TU WIEN Informatics

Beyond Sparks

Crafting Clarity through Fine-grained Data Lineage

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Florian Schabasser, BSc
Matrikelnummer 11810319

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Mag.rer.nat. Dr.techn. Reinhard Pichler

Wien, 2. Dezember 2024

Florian Schabasser — Reinhard Pichler

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

TU WIEN Informatics

Beyond Sparks

Crafting Clarity through Fine-grained Data Lineage

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Florian Schabasser, BSc
Registration Number 11810319

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Mag.rer.nat. Dr.techn. Reinhard Pichler

Vienna, December 2, 2024

Florian Schabasser   Reinhard Pichler

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Florian Schabasser, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel" habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 2. Dezember 2024

Florian Schabasser

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Acknowledgements

First and foremost, I would like to thank my supervisor, Prof. Reinhard Pichler, for his invaluable feedback and guidance throughout my research. His expertise and encouragement were important elements in shaping this thesis.

I would also like to extend my gratitude to my colleagues and supervisors for their understanding and support during this time.

I am deeply grateful to my family for their backing throughout this intensive period. A special thanks goes to my girlfriend, Paula, whose understanding and tireless encouragement have been an essential source of support.

Finally, I would like to express my deepest gratitude to all those who have supported me during the creation of this thesis. Without their encouragement, this work would not have been possible.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Da Data Scientists zunehmend größere Datenmengen verarbeiten, stoßen traditionelle Programme, die auf einem einzelnen Rechner laufen, an ihre Grenzen. Verteilte Architekturen, die die Datenverarbeitung auf mehrere Rechner verteilen, gewinnen zunehmend an Bedeutung. Eine wesentliche Herausforderung bei der Nutzung von Data Intensive Scalable Computing (DISC)-Systemen ist die Nachvollziehbarkeit der Datenverarbeitung. Dies ist insbesondere in stark regulierten Sektoren, wie der Bankenbranche, von entscheidender Bedeutung. Bestehende Forschungsarbeiten haben Methoden zur effizienten Erfassung einer detaillierten data lineage in DISC-Systemen vorgeschlagen. Diese Ansätze weisen jedoch Einschränkungen in Bezug auf ihren funktionalen Umfang oder ihre Systemmerkmale auf.

Diese Arbeit präsentiert Lineage Master – einen neuartigen Ansatz, der Konzepte aus bestehenden Forschungsarbeiten kombiniert, um ein wiederverwendbares, transparentes und skalierbares System zur Erfassung feingranularer data lineage zu entwickeln. Mit *Lineage Master* können Data Scientists Fehlerursachen in ihren Datenpipelines schnell identifizieren, Ergebnisse schrittweise reproduzieren und tiefere Einblicke in ihre mit höheren Programmiersprachen erstellten Programme gewinnen. Angesichts der Kurzlebigkeit der eingesetzten Technologien liegt ein Fokus dieser Arbeit auf der Wiederverwendbarkeit des Systems. Um die Praktikabilität des Ansatzes zu demonstrieren, haben wir *Lineage Master* genutzt, um Apache Spark zu instrumentieren.

Die Ergebnisse unserer Experimente zeigen, dass unser Ansatz die Erfassung einer feingranularen data lineage bei der Datenverarbeitung in verschiedenen DISC-Systemen ermöglicht. Um die Wiederverwendbarkeit des Systems zu verbessern, haben wir die Erfassung der data lineage von ihrer Verarbeitung entkoppelt und eine klar definierte Schnittstelle zwischen den beiden Schichten etabliert. Die Evaluierung unseres praxisnahen Anwendungsfalls zeigt, dass *Lineage Master* unter bestimmten Voraussetzungen für die Erfassung der feingranularen data lineage effizient eingesetzt werden kann, selbst bei der Verarbeitung großer Datenmengen. Damit stellt *Lineage Master* eine praktikable und skalierbare Lösung dar, um die Nachvollziehbarkeit zahlreicher Data-Science-Workflows zu verbessern.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Abstract

As data scientists process ever-increasing volumes of data, traditional programs operating on a single machine are reaching their limits. Consequently, distributed architectures that divide data processing across multiple machines are becoming increasingly important. A significant challenge faced by Data Intensive Scalable Computing (DISC) systems is the lack of support for data tracking and workflow provenance, which is especially critical in highly regulated sectors like the banking industry. While existing research has proposed methods for efficiently collecting fine-grained data lineage in DISC systems, many approaches have limitations in their functional scope or system characteristics.

This thesis introduces *Lineage Master*, a novel approach that integrates concepts from previous work to create a reusable, transparent, and scalable lineage tracking system. Data scientists using *Lineage Master* can quickly identify root causes of errors, reproduce results step by step, and gain deeper insights into their programs written in high-level languages. Given the short-lived nature of data science frameworks, the reusability of the lineage tracking system is a primary focus of this work. To demonstrate the feasibility of our approach, we used *Lineage Master* to instrument Apache Spark.

The results of our study demonstrate that our approach effectively captures fine-grained data lineage across different DISC systems. To enhance the system's reusability, we decoupled the data lineage collection from its analysis and established a well-defined Application Programming Interface (API) between the two layers. Our evaluation using a real-world example shows that, under specific conditions, *Lineage Master* can efficiently capture fine-grained data lineage even when processing large volumes of data. Consequently, *Lineage Master* provides a practical and scalable solution for improving the traceability of numerous data science workflows.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Kurzfassung ix

Abstract xi

Contents xiii

1 Introduction 1
1.1 Problem Statement and Motivation . . . 1
1.2 Aim of the Thesis and Expected Results . . . 3
1.3 Outline . . . 3

2 Preliminaries 5
2.1 MapReduce . . . 5
2.2 Spark . . . 6
2.3 Further Technologies . . . 10
2.4 Data Lineage . . . 12

3 Previous Data Lineage Systems 17
3.1 Inspector Gadget (2011) . . . 18
3.2 Ramp (2011) . . . 21
3.3 HadoopProv (2013) . . . 24
3.4 Newt (2013) . . . 26
3.5 Titian (2015) . . . 29
3.6 Spline (2018) . . . 33
3.7 Ursprung (2020) . . . 34
3.8 SAMbA-RaP (2020) . . . 36
3.9 Further Approaches . . . 40
3.10 Discourse of Previous Approaches . . . 40

4 Lineage Master 45
4.1 Architectural Model . . . 46
4.2 Collection of Lineage Information . . . 50
4.3 Transmission of Lineage Information . . . 55
4.4 Analysis of Lineage Information . . . 57

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4.5 System Capabilities 59
4.6 Characteristics and Limitations 62

5 Results **65**
5.1 Performance Evaluation 65
5.2 Comparison of the Lineage Systems 80

6 Conclusion and Future Work **83**

Overview of Generative AI Tools Used **85**

List of Figures **87**

Listings **88**

List of Tables **89**

Acronyms **91**

Bibliography **93**

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

CHAPTER 1

Introduction

1.1 Problem Statement and Motivation

In recent years, the amount of data IT systems need to process has increased significantly. Estimates suggest that this trend will continue, leading to an exponential growth in data creation. By 2025, it is predicted that there will be more than 180 zettabytes of data [SI21]. To cope with this high demand, Data Intensive Scalable Computing (DISC) systems have emerged and are gradually replacing legacy programs that can no longer handle the high volume, velocity, and variety of Big Data. Apache Spark has become a widely used technology in this field, with a market share of 6.91% among Big Data processing technologies worldwide in 2023 [Dat23]. Its extension of MapReduce with in-memory buffering accelerates processing many times over.

However, a significant problem of many DISC systems like Apache Spark is the lack of support for data tracking and workflow provenance [GMF+20]. This is crucial, especially in highly regulated sectors like the banking industry. To comply with regulations like rule 239 of the Basil Committee on Banking Supervision (BCBS) [Bas13], especially principles 7-9, it is essential to identify the data origin and its transformations. Implementing a fine-grained data lineage system simplifies compliance with these regulations.

In addition, the lineage can be used to (1) simplify the error analysis and debugging process, (2) improve the program by better understanding the operations resulting from a high-level programming language, (3) establish trust in data by revealing its origin and which transformations applied to it and (4) reproduce a particular result step-by-step. [HDB17]

However, it is not a trivial task to efficiently and unobtrusively obtain all relevant information in a distributed environment. Recent work [RDA+20, GMF+20, SNV18, GIY+16, IST+15, LDY13, ASH13, IPW11, ADD+11, OR11] addressed several facets of provenance following various capturing approaches and offering different options for using

the provenance information at runtime or in a retrospective view. Essential capabilities in this area are *backward tracing*, determining the input tuples that contributed to a particular output tuple, *forward tracing*, identifying the output tuples affected by a particular input tuple, depicting *intermediate results* for debugging purposes, as well as the overall visualisation of the performed tasks in a *lineage graph*.

The proof of concepts and applications from current research [RDA^{+}20, GMF^{+}20, SNV18, GIY^{+}16, IST^{+}15, LDY13, ASH13, IPW11, ADD^{+}11, OR11] already make this possible. However, existing solutions come with significant drawbacks since they only offer a limited feature set, are accompanied by a significant runtime and output storage overhead, can not be generalized beyond their specific framework (fixed-toolset instead of multi-toolset) or require a high manual effort by the developer (not transparent to the user). The fixed-toolset capture approaches have been criticized by Rupprecht et al. [RDA^{+}20] because it's impractical to shoehorn the user into a specific framework, considering the quickly changing data science landscape. The main characteristics of state-of-the-art systems are summarized in Table 1.1.

Newt [LDY13] fulfils almost all the listed criteria. However, Interlandi et al. [IST^{+}15] integrated Newt in Spark and encountered an unacceptable performance overhead. Using the WordCount example and an input dataset with 50GB in size, their evaluation quantified the overhead with a multiplier of 29 of the unmonitored execution time. Inspector Gadget [OR11] exhibits certain limitations with respect to intermediate results, as it only records data samples. Additionally, its reusability is constrained by a number of prerequisites that must be met in order to instrument a system.

	Lineage Graph	Backward Tracing	Forward Tracing	Intermediate Results	Reusable	Transparent
SAMbA-RaP (2020)	✓	✓	✓	✓		✓
Ursprung (2020)					✓	✓
Spline (2018)	✓				✓	✓
Big Debug (2016)		✓	✓	✓		
Titian (2015)		✓	✓	✓		
Newt (2013)		✓	✓	✓	✓	✓
HadoopProv (2013)		✓	✓			✓
Ramp (2011)		✓	✓			✓
Inspector Gadget (2011)		✓	✓	✓⋆	✓⋆	✓

⋆ ... with limitations

Table 1.1: System Characteristics

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

1.2 Aim of the Thesis and Expected Results

This thesis aims to develop a comprehensive but still transparent, reusable, lightweight and scalable system. By comprehensive, we mean that it should support the essential capabilities of a data lineage system, hence backward tracing, forward tracing, determination of intermediate results and the construction of the lineage graph. Despite the wide range of functions, the system should be as unobtrusive as possible. Ideally, it should be transparent to the user. Since a fully transparent data lineage collection is particular to a specific framework, we strive for a two-tier architecture, separating the emission of data lineage from its exploitation. This should also promote the reusability and scalability of the system. Finally, the data collection should be lightweight in the sense that it should minimise runtime and output storage overhead. We will demonstrate the functionality using Resilient Distributed Datasets (RDDs) in Apache Spark. Our ultimate goal is to integrate the most promising solutions of previous work [RDA+20, GMF+20, SNV18, GIY+16, IST+15, LDY13, ASH13, IPW11, ADD+11, OR11] into a single coherent system to enhance data tracking and workflow provenance support in Apache Spark and potentially any other DISC system.

In particular, the following research questions will be investigated:

- How can a combination and extension of existing approaches be leveraged to fulfil all functions and characteristics listed in Table 1.1?
- How can a two-tier approach, utilizing a framework-specific extension for emitting and an agnostic backend system for analysing the data lineage, promote the system's reusability across different Big Data technologies?
- Which data collection algorithm is the most appropriate for this task?

1.3 Outline

The thesis is structured as follows. In Chapter 2, we focus on the theoretical background. In particular, Section 2.1 gives a brief overview of the MapReduce programming paradigm. Section 2.2 discusses the core concepts of Apache Spark and how they have further developed the concepts of MapReduce. In Section 2.3, we delve into the foundational principles of additional technologies used in our implementation. Finally, Section 2.4 deals with data lineage in DISC systems and its relevance for the traceability of the data processing.

Then, Chapter 3 examines previous data lineage systems in detail and describes each framework's architectural model, lineage collection algorithm, capabilities, limitations, and performance properties. We conclude the chapter in Section 3.10 by comparing the approaches concerning their architectural model and lineage collection algorithm.

Moreover, in Chapter 4, we will describe how we designed our lineage tracking system, Lineage Master, by combining the most promising solutions of previous work. In particular,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Section 4.1 discusses the architectural model. Section 4.2 describes how we extended the DISC framework, Apache Spark, to extract the necessary information while processing a file. Section 4.3 deals with the transfer of the data lineage, whereas Section 4.4 delves into analysing the information. Then, in Section 4.5, we demonstrate the capabilities of our system, giving concrete examples for backward tracing, forward tracing, and the visualisation of the lineage graph, including its intermediate results. We conclude this chapter by briefly discussing the system characteristics and limitations of Lineage Master in Section 4.6.

Then, in Chapter 5, we present our results. Section 5.1 gives an overview of our infrastructure for evaluating the performance and discusses the results. In Section 5.2, we compare our system with existing work and answer our research questions.

Finally, in Chapter 6, we conclude the thesis and provide an outlook for future work.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 2

Preliminaries

Distributed execution patterns like MapReduce and Spark provide a higher-level programming model to simplify parallelized data processing while retaining fault tolerance. The processing logic is compiled into a Directed Acyclic Graph (DAG) of operations, with the root operator consuming data from an input source and every subsequent operator consuming intermediate data from its predecessors. [DG04, ZCF+10]

2.1 MapReduce

The MapReduce programming model, initially proposed by Jeffrey Dean and Sanjay Ghemawat [DG04] in 2004, was the first to provide a standardized way to process large amounts of data in a distributed manner efficiently. The idea is based on the divide-and-conquer principle, distributing the data across a cluster of loosely coupled worker nodes.

As shown in Figure 2.1, the MapReduce library first (1) forks the data into M pieces and spawns several worker nodes and a master node, each holding a copy of the user program. Then (2), the master assigns M Map-tasks and R Reduce-tasks to idle worker nodes. Each map task reads (3) its corresponding split of the input data, parses key/value pairs and applies the user-defined mapping function. The pairs are buffered and periodically written (4) to the local disk into R regions using a partitioning function (e.g. *hash(key) mod* R), ensuring that tuples holding the same key are assigned to the same region. After the completion of the mapping phase, each reduce worker is notified by the master, instructing it to read (5) its corresponding split from the local disk of the map worker using a Remote Procedure Call (RPC). Subsequently, the data is sorted and grouped by the intermediate key. Hence, the user-defined reduce function receives a key and potentially multiple values, which can be aggregated or processed in another way to calculate the desired result. Finally, the output of the reduce function is written (6) to disk, resulting in R output files, one for each reduce partition. [DG08]

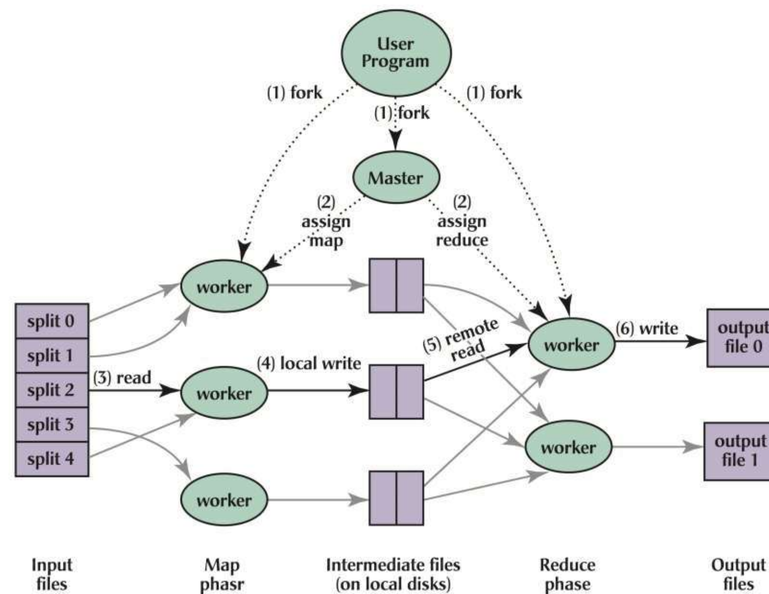


Figure 2.1: MapReduce Programming Model [DG08]

It is important to note that the algorithm writes intermediate files to disk, creating a barrier between MapReduce executions. The performed re-reading of the dataset results in a significant performance penalty for iterative jobs, larger workflows, or repeating analyses of the same dataset. This can be omitted by allowing datasets to be cached in memory. Zaharia et al. [ZCF+10] introduced this improved programming model with the Spark project. [ZCF+10]

2.2 Spark

Spark is a highly flexible, performant, and easy-to-use data processing engine. It improves MapReduce's programming model by caching datasets in memory. The project's history began at the University of California in 2009 when some AMPLab researchers observed the MapReduce framework's inefficiencies in iterative data processing. To overcome these issues, they introduced solutions like in-memory storage and recovery mechanisms to deal with faults more efficiently. Their solution claimed to be three times faster and use ten times fewer resources than MapReduce when sorting 100TB of data. In 2010, the project was open-sourced and became a top-level project of the Apache Software Foundation in 2013. [Luu21]

The third iteration of the software was released in June 2020. It offers over 80 high-level data processing operators in multiple programming languages, simplifying the use of the framework. Furthermore, it operates with various file systems, e.g. Hadoop Distributed File System (HDFS), local file system or Simple Storage Service (S3), underpinning its flexibility. [Luu21]

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

In the following sections, we will dive into the core concepts of Apache Spark version 3.0.

2.2.1 Concepts and Architecture

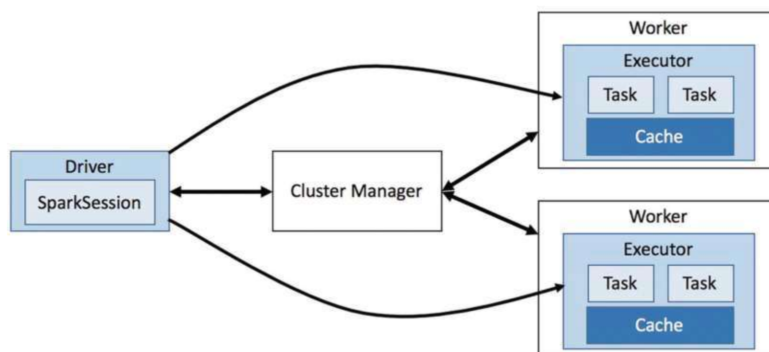


Figure 2.2: Spark Core Components [Luu21]

In the following, we will describe Spark's architecture, also depicted in Figure 2.2. Apache Spark is designed to process vast amounts of data in a distributed fashion. The system is typically deployed across a collection of machines known as a cluster. The size of a cluster can vary considerably, encompassing a few to a thousand machines. [Luu21]

A resource management system like Apache YARN maintains the cluster. The resource management system comprises the cluster manager and worker. The cluster manager knows the workers and their available resources, e.g. memory and Central Processing Unit (CPU). The primary responsibility of the cluster manager is to assign work to workers. [Luu21]

The application holds the data processing logic expressed using Spark's Application Programming Interface (API) and the driver program. As a central coordinator, the driver has multiple roles: (1) It creates the session, which acts as an entry point to Spark's functionality; (2) It interacts with the cluster manager, requesting the resources for the executors; and (3) It transforms the processing logic and distributes it as tasks across the executors. [Luu21]

An executor is a dedicated Java Virtual Machine (JVM) process. Executors run isolated from each other. While this is usually a desired characteristic, it complicates the data exchange between executors. [Luu21]

A major responsibility of the driver is transforming the processing logic into executable tasks, as shown in Figure 2.3. Therefore, it converts an application into one or multiple jobs. Each job represents a DAG of stages. A stage combines operations that can be executed in parallel. Hence, a wide transformation that performs data exchange between the executors presents a natural boundary. Each stage is divided into tasks, representing a single execution unit. Each task runs the same code but on a different data partition,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

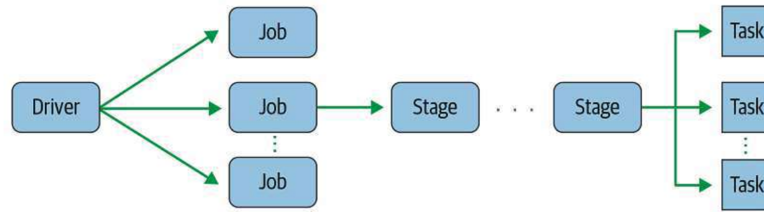


Figure 2.3: Spark Execution Model [DLWD20]

holding a subset of the data. Hence, a task is used to parallelize the execution of a stage. It is executed on a single core of an executor. [DLWD20]

The data is distributed across the physical cluster in potentially different storage systems, e.g. HDFS, S3, Azure Blob. Spark tries to exploit data locality, allocating tasks to executors near the data partitions that must be read. [DLWD20]

Transformations performed within a task can be distinguished into narrow and wide transformations. In narrow transformations, each partition is created from one other partition. Hence, narrow transformations can be executed within a stage. In wide transformations, a partition is created from several different partitions, requiring an exchange between executors. Hence, wide transformations define a boundary between the stages. [Luu21]

2.2.2 Resilient Distributed Dataset

RDDs are the primary abstraction in Spark. They represent a read-only collection of objects partitioned across multiple machines. The data represented by an RDD can be cached and reused across MapReduce-like operations, omitting the need to re-read the data from disk. This significantly improves the performance of iterative jobs and interactive analysis. [Luu21]

Every partition of an RDD has enough information about how it was constructed from other RDDs, respectively the input, that it can rebuild itself. In fact, if not explicitly cached through the user or a checkpoint, the data is always reconstructed and not present in physical storage. Through this notion of lineage, they are also fault-tolerant. [Luu21]

RDDs expose a set of data processing operations, e.g. filter, map, join and actions, e.g. count and collect. Operations are evaluated lazily only after an action is called. Each operation performed on an RDD produces a new RDD representing the result of the performed transformation. [Luu21]

2.2.3 DataFrame/Dataset

While RDDs serve as a solid underlying foundation at the core of Spark, further high-level APIs emerged over the years, imposing improved data abstraction and a better

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

computational model. One of these optimisations is the DataFrame API introduced in Spark 1.3 [Apa15] and the Dataset API introduced in Spark 1.6 [Apa16a].

With version 2.0 [Apa16b], the two APIs have been unified into a typed and an untyped version of the Dataset. However, it is essential to note that the Dataset is only present in strongly typed languages like Java and Scala. It does not exist in languages with no compile-time type safety, e.g. Python and R. The DataFrame remains the primary abstraction in these languages, as summarized in Table 2.1. In Scala, the DataFrame is just an alias for a Dataset holding the generic Row object. [DLWD20]

Language	Typed and untyped main abstraction	Typed or untyped
Scala	Dataset[T]/DataFrame (alias for Dataset[Row])	Typed/Untyped
Java	Dataset<T>	Typed
Python	DataFrame	Untyped
R	DataFrame	Untyped

Table 2.1: Main Data Abstractions in Apache Spark: Adapted from [DLWD20]

DataFrames/Datasets are very similar to RDDs concerning their evaluation semantics. Transformations are lazy, and actions are eagerly evaluated. DataFrames/Datasets can be created by reading from a structured source or transforming an RDD with appropriate schema information. [Luu21]

The DataFrame or Dataset<T>, where T is either the generic Row or an object encapsulating multiple properties, is conceptually equivalent to a relational table. It has named columns and a schema assigning a concrete datatype to each column. DataFrames/-Datasets are immutable, and Spark keeps track of all applied transformations. Hence, adding or changing columns creates a new DataFrame/Dataset. [DLWD20]

2.2.4 Optimisation

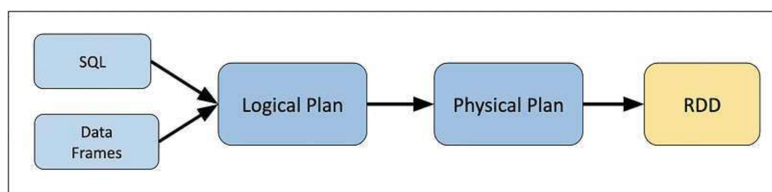


Figure 2.4: Catalyst Optimizer [Luu21]

The built-in catalyst optimizer automatically improves data processing logic expressed in Spark Structured Query Language (SQL) or by using the DataFrame/Dataset API. The optimisation involves multiple steps, depicted in Figure 2.4. Initially, it translates the SQL query or DataFrame into a logical plan. The logical plan expresses the program

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

as a tree of operators and expressions. The logical plan is optimized using rule-based and cost-based optimisation techniques. Rule-based optimisations, also well-known in relational algebra, include predicate pushdown, constant folding, and project pruning. With Spark 2.2, more intelligent cost-based optimisation techniques were introduced. Cost-based optimisation strategies attempt to determine the optimal strategy by analysing the statistical data associated with the columns involved in the join or filter conditions. [Luu21]

After optimizing the logical plan, the catalyst optimizer generates multiple physical plans. Subsequently, these are optimised through further rule-based methodologies, such as moving projections and filters to the data sources, if supported. Finally, the cheapest physical plan is translated into Java bytecode, concluding the process. [Luu21]

2.3 Further Technologies

In the following section, we will describe some essential characteristics of the Big Data technologies employed in the lineage tracking system, which will be introduced in greater detail in Chapter 4.

2.3.1 Hadoop Distributed File System (HDFS)

The HDFS was designed as a distributed, fault-tolerant storage for large files. Its data access over streams provides a high throughput but is also accompanied by a high latency. The system is built around the write-once, read-many-times pattern. It does not allow for modifications of arbitrary records. Writes to an existing file are append-only. The HDFS stores the data in blocks of 128MB. Hence, a larger file is broken into chunks of 128MB, which are stored as independent units on potentially different DataNodes. [Apa19]

An HDFS cluster has two essential components, the NameNode and the DataNode, operating in a master-slave style. The NameNode maintains the metainformation of all files and directories in the cluster. It is aware of available DataNodes and knows the location of all blocks of a given file. The HDFS cluster maintains multiple DataNodes, typically one for each physical node in the cluster. They serve read and write requests from the clients and perform block creation, deletion and replication as instructed by the NameNode. To ensure fault tolerance, the blocks of a file are replicated across multiple DataNodes. [Apa19]

Although all data is stored on the DataNodes, the NameNode is an indispensable part of the system. Without the NameNode, it would be impossible to reconstruct a given file from the blocks distributed over the cluster. Hence, the NameNode must also be resilient to failure. The HDFS provides two mechanisms for achieving this: (1) Maintain multiple copies of the persistent part of the NameNode, or (2) deploy multiple NameNodes with shared storage or a distributed edit log. [Apa19]

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

2.3.2 Kafka

The Kafka messaging system, initially developed by LinkedIn to track user behaviour on their platform, was designed as a distributed, fault-tolerant system following the publish/subscribe pattern. It allows for a loose coupling between consumers and producers. Furthermore, due to its organisation into topics and partitions, it can be scaled as needed and deliver high throughputs with low latencies. Kafka internally uses a Write Ahead Log (WAL) system. Hence, it stores all published messages in a log file before making them available to consumers. [KS17]

The architecture further differentiates between topics and partitions. A topic is a logical unit. It is divided into partitions, commonly distributed across multiple brokers. Each partition is assigned a single consumer within a consumer group, and each message in that partition is guaranteed to be read in the same order as it was written. Multiple producers may write to a single partition depending on the event key. Events with the same event key always end up in the same partition. [Apa]

Each topic's partition has a leader and zero or more followers, replicating the leader's state. Leaders and followers run on the brokers of a cluster. In case of a leader's failure, a follower will be chosen as the new leader. Hence, this mechanism ensures fault tolerance within the cluster. [KS17]

Zookeeper is the stateful component of the cluster, managing brokers and consumers. It knows about active brokers and the current leader of a topic's partition and passes this information on to producers and consumers. Furthermore, Zookeeper maintains the consumer offset, which marks the already-read messages within a topic's partition. [KS17]

The system can be configured to guarantee different message delivery semantics, listed in [Apa] and roughly described in the following:

- **At most once**: Messages can be lost but are never delivered multiple times. Deactivating retries on the producer side or committing the offset before processing the message on the consumer side leads to this.
- **At least once**: Messages can not be lost but may be delivered multiple times. To ensure this, the broker must acknowledge the receipt of the message on the producer side, and the offset may only be committed after successful processing on the consumer side.
- **Exactly once**: Messages are delivered once and only once. This can be accomplished by using Kafka's transactional consumer and producer.

2.3.3 Neo4j

Neo4j is an open-source graph database system written in Java. Unlike many other graph database systems, it does not transform data into a relational schema internally

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

but stores it natively in the form of nodes and edges. Neo4j is built to store and query extensive graphs that don't fit in memory. Furthermore, it provides an intuitive query language called Cypher and a visualisation framework to display the results of a given query. [Raj15]

Despite its optimisation for scalability and performance, it still ensures Atomicity, Consistency, Isolation, and Durability (ACID) properties, listed in [Raj15] and roughly described in the following:

- **Atomicity** (A): Multiple database operations can be wrapped into a single transaction and executed atomically. Hence, if one transaction fails, the others will be rolled back.
- **Consistency** (C): Every client accessing the database will read the latest updates.
- **Isolation** (I): Transactions are isolated from each other. Hence, writing in one transaction won't affect reading in another transaction.
- **Durability** (D): Successfully committed data will be durably stored and available, even after a restart or crash.

Achieving high availability and read scalability requires a cluster setup using the Neo4j enterprise edition. In a cluster, we need to differentiate between primary and secondary hosts. A logical database in a cluster might have one or multiple primary hosts. A single primary can be used for minimal write latency if high availability is not required. If multiple primaries are in use, Neo4j utilises the Raft protocol to ensure that a majority of the primaries ($N/2+1$) acknowledge the transaction. However, only one primary interacts with the client and synchronously replicates the transaction to the other primaries. A database in a cluster might have several secondary hosts. They periodically poll a primary node for updates to store the latest transactions. Secondary hosts do not influence the cluster's availability but only the read scalability. Introducing a cluster weakens the ACID compliance. To be precise, only causal consistency can be guaranteed in a cluster. This ensures, among other things, that a client application can read its write operations. [Neo]

2.4 Data Lineage

The idea of a data lineage, tracing the dependencies between input and output values of a computation, originates in the database field. Initial work focused on establishing lineage support for relational queries in a data warehouse with typical operators, e.g. selection, projection, join, union and aggregation [CWW00, CW01].

Subsequently, the term "data lineage" was refined in 2001 by Buneman et al. [BKT01] to why- and where-provenance. Why-provenance explains why a particular piece of data appears in the output. In contrast, where-provenance focuses more on the data origin and

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

indicates the specific locations from which the data was extracted. In 2007, Green et al. [GKT07] went even further by establishing how-provenance using provenance semirings.

Provenance semirings, introduced by Green et al. [GKT07], provide a way to track how data is derived from its sources using an algebraic structure called semiring. A commutative semiring $(K, +, *, 0, 1)$ is defined over a non-empty set K, with the addition and multiplication operation and their identity elements 0 and 1, such that $(K, +, 0)$ and $(K, *, 1)$ are commutative monoids. The multiplication is distributive over the addition $(\forall a, b, c \in K, a * (b + c) = a * b + a * c\ )$, and $\forall a,\ 0 * a = a * 0 = 0$. [GKT07]

Concerning provenance semirings, multiplication indicates joint use, whereas addition indicates alternative use. Chothia et al. [CLMR16] introduced a simple example, shown in Figure 2.5. The query selects the name and address of those customers who have ordered furniture at least once. Therefore, two simple database tables, *Customers* and *Orders*, are given. In Figure 2.5, circular nodes represent data tuples, and rectangular nodes represent operations. The why-provenance can be expressed as the multi-set $((t_1, t_3), (t_1, t_4))$ since each of the two subsets is sufficient to have t_8 in the result set. The where-provenance is defined on the attribute level. For $t_8.Address$ the where-provenance would be $t_1.Address$, since the value is propagated from the tuple t_1 to the output t_8. Finally, the how-provenance can be expressed as a semiring of the form $t_1 * (t_3 + t_4)$, meaning "tuple t_1 and either tuple t_3 or t_4". Later, Amsterdamer et al. extended the formalism in [ADT11] with aggregation and group-by.

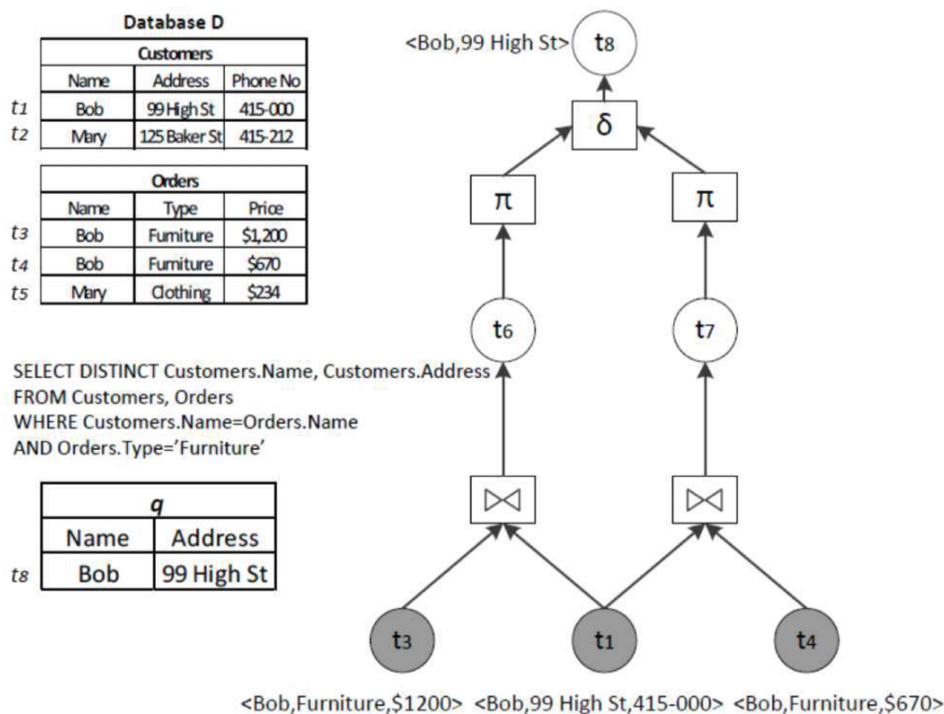


Figure 2.5: Example of a Provenance Graph [CLMR16]

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

However, a decisive disadvantage of how-provenance is the limited information about the performed transformations. Only the provenance graph in Figure 2.5 captures all relevant information, including the query operators. Chothia et al. [CLMR16] describe this more verbose form of how-provenance as transformation-provenance.

While these concepts originated from the database domain, some work, e.g., Lipstick [ADD+11], tries translating them to the Big Data domain. We will describe their approaches in more detail in Section 3.9.

2.4.1 Data Lineage in DISC Systems

As we will show in Chapter 3, there has been numerous research about data lineage in DISC systems. These approaches aim to determine a fine-grained or coarse-grained lineage as efficiently as possible. While a coarse-grained lineage only captures the operations performed, a fine-grained lineage records the processing of the individual records, similar to the transformation-provenance described in the previous section.

The fine-grained lineage is particularly suitable for debugging and error analysis, which pose significant challenges in a distributed system. There are multiple potential sources of error, e.g. semantic bugs in the program, unexpected/bad values in the data or problems with the infrastructure. Running individual tasks with a subset of the data locally might be inappropriate for identifying pathological cases that only arise when running at scale. Hence, a detailed lineage, recorded during the execution on the cluster, may prove beneficial in this case. Furthermore, it provides all the necessary information to reproduce a particular result step-by-step and thus establishes trust in the data. [LDY13]

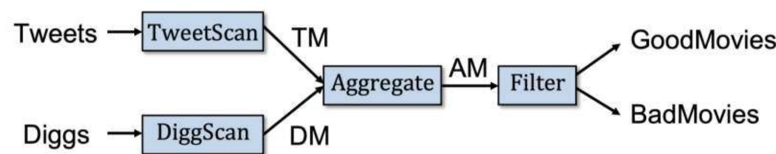


Figure 2.6: Simple MapReduce Workflow for Movie Ratings [IPW11]

In their study, Ikeda et al. [IPW11] presents a simple yet illuminating example of the potential benefits of backward tracing. Let's consider the MapReduce workflow, shown in Figure 2.6, processing movie ratings from social media platforms Twitter and Digg. The initial map function (TweetScan, DiggScan) analyses a post for a movie name and positive/negative adjectives. It emits a key-value pair with the movie name as the key and a rating as the value. The subsequent reduce function (aggregate) computes the number of ratings and the median rating for each movie. The final function (Filter) then picks up the records, categorizing them into good/bad movies, depending on their ratings. Movies with over 1000 ratings and a median of six or higher are considered good movies. Movies with over 1000 ratings and a median lower than five are considered bad movies. [IPW11]

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Let's assume we find our favourite movie in the bad movie category and, hence, question the accuracy of the result. Backward tracing of this particular result shows us that 2000 persons rated this movie, and the average score is 4.8. Further backtracing the execution to the original postings, we identify a significant time gap between the postings. We recognise that some recent postings mentioning the movie's second part have been wrongly classified and distorted the result. [IPW11]

The error would likely have been identified during manual debugging. Nevertheless, given the considerable volume of data, this process would have been considerably more arduous and time-consuming.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 3

Previous Data Lineage Systems

The following chapter delves into existing data lineage systems and describes their approaches. Specifically, we summarize their architectural model, how they implement data collection and analysis, capabilities, limitations, and performance properties.

Numerous studies have already addressed the lack of data tracking and workflow provenance support. Initial work focused on Pig [ORS+08], which provides a high-level programming language called Pig Latin to simplify the ad hoc creation of complex MapReduce jobs [OR11, ADD+11]. Another approach is to directly modify the MapReduce framework and construct the lineage during the execution [IPW11, ASH13, LDY13]. With the introduction of Spark by Zaharia et al. [ZCF+10] in 2010 and its increased usage, the focus moved towards this new technology [IST+15, GIY+16, SNV18, GMF+20]. More recently, there has been a shift towards reusable generic solutions [RDA+20].

According to Rupprecht et al. [RDA+20], one can distinguish between fixed-toolset and whole-system capture approaches. Fixed-toolset capture approaches are specialized for a particular framework and can capture fine-grained data lineage information. In contrast, whole-system capture approaches operate on the file or operating system and only capture basic lineage information. The former has been described as impractical because it shoehorns the user into a specific technology, which is challenging in today's short-lived and broad data science landscape. The latter can not provide the same level of granularity. Hence, more recent work, e.g. Ursprung [RDA+20], proposes a combination of both to automatically establish a base provenance and enrich the lineage further with data from application-level sources.

In the following, we will describe the previously mentioned work in detail.

3.1 Inspector Gadget (2011)

With Inspector Gadget, Olston and Reed [OR11] introduced a monitoring and debugging solution for DISC systems. It can be adapted to record the lineage of various systems. However, these adaptions are specialized for a particular framework and can be categorized as fixed-toolset capture solutions. To demonstrate the feasibility of their approach, they provide a reference implementation in Pig called Penny.

3.1.1 Architectural Model

An Inspector Gadget application consists of three main components, also depicted in Figure 3.1: (1) the monitoring agents positioned along the dataflow edges to observe the data records as they flow through the system, (2) the central coordinator communicating with the agents and (3) the driver, communicating with the user and configuring and launching instrumented runs of an application. The driver applies principles from aspect-oriented programming to inject code just before the actual execution of the program. The injected agents behave as no-operation functions from the dataflow engine's perspective. Hence, the instrumentation is fully transparent to the user and the framework, not requiring any framework modifications or tampering with the data. Figure 3.1 gives a rough overview of how a monitored execution could look like.

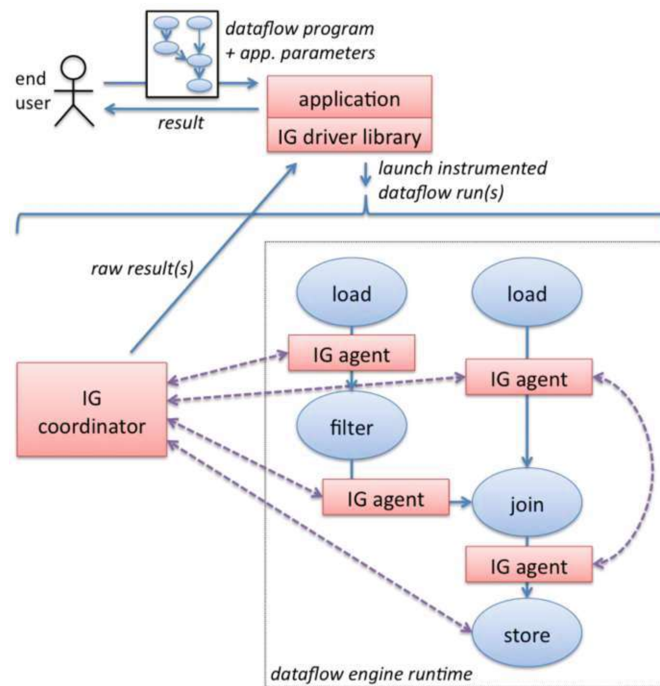


Figure 3.1: Architecture of Inspector Gadget [OR11]

Each main component implements a set of methods specified by an API.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- **Agent**: Implements the methods $init(args)$, $observeRecord(record, tags) : tags$, $receiveMessage(source, message)$ and $finish()$. Each time a record is passed through a dataflow edge, the agent's *observeRecord* method is invoked. One record is associated with zero or more tags. However, the tags are not directly embedded into the record, which we explain in more detail in the next section. The $receiveMessage$ method is invoked every time the agent receives a message from the coordinator or another agent. For this purpose, a separate message sending API is provided.

- **Coordinator**: Implements the methods: $init(args)$, $receiveMessage(source, message)$ and $finish() : output$. The $receiveMessage$ method is invoked every time the coordinator receives a message from an agent. Since the coordinator represents the outer layer of the application, the $finish$ method also returns the result to the driver, if any.

- **Driver**: Implements the methods $parse(dataflowSpec) : parsedDataflow$ and $launch(parsedDataflow, agentMap, coordinator, coordinatorArgs) : output$. It translates the dataflow specification into a graph representation and selects edges to deploy agents. This yields the parsed dataflow and the agent's map. Together with the coordinator and the coordinator's arguments, the instrumented process can be launched.

3.1.2 Collection of Lineage Information

For forward and backward tracing, the system tags and tracks the records of interest. Therefore, agents positioned at the origin assign records a unique tag. Agents positioned on downstream edges notify the coordinator when observing a tagged record. The coordinator keeps track of the messages received, which can be used to construct the data lineage.

As mentioned, Olston and Reed designed the system as unobtrusive as possible. They use a wrapper around the provided agent code called agent harness to pass the tags without tampering with the data records. It is implemented as a User Defined Function (UDF) and behaves like a no-operation function from the data engine's point of view.

As depicted in Figure 3.2, the harness implements a queueing mechanism to buffer received messages and tags. Upstream agents can notify downstream agents about assignable tag(s) by calling the $sendDownstream(message)$ method from the sending API. If the subsequent agent is in the same stage, the successor agent can be notified directly about the assignable tag. If the subsequent agent is not in the same stage, e.g. because of a group-by operation, the messages are buffered in the coordinator's message queue and passed to the downstream agents as they register in the coordinator's agent registry.

Once the harness receives a data record, it assigns the corresponding tag(s) received over the queue and then calls the $observRecord()$ method on the current monitor agent code. After completion of the $observRecord()$ method and depending on its return value, the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

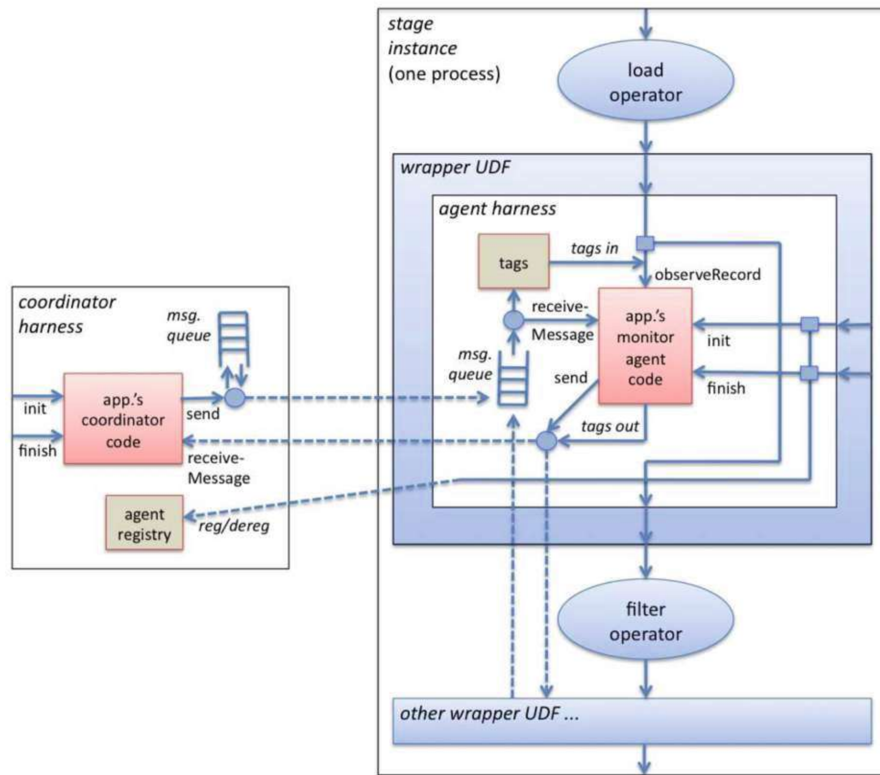


Figure 3.2: Implementation Details of Inspector Gadget [OR11]

original input record is passed on by the harness. The return value of *observRecord*() may be empty, instructing the framework to omit it. This forms the basis for a restricted execution. Otherwise, the return value is a new tag forwarded to the subsequent agent harness via the sending API.

3.1.3 Analysis of Lineage Information

Since Olsten et al. used trivial, conservative implementations, the desired information can usually be read off directly. Regarding forward and backward tracing, the tags are propagated through the system, and the relationship between input records $i \in I$ and output records $o \in O$ does not need to be calculated. The same holds for intermediate results.

3.1.4 Capabilities and Limitations

Inspector Gadget supports a wide range of desired functions, including forward and backward tracing and the output of some intermediate results (data samples). As shown with the reference implementation Penny, it can be reused across different DISC frameworks and is transparent to the user and the framework.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

However, Inspector Gadget focuses on monitoring and debugging capabilities. The program to be instrumented must be submitted to and launched by Inspector Gadget's driver with specific instructions, such as "Which input is responsible for my program's crash?"

The program assumes a direct translation of the given script into an execution graph without extensive optimisations. Aggressive optimisations of the user script would cause a mismatch between the parsed dataflow and the real execution graph. However, observing and instrumenting the post-optimised graph would be a possible workaround.

Furthermore, the tagging solution encompasses some significant problems. The implementation has a relatively high overhead and is not suitable for tracing a large number of records. Furthermore, it is based on several assumptions, e.g., buffering/queuing inside an operator would render the approach useless.

3.1.5 Performance

As mentioned, Inspector Gadget was designed in a debugging context where performance is less important than the added functionality as long as the overhead remains within an acceptable range. Olston and Reed executed four different Pig Latin scripts (distinct inlinks, frequent anchor text, big site count, and linked by large) on 10 million website records with a size of 10GB. Depending on the concrete feature, the execution time varies but always remains within a factor of two of the unmonitored executions. Concrete numbers for the features of interest (forward tracing, backward tracing and data samples) are not reported.

3.2 Ramp (2011)

Ramp, developed in 2011 by Ikeda et al. [IPW11], is a fixed-toolset capture solution for Generalized Map and Reduce Workflows (GMRW), representing an acyclic graph of MapReduce functions.

3.2.1 Architectural Model

With Ramp, Ikeda et al. extended the MapReduce framework. More precisely, they applied wrapper functions, which are fully transparent to Hadoop and the user. Hence, their approach retains the system's fault tolerance and parallel execution. The generated provenance information is stored in files and analysed using a simple algorithm described in Section 3.2.3.

3.2.2 Collection of Lineage Information

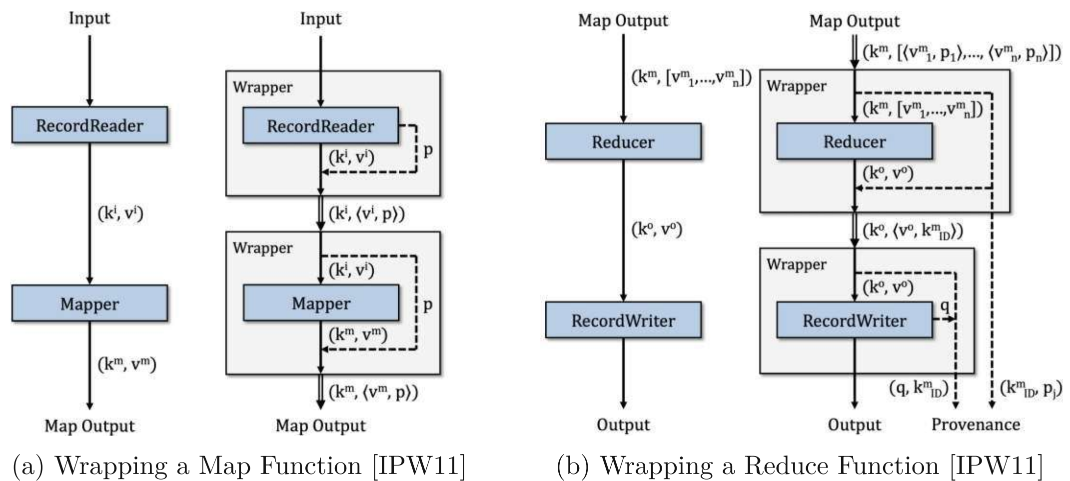
Considering the initial mapping part of a MapReduce job, depicted in Figure 3.3a, the RecordReader assigns a unique identifier to the key-value pair (k^i, v^i) it emits. The RecordReader wrapper appends this identifier to the value and then emits the pair

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

$(k^i, < v^i, p >)$, received by the subsequent mapper wrapper. It removes and stores the passed identifier and passes the plain key-value pair (k^i, v^i) to the original mapper function, which produces some mapping output (k^m, v^m). Again, the wrapper extends the value by appending the identifier and emits the pair $(k^m, < v^m, p >)$. Extending the value, Hadoop can still correctly group the mapped elements by their key.

Considering the reduce part, depicted in Figure 3.3b, the reduce wrapper receives a collection of $< v_n^m, p_n >$ for each grouping key k^m. It extracts the identifier p_n for each value v_n^m and stores the pair (k_{ID}^m, p_n) for each identifier. Subsequently, the wrapper passes the plain key-value list $(k^m, [v_1^m, ..., v_n^m])$ to the reducer function, producing some output (k^0, v^0) by aggregating the values $v_1^m, ..., v_n^m$. The reducer wrapper appends the identifier k_{ID}^m to the value part of the reducer output and then emits the pair $(k^0, < v^0, k_{ID}^m >)$ received by the subsequent RecordWriter wrapper. The wrapper removes and stores the identifier k_{ID}^m and passes the plain key-value pair (k^0, v^0) to the RecordWriter, which assigns an identifier q to each written element. This identifier is used by the wrapper function to store the provenance information (q, k_{ID}^m).

Using the provenance information (q, k_{ID}^m) and (k_{ID}^m, p_n), it is possible to trace a value written by the RecordWriter, back to its input values, read by the RecordReader. Assuming all data (input, intermediate and output) is stored in files, the pair *(filename, offset)* serves as a unique ID for input elements p and the *offset* as a unique ID for output elements q. The filename can be omitted for output elements since a provenance data file is always linked to exactly one output data file.



(a) Wrapping a Map Function [IPW11] (b) Wrapping a Reduce Function [IPW11]

Figure 3.3: Hadoop Extension in Ramp [IPW11]

3.2.3 Analysis of Lineage Information

The algorithm for backward tracing, using the generated provenance files, is simple. Given the RecordWriter's identifier q, which corresponds to the offset in the output file,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

the provenance file can be used to determine the grouping key k_{ID}^m. This, in turn, can be used to retrieve all input element IDs p_n, which correspond to the tuple (filename, offset).

However, the system is biased towards backward tracing. Forward tracing is much more expensive as it requires searching the whole map provenance file (k_{ID}^m, p_n), which is not sorted on the identifier p_n.

3.2.4 Capabilities and Limitations

Ikeda et al. focused their work on the capability of backward and forward tracing. They assume backward tracing to be more common and have optimized their solution in this respect.

There is generally a lack of optimisation in the implementation. As shown in Figure 3.3b, the reduce wrapper permanently stores the pairs (k_{ID}^m, p_n) regardless of the result of the reduce task. Hence, it might store unnecessary, unreferenced data if the reduce function does not emit an output for a particular grouping key.

Ramp assumes basic, acyclic MapReduce jobs, consisting of a map function followed by a reduce function. If a MapReduce job has no map function, it is treated as a MapReduce job with the identity map function. However, no further intervention is required if a MapReduce job has no reduce function.

Although MapReduce does not prohibit these functions, they must be deterministic and free from side effects to ensure correct tracking.

3.2.5 Performance

Ikeda et al. evaluated their implementation's performance using the MapReduce jobs WordCount and Terasort. For WordCount, they used input texts generated from 8000 distinct words with sizes of 100, 300, and 500GB. For Terasort, they used random records with sizes of 93, 279, and 466GB.

For WordCount, the runtime overhead for provenance capture was quantified at 72-76% and for Terasort at 16-20%. Interlandi et al. [IST+15] developed a version of Ramp in Spark and encountered an even more significant performance overhead of up to 320% for the WordCount example. Akoush et al. [ASH13] attribute this significant overhead to three factors: (1) the provenance information is shuffled to the reducers, (2) the wrappers constantly map the processed data, and (3) the whole provenance graph is constructed in advance during the execution of the MapReduce job.

The space overhead of many-to-one transformations, e.g., WordCount, can be made arbitrarily large. Since each output element references multiple input elements, increasing input data leads to a linear increase in provenance records. For the one-to-one transformation, such as Terasort, the observed overhead was relatively low at 19-21%.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3.3 HadoopProv (2013)

HadoopProv, developed in 2013 by Akoush et al. [ASH13], is a fixed-toolset capture solution for MapReduce. Its implementation is based on Ramp [IPW11] and improves its data collection model.

3.3.1 Architectural Model

HadoopProv is conceptually different to previous work [IPW11]. Instead of relying on a wrapper-based approach, Akoush et al. consider provenance an intrinsic feature of MapReduce and modify the framework itself. Furthermore, they aimed to minimise the temporal overhead by (1) treating provenance in the map and reduce phases separately and (2) deferring the construction of the provenance graph to the query stage. Due to the division of the provenance into map and reduce parts, there is no need to shuffle the provenance data, resulting in reduced I/O overhead. However, the optimisations made are at the expense of provenance query time, as discussed in Section 3.3.5.

Their approach is transparent to Hadoop and the user. Hence, it also retains the system's fault tolerance and parallel execution. The generated provenance information is stored in files and analysed using a simple algorithm described in Section 3.3.3.

3.3.2 Collection of Lineage Information

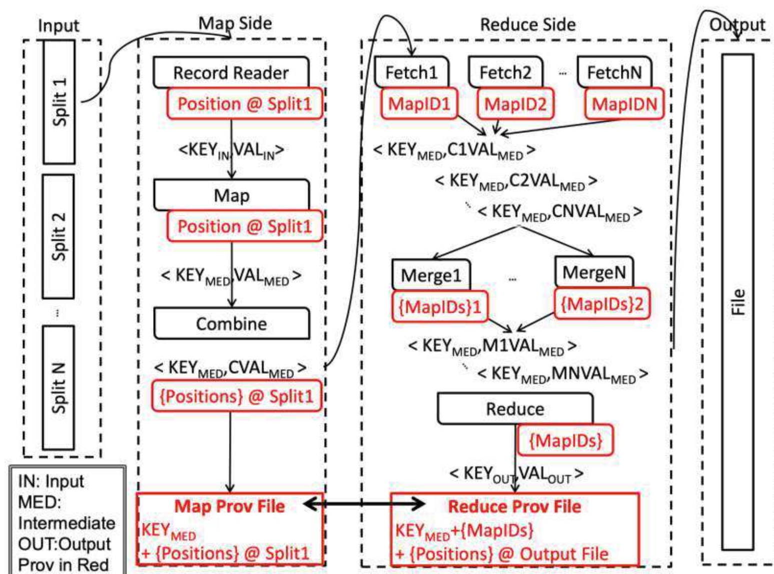


Figure 3.4: Provenance Collection in HadoopProv [ASH13]

As depicted in Figure 3.4, Akoush et al. assign an identifier to each emitted record, pointing to its origin records, and piggyback it until it materializes in the map/reduce provenance file. Map provenance files establish a link between input and intermediate

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

records. Reduce provenance files establish a link between intermediate and output records. Therefore, entries in the map provenance file contain the intermediate record key and a list of positions in the input split. One provenance record may contain multiple positions, as we can merge entries with the same intermediate key during the combine phase. Entries in the reduce provenance file also contain the intermediate record key, mapperIDs for reconstructing the provenance graph, and the positions in the output file.

The MapID is the only provenance information transferred to and shuffled in the reduce phase. While this leads to higher temporal overhead, it is essential for efficiently constructing the lineage graph.

3.3.3 Analysis of Lineage Information

Reconstructing the lineage graph within a MapReduce job involves joining the created provenance files on all matching intermediate keys. Between MapReduce jobs, the join is executed on the input/output file positions with inputA=outputB. This is possible because the framework always materializes the data between MapReduce jobs.

Despite the delayed construction of the provenance graph, queries can be serviced in $O(k * log(n))$, where n is the number of records per map task and k is the number of map provenance files that need to be searched. K is only a subset of all available map provenance files since the reduce provenance file references the MapIDs. HadoopProv further exploits the fact that the framework sorts intermediate records according to their key. Hence, within a map provenance file, the overhead for a single-key lookup is $O(log(n))$.

3.3.4 Capabilities and Limitations

Akoush et al. also focused their work on the capability of backward and forward tracing. They assume basic, acyclic MapReduce jobs with deterministic functions free from side effects.

3.3.5 Performance

Akoush et al. evaluated their implementation's performance using the MapReduce job, WordCount. They used a Wikipedia dump with sizes of 60GB, 90GB, and 300GB. They further assumed that the Wikipedia dump is skewed toward popular words to stress the system during the combine and merge phases.

The runtime overhead was quantified at a maximum of 10%. They also noticed that the output storage overhead grows proportional to the input and can become arbitrarily large. In their most comprehensive test, with an input of 300GB in size, the provenance file was already 220GB large.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3.4 Newt (2013)

Developed in 2013 by Logothetis et al. [LDY13], Newt is a reusable fixed-toolset capture solution. It offers a well-defined API developers use to instrument a specific DISC system. Logothetis et al. implemented an extension for Hadoop to demonstrate their approach's feasibility and evaluate its performance.

3.4.1 Architectural Model

The architectural model, depicted in Figure 3.5, consists of actors, clients, colocated peers, an SQL cluster and a controller (not depicted). Actors, which can be any data-transforming entities, are instrumented by developers to capture their timestamped input and output values. Therefore, developers utilize the provided Newt API, made accessible over the client. Emitted records are directly written to a local, possibly replicated, lineage log on the colocated peer. Once the capturing has been finished and all associations have been inferred from the raw logs, the system imports them into an indexed SQL table.

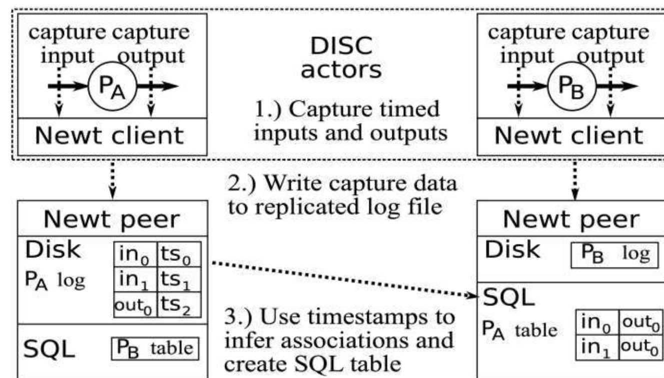


Figure 3.5: Newt Architecture [LDY13]

An important building block of the system, although not shown in Figure 3.5, is the central controller. It (1) distributes the load encountered with the ingestion of the lineage log, (2) keeps track of the association tables, (3) handles failures and may rebuild tables, (4) performs the tracing queries, and (5) orchestrates a replay by restarting actors as needed.

3.4.2 Collection of Lineage Information

Logothetis et al. decided to use an eager collection approach, questioning the premise that it is too expensive and impractical. According to them, the benefits of having all lineage information immediately available, which allows for faster analysis, debugging, and replays, outweigh the penalties encountered. However, Logothetis et al. separate the actual data and the lineage information from each other, only recording the identifier of a data record and leaving the materialization of the input to the actors.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Regarding the lineage information, Newt collects the pairs (d_{in}, d_{out}) for each actor, representing the association between input records $i \in I$ and output records $o \in O$. d_{in}, respectively d_{out} are typically cryptographic hashes or prominent values the system will preserve, e.g. the triple (file, offset, length) or a grouping key.

Developers instrumenting the actors must use the standard or paired capture API to emit the association between d_{in} and d_{out}. Assuming that developers have the input and output available at some point in the actor's execution, the standard capture API is sufficient. However, Logothetis et al. noticed that this is often not the case. It forces developers to buffer the input value until the output is known. Therefore, they introduced the paired capture API providing three distinct methods: (1) *addInput* to associate d_{in} with a specific actor, (2) *addOutput* to associate d_{out} with a specific actor, and (3) *reset* to clean the associations of the actor with the given id. The pairs (d_{in}, d_{out}) are constructed by observing the temporal order of emitted inputs and outputs, assigning an output to all inputs of that actor that happened before. However, the observations are added to the lineage log on the colocated peer without further processing. The associations are constructed after the capturing is completed.

On completion of an actor, hence when it calls the commit method of the API, a relational table holding the aforementioned pairs $((d_{in}, d_{out}))$ is created for each actor. Hence, the lineage information is scattered across multiple tables of the database cluster, and a connected set of actors is necessary to perform tracing and replays correctly. For this purpose, the API provides two management endpoints that must be utilized before the execution: (1) register to enrol the actor with Newt and to obtain its unique identifier and (2) flow_link to relate two actors with each other.

Hadoop Extension

Logothetis et al. implemented a Hadoop extension using their API, which enables an automatic and transparent recording of lineage information. Therefore, the instances of RecordReader, Map, Reduce, and RecordWriter are instrumented. The job controller links related actors and informs Newt when an actor completes its work.

- **RecordReader**: The RecordReader consumes a part, e.g. a line of the input, specified by the triple (file, offset, length) and produces the next key-value pair. The triple identifying the portion of the input file and a hash of the emitted key-value pair is sent over the capture API.
- **RecordWriter**: The RecordWriter is instrumented similarly. It receives a key-value pair and writes it to a certain part of the output file specified by the triple (file, offset, length). As before, a hash of the key-value pair and the identifying triple is transferred.
- **Map**: Map operators transform the data using a user-defined map function. Therefore, the function receives an input pair $M_{in} = (k, v)$ and produces an output pair

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

$M_{out} = (k, v)$, which is emitted using Hadoop's context interface. Instrumenting the context, it is possible to record the relevant provenance information and emit the tuple $(hash(M_{in}), hash(M_{out}))$ using the standard capture API.

- **Reduce**: Reducers combine multiple values using a user-defined reduce function. Therefore, the function receives a key and a value list as an iterator $R_{in} = (k, v[])$ and produces an output $R_{out} = (k, v)$. They instrument the context mechanism to intercept emitted input/output pairs and use the paired capture API (addInput, addOutput, reset) to record the provenance.

3.4.3 Analysis of Lineage Information

Forward and backward tracing can be performed by recursively joining the association tables of connected actors. We will describe the process in more detail in Section 3.5.3.

Selective Replays

For replay and ex-replay to work, each replayable actor must materialize its input data. Newt installs the tracing table at the actor and utilizes the return value of the methods *capture* and *addInput* to drop values that are not part of the replay.

3.4.4 Capabilities and Limitations

Logothetis et al. introduced a reusable, scaleable and fault-tolerant lineage capture system. With the Hadoop Extension, the lineage collection is even transparent to the user. Hence, data scientists do not need to integrate the capture API within their workflows.

Newt supports forward tracing, backward tracing, selective replays, and an inverted version called ex-replays. Ex-replays exclude certain faulty records from the input and replay the execution with the remaining entries. However, transformations must be deterministic, and actors must persist the input data during the initial execution for replays and ex-replays to work. The associations stored by Newt are only cryptographic hashes and do not include information about the actual data.

3.4.5 Performance

Logothetis et al. evaluated the runtime overhead and output storage penalty with the PigMix benchmark, a set of Pig Latin queries translated into MapReduce jobs. They encountered a 12-49% runtime overhead and an output storage penalty of 34-69% in their evaluation.

However, Interlandi et al. encountered significant performance problems running the Grep and WordCount example on Spark instrumented with Newt's capture API. The runtime overhead increased significantly with the input size, leading to an overhead of 2900% for a dataset of 50GB and not even terminating for a dataset of 500GB. Their

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

research revealed that the database cluster poses a bottleneck when capturing significant amounts of data.

3.5 Titian (2015)

Titian, developed in 2015 by Interlandi et al. [IST+15], is a fixed-toolset capture solution introducing interactive data lineage support in Spark. Therefore, they introduced Lineage Resilient Distributed Datasets (LRDDs) and implemented tracing methods, which can be used in a Spark session to explore the lineage and debug the system.

3.5.1 Architectural Model

With Titian, Interlandi et al. introduced the LRDD class as an extension of the RDD class. All derivates of the LRDD class implement methods for tracing one step forwards/backwards or jumping to the start/end of the dataflow. Hence, they can be used in a Spark session to explore the lineage and debug the system.

LRDDs behave very similar to Spark's native RDDs. Calling a lineage method returns a new LRDD without performing an actual evaluation. The tracing is performed only after calling an action method inherited from the native RDD. In general, calling an RDD method on a LRDD returns a native RDD that refers to the raw data at the tracked point. This property is the basis for selective replays.

To instrument an execution, Titian proceeds similarly to Inspector Gadget [OR11] and injects capture agents into the execution plan. Recall from Section 2.2.1, that Spark translates a program into a DAG of stages. By wrapping the native context with a LineageContext, Titian hijacks this process and supplements the DAG of stages with capturing agents.

3.5.2 Collection of Lineage Information

The primary responsibility of capturing agents is to (1) assign new records a unique identifier and (2) keep track of the relationship between input and output values.

The propagation of identifiers is carried out in two different ways. Within a stage, records are pipelined, i.e. Spark processes one record at a time and preserves the TaskContext. Titian exploits this fact and stores the identifier of a single record in the TaskContext at the input of a stage. Between stages, Titian attaches the identifier to the outgoing record. After the shuffle operation, all record identifiers with the same key are grouped. The group is assigned a new identifier linked to all piggybacked record identifiers.

In the following, we explain the available agents in detail:

- **HadoopRDD**: The HadoopRDD reads a file stored in the HDFS. The HadoopLineageRDD intercepts this process and assigns an identifier to each record, indicating

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

its partition and position in that partition. Since it is the first RDD within a stage, it stores the identifier in the TaskContext.

- **CombinerRDD**: The CombinerRDD pre-aggregates the records before the shuffle operation to minimise the Input/Output (I/O) overhead. Therefore, it maps the input into a hash table, using the record key as the hash key and a hash bucket holding the grouped records as a value. CombinerLineageRDDs observe this process and likewise build an internal hash table with the record key and a list of the corresponding identifiers from the TaskContext as value.
- **ReducerRDD**: After processing the inputs, the ReducerRDD returns the resulting records. ReducerLineageRDDs introspect this process and perform a lookup in the internal hash table using the record key, which returns the list of record identifiers previously added by the CombinerLineageRDD.
- **JoinRDD**: The JoinRDD combines two inputs using a specified join key. The JoinLineageRDD work similarly to the Combiner-/ReducerLineageRDD, with the only difference being that it operates on two inputs. The join key is a criterion for building the internal hash table.
- **StageLineageRDD**: The StageLineageRDD is positioned at the end of a stage to track the lineage if no combiner is present. It associates each record with a unique identifier and links it with the identifier stored in the TaskContext.

During execution, the agents batch the values in a local buffer until the operation's end, i.e., all input records have been processed. Then, the lineage data is flushed to the BlockManager, storing the information local to the capture agent. Hence, the lineage information is held in memory as long as possible to avoid the additional cost of an I/O operation. Titian only materializes the data when it runs out of memory.

3.5.3 Analysis of Lineage Information

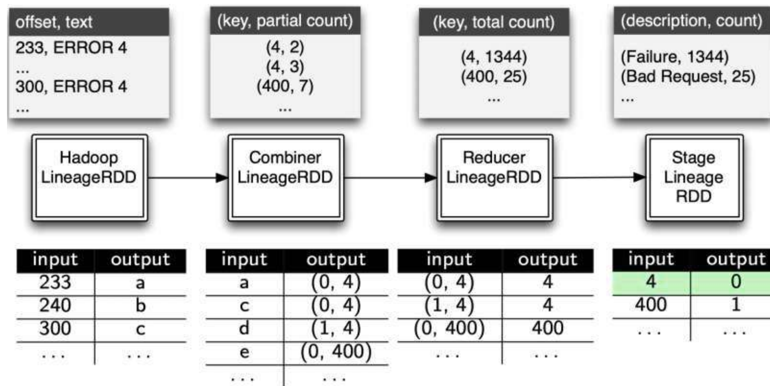
Titian constructs a trace plan by recursively joining the input and output identifiers of the data lineage tables. Thus, a trace plan can be seen as a ledger that tracks the tuples flowing through the system. It is important to note that Titian utilises Spark itself to construct this plan.

Figure 3.6a shows a simplified example for tracing back the error code with number 4. By recursively joining the input record identifiers with the output record identifiers of the preceding operation, we end up with a collection of partition offsets marking the original values of the corresponding partition.

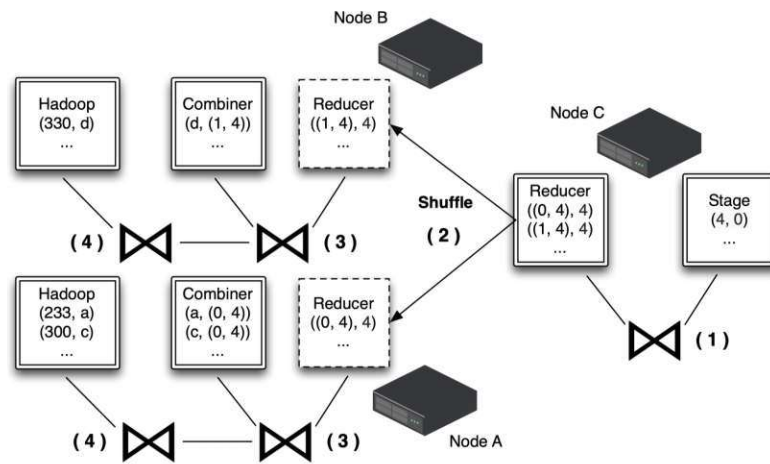
However, when executing a Spark job, the tasks are typically distributed among several executors, as described in Section 2.2.1. This results in a distributed trace plan depicted in Figure 3.6b. The data lineage tables are distributed across three nodes (A, B and C), where nodes A and B process different partitions. Again, we recursively join input and

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

output identifiers. It is important to note that the record identifier in (2) also contains partitioning information, e.g. (0,4), indicating that error code 4 occurred in partition 0. Using this information, Interlandi et al. implemented an improved version of the join, only shuffling the data once in (2) and continuing with local joins in (3) and (4). Spark's naive join strategy would also shuffle the data in (3) and (4) since it has no knowledge about the partitions.



(a) Logical Trace Plan [IST+15]



(b) Distributed Trace Plan [IST+15]

Figure 3.6: Trace Plan in Titian [IST+15]

Intermediate Results

The intermediate data is stored in dedicated data partitions and is referenced by the record identifiers. Hence, to retrieve intermediate results, the data partitions must be scanned for the given identifiers.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Selective Replays

Navigating backwards/forwards to a certain point in the transformation pipeline and replaying the execution requires Titian to retrieve the raw records referenced by the record identifiers and re-execute the original transformations.

3.5.4 Capabilities and Limitations

Titian introduced a tool for interactive data lineage exploration and debugging, enabling Spark programmers to trace the execution and display its intermediate results. Hence, Titian enables backward tracing, forward tracing, selective replays and the visualisation of intermediate results.

By default, however, only the data lineage at stage boundaries is captured. Thus, narrow transformations like map and filter are not included.

3.5.5 Performance

Interlandi et al. evaluated their implementation's performance using the MapReduce jobs, WordCount, and Grep. Inspired by Ramp [IPW11], they generated datasets of different sizes from a vocabulary of 8000 words.

According to their experiments, the implementation has a low runtime overhead, with a maximum of 27% for Grep and 29% for WordCount. The typical space overhead amounts to 30% of the input dataset and is, therefore, also within an acceptable range. They also ported and evaluated more complex PigMix examples, including aggregates, joins, and nested plans. On these queries, the overhead of Titian-D was constantly below 26%.

3.5.6 Further Considerations

Interlandi et al. have also evaluated different ways of propagating the identifiers. In addition to the distributed approach (Titian-D), described in Section 3.5.2, a propagated version (Titian-P) and a centralized version (Titian-C) have been evaluated.

Titian-P can be easily implemented by appending the lineage references to a list, piggy-backed downstream, instead of locally storing or transferring the information. The final agent will then receive the complete transformation history of each record and store it on the local BlockManager for further processing. While tracing queries are relatively fast since they boil down to traversing the list, the collection might be expensive. Hence, the propagated approach trades off space overhead for a reduced retrieval time.

With Titian-C, agents write the observed lineage over an asynchronous channel to a centralized server. Transferring provenance data into an external system provides early access, enabling users to query lineage data without requiring the execution to be completed.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3.6 Spline (2018)

Spline, developed in 2018 by Scherbaum et al. [SNV18], is a fixed-toolset capture solution that enables coarse-grained data lineage tracking in Spark. Therefore, they use Spark's internally constructed execution plan, obtained by a QueryExecutionListener. Recently, a community has developed around Spline, which has improved and expanded its original implementation [Abs].

3.6.1 Architectural Model

The data lineage server is at the centre of the architecture. It provides two external interfaces: the producer and the consumer API.

Agents capture the lineage from executions and transfer it to the central server in a standardized format over the producer API. In particular, the Spark agent uses a QueryExecutionListener to capture the execution plan and a LineageDispatcher to forward the lineage information to the central server in a standardized format. The Spline user interface or third-party applications can query and read the stored lineage using the consumer API.

3.6.2 Collection of Lineage Information

The Spline Spark agent implements its own *QueryExecutionListeners* to obtain the execution plan. It is passed to the agent as an instance of the QueryExecution class, which holds the logical, analysed, optimized and physical execution plan. As described in Section 2.2, the logical plan undergoes several optimisation steps before it becomes a physical plan, which can be translated into JVM bytecode. The agent translates the analysed plan into the Spline model and transfers it to the centralized server using the producer API.

3.6.3 Analysis of Lineage Information

The execution plan requires no further processing. Once saved in the central server's storage system, it is available through the consumer API.

3.6.4 Capabilities and Limitations

Since Spline only records the analysed execution plan, its capabilities are limited. It can only depict coarse-grained lineage, showing the data sources, destination, and applied transformations. Hence, it captures the coarse-grained lineage graph of a data transformation.

3.6.5 Performance

With Spline, Scherbaum et al. introduced a particularly lightweight approach. Since it only collects the execution plan, its computational overhead is independent of the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

processed data. It depends only on the number of operations and, thus, on the size of the execution plan.

Scherbaum et al. evaluated the performance using a simple example that first filters a collection of articles by year and language and then groups them. They executed the experiment with different data sizes from 10 million to 100 million rows and measured the overall execution time. Their experiments showed that the time difference between processing the dataset with and without Spline is in the millisecond range and independent of the data size.

Furthermore, they conducted experiments to measure the storage overhead. Since it only depends on the size of the logical plan, they randomly generated transformations with 100 to 1000 operations. The experiments have shown that the overhead is negligible. Even with 1000 operations, it is less than 2MB.

3.7 Ursprung (2020)

Given the vast and rapidly changing data science landscape, Rupprecht et al. [RDA+20] emphasise the importance of technology-independent systems. With Ursprung, they developed a fundamentally different lineage tracking system, combining techniques from fixed-toolset and whole-system capture approaches.

3.7.1 Architectural Model

Rupprecht et al. propose using system-level artefacts, such as file system interactions, to establish base provenance and application-level artefacts, such as application log files, to enrich the lineage further.

As shown in Figure 3.7, Rupprecht et al. introduced an event-driven architecture. Each provenance source acts as a producer, pushing its events to a scalable distributed message queue from which one or multiple consumers pick them up. The producers have Provenance Daemons in place to (1) configure the collection according to some Provenance Class Filters, (2) transform events to semantically richer ones according to some Event Aggregation Model, and (3) to emit obtained execution artefacts.

The Event Aggregation Model and the Provenance Class Filters have been introduced to cope with many events. As shown in Figure 3.7, they are present in the collection system but also pushed to the Provenance Daemons (provd) to filter and aggregate events as soon as possible.

Provenance Class Filters provide a simple abstraction for configuring the events of interest and tailoring the collected information to the user's needs. Event Aggregation Models define rules to create semantically richer provenance events from lower-level ones. These transformations can be stateless or stateful. The latter is the task of the Stateful Event Curator in the collection system since it accumulates information from subsequent events and cannot be carried out by a single Provenance Daemon.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

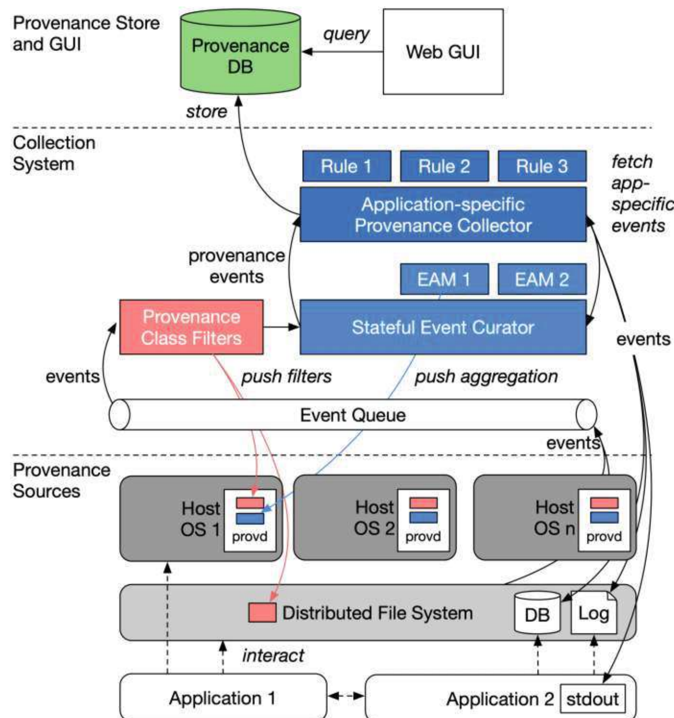


Figure 3.7: Architecture of Ursprung [RDA+20]

The filtered and aggregated provenance events are then passed to the Application-specific Provenance Collector, which matches them against user-defined capture rules. We will describe these rules in more detail in Section 3.7.2. The data fetched by the Application-specific Provenance Collector is fed back to the Stateful Event Curator for potential aggregation. Finally, the provenance events are stored in a relational database labelled ProvenanceDB in Figure 3.7 and can be accessed over a web interface.

3.7.2 Collection of Lineage Information

System-level Sources

System-level sources, such as the operating or file system, generate low-level events, e.g. system calls or interactions with the file system, establishing a base provenance.

Rupprecht et al. point out different ways to obtain these low-level events, such as using notification mechanisms provided by Linux's auditd to receive system call notifications or IBM's Spectrum Scale Watch Folders to monitor file system interactions.

Application-level Sources

Application-level sources, such as log files, temporary files, or databases, can augment the base provenance established by system-level sources. Therefore, the Application-

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

specific Provenance Collector matches the received base provenance events emitted by system-level sources against the defined capture rules.

Capture Rules can be described as self-defined hooks following a specific syntax. For example, when configuring Ursprung to load temporary data from a file using the capture rule *"FILELOAD path INTO destination"*, it observes the system notifications for interactions with that file. It triggers the rule when a write operation is performed. A total of five different rules have been established: (1) load temporary data from a file, (2) extract some records from a log file, (3) transfer data from a database, (4) extract some records from standard output and (5) track the content of a file.

3.7.3 Analysis of Lineage Information

Due to Ursprung's generic nature and the potentially very different application-level sources, Rupprecht et al. do not analyse the application-level data in detail. The user is left to analyse the data extracted by capture rules.

3.7.4 Capabilities and Limitations

Rupprecht et al. implemented a graph explorer that visualises processes with their input and output dependencies using the base provenance established by system-level sources.

As mentioned, they don't analyse application-specific data in detail and hence can't provide backward tracing, forward tracing, selective replays and the output of intermediate results.

Furthermore, the current implementation of Ursprung uses IBM's Spectrum Scale Watch Folders to monitor file system interactions. Therefore, data science pipelines must adopt this technology for Ursprung to function.

3.7.5 Performance

Rupprecht et al. evaluated the performance of their implementation using four different workloads (CleanML, Vanderbilt, Spark, and ImageML). Spark was used to train a logistic regression machine learning model with a 55GB data set.

Spark shows no visible overhead when the provenance collection is activated. In general, pipelines with long-running jobs and few I/O operations encounter no significant overhead. For I/O intensive pipelines, the overhead remains within an acceptable range of 4%. However, it should be noted that the information collected was reduced to a minimum and is, therefore, far from the level of detail of a fine-grained data lineage.

3.8 SAMbA-RaP (2020)

SAMbA-RaP, developed in 2020 by Guedes et al. [GMF+20], is a fixed-toolset capture solution for Spark that enables fine-grained data lineage tracking. Therefore, they

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

extended RDDs to envelope its structure and content.

3.8.1 Architectural Model

Guedes et al. introduced SAMbA-Rap, a transparent extension for Apache Spark designed to capture prospective, retrospective, and domain-specific data. Prospective data focuses on workflow specifications, while retrospective and domain-specific data relate to artefacts generated during execution and information about the execution environment.

SAMbA-Rap's architecture, shown in Figure 3.8, has three main components for capturing, respectively processing the provenance information:

- **Retrospective And Domain Provenance Manager**: Acts as an additional processing layer inside the executor.
- **Prospective Provenance Manager**: Communicates with the Cluster Manager in the SparkContext.
- **Provenance Data Server**: Responsible for converting, storing and providing the provenance data.

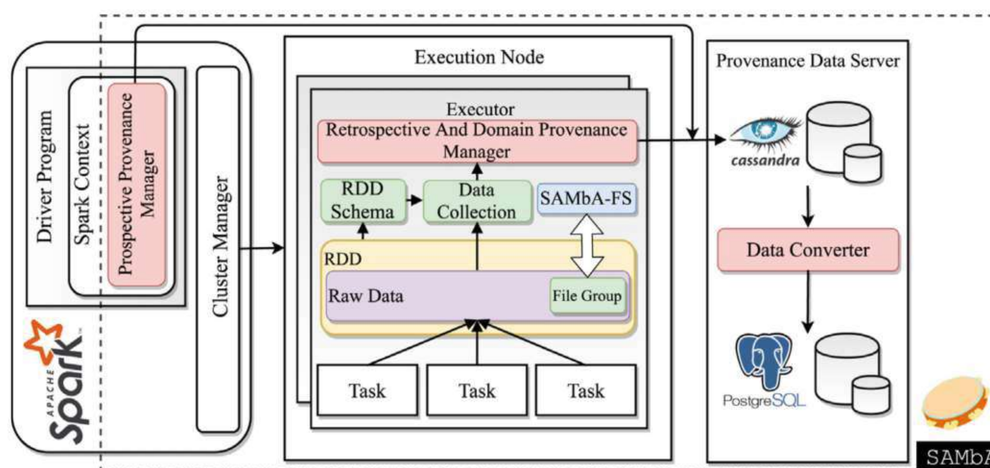


Figure 3.8: Architecture of SAMbA-RaP [GMF+20]

An important building block for reducing the amount of collected lineage data is the RDD schema. Using this schema, domain experts can select the attributes of interest whose lineage is automatically recorded using Data Collections. The collection process, performed by the Retrospective And Domain Provenance Manager, is described in detail in Section 3.8.2.

To further enhance the information collected by the Retrospective And Domain Provenance Manager, the Prospective Provenance Manager gathers additional information about the transformations. It establishes a link to the Data Collection envelope using

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

the transformation identifier. This allows semantic enrichment of the lineage, e.g., by naming the recorded transformations. However, it's also possible to exclude certain transformations, e.g., semantically weak transformations like typecast, reducing the collected data.

The Provenance Data Server successively receives the lineage data from the Prospective, Retrospective and Domain Provenance Manager and stores it into a CassandraDB, a high throughput Database Management System (DBMS). Guedes et al. opted for this architecture as CassandraDB has significant performance advantages regarding data ingestion. However, since its querying language is limited, the data is subsequently translated into a relational schema by a data converter and can be queried using simple SQL statements.

In addition, Guedes et al. implemented an in-memory file system called SAMbA-FS to optimize the execution of external programs, also referred to as black-box activities. External programs are invoked over Spark's pipe operator, temporarily delegating the control. The data between black-box activities is transferred over files. The SAMbA-RaP file system optimizes this process by intercepting the I/O instructions of black-box activities and handling the data in memory.

The remaining components shown in Figure 3.8 are described in Subsection 2.2.1.

3.8.2 Collection of Lineage Information

In the following section, we introduce the data collection mechanism of SAMbA-RaP, also depicted in Figure 3.9.

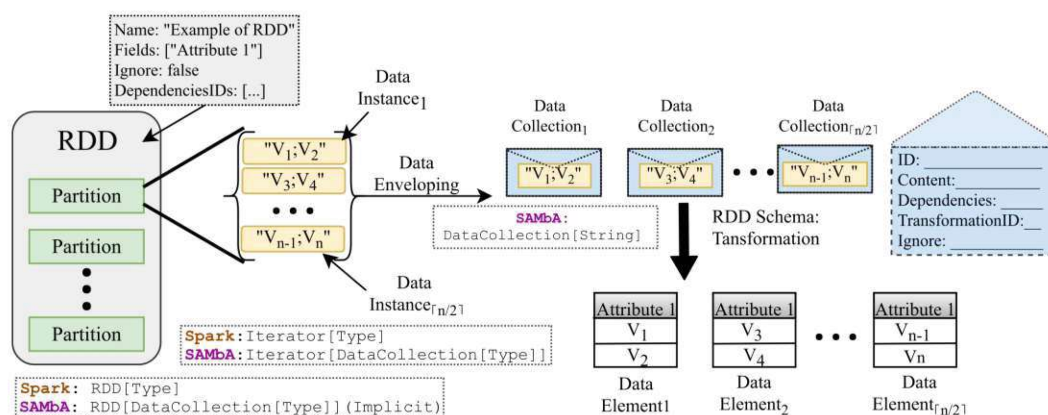


Figure 3.9: Data Collection Mechanism in SAMbA-RaP [GMF+20]

The main abstractions are the Data Collection, the RDD Schema and the Data Element. The Data Collection envelopes Data Instances of RDD partitions with their dependencies and an identifier. Entries of Data Collection are linked with each other to keep track of the entire lineage.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

The RDD schema transformation is a crucial part of the implementation. It transforms the Data Collection into Data Elements by applying a defined schema. The Data Element includes the information of the Data Collection in a structured and iterable format. Data Elements have a unique identifier, and consequently, the dependencies of a Data Element are represented as a sequence of identifiers representing the entire data lineage.

3.8.3 Analysis of Lineage Information

As described in Section 3.8.1, the data is available in a relational schema and can be queried over simple SQL statements. Guedes et al. do not describe further details on analysing the data lineage. However, they implemented a user interface that analyses the data.

3.8.4 Capabilities and Limitations

The various reports provided over the web interface (retrospective provenance report, prospective provenance report and domain data report) enable the user to explore the transformation graph with its Data Collections and Data Elements. The prospective provenance report shows the transformation graph, the retrospective provenance report focuses on the data flow, and the domain data report presents the domain data for certain workflow activities.

However, the proposed architecture also comes with a limitation. Similar to the centralized version of Titian (Titian-C), described in Section 3.5.6, the employed DBMS system (CassandraDB) could represent an upper limit for the throughput.

3.8.5 Performance

Guedes et al. evaluated the performance of their implementation using five different workflows (SciPhy, Montage, WordCount, BuzzFlow and SalesForecasts). In the interests of comparability, we focus on the WordCount example, which has been used many times before for evaluating performance.

The WordCount workflow was executed with the content of 44 books from William Shakespeare, totalling 17.1GB. They directly compared SAMbA-RaP with provenance support (SAMbA-RaP-C3) and SAMbA-RaP without provenance support (SAMbA-RaP-C1) and found that the provenance management added a significant performance overhead. The median execution time increased from approximately 10 seconds to approximately 95 seconds.

However, it is important to note that the WordCount example represents an exceptional case, as it cannot benefit from SAMbA-FS due to its usage of native Spark transformations. In contrast, the SciPhy workflow, which incorporates black-box transformations, can achieve even faster execution on SAMbA-RaP with SAMbA-FS than on a standalone Apache Spark instance.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3.9 Further Approaches

Lipstick, developed by Amsterdamer et al. [ADD+11] in 2011, brings together database-style and workflow-style provenance utilizing the semiring framework of [GKT07] extended with capabilities for aggregation and group-by. Therefore, they translate Pig Latin expressions into the bag semantics of Nested Relational Calculus (NRC) and apply the semi-ring framework to establish provenance. Since the statements are processed, the tracking does not require any modifications to the Pig Latin engine. While this is a remarkable approach, we believe it is not feasible since we don't want to limit our work to a specific language like Spark SQL but rather track any executed RDD.

Motivated by the user studies of Inspector Gadget [OR11], Gulzar et al. have further developed the debugging capabilities of Titian [IST+15]. Their system BigDebug [GIY+16] introduces (1) crash culprit determination and remediation, (2) latency alerts, (3) simulated breakpoints for step-through debugging without pausing the execution, and (4) guarded watchpoints to observe a specific variable and retrieve only those values that match certain user-defined criteria. Hence, BigDebug has developed in the direction of monitoring and debugging capabilities, which is not the primary aim of this work.

3.10 Discourse of Previous Approaches

In this section, we discuss and compare the approaches presented previously. First, we explore the architectural models in Subsection 3.10.1. Next, we delve into the various algorithms used for lineage collection in Subsection 3.10.2 and draw some conclusions for our architecture in Subsection 3.10.4.

3.10.1 Architectural Model

The architecture plays a vital role in achieving our goal of designing a transparent, reusable, lightweight, and scalable system. In this subsection, we explore and evaluate possible designs.

External Provenance Data Server

Recent architectures ([RDA+20, SNV18, GMF+20]) have employed an external server to store and analyse the collected lineage information. Therefore, the data transfer between the external server and the Big Data framework is an essential performance criterion.

Rupprecht et al. [RDA+20] provides an excellent example of employing an external provenance data server. As described in Section 3.7, they introduced filters and aggregators to reduce the number of events at its creation. The remaining events are buffered in an event queue and then processed.

Guedes et al. [GMF+20] utilises a high-throughput DBMS (CassandraDB) to cope with the vast amount of messages. As described in Section 3.8, it also acts as a buffer, and the messages are subsequently translated into a relational schema by a data converter.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Interlandi et al. [IST+15] also evaluated a centralized version of Titian (Titian-C). They encountered significant performance issues in their evaluation. With the WordCount example, Titian-C could not handle dataset sizes beyond 2GB. Regarding the Grep example, Titian-C performed similarly to Titian-D for dataset sizes below 5GB. For bigger dataset sizes, the numbers diverged, with Titian-C unable to go beyond 20GB. However, in Titian-C, agents write the data over an asynchronous message channel to a centralized server, which stores the data in its local file system. Recent developments utilising scalable architectures could give this approach a decisive turnaround.

Local Processing

Due to the vast amount of lineage data, storing it locally instead of transferring it to an external system seems reasonable. With Titian, Interlandi et al. [IST+15] provides an excellent example. They store all lineage data in the BlockManager, Spark's internal storage and analyse it using Spark. Their performance evaluation showed promising results, even for datasets of 500GB.

However, our overall goal is to design a reusable system that can be used to instrument any DISC system by separating the lineage collection from its analysis.

3.10.2 Collection of Lineage Information

The lineage collection is a crucial success factor, as the resulting overhead mainly depends on the efficiency of this algorithm. Existing solutions propose different methods, with or without manipulating the underlying execution engine. A solution with minimal changes to the engine's code would be preferable regarding reusability and maintainability. However, the optimal data collection algorithm and its implementation are particular to the employed framework.

Extending the Execution Plan

One approach that avoids modifying the codebase is extending the execution plan, as detailed in Section 3.1. Olston and Reed [OR11] assume a direct translation of the given script into an execution graph without extensive optimisations since this may cause a mismatch between the instrumented and the actual execution. This is a significant limitation since Spark internally uses an optimizer to improve the given programming logic before translating it into a physical plan.

However, as we learned from Spline [SNV18], it's reasonable to obtain the post-optimised physical execution plan in Spark by (1) using a QueryExecutionListener or (2) invoking the explain function. It would then be necessary to analyse the optimized plan and convert it into a sequence of RDDs with UDFs in-between, intercepting the dataflow and collecting the lineage information.

While this would be technically feasible, Inspector Gadget's [OR11] evaluation, described in Section 3.1.5, shows that this approach is not the best in performance.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Instrumenting the Execution Environment

A frequently pursued approach is directly integrating the code required for collecting the lineage into the framework.

Ramp [IPW11] uses a wrapper-based approach to assign and manage an identifier for each record. Its improved version, HadoopProv [ASH13], considers provenance an intrinsic feature and modifies the components directly instead of wrapping them. The Hadoop extension of Newt [LDY13] works similarly. However, they use their API to emit the collected data lineage.

In the case of Apache Spark, Interlandi et al. [IST+15] have shown a performant way for data lineage collection by directly modifying the RDDs from the core package. In contrast, SAMbA-RaP [GMF+20] wraps the data, leaving the RDD itself unchanged.

Utilization of Application Artifacts

Another lightweight approach utilized by Rupprecht et al. [RDA+20] that does not require changes in the execution engine's code is to use generated artefacts. Therefore, they have implemented a sophisticated system for extracting lineage information from application-level and system-level sources, as described in Section 3.7.2.

However, the data processing system, e.g. Apache Spark, would still need to collect and store the fine-grained data lineage and provide it as an application-level source.

3.10.3 Analysis of Lineage Information

Analysing the data lineage is crucial as it enables users to extract meaning from the data. There are two main methods to analyse the data lineage, namely external and integrated systems, which we will describe in more detail in the following.

External Systems

Various lineage tracking systems employ an external storage system such as Relational Database Management System (RDBMS) [GMF+20, RDA+20, LDY13], graph databases [SNV18] or the HDFS [ASH13, IPW11]. Specifically for Newt [LDY13] and Ramp [IPW11] Interlandi et al. [IST+15] argued that the external storage system has a high overhead and is less interactive. However, it provides a clear separation between the collection and the storage/analysis of data lineage. Depending on the external storage, it may provide additional options for analysing and visualising the data lineage, e.g., using Neo4j's graph explorer, Bloom.

Integrated Systems

Interlandi et al. [IST+15] have incorporated the analytical functions directly into Spark, enhancing interactivity. While executing a program in the Spark Shell, users can perform actions like *goBackAll*, *goBack*, *goNextAll* or *goNext* to inspect the lineage of each

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

executed RDD. Analyses such as forward- or backward tracing are performed using Spark itself. This provides a seamless and interactive debugging experience. However, the implementation is specific to the framework and can not be reused across different DISC systems.

3.10.4 Conclusion

We aim to identify and combine the most promising solutions in a new system. From the previous discussion and Sections 3.1 to 3.9, we determine the following:

- Using an API enables more straightforward instrumentation of DISC frameworks and promotes the reusability of the lineage processing/analysis system [LDY13].
- A message queue between the lineage collection and lineage processing/analysis system acts as a buffer and promotes scalability [RDA+20].
- Piggybacking and shuffling the provenance information with the data has a noticeable impact on performance and should be avoided [ASH13, IST+15].
- Using two hashes per data element and operation (HashIn, HashOut), it is possible to create a ledger with the lineage information of this data element [IST+15].
- A high-performant acquisition of fine-grained data lineage requires a direct extension of the DISC framework [IST+15, GMF+20].
- Data analysis should be postponed until after execution [ASH13].
- Coarse-grained lineage information can be collected transparently without modifying the DISC framework [RDA+20, SNV18].
- To persist the lineage, previous solutions employed RDBMS [GMF+20, RDA+20, LDY13], graph databases [SNV18], the HDFS [ASH13, IPW11] or internal storage layers like the BlockManager in Spark [GIY+16, IST+15].
- Previous approaches, specifically Ramp [IPW11] and Newt [LDY13] do not perform well at scale because they employ external storage systems (RDBMS, HDFS) [IST+15]. However, more recent solutions, such as [GMF+20, RDA+20], do so without performance problems.
- Using an asynchronous channel to store the data lineage on the local file system of a centralized server does not perform well at scale [IST+15]. However, more recent, scalable solutions may provide a decisive turnaround to this approach.

Furthermore, we should consider the following concerning the Spark extension:

- Using the TaskContext for propagating identifiers within a stage is feasible [IST+15].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- A wrapper around the SparkContext can be used as an entry point for the monitored execution [IST+15].
- It is a viable solution to extend relevant RDDs to add lineage tracking capabilities [IST+15].
- Each RDDs *compute* method returns an iterator calling the iterator from its parent RDD and performing some actions on the data. This chain of iterators can be intercepted to record the lineage [IST+15].

Based on these findings, we have designed a new system for capturing the lineage information of DISC systems.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 4

Lineage Master

This chapter presents our lineage tracking system, Lineage Master. In Section 4.1, we will describe the architectural model, which is divided into three layers: (1) Data Processing & Lineage Collection, (2) Lineage Processing and (3) Storage and Presentation.

Section 4.2, describes the data lineage collection in detail and how instrumented Apache Spark. Specifically, we will explain our RDD extension called LRDD, necessary identifiers and tracing of narrow/wide transformations.

In Section 4.4, we examine the construction of the fine- and coarse-grained lineage graph in depth. Specifically, we will describe how we persist the data lineage.

Section 4.5 demonstrates our system's capabilities, including backward and forward tracing, visualisation of intermediate results, and the lineage graph.

Finally, Section 4.6 points out some vital system characteristics and limitations that must be observed.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4.1 Architectural Model

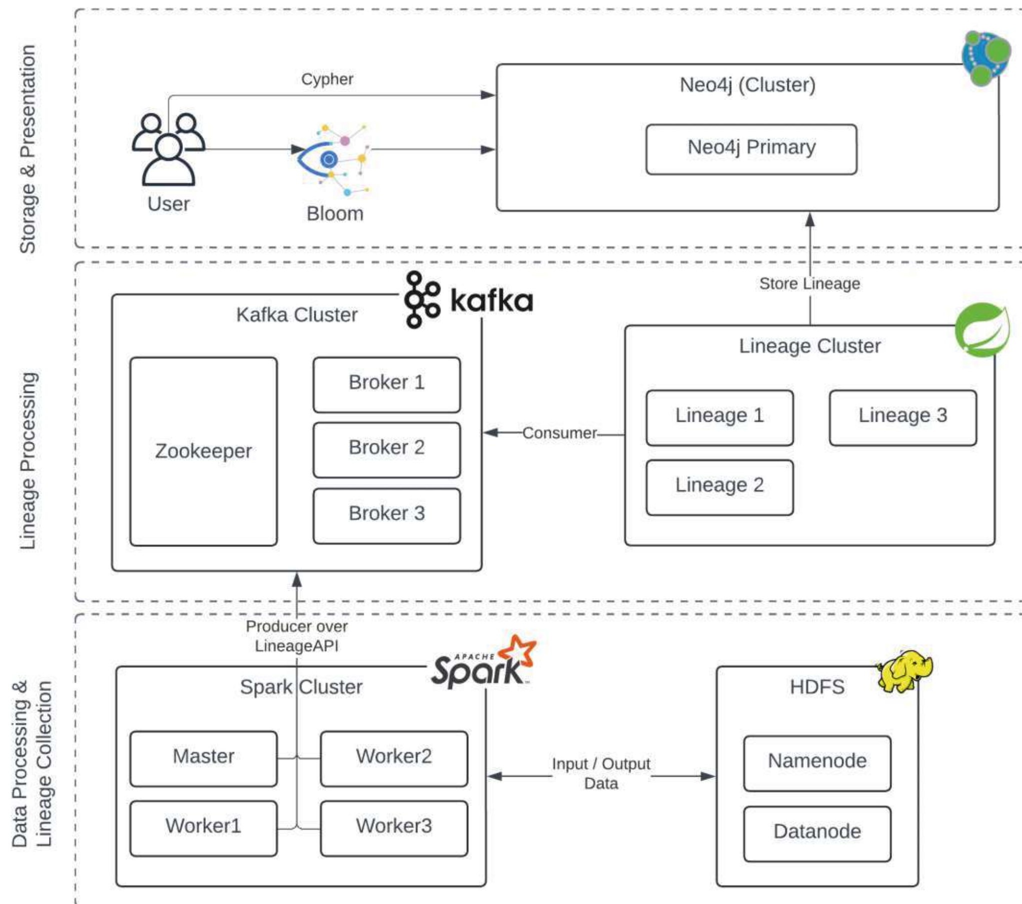


Figure 4.1: Architectural Model

We strive for a scaleable two-tier architecture that separates the collection and emission of lineage information from its exploitation. As shown in Figure 4.1, the whole architecture is divided into three layers: (1) Data Processing & Lineage Collection, (2) Lineage Processing and (3) Storage & Presentation. The architectural design thus corresponds to an external provenance data server described in Subsection 3.10.1.

In our reference implementation, the data resides in an HDFS directory and is processed by a standalone Spark cluster consisting of one master and three worker nodes. The Spark application extracts and emits the data lineage while processing. The message broker acts as a buffer until an instance of the lineage cluster can consume and process the message. The Kafka cluster has multiple brokers to ensure high availability and improve performance.

The lineage service is written in SpringBoot and is responsible for processing and persisting

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

the lineage information. It constructs the coarse- and fine-grained lineage graph. The coarse-grained lineage graph shows the transformations that were performed during the run of the Spark application. It is composed of nodes from the type LineageNode. The fine-grained lineage graph visualises how the data flowed through the system. It is composed of nodes from the type LineageFlow.

The graph database, Neo4j, allows users to query the lineage and visualise the results, as demonstrated in Section 4.5. As described in Subsection 2.3.3, a Neo4j cluster can achieve high availability and read scalability. However, the cluster mode requires some functionalities only available in the enterprise edition. Therefore, we decided to use a single but well-equipped instance. This also minimises write latency, as the Raft protocol is not used.

The individual clusters can be scaled as needed depending on the expected load. However, the design described forms the basis for the performance evaluation in Section 5.1.

4.1.1 Lineage Collection

LineageApi	
Function	Description
Management API	
void register(String nodeId, String name, String description)	Register a node with an identifier (nodeId), a name and a description
void link(String srcNodeId, String destNodeId)	Link two registered nodes with each other using their unique identifiers
Capture API	
void capture(String flowId, String hashIn, String hashOut, String value)	Create a lineage association (hashIn, hashOut) with an identifier (flowId) and, if applicable, a value

Table 4.1: LineageApi

We provide developers an API to simplify the instrumentation of a DISC framework. It provides general-purpose functions for emitting the lineage information, similar to the Newt API described by Logothetis et al. [LDY13].

The management API aims to register and link LineageNodes and thus construct the coarse-grained lineage graph described in Subsection 4.5.1. A globally unique NodeId is required to identify and link the LineageNodes when invoking the *link* method.

The capture API collects the fine-grained data lineage. Thus, we record a pair of hashes (HashIn, HashOut) for each data record. It forms the basis for forward tracing, backward tracing and the visualisation of intermediate results. As described in Section 4.2, special care is required when creating the hash values. Furthermore, a globally unique FlowId is needed to identify the lineage elements and thus eliminate duplicates.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Parts of the lineage collection are, by necessity, specific to the framework. However, we are confident that this API simplifies the instrumentation of a DISC framework.

4.1.2 Data Transmission

Communication between the lineage collection and the lineage processing system is crucial since the emission of this information may impose a bottleneck. Hence, we need an asynchronous, high-performing, and persistent communication channel for exchanging messages.

We decided to use the state-of-the-art event streaming platform Apache Kafka since it has higher throughput than most other messaging systems, built-in partitioning and replication [Apa]. Partitions are used to parallelize the processing of lineage information. The PartitionId of a Spark partition is used to determine the corresponding Kafka partition. Hence, the lineage information of a particular Spark partition always ends up in the same Kafka partition.

We configured two topics: (1) Topic lineage-node, handling calls to the management API and (2) topic lineage-flow, handling calls to the capture API. The messages from these topics can be processed independently from each other.

The expected number of messages varies significantly between these topics. We expect a much higher load on the lineage-flow topic and, therefore, created thirty partitions for it and only three for the lineage-node topic. To ensure that each Kafka partition receives a dedicated consumer thread, each instance of the lineage backend has ten threads on the lineage-flow topic and one thread on the lineage-node topic. With a replication factor of three on this component, we get the desired number of thirty, respectively, three threads.

The Kafka consumer and producer configuration is described in more detail in Section 4.3.

4.1.3 Lineage Processing

The lineage processing system acts as a Kafka consumer. It parses, validates, and persists coarse- and fine-grained data lineage as a connected graph. Figure 4.2 visualises the architecture as an Unified Modeling Language (UML) class diagram. Messages are consumed over the *KafkaConsumerService*. The corresponding service, LineageNodeService or LineageFlowService, is invoked depending on the topic.

The LineageNodeService is responsible for generating the LineageNodeEntity from a LineageNodeRegistration and establishing a connection between two LineageNode entities, as defined in the LineageNodeLink.

The LineageFlowService is responsible for generating the LineageFlowEntity from a LineageFlow. For performance reasons, the LineageFlowService always processes a batch of LineageFlows. We will describe the generation and the linking process in more detail in Section 4.4.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

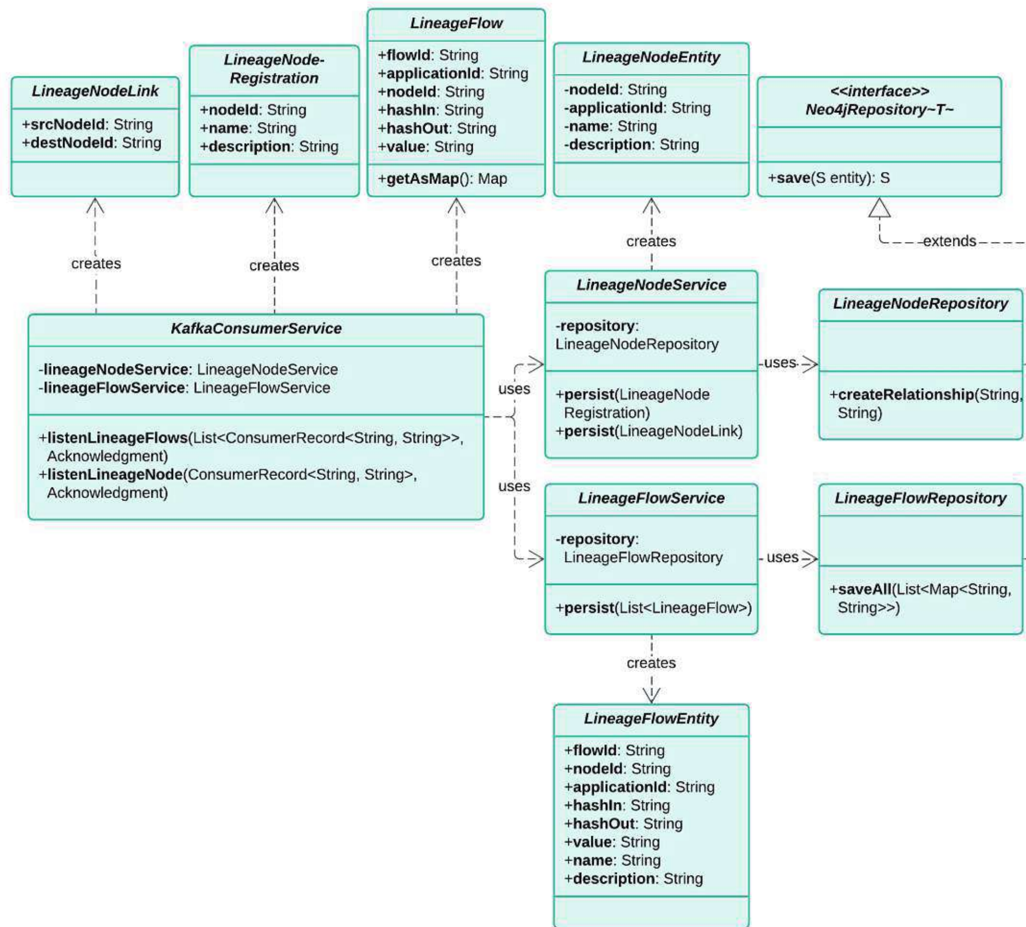


Figure 4.2: Architecture Lineage Processing System

4.1.4 Storage and Presentation

The storage system is also a crucial component, as it has to cope with the high velocity and volume of data. Previous approaches employed relational databases, if any. However, we have realized that graph databases are better suited for our purpose since:

1. Fine-grained and coarse-grained lineage data is displayed as a graph in its natural form.
2. The data is highly connected. Each node in the lineage graph, except for the first and last, has one or more incoming and outgoing connections. In relational databases, the construction and traversal of larger graphs would result in a join-intensive query whose performance deteriorates as the data grows [REW13].
3. With forward and backward tracing, we want to discover relationships between distant items. Such queries can be efficiently executed in a graph database [REW13].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

We decided to use the state-of-the-art graph database Neo4j since:

1. It has high read-and-write performance while maintaining ACID properties [Raj15]. As described in Subsection 2.3.3, the cluster setup weakens the consistency guarantees.
2. It has a native graph storage and graph processing [Raj15].
3. It provides a user interface to visualise and explore the lineage graph [Raj15]
4. It has a powerful query language called Cypher for more advanced use cases like forward and backward tracing [Raj15].

4.2 Collection of Lineage Information

The following section delves into the Spark extension and explains how we extracted a fine-grained data lineage for selected Spark operations.

Similar to the lineage collection in Titian [IST+15], described in Subsection 3.5.2, and Newt [LDY13], described in Subsection 3.4.2, we decided to use a hash-based approach. The fine-grained lineage is constructed by iteratively combining the HashIn and HashOut values of adjacent lineage elements of the type LineageFlow. Hence, the identifier must be carefully crafted.

4.2.1 LineageApi Implementation

In the following, we will describe the LineageApi implementation, depicted in the UML class diagram in Figure 4.3.

The interface ILineageApi corresponds to the API described in Table 4.1 and is implemented by the LineageApi class. The LineageDispatcher holds an instance of the KafkaProducer and encapsulates the methods: (1) *register* for registering a node, (2) *link* to connect two previously registered nodes and (3) *capture* to transfer the provenance association.

Each executor in the Spark cluster has a single instance of the LineageApi and LineageDispatcher, which are shared across the tasks. The LineageDispatcher opens a connection to the Kafka cluster using the KafkaProducer and transmits the serialised LNodeRegistration, LNodeLink or LFlow objects. Hence, every executor has a dedicated connection to the message broker. In the case of multiple tasks running on a single executor, the connection is shared between them. With this in mind, we ensured thread safety by using thread-local variables where necessary.

It is important to note that the LFlow has an additional method *toCsvString()*. We could minimise the size of the transferred messages by serializing the LFlow class to a semicolon-separated CSV string instead of a JavaScript Object Notation (JSON). This

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

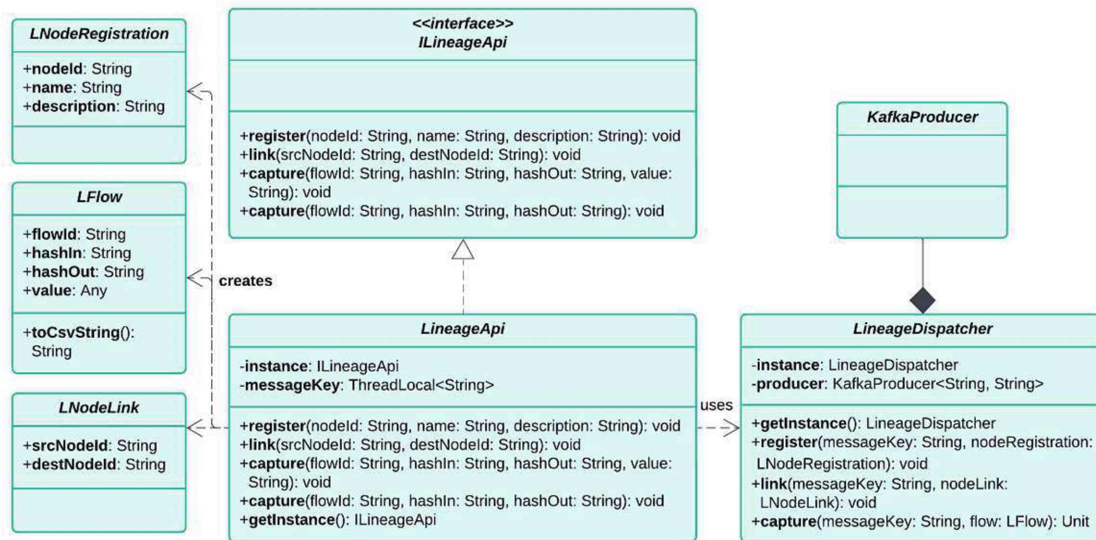


Figure 4.3: Class Diagram Showing the LineageApi

makes sense due to the vast amount of data expected on the lineage-flow topic. As already mentioned, we only expect a few messages on the lineage-node topic. Hence, we decided to serialize the LNodeRegistartaion and LNodeLink classes in JSON format for better readability.

Identifier

To use the LineageApi, we must construct two necessary identifiers: (1) a globally unique NodeId and (2) a globally unique FlowId.

Identifier	Description	Purpose
App.Id	Random identifier	Globally identify a Spark application
RddId	Consecutive numbers, starting from zero for each application	Identify a transformation within a Spark application
NodeId	ApplicationId #RddId	Globally identify a transformation within a Spark application
RecordId	Partition number and offset within that partition	Identify an input within a Spark application
FlowId	ApplicationId #RddId #RecordId	Globally identify an input within a Spark application at a specific transformation

Table 4.2: Identifier used in the LineageApi

As shown in Table 4.2, in our extension of the Spark framework, the NodeId comprises

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

the ApplicationId and the RddId. This combination yields a globally unique identifier for a transformation within a Spark application. Hence, for the LineageNode. The FlowId additionally includes the RecordId to identify an input value globally at a specific transformation in a Spark application. Hence, it serves as an identifier for the LineageFlow.

4.2.2 Lineage RDD

Like Titian, developed by Interlandi et al. [IST+15], we introduced LRDDs and the LineageContext to record the data lineage. In our concrete example, we introduced the following five LRDD types :

- HadoopLRDD (HadoopRDD): Reads a file line-by-line and emits the lineage information *(read#PARTITION#ROW, HASH_OUT)* for each read row.
- MapPartitionsLRDD (MapPartitions-RDD): Tracks the lineage of simple map and filter operations by emitting *(HASH_IN, HASH_OUT)*.
- FlatMapPartitionsLRDD (MapPartitions-RDD): Tracks the lineage of flatMap operations with a fan-out by emitting *(HASH_IN, HASH_OUT_X)* for each output value *X*.
- ShuffledLRDD (ShuffledRDD): Combines multiple LineageFlows during a reduceByKey operation by emitting *(HASH_IN, NODE_ID#REDUCE_KEY)* in the preceding RDD and *(PREV_NODE_ID#REDUCE_KEY, HASH_OUT)* after the successful shuffle operation.
- PersistLRDD (MapPartitions-RDD): Persists the processed data line-by-line and emits the tuple *(HASH_IN, write#PARTITION#ROW)* for each persisted entry.

Each LRDD inherits the core functionalities from its pendant. Methods relevant to the lineage capture are overwritten. The LineageContext represents the entry point to the lineage capture. When reading a file from the HDFS, it creates an instance of HadoopLRDD instead of *HadoopRDD*.

Figure 4.4 illustrates the implementation of the MapPartitionsLRDD using a UML class diagram. It inherits from the MapPartitionsRDD and the lineage trait, which, in turn, inherit from the RDD class.

```
1 override def filter(f: T ⇒ Boolean): Lineage[T] = {
2   val cleanF = sparkContext.clean(f)
3   new MapPartitionsLRDD[T, T](this, (_, _, iter) ⇒ iter.filter(
      cleanF),
4     preservesPartitioning = true, name = "Filter")
5 }
```

Listing 4.1: Overwritten Filter Method in Lineage Trait

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

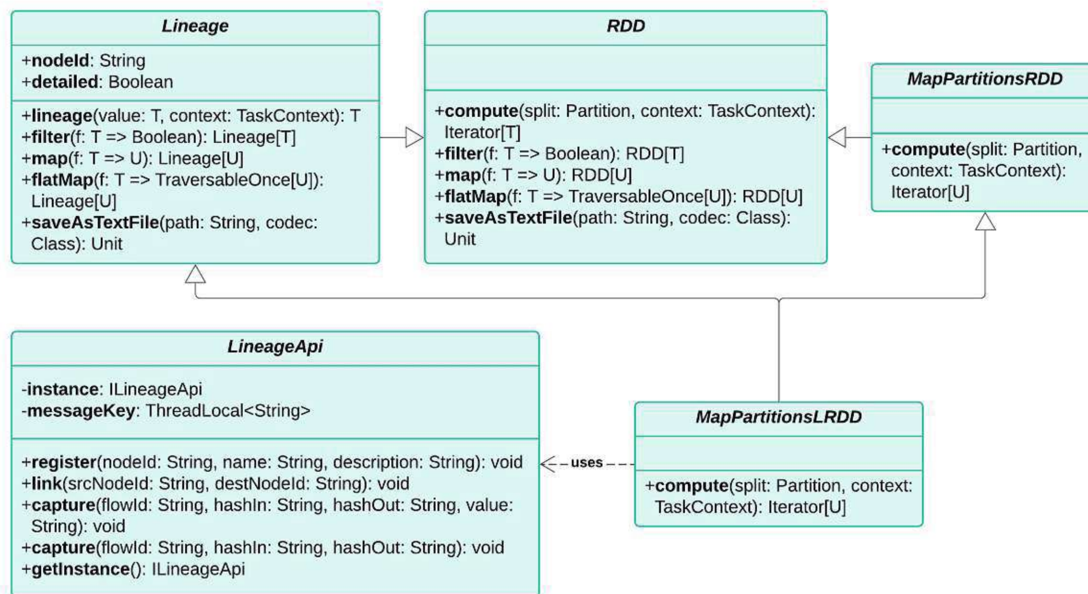


Figure 4.4: LRDD Structure for MapPartitionsLRDD

The lineage trait has a special role since it provides a standard implementation for capturing the lineage and overrides the methods filter, map, flatMap and saveAsTextFile. Within the overwritten methods, we create an instance of the LRDD instead of the RDD, as shown in Listing 4.1 for the filter method. Generating an instance of LRDD also triggers the methods register and link from the LineageApi.

```
override def compute(split: Partition, context: TaskContext):
    Iterator[U] =
  f(context, split.index, firstParent[T].iterator(split, context))
```

Listing 4.2: RDD Compute Method

```
override def compute(split: Partition, context: TaskContext):
    Iterator[U] = {
  super.compute(split, context).map(v => lineage(v, context))
}
```

Listing 4.3: Overwritten Compute Method in MapPartitionsLRDD

The compute method defined in Spark's RDD class is essential for collecting the data lineage. As shown in Listing 4.2 it passes the partition and the TaskContext to its first parent and wraps the returned iterator in its own, which results in a chain of iterators. LRDDs intercept this process by overriding the compute method and plugging a map operation between two iterators, passing the value and the TaskContext to a lineage method defined in the lineage trait as shown in Listing 4.3.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The standard implementation of the lineage method creates a new hash according to a predefined function and subsequently invokes the capture method from the LineageApi. Recall that the capture method takes four arguments: FlowId, HashIn, HashOut and an optional value. The FlowId is constructed from the NodeId of the current RDD and the RecordId stored in the TaskContext. The HashIn is read from the TaskContext, the HashOut is generated, and the value is extracted from the passed value object.

Finally, we replace the HashIn in the TaskContext with the generated HashOut since it is used as HashIn in the next RDD and return the unmodified value. It is important to note that the lineage method is slightly different for FlatMapPartitionsLRDD, PersistLRDD and ShuffledLRDD, but the concept remains the same.

4.2.3 Adjustable Granularity

To reduce the amount of collected lineage information, we implemented two modes of operation. By using the configuration property *spark.rdd.lineage.detailed* users may turn off the collection of intermediate results and inter-stage transformations.

Consequently, when the detailed lineage is false, our implementation will only track the lineage between stages, omitting the transformations within a stage. This information is still sufficient for backward and forward tracing.

The flag is part of the lineage trait and can be observed in Figure 4.4.

4.2.4 Narrow Transformations

As proposed by Interlandi et al. [IST+15], the TaskContext propagates the identifier during narrow transformations. This is possible because the processing within a single stage is sequential. Hence, one row after the other is read and completely processed.

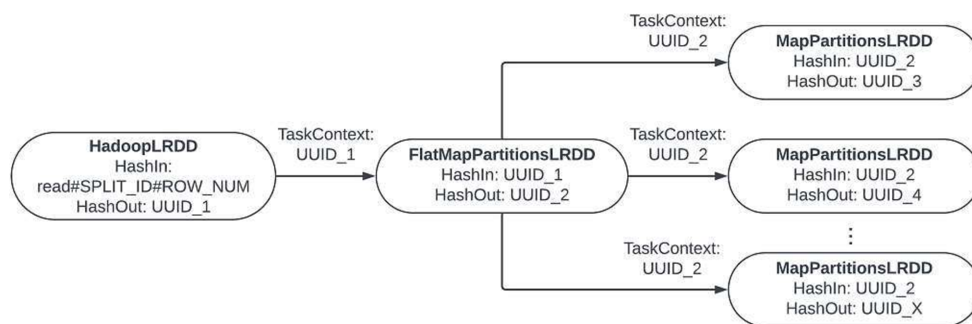


Figure 4.5: Lineage Collection Narrow Transformations

Figure 4.5 shows the propagation of identifiers between narrow transformations. The initial task (HadoopLRDD) links the read line by emitting the tuple *(hashIn = read #{SPLIT_ID} #{ROW_NUM}, hashOut = {UUID_1})*. It stores the HashOut in the TaskContext used as HashIn by the consecutive task.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The generation of the HashOut value depends on the specific RDD type. For RDDs producing a single output record, it is sufficient to create a single Universally Unique Identifier (UUID) and persist it in the TaskContext. RDDs producing multiple output records, e.g., FlatMapPartitionsLRDDs create one UUID for each emitted record and link it to the initial HashIn.

4.2.5 Wide Transformations

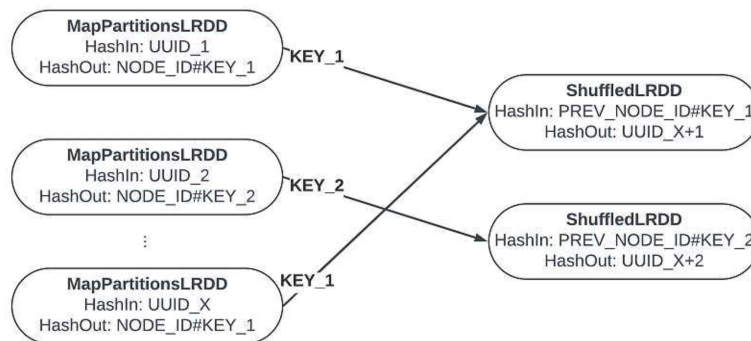


Figure 4.6: Lineage Collection Wide Transformations

A shuffle operation might take place between consecutive stages. It combines the data by their key, aggregating the value according to a user-defined aggregation function. Compared to previous approaches, e.g. Titian [IST+15], the lineage information is not drained into a HashMap nor shuffled with the data. As depicted in Figure 4.6, the HashOut comprises the NodeId and the ShuffleKey. In the subsequent stage, the ShuffledLRDD receives the combined values with their ShuffleKey and constructs the corresponding HashIn using the previous NodeId and the ShuffleKey. Using this information, it is straightforward to establish a link between the lineage of consecutive stages.

4.3 Transmission of Lineage Information

The transmission of lineage information is critical, as it must process a vast amount of messages and act as a buffer between producers and consumers.

4.3.1 At Least Once Delivery Guarantee

As described in Subsection 2.3.2 there are three different types of delivery guarantees. Since our system was designed to deal with reoccurring messages, we can weaken the necessary message delivery guarantees to at least once semantics.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Kafka Consumer

On the consumer side, we want to acknowledge messages manually, so we set the *enable.auto.commit* to false and the *ack.mode* of the ConcurrentKafkaListenerContainer-Factory to *MANUAL_IMMEDIATE*. [Apa, Sprb]

Kafka Producer

On the producer side, we set *retries* to three and *retry.backoff.ms* to 500ms, which means that in case of an error, we resend a message up to three times and wait for 500ms before attempting a retry. Furthermore, we set *acks* to 1, meaning we wait for the partition's leader to acknowledge the message and *enable.idempotence* to false since we don't want to deduplicate messages in the stream automatically. [Apa]

4.3.2 Processing in Batches

Processing each node individually is suboptimal in terms of performance. This applies equally to both consumer and producer applications.

Kafka Consumer

We want to process multiple messages within a single transaction on the consumer side. However, to ensure the delivery guarantees described above, it is not practical to manually cache and acknowledge the messages before processing. Therefore, a different approach was necessary.

Using Apache Kafka for Spring, we used the ConcurrentKafkaListenerContainerFactory with an active batch listener. We increased *max.poll.records* to the desired batch size of 15.000 records. *Max.poll.records* corresponds to the maximum number of records returned when the *poll()* method is invoked. Hence, the application's Kafka listener receives a list with a maximum of 15.000 records at a time and acknowledges the processing of all items as soon as it is completed. [Apa, Spra]

It is important to note that the setting *Max.poll.records* does not influence the consumer's fetching behaviour. Instead, the consumer internally caches the records and incrementally returns them with each call to *poll()*. [Apa]

We further adapted the fetching behaviour by setting the minimum amount of data the server should return within a single fetch request (*fetch.min.bytes*) to 1MB and the maximum duration the server should block if it is not able to satisfy the requirement immediately (*fetch.max.wait.ms*) to one second. [Apa]

Kafka Producer

On the producer side, Kafka provides various settings to transfer messages in batches. We increased the *batch.size* to 500KB and *linger.ms* to 100ms. Hence, for each partition, the producer sends batches with a maximum size of 500KB. The batches may be smaller

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

since the maximum linger time is 100ms. Furthermore, *max.request.size* effectively limits the number of batches the producer can send in a single request to the server. To transfer a maximum of four batches in a single request, we set *max.request.size* to 2MB plus a buffer of 100KB. [Apa]

4.4 Analysis of Lineage Information

The following section deals with converting the raw lineage information into an analysable form. Therefore, we consider the two types, LineageFlow and LineageNode.

4.4.1 LineageNode

The LineageNode is persisted over the LineageNodeRepository, which extends from the Spring Data Neo4jRepository (*Neo4jRepository<LineageNodeEntity, String>*). The derived save method is used to persist new LineageNode entities, transmitted via the register method from the management API.

```
MATCH (src:LineageNodeEntity { nodeId: $srcNodeId })
MATCH (dest:LineageNodeEntity { nodeId: $destNodeId })
MERGE (src)-[:sendsTo]->(dest)
```

Listing 4.4: Insert and Link a LineageNode

Listing 4.4 shows the query that links two LineageNode entities using their NodeId. Hence, it processes the calls to the link method of the management API. If both the source and the destination nodes are found in Line 1/2, an association with the name sendsTo is created in Line 3.

4.4.2 LineageFlow

The LineageFlow is persisted over the LineageFlowRepository, which extends from the Spring Data Neo4jRepository (*Neo4jRepository<LineageFlow, String>*).

```
UNWIND $serializedEntities AS item
MERGE (current:LineageFlow { flowId: item.flowId })
SET current.nodeId = item.nodeId,
    current.applicationId = item.appId,
    current.hashIn = item.hashIn,
    current.hashOut = item.hashOut,
    current.val = item.val
```

Listing 4.5: Cypher Query to Insert LineageFlow Entities

Listing 4.5 illustrates the query for persisting LineageFlow entities in batches using the *UNWIND* command to improve query performance. In Line 2, we use the *MERGE* statement with the FlowId to create new or override existing nodes. As we will explain

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

in Section 4.6.2, this is an important property for achieving fault tolerance and handling retries of the Spark application. Using the *SET* command, we set the remaining properties in Lines 3 to 8.

```
CALL apoc.periodic.iterate(
  "MATCH (current:LineageFlow {applicationId: $applicationId}) RETURN
      current",
  "MATCH (successor:LineageFlow {applicationId: $applicationId,
     hashIn: current.hashOut})
   MERGE (current)-[:flow]->(successor)",
  {batchSize:10000, params: {applicationId: $applicationId}}
)
```

Listing 4.6: Cypher Query to Link LineageFlow Entities

After all LineageNode entities have been successfully saved, they must be linked with each other according to their HashIn/HashOut values. Due to the computational complexity of the linking process, it must be executed as an Awesome Procedures On Cypher (APOC) query. In Line 2 of Listing 4.6, we retrieve all LineageFlow nodes with a given ApplicationId. Using the *apoc.periodic.iterate* query, we execute the second part (Lines 3-4) for each node retrieved in the first part. In Line 3, the objective is to identify predecessor nodes whose HashOut value is identical to the current HashIn value. If a match is found, an association is created in Line 4.

It is important to note that an additional verification of the ApplicationId is conducted before creating an association. This is necessary because the hash alone does not have to be unique. To illustrate this point, consider the case of the HashIn *read#0#1*, which may occur in multiple applications. This restriction also allows us to reduce the size of the hashes without losing collision resistance.

```
UNWIND $props AS item
MERGE (current:LineageFlow { flowId: item.flowId })
SET current.applicationId = item.appId, current.nodeId = item.nodeId,
    current.hashIn = item.hashIn, current.hashOut = item.hashOut,
    current.val = item.val
WITH current, item
MATCH (pred:LineageFlow { applicationId: item.appId, hashOut: item.
   hashIn })
MERGE (pred)-[:flow]→(current)
WITH current, item
MATCH (succ:LineageFlow { applicationId: item.appId, hashIn: item.
   hashOut })
MERGE (current)-[:flow]→(succ)
```

Listing 4.7: Cypher Query to Insert and Link LineageFlow Entities

Listing 4.7 shows an alternative approach, creating and linking LineageFlow entities in a single transaction. While this seems a simple alternative, it comes with a few problems described in the following:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

- Starting a new stage, nodes may be added to different Kafka partitions. Hence, they may be processed immediately and potentially before the nodes of a previous stage. Therefore, it is necessary to check for both predecessors and successors and establish a connection if it does not exist.

- Creating multiple nodes in a single transaction is necessary for performance reasons. However, creating a relationship also sets an exclusive lock on predecessor and successor nodes, leading to locking exceptions for other clients executing the same query for a predecessor or successor.

- If two related nodes are created simultaneously in different transactions, their relationship is not established because neither of the nodes is aware of the existence of the other.

Besides establishing relationships between the LineageFlow nodes, we can also enrich them further. The name and description of a transformation are only part of the LineageNode. They are not included in the LineageFlow to minimise redundant information and, thus, the total amount of data transferred. However, we can easily add the property afterwards to increase the readability of the fine-grained data lineage.

```
1 CALL apoc.periodic.iterate(
2   "MATCH (current:LineageFlow {applicationId: $applicationId}) RETURN
      current",
3   "MATCH (meta:LineageNode {applicationId: $applicationId, nodeId:
     current.nodeId})
4    SET current.name = meta.name, current.description = meta.
      description",
5   {batchSize:10000, parallel: true, params: {applicationId:
     $applicationId}}
6 )
```

Listing 4.8: Cypher Query to add Name and Description to LineageFlow Nodes

Listing 4.8 shows a Cypher query matching nodes of type LineageNode and LineageFlow based on their ApplicationId and NodeId. If a match is found, we copy the name and description property from the LineageNode to the LineageFlow in Line 4. Due to the amount of data, this query is also executed as an APOC procedure.

4.5 System Capabilities

The following section describes how our system can provide all the core functionalities of a lineage tracking system.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4.5.1 Lineage Graph

The coarse-grained lineage graph gives a broad overview of the transformations performed, the origin of the data, and its target. Each run of a Spark application has its lineage graph, corresponding to the created LRDDs.

```
1 MATCH (n:LineageNodeEntity)
2 WHERE n.applicationId = $applicationId
3 RETURN n
```

Listing 4.9: Query Lineage Graph

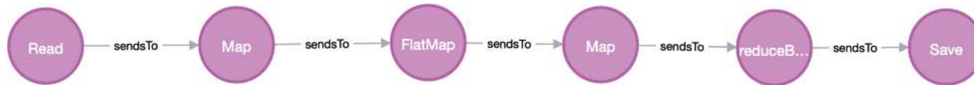


Figure 4.7: Lineage Graph of a WordCount Example in Bloom

The Cypher query in Listing 4.9 shows a simple example of retrieving an application's lineage graph with a given ApplicationId. Figure 4.7 visualises the result in Neo4j Bloom.

4.5.2 Backward Tracing

Backward tracing is essential for finding the root causes of a wrong value. Given a specific value in the result set, it determines all inputs that contributed to that result.

```
1 MATCH (end:LineageFlow)
2 WHERE end.applicationId= $applicationId AND end.hashOut= 'write#0#1'
3 MATCH path = (end)<-[*0..]-(flow)
4 RETURN path
```

Listing 4.10: Query Backward Tracing

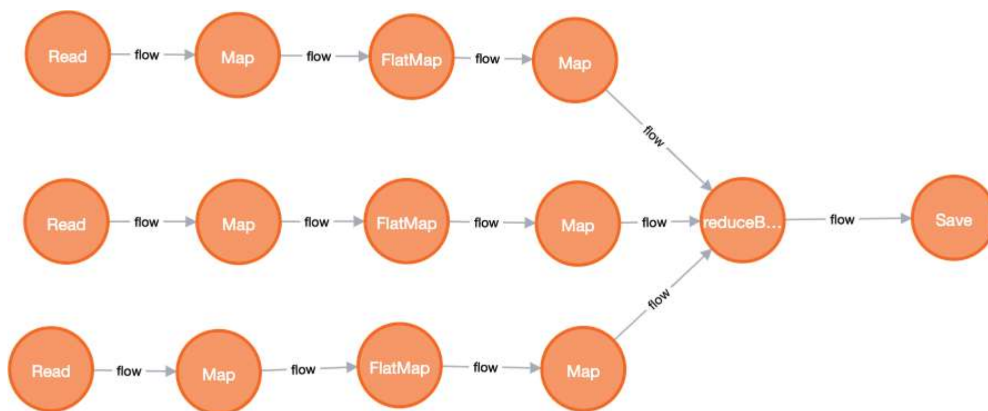


Figure 4.8: Backward Tracing of a WordCount Example in Bloom

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

The Cypher query in Listing 4.10 traces back the value stored in row one of partition zero of the result set (*write#0#1*). The path in Line 3 recursively follows all incoming edges until we reach a read operation, only containing outgoing edges. Figure 4.8 visualises the result in Neo4j Bloom.

4.5.3 Forward Tracing

Forward tracing is essential for finding all values in the result set affected by a wrong input value. It forms the basis for partial updates.

```
MATCH (start:LineageFlow)
WHERE start.applicationId = $applicationId AND
        start.hashIn = 'read#0#6'
MATCH path = (start)-[*0..]→(flow)
RETURN path
```

Listing 4.11: Query Forward Tracing

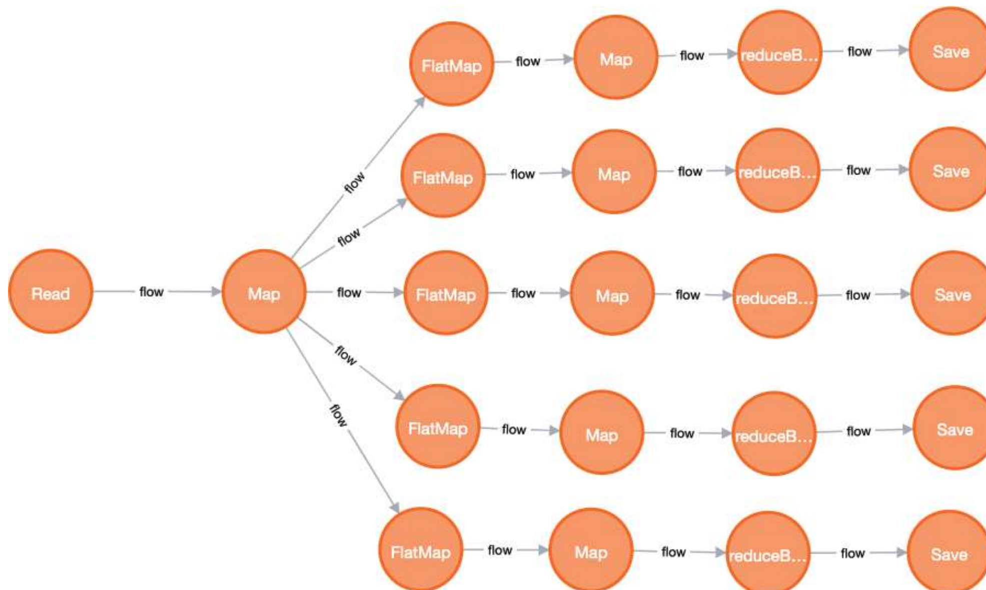


Figure 4.9: Forward Tracing of a WordCount Example in Bloom

The Cypher query 4.11, determines all values in the result affected by the value stored in row six of partition zero of the input (*read#0#6*). The path in Line 3 recursively follows all outgoing edges until we reach a write operation, only containing incoming edges. Figure 4.9 visualises the result in Neo4j Bloom.

4.5.4 Intermediate Results

Intermediate results can be recorded by setting the config *spark.rdd.lineage.detailed* to true. One option would be to pass *–conf "spark.rdd.lineage.detailed=true"* with the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Spark-Submit command. If this configuration is active, the current value is transferred with the regular lineage information, utilizing the value parameter of the capture API. As a result, the fine-grained data lineage contains the intermediate values of the processing. Figure 4.10 shows a simple example of a WordCount job holding its intermediate results in the value property.

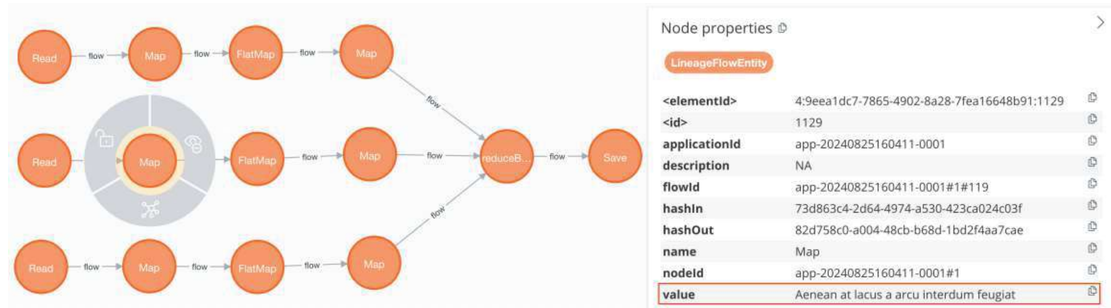


Figure 4.10: Lineage of a Simple WordCount Example with Intermediate Results in Bloom

4.6 Characteristics and Limitations

4.6.1 Limitations

Spark Extension

The instrumentation of a DISC system, e.g. Apache Spark, is accompanied by considerable effort. Due to limited time resources, we couldn't adapt all RDDs. Instead, we focused on the necessary adjustments to collect the data lineage of our use cases, WordCount and Grep. Furthermore, an extensive adaptation of Spark was not required to address our research questions.

Data Immutability

The persisted data lineage holds information about the processed files. It links the first and last element of a LineageFlow with a concrete row in the input/output file. Subsequent modification to these files may corrupt the lineage. Hence, it would be necessary to freeze the data, e.g., by creating snapshots before executing a Spark job or by providing a general versioning mechanism.

Limited Data Volume

Since we decided to employ an external system for data lineage collection to improve reusability across different Big Data platforms, the transmission of lineage information is limited. Depending on the granularity, input data and processing script, the size of the lineage information can quickly surpass that of the original dataset. Hence, we

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

believe capturing the entire lineage of a data processing operation is rarely practical. As an alternative approach, we propose the application of filters prior to the monitored execution. This restricts the lineage information to the potentially problematic subset of the data.

4.6.2 Characteristics

Fault-tolerance

The lineage tracking is transparent to the instrumented framework and does not break its fault-tolerance model. However, in case of a failure, provenance is already emitted and durably saved. We introduced a unique identifier for each lineage element to overcome this issue without the need to buffer the lineage information until the successful completion of a stage. We already described the introduced identifiers in Subsection 4.1.1.

Furthermore, as described in Section 4.1.2, it is essential to note that the Spark PartitionId determines the corresponding Kafka partition. Hence, the lineage information of a Spark partition is always sent to the same Kafka partition and processed sequentially in the lineage backend. In case of a failure, the reprocessing will create new LineageFlow records with the same FlowId, ending up in the same Kafka partition and overriding existing ones during processing in the lineage backend.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 5

Results

This chapter evaluates our lineage tracking system, Lineage Master. In Section 5.1, we will describe the results of our performance evaluation in terms of time and space overhead compared to the native Spark job. Furthermore, we will report the lineage processing time on the backend side.

In Section 5.2, we directly compare our system with existing solutions and draw some conclusions.

5.1 Performance Evaluation

The performance evaluation is based on the three examples, WordCount, Grep and Tweets. We implemented them in Scala and executed the program using the Spark-Submit command, as shown in Listing 5.1.

```
1 podman exec -it spark-master-lineage spark-submit \
2   --class WordCountLineage \
3   --master spark://spark-master-lineage:7077 \
4   --conf "spark.rdd.lineage.detailed=false" \
5   hdfs://namenode:9000/user/root/eval_examples-3.5.3.jar 10MB.txt 10
      MB.txt
```

Listing 5.1: Spark-Submit Command for WordCount with Lineage

The evaluation programs are packed into a Java Archive (JAR) named *eval_examples-3.5.3.jar*. Hence, we reference a concrete class in Line 2. After setting the Spark master address, we pass some configuration properties in Line 4. In this case, we deactivate the detailed lineage capture, as described in Subsection 4.2.3.

5.1.1 Infrastructure

We executed the evaluation on a single machine using Podman. The machine was equipped with 32 CPUs (Intel Xeon Gold 6254 at 3.10GHz), 100GB of Random-Access Memory (RAM), and 100GB of disc storage. The operating system was 64-bit Red Hat Enterprise Linux 9.4. Each worker node had 8GB of RAM and 2 CPUs. All services were placed on a single network to simplify service discovery, minimise latency, and maximize the throughput. Furthermore, we restricted the resource consumption of each container, as shown in Table 5.1.

Service	CPU	RAM	Image	Version
HDFS Cluster				
NameNode	2	4g	bde2020/hadoop-namenode	2.0.0 (Hadoop 3.2.1)
DataNode	2	4g	bde2020/hadoop-datanode	2.0.0 (Hadoop 3.2.1)
Kafka Cluster				
Zookeeper-1	2	4g	confluentinc/cp-zookeeper	7.4.4 (Kafka 3.4.4)
Broker-1	2	7g	confluentinc/cp-kafka	7.4.4 (Kafka 3.4.4)
Broker-2	2	7g	confluentinc/cp-kafka	7.4.4 (Kafka 3.4.4)
Broker-3	2	7g	confluentinc/cp-kafka	7.4.4 (Kafka 3.4.4)
Lineage Backend				
Lineage-1	3	8g	fd/lineage-backend	1.0.x (SpringBoot 3.2.5)
Lineage-2	3	8g	fd/lineage-backend	1.0.x (SpringBoot 3.2.5)
Lineage-3	3	8g	fd/lineage-backend	1.0.x (SpringBoot 3.2.5)
Neo4j				
Neo4j	4	20g	neo4j	5.24.2 (Neo4j 5.24.2)
Spark Cluster with Lineage				
Spark-Master	2	8g	fd/spark-lineage	3.5.3.x (Spark 3.5.3)
Spark-Worker-1	3	10g	fd/spark-lineage	3.5.3.x (Spark 3.5.3)
Spark-Worker-2	3	10g	fd/spark-lineage	3.5.3.x (Spark 3.5.3)
Spark-Worker-3	3	10g	fd/spark-lineage	3.5.3.x (Spark 3.5.3)
Spark Cluster				
Spark-Master	2	8g	fd/spark	3.5.3 (Spark 3.5.3)
Spark-Worker-1	3	10g	fd/spark	3.5.3 (Spark 3.5.3)
Spark-Worker-2	3	10g	fd/spark	3.5.3 (Spark 3.5.3)
Spark-Worker-3	3	10g	fd/spark	3.5.3 (Spark 3.5.3)

fd ... abbreviation for floriansdocker

Table 5.1: Resource Limitations of the Docker Containers

It is important to note that the overall resource consumption exceeds the server's capacity. As a result, we executed each experiment in two phases. In the first phase, we ran the Spark program on the Spark cluster with lineage enabled and stored the messages in

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

the message queue. During this phase, only the HDFS cluster, the Spark cluster, and the Kafka cluster were active. In the second phase, we shut down the Spark cluster and reallocated its resources to the lineage backend and the Neo4j database.

5.1.2 Measurement Method

For Grep and WordCount, the lineage tracking was only conducted at the stage level without intermediate results (spark.rdd.lineage.detailed=false). Regarding the Tweets example, we collected an even more detailed lineage, including inter-stage transformations and intermediate results (spark.rdd.lineage.detailed=true).

Runtime Overhead

The runtime overhead encompasses the additional time necessary for capturing an execution's lineage. To quantify the overhead as a multiplier of Spark execution time, we conducted two experiments for each example (Grep, WordCount, Tweets), one with the modified and one with the unmodified Spark runtime. Furthermore, we executed each experiment three times and calculated the mean of the execution time reported by the Spark master.

Storage Overhead

The storage overhead includes the coarse-grained and fine-grained lineage graph. Regarding the fine-grained lineage graph, we did not further enrich it with a node's name and description as described in Subsection 4.4.2. To quantify the overhead, we calculated the size of all files in the corresponding database using the command *du -hc *store.db**.

Lineage Processing Time

The lineage processing time describes the timespan between the emission of the data lineage by the Spark job and the analysability of the data in the graph database. It can be divided into the node creation and node linking time, as they are two distinct processes. We described them in more detail in Subsection 4.4.2. The node creation time was calculated from the logs. The node linking time was reported by the APOC procedure.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.1.3 Grep

Grep is a simple program that filters a given input file, such as a log file, for a particular term. In the following subsection, we will discuss the algorithm and report some performance metrics related to lineage tracking.

Spark Program

```
val conf = new SparkConf().setAppName("GrepLineage")
val sc = new SparkContext(conf)
val lc = new LineageContext(sc)

val inputPath = "hdfs://namenode:9000/user/root/input/" + args(0)
val outputPath = "hdfs://namenode:9000/user/root/output/" + args(1)
val searchTerm = args(2)

val inputRdd = lc.textFile(inputPath, 30)
val matchedRdd = inputRDD.filter(line ⇒ line.contains(searchTerm))
matchedRdd.saveAsTextFile(outputPath)
```

Listing 5.2: Grep in Scala

Listing 5.2, shows the Spark program executing the Grep example. In Lines 1-3, we generate and wrap the SparkContext within the LineageContext. As mentioned in Section 4.2, the LineageContext serves as an entry point by returning an LRDD instead of an RDD. Lines 5-7 define some essential variables holding the input and output path and the search term. In Line 9, the file is read into 30 partitions, filtered for the search term in Line 10 and saved to the output path in Line 11. It is important to note that the file is read in 30 partitions to optimise the lineage tracking process.

Sample Data

We generated datasets of 100MB, 300MB, 500MB, and 1000MB from a vocabulary of 8000 terms (word1 to word8000) selected based on a Zipf distribution with a distribution parameter equal to 2. Each line in the generated document is between eight and twelve words long. The sample data was used to run the Spark jobs, Grep, and WordCount. Interlandi et al. [IST+15] followed a similar approach to generate the test data, which should improve the comparability of the results.

Runtime

The performance overhead encountered during the execution of the Grep example strongly depends on the search term since it may or may not reduce the number of records. Hence, we analysed the datasets and always chose a word with a frequency equal to the median. Table 5.2 shows the concrete words, including their frequency.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Data Analysis - Grep		
Dataset	Word	Occurrences
10MB	word743	29
50MB	word432	127
100MB	word90	306
300MB	word635	680
500MB	word128	1109
1000MB	word97	1709

Table 5.2: Median Word Frequency in the Sample Datasets

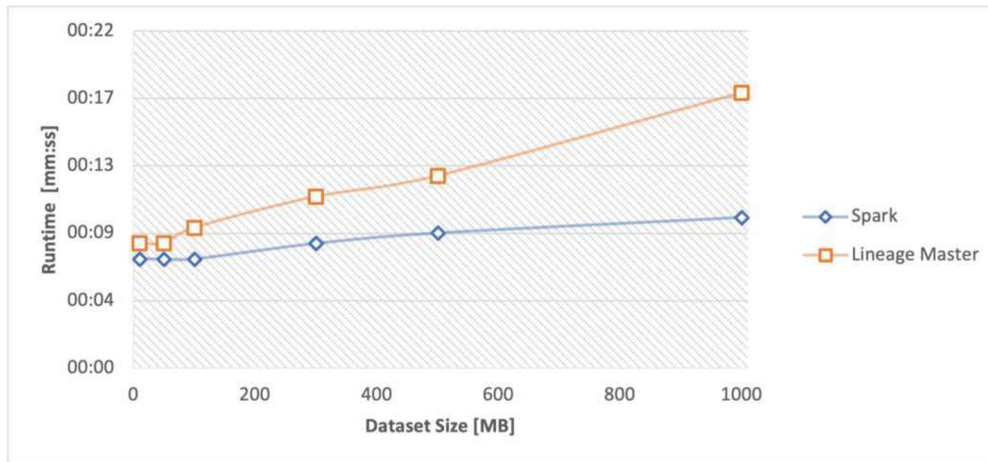


Figure 5.1: Runtime of Spark and Lineage Master for Grep

In Figure 5.1, the orange line with square markers depicts the runtime of Lineage Master, and the blue line with diamond markers shows the runtime of Spark. The overhead keeps within limits since the search term reduces the amount of data. However, the execution time of Lineage Master increases more than that of the unmodified Spark version.

Runtime Overhead - Grep				
Dataset	Lineage Master	Titian-D	RAMP	Newt
10MB	1.14x	/	/	/
50MB	1.14x	/	/	/
100MB	1.29x	/	/	/
300MB	1.38x	/	/	/
500MB	1.42x	1.27x	1.40x	1.58x
1000MB	1.83x	/	/	/

Table 5.3: Runtime of Lineage Master, Titian-D, RAMP and Newt for Grep: Adapted from [IST+15]

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Table 5.3 summarises the results at six dataset sizes and indicates the runtime as a multiplier of the native Spark runtime. For the Grep example, Lineage Master performs worse than Titian-D but better than Newt.

Storage Consumption

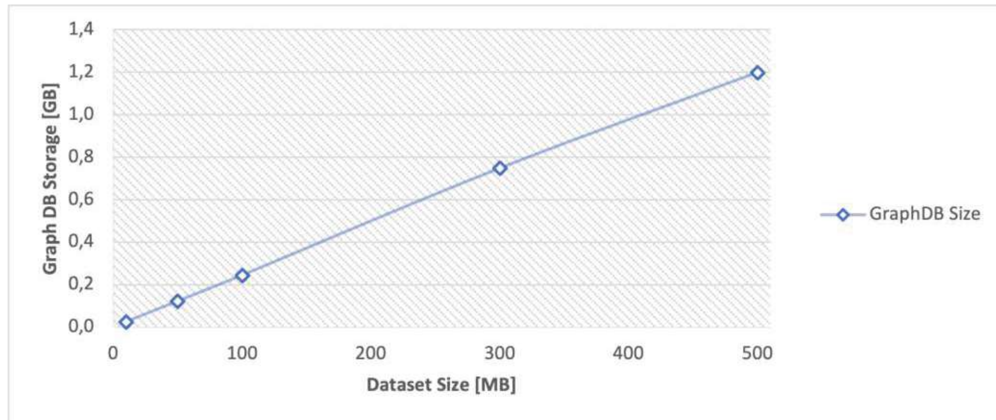


Figure 5.2: Storage Consumption of Lineage Master for Grep

Figure 5.2 visualises Lineage Master's storage consumption for the Grep example. The blue line with diamond markers depicts the size of the Neo4j database. It grew almost linear with the dataset size and was always about 2.5 times the input size.

Lineage Processing Time

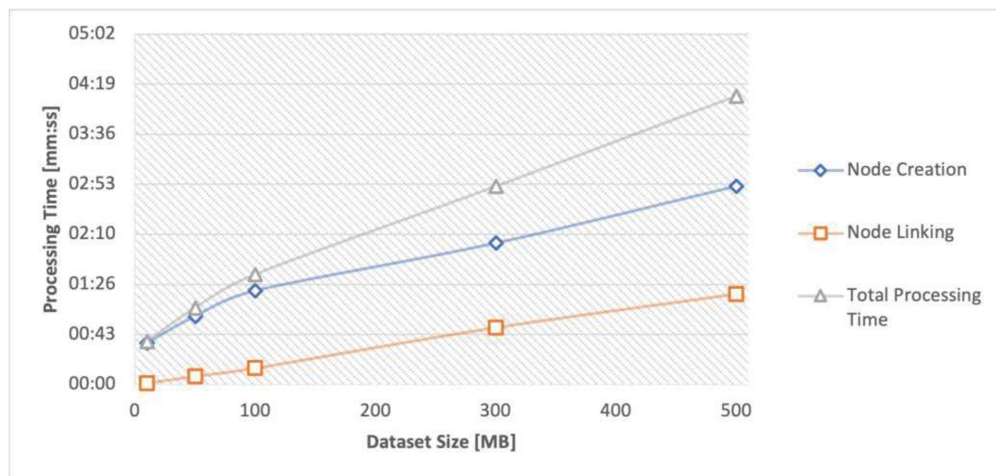


Figure 5.3: Lineage Processing Time of Lineage Master for Grep

Figure 5.3 visualises the time necessary for processing the lineage information produced by executing the Grep example. We distinguish between the creation process, represented

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

by the blue line with diamond markers, and the linking process, represented by the orange line with square markers. For an input dataset of 500MB, we produced a graph database of 1.2GB in size, which took 4 minutes and 9 seconds to complete. Of this time, 2 minutes and 51 seconds were spent on creating the nodes and 1 minute and 18 seconds on linking them.

5.1.4 WordCount

The WordCount example is a classic Hadoop MapReduce application that counts the occurrences of any word in a given input text. It consists of two phases: (1) in the Map phase, the input text is split into individual words. For each word, a key-value pair with the word as a key and the value 1 is emitted. In the subsequent reduce phase, the key-value pairs are aggregated by their key, summing the values to produce the desired count for each word. [Apa24]

In the following subsection, we will discuss the algorithm and report some performance metrics related to lineage tracking.

Spark Program

```scala
val conf = new SparkConf().setAppName("WordCountLineage")
val sc = new SparkContext(conf)
val lc = new LineageContext(sc)

val inputPath = "hdfs://namenode:9000/user/root/input/" + args(0)
val outputPath = "hdfs://namenode:9000/user/root/output/" + args(1)

val inputRdd = lc.textFile(inputPath, 30)
val splitedRdd = inputRdd.flatMap(l => l.split(" "))
val mappedRdd = splitedRdd.map(w => (w, 1))
val reducedRdd = mappedRdd.reduceByKey(_ + _)
reducedRdd.saveAsTextFile(outputPath)
```

Listing 5.3: WordCount in Scala

Listing 5.3, shows the WordCount example written in Scala. In Lines 1-3, the LineageContext is created from the SparkContext, which represents the entry point for the monitored execution. Lines 5 and 6 define some essential variables holding the input and output path. The map phase is executed by Lines 8 - 10. While the file is read in Line 8, the content is split up by whitespace characters in Line 9. In Line 10, each word receives a count of 1, and the pairs (word,1) are reduced by key in Line 11. Finally, in Line 12, the produced file is saved into an HDFS directory.

Sample Data

For WordCount, we used the same sample data as for Grep, described in Subsection 5.1.3.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Runtime

Figure 5.4: Runtime of Spark and Lineage Master for WordCount

The performance overhead encountered during the execution of the WordCount example was significant. This is mainly due to the fan-out when splitting a sentence into a sequence of words. In our sample data, each sentence has between eight and twelve words, which leads to an average of ten words per sentence. Since each sentence word might end up in a different result row, we must track each word separately. This leads to an eightfold to twelvefold increase in lineage information during execution.

In Figure 5.4, the orange line with square markers depicts the runtime of Lineage Master, and the blue line with diamond markers shows the runtime of Spark. For this example, the runtime of Lineage Master is a multiple of the runtime of the unmodified Spark version.

Runtime Overhead - WordCount				
Dataset	Lineage Master	Titian-D	RAMP	Newt
10MB	1.10x	/	/	/
50MB	1.47x	/	/	/
100MB	1.85x	/	/	/
300MB	3.11x	/	/	/
500MB	4.03x	1.18x	1.34x	1.72x
1000MB	6.09x	/	/	/

Table 5.4: Runtime of Lineage Master, Titian-D, RAMP and Newt for WordCount: Adapted from [IST+15]

Table 5.4 summarises the results at six dataset sizes and indicates the runtime as a multiplier of the native Spark runtime. These metrics make it even more evident that the system has considerable performance problems in this example. For a dataset of

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

500MB, the runtime of Lineage Master was four times the native Spark job execution time. Compared to the other implementations, this system ranks last in performance.

Storage Consumption

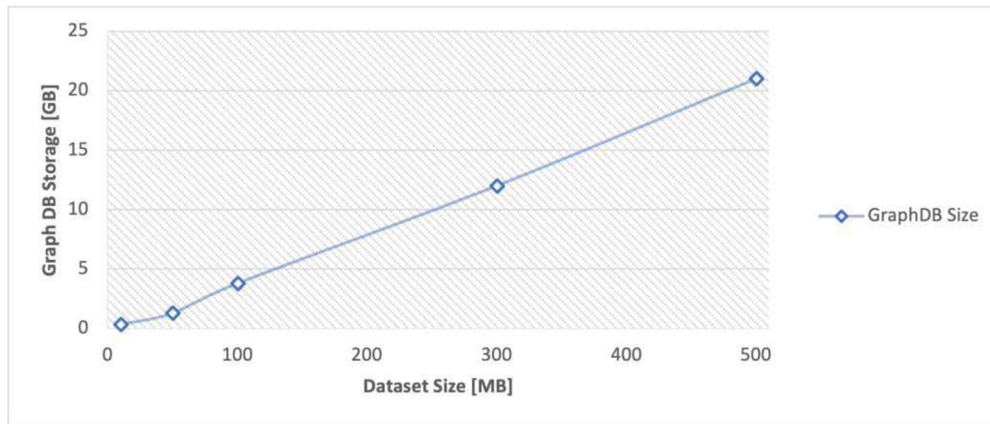


Figure 5.5: Storage Consumption of Lineage Master for WordCount

Figure 5.2 visualises Lineage Master's storage consumption for the WordCount example. The previously described fan-out and tracking of each word is also problematic for the database size, depicted as the blue line with diamond markers. An input of 500MB resulted in a database size of 21GB, representing a 42-fold increase.

Lineage Processing Time

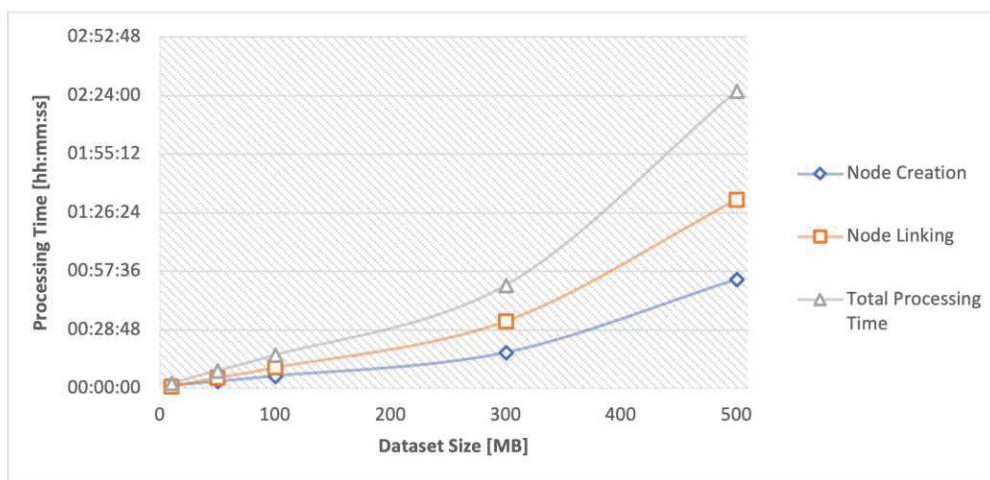


Figure 5.6: Lineage Processing Time of Lineage Master for WordCount

The vast amount of lineage information also significantly impacts lineage processing time, as shown in Figure 5.6. As before, we distinguish between the creation process,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

represented by the blue line with diamond markers, and the linking process, represented by the orange line with square markers. For an input dataset of 500MB, we produced a graph database of 21GB in size, which took 2 hours, 26 minutes and 17 seconds to complete. Of this time, 53 minutes and 39 seconds were spent on creating the nodes and 1 hour, 32 minutes and 38 seconds on linking them.

With increasing lineage information, processing time is also rising faster. We attribute this to the limited resources available on the Neo4j side and the continuous growth in index entries.

5.1.5 Tweets - Real World Example

To illustrate the system's applicability in a realistic context, we have created an evaluation based on the example outlined in Subsection 2.4.1.

Let's suppose we receive a large file containing various tweets. We want to filter all movie-related contributions, extract the movie name, and rate the text as negative and/or positive. Subsequently, we want to count the number of positive/negative comments for each film and thus compute a rating. This can be implemented as a sequence of map, filter, and reduceByKey operations. Figure 5.7 shows the corresponding coarse-grained lineage graph generated by Lineage Master and visualised in Bloom. For illustrative purposes, the corresponding code snippets have been added to each node in the graph. Additionally, they can be found in Lines 12 to 23 of Listing 5.4.

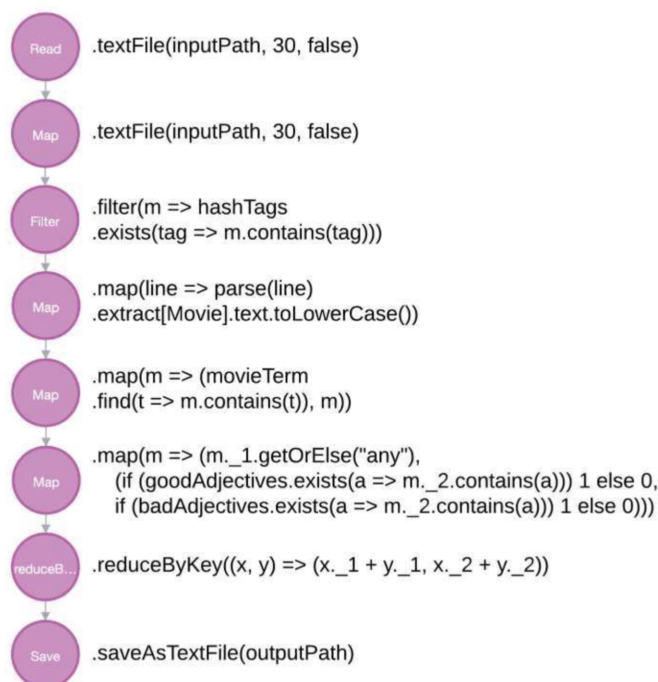


Figure 5.7: Tweets Lineage Graph

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Let's further suppose that we find our favourite movie, Breaking Bad, with a very poor rating. To identify a potential problem, we want to review the individual processing steps in detail. Hence, in this example, we want a fine-grained data lineage, including intermediate results. This can be achieved by starting the program with the configuration $spark.rdd.lineage.detailed = true$.

Spark Program

```
1  val conf = new SparkConf().setAppName("TweetLineage")
2  val sc = new SparkContext(conf)
3  val lc = new LineageContext(sc)
4
5  val inputPath = "hdfs://namenode:9000/user/root/input/" + args(0)
6  val outputPath = "hdfs://namenode:9000/user/root/output/" + args(1)
7  val movieTerm = List("badboys","inception","gangstar","breakingbad","
       matrix","interstellar", "thegodfather","pulpfiction","fightclub","
       thedarkknight")
8  val hashTags = List("#movie","#film","#cinema","#hollywood","#
       blockbuster")
9  val goodAdjectives = List("incredible","great","top-notch","
       masterpiece","perfect")
10 val badAdjectives = List("confusing", "overrated", "unsatisfied", "
       poor", "bad", "hard")
11
12 val inputRdd = lc.textFile(inputPath, 30, false)
13 val filteredRdd = inputRdd.filter(m => hashTags.exists(tag => m.
       contains(tag)))
14
15 val textRdd = filteredRdd.map(line => parse(line).extract[Movie].text
       .toLowerCase())
16 val movieTextRdd = textRdd.map(m => (movieTerm.find(t => m.contains(t
       )), m))
17
18 val ratedRdd = movieTextRdd.map(m => (m._1.getOrElse("any"),
19     (if (goodAdjectives.exists(a => m._2.contains(a))) 1 else 0,
20     if (badAdjectives.exists(a => m._2.contains(a))) 1 else 0)))
21
22 val reducedRdd = ratedRdd.reduceByKey((x, y) => (x._1 + y._1, x._2 +
       y._2))
23 reducedRdd.saveAsTextFile(outputPath)
```

Listing 5.4: Tweets Example

Listing 5.4 implements the process in Scala. In Lines 1-3, the LineageContext is created from the SparkContext, which represents the entry point for the monitored execution. Lines 5-10 define important variables holding the input and output path, the movies we are searching for, relevant hashtags, and good/bad adjectives to categorize the text.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

In Line 12, the file is read into thirty partitions and filtered for relevant hashtags in Line 13. In the subsequent process, only the text is necessary to perform the remaining operations. Hence, we remove all other attributes in Line 15 and try to extract a movie name from the text in Line 16. The MovieTextRdd then holds tuples with the movie name and the text.

In Lines 18 - 20, we categorize the text as negative and/or positive by matching it against a list of good and bad adjectives. Finally, in Line 22, we reduce the tuples by key, the movie name in our example, and sum up the ratings. In Line 23, we save the generated output to the HDFS.

Sample Data

Using a Python script, we created synthetical datasets with 500MB, 1000MB, 2000MB, 4000MB, and 5000MB. The generator was programmed so that one in 100 tweets is about a movie. It then generates a JSON for each tweet with the following properties: (1) id of the tweet, (2) username of the tweeter, (3) location from where the tweet was sent, (4) text message of the tweet, (5) timestamp when the tweet was sent, (6) likes the tweet received, (7) retweets of the tweet, (8) replies to the tweet, (9) hashtags of the tweet, (10) language of the tweet, (11) device the tweet was sent from, (12) is_verified, indicating if the tweeter account has been checked, (13) follower_count of the tweeter and (14) following_count of the tweeter. Each JSON is written in a single line and is about 420 bytes large.

The following represents a movie-related example:

```
1 {"id": "4e48f59c-55a3-4cf9-990f-24e652765a45", "username": "
    sqx7rlrirfjszd", "location": "Los Angeles, USA", "text": "The
    concept was interesting, but execution was poor. I couldn't
    connect with the characters. I really wanted to like Matrix, but
    ... #Disappointed", "timestamp": "2024-03-15T22:45:37.638564", "
    likes": 630, "retweets": 126, "replies": 37, "hashtags": "#
    hollywood,#movie,#blockbuster", "language": "en", "device": "
    Twitter for Android", "is_verified": false, "follower_count":
    2173, "following_count": 301}
```

Listing 5.5: Exemplary Tweet Related to the Movie Matrix

Important Remarks

In our example, we can significantly reduce the amount of data during the filter operation since only one per cent of all tweets are about films.

In Line 12 of Listing 5.4, we disable the intermediate results for the initial read operation as it would simply contain the whole file content. Therefore, the first operation capturing intermediate results is the filter operation. This is an important consideration since we can reduce the amount of data to one per cent during the filter operation.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Runtime

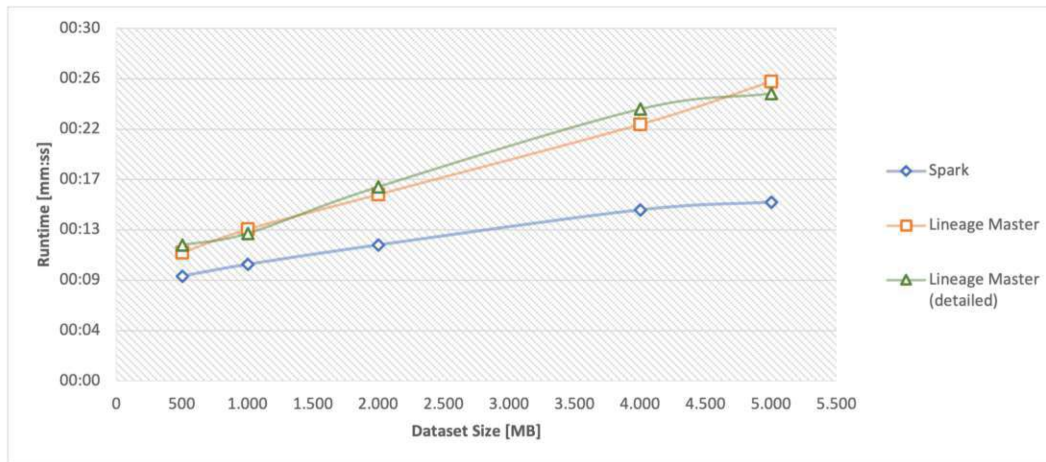


Figure 5.8: Runtime of Spark, Lineage Master and Lineage Master with Details for Tweets

Figure 5.8 compares the runtime of Spark, Lineage Master and Lineage Master with details. The blue line with diamond markings represents the Spark runtime. The runtime of Lineage Master is shown with an orange line and square markers, and its detailed counterpart with a green line and triangular markers. In our example, the additional capture of inter-stage transformations and the transfer of intermediate results did not significantly impact the runtime.

Runtime Overhead - Tweets		
Dataset	Lineage Master	Lineage Master Detailed
500MB	1.22x	1.30x
1.000MB	1.30x	1.27x
2.000MB	1.37x	1.43x
4.000MB	1.50x	1.59x
5.000MB	1.67x	1.61x

Table 5.5: Runtime of Lineage Master and Lineage Master with Details for Tweets

Table 5.5 summarises the results at five dataset sizes and indicates the runtime as a multiplier of the native Spark runtime. Generally, the overhead remains acceptable, even for larger datasets.

Storage Consumption

Figure 5.9 visualises Lineage Master's storage consumption for the Tweets example. The orange line with square markers depicts the size of the Neo4j database, which grows approximately linearly with the input dataset. It is essential to repeat that the fine-

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

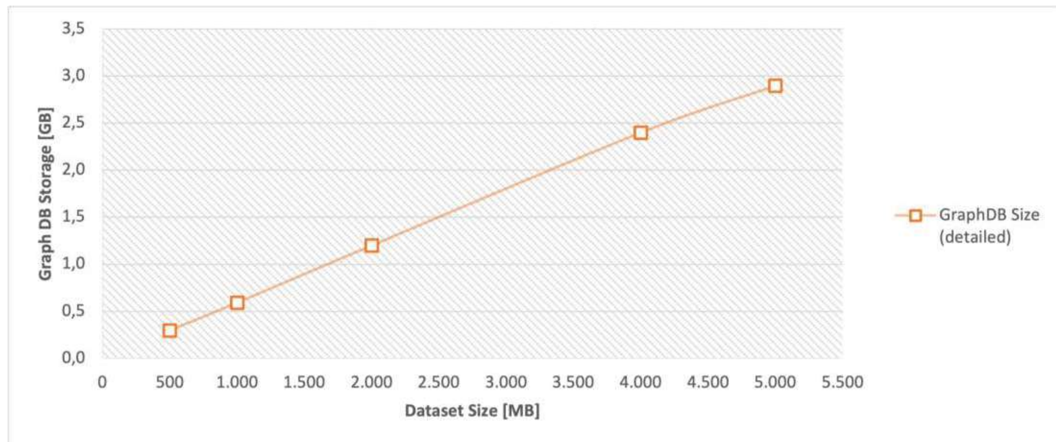


Figure 5.9: Storage Consumption of Lineage Master for Tweets

grained lineage graph in this example also encompasses inter-stage transformations and intermediate results for all operations except the initial read.

Lineage Processing Time

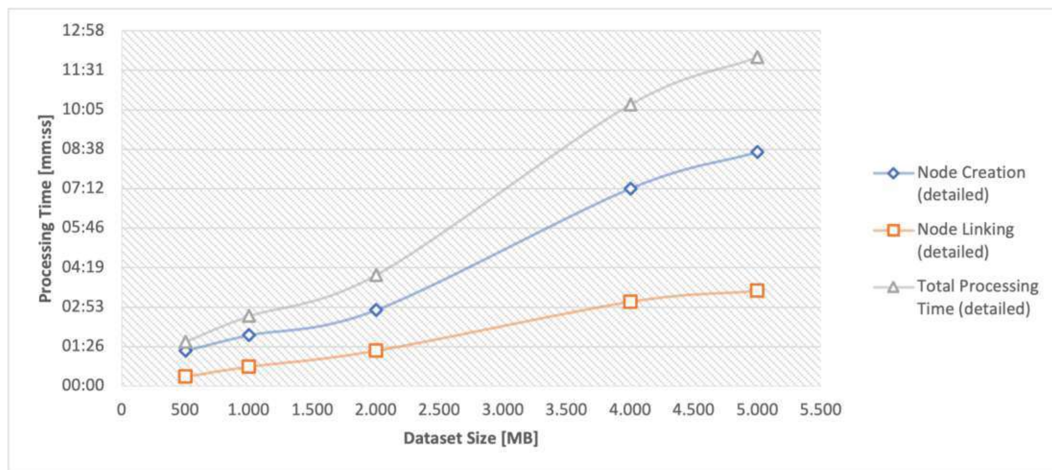


Figure 5.10: Lineage Processing Time of Lineage Master for Tweets

As before, we distinguish between the creation process, represented by the blue line with diamond markers, and the linking process, represented by the orange line with square markers. For an input dataset of 5GB, we generated a graph database of 2.9GB, which took 12 minutes to complete. Of this time, 8 minutes and 32 seconds were spent on creating the nodes and 3 minutes and 28 seconds on linking them.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Using the Data Lineage

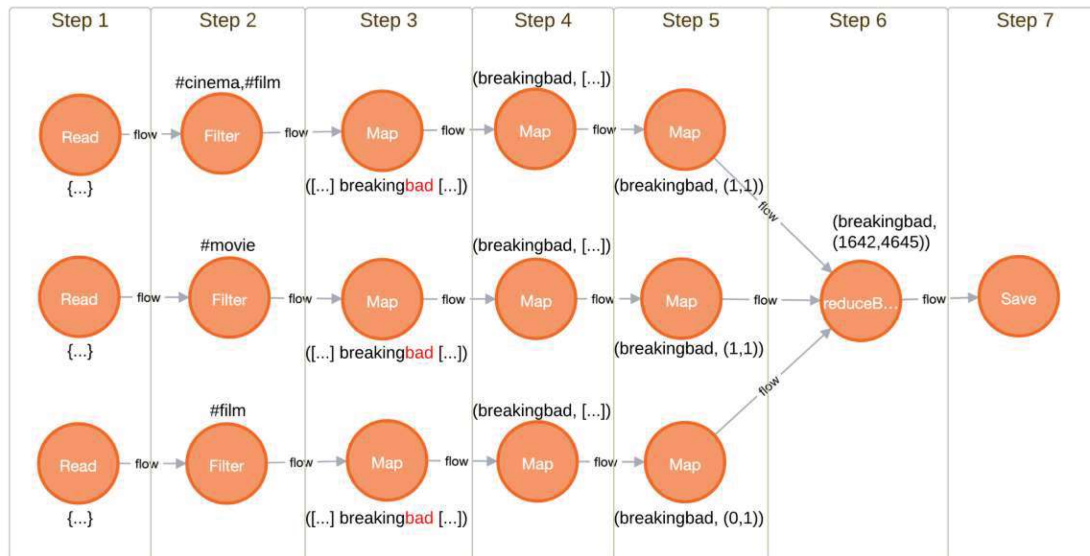


Figure 5.11: Fine-grained Data Lineage for Tweets

Figure 5.11 shows the fine-grained data lineage for three randomly chosen tweets. The graph was generated by backward tracing from the node with HashOut *write#2#1*. For illustrative purposes, we have also added a part of the recorded value to each node in the graph.

Analysing the fine-grained data lineage, we recognise that the reduceByKey operation in Step 6 counted 4,645 contributions classified as negative and only 1,642 classified as positive. Tracing back further to Step 5, it is noticeable that the text of each randomly chosen tweet was categorised as either negative or neutral (positive and negative). Step 4 extracts the movie name from the text attribute, extracted from the whole JSON in Step 3. Further examining the text, we realize it always contains the term "bad" due to the film title. Hence, in Step 5, at least one negative attribute is always found, explaining the many negatively classified tweets.

This gives us a perfect example of why the fine-grained data lineage can be beneficial during the debugging process. By visualising and examining a subset of the data, we could quickly identify an error in the processing pipeline.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

5.2 Comparison of the Lineage Systems

	Lineage Graph	Backward Tracing	Forward Tracing	Intermediate Results	Reusable	Transparent
Lineage Master (2024)	✓	✓	✓	✓	✓	✓
SAMbA-RaP (2020)	✓	✓	✓	✓		✓
Ursprung (2020)					✓	✓
Spline (2018)	✓				✓	✓
Big Debug (2016)		✓	✓	✓		
Titian (2015)		✓	✓	✓		
Newt (2013)		✓	✓	✓	✓	✓
HadoopProv (2013)		✓	✓			✓
Ramp (2011)		✓	✓			✓
Inspector Gadget (2011)		✓	✓	✓⋆	✓⋆	✓

⋆ ... with limitations

Table 5.6: System Characteristics with Lineage Master

With Lineage Master, we have developed a system that fulfils all characteristics described in Table 5.6. We achieved this by identifying and combining the most promising solutions of previous work into a new lineage tracking system, as discussed in Section 3.10.

Our architectural model, described in Section 4.1 separates the lineage collection from the lineage processing using a simple and well-defined API. It is oriented towards the Newt API developed by Logothetis et al. [LDY13] with reusability in mind. They already used it to instrument Hadoop MapReduce and Hyracks. With Lineage Master, we can add another system, which has been instrumented with the assistance of the API. Hence, the two-tier approach allows for more straightforward instrumentation of various DISC systems. This answers our first research question: "How can a two-tier approach, utilizing a framework-specific extension for emitting and an agnostic backend system for analysing the data lineage, promote the system's reusability across different Big Data technologies?".

Apart from the Newt [LDY13] API, we were also able to identify and employ other promising ideas in existing solutions as discussed in Subsection 3.10.4. As a result, we have succeeded in implementing a comprehensive, transparent and reusable system. Our adjustments to the framework are transparent, as no changes to the instrumented code are required apart from using the LineageContext. As previously argued, dividing the system into two parts with an API in between improves reusability. Furthermore, we demonstrated the listed functionalities (Lineage Graph, Forward Tracing, Backward

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Tracing and Intermediate Results) in Section 4.5. Hence, this answers our second research question: "How can a combination and extension of existing approaches be leveraged to fulfil all functions and characteristics described in Table 5.6?".

We also reviewed the different data collection algorithms for extracting a fine-grained data lineage in Section 3.10 and concluded that in our case, it is optimal to:

- Directly extend the DISC framework [IST+15, GMF+20].
- Use a tuple (x,y) per data element and operation, where x and y are hashes or prominent values the system will preserve, to construct a ledger with the lineage information [IST+15].
- Avoid shuffling of lineage identifiers [ASH13, IST+15].
- Avoid piggybacking of lineage identifiers [ASH13, IST+15].

This answers our third and last research question: "Which data collection algorithm is the most appropriate for this task?". A detailed explanation of the employed collection algorithm can be found in Section 4.2.

Another goal of our work was to keep the overhead for data lineage collection low and influence the execution time of a Spark job as little as possible. With Titian, Interlandi et al. [IST+15] already concluded that an external storage system is not optimal in terms of performance. In Titian-C, agents write the data lineage over an asynchronous message channel to a centralized server, storing it in its local file system. However, our ultimate goal was to design a reusable system by targeting a two-tier approach, separating the data lineage collection from its exploitation. Hence, we have decided to employ an external storage system anyway and evaluate whether the performance problems can be solved using scalable state-of-the-art technologies.

Our evaluation in Section 5.1 shows a similar result to Titian-C [IST+15]. The overhead remains manageable in the Grep example when working with small dataset sizes. The execution time of the Spark job, as well as the time required to build both the coarse- and fine-grained lineage graphs, significantly increases in the case of the WordCount example. Therefore, we conclude that it is not a viable approach to store fine-grained data lineage in an external storage system for each execution of a Spark job. Instead, it should be employed selectively to debug a program with a subset of the data. In contrast, the coarse-grained lineage graph can be captured and stored for every execution, as its acquisition involves only a low overhead.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 6

Conclusion and Future Work

This thesis explored the collection of coarse- and fine-grained data lineage in DISC systems, emphasizing its potential to unlock novel possibilities in data-intensive, scalable computing. When looking at the runtime and memory requirements of existing systems, however, it is evident that efficiently collecting such information is not trivial. Depending on the input data, transformation script, and granularity of the data lineage, its size can quickly surpass that of the original dataset. Several papers have already been published that deal with efficiently acquiring a fine-grained data lineage in DISC systems. However, they have significant limitations regarding their functional scope or system characteristics.

In this thesis, we have analysed existing solutions and combined the most promising approaches. This has enabled us to design a comprehensive yet transparent system that can be reused across different frameworks and scaled as needed. In this context, comprehensive means that it supports the essential capabilities of a data lineage system, namely backward tracing, forward tracing, determination of intermediate results and the construction of the coarse-grained lineage graph. Transparency means that no extensive modifications to the executed scripts are required. We further focused on reusability, employing a two-tier approach utilizing a framework-specific extension for emitting and an agnostic backend system for analysing the data lineage.

Our result again reveals that developing a reusable lineage tracking system without losing efficiency is not trivial. Previous work has shown that an efficient collection of fine-grained data lineage requires a direct extension of the DISC framework and local storage/analysis. A two-tier approach separating the lineage collection from its storage/analysis promotes the system's reusability but impacts the overall performance.

Implementing Lineage Master using scalable state-of-the-art technologies has given us valuable insights into this two-tier approach. Our performance evaluation shows that our solution has significant problems as data lineage increases. This can be observed in the WordCount example, where the fan-out during the flatMap operation leads to a

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

tenfold increase in data lineage. However, our real-world example, Tweets, has shown the usability of Lineage Master under certain circumstances. We have demonstrated that, in this example, it is possible to record the data lineage of large input files with moderate overhead. The employed data collection algorithm assigns each tracked element two hashes (HashIn, HashOut), where the HashOut of one element matches the HashIn of the subsequent element(s), to construct a ledger with the lineage information.

Overall, the volume of data generated by collecting a fine-grained data lineage is considerable. Titian-D performs well because it uses Spark's internal storage layer for buffering and Spark itself to analyse the data lineage. This requires implementing all lineage tracking capabilities within the specific framework, complicating the reusability. Nevertheless, we would prefer this approach due to its scalability and interactive exploration of the data lineage. Future work on acquiring a fine-grained data lineage should focus on such embedded solutions.

Concerning our implementation, Lineage Master, future work should address the limitations described in Section 4.6. Developing a more generic, pluggable solution for instrumenting the Spark execution is expected to cover more operations of the DISC system while being easier to maintain. This could be achieved, for example, by implementing the necessary extensions in a separate Java project and setting the generated JAR file on the classpath.

Finally, it would be interesting to investigate the effects of data lineage filters on the overall performance and storage overhead. Similar to the Provenance Class Filters described in Section 3.7, these could minimise the amount of collected data lineage.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Overview of Generative AI Tools Used

ChatGPT/Perplexity

- **Coding Assistance**: ChatGPT/Perplexity assisted in creating Cypher queries, Dockerfiles, and Kafka configurations. They were used as an assistive tool, and their output served as an orientation to achieve a specific academic goal I developed.
- **Scripting Assistance**: ChatGPT/Perplexity assisted in creating Python scripts that generated the sample data for our performance evaluation. The scripts and generated sample data have been shaped by my academic input and were thoroughly examined and significantly adjusted to ensure they reflect my original ideas.
- **Grammar and Style Improvement**: ChatGPT was used to correct grammatical errors and to improve sentence structure and general readability.

Grammarly Text Assistant

- **Grammar and Style Improvement**: The Grammarly Text Assistant was used to correct grammatical errors and to improve sentence structure and general readability.

Lucidchart AI

- **UML Diagrams**: Lucidchart AI assisted in creating UML diagrams. Lucidchart AI was only used as an assistive tool. The diagrams have been shaped by my academic input and were thoroughly examined and significantly adjusted to ensure they reflect my original ideas.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Figures

2.1 MapReduce Programming Model [DG08] 6
2.2 Spark Core Components [Luu21] 7
2.3 Spark Execution Model [DLWD20] 8
2.4 Catalyst Optimizer [Luu21] 9
2.5 Example of a Provenance Graph [CLMR16] 13
2.6 Simple MapReduce Workflow for Movie Ratings [IPW11] 14

3.1 Architecture of Inspector Gadget [OR11] 18
3.2 Implementation Details of Inspector Gadget [OR11] 20
3.3 Hadoop Extension in Ramp [IPW11] 22
3.4 Provenance Collection in HadoopProv [ASH13] 24
3.5 Newt Architecture [LDY13] 26
3.6 Trace Plan in Titian [IST+15] 31
3.7 Architecture of Ursprung [RDA+20] 35
3.8 Architecture of SAMbA-RaP [GMF+20] 37
3.9 Data Collection Mechanism in SAMbA-RaP [GMF+20] 38

4.1 Architectural Model 46
4.2 Architecture Lineage Processing System 49
4.3 Class Diagram Showing the LineageApi 51
4.4 LRDD Structure for MapPartitionsLRDD 53
4.5 Lineage Collection Narrow Transformations 54
4.6 Lineage Collection Wide Transformations 55
4.7 Lineage Graph of a WordCount Example in Bloom 60
4.8 Backward Tracing of a WordCount Example in Bloom 60
4.9 Forward Tracing of a WordCount Example in Bloom 61
4.10 Lineage of a Simple WordCount Example with Intermediate Results in Bloom 62

5.1 Runtime of Spark and Lineage Master for Grep 69
5.2 Storage Consumption of Lineage Master for Grep 70
5.3 Lineage Processing Time of Lineage Master for Grep 70
5.4 Runtime of Spark and Lineage Master for WordCount 72
5.5 Storage Consumption of Lineage Master for WordCount 73
5.6 Lineage Processing Time of Lineage Master for WordCount 73

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.7 Tweets Lineage Graph 74
5.8 Runtime of Spark, Lineage Master and Lineage Master with Details for Tweets 77
5.9 Storage Consumption of Lineage Master for Tweets 78
5.10 Lineage Processing Time of Lineage Master for Tweets 78
5.11 Fine-grained Data Lineage for Tweets 79

Listings

4.1 Overwritten Filter Method in Lineage Trait 52
4.2 RDD Compute Method 53
4.3 Overwritten Compute Method in MapPartitionsLRDD 53
4.4 Insert and Link a LineageNode 57
4.5 Cypher Query to Insert LineageFlow Entities 57
4.6 Cypher Query to Link LineageFlow Entities 58
4.7 Cypher Query to Insert and Link LineageFlow Entities 58
4.8 Cypher Query to add Name and Description to LineageFlow Nodes . . 59
4.9 Query Lineage Graph 60
4.10 Query Backward Tracing 60
4.11 Query Forward Tracing 61
5.1 Spark-Submit Command for WordCount with Lineage 65
5.2 Grep in Scala 68
5.3 WordCount in Scala 71
5.4 Tweets Example 75
5.5 Exemplary Tweet Related to the Movie Matrix 76

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

List of Tables

1.1 System Characteristics 2

2.1 Main Data Abstractions in Apache Spark: Adapted from [DLWD20] . . . 9

4.1 LineageApi 47
4.2 Identifier used in the LineageApi 51

5.1 Resource Limitations of the Docker Containers 66
5.2 Median Word Frequency in the Sample Datasets 69
5.3 Runtime of Lineage Master, Titian-D, RAMP and Newt for Grep: Adapted from [IST+15] 69
5.4 Runtime of Lineage Master, Titian-D, RAMP and Newt for WordCount: Adapted from [IST+15] 72
5.5 Runtime of Lineage Master and Lineage Master with Details for Tweets . 77
5.6 System Characteristics with Lineage Master 80

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acronyms

ACID Atomicity, Consistency, Isolation, and Durability. 12, 50

API Application Programming Interface. xi, 7–9, 18–20, 26–28, 33, 42, 43, 47, 48, 50, 57, 62, 80

APOC Awesome Procedures On Cypher. 58, 59, 67

CPU Central Processing Unit. 7, 66

DAG Directed Acyclic Graph. 5, 7, 29

DBMS Database Management System. 38–40

DISC Data Intensive Scalable Computing. ix, xi, 1, 3, 4, 14, 18, 20, 26, 41, 43, 44, 47, 48, 62, 80, 81, 83, 84

GMRW Generalized Map and Reduce Workflows. 21

HDFS Hadoop Distributed File System. 6, 8, 10, 29, 42, 43, 46, 52, 67, 71, 76

I/O Input/Output. 30, 36, 38

JAR Java Archive. 65, 84

JSON JavaScript Object Notation. 50, 51, 76, 79

JVM Java Virtual Machine. 7, 33

LRDD Lineage Resilient Distributed Dataset. 29, 45, 52, 53, 60, 68, 87

NRC Nested Relational Calculus. 40

RAM Random-Access Memory. 66

RDBMS Relational Database Management System. 42, 43

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

RDD Resilient Distributed Dataset. 3, 8, 9, 29, 30, 37–45, 52–55, 62, 68

RPC Remote Procedure Call. 5

S3 Simple Storage Service. 6

SQL Structured Query Language. 9, 26, 38–40

UDF User Defined Function. 19, 41

UML Unified Modeling Language. 48, 50, 52, 85

UUID Universally Unique Identifier. 55

WAL Write Ahead Log. 11

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Bibliography

[Abs] AbsaOSS. Spline - data lineage tracking and visualization solution. https://absaoss.github.io/spline/0.7.html. Accessed: May 30, 2024.

[ADD+11] Yael Amsterdamer, Susan B. Davidson, Daniel Deutch, Tova Milo, Julia Stoyanovich, and Val Tannen. Putting lipstick on Pig: Enabling database-style workflow provenance. *Proc. VLDB Endow.*, 5(4):346–357, 2011.

[ADT11] Yael Amsterdamer, Daniel Deutch, and Val Tannen. Provenance for aggregate queries. In Maurizio Lenzerini and Thomas Schwentick, editors, *Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2011, June 12-16, 2011, Athens, Greece*, pages 153–164. ACM, 2011.

[Apa] Apache Software Foundation. Apache Kafka documentation. https://kafka.apache.org/34/documentation.html. Accessed: October 18, 2024.

[Apa15] Apache Software Foundation. Apache Spark 1.3.0 release notes. https://spark.apache.org/releases/spark-release-1-3-0.html, March 2015. Accessed: March 30, 2024.

[Apa16a] Apache Software Foundation. Apache Spark 1.6.0 release notes. https://spark.apache.org/releases/spark-release-1-6-0.html, January 2016. Accessed: March 30, 2024.

[Apa16b] Apache Software Foundation. Apache Spark 2.0.0 release notes. https://spark.apache.org/releases/spark-release-2-0-0.html, July 2016. Accessed: March 30, 2024.

[Apa19] Apache Software Foundation. HDFS architecture. https://hadoop.apache.org/docs/r3.2.1/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html, September 2019. Accessed: October 19, 2024.

[Apa24] Apache Software Foundation. MapReduce tutorial. https://hadoop.apache.org/docs/r3.4.0/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html, March 2024. Accessed: October 16, 2024.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[ASH13] Sherif Akoush, Ripduman Sohan, and Andy Hopper. HadoopProv: Towards provenance as a first class citizen in MapReduce. In Alexandra Meliou and Val Tannen, editors, *5th Workshop on the Theory and Practice of Provenance, TaPP'13, Lombard, IL, USA, April 2-3, 2013*. USENIX Association, 2013.

[Bas13] Basel Committee on Banking Supervision. Principles for effective risk data aggregation and risk reporting. `https://www.bis.org/publ/bcbs239.pdf`, January 2013. Accessed: November 1, 2024.

[BKT01] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. Why and where: A characterization of data provenance. In Jan Van den Bussche and Victor Vianu, editors, *Database Theory - ICDT 2001, 8th International Conference, London, UK, January 4-6, 2001, Proceedings*, volume 1973 of *Lecture Notes in Computer Science*, pages 316–330. Springer, 2001.

[CLMR16] Zaheer Chothia, John Liagouris, Frank McSherry, and Timothy Roscoe. Explaining outputs in modern data analytics. *Proc. VLDB Endow.*, 9(12):1137–1148, 2016.

[CW01] Yingwei Cui and Jennifer Widom. Lineage tracing for general data warehouse transformations. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 471–480. Morgan Kaufmann, 2001.

[CWW00] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25(2):179–227, 2000.

[Dat23] Datanyze. Market share of leading big data processing technologies worldwide in 2023. `https://www.statista.com/statistics/1258671/big-data-processing-software-market-share-technology-worldwide/`, November 2023. Accessed: March 28, 2024.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In Eric A. Brewer and Peter Chen, editors, *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150. USENIX Association, 2004.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[DLWD20] Jules S. Damji, Denny Lee, Brooke Wenig, and Tathagata Das. *Learning Spark: Lightning-Fast Data Analytics*. O'Reilly Media, Sebastopol, California, 2nd edition, 2020.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[GIY+16] Muhammad Ali Gulzar, Matteo Interlandi, Seunghyun Yoo, Sai Deep Tetali, Tyson Condie, Todd D. Millstein, and Miryung Kim. BigDebug: Debugging primitives for interactive big data processing in Spark. In Laura K. Dillon, Willem Visser, and Laurie A. Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 784–795. ACM, 2016.

[GKT07] Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance semirings. In Leonid Libkin, editor, *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China*, pages 31–40. ACM, 2007.

[GMF+20] Thaylon Guedes, Lucas Bertelli Martins, Maria Luiza Furtuozo Falci, Vítor Silva, Kary A. C. S. Ocaña, Marta Mattoso, Marcos V. N. Bedo, and Daniel de Oliveira. Capturing and analyzing provenance from Spark-based scientific workflows with SAMbA-RaP. *Future Gener. Comput. Syst.*, 112:658–669, 2020.

[HDB17] Melanie Herschel, Ralf Diestelkämper, and Houssem Ben Lahmar. A survey on provenance: What for? What form? What from? *VLDB J.*, 26(6):881–906, 2017.

[IPW11] Robert Ikeda, Hyunjung Park, and Jennifer Widom. Provenance for generalized map and reduce workflows. In *Fifth Biennial Conference on Innovative Data Systems Research, CIDR 2011, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 273–283. CIDR, 2011.

[IST+15] Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd D. Millstein, and Tyson Condie. Titian: Data provenance support in Spark. *Proc. VLDB Endow.*, 9(3):216–227, 2015.

[KS17] Manish Kumar and Chanchal Singh. *Building Data Streaming Applications with Apache Kafka: Design and Administer Fast, Reliable Enterprise Messaging Systems with Apache Kafka.* Packt Publishing, Limited, Birmingham, 1st edition, 2017.

[LDY13] Dionysios Logothetis, Soumyarupa De, and Kenneth Yocum. Scalable lineage capture for debugging DISC analytics. In Guy M. Lohman, editor, *ACM Symposium on Cloud Computing, SOCC '13, Santa Clara, CA, USA, October 1-3, 2013*, pages 17:1–17:15. ACM, 2013.

[Luu21] Hien Luu. *Beginning Apache Spark 3: With DataFrame, Spark SQL, Structured Streaming, and Spark Machine Learning Library.* Apress L. P, Berkeley, CA, 2nd edition, 2021.

[Neo] Neo4j. Introduction to clustering in Neo4j. `https://neo4j.com/docs/operations-manual/5/clustering/introduction/`. Accessed: August 31, 2024.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[OR11] Christopher Olston and Benjamin C. Reed. Inspector gadget: A framework for custom monitoring and debugging of distributed dataflows. *Proc. VLDB Endow.*, 4(12):1237–1248, 2011.

[ORS+08] Christopher Olston, Benjamin C. Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In Jason Tsong-Li Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 1099–1110. ACM, 2008.

[Raj15] Sonal Raj. *Neo4j high performance: Design, build, and administer scalable graph database systems for your applications using Neo4j*. Community Experience Distilled. Packt Publishing, Birmingham, England ; Mumbai, [India], 1st edition, 2015.

[RDA+20] Lukas Rupprecht, James C. Davis, Constantine Arnold, Yaniv Gur, and Deepavali Bhagwat. Improving reproducibility of data science pipelines through transparent provenance capture. *Proc. VLDB Endow.*, 13(12):3354–3368, 2020.

[REW13] Ian Robinson, Emil Efrem, and James Webber. *Graph databases*. O'Reilly Media, Sebastopol, California, 4th edition, 2013.

[SI21] Statista and IDC. Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2020, with forecasts from 2021 to 2025. `https://www.statista.com/statistics/871513/worldwide-data-created/`, June 2021. Accessed: March 28, 2024.

[SNV18] Jan Scherbaum, Marek Novotny, and Oleksandr Vayda. Spline: Spark lineage, not only for the banking industry. In *2018 IEEE International Conference on Big Data and Smart Computing, BigComp 2018, Shanghai, China, January 15-17, 2018*, pages 495–498. IEEE Computer Society, 2018.

[Spra] Spring Framework Team. Kafkalistener annotation. `https://docs.spring.io/spring-kafka/reference/3.2/kafka/receiving-messages/listener-annotation.html`. Accessed: October 18, 2024.

[Sprb] Spring Framework Team. Manually committing offsets. `https://docs.spring.io/spring-kafka/reference/3.2/kafka/receiving-messages/ooo-commits.html`. Accessed: October 18, 2024.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In Erich M. Nahum and Dongyan Xu, editors, *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*. USENIX Association, 2010.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub