Universität Stuttgart
Institut für Industrielle Fertigung
und Fabrikbetrieb

Fraunhofer
IPA

Bachelor Thesis

My Bachelor Thesis

vorgelegt von

Marcel Pascal Stolin

32168

geboren am 03.04.1993 in Kamen

Verfasst am

Fraunhofer-Institut fÃ¼r Produktionstechnik und Automatisierung IPA

und

Hochschule der Medien Stuttgart

ErstprÃ¼fer:	Prof. Walter Kriha
ZweitprÃ¼fer:	Univ.-Prof. Dr.-Ing. habil. Marco Huber
Betreuung:	M.Sc. Peter Trom

Stuttgart, 25. Oktober 2020

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich, Marcel Pascal Stolin, die vorliegende Bachelor Thesis selbstständig angefertigt, keine anderen als die angegebenen Hilfsmittel benutzt und sowohl wörtliche als auch sinngemäß entlehnte Stellen als solche kenntlich gemacht habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Ort, Datum: ____________________________

Unterschrift: ____________________________
Marcel Pascal Stolin

Zusammenfassung

Hier kommt eine deutschsprachige Zusammenfassung hin.

Abstract

Abstract in English.

Inhaltsverzeichnis

Notation

Konventionen

x	Skalar
$\underline{x}$	Spaltenvektor
$\boldsymbol{x}, \underline{\boldsymbol{x}}$	Zufallsvariable/-vektor
$\hat{x}, \hat{\underline{x}}$	Mittelwert/-vektor
$x^*, \underline{x}^*$	Optimaler Wert/Vektor
$x_{0:k}, \underline{x}_{0:k}$	Folge von Werten $(x_0, x_1, \ldots, x_k)$ / Vektoren $(\underline{x}_0, \underline{x}_1, \ldots, \underline{x}_k)$
$\mathbf{A}$	Matrizen
$\mathcal{A}$	Mengen
$\preceq, \prec$	schwache/strenge Präferenzrelation
$\mathbb{R}$	Reelle Zahlen
$\mathbb{N}$	Natürliche Zahlen
$\blacksquare$	Ende eines Beispiels
$\square$	Ende eines Beweises

Operatoren

$\mathbf{A}^{\mathrm{T}}$	Matrixtransposition
$\mathbf{A}^{-1}$	Matrixinversion
$\lvert\mathbf{A}\rvert$	Determinante einer Matrix
$\lvert\mathcal{A}\rvert$	Kardinalität der Menge $\mathcal{A}$
$\mathrm{pot}(\mathcal{A})$	Potenzmenge von $\mathcal{A}$
$\mathrm{E}\{\cdot\}$	Erwartungswertoperator
$\mathcal{O}(g)$	O-Kalkül entsprechend der Landau-Notation bei welcher beispielsweise $f(x) \in \mathcal{O}(g(x))$ besagt, dass die Funktion $f(x)$ die Komplexität $\mathcal{O}(g(x))$ besitzt

Spezielle Funktionen

$\Pr(\mathcal{E})$	Wahrscheinlichkeitsmaß, welches die Wahrscheinlichkeit angibt, dass Ereignis $\mathcal{E}$ eintritt
$p(\underline{x})$	(Wahrscheinlichkeits-)Dichtefunktion für kontinuierliche $\underline{x}$

	und Zähldichte für diskrete $\underline{x}$
$p(\underline{x}\|\underline{y})$	Bedingte Dichtefunktion
$P(\underline{x})$	(Wahrscheinlichkeits-)Verteilungsfunktion
$\mathrm{erf}(x)$	Gauß'sche Fehlerfunktion
$\exp(x)$	Exponentialfunktion e^x
$\mathcal{N}(\underline{x}; \hat{\underline{x}}, \mathbf{C}_x)$	Gaußdichte, d. h. Dichtefunktion eines normalverteilten Zufallsvektors $\underline{\boldsymbol{x}}$ mit Mittelwertvektor $\hat{\underline{x}}$ und Kovarianzmatrix $\mathbf{C}_x$

Kapitel 1

Introduction

Storing huge amounts of data has become inexpensive in recent years, but processing it, requires parallel computations on clusters with multiple machines [CZ18]. Complex computation operations rely on a IT infrastructure with the ability to perform operations on scale.

In order to achieve high scalability, computing systems need to adapt dynamically to demands and conditions of the workload.

1.1 Problem statement

ETL [1] operations are compute-intensive. During the execution of analytic applications, performance thresholds can be reached and the computing system can become out-of-order.

In addition, many algorithms profit from data-parallelism.

1.2 Research questions

1.3 Thesis structure

1 Extract transform load

Kapitel 2

Related Work

Kapitel 3

Background

In this chapter bla bla

TODO: Describe Chapter

3.1 Autonomic computing

Autonomic computing is an architecture for computing systems to enable the ability to manage themselves in accordance to high level objectives configured by administrators [KC03]. These computing systems dynamically adapt to demands and conditions of the workload [KC03]. An intelligent control-loop is responsible to collect all important details of the computing system and make decision according to the collected details. To automate the tasks, the intelligent control-loop is organized into four categories:

- **Self-configuring** - Components in the environment have to adapt dynamically to system changes using policies. For example, deploying or removing new components.
- **Self-healing** - If system errors have being detected, the control-loop has to perform policy-based actions without disrupting the environment.
- **Self-optimizing** - The control-loop has to monitor the resources and should adapt to changes dynamically.
- **Self-protecting** - Detection and protection against threats.

An autonomic computing environment consists of an autonomic manager, managed-resources and a knowledge-base.

3.1.1 Managed resources

Managed resources are software or hardware components in the computing environment. For example, a managed resource can be a database, service, application, server or different entity. Each managed resource implements an interface to enable the autonomic manager to communicate with the managed resource.

These interface are called touchpoints.

3.1.2 Autonomic manager

The autonomic manager implements an intelligent control-loop to collect system metrics from the managed resources and acts according to the collected details. It can only make adjustments within it‘s own scope and uses policies to make decisions of what actions have to be executed to accommodate the objectives. To be self-managing, the autonomic manager has to implement the following four automated functions.

- **Monitor** - The monitor function is responsible to collect the needed metrics from all managed resources and applies aggregation and filter operations to the collected data. After that the function reports the metrics.
- **Analyze** - To determine if changes have to be made to the computing system, the collected data has to be analyzed.
- **Plan** - If changes have to be made, an appropriate change plan has to be generated. A change plan consists of actions that are needed to achieve the configured goals and objectives. The change plan needs to be forwarded to the execute function.
- **Execute** - The execute function applies all necessary changes to the computing system.

Multiple autonomic manager can exist in an autonomic computing environment to perform only certain parts. For example, there can be one autonomic manager which is responsible to monitor and analyze the system and another autonomic manager to plan and execute. To create a complete and closed control-loop, multiple autonomic manager can be composed together.

3.2 Docker

3.3 Apache Spark

Apache Spark is an open-source computing framework for parallel data processing on a large computer cluster. Spark manages the available resources and distributes computation tasks across a cluster to perform big-data processing operations at large scale [CZ18]. Before Spark was developed, Hadoop MapReduce [DG10] was the framework of choice for parallel operations on a computer cluster [ZCF$^+$10]. Spark accomplished to outperform Hadoop by

10x for iterative Machine Learning [ZCF+10]. It is implemented in Scala[1], a JVM-based language and provides a programming interface for Scala, Java[2], Python[3] and R[4]. In addition, Spark includes an interactive SQL shell and libraries to implement Machine Learning and streaming applications [CZ18]. It was developed in 2009 as the Spark research project at UC Berkeley and became an Apache Software Foundation project in 2013 [CZ18].

3.3.1 Spark programming model

Spark provides resilient distributed datasets (RDDs) as the main abstraction for parallel operations [ZCF+10]. Core types of Spark‘s higher-level structured API are built on top of RDDs [CZ18] and will automatically be optimized by Spark‘s Catalyst optimizer to run operations quick and efficient [Luu18].

TODO: Master-Slave Architektur + Bild

Resilient distributed datasets: Resilient distributed datasets are fault-tolerant, parallel data structures to enable data sharing across cluster applications [ZCD+12]. They allow to express different cluster programming models like MapReduce, SQL and batched stream processing [ZCD+12]. RDDs have been implemented in Spark and serve as the underlying data structure for higher level APIs (Spark structured API) [ZCD+12]. RDD‘s are a immutable, partitioned collection of records and can only be initiated through transformations (e.g. map, filter) on data or other RDD‘s. An advantage of RDDs is, that they can be recovered through lineage. Lost partitions of an RDD can be recomputed from other RDDs in parallel on different nodes [ZCD+12]. RDDs are lower level APIs and should only be used in applications if custom data partitioning is needed [CZ18]. It is recommended to use Sparks structured API objects instead. Optimizations for RDDs have to be implemented manually while Spark automatically optimize the execution for structured API operations [CZ18].

Spark structured API: Spark provides high level structured APIs for manipulating all kinds of data. The three distributed core types are Datasets, DataFrames and SQL Tables and Views [CZ18]. Datasets and DataFrames are immutable, lazy evaluated collections that provide execution plans for operations [CZ18]. SQL Tables and Views work the same way as DataFrames, except that SQL is used as the interface instead of using the DataFrame programming interface [CZ18]. Datasets use JVM types and are therefore only available for JVM based languages. DataFrames are Datasets of type Row, which is the Spark internal optimized format for computations. This has advantages over JVM types which comes with garbage collection and object instantiation [CZ18].

1 Scala programming language. https://www.scala-lang.org/
2 Java programming language. https://www.oracle.com/java/
3 Python programming language. https://www.python.org/
4 R programming language. https://www.r-project.org/

Spark Catalyst: Spark also provides a query optimizer engine called Spark Catalyst. Abbildung 3.1 illustrates how the Spark Catalyst optimizer automatically optimizes Spark applications to run quickly and efficient. Before executing the user‘s code, the Catalyst optimizer translates the data-processing logic into a logical plan and optimizes the plan using heuristics [Luu18]. After that, the Catalyst optimizer converts the logical plan into a physical plan to create code that can be executed [Luu18].

Logical plans get created from a DataFrame or a SQL query. A logical plan represents the data-processing logic as a tree of operators and expressions where the Catalyst optimizer can apply sets of rule-based and cost-based optimizations [Luu18]. For example, the Catalyst can position a filter transformation in front of a join operation [Luu18].

From the logical plan, the Catalyst optimizer creates one ore more physical plans which consist of RDD operations [CZ18]. The cheapest physical will be generated into Java bytecode for execution across the cluster [Luu18].

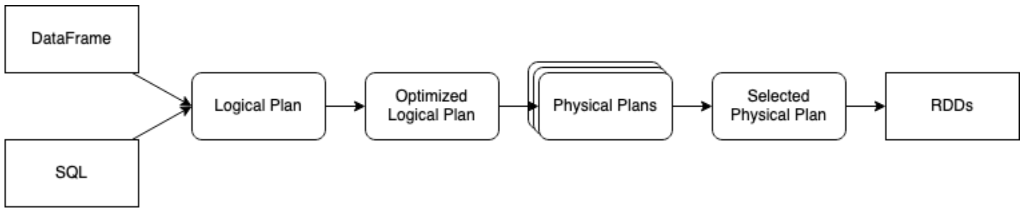

Abbildung 3.1: Optimization process of the Spark Catalyst - Source: Authors own model, based on [Luu18].

TODO: Bild nochmal machen mit abstand + QUELLE anpassen

3.3.2 Spark application architecture

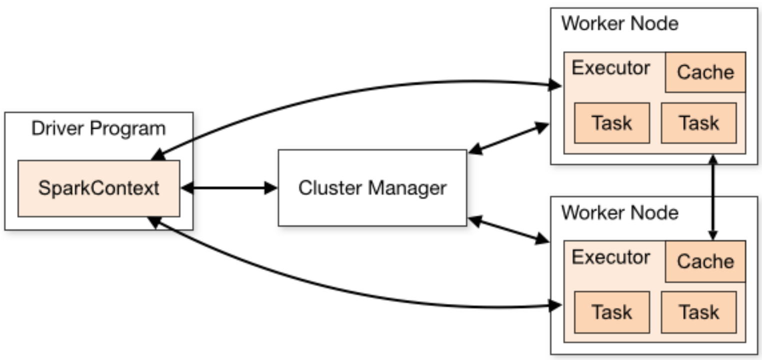

Abbildung 3.2: Overview of a Spark cluster architecture - Source: Authors own model, based on [Apa20].

TODO: Lieber Doch Master/Slave ???. Was ist mit dem CLuster-Manager

TODO: Nicht ganz richtig, Master und Worker sind machines und driver und executor sind prozesse

Abbildung 3.2 illustrates the main architecture of a Spark cluster. The architecture follows the master-worker model where the Spark driver is the master and the Spark executors are the worker [Luu18].

Spark driver: The Spark driver is a JVM process on a physical machine and responsible to maintain the execution of a Spark application [CZ18]. It coordinates the application tasks onto each available executor [Luu18]. To get launch executors and get physical resources, the Spark driver interacts with the cluster manager [CZ18, Luu18].

Spark Executor: A Spark executor performs the tasks given by the Spark driver [CZ18]. It runs as a JVM process and runs until the Spark application finishes [Luu18]. After the executor finishes, it reports back to the Spark driver [CZ18]. Each task will be performed on a separate CPU core to enable parallel processing [Luu18].

Cluster manager: The cluster manager is an external service that orchestrates the work between the machines in the cluster [Luu18, Apa20]. The cluster manager knows about the resources of each worker and decides on which machine the Spark driver process and the executor processes run [Luu18, CZ18]. Spark supports different services that can run as the cluster manager: Standalone, Apache Mesos[5], Hadoop YARN[VMD$^+$13] and Kubernetes[6] [Apa20]. The cluster manager provides three different deploy modes for acquiring resources in the cluster.

TODO: Explain standalone

- Cluster mode
- Client mode
- Local mode

To run an application in cluster mode, the user has to submit a precompiled JAR, python script or R script to the cluster manager [CZ18]. After that, the cluster manager starts the driver process and executor processes exclusively for the Spark application on machines inside the cluster [CZ18, Luu18].

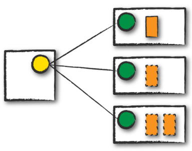

Abbildung 3.3: Spark's cluster mode - Source: Authors own model, based on [CZ18].

The difference between the client mode and the cluster mode is that, the driver process runs on the client machine outside of the Spark cluster [CZ18].

The local mode starts a Spark application on a single computer [CZ18]. It is not recommended to use the local mode in production, instead it should be used for testing Spark applications or learning the Spark framework [CZ18].

5 Apache Mesos. https://mesos.apache.org/
6 Kubernetes. https://kubernetes.io/

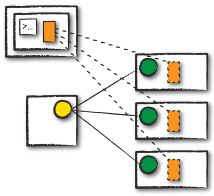

Abbildung 3.4: Spark‘s client mode - Source: Authors own model, based on [CZ18].

3.3.3 Spark application implementation

The concept of a Spark application consists of calling transformations and actions. A transformation creates a DataFrame or a Dataset, the logical data structures of a Spark application. The computation of a Spark application gets processed when an action gets called in the application. The transformations of a Spark application build up a directed acyclic graph (DAG) of instructions. By calling an action, the DAG will break down into stages and tasks to create a single job for execution [CZ18].

```
# Initialize a SparkSession
sparkSession = SparkSession\
    .builder\
    .getOrCreate()

# Create a dataframe with a transformation
dataframe = sparkSession.range(1, 1000)
# Apply another transformation
dataframe = dataframe.filter(dataframe.id % 2 == 0)
# Call an action
count = dataframe.count()
```

Listing 3.1: Example of a Python3 Spark application

3.1 demonstrates an example implementation of a Spark application. At first a SparkSession gets initialized. Each Spark application must include a SparkSession to initialize the application driver and executors [CZ18]. In Addition, the SparkSession provides an API for data-processing logic and configuration of the Spark application [Luu18]. After that, a DataFrame gets created with the range transformation to include each number from 1 to 1000 in the DataFrame. Next, a filter transformation is applied on the DataFrame to sort out any odd number. At the end, the number of rows gets saved in a variable with the count action.

TODO: Create listing macro

TODO: Tabelle von Luu18 vll als Anhang(Table 2-1 Subdirectories Inside the spark-2.1.1-bin-hadoop2.7 Directory)

The Spark framework provides a spark-submit executable to launch a Spark application inside a cluster.

```
$SPARK_HOME/bin/spark-submit \
    --master spark://spark-master:7077 \
    application.py
```

Listing 3.2: Execution of a Spark Python application using the spark-submit executable

3.2 provides an example how the spark-submit executable can be used to launch a Spark Python application.

3.3.4 Spark standalone cluster deployment

The standalone mode is a basic cluster-manager build specifically for Spark. It is developed to only run Spark but supports workloads at large scale [CZ18].

TODO: Why only standalone

Spark provides build-in scripts to start a master node and worker nodes in standalone mode. ABC demonstrates how a master node and worker node gets launched using the build-in scripts.

3.4 Prometheus

Prometheus is an open-source monitoring and alerting system that stores all data as time-series in a time-series database (TSDB) [Pro20]. It follows the pull-based approach to scrape metrics from hosts and services [BP19]. A monitoring system keeps track of the health status of components in a computing cluster. Because of the complexity of modern computing clusters, the effort is to high to observe components manually [BP19].

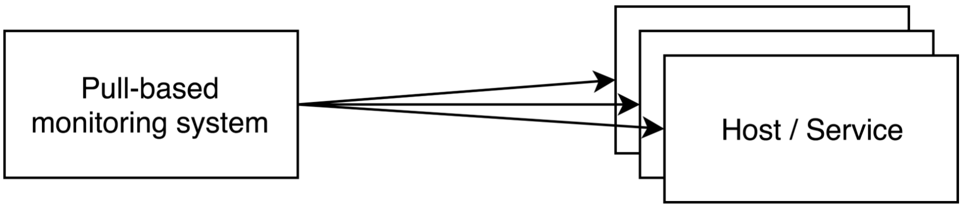

Abbildung 3.5: Pull-based approach to scrape metrics - Source: Authors own model, based on [BP19].

As Abbildung 3.5 demonstrates, a pull-based monitoring system scrapes metrics from services which make metrics available for the monitoring system. In this case, the monitoring system needs a list of all hosts and services to monitor [BP19].

3.4.1 Prometheus architecture

Abbildung 3.6 illustrates the high-level architecture of Prometheus. The Prometheus ecosystem provides multiple components. Components can be optional, depending on the monitoring needs of the environment [BP19]. The main components of a Prometheus system consist of Prometheus server, Alertmanager, Pushgateway, service discovery, exporters and visualization [Pro20].

Prometheus server: The Prometheus server is the main components of a Prometheus system. It is responsible to collect metrics as time-series data and store the collected data in the built-in TSDB. Additionally, the Prometheus

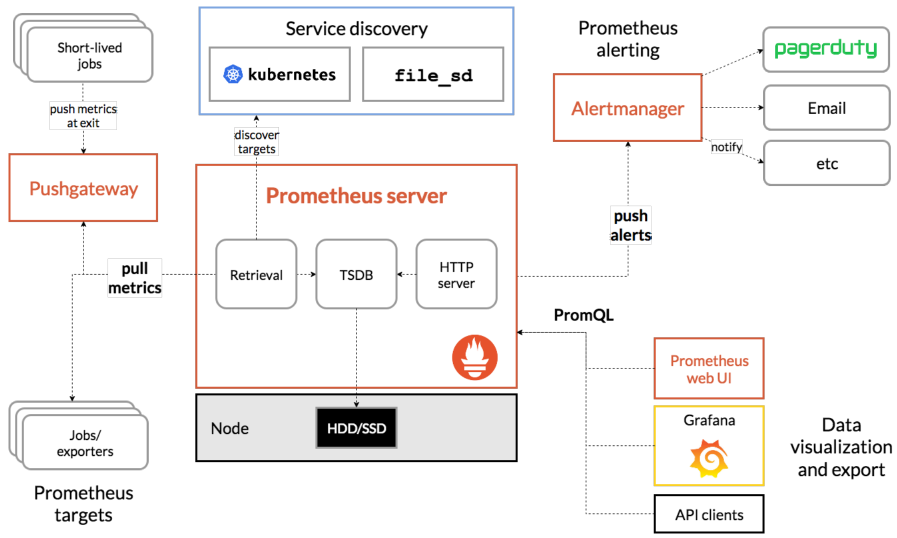

Abbildung 3.6: Prometheus high-level architecture - Source: Authors own model, based on [Pro20, BP19].

server triggers alerts to the Alertmanager if a configured condition becomes true [Pro20, BP19].

Alertmanager

3.4.2 Monitoring Docker container

3.5 NVIDIA RAPIDS

3.6 Gitlab CI/CD

3.7 K-MEANS

3.8 Naive Bayes Classifier

3.9 Scaling heat

3.10 KHP

Literaturverzeichnis

[Apa20] *Spark 3.0.1 Documentation.* https://spark.apache.org/docs/3.0.1/. Version: Oktober 2020

[BP19] Bastos, Joel ; Pedro, Araujo: *Hands-On Infrastructure Monitoring with Prometheus.* Packt Publishing, 2019. – ISBN 9781789612349

[CZ18] Chambers, Bill ; Zaharia, Matei: *Spark: The Definitive Guide.* 1st. O'Reilly Media, 2018. – ISBN 9781491912218

[DG10] Dean, Jeffrey ; Ghemawat, Sanjay: MapReduce: A Flexible Data Processing Tool. In: *Commun. ACM* 53 (2010), Januar, Nr. 1, 72â77. http://dx.doi.org/10.1145/1629175.1629198. – DOI 10.1145/1629175.1629198. – ISSN 0001–0782

[KC03] Kephart, J. O. ; Chess, D. M.: The vision of autonomic computing. In: *Computer* 36 (2003), Nr. 1, S. 41–50

[Luu18] Luu, Hien: *Beginning Apache Spark 2: With Resilient Distributed Datasets, Spark SQL, Structured Streaming and Spark Machine Learning library.* 1st. Apress, 2018. – ISBN 9781484235799

[Pro20] *Prometheus Online Documentation.* https://prometheus.io/docs/. Version: Oktober 2020

[VMD+13] Vavilapalli, Vinod K. ; Murthy, Arun C. ; Douglas, Chris ; Agarwal, Sharad ; Konar, Mahadev ; Evans, Robert ; Graves, Thomas ; Lowe, Jason ; Shah, Hitesh ; Seth, Siddharth ; Saha, Bikas ; Curino, Carlo ; O'Malley, Owen ; Radia, Sanjay ; Reed, Benjamin ; Baldeschwieler, Eric: Apache Hadoop YARN: Yet Another Resource Negotiator. New York, NY, USA : Association for Computing Machinery, 2013 (SOCC '13). – ISBN 9781450324281

[ZCD+12] Zaharia, Matei ; Chowdhury, Mosharaf ; Das, Tathagata ; Dave, Ankur ; Ma, Justin ; McCauly, Murphy ; Franklin, Michael J. ; Shenker, Scott ; Stoica, Ion: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster

Computing. In: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA : USENIX Association, April 2012. – ISBN 978–931971–92–8, 15–28

[ZCF+10] Zaharia, Matei ; Chowdhury, Mosharaf ; Franklin, Michael J. ; Shenker, Scott ; Stoica, Ion: Spark: Cluster Computing with Working Sets. In: *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*. USA : USENIX Association, 2010 (HotCloud'10), S. 10

Abbildungsverzeichnis