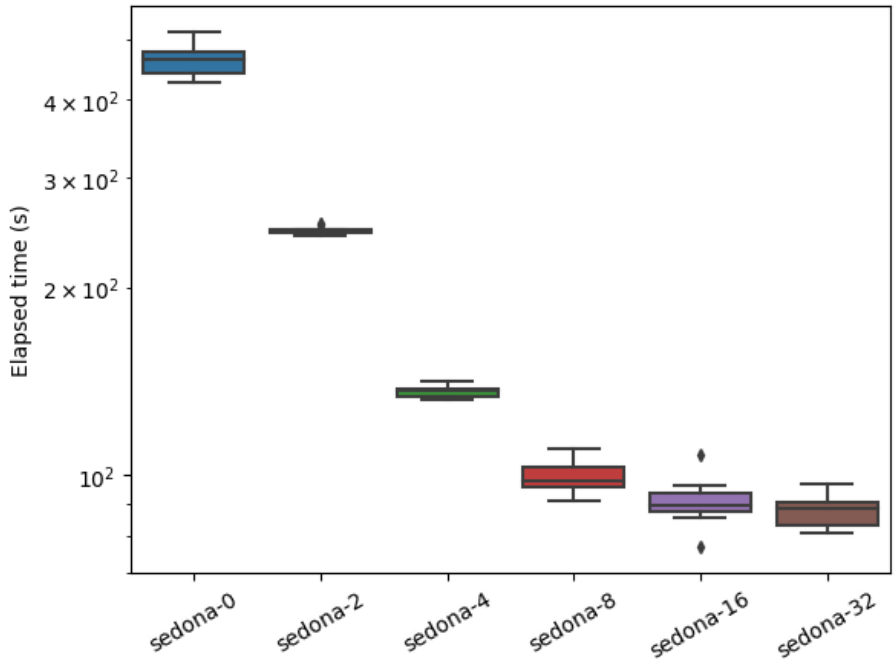


Figure 8.6: Scalability test for the implementation using Sedona on the six different cluster sizes. Note that the vertical axis is logarithmic.

8.2.5 STARK Implementation

The STARK implementation was unable to complete its execution within an hour for all of the cluster sizes, at which point we intervened and stopped the jobs. Given that the slowest non-STARK implementation completed the job in 15m14s there was no point in letting the job continue past the hour-mark. As such there are no runtime results to report from the STARK implementation other than that it did not successfully complete the task within an hour.

8.3 Discussion

The results of the evaluation show that the two approaches using libraries were not as efficient as the UDF implementations of the 3D spatio-temporal predicate. Furthermore, we found that the queries utilizing UDFs were not less comprehensible compared to those which utilize libraries. In this section, these findings are elaborated for the narrowly-scoped problem and put in the context of the wider scope of the research questions. After that, the relation of these results to previous research results exploring spatial data in Spark is discussed. Then, the execution plans for the different implementations are discussed to provide an understanding of why the implementations perform differently. This discussion helps summarize the key concepts of the query execution and how they relate to the observed performance. Finally, the impact of choices made during problem definition, library selection, and query definition are discussed.

8.3.1 Why are Spatial Libraries Slower?

In contrast to the findings in evaluations by the authors of Sedona [56] and STARK [21], these results show no performance improvement over the native approach for spatial and temporal predicates. One of the major differences between this evaluation and those cited is that the spatial join query is performed with annotated LiDAR data. Since the LiDAR table is orders of magnitude larger than the annotations table, a map-side join is performed through broadcasting the annotations table in `BroadcastHashJoin`. As the optimization strategies implemented by Sedona optimize reduce-side joins for when this is not possible, these optimizations do not accelerate the evaluated query.

Another thing to note is that the amount of data available to Catalyst can be more important than letting the library perform all evaluations. The initial suitability hypothesis focused on matching supported data types, where a lack of 3D spatio-temporal data types was of concern. As observed in STARK, which supports 2D spatio-temporal data types, this hides more of the predicate logic away from Catalyst, reducing the available optimization strategies. In this case, the join is evaluated with a `BroadcastNestedLoopJoin` instead. This has a complexity several orders of magnitude higher, leading to the analysis not completing successfully.

8.3.2 What major factors impact efficiency?

One major factor in the overall performance is the number of contains tests performed. The worst-case here is to test every point with every volume,

$$points \times volumes$$

which yields

$$3.621 \cdot 10^8 \times 2.301 \cdot 10^4 = 8.333 \cdot 10^{12}$$

comparisons. As the temporal predicate is separate from the spatial, Spark manages to reduce the number of computations by performing a `BroadcastHashJoin`. This join strategy broadcasts the smaller dataset to all partitions of the larger dataset. The HashJoin is then performed locally by joining on equal timestamps using a HashSet. While there is an additional overhead cost of hashing, the UDF is only called once for each point that has the same timestamp as a volume. This number of comparisons can be estimated from the number of points, the number of volumes, and the number of timestamps using the following formula.

$$\frac{points}{timestamp} \times \frac{volumes}{timestamp} \times timestamp$$

For our dataset, this yields

$$\frac{3.621 \cdot 10^8}{1.349 \cdot 10^4} \times \frac{2.301 \cdot 10^4}{1.349 \cdot 10^4} \times 1.349 \cdot 10^4 = 6.175 \cdot 10^8$$

comparisons. This means that the number of comparisons is optimized by an order of four compared to the possible worst case.

For the two spatial predicates implemented using UDFs, there is also a strong correlation between the complexity of the shape and the computational complexity of the inclusion test. This means that specializing in simpler shapes may be beneficial. Beyond reducing the number of expensive comparisons by performing simpler comparisons first, efficiency can be greatly improved by introspecting the type of shape the volume has. Even though both UDFs in Approach 1 are executed the same number of times, our results show that the general method performs $5 - 6\times$ worse than the one assuming cuboid shapes when not performing axis-aligned filtering.

While Sedona supports spatial partitioning and STARK supports spatio-temporal partitioning, this is not utilized in the execution. In Sedona, this is due to the temporal predicate taking precedence in query optimization, and in STARK this is due to the Spark SQL integration being a proof-of-concept that does not employ the partitioning or indexing techniques it provides through the RDD API.

Although temporal partitioning might be sufficient, local indexing or query rewrites should still be able to further reduce the number of comparisons. By providing simpler tests as query predicates, the number of expensive comparisons should also be reduced.

With Approach 1, the secondary axis-aligned predicate provides a big performance benefit. Previously there have been discussions on further optimizing this type of inner range by using queues to only perform this simpler predicate until the upper bound is reached [58]. While these changes never made it upstream, they have been implemented and released as part of the AXS package [59]. One observation to make here is that astronomical applications are driving the optimization of multidimensional spatial workloads, with Spark3D [38], ASTROIDE [6], and AXS [59].

8.3.3 Limitations

We have seen that for the problem specification in Chapter 4, the opportunity to execute the join as a `BroadcastHashJoin` together with axis-aligned filtering performed best. While a major improvement comes from the axis-aligned filtering, this was only investigated in Approach 1 due to libraries having the ability to add these predicates during query processing.

While there are many other queries useful in validating LiDAR perception, the small perception model of object annotations means that broadcasts will be cheap generally. For larger perception models, such as segmentation, this might no longer be available. The problem specification is a method for generating a semantic segmentation perception model from volumetric annotations. Further analysis of the generated model might have other data characteristics, but the contains predicate is still fundamental for segmentation generation as well as other queries useful for analyzing object detection models.

9

Conclusion

From analyzing different LiDAR perception model evaluations and their requirements, the problem was scoped to joining three-dimensional spatio-temporal data sets with the spatial contains relation. To conclude the result, first the research questions with corresponding answers are presented. After this, the answers are briefly analyzed. Finally we share some concluding thoughts.

RQ1: What are the requirements of a framework for developing large-scale LiDAR data processing pipelines?

> The key requirement based on the pre-study was whether it could provide computational efficiency for implementations written with high-level, declarative code.

RQ2: What useful properties that could speed up the processing of LiDAR data are not utilized?

> While several 2D spatial and spatio-temporal libraries provide partitioning, indexing and join strategies, our evaluation shows that neither Apache Sedona nor STARK utilize these effectively in optimizing the spatio-temporal join.

RQ3: Can we utilize the properties identified in the previous question to reduce the computation time for a LiDAR data processing problem defined by Annotell?

> Yes, in the evaluation, the queries that use the hand-written implementations of the contains predicate were more efficient than the queries using the predicate implementations provided by Apache Sedona and STARK. While it is insufficient to completely disregard the libraries after evaluating one query, our results suggest that the performance of the evaluated libraries may vary. In this case, it was found that moving the temporal equality condition and axis-aligned range filtering outside the UDF, played a vital role in the optimization performed by Catalyst.

While the results were superior for the UDF predicate in the case for spatial join, this does not necessarily mean writing UDFs are superior to utilizing libraries for other queries. Generally, libraries can be seen as UDFs with *possible* acceleration

and future work may investigate how other types of LiDAR perception analysis can be accelerated using global and local indexing.

To conclude, Apache Spark is a suitable framework for developing and optimizing analysis of LiDAR perception systems. In our scope we have not proposed any optimization rules, as we believe that in order to make optimization efforts, a more clearly defined query domain needs to be established. The early phase of applying perception systems in autonomous vehicles means that there is a lack of industry standards for metrics and key performance indicators of perception models. Furthermore, the data sets of these models are not public as competing actors in the industry have no incentives to share their data sets.

While actors in the industry are running their own types of perception model evaluation, ensuring the safety of these systems must be performed by independent actors. This means that some sort of industry benchmark for answering queries in the standardized domain needs to be created.

As the field evolves, the opinions on what metrics are critical are likely to change. For each draft of a standardized benchmark, efforts in improving the query-response time for the included metrics can be pursued. This means that the further development of user defined types, functions and optimization rules in Apache Spark needs to target the latest benchmark.

With these challenges in mind, we suggest future work to include building a Spark-library that aims to enable domain-experts in expressing declarative queries which analyze LiDAR perception models on large amounts of diverse driving scenario data.

Bibliography

[1] Apache arrow. `https://arrow.apache.org/`. (Accessed on 06/20/2021).

[2] pandas - python data analysis library. `https://pandas.pydata.org/`. (Accessed on 06/20/2021).

[3] Md Mahbub Alam, Luis Torgo, and Albert Bifet. A survey on spatio-temporal data analytics systems. *arXiv preprint arXiv:2103.09883*, 2021.

[4] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1383–1394, 2015.

[5] Furqan Baig. *High Performance Spatial and Spatio-Temporal Big Data Processing*. PhD thesis, State University of New York at Stony Brook, 2021.

[6] Mariem Brahem. *Spatial Query Optimization and Distributed Data Server-Application in the Management of Big Astronomical Surveys*. PhD thesis, Université Paris-Saclay, 2019.

[7] Holger Caesar, Varun Bankiti, Alex H Lang, Sourabh Vora, Venice Erin Liong, Qiang Xu, Anush Krishnan, Yu Pan, Giancarlo Baldan, and Oscar Beijbom. nuscenes: A multimodal dataset for autonomous driving. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11621–11631, 2020.

[8] Edgar F Codd. A relational model of data for large shared data banks. In *Software pioneers*, pages 263–294. Springer, 2002.

[9] Open Geospatial Consortium et al. Opengis implementation specification for geographic information-simple feature access-part 2: Sql option. *OpenGIS Implementation Standard*, 2010.

[10] Open Geospatial Consortium et al. Geographic information–well- known text representation of coordinate reference systems, 2015.

[11] Martin Davis. Jts topology suite, 2012.

[12] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[13] Romaric Duvignau, Vincenzo Gulisano, Marina Papatriantafilou, and Vladimir Savic. Streaming piecewise linear approximation for efficient data management

in edge computing. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, page 593–596, New York, NY, USA, 2019. Association for Computing Machinery.

[14] Max J Egenhofer and Robert D Franzosa. Point-set topological spatial relations. *International Journal of Geographical Information System*, 5(2):161–174, 1991.

[15] Sami El-Mahgary, Juho-Pekka Virtanen, and Hannu Hyyppä. A simple semantic-based data storage layout for querying point clouds. *ISPRS International Journal of Geo-Information*, 9(2):72, 2020.

[16] Ahmed Eldawy and Mohamed F Mokbel. The era of big spatial data. *Proceedings of the VLDB Endowment*, 10(12), 2017.

[17] R Elmasri, Shamkant B Navathe, R Elmasri, and SB Navathe. *Fundamentals of Database Systems*. Springer, 2000.

[18] Jonathan Engélinus and Thierry Badard. Elcano: A geospatial big data processing system based on sparksql. In *GISTAM*, pages 119–128, 2018.

[19] Google. Machine types | compute engine. `https://cloud.google.com/compute/docs/machine-types`, 2021. (Accessed on 26/5/2021).

[20] Timo Hackel, Nikolay Savinov, Lubor Ladicky, Jan D Wegner, Konrad Schindler, and Marc Pollefeys. Semantic3d. net: A new large-scale point cloud classification benchmark. *ISPRS Annals of Photogrammetry, Remote Sensing & Spatial Information Sciences*, 4, 2017.

[21] Stefan Hagedorn, Philipp Götze, and Kai-Uwe Sattler. Big spatial data processing frameworks: Feature and performance evaluation. In *EDBT*, pages 490–493, 2017.

[22] Stefan Hagedorn, Philipp Gotze, and Kai-Uwe Sattler. The stark framework for spatio-temporal data analytics on spark. *Datenbanksysteme für Business, Technologie und Web (BTW 2017)*, 2017.

[23] Eric Haines. I.4. - point in polygon strategies. In Paul S. Heckbert, editor, *Graphics Gems*, pages 24–46. Academic Press, 1994.

[24] Bastian Havers, Romaric Duvignau, Hannaneh Najdataei, Vincenzo Gulisano, Ashok Chaitanya Koppisetty, and Marina Papatriantafilou. Driven: a framework for efficient data retrieval and clustering in vehicular networks. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1850–1861, 2019.

[25] Isak Hjortgren and Andreas Lindholm. Measuring data quality efficiently. `https://www.annotell.com/?news=measuring-data-quality-efficiently`, Dec 2020. (Accessed on 26/5/2021).

[26] ISO19125-1:2004. Geographic information — Simple feature access — Part 1: Common architecture. Standard, International Organization for Standardization, Geneva, CH, August 2004.

[27] Sunitha Kambhampati. How to extend apache spark with customized optimizations. `https://databricks.com/session/`

`how-to-extend-apache-spark-with-customized-optimizations`, 2019. Spark+AI Summit.

[28] Amir Keramatian, Vincenzo Gulisano, Marina Papatriantafilou, and Philippas Tsigas. Parma-cc: Parallel multiphase approximate cluster combining. In *Proceedings of the 21st International Conference on Distributed Computing and Networking*, ICDCN 2020, New York, NY, USA, 2020. Association for Computing Machinery.

[29] Amir Keramatian, Vincenzo Gulisano, Marina Papatriantafilou, and Philippas Tsigas. Mad-c: Multi-stage approximate distributed cluster-combining for obstacle detection and localization. *Journal of Parallel and Distributed Computing*, 147:248–267, 2021.

[30] Taehoon Kim, Jun Lee, Kyoung-Sook Kim, Akiyoshi Matono, and Ki-Joune Li. Utilizing extended geocodes for handling massive three-dimensional point cloud data. *World Wide Web*, pages 1–24, 2020.

[31] Kun Liu, Jan Boehm, and Christian Alis. Change detection of mobile lidar data using cloud computing. In *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences-ISPRS Archives*, volume 41, pages 309–313. International Society of Photogrammetry and Remote Sensing (ISPRS), 2016.

[32] Hannaneh Najdataei, Yiannis Nikolakopoulos, Vincenzo Gulisano, and Marina Papatriantafilou. Continuous and parallel lidar point-cloud clustering. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 671–684. IEEE, 2018.

[33] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110, 2008.

[34] M Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems*, volume 3. Springer, 1999.

[35] Vladimir Pajić, Miro Govedarica, and Mladen Amović. Model of point cloud data management system in big data paradigm. *ISPRS International Journal of Geo-Information*, 7(7):265, 2018.

[36] Dimitris Palyvos-Giannas, Bastian Havers, Marina Papatriantafilou, and Vincenzo Gulisano. Ananke: a streaming framework for live forward provenance. *Proceedings of the VLDB Endowment*, 14(3):391–403, 2020.

[37] Varun Pandey, Andreas Kipf, Thomas Neumann, and Alfons Kemper. How good are modern spatial analytics systems? *Proceedings of the VLDB Endowment*, 11(11):1661–1673, 2018.

[38] Julien Peloton, Christian Arnault, and Stéphane Plaszczynski. Analyzing astronomical data with apache spark. *arXiv preprint arXiv:1804.07501*, 2018.

[39] Xavier Roynard, Jean-Emmanuel Deschaud, and François Goulette. Paris-lille-3d: A large and high-quality ground-truth urban point cloud dataset for auto-

matic segmentation and classification. *The International Journal of Robotics Research*, 37(6):545–557, 2018.

[40] Shai Shalev-Shwartz, Shaked Shammah, and Amnon Shashua. On a formal model of safe and scalable self-driving cars. *arXiv preprint arXiv:1708.06374*, 2017.

[41] Yasin N Silva, Walid G Aref, and Mohamed H Ali. The similarity join database operator. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 892–903. IEEE, 2010.

[42] Satendra Singh and Jaya Sreevalsan-Nair. A distributed system for multiscale feature extraction and semantic classification of large-scale lidar point clouds. In *2020 IEEE India Geoscience and Remote Sensing Symposium (InGARSS)*, pages 74–77. IEEE, 2020.

[43] Ram Sriharsha. Magellan: Geospatial processing made easy. `https://magellan.ghost.io/magellan-geospatial-processing-made-easy/`, July 2017. (Accessed on 29/04/2021).

[44] Knut Stolze. Sql/mm spatial: The standard to manage spatial data in a relational database system. In *BTW 2003–Datenbanksysteme für Business, Technologie und Web, Tagungsband der 10. BTW Konferenz*. Gesellschaft für Informatik eV, 2003.

[45] Ruby Y Tahboub, Grégory M Essertel, and Tiark Rompf. How to architect a query compiler, revisited. In *Proceedings of the 2018 International Conference on Management of Data*, pages 307–322, 2018.

[46] Ruby Y Tahboub and Tiark Rompf. Architecting a query compiler for spatial workloads. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2103–2118, 2020.

[47] Mingjie Tang, Yongyang Yu, Qutaibah M Malluhi, Mourad Ouzzani, and Walid G Aref. Locationspark: A distributed in-memory data management system for big spatial data. *Proceedings of the VLDB Endowment*, 9(13):1565–1568, 2016.

[48] Bruno Vallet, Mathieu Brédif, Andrés Serna, Beatriz Marcotegui, and Nicolas Paparoditis. Terramobilita/iqmulus urban point cloud analysis benchmark. *Computers & Graphics*, 49:126–133, 2015.

[49] Hoang Vo, Ablimit Aji, and Fusheng Wang. Sato: a spatial data partitioning framework for scalable query processing. In *Proceedings of the 22nd ACM SIGSPATIAL international conference on advances in geographic information systems*, pages 545–548, 2014.

[50] David Vrba. Performance in apache spark: Benchmark 9 different techniques. `https://towardsdatascience.com/performance-in-apache-spark-benchmark-9-different-techniques-955d3cc93266`, March 2021. (Accessed on 06/20/2021).

[51] MaryAnn Xue Wenchen Fan, Herman van Hövell. How to speed up sql queries with adaptive query execution. `https://databricks.com/blog/2020/05/29/`

adaptive-query-execution-speeding-up-spark-sql-at-runtime.html, May 2020. (Accessed on 18/05/2021).

[52] Randall T Whitman, Bryan G Marsh, Michael B Park, and Erik G Hoel. Distributed spatial and spatio-temporal join on apache spark. *ACM Transactions on Spatial Algorithms and Systems (TSAS)*, 5(1):1–28, 2019.

[53] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. Simba: Efficient in-memory spatial analytics. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1071–1085, 2016.

[54] Reynold S Xin, Josh Rosen, Matei Zaharia, Michael J Franklin, Scott Shenker, and Ion Stoica. Shark: Sql and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*, pages 13–24, 2013.

[55] Simin You, Jianting Zhang, and Le Gruenwald. Large-scale spatial join query processing in cloud. In *2015 31st IEEE international conference on data engineering workshops*, pages 34–41. IEEE, 2015.

[56] Jia Yu, Zongsi Zhang, and Mohamed Sarwat. Spatial data management in apache spark: The geospark perspective and beyond. *GeoInformatica*, 23(1):37–78, 2019.

[57] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pages 15–28, 2012.

[58] Petar Zecevic. [SPARK-24020] sort-merge join inner range optimization - ASF JIRA. https://issues.apache.org/jira/browse/SPARK-24020, Apr 2018.

[59] Petar Zečević, Colin T Slater, Mario Jurić, Andrew J Connolly, Sven Lončarić, Eric C Bellm, V Zach Golkhou, and Krzysztof Suberlak. Axs: A framework for fast astronomical data processing based on apache spark. *The Astronomical Journal*, 158(1):37, 2019.