Institut für Parallele und Verteilte Systeme
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3033

Konzeption und Realisierung eines Auslastungsdienstes für eine verteilte Ausführungsumgebung

Raimund Huber

Studiengang:	Informatik
Prüfer:	Dr. Holger Schwarz
Betreuer:	Dipl.-Inf. Nazario Cipriani
begonnen am:	3. Mai 2010
beendet am:	2. November 2010
CR-Klassifikation:	C.4, D.4.1, H.3.3

Kurzfassung

In dieser Arbeit wird ein Auslastungsdienst für die verteilte Streaming-Middleware NexusDS entwickelt. Der Auslastungsdienst trifft Vorhersagen über geeignete Ausführungsknoten für die Ausführung von Operatoren. Für die Vorhersagen werden laufend auf den Ausführungsknoten erfasste Leistungsmessdaten über Operatoren und Ausführungsknoten verwendet.

Es wird ein Konzept zur effizienten Filterung der in Frage kommenden Ausführungsknoten entwickelt. Für die Vorhersage wird eine Clusterbildung verwendet, die Knoten mit ähnlichen Eigenschaften gruppiert. Um das Clustering schneller berechnen zu können, wird eine Map/Reduce-basierte, verteilte Implementierung entwickelt.

Abstract

In this thesis a utilization service will be developed for the distributed streaming middleware NexusDS. The utilization service predicts, which of the execution nodes known to him are suitable for the execution of an operator. The prediction is calculated by exploiting performance meters of the operators and execution nodes.

A concept for an efficient filtering of suitable execution nodes will be developed. For the forecasting a clustering algorithm is applied to group nodes with similar attributes. For the implementation of the clustering-algorithm a Map/Reduce based approach is applied to allow a faster calculation in a distributed manner.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Liste der Abkürzungen

AD	Auslastungsdienst
AFD	Anfragedienst
AWM	Augmented World Model
AWML	Augmented World Model Language
AWQL	Augmented World Query Language
CPU	Central Processing Unit
CQS	Core Query Service
GPU	Graphics Processing Unit
ID	Identifier
JVM	Java Virtual Machine
MiB	Mebibyte: Entsprechend Standard IEEE 1541 2002 [IEE09]: 1 MiB = 2^{20} Byte
NPGM	Nexus Plan Graph Model
OES	Operator Execution Service
QF	Query Fragmenter
XSD	XML Schema Definition

Kapitel 1

Einleitung

Trotz allgemein steigender Leistungsfähigkeit von mobilen Computern und Arbeitsplatzrechnern ist die Rechenleistung und Speicherkapazität eines einzelnen Gerätes häufig nicht ausreichend, um große Datenmengen vorzuhalten und schnell genug zu verarbeiten. Das NexusDS Framework erlaubt es, datenstrombasierte Berechnungen in eine verteilte Plattform zu verlagern. Die gebündelte Leistungsfähigkeit der in der Plattform verfügbaren Rechner wird so einem einzelnen Gerät verfügbar gemacht und ermöglicht es, Anwendungen, die aufgrund beschränkter Ressourcen sonst nicht lauffähig wären, verfügbar zu machen.

In NexusDS können Berechnungen für Datenströme auf sogenannte Operatoren innerhalb der Plattform verteilt werden. Bisher werden mit Hilfe eines graphischen Editors oder über XML-Steuerdateien die Datenquellen mit Operatoren verknüpft, die Operatoren miteinander verknüpft und letztlich in die Datensenke – die Ausgabe – geleitet. Allerdings muss bisher die Verteilung der Operatoren auf die einzelnen Knoten manuell unter Zuhilfenahme von externem Wissen des Benutzers über die Eigenschaften und die Leistung der Knoten vorgenommen werden.

Für eine automatische Platzierung der Operatoren ist ein Anfragedienst in NexusDS vorgesehen. Bei der Planung einer Vielzahl voneinander abhängigen Operatoren steigt die kombinatorische Komplexität. Um dem Anfragedienst um einen Teil der Komplexität zu entlasten, ist in NexusDS ein Auslastungsdienst vorgesehen, der die verfügbaren Ausführungsknoten verwaltet und beobachtet, um ihre Eigenschaften zu bestimmen und ihre Verfügbarkeit vorherzusagen.

Diese Eigenschaften können zum Beispiel Rechenleistung, Auslastung, spezielle Hardware wie Grafikkarten, spezielle Systemumgebung, Speicherplatz und Netzwerkanbindung sein. Zum einen werden aktuelle, also dynamische Daten für eine Bewertung gebraucht, um festzustellen, ob der Knoten momentan überhaupt verfügbar, gerade ausgelastet

oder nicht erreichbar ist. Zum anderen werden statische Daten über die Ausstattung, die maximale, die typische und die zu erwartende Leistungsfähigkeit der Knoten benötigt.

1.1 Aufgabenstellung

Das Ziel dieser Arbeit ist die Konzeption und Implementierung eines Auslastungsdienstes als Erweiterung des NexusDS Frameworks. Der Auslastungsdienst soll die verfügbaren Knoten verwalten und auf Anfrage nach einem Ausführungsknoten für einen bestimmten Operator Vorschläge liefern, welche Knoten sowohl den Anforderungen an die Hard- und Software-Umgebung entsprechen, als auch eine Vorhersage treffen, welche Knoten nach statistischen Erfahrungswerten am besten geeignet sind, den Operator mit seiner Konfiguration innerhalb vorgegebener Leistungsanforderungen auszuführen.

Um diese Ausführungsknotenanfragen bearbeiten zu können, sammelt der Auslastungdienst entsprechende geeignete Messdaten und bereitet sie auf, um die Anfragen innerhalb kurzer Zeit beantworten zu können.

1.2 Aufbau der Arbeit

Diese Arbeit beginnt mit einem Grundlagenkapitel, das zuerst einen Überblick über das Nexus Framework und das darauf aufbauende NexusDS Framework gibt. Außerdem werden verwandte Arbeiten vorgestellt, die sich mit Vorhersagen von Ressourcenverfügbarkeit und dem Scheduling von verteilten Anwendungen beschäftigen.

In Kapitel 3 wird eine Einordnung des Auslastungsdienstes in die Architektur vorgenommen und die Problemstellung analysiert. Auf dieser Basis werden die Architektur des Auslastungsdienstes und seine Komponenten entwickelt.

Aufbauend auf der zuvor vorgestellten Architektur erfolgt in Kapitel 4 ein detaillierter Entwurf mit der Spezifikation der zu erfassenden Daten und der verwendeten Verfahren. Es wird auf Basis einer Übersicht über verschiedene Clusteringverfahren ein Verfahren ausgewählt und ein Algorithmus zur Implementierung vorgestellt.

In Kapitel 5 wird die Implementierung des Auslastungsdienstes beschrieben. Für das Clusteringverfahren wird eine parallelisierbare Implementierung vorgestellt und in einem kurzen Test gezeigt, dass diese einen deutlichen Geschwindigkeitszuwachs erbringt.

Danksagung

Ich bedanke mich an dieser Stelle bei den Personen, die mir bei der Bearbeitung der Diplomarbeit geholfen haben: Meinem Betreuer Nazario Cipriani, der mich mit vielen Ratschlägen unterstützt hat, meiner Mutter, die auch in schwierigsten Fällen der Kommasetzung eine fundierte Meinung vertreten hat, Sevgi, die mich besonders in anstrengenden Momenten mit einem lächeln und mit freundlichen Worten aufgemuntert hat sowie all den Kommilitonen und Freunden die mich nicht nur während der Diplomarbeit als Gesprächspartner zu neuen Ideen angeregt haben.

Kapitel 2

Grundlagen und verwandte Arbeiten

In diesem Kapitel werden Grundlagen für die Arbeit eingeführt sowie Forschungsarbeiten, die sich mit verwandten Themenbereichen auseinandergesetzt haben, vorgestellt.

Zuerst wird ein Überblick über das NexusDS Framework gegeben. Aus NexusDS stammt die Rahmenarchitektur für den in dieser Arbeit entwickelten Auslastungsdienst. Außerdem ist NexusDS die Plattform für die Implementierung des Auslastungsdienstes. Anschließend werden verwandte Arbeiten zu folgenden vier Schwerpunkten vorgestellt: Welche Indikatoren werden für ein möglichst optimales Scheduling benötigt, wie können sie in einem Peer2peer System erfasst werden, wie und wo werden diese Daten gespeichert und wie können die Daten aufbereitet werden.

2.1 Nexus

Nexus ist eine Plattform für ortsbasierte und kontextbezogene Anwendungen [BDG+04]. Ortsbasierte Anwendungen verwenden Daten abhängig von dem aktuellen Ort des Benutzers und passen sich entsprechend an. Kontextbasierte Anwendungen reagieren auf den aktuellen Benutzerkontext. Durch das Kontextmodell von Nexus, das Augmented World Model (AWM), lassen sich Objekte aus der realen Welt, virtuelle Objekte und ihre Bezüge als sogenannte *context models* zueinander modellieren.

Die Architektur von Nexus besteht aus drei Schichten: der Application Tier, der Federation Tier und der Service Tier.

Die **Application Tier** enthält Anwendungen, die die Nexus Plattform verwenden. Die Federation Tier enthält Server, die die Funktionen der Plattform für Anwendungen und

Dienste bereitstellen. Anwendungen können context models über die Federation Tier abrufen und sich dort für Mitteilungen über Ereignisse registrieren.

Die Server der **Federation Tier** halten – bis auf Caching – keine Daten vor, sondern sie bearbeiten Anfragen von Anwendungen, sie suchen die Contextserver, die diese Daten bereithalten, fragen die Daten ab, bereiten sie zu einer konsistenten Sicht auf und geben sie an die Anwendung zurück. Sie überwachen Ereignisse und benachrichtigen Anwendungen, die sich hierfür registriert haben.

Die **Service Tier** enthält Context Server, die *context models* speichern und bereitstellen. Context Server verwenden als Anfragesprache die *Augmented World Query Language* (AWQL), die räumliche Anfragen ermöglicht. Daten werden in der Augmented World Model Language (AWML) zurückgeliefert.

2.1.1 AWML – Augmented World Model Language

Als grundlegendes Datenaustauschformat von Nexus und allen darauf basierenden Projekten und auch den in dieser Arbeit eingeführten Schnittstellen wird AWML hier etwas ausführlicher vorgestellt.

AWML ist ein XML-basiertes Datenaustauschformat, das durch XML-Schemata definiert wird.

Die Extensible Markup Language **XML** [BYS+06] ist eine textbasierte Auszeichnungssprache für maschinenlesbare Dokumente. Sie basiert auf einer recommendation, einer Empfehlung, des W3C[1]. Durch Einschränkung der erlaubten Elemente von XML lassen sich anwendungsspezifische Sprachen erzeugen.

Die Einschränkungen für XML, also die Definition der erlaubten Elemente und ihrer Struktur in einem Dokument, wird durch Schemasprachen erreicht. XML-Schema ist eine selbst in XML definierte Schemasprache [WF04]. XML-Schema stellt unter anderem vordefinierte atomare Typen wie *xsd:string* und *xsd:float* bereit. Zusätzlich können weitere komplexe Typen definiert werden. XML-Schemata werden meist in xsd-Dokumenten (XML Schema Definition) gespeichert.

Dokumente, die dem XML-Standard entsprechen, werden als wohlgeformt bezeichnet. Dokumente, die einen Verweis auf ein XML-Schema enthalten und diesem Schema entsprechen, nennt man gültig.

[1] W3C World Wide Web Consortium - ein Gremium zur Standardisierung von World Wide Web Techniken.

AWML wird durch das *Nexus AWML Schema*[2] definiert und verwendet seinerseits das *Nexus Standard Attribute Schema*[2], das die erlaubten Datentypen in Nexus definiert.

2.2 NexusDS

NexusDS [CEB+09, CLM10] ist eine Middleware für die verteilte Verarbeitung von Datenströmen. Eine Middleware verbirgt die Komplexität der ihr zugrundeliegenden Funktionalität für eine auf sie aufsetzende Anwendung. NexusDS setzt auf die Nexus Plattform [BDG+04] auf und erweitert Nexus um die Verarbeitung von Datenströmen. Datenströme sind Daten, die potenziell unendlich lang sind und auf die kein wahlfreier Zugriff besteht.

Ein Einsatzbeispiel für NexusDS ist eine interaktive Streamingpipeline für eine ortsbasierte Anwendung. Die in Abbildung 2.1 dargestellte Anwendung zeigt auf einem Handheld eine rotierbare dreidimensionale Karte an. Da das Endgerät weder die notwendigen Daten noch die Rechenleistung für die Berechnung der Darstellung hat, werden die Berechnungen in das Netzwerk verlagert.

Die einzelnen Operationen der Streamingpipeline können durch verschiedene Knoten, sogenannte **Stream Nodes**, zur Verfügung gestellt werden, die die von der jeweiligen Operation benötigten Eigenschaften aufweisen. So sollte die Merge and Filter Operation schnellen Zugriff auf die Datenbanken mit den Umgebungsinformationen haben, während der Rendering-Knoten eine GPU benötigt. Als Datenquellen der Pipeline dienen Datenströme wie die Position des Handhelds und aktuelle Daten über die Umgebung. Die Datensenke – die Ausgabe – ist hier das Display des Handhelds. Das Userinterface am Handheld dient auch dazu, die aktuellen Parameter für die einzelnen Schritte der Pipeline vorzugeben. So können am Handheld unterschiedliche Datensätze für die Kartendarstellung oder unterschiedliche Blickwinkel auf die 3D-Karte ausgewählt werden.

Eine von einer Anwendung verwendete Streamingpipeline wird durch einen Nexus Plan Graph Model Graphen (NPGM) beschrieben. Ein NPGM-Graph enthält die Datenquellen und Senken und die sie verknüpfenden Operationen. Die in einem NPGM-Graphen vorkommenden Operation können jeweils entweder logische oder physische Operationen sein. Eine logische Operation ist eine abstrakte Beschreibung einer Operation, während eine physische Operation einen bestimmten Operator in einer festgelegten Implementierung meint.

[2] Verfügbar unter http://nexus.informatik.uni-stuttgart.de/en/research/documents/

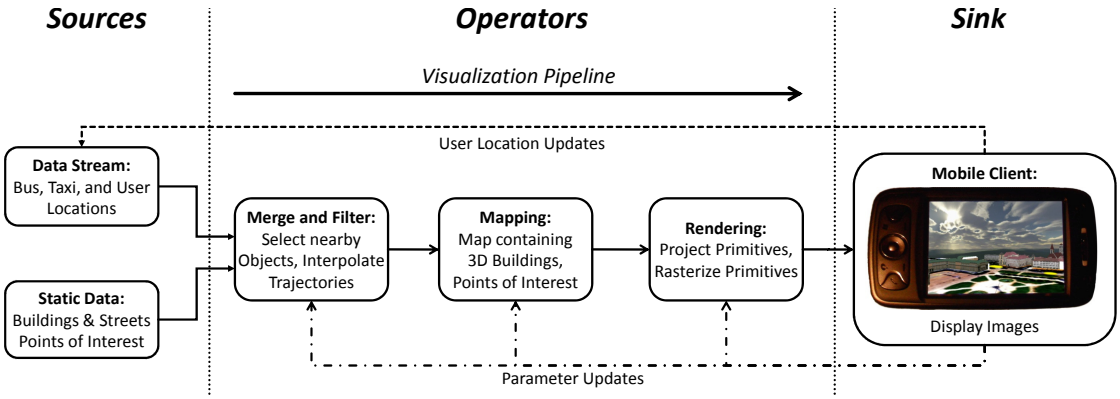


Abbildung 2.1: Beispiel interaktive ortsbasierte Visualisierungspipeline mit mobilem Client (aus [CLM10])

Die Eigenschaften der Operatoren in NexusDS werden durch Metadaten beschrieben. Die Metadaten definieren unter anderem die Anzahl der Ein- und Ausgänge, ihre Datentypen sowie spezielle Anforderungen des Operators. Anforderungen können zum Beispiel eine bestimmte Hardware-Plattform oder spezielle Leistungs- und Umgebungsanforderung sein.

2.2.1 Architektur

Die NexusDS Architektur besteht aus vier Schichten: Der *Communication and Monitoring*, der *Nexus Core*, der *Nexus Domain Extensions* und der *Nexus Application Extensions* Schicht. In Abbildung 2.2 sind die vier Schichten zu sehen. Dienste sind mit durchgezogenen Kästen versehen, während Operatoren die gestrichelten Kästen innerhalb der jeweiligen Schicht sind.

Die **Communication and Monitoring** Schicht stellt die Kommunikationsinfrastruktur basierend auf einem Peer2Peer-Netz zur Verfügung. Sie enthält den Monitoring Service, der das System überwacht und die teilnehmenden Knoten und ihre Eigenschaften verwaltet. Der Service Publisher Service verwaltet die im System angebotenen Services und ermöglicht die Veröffentlichung und Nutzung von Services.

Die **Nexus Core** Schicht enthält die Kerndienste von NexusDS. Hierzu gehören Core Operators, Core Query Service, Operator Repository Service und Operator Execution Service.

Der Operator Repository Service ist ein Speicherplatz für Operatoren und ihre Metadaten. Es enthält auch die Informationen zu den Core Operatoren, den grundlegenden

Operatoren wie Join und Selection. Datenquellen und Senken werden als Operatoren mit nur Eingängen oder nur Ausgängen vom Operator Repository Service gespeichert. Zusätzliche Operatoren, die neue oder spezifische Funktionalität bereitstellen, werden in das Operator Repository eingepflegt und so Anwendungen zur Verfügung gestellt.

Der Core Query Service verarbeitet Nexus Query Graphen und übersetzt die logischen Anfragegraphen in einen ausführbaren Graphen, indem er in mehreren Schritten den NPGM-Graphen optimiert, fragmentiert und ausführt. Die Optimierung wird durch den Query Optimizer durchgeführt. Die Fragmentierung, also die Aufteilung in kleinere Elemente, wird durch den Query Fragmenter durchgeführt. Der Query Fragmenter belegt die logischen Operationen mit physischen Operatoren und findet geeignete Ausführungsknoten für die physischen Operatoren. Hierfür greift der Query Fragmenter auf Ausführungsstatistiken zurück, um eine geeignete, ausführbare Aufteilung zu erzeugen. Die einzelnen Fragmente werden letztlich vom Execution Manager ausgeführt. Der Execution Manager lädt, parametrisiert und startet die Operatoren, er kontrolliert ihre Ausführung und kann bei drohender Überlastung von Knoten eine Neuberechnung und Verteilung des NPGM-Graphen veranlassen.

Die **Nexus Domain Extensions** sind eine Zusammenstellung von Nexus Core Diensten für einen bestimmten Anwendungsbereich. Es werden ausschließlich für diesen Bereich interessante Operatoren und Dienste angeboten; alle anderen Elemente des Nexus Core werden ausgeblendet.

Die **Nexus Application and Extensions** Schicht enthält Operatoren und Services, die ausschließlich für eine bestimmte Anwendung bestimmt sind. Sie ermöglicht, Teile einer Anwendung in die Middleware zu verschieben, um die Anwendung und damit die Hardware auf dem Endgerät zu entlasten.

2.2.2 Anwendungsstart

Um das Zusammenspiel der einzelnen Komponenten in NexusDS zu verdeutlichen, wird hier anhand des Beispieles eines Anwendungsstarts der Ablauf durchgegangen.

Eine Anwendung, die NexusDS verwendet, startet mit einer Anfrage an den ihr entsprechenden Application Service und bekommt eine Query ID oder eine Fehlermeldung zurück. Der Application Service bildet die Anfrage auf einen logischen Operator Graphen ab und gibt ihn an den Core Query Service weiter.

Der Core Query Service (CQS) führt auf dem logischen NPGM-Graphen Optimierungen wie das Vorziehen von Selektionen oder frühe Projektionen aus. Der optimierte Graph wird innerhalb des CQS vom Query Fragmenter in für eine Ausführung geeignete Fragmente zerlegt. Der QF fragt den Monitoring Service hierfür nach geeigneten Knoten

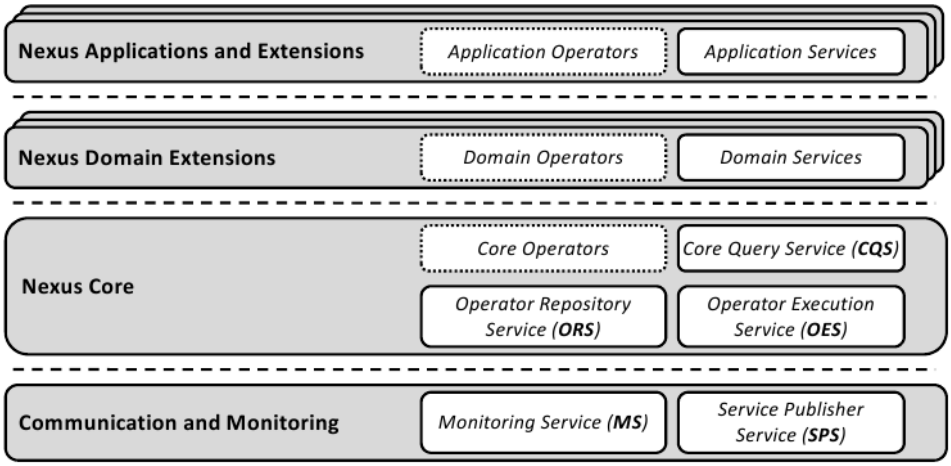


Abbildung 2.2: Architektur NexusDS aus [CEB+09]

und ihren Statistiken an. Die Anfrage an den Monitoring Service enthält Informationen über den physischen Operator sowie seine Konfiguration und Einschränkungen, die zu beachten sind. Der Monitoring Service erstellt anhand seiner Statistiken und der aktuellen Auslastungsdaten der Knoten eine Liste von Vorschlägen mit verfügbaren Knoten, die den Operator ausführen können. Anhand der vorgeschlagenen Knoten berechnet der QF einen Gesamtgraphen, der aus lauter ausführbaren Fragmenten besteht.

Die einzelnen ausführbaren Fragmente des NPGM-Graphen werden nun vom Execution Manager des CQS an den Operator Execution Service (OES) zur Ausführung übertragen.

Der Operator Execution Service überprüft bei Empfang eines auszuführenden NPGM-Graph-Fragments, ob die zur Ausführung benötigten Operatoren lokal vorhanden sind. Fehlende Operatoren werden vom Operator Repository Client von dem Operator Repository Service nachgeladen und installiert.

Die Operatoren des NPGM-Graph-Fragments werden parametrisiert und innerhalb der Operator Execution Sandbox gestartet. Sobald alle Operatoren des NPGM-Graph-Fragments gestartet sind, wartet der OES auf das Eintreffen von Daten für die Bearbeitung durch das Fragment. Innerhalb des OES arbeitet der Statistics Collector, der Leistungs- und Auslastungsdaten der Operatoren und des Knotens, auf dem der OES läuft, sammelt und an den Monitoring Service meldet.

2.3 Auslastungsmessung und Messdaten für Scheduling

Um eine effiziente Verteilung von Operatoren auf Knoten, die eine ausreichende Leistung zur Verfügung stellen, zu ermöglichen, braucht der Verteilungsalgorithmus Informationen über die verfügbaren Knoten, ihre Leistungsfähigkeit und aktuelle Auslastung. Diese werden dem Ressourcenbedarf der einzelnen Operatoren gegenübergestellt. Es stellt sich die Frage, ob und wie aus historischen Daten Prognosen für zukünftiges Systemverhalten abzulesen sind.

In diesem Abschnitt werden zuerst Arbeiten, die sich mit der Messung des Ressourcenbedarfs von verteilten Anwendungen oder verallgemeinert mit „Problemen“, wie der Vorhersage von Ressourcenverfügbarkeit und der Optimierung der Verteilung von Berechnungen, beschäftigen. Im Folgenden werden drei Ansätze vorgestellt: Der erste Lösungsansatz für Scheduling sieht vor, dass die berechnete Funktion des Problems bekannt ist und sich als mathematisches Minimierungsproblem darstellen lässt. Der zweite Ansatz beobachtet die Ausführung des verteilten Programms im laufenden Betrieb, um daraus für zukünftige Ausführungen Statistiken zu erstellen. Zuletzt wird ein Verfahren vorgestellt, das durch Testen des Programmes mit unterschiedlichen Problemgrößen und auf unterschiedlichen Systemen eine Annäherung von mathematischen Funktionen versucht, über deren Parameter die Abbildung auf unterschiedliche Architekturen und Systemgrößen möglich wird.

2.3.1 Application-level scheduling on distributed heterogeneous networks und Network Weather Service

In [BWF+96] wird anhand des Beispiels einer zweidimensionalen Jacobi Matrix als Berechnungsproblem demonstriert, wie sich die Berechnung des Problems über verteilte Knoten mit unterschiedlichen Netzwerkanbindungen optimal verteilen lässt. Die Problemstellung ist, wie in Abbildung 2.3 dargestellt, eine Matrix, in der jeder Wert der Durchschnittswert seiner vier nächsten Nachbarn aus der letzten Iteration ist.

Die zu berechnende Matrix wird in Streifen aufgeteilt, deren Randbereiche nach jedem Rechenschritt mit den Knoten, die den benachbarten Bereich bearbeiten, ausgetauscht werden müssen. Für dieses exakt definierte Problem wird für die Aufteilung ein lineares Gleichungssystem angegeben, dessen Lösung eine nahezu optimale Verteilung angibt.

Als Eingangsparameter für das lineare Gleichungssystem wird sowohl die aktuelle Systemlast der verfügbaren Knoten als auch eine Vorhersage der voraussichtlichen Last und Netzwerkverfügbarkeit durch einen „Network Weather Service“ verwendet.

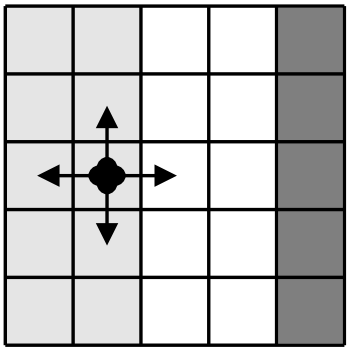


Abbildung 2.3: Jacobi Matrix: Jeder Wert ist der Durchschnitt seiner Nachbarn in der letzten Iteration

Der „Network Weather Service“ [Wol97, WSP97] implementiert zwei unterschiedliche Typen von Sensoren, sogenannte aktive und passive Sensoren. Die aktiven messen experimentell, indem sie die zu messende Ressource belasten. Sie werden zum Beispiel für die Netzwerk-Bandbreitenmessung und -Latenzmessung eingesetzt. Im Gegensatz dazu lesen die passiven Sensoren im laufenden Betrieb anfallende Daten aus, sie liefern Informationen über die CPU-Auslastung, den freien und belegten Arbeitsspeicher sowie die Festplattenauslastung oder die Anzahl der Prozesse.

Wolski et al. sehen bei den aktiven Sensoren mehrere Probleme. Jede experimentelle Messung beeinflusst die verfügbaren Ressourcen und kann so zu Engpässen führen und somit wiederum das Messergebnis beeinflussen. Vor allem bei der parallelen Durchführung mehrere Bandbreitenmessungen zwischen Knoten beeinflussen sich die Messungen gegenseitig sehr stark und werden somit verfälscht. Um die parallele Messung zu verhindern, wird im „Network Weather Service“ ein Tokenverfahren verwendet. Es darf jeweils nur der Knoten, der das Token besitzt, eine Messung durchführen; danach reicht er das Token weiter. Das Tokenverfahren hat jedoch den Nachteil, dass es für große Verbünde nicht skaliert und eine Tokenverwaltung zum Schutz gegen einen Verlust des Tokens notwendig ist.

Sobald eine Messung auf einem Knoten abgeschlossen ist, wird eine neue Vorhersage berechnet. Der Network Weather Service in [Wol97] verwendet drei unterschiedliche Berechnungsmethoden für jede Vorhersage und wählt diejenige, aus der die Abweichung der letzten Vorhersage am geringsten zum aktuellen Zustand ist. Die drei verwendeten Berechnungsmethoden sind durchschnittsbasiert, medianbasiert und „Auto Regressive Moving Average“. Bei der durchschnittsbasierten Methode wird nicht zwingend die vollständige Historie berechnet, sondern nur ein einstellbarer Teilbereich aus der jüngeren Vergangenheit. Die medianbasierte Berechnung verwendet ebenfalls nur eine Teilhistorie und hat als Vorteil, dass starke Ausreißer zuverlässig gefiltert werden.

2.3.2 MARS – A framework for minimizing the job execution time in a metacomputing environment

MARS (Meta Computer Adaptive Runtime System) [Geh96] ist ein Framework, das parallele Programme in einem heterogenen verteilten System auch über WAN-Verbindungen optimiert. MARS sammelt Daten über das Programmverhalten und das Systemverhalten, um aus den statistischen Daten und der aktuellen Auslastung eine optimierte Verteilung zu erstellen. Das Programmverhalten, also der Rechenzeitbedarf in einzelnen Programmabschnitten, und das Kommunikationsverhalten werden als gerichteter Abhängigkeitsgraph dargestellt, während die knotenspezifischen Daten in tabellarischer Form verwaltet werden.

Die Abhängigkeitsgraphen jeder Programmausführung werden gespeichert, um so den nächsten Programmaufruf optimieren zu können. Programme, die bei jedem Aufruf unterschiedliche Graphen erzeugen, werden anhand der großen Varianz in ihren historischen Graphen erkannt und können so entsprechend behandelt werden.

Das Monitoring der anwendungsspezifischen Daten erfolgt durch Einfügen von zusätzlichen Programmkonstrukten vor jeder Send oder Receive Operation in den Programmcode. Ein „Application Monitor“ wird von dem zusätzlichen Code über Zeitpunkt und Sende-/Empfangsvolumen informiert. Um den Übergang von einer Programmphase in eine andere, die mit einer Veränderung des Programmverhaltens einhergeht, erkennen zu können, sieht MARS vor, dass zusätzliche Befehle in das Programm eingefügt werden, die das Framework informieren, so dass eventuell eine Umverteilung erfolgen kann.

Das Netzwerk wird von Netzwerkmonitoren auf jeder Station überwacht, die regelmäßig ihre Daten mit den anderen Monitoren im Netzwerk austauschen. Sie messen CPU-Auslastung und Netzwerkbelastung. Jeder Monitor berechnet aus den selbst gemessenen und den von anderen Netzwerkmonitoren empfangen Daten eine globale Sicht. Ausschließlich die auf allen Knoten konsistente globale Sicht wird für Verteilungsentscheidungen verwendet. Die Daten der Netzwerkmonitore werden jeweils als Tupel zwischen zwei Knoten gespeichert und regelmäßig auf Durchschnittswerte für einen Zeitraum verdichtet.

2.3.3 Performance modeling of parallel applications for grid scheduling

Sanjay und Vadhiyar stellen in [SV08] Strategien vor, um durch Annäherung von Funktionen Ausführungszeitvorhersagen für unterschiedliche Systeme zu ermöglichen. In mehreren Phasen werden jeweils mehrere Tests mit unterschiedlichen Problemgrößen durchgeführt. Dabei werden insgesamt sechs Funktionen, die unterschiedliche Teilaspekte der Ausführung beschreiben, angenähert. In die Gesamtformel gehen Funktionen für

die Kommunikations- und Berechnungskomplexität, Funktionen für den Einfluss von Netzwerk und CPU-Auslastung und jeweils eine Funktion für die Abhängigkeit der Geschwindigkeitsveränderung durch Parallelisierung ein.

Für die Bestimmung der beschreibenden Funktionen wird zuerst auf einer einzelnen Ein-CPU-Maschine eine Testreihe gestartet. Aus einem Set von 77 Funktionen werden bis zu 20 Funktionen ausgewählt, die die Laufzeit abhängig von der Problemgröße mit möglichst kleinem Fehler beschreiben. In der zweiten Phase wird auf zwei Rechnern getestet, um den Einfluss unterschiedlicher Netzwerklast zu messen. In der letzten Phase wird die Anwendung auf 2, 4 und 8 Prozessoren getestet und wiederum eine Auswahl an Funktionen getroffen, die die Auslastung beschreiben.

Um die Ausführungszeit für eine gegebene Anwendung, Problemgröße und CPU-Anzahl vorherzusagen, werden die für die Anwendung erstellten Modelle zusätzlich mit Vorhersagen über die Bandbreiten und Rechenzeitverfügbarkeit von einer Network Weather Service Instanz (siehe 2.3.1) gespeist. Die vorhergesagte Laufzeit bei gegebenen Ressourcen ergibt sich dann durch Auswertung der Funktionsannäherungen, die in den Testläufen die geringste Abweichung gezeigt hatten. Nach einem Anwendungsdurchlauf werden die währenddessen aufgezeichneten Daten wiederum den Trainingsdaten hinzugefügt.

Damit sich die zuvor gewonnen Daten über eine Anwendung von einer Plattform auf eine andere übertragen lassen, führen Sanjay und Vadhiyar zusätzliche Tests auf der Zielplattform aus, um neue Koeffizienten für Funktionen zu bestimmen. Zum einen wird in einem einzelnen Test ein Faktor für die CPU-Geschwindigkeit gemessen, zum anderen in mehreren Tests für zwei CPUs der Einfluss auf die Datenübertragung bestimmt.

Da in jeder einzelnen Phase eine große Anzahl von Kombinationen für die möglichen Funktionen ausgewertet wird, müssen in einem einzelnen Schritt bis zu 12 320 Funktionen und ihre Abweichung berechnet werden. Um den Berechnungsaufwand im Betrieb zu senken, werden beim Eintreffen neuer Trainingsdaten Funktionen, die zu häufig hohe Abweichungen aufgewiesen haben, aus der Liste der für diese Anwendung möglichen Funktionen gestrichen.

2.3.4 VisPerf: Monitoring Tool for Grid Computing

Ein weiteres Monitoring System wird in [LDR03] vorgestellt. Es unterscheidet sich von den vorherigen, indem es Peer2Peer-Verfahren verwendet, um eine besser Skalierbarkeit zu erreichen.

Im vorgestellten Monitoring-Netz existieren einfache Sensoren, repräsentative Sensoren und ein Monitoring Directory Service. Die einfachen Sensoren nehmen über diverse Schnittstellen Messdaten ihres lokalen Knotens auf. Sie verfügen über eine Filterfunktion

und können die Messdaten entweder zur Abfrage bereitstellen (pull) oder laufend an einen anderen Knoten weitergeben (push).

Die repräsentativen Sensoren melden sich als push-Empfänger an einfachen Sensoren an und empfangen deren Daten, die empfangenen Daten werden aggregiert und wiederum als Sensor repräsentativ für die Domäne bereitgestellt. Der repräsentative Sensor meldet sich beim Start am Monitoring Directory Service an und kann von Interessenten, zum Beispiel dem Scheduler, über eine Anfrage beim Monitoring Directory Service aufgefunden werden. Wenn exakte Daten eines einzelnen Sensors abgefragt werden sollen, kann der repräsentative Sensor nach der Adresse gefragt werden und über diese eine Anfrage direkt an den entsprechenden Sensor gestellt werden.

Die einfachen Sensoren werden von den repräsentativen Sensoren in einer zentralisierten Struktur verwaltet, während die repräsentativen Sensoren untereinander ein Peer2Peer-Netzwerk bilden und den Monitoring Directory Service bereitstellen.

Kapitel 3

Architektur des Auslastungsdienstes

Dieses Kapitel gibt eine Übersicht über den Auslastungsdienst und die mit ihm zur Messdatensammlung verbundenen Sensoren. Zuerst wird die Einbettung des Auslastungsdienstes in die Architektur des bereits in Abschnitt 2.2 vorgestellten NexusDS erklärt und ein Überblick über die Komponenten und Zusammenhänge des Gesamtsystems gegeben.

Im zweiten Abschnitt wird die Problemstellung analysiert und daraus die grundlegende Arbeitsweise des Auslastungsdienstes entwickelt. Die Architektur des Auslastungsdienstes wird zunächst in einer logischen Sicht als Einheit betrachtet.

Darauf folgend werden in Abschnitt 3.3 die vom Auslastungsdienst gespeicherten Daten und Statistiken vorgestellt.

In Abschnitt 3.4 werden die Sensoren – die Messdatenlieferanten – des Auslastungsdienstes vorgestellt und die von ihnen zu sammelnden Messdaten und die Verarbeitung der gesammelten Daten erläutert.

In Abschnitt 3.5 wird der Ablauf einer Knotenanfrage des Anfragedienstes bis zu einer Antwort des Auslastungsdienstes vorgestellt. Hierbei werden die einzelnen Schritte des Filterprozesses vorgestellt.

Im letzten Abschnitt wird die logische Sicht als Einheit auf den Auslastungsdienst erweitert und das Konzept für eine verteilte Ausführung des Auslastungsdienstes vorgestellt.

Der Entwurf mit detaillierten Daten, Verfahren, Schnittstellen und Spezifikationen folgt dann in Kapitel 4.

3.1 Überblick

Das in Abschnitt 2.2 vorgestellte Streaming Framework NexusDS sieht in seiner Architektur die Funktionalitäten des Auslastungsdienstes und der Sensoren vor. Da die Implementierung des Auslastungsdienstes NexusDS als Laufzeitsystem verwendet, wird hier der Bezug der Komponenten des Auslastungsdienstes zur Architektur von NexusDS hergestellt.

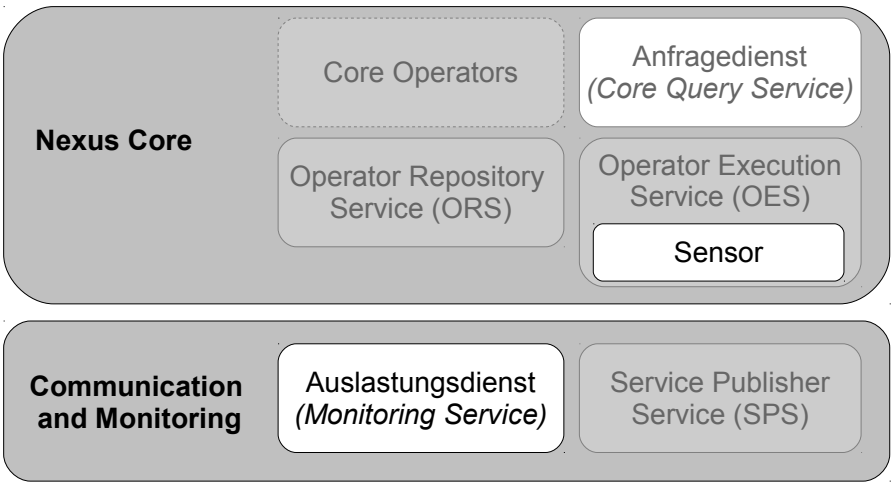


Abbildung 3.1: Bezug zur NexusDS Architektur

Abbildung 3.1 ist ein Ausschnitt der untersten zwei Schichten der NexusDS Architektur, vergleiche auch Abbildung 2.2. Weiß hervorgehoben in der Communication and Monitoring Schicht definiert NexusDS den Monitoring Service, der hier als Auslastungsdienst referenziert wird. Die Sensoren befinden sich, ebenfalls weiß hervorgehoben, in der NexusDS Architektur als Teilfunktion der Operator Execution Services in der Nexus Core Schicht. NexusDS verwendet ein Service Konzept, über das die verschiedenen Dienste gestartet und im Netzwerk verfügbar gemacht werden [Koco9].

Der Auslastungsdienst hat die Aufgabe, dem Anfragedienst – in Abbildung 3.1 ebenfalls weiß markiert – Ausführungsknoten für vom Anfragedienst vorgegebene Operatoren zu liefern.

Abbildung 3.2 stellt die Zusammenhänge bei der Ausführung dar. Der Anfragedienst erhält aus der Nexus Application Schicht NPGM-Graphen, die er zur Ausführung bringt. Dafür verwendet er den Auslastungsdienst, um geeignete Ausführungsknoten zu erhalten, dargestellt durch die gestrichelte Verbindung. Damit der Auslastungsdienst dem

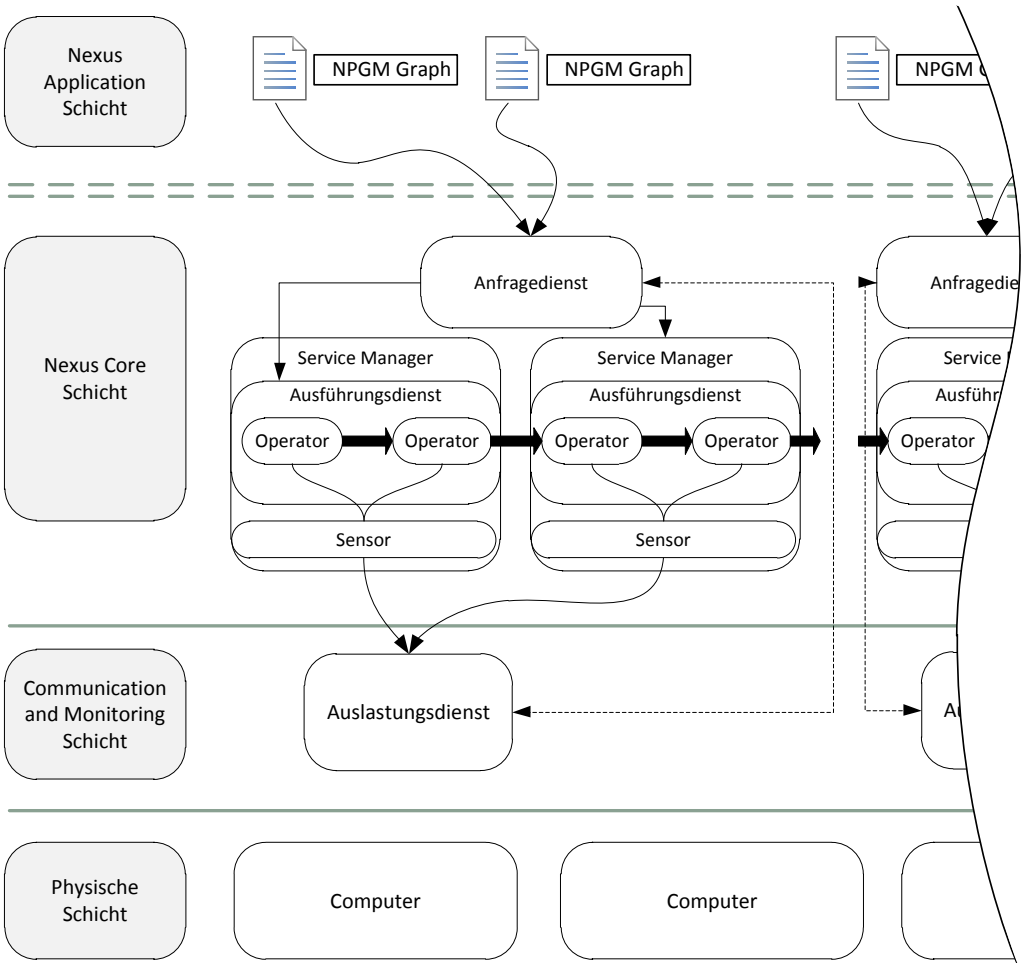


Abbildung 3.2: Übersicht über die Abläufe um Auslastungsdienst und Sensoren

Anfragedienst geeignete Knoten nennen kann, erhält er Messdaten von Sensoren, die als Service auf jedem Ausführungsknoten vom Service Manager gestartet werden. Der Service Manager startet auch den Ausführungsdienst, der vom Anfragedienst NPGM-Graph-Fragmente zur Ausführung erhält. Die Ausführung erfolgt durch die Installation und Ausführung von Operatoren. In der Abbildung ist der Ausführungsknoten selbst nicht dargestellt; er entspricht der Reichweite des Service Managers.

3.2 Problemanalyse

Die Aufgabe des Auslastungsdienstes (AD) ist es, dem Anfragedienst (AFD) eine Auswahl an Ausführungsknoten zur Ausführung von Operatoren zur Verfügung zu stellen. Die Knoten müssen vom AFD vorgegebene Operatoren mit ihren Anforderungen an die Ausführungsumgebung und ihrer aktuellen Parametrisierung innerhalb der vom AFD vorgegebenen Einschränkungen bzw. Leistungsanforderungen ausführen können. Kommen mehr Knoten als angefordert infrage, wird eine Vorselektion nach in der Anfrage enthaltenen Kriterien getroffen.

Damit ein Knoten für die Ausführung eines Operators geeignet ist, muss er einer Vielzahl unterschiedlicher Anforderungen entsprechen. Diese Anforderungen bestehen aus den Plattformanforderungen des Operators, wenn dieser zum Beispiel nur auf einer bestimmten CPU-Architektur, einem speziellen Betriebssystem oder nur bei Vorhandensein spezieller Softwarebibliotheken lauffähig ist. Diese Plattformanforderungen der Operatoren und die ihnen entsprechenden Plattformeigenschaften der Knoten sind statisch. Es wird von der Annahme ausgegangen, dass sich weder die Anforderungen des Operators noch die Plattformeigenschaften des Knotens über die Zeit ändern. Die Anforderungen und Eigenschaften sind harte Entscheidungskriterien. Entweder werden sie exakt erfüllt, oder der Operator ist auf einem die Kriterien nicht erfüllenden Knoten definitiv nicht lauffähig.

Zusätzlich hat ein Operator Anforderungen, die aufgrund verschiedener Einflüsse variabel sind und von Knoten besser oder schlechter erfüllt werden können. Diese Anforderungen unterliegen zum Beispiel der aktuellen Konfiguration eines Operators oder den Einschränkungen durch den NPGM-Graphen (siehe Abschnitt 2.2). Als Veranschaulichung für variable Anforderungen durch unterschiedliche Operatorkonfigurationen stelle man sich einen Operator *Matrixmultiplikation* mit den Parametern Dimension 1 und Dimension 2 vor. Wird dieser Operator mit der Parametrisierung $(1, 1)$ ausgeführt, ist der Anspruch an einen Rechenknoten sehr gering, jedoch würde eine Parametrisierung mit extrem großen Werten für Dimension 1 und 2 sowohl die Ansprüche an den verfügbaren Speicher als auch an die verfügbare Rechenzeit schnell anwachsen lassen.

Einschränkungen durch den NPGM-Graphen können zum Beispiel dadurch entstehen, dass ein NPGM-Graph eine maximal zulässige Zeitspanne hat, um auf Eingaben durch eine Ausgabe zu reagieren. Je nachdem, wie viele Operatoren in Reihe in diesem NPGM-Graphen arbeiten, ist die verfügbare Latenz pro Operator größer oder kleiner. Neben dem Abgleich der Kompatibilität anhand der statischen Anforderungen und Eigenschaften soll der Auslastungsdienst auch Vorhersagen treffen können, welche Knoten für die Ausführung bestimmter Operatoren geeignet sind. Die Operatoren können jedoch, abhängig von ihrer Parametrisierung, sehr unterschiedliche Laufzeiteigenschaften aufweisen. Zusätzlich

kann ein und derselbe Operator mit der gleichen Parametrisierung auf unterschiedlichen Computern unterschiedlich performant arbeiten.

Um trotz diesem komplexen Problem möglichst gute Vorhersagen zu treffen, werden auf allen Knoten Sensoren eingesetzt, die die Ausführung der Operatoren beobachten. Die Sensoren leisten eine Inventur der Knoten, um die statischen Plattformeigenschaften zu bestimmen, und sie sammeln laufend Daten über die Ausführung der Operatoren und die Auslastung des Knotens.

Die Aufgabe des Auslastungsdienstes ist es, unter Zuhilfenahme von Messdaten Aussagen über die Ausführung zukünftiger Operatoren zu treffen. Anders ausgedrückt, muss der Auslastungsdienst aus einer Menge von Knoten solche Knoten auswählen, die für die Ausführung geeignet erscheinen. Für die Auswahl beziehungsweise für das Auffinden von bestimmten Elementen aus einer Menge gibt es verschiedene Verfahren. So kann jedes Element betrachtet und bewertet werden. Um den Aufwand zu verringern, werden in Datenbanken üblicherweise Indexe angelegt, die häufig abgefragte Eigenschaften enthalten. Sind mehrere Eigenschaften gemeinsam zu betrachten, können multidimensionale Indexe eingesetzt werden.

Wenn man davon ausgeht, das alle Knoten, die für einen Operator geeignet sind, *ähnlich* sind kann man versuchen, mit Clustering Algorithmen die Knoten zu gruppieren und dann Knoten aus dem *richtigen* Cluster auszuwählen.

Diese Ansätze haben jedoch jeweils Nachteile. Das Bewerten eines jeden Knotens kann sehr aufwendig sein, da sehr viele Knoten verfügbar sein können, von denen jedoch ein Großteil inkompatibel sein kann. Versucht man, die Auswahl über Indexstrukturen, zu treffen kommt man schnell in kombinatorische Probleme. Die Anforderungen der Operatoren und die Plattformeigenschaften der Knoten sind frei definierbar. Eine Indexstruktur über diese unbegrenzt erweiterbaren Dimensionen hat potenziell sehr viele leere Einträge für Knoten, die Eigenschaften nicht haben, und eine Unzahl von Dimensionen. Für große Dimensionsanzahlen, die auch noch dünn besetzt sind, sind Indexe meist nicht effizient. Hinzu kommt, dass sich manche Eigenschaften der Knoten häufig ändern und andere statisch sind. Zum Beispiel unterliegt die Auslastung einer laufenden Veränderung, während sich die CPU-Architektur eines Knotens nie ändert. Ein Teil dieser großen Indexstruktur müsste also laufend angepasst werden, während andere Bereiche nahezu statisch sind.

Bei der Verwendung von Clusteringverfahren werden Knoten, die nach bestimmten Kriterien ähnlich sind, zu Clustern zusammengefasst. Werden hierfür zu wenige Eigenschaften betrachtet, kommt es zu wenigen großen Clustern, die womöglich viele inkompatible oder ungeeignet Knoten enthalten. Werden andererseits zu wenige Parameter betrachtet, kommt es zu vielen Clustern, die dann jeweils wieder getrennt betrachtet werden müssen.

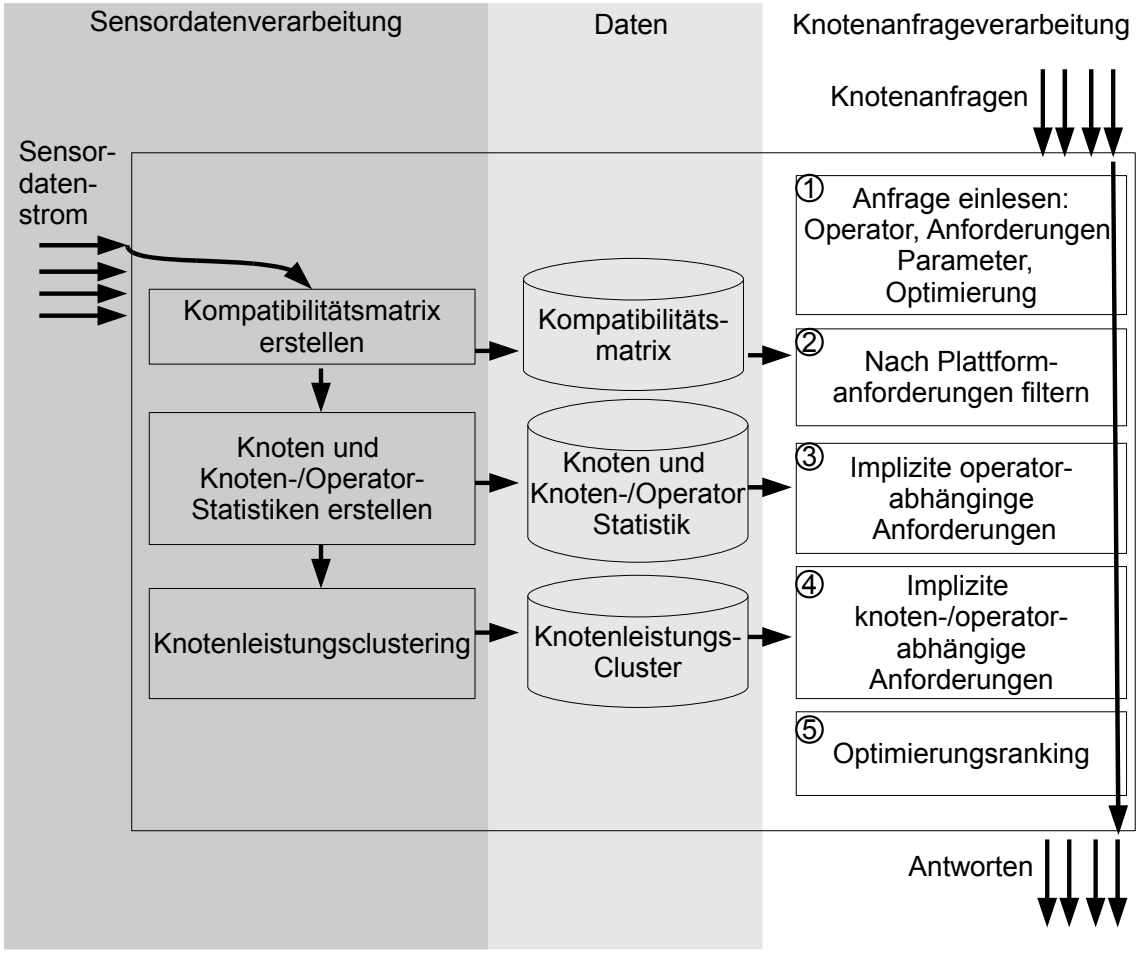


Abbildung 3.3: Architektur Auslastungsdienst

Da alle drei vorgestellten Ansätze für sich nicht geeignet scheinen, um die kombinatorische Komplexität zu beherrschen, wird hier ein kombinierter Ansatz verwendet, indem versucht wird, die Anzahl der zu betrachtenden Elemente möglichst früh zu verringern und aufwendige Berechnungen nur noch auf kleinen Teilmengen durchzuführen.

Hierzu werden die gesammelten Daten in verschiedenen Formen aufbereitet und gespeichert, um für die jeweiligen Aspekte einer Anfrage eine Entscheidungsgrundlage zu bilden.

Um aus den unterschiedlichen Daten unterschiedlicher Quellen, unterschiedlicher Struktur und Granularität geeignete Knoten auszuwählen, verarbeitet der Auslastungsdienst die

Anfragen, indem er die verfügbaren Knoten filtert. Die Filter bestehen zum einen aus Anforderungen, die sich direkt aus der Anfrage ergeben und zum anderen aus statistischen Daten über die Operatoren und ihre Parametrisierungen aus denen Anforderungen bestimmt werden.

Der Auslastungsdienst ist in Abbildung 3.3 schematisch dargestellt. Er verfügt über eine Schnittstelle zur Anbindung der Sensoren, die in Abschnitt 3.4 vorgestellt werden, sowie über eine Schnittstelle zur Annahme von Knotenanfragen und zur Auslieferung von Antworten auf die Knotenanfragen. Zunächst werden in Abschnitt 3.3 die Daten vorgestellt, die für die Auswahl und Vorhersage benötigt werden; sie entsprechen dem hellgrau dargestellten Abschnitt in Abbildung 3.3. Die Daten werden aus den Sensordaten durch die im linken, dunkelgrau dargestellten Abschnitt der Grafik gezeigten Verarbeitungsschritte aus den Messdaten der Sensoren erzeugt. Auf die im rechten Abschnitt dargestellten Knotenanfrageverarbeitungsschritte wird in Abschnitt 3.5 eingegangen.

3.3 Daten

Die verschiedenen zu speichernden Daten sind in Abbildung 3.3 sind im Abschnitt Daten als getrennte Datenspeicher dargestellt. In den folgenden Abschnitten werden diese Daten für die Kompatibilitätsmatrix, die Knoten- und Operatorstatistik und die Daten für das Knotenleistungsclustering jeweils kurz vorgestellt. Wie die Daten erfasst, berechnet und gespeichert werden, wird dann in Abschnitt 3.4 erläutert.

3.3.1 Kompatibilitätsmatrix

Um die Kompatibilität eines Knotens zu einem Operator zu bestimmen, werden die Plattformanforderungen des Operators mit den Plattformeigenschaften des Knotens verglichen. Wenn alle Anforderungen durch entsprechende Eigenschaften erfüllt sind, gilt ein Knoten als kompatibel. Um bei einer Knotenanfrage nicht jeden Knoten einzeln überprüfen zu müssen, wird eine Kompatibilitätsmatrix erzeugt. Beim Hinzufügen eines neuen Knotens zum System, zum Beispiel beim Start des Knotens, werden seine Plattformeigenschaften durch den Sensor ermittelt und im Auslastungsdienst werden diese mit den Anforderungen aller Operatoren abgeglichen. Für den Vergleich werden aus dem Operatorrepository alle Plattformanforderungen aller Operatoren abgerufen. Das Ergebnis der Überprüfung wird in die Kompatibilitätsmatrix eingetragen.

Steht ein Knoten nicht mehr zur Verfügung, weil er zum Beispiel abgeschaltet oder ausgefallen ist, wird sein Eintrag aus der Kompatibilitätsmatrix entfernt. Knoten, die

	Operator 1	Operator 2	...	Operator n
Knoten 1	0	1	1	1
Knoten 2	0	0	0	1
...	1	0	1	1
Knoten n	1	0	0	0

Abbildung 3.4: Beispiel Kompatibilitätsmatrix

abgeschaltet werden, melden dies dem Auslastungsdienst, während ausgefallene Knoten anhand des Ausbleibens von Aktualisierungen erkannt werden.

Die Kompatibilitätsmatrix ist eine Matrix über alle zur Verfügung stehenden Knoten und alle Operatoren des Operatorrepositories. Die Matrix erlaubt ein schnelles Filtern von Knoten, die zu den Plattformanforderungen eines Operators kompatibel sind. Bei einer Kompatibilitätsanfrage für einen Operator wird die Spalte des Operators selektiert, die kompatiblen Knoten können direkt abgelesen werden. Siehe auch Abbildung 3.4.

3.3.2 Knoten- und Operatorstatistik

Um Vorhersagen über die zu erwartenden Leistungsdaten und den zu erwartenden Ressourcenverbrauch eines Operators zu treffen, werden statistische Daten verwendet. Die Auswertungen, die in Abschnitt 3.5.1 vorgestellt werden, arbeiten auf diesen Daten.

Die Sensoren, die in Abschnitt 3.4 vorgestellt werden, liefern dem AD einen Datenstrom, der zuerst die statischen Knoteneigenschaften enthält und danach fortlaufend aktuelle Messwerte überträgt. Der AD bereitet diese Daten auf und aggregiert sie für die spätere Auswertung.

Die Statistiken werden nach Herkunft der Daten in vier Kategorien aufgeteilt und hier der Übersichtlichkeit halber getrennt betrachtet. Es sind die **Knotenstatistik**, die **Operatorstatistik**, die **Knoten/Operatorstatistik** und die **Knoten/Knotenstatistik**.

Die **Knotenstatistik** enthält historische, aufaggregierte Daten zu den einzelnen Knoten über ihre Auslastung (Load, Anzahl der Prozesse) sowie der durchschnittlichen und maximalen Netzwerkauslastung. Neben den historischen Daten enthält die Knotenstatistik auch aktuelle Daten über die Auslastung und die verfügbaren Ressourcen der Knoten (Fest-, Arbeitsspeicher, verfügbare Netzwerkkapazität und verfügbare Rechenleistung).

Die **Operatorstatistik** enthält statistische Daten über den Ressourcenkonsum und die Produktivität einzelner Operatoren, abhängig von ihrer Parametrisierung. Hierfür werden die Daten pro Operator nach Parameter-Presets gruppiert und Parametrisierungen, die keinem Preset entsprechen, auf ähnliche Presets abgebildet. Es werden jeweils Durchschnitts-,

Minimal-, Maximalwerte und die Varianz, also die typische Abweichung vom Durchschnittswert, pro Operator-Presetpaar gespeichert. Die Minimal- und Maximalwerte erlauben die Bestimmung zu erwartender äußerer Grenzen, die voraussichtlich nicht über- oder unterschritten werden. Der Durchschnittswert kann zusammen mit der Varianz verwendet werden, um einen Erwartungswert, der von Ausreißern bereinigt ist, für die Werte zu bestimmen.

Die automatische Bestimmung von Ähnlichkeit zwischen Parametern und Presets ist keine triviale Aufgabe, da eine Vielzahl von Parametern unterschiedlich großen Einfluss auf das Leistungsverhalten eines Operators haben kann. Um irrelevante Parameter von der Ähnlichkeitsanalyse auszuschließen, werden Operatorparameter vom Ersteller des Operators als leistungsrelevant markiert und ausschließlich diese für die Ähnlichkeitsbestimmung verwendet. Bei numerischen Parametern wird der Abstand zwischen einem Preset und einer aktuellen Parametrisierung durch die Summe der Beträge der Differenzen zwischen jeweils gleichnamigen Parametern berechnet.

Bei relevanten Parametern, für die keine Distanz definiert ist, zum Beispiel *String* Parametern, müssen aktuelle und Preset-Parameter übereinstimmen, um eine Abbildung von aktuellen Parametern auf ein Preset zu erlauben. Ein Satz aktueller Parameter wird auf ein Preset abgebildet, dessen relevante *String* Parameter übereinstimmen und dessen weitere relevante Parameter die kleinste Distanz aufweisen. In den Statistiken wird erfasst, ob die Daten exakt einem Preset entsprechen oder einem Preset nur ähnlich sind. In der Statistik werden jeweils für exakt auf ein Preset zutreffende Daten und für solche, die ihm ähnlich sind, getrennt Statistiken geführt.

Die **Knoten/Operatorstatistik** speichert aggregierte Daten über den Rechenzeitbedarf, Ressourcenverbrauch, Latenz und die Produktivität von Operatoren auf einem Knoten. Die statistischen Daten für verschiedene Knoten werden später anhand des Knotenleistungsclusterings zusammengefasst, um eine breitere Datenbasis zu haben.

Die **Knoten/Knotenstatistik** speichert Daten über die Kommunikation zwischen zwei Knoten. Sie soll Vorhersagen über den möglichen Netzwerkdurchsatz und die Netzwerklatenz ermöglichen. Da die laufende Vermessung und Beobachtung eines Netzwerkes sehr stark von der Technologie und der Topologie abhängig ist, sei auf entsprechende Ansätze wie zum Beispiel den bereits in Abschnitt 2.3.1 vorgestellten Network Weather Service verwiesen.

3.3.3 Knotenleistungsclustering

Das Knotenleistungsclustering fasst Knoten mit vergleichbarer zur Verfügung gestellter Rechenleistung in Klassen zusammen. Knoten, die die gleiche CPU-Architektur besitzen, ähnliche Rechenleistungswerte erreichen sowie eine ähnliche Hardwareausstattung haben,

werden in eine gemeinsame Knotenklasse eingefügt. Anhand der Knotenklasse können statistische Daten über Operatorverhalten von einem Knoten auf einen anderen übertragen werden, da zu erwarten ist, dass sich Operatoren auf Knoten, die eine hohe Ähnlichkeit sowohl in der Ausstattung als auch bei den bisherigen statistischen Ergebnissen haben, bei der Ausführung von Operatoren ähnlich verhalten.

Durch die Übertragung der statistischen Daten zwischen mehreren Knoten wird die Datenbasis für die Vorhersagen größer. Es ist so nicht notwendig, für jeden Operator mit jeder Parametrisierung und jedem Knoten Tests durchzuführen.

Die CPU-Architektur und Hardwareausstattung wird aus den Plattformeigenschaften zu Beginn eines Sensordatenstromes bestimmt. Die Rechenleistungswerte können aus Operatorausführungen mit gleichen Parametern auf verschiedenen Knoten bestimmt werden.

Alternativ zur reinen Beobachtung eines laufenden Systems können auch Testoperatoren auf allen Knoten ausgeführt werden, die unterschiedliche Aspekte des Knotens durch Experimente bestimmen und als Produktivitätskennzahl die Leistung des Knotens angeben. So könnte das System mit synthetischen Tests vorab trainiert werden und die Ähnlichkeit der Rechner ohne die Unschärfe der zusätzlichen Einflüsse – wie Parametrisierung, Datenquelle und Senke – bei der Verwendung von Operatoren zur Datensammlung bestimmt werden. Nachteil der Datensammlung durch Experimente ist, dass diese zusätzliche Rechenzeit entweder vor Inbetriebnahme oder im laufenden Betrieb kosten. Bei der Durchführung von Experimenten im laufenden Betrieb bestünde wiederum ein Einfluss durch den regulären Betrieb, und seine Störung wäre nicht auszuschließen.

3.4 Sensordatenverarbeitung

Die im vorherigen Abschnitt vorgestellten Daten werden, wie in Abbildung 3.2 dargestellt, von Sensoren gesammelt. Der Sensor hat die Aufgabe, die Plattformeigenschaften zu bestimmen, die Auslastung und Leistungsfähigkeit des Knotens sowie den Ressourcenbedarf und die Produktivität der auf ihm ausgeführten Operatoren zu beobachten.

Der in Abbildung 3.5 dargestellte Sensor wird vom Service Manager gestartet. Bei seinem Start untersucht der Sensor die statischen Eigenschaften des Knotens und erstellt daraus eine Liste von Plattformeigenschaften. Sobald alle Plattformeigenschaften vollständig bestimmt wurden, wird ein Sensordatenstrom zum Auslastungsdienst (AD) geöffnet und die Liste der Plattformeigenschaften übertragen. Der Sensor erhält von den Operatoren, die auf demselben Knoten ausgeführt werden, Leistungsmessdaten, die er sammelt und an den Auslastungsdienst überträgt.

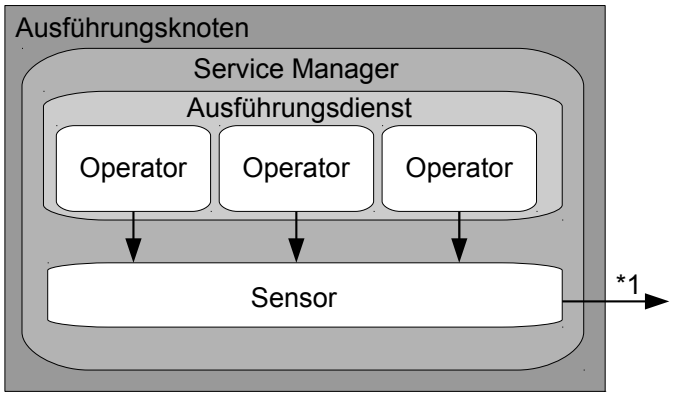


Abbildung 3.5: Knoten, Framework, Sensor und Operatoren
*1 Datenstromschnittstelle zum Auslastungsdienst

Im laufenden Betrieb erfasst der Sensor regelmäßig eine Übersicht über die aktuellen Auslastungsdaten des Knotens und die auf dem Knoten ausgeführten Operatoren und überträgt sie im Sensordatenstrom an den AD. Für den Knoten werden Daten wie Speicher-, CPU-Auslastung und Datentransfer erfasst und für jeden ausgeführten Operator werden Daten über seinen jeweiligen Ressourcenkonsum und seine Produktivität abgefragt. Eine detaillierte Auflistung der erfassten Daten erfolgt in Abschnitt 4.1.2 für den Knoten und in Abschnitt 4.1.3 für die Operatoren. Neben dem Auslastungsdienst können auch weitere Anwendungen als Empfänger des Sensordatenstromes eingefügt werden. So können entweder weitere Monitoring- oder Lastkontrollaufgaben erfüllt werden oder Operatoren können Messdaten eines Knotens als Steuergröße verwenden, um zum Beispiel ihre Verarbeitung an die aktuelle Auslastungssituation anzupassen. Die Schnittstelle zwischen Sensor und Operator wird in 3.4.1 eingeführt.

Wenn die Ausführungsumgebung auf einem Knoten beendet wird, meldet der Sensor das Ende der Knotenverfügbarkeit an den AD und beendet den Sensordatenstrom.

3.4.1 Operatorleistungsmessung

Jeder Operator verfügt über eine Schnittstelle zum Sensor, die es ermöglicht, seine Produktivität, seinen Ressourcenverbrauch, seine Verzögerung und Parametrisierung abzufragen.

Der Ressourcenverbrauch eines Operators setzt sich aus verbrauchte CPU-Zeit pro Zeiteinheit, belegter Arbeitsspeicher, belegter Festspeicher und Netzwerkbelastung zusammen. Zusätzlich können die Programmierer der Operatoren noch eigene Messwerte definieren, die vom Sensor übernommen und mit den Operatordaten gespeichert werden. Ein Beispiel

für zusätzlich definierte Messwerte ist die Messung von weiteren Indikatoren, wie zum Beispiel der GPU-Leistung. Ein Operator, dessen Funktion sowohl die CPU als auch die GPU benötigt, kann so für CPU und GPU getrennte Messwerte erfassen und über Einschränkungen Mindestanforderungen für diese stellen.

Da die Operatoren äußerst unterschiedliche Aufgaben haben können, ist eine allgemeine Definition von Produktivität anhand von Eingabe und Ausgabevolumen nicht für alle Operatoren sinnvoll. Daher ist die **Produktivität eines Operators** eine einheitenlose, operatorspezifische Größe. Die Definition der Produktivität eines Operators und ihre Berechnung obliegt dem Ersteller des Operators. Die Produktivitätskennzahl soll die Leistungsfähigkeit des Operators auf dem Knoten widerspiegeln. Dementsprechend ist bei der Definition der Kennzahl darauf zu achten, dass sie unabhängig von der Eingangsdatenverfügarkeit durch einen vorhergehenden Operator ist und nur von der Leistung des Knotens abhängt. Ein Negativbeispiel wäre die Berechnung durch erzeugte Elemente pro Zeit. Bei einem Operator, der nur auf Eingabedaten arbeitet, könnte die Produktivitätskennzahl also beliebig schlecht werden, wenn keine Eingabeelemente verfügbar sind. Zweckdienlicher wäre es in diesem Fall, CPU-Zeit pro Eingabeelement als Kennzahl zu verwenden.

Wenn mehrere unterschiedliche Implementierungen eines logischen Operators vorliegen, ist darauf zu achten, dass ihre jeweiligen Produktivitätskennzahlen nach den gleichen Berechnungsvorschriften erzeugt werden. So können die statistischen Daten für austauschbare Operatoren auf höheren Architekturebenen verwendet werden, um die Auswahl zu treffen, welche Implementierung eines Operators verwendet wird. Um die Vergleichbarkeit erkennbar zu machen, wird die Produktivitätskennzahl über eine ID identifiziert. Diese ID wird zusammen mit den Produktivitätskennzahlen bei der Erfassung der Operatormessdaten gespeichert.

Wenn ein neuer Operator dem System hinzugefügt wird, bestehen zwei Möglichkeiten. Wenn der Operator eine neue logische Operation implementiert, ist sicherzustellen, dass der Operator eine eindeutige, also noch nicht vorhandene ID für seine Produktivitätskennzahl hat.

Ist der neue Operator jedoch eine weitere Implementierung für eine bereits im System bekannte Operation, muss die Produktivitätskennzahl den vorhandenen Operatoren entsprechen. Es muss also durch einen Administrator überprüft werden, dass alle Operatoren, die die gleiche logische Funktion implementieren, die gleiche ID und Berechnung für die Produktivitätskennzahl verwenden. Eine abweichende Berechnung der ID könnte sonst zu einer fälschlichen Bevorzugung eines Operators führen, da die Werte unterschiedlich berechneter Produktivitätskennzahlen nicht vergleichbar sind.

Gerade bei Echtzeitanwendungen ist es wichtig, Daten über die Verzögerung durch die einzelnen Operatoren zu haben, um eine Gesamtlatenz der Berechnung bestimmen zu

können. Jeder Operator gib daher als Messwert seine Latenz mit an. Je nach Operatortyp ist ihre Bedeutung leicht unterschiedlich. So ist bei einem Operator, der nur Eingabeelemente bearbeitet und direkt wieder ausgibt, die Latenz die Zeit zwischen dem Eintreffen und der Ausgabe des Elementes. Wenn man jedoch einen Videobildgenerator als Beispiel nimmt, der in einem festen Takt Bilder ausgibt, auch wenn keine neuen Eingaben vorliegen, muss die Latenz anders berechnet werden. Im Fall des Videobildgenerators ist die Latenz als die Zeitspanne zwischen dem Eintreffen neuer Daten und ihrer Auswirkung in der Ausgabe definiert. Bei einem Selektionsoperator hingegen ist die Latenz die Zeitspanne vom Eintreffen der Daten bis zum Zeitpunkt, zu dem die Selektionsentscheidung getroffen wurde und an der Ausgabe anliegt.

Anhand von Basisoperatoren von NexusDS wird verdeutlicht, wie die Definition einer Produktivitätskennzahl aussehen kann. Die Basisoperatoren von NexusDS sind in [Dör09] definiert. Für die Operatoren Selektion und Sortierung werden mögliche Produktivitätskennzahlen angegeben. Für die weiteren Basisoperatoren erfolgt die Definition der Produktivitätskennzahl analog.

Die Operation **Selektion** vergleicht die Objekte eines Eingangsdatenstroms mit einem vorgegebenen Attribut oder sie vergleicht die Elemente zweier eingehender Datenströme paarweise. Es wird also Element 1 von Datenstrom 1 mit Element 1 von Datenstrom 2 verglichen. Elemente, für die der Vergleich *wahr* ergibt, werden ausgegeben. Eine Produktivitätskennzahl für diese Operation soll nun bewerten, wie leistungsfähig ein Operator auf einem Knoten ist. Es wird davon ausgegangen, dass zu jedem Zeitpunkt Datenelemente im Eingang verfügbar sind. Die Leistungsfähigkeit des Knotens zeigt sich darin, wie viele Objekte pro Zeiteinheit verglichen werden können. Dabei ist zu beachten, dass der Vergleich eines Attributs weniger aufwendig ist als der Vergleich mehrerer Attribute. Als Produktivitätskennzahl wird daher `(Vergleiche/Zeit)` vorgeschlagen. Hierbei gibt `Vergleiche` die Anzahl der durchgeführten Vergleiche an, also zum Beispiel für zwei Datenströme mit jeweils 5 Elementen, für die 3 Attribute verglichen werden, 15. `Zeit` ergibt sich aus dem Zeitraum, in dem die Anzahl der Vergleiche gezählt wurde. Dies ist der Zeitraum seit der letzten Übertragung der Messdaten. Die Messdaten können hier entweder nach x Elementen übertragen werden oder zum Beispiel alle 30 Sekunden.

Die Operation **Sortierung** sortiert endliche Datenströme oder unendliche Datenströme, über die Informationen zu einer Vorsortierung bekannt sind. Diese unendlichen Datenströme können dann wiederum wie endliche Datenströme behandelt werden. Da verschiedene Sortierverfahren implementiert werden können und anhand der Produktivitätskennzahl auf höherer Architekturebene eventuell eine Auswahl der konkreten Implementierung erfolgen soll, ist es sinnvoll, die unterschiedlichen Verfahren vergleichbar zu machen. Es wird davon ausgegangen, dass die Sortierverfahren *in-place*-Verfahren sind, also die zu sortierenden Objekte nicht innerhalb des Speichers verschoben werden müssen. Deshalb wird der Aufwand dafür nicht betrachtet. Bei Sortierverfahren steigt der Aufwand

für die Sortierung mit der Anzahl der zu sortierenden Elemente und ist nicht, wie bei der zuvor vorgestellten Selektion, linear. Um diesem Anstieg des Aufwands abhängig vom Sortierverfahren genüge zu tragen, ist der Vorschlag für die Produktivitätskennzahl von Sortieroperatoren `(Elemente / (Aufwand * Zeit))`. Es wird also die Anzahl der Elemente um den durchschnittlichen Aufwand des Sortierverfahrens bereinigt und auf ein einheitliches Zeitmaß gebracht. Die Produktivitätskennzahl wird hier immer berechnet, wenn eine Sortierung abgeschlossen ist.

Für beide angegebenen Operatoren gilt, dass das Warten auf Eingabeelemente nicht mit in die Berechnung der Produktivitätskennzahl einfließen darf, da die Wartezeit keinerlei Aussagewert über die Leistungsfähigkeit der Kombination von Operator und Knoten hat. Die Angabe von `Zeit` ist daher immer ohne Wartezeiten an der Ein- oder Ausgabewarteschlange zu verstehen.

3.5 Knotenanfragebearbeitung

Nachdem die notwendigen Daten für die Beantwortung von Knotenanfragen bereits vorgestellt wurden, wird nun die Knotenanfragebearbeitung erläutert.

Eine Knotenanfrage vom Anfragedienst an den Auslastungsdienst nach Ausführungsknoten besteht immer aus einem einzelnen **Operator** und Parametern für die Knotenanfrage. Diese Parameter enthalten **Leistungsanforderungen**, eventuell weitere Anforderungen wie Sicherheitsanforderungen oder Einschränkungen auf bestimmte Knoten. Außerdem wird die **Anzahl** der zurückzuliefernden Knoten und eine **Optimierungsreihenfolge** für den Fall, dass mehr Knoten verfügbar sind, als angefordert wurden, angegeben.

Anhand des angeforderten **Operators** können die Plattformanforderungen aus dem Operatorrepository abgefragt werden. Die **Leistungsanforderungen** geben an, welche operatorspezifische Produktivitätskennzahl erreicht werden soll. Alternativ zur Definition über die Produktivitätskennzahl können auch Leistungsangaben in Rechnergröße wie verfügbare CPU-Zeit, CPU-Geschwindigkeit, maximale Load und verfügbarer RAM angegeben werden. Wenn exakte Leistungsangaben – außer der Produktivitätsanzahl – angegeben werden, wird die Vorhersage über die benötigten Werte anhand statistischer Daten umgangen. Dies ermöglicht es, Wissen über Leistungsanforderungen, das nicht innerhalb des AD bekannt ist, explizit zu verwenden.

Die Anforderungsangaben bestehen aus Tripeln der Form `(Name, Vergleich, Wert)`, wobei der Name vom Typ *String* und der Vergleich aus $\{<, =, >, \neq\}$ ist. Der Wert kann entweder numerisch oder vom Typ *String* sein. Bei *String* Werten ist jedoch nur $\{=, \neq\}$ als Vergleich zulässig.

	Name	Vergleich	Wert
1	Produktivitätskennzahl	>	50
2	Freier RAM	>	1024
3	Operator-Latenz	<	100
4	CPU-VendorID	=	*GenuineAMD*

Tabelle 3.1: Beispiele für Anforderungen

Tabelle 3.1 stellt beispielhaft vier solche Anforderungen dar. Anforderung 1 verlangt eine operatorspezifische Produktivitätszahl größer 50. Es werden also ausschließlich Knoten selektiert, die anhand der statistischen Daten für die gegebene Konfiguration des Operators eine Produktivität > 50 erreichen. Die Produktivitätskennzahl ist, wie bereits in 3.4.1 erläutert, eine einheitenlose abstrakte Maßzahl.

Anforderung 2 fordert mindestens 1024 MB freien RAM; hierbei ist die Maßeinheit für diesen Parameter MB. Anforderung 3 verlangt eine Operatorlatenz kleiner 100 ms, mit ebenso impliziter Einheit. Die 4. Anforderung verlangt, dass die CPU-VendorID einen bestimmten Wert hat, zum Beispiel um herstellerspezifische Fähigkeiten zu verwenden.

Die **Optimierungsreihenfolge** besteht aus geordneten Tupeln der Form `(Name, Optimierung)`, wobei Optimierung aus `{minimieren, maximieren}` ist. Optimierungsangaben sind nur für numerische Werte möglich. Eine Optimierungsreihenfolge könnte zum Beispiel „`(CPU-Takt, Maximieren), (RAM, Maximieren), (Latenz, Minimieren)`“ sein. Die für die Knotenanfrage infrage kommenden Knoten werden im letzten Schritt dementsprechend zuerst nach CPU-Takt (absteigend) sortiert. Knoten mit gleichem CPU-Takt werden nach RAM (absteigend) sortiert und Knoten mit gleichem CPU-Takt und RAM zusätzlich nach Latenz aufsteigend angeordnet.

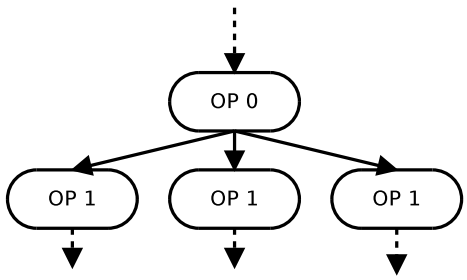


Abbildung 3.6: Beispiel NPGM-Graph Ausschnitt mit mehrfacher, paralleler Ausführung von Operator OP 1

Mit dem Parameter **Anzahl** der Knotenanfrage kann der Anfragedienst dem Auslastungsdienst mitteilen, wie viele Operatoren der Anfragedienst zurückliefern soll. Die

Anforderung von mehr als einem Ausführungsknoten für einen Operator kann mehrere Gründe haben. Zum einen können die Ausführungsknoten zur Auswahl verwendet werden, der Anfragedienst kann also weitere Kriterien zur Selektion verwenden, die nur auf seiner höheren Architekturebene bekannt sind. Zum anderen kann für den Fall, dass der Anfragedienst in seinem NPGM-Graphen mehrere gleiche Operatoren parallel ausführen will, Anfrageaufwand gespart werden. Bei dem in Abbildung 3.6 dargestellten Ausschnitt eines NPGM-Graphen könnten entweder vier Knotenanfragen an den Auslastungsdienst gestellt werden oder stattdessen zwei Knotenanfragen, die jedoch für `OP 1` eine Anzahl von mindestens drei enthält. Dies ist aber nur möglich, wenn alle Operatoren, die zu einer Knotenanfrage zusammengefasst werden, übereinstimmende Anforderungen haben.

3.5.1 Ablauf Knotenanfrageverarbeitung

Eine Anfrage nach geeigneten Knoten wird vom Auslastungsdienst in fünf aufeinander aufbauenden Schritten bearbeitet. Die Schritte 2 - 5 sind eine schrittweise Filterung der verfügbaren Knoten mit dem Ziel, am Ende nur noch geeignete Knoten für die Operatorausführung in einer Liste zu haben.

Die Filterung erfolgt in einzelnen aufeinander aufbauenden Schritten, um den Berechnungsaufwand gering zu halten. So werden zuerst Schritte ausgeführt, die mit geringem Berechnungsaufwand die Knotenmenge auf relevante Knoten einschränken. In den späteren Schritten werden dann Statistiken für einzelne Knoten ausgewertet, eine Operation, die auf der vollständigen Knotenmenge schon in kleinen Systemen zu Engpässen führen würde.

Abbildung 3.3 auf Seite 34 stellt diese Schritte im rechten Drittel unter *Knotenanfrageverarbeitung* in ihrem Ablauf dar.

Schritt 1: Sobald eine neue Knotenanfrage eintrifft, wird diese eingelesen und in die Bestandteile Operator, Anforderungen, Operatorparameter und Optimierungsreihenfolge zerlegt. Die in der Knotenanfrage vorgegebenen Anforderungen werden direkt um die Plattformanforderungen des Operators, die im Operatorrepository verfügbar sind, ergänzt.

Schritt 2: In diesem Schritt wird die Menge der Knoten anhand der Kompatibilitätsmatrix gefiltert. Knoten, die nicht den Plattformanforderungen des Operators entsprechen, werden nicht weiter betrachtet.

Schritt 3: Anhand der leistungsbezogenen Anforderungen des Operators und der Statistik werden implizite operatorabhängige Anforderungen berechnet. Die impliziten

operatorabhängigen Anforderungen geben ausschließlich operator- und parametrisierungsabhängige Leistungsanforderungen an. So sind die benötigte Netzwerkkapazität oder der Speicherverbrauch primär operator- und parametrisierungsabhängig und nicht wie die benötigte CPU-Zeit von Knoteneigenschaften wie CPU-Architektur und Takt abhängig.

Um diesen Unterschied zu verdeutlichen, nehmen wir als Beispiel einen Operator, der Videoframes erzeugt, wie er auch in Abbildung 2.1 auf Seite 20 als Rendering Operator vorkommt. Als Parameter hat er unter anderem das Format der zu erzeugenden Videobilder in der Form `(Höhe, Breite, Farbtiefe, Bildrate)`. Der erzeugte Bilderstrom wird per Netzwerk ausgegeben. Es ist offensichtlich, dass in diesem Beispiel die Kombination der Parameter das Ausgabedatenvolumen beeinflusst, während der verwendete Ausführungsknoten hingegen, solange er die Bilder schnell genug berechnen kann, keinen Einfluss auf das Ausgabedatenvolumen hat. Im Gegensatz zu der ausschließlich operator- und parametrisierungsabhängigen Berechnung des Ausgabedatenvolumens ist die benötigte CPU-Zeit stark knotenabhängig, da ein Knoten je nach CPU-Leistung länger oder kürzer braucht, um die gleiche Berechnung durchzuführen.

Die impliziten Anforderungen ergeben sich aus den statistischen Daten für den Operator mit seinem Preset. Wie in Abschnitt 3.3.2 beschrieben existieren für den Operator und seine Presets Statistiken, die die Minimal-, Durchschnitts- und Maximalwerte enthalten. Die impliziten Anforderungen entsprechend den Minimalwerten aus der Statistik. Als Beispiel sei der minimale Arbeitsspeicherverbrauch aus der Statistik für einen Operator OP_a 128 MB. Ein Knoten, der aktuell nur einen freien Arbeitsspeicher von 96 MB hat, kann diesen Anforderungen also nicht entsprechen.

Die in diesem Schritt errechneten Anforderungen werden verwendet, um die Menge der verfügbaren Knoten einzuschränken. Knoten, die anhand ihrer aktuellen Auslastungsdaten und ihrer statistischen Daten diese Anforderungen nicht erfüllen können, werden herausgefiltert. Beim Widerspruch von expliziten Anforderungen und den errechneten impliziten Anforderungen werden die impliziten verworfen, ebenso wenn die Menge der verfügbaren Knoten die Anzahl der angeforderten Knoten unterschreitet.

Schritt 4: In diesem Schritt werden die kompatiblen Knoten aus dem vorherigen Schritt anhand der Daten aus dem Knotenleistungscluster nach vergleichbarer Rechenleistung gruppiert. Für jedes dieser Cluster wird, wenn für das Cluster entsprechende Statistiken über die Ausführungen eines Operators mit seiner aktuellen Parametrisierung vorhanden sind, berechnet, ob die Knoten ausreichend Rechenleistung zur Verfügung stellen und wie hoch die aktuelle Systemauslastung maximal sein darf, um den Operator ausführen zu können.

Für Knotencluster, über die noch keine Statistiken vorhanden sind, könnte, sofern für den Operator überhaupt Messdaten vorliegen, über die Produktivität anderer Operatoren im Vergleich der Cluster abgeschätzt werden, ob die Knoten leistungsfähig genug sind. Alternativ könnte auch ein Testlauf mit dem Operator und seiner Parametrisierung gestartet werden. Beide Ansätze werden hier nicht weiter verfolgt.

Knoten, für deren Cluster noch keine Daten vorhanden sind, werden dem Anfragedienst in der Antwort auf die Knotenanfrage getrennt von den vorgeschlagenen Knoten mitgeteilt. Der Anfragedienst kann so entscheiden, ob für diese Knoten Statistiken erzeugt werden. Die Statistiken können zum Beispiel durch einen parallelen Testlauf ohnehin anstehender Operationen erzeugt werden.

Anhand dieser impliziten, aus den statistischen Daten der Knotenleistungscluster gewonnenen Anforderungen und der expliziten Anforderungen aus der Knotenanfrage wird die Knotenmenge weiter gefiltert. Ergeben sich aus den impliziten Anforderungen und den expliziten Anforderungen der Knotenanfrage Widersprüche oder sollte die Anzahl der Knoten unter die Anzahl der angeforderten Knoten fallen, werden die impliziten Anforderungen aus diesem Schritt verworfen.

Schritt 5: Im abschließenden Schritt werden die Knoten, die nach den bisherigen Berechnungen als geeignet gelten, nach den Parametern der Optimierungsreihenfolge sortiert. Die Sortierung gibt in absteigender Reihenfolge die geeignetsten Knoten an.

Die ersten der angefragten Anzahl entsprechenden Knoten werden zusammen mit ihren statistischen Daten, den aktuellen Auslastungsdaten, den Statistiken des Operators, den Statistiken aus der Knoten/Operatorstatistik sowie den Netzwerkdaten des Knotens als Antwort an den Auslastungsdienst zurückgeliefert.

3.6 Verteilte Ausführung des Auslastungsdienstes

Bisher wurde der Auslastungsdienst als eine Einheit betrachtet. Diese logische Sicht wird in diesem Abschnitt erweitert. Da NexusDS als verteiltes, ausfallsicheres System konzipiert ist, muss auch der Auslastungsdienst diesem Paradigma folgen.

In einem verteilten System werden verschiedene Formen der Transparenz für die Verteilung betrachtet. Als grundlegende Eigenschaften eines verteilten System gelten Ortstransparenz, Zugriffstransparenz, Replikationstransparenz und Fragmentierungstransparenz (vergleiche auch [Gra05, S. 132]).

Hierbei steht die Ortstransparenz für den Zugriff auf Daten oder Dienste, unabhängig von ihrem aktuellen Speicherort und von der Kenntnis einer ortsbasierten Adresse.

Zugriffstransparenz steht für Zugriff ohne Datenkonvertierungsprobleme, Replikationstransparenz sorgt für die automatische Replikation der Daten ohne weitere Interaktion durch die Anwendung. Fragmentierungstransparenz erlaubt den Zugriff auf physisch nicht zusammenhängend oder am gleichen Ort gespeicherte Daten, ohne eine spezielle Behandlung dafür zu benötigen.

Nach außen, also für auf den Auslastungsdienst zugreifende Anwendungen, sind alle Transparenzen durch das Dienstekonzept von Nexus DS gegeben. Der Auslastungsdienst kann über seine Netzwerkgruppe aufgefunden werden, und auf seine Daten wird nur über seine Schnittstellen zugegriffen. Durch die Festlegung der Schnittstelle ist die Zugriffstransparenz gegeben. Die Replikation der Daten wird durch den Auslastungsdienst erbracht und ist somit von außerhalb auch transparent.

Damit innerhalb des Auslastungsdienstes die Transparenzen erreicht werden, muss dieser jedoch für die Replikation der Daten sorgen und selbst replizierbar sein, um Ausfälle oder Lastspitzen zu überstehen.

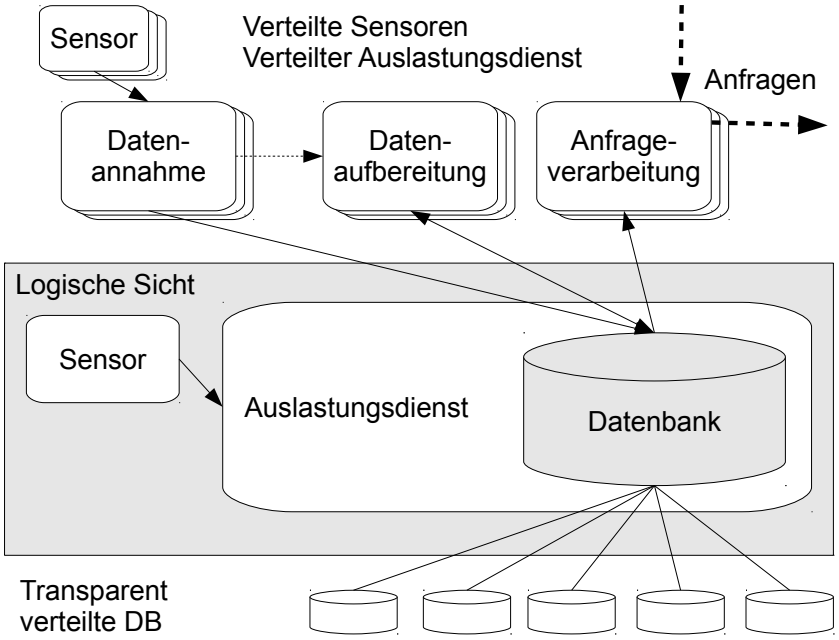


Abbildung 3.7: Übersicht Replikation des Auslastungsdienstes

Abbildung 3.7 stellt den replizierten Auslastungsdienst und die logische Sicht auf ihn dar. Der bisher monolithisch dargestellte Auslastungsdienst wird in drei Komponenten zerlegt. Jede der Komponenten kann je nach nach Bedarf mehrfach repliziert werden. Zur

Synchronisation der einzelnen Komponenten miteinander und zur Datenhaltung dient eine Datenbank, die wiederum logisch eine einzelne Komponente darstellt, jedoch als verteilte, replizierte Datenbank implementiert ist.

Der Ablauf im Auslastungsdienst stellt sich nun wie folgt dar: Sensoren verbinden sich mit einem (durch die Dienstschnittstelle von NexusDS bestimmten) Datenannahme-Auslastungsdienst. Sollten die verfügbaren Datenannahmedienste überlastet sein, werden weitere gestartet. Der Datenannahmedienst speichert die Daten zwischen und informiert einen Datenaufbereitungsdienst darüber, dass Daten verfügbar sind und in die Datenbank eingepflegt werden können.

Der Datenaufbereitungsdienst erzeugt die in Abschnitt 3.2 bereits vorgestellten Statistiken und Auswertungen. Je nach Berechnungsaufwand kann auch der Datenaufbereitungsdienst repliziert werden, solange die Berechnungen getrennt durchführbar sind. Die Ergebnisse der Berechnungen werden in der Datenbank gespeichert und sind somit unabhängig von bestimmten Instanzen des Datenaufbereitungsdienstes verfügbar. Bei besonders aufwendigen Berechnungen, wie dem Knotenleistungsclustering, kann der Datenaufbereitungsdienst asynchron zu sämtlichen anderen Operationen arbeiten und ein neues Clustering berechnen, das ein vorhandenes Clustering in der Datenbank erst bei Fertigstellung ersetzt.

Der Knotenanfrageverarbeitungsdienst verarbeitet die Knotenanfragen nach bestimmten Knoten, indem er die bereits aufbereiteten Daten aus der Datenbank abruft, filtert und die geeigneten Knoten berechnet.

Aus Sicht des verteilten Knotenanfragedienstes ist die Datenbank wiederum eine logische Einheit, die jedoch durch eine verteilte Datenbank bereitgestellt wird. Die Implementierung solcher verteilten Datenbanken würde jedoch den Rahmen dieser Arbeit sprengen. Hierfür sei auf die entsprechende Literatur verwiesen [Öz99, Tan06].

Kapitel 4

Detailentwurf

Nachdem in Kapitel 3 ein Überblick über die Architektur des gesamten Systems und besonders über den Auslastungsdienst und die Sensoren gegeben wurde, wird in diesem Kapitel auf die Details und genauen Verfahren der Sensoren und des Auslastungsdienstes eingegangen.

In Abschnitt 4.1 werden die von den Sensoren gesammelten Daten spezifiziert und in Abschnitt 4.2 ihre Speicherung.

Abschnitt 4.3 geht auf die Erzeugung und Speicherung der Kompatibilitätsmatrix ein. In Abschnitt 4.4 wird die Erzeugung der Knotenleistungscluster erläutert. Hierzu werden verschiedene Clusteringverfahren vorgestellt und bewertet. Schließlich wird der Aufwand für den verwendeten Algorithmus angegeben. Die Berechnung der knoten-/operatorabhängigen Anforderungen auf Basis des Knotenleistungsclusterings wird in Abschnitt 4.5 erläutert. Zuletzt wird in Abschnitt 4.6 die Durchführung des Optimierungsrankings beschrieben.

4.1 Spezifikation der Sensormessdaten

In diesem Abschnitt werden die von den Sensoren auf den Knoten und in den Operatoren gesammelten Daten genauer spezifiziert. Die Sensoren sammeln Daten aus zwei unterschiedlichen Quellen mit unterschiedlichem Fokus. Es werden Daten über den Knoten selber und seine Ressourcen und seine Leistungsfähigkeit gesammelt sowie Daten über die Operatoren, ihre Produktivität und ihren Ressourcenverbrauch. Zuerst wird die Datenerfassung der Knoten erläutert und dann die Datenerfassung der Operatoren.

	Parametername	Datentyp
Hardware	CPU-Architektur	*String*
	Maximaler CPU-Takt (in MHz)	*Float*
	Anzahl der CPUs	*Float*
	Arbeitsspeicher (in MiB)	*Float*
	GPU-Typ[ID]	*String*
	GPU-Speicher[ID] (in MiB)	*Float*
	Festspeicher (in MiB)	*Float*
Software	Betriebssystem	*String*
	Betriebssystem Variante	*String*
	Programm[ID]	*String*
	Programmversion[ID]	*String*

Tabelle 4.1: Erfasste Plattformeigenschaften: Parametername und Datentyp

4.1.1 Plattformeigenschaften – statische Knotendaten

Die Plattformeigenschaften beschreiben die statischen Soft- und Hardware-Eigenschaften des Rechenknotens. Sie werden zur Erkennung der Kompatibilität von Operatoren mit speziellen Anforderungen, zur Zusammenfassung ähnlicher Rechenknoten und zur Berechnung von Leistungsdaten verwendet.

Die Plattformeigenschaften sind statische Werte, die einmalig beim Start des Frameworks auf dem Knoten erfasst und an den Auslastungsdienst gemeldet werden. Die Plattformeigenschaften lassen sich nach ihrer Herkunft in zwei Kategorien, in Hardware- und Software-Eigenschaften, unterteilen. In Tabelle 4.1 werden zuerst die erfassten Hardware-Eigenschaften und dann die Software-Eigenschaften aufgelistet. Eigenschaften, die mehrfach auftreten können, wie zum Beispiel mehrere GPUs (Grafikprozessoren) oder verfügbare Softwarepakete, haben eine [ID] im Parameternamen, die die Mehrfachnennung ohne Namenskonflikt ermöglicht und gleichzeitig unterschiedliche zusammengehörende Eigenschaften verknüpft. So steht zum Beispiel in Programm[ID_a] der Name des Programms und Programmversion[ID_a] beschreibt die Version desselben Programms. Der Typ von [ID] ist *Float* und wird bei jeder Verwendung von Null an um eins inkrementiert.

4.1.2 Dynamische Knotendaten

Neben den Plattformeigenschaften sammelt der Sensor regelmäßig Daten über die aktuelle Auslastung des Knotens. Anhand der gemessenen Werte werden die folgenden

Parametername	Datentyp
CPU-Queue-Länge	*Float*
freier Arbeitsspeicher (in MiB)	*Float*
freier Festspeicher (in MiB)	*Float*
Netzwerkvolumen seit letzter Messung (in MiB)	*Float*
Zeit seit letzter Messung (in Sekunden)	*Float*

Tabelle 4.2: Erfasste Knotenleistungsmessdaten

Vorhersagen getroffen: Die Messung des Netzwerkvolumens gibt Auskunft über die aktuelle Auslastung und als Differenz zu einem historischen Maximum auch über die aktuell freie Netzwerkkapazität. Die aktuelle Arbeits- und Festspeicherauslastung erlaubt eine Vorhersage welche Speichermenge reserviert werden kann. Hingegen soll die Messung der CPU-Auslastung zum einen den Verbrauch der bereits laufenden Operatoren bestimmen als auch erlauben, die freien Rechenkapazitäten für weitere Operatoren vorherzusagen.

Für die CPU-Auslastungsmessungen existieren verschiedene Verfahren. Ferrari und Zhouh [FZ87] unterteilen sie in queuelängenbasierte (Anzahl der auf die CPU wartenden Prozesse) und auslastungsbasierte (% CPU-Auslastung) Verfahren. Anhand von empirischen Messungen kommen sie zum Schluss, dass die Messung der Queuelänge genauere Vorhersagen erlaubt. Die besten Ergebnisse werden in der Untersuchung mit einem 4-Sekunden-Durchschnitt von CPU+Speicher+Eingabe/Ausgabe-Queuelänge erzielt. Da diese Messwerte in normalen Systemen nicht verfügbar sind – Ferrari und Zhouh verwendeten einen speziell angepassten Kernel zur Messung – wird hier nur die aktuelle CPU-Queuelänge gemessen, die immer noch gute Ergebnisse erzielt. Die CPU-Queuelänge lässt sich sowohl bei aktuellen Windows Betriebssystemen als auch bei Unix und Linux Systemen vom Kernel auslesen.

Tabelle 4.2 listet die erfassten Werte nochmals auf und gibt die verwendeten Einheiten und Datenformate wieder. Die Zeitdifferenz seit der letzten Messung wird zur Normalisierung der Werte auf einheitliche Zeitabschnitte und zur Gewichtung der absoluten Werte in der Statistik verwendet. Bei einem Messwert für das Netzwerkvolumen von 10 MiB[1] seit der letzten Messung und einer Zeitdifferenz von 5 Sekunden wird also 2 MiB/s gespeichert.

4.1.3 Operatormessdaten

Für jeden Operator auf einem Knoten werden Leistungs- und Verbrauchsmessdaten erfasst, um die Leistungsfähigkeit des Operators im Vergleich zwischen verschiedenen

[1] Mebibyte: Entsprechend Standard IEEE 1541-2002 [IEE09]: 1 MiB = 2^{20} Byte

Parametername	Datentyp
Operator ID	*String*
Parametrisierung	*Struktur*
Produktivität	*Float*
Produktivitätskennzahlen ID	*String*
Latenz (Sekunden$*10^{-3}$)	*Float*
CPU-Zeit seit letzter Messung (Sekunden$*10^{-3}$)	*Float*
belegter Arbeitsspeicher (in MiB)	*Float*
belegter Festspeicher (in MiB)	*Float*
Netzwerkvolumen seit letzter Messung (in MiB)	*Float*
Zeit seit letzter Messung (in Sekunden)	*Float*
$\sum$Länge der Input Queues [a]	*Float*
$\sum$Länge der Output Queues [b]	*Float*
Operatorspezifische Messwerte[ID]	*Float*

Tabelle 4.3: Erfasste Operatormessdaten

[a] falls Input Queues im Operator vorhanden
[b] falls Output Queues im Operator vorhanden

Knoten beurteilen zu können und um daraus eine Vorhersage über die Leistungsfähigkeit und den Ressourcenverbrauch eines Operators auf einem Knoten treffen zu können.

Da die erbrachte Leistung eines Operators im Allgemeinen von seiner Parametrisierung abhängt, wird neben der Produktivitätskennzahl, wie in Abschnitt 3.4.1 vorgestellt, die Parametrisierung des Operators erfasst. Zusätzlich zu den Produktivitätskennzahlen werden die vom Operatorprozess verbrauchte CPU-Zeit, der von ihm belegte Arbeits- und Festspeicher sowie das Netzwerkvolumen seit der letzten Messung erfasst.

Die Operatoren in NexusDS können über Ein- und Ausgabequeues verfügen. Die Queues erlauben es, Daten als nicht blockierende Operation von einem Operator an einen anderen weiterzugeben. Sofern der Operator über Ein- oder Ausgabequeues verfügt, wird die Länge dieser Warteschlangen gespeichert. Anhand der Daten über die Ein- und Ausgabequeuelängen kann der Scheduler eine Überlastung beziehungsweise einen Flaschenhals erkennen und durch Rescheduling beheben. So ist zum Beispiel eine durchgehend vollständig gefüllte Eingabequeue bei gleichzeitig leerer Ausgabequeue ein Hinweis, dass der Operator mit der Verarbeitung der Eingabedaten überfordert ist.

Die erfasste Zeitspanne seit der letzten Messung dient wiederum der Normalisierung und Gewichtung der Messdaten. Alle erfassten Operatormessdaten sind in Tabelle 4.3 zusammengefasst.

4.2 Speicherung der Sensordaten

Die von den Sensoren gesammelten Daten (siehe vorheriger Abschnitt) werden über die Sensordatenstromschnittstelle an den Auslastungsdienst übertragen und hier aufbereitet und gespeichert, um bei Anfragen nach geeigneten Knoten Auswertungen über diese Daten durchführen zu können.

Die Sensoren übertragen regelmäßig ihre aktuellen Daten. Bei einer großen Anzahl von Knoten und damit Sensoren würde die vollständige Speicherung hohe Ansprüche an die Speicherkapazität und die Rechenleistung des Auslastungsdienstes stellen. Zum einen würden die Daten viel Speicherplatz benötigen und zum anderen würden Berechnungen, die statistische Daten verwenden, große Datenmengen zu verarbeiten haben. Daher werden die Sensormessdaten gefiltert und nach Möglichkeit verdichtet.

Im folgenden Abschnitt wird angegeben, welche Daten in welcher Verdichtung gespeichert werden und über welche Zugriffspfade auf sie zugegriffen wird.

Im zweiten Abschnitt wird dann das Verfahren zur Historisierung, also zum Aussortieren und Vergessen nicht mehr relevanter Daten vorgestellt.

4.2.1 Gespeicherte Daten und Zugriffspfade

Drei Auswertungen – Filterung nach Plattformanforderungen, Berechnung der operatorabhängigen Anforderungen und Berechnung der knoten-/operatorabhängigen Anforderungen – verwenden die Daten der Sensoren. Sie haben jeweils unterschiedliche Anforderungen an die Datengranularität. Je nach Auswertung werden unterschiedliche Zugriffsstrukturen auf teilweise gleiche Daten benötigt, um die gesuchten Daten effizient aufzufinden.

In den folgenden Abschnitten werden zuerst die für die jeweiligen Auswertungen benötigten Daten aufgeführt und die benötigten Zugriffsstrukturen angegeben.

Filterung nach Plattformanforderungen: Für die Filterung nach Plattformanforderungen wird die Kompatibilitätsmatrix, wie bereits in Abschnitt 3.3.1 vorgestellt, verwendet. Für die Berechnung der Kompatibilitätsmatrix werden die statischen Plattformeigenschaften verwendet. Sie müssen daher dementsprechend vorgehalten werden. In der Übersichtstabelle 4.4 sind dies die Daten der ersten Zeile. Der Zugriff auf die Daten erfolgt immer über eine KnotenID, die hier auch als Index verwendet wird.

Operatorabhängige Anforderungen: Für die operatorabhängigen Anforderungen werden die Daten über den Ressourcenverbrauch und die Leistung der Operatoren abhängig von ihrer Parametrisierung benötigt. Einzelne Operatorausführungen

	Daten	Granularität	Indexe
1	Plattformeigenschaften	statischer Datensatz	Knoten-ID
2	Operatormessdaten	n-Bucket-(Min, ∅, Max)	OP-ID, Konfiguration, Knoten-ID Bucket, OP-ID, Konfiguration
3	Knotenauslastung	n-Bucket-(Min, ∅, Max) und aktuelle Daten	Knoten-ID, Bucket

Tabelle 4.4: Zusammenfassung der gespeicherten Sensormessdaten, ihrer Granularität und der Indexe für den Datenzugriff. Unterstreichungen in der Spalte Indexe definieren jeweils einen kombinierten Index.

können starke Abweichungen aufgrund ihrer Eingangsdaten oder anderer Umgebungseinflüsse wie Knotenauslastung haben. Diese Ausreißer sind im einzelnen nicht interessant. Für eine Vorhersage der benötigten Ressourcen und Leistung reichen Minimal-, Durchschnitts- und Maximalwert des Ressourcenverbrauchs und der Leistung. Die Ressourcenverbrauchsdaten der Operatoren sind in Tabelle 4.4 in der zweiten Zeile zu finden. Der Zugriff erfolgt über die Kombination von OperatorID und seiner Konfiguration. Es wird also ein kombinierter Index über OperatorID und Konfiguration benötigt.

Knoten-/operatorabhängige Anforderungen: Zur Berechnung der knoten-/operatorabhängigen Anforderungen wird das Knotenleistungsclustering verwendet. Für die Berechnung der knoten-/operatorabhängigen Anforderungen anhand des Knotenleistungsclusterings werden Daten über den Ressourcenverbrauch und die Leistung der Operatoren wie bei den operatorabhängigen Anforderungen benötigt (Tabelle 4.4 Zeile 2). Die operatorabhängigen Anforderungen werden über die Kombination von OperatorID und Operatorkonfiguration angesprochen. Zusätzlich werden in dieser Auswertung Daten über die Knotenauslastung hinzugezogen, um festzustellen, ob der Knoten ausreichend freie Kapazität hat. Die Knotenauslastungsdaten bestehen sowohl aus statistischen Daten als auch aus den aktuellen Daten der Knoten. Da die Sensoren laufend Messungen durchführen, kann durch ein solches Zwischenspeichern der aktuellen Auslastungsdaten verhindert werden, dass alle Knoten, die bei einer Anfrage betrachtet werden, nach ihrer aktuellen Auslastung abgefragt werden müssen. Die Knotenauslastungsdaten sind in der Übersichtstabelle 4.4 in Zeile 3 dargestellt. Sie werden anhand eines Indexes über die KnotenID angesprochen.

Für die Erzeugung des Knotenleistungsclusterings als Hilfsstruktur für die knoten-/operatorabhängigen Anforderungen werden die statischen Plattformeigenschaften sowie Informationen über die Berechnungsleistung der Knoten benötigt. Die Berechnungsleistung eines Knotens ergibt sich aus den Produktivitäten der Operatoren auf diesem Knoten. Da verschiedene Konfigurationen von Operatoren im

Allgemeinen nicht vergleichbar sind, wird für jeden Knoten ein Satz Durchschnittswerte pro Operator-Konfigurations-Tupel benötigt. Die exakten Daten einzelner Operatorausführungen müssen hierfür nicht vorgehalten werden. Alle hierfür benötigten Daten wurden schon durch vorhergehende Auswertungen gefordert. Sie sind in der Tabelle 4.4 in Zeile 1 und 2 zu finden. Als Zugriffsstruktur kommt jedoch der Zugriff über die KnotenID hinzu.

4.2.2 Zeitgranularität und Historisierung

Da sich das System über eine längere Laufzeit verändern kann, zum Beispiel beim Wechsel eines Start- und Testbetriebes in einen regulären Betrieb oder durch die Installation neuer Operatoren und Anwendungen, ist es nicht sinnvoll, die historischen Daten beliebig lange zu speichern. Alte Daten, die schon lange keinen Bezug mehr zum aktuellen System haben, würden in den Statistiken nach wie vor Einfluss ausüben.

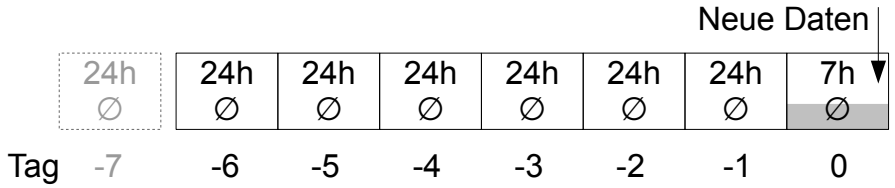


Abbildung 4.1: Beispiel Bucket Statistik für 7 *x* 24 Stunden; das aktuelle Bucket enthält hier erst Daten von 7 Stunden.

Es ist wünschenswert, die statistischen Daten in aggregierter Form vorzuhalten, um Speicherplatz zu sparen, jedoch so, dass Daten, die eine bestimmtes Alter erreicht haben, gezielt entfernt werden können. Bei einer einfachen Berechnung von Durchschnittswerten wäre es unmöglich, alte Werte wieder herauszurechnen, ohne die Originaldaten dafür vorzuhalten. Stattdessen wird ein Bucket-Verfahren angewandt, indem Daten eines Zeitabschnittes gemeinsam verdichtet werden. Als Beispielzeitraum sei die Statistikerhaltung für eine Woche angenommen und die Bucketgröße von 24 Stunden. Alle innerhalb von 24 Stunden anfallenden Daten werden in einem Bucket verdichtet (in diesem Fall *Minimum*, *Durchschnitt*, *Maximum* pro Datensatz). Buckets, die älter als eine Woche sind, werden verworfen. Der Zugriff auf die Statistik erfolgt durch Durchschnittsbildung über alle Buckets, wobei das aktuelle Bucket mit `(Alter in Stunden)/24` gewichtet wird. Abbildung 4.1 verdeutlicht dies nochmals graphisch.

Im aktuellen System sind die Größen der Buckets und die Länge der Historie an die Gegebenheiten anzupassen. Sie sind abhängig von der Veränderungsrate des Systems,

also davon, ob das System einen relativ statischen Zustand hat oder häufig wechselnde Anwendungen ausführt. Ebenso ist die Laufzeit von Operatoren zu beachten, da es nicht sinnvoll ist, bei lange laufenden Operationen bereits während ihrer Ausführung die Messdaten zu verwerfen. Bei einem nahezu statischen System kann ein sehr großer Zeitraum für die Historie gewählt werden, während bei einen dynamischen System eher kürzere Zeiträume erfolgversprechend sind.

Die Daten der einzelnen Buckets werden, wie in Tabelle 4.4 auf Seite 54 gezeigt, in einem Datensatz pro Bucket gespeichert. Für jeden Knoten werden also entsprechend der Bucketanzahl Datensätze gespeichert. Ebenso wird für jedes Operator-Konfigurations-Tupel ein Datensatz pro Bucket gespeichert.

Um den Wert des aktuellen Buckets zu bestimmen, gibt es zwei Möglichkeiten. Zum einen kann bei Eintreffen neuer Daten der Wert des Buckets mit der Anzahl der Datensätze mit dem neuen Wert als Durchschnitt verrechnet werden. Zum anderen können alle Datensätze des Bucketzeitraumes gespeichert werden und jeweils ein Durchschnitt über sie gebildet werden und dieser als Wert des aktuellen Buckets gespeichert werden. Der erste Ansatz hat den Vorteil, dass sein Speicherverbrauch sehr gering ist. Es werden nur der ohnehin gespeicherte Wert des aktuellen Buckets, die Anzahl der bisher gespeicherten Datensätze sowie der neu zu integrierende Datensatz benötigt. Beim zweiten Ansatz müssen hingegen alle Datensätze, die bereits in das aktuelle Bucket eingegangen sind, erhalten bleiben, um über sie einen neuen Durchschnitt bilden zu können. Aufgrund der Speicherplatzersparnis wird dementsprechend der erste Ansatz verwendet. Es wird also für das aktuelle Bucket immer ein Zähler für den Messzeitraum den es bereits enthält, mitgeführt.

4.3 Erzeugung und Speicherung der Kompatibilitätsmatrix

Die Erstellung der in Abschnitt 3.3.1 eingeführten Kompatibilitätsmatrix erfordert ein Abgleichen der Plattformeigenschaften der Knoten mit den Plattformanforderungen der Operatoren.

Beim Start des Auslastungsdienstes ruft dieser die Plattformanforderungen für alle Operatoren aus dem Operatorrepository ab und beantragt dort eine Benachrichtigung, um bei Installation neuer Operatoren informiert zu werden. Die Plattformanforderungen der Operatoren werden lokal im Auslastungsdienst zwischengespeichert, um einen effizienten Zugriff auf sie zu ermöglichen.

Die Plattformeigenschaften eines Knotens werden beim Start des Knotens einmalig im Sensordatenstrom an den Auslastungsdienst übertragen und im Auslastungsdienst gespeichert.

String:	KnotenID	Eigenschaftsname	Eigenschaftswert
	K1	CPU-Architektur	x86
	K2	CPU-Architektur	amd64

Float:	KnotenID	Eigenschaftsname	Eigenschaftswert
	K1	Maximaler CPU-Takt	2400
	K2	Maximaler CPU-Takt	1800

Tabelle 4.5: Beispieltabellen für *String* und *Float* Plattformeigenschaften

String:	OperatorID	Eigenschaftsname	Eigenschaftswert	Vergleich
	O1	CPU-Architektur	x86	$=$
	O2	CPU-Architektur	sparc	$\neq$

Float:	OperatorID	Eigenschaftsname	Eigenschaftswert	Vergleich
	O1	Maximaler CPU-Takt	2000	$>$

Tabelle 4.6: Beispieltabellen für *String* und *Float* Plattformanforderungen

Die Datenhaltung für die Plattformeigenschaften besteht aus einer Tabelle pro Datentyp, die jeweils die Spalten KnotenID, Eigenschaftsname und Eigenschaftswert hat. Da die Plattformeigenschaften nur zwei Datentypen enthalten können, *String* und *Float*, sind sie durch zwei Tabellen beschrieben. Die Beispieltabellen 4.5 und 4.6 enthalten jeweils zwei Operatoren mit unterschiedlicher CPU und Taktrate.

Die Plattformanforderungen der Operatoren werden in ähnlicher Form gespeichert, jedoch enthalten beide Tabellen eine zusätzliche Spalte, die einen Vergleich spezifiziert. Der Vergleich kann bei Anforderungen vom Typ *String* $=$ oder $\neq$ sein und bei *Float* Anforderungen $=, <, >$. Der Operator O1 in der Tabelle 4.6 erfordert eine x86 Architektur und einen CPU-Takt, der größer als 2400 Mhz ist.

OperatorID	KnotenID	Kompatibel
O1	K1	True
O1	K2	False
O2	K1	True
O2	K2	True

Tabelle 4.7: Beispiel für die Speicherung einer Kompatibilitätsmatrix für die Operatoren O1, O2 und die Knoten K1, K2

Sobald die Plattformeigenschaften eines Knotens gespeichert wurden, wird für jeden in den Tabellen mit Operatoranforderungen enthaltenen Operator überprüft, ob der Knoten zu ihm kompatibel ist, also die Anforderungen erfüllt. Hierzu wird zuerst überprüft, ob der Eigenschaftsname jeder Anforderung in den Eigenschaften des Operators vorkommt und ob für diese Paare aus Plattformanforderung PA und Operatoreigenschaft OE gilt:
`(OE.Eigenschaftswert PA.Vergleich PA.Eigenschaftswert) = True`

Das Ergebnis dieser Überprüfungen ist eine Matrix, die in einer Tabelle der Form von Tabelle 4.7 gespeichert wird. Jedem Knoten-/Operatorpaar wird ein Boolescher Wert zugeordnet, der angibt, ob das Paar kompatibel ist. Über eine Anfrage wie in Listing 4.1 können alle zu einem Operator `<AnfrageOperatorID>` kompatiblen Knoten abgefragt werden.

Listing 4.1 SQL Beispiel zur Abfrage kompatibler Knoten für `<AnfrageOperatorID>`

```
SELECT KnotenID FROM Kompatibilittsmatrix
       WHERE  OperatorID = <AnfrageOperatorID>
       AND    Kompatibel = TRUE;
```

4.4 Erzeugung des Knotenleistungsclusterings

Wie bereits in Abschnitt 3.5 eingeführt, dient das Knotenleistungsclustering dazu, Knoten zu finden, deren Rechenleistung vergleichbar ist, um die statistische Datenbasis für Operatormessdaten und damit die Leistungsvorhersagen zu verbessern, indem die Daten vergleichbarer Knoten zusammengeführt werden.

Damit Knoten für die Ausführung von Operatoren als vergleichbar gelten, werden hier drei Voraussetzungen definiert:

Voraussetzung 1: gleiche CPU-Architektur und gleiches Betriebssystem

Voraussetzung 2: ähnliche Plattformeigenschaften

Voraussetzung 3: ähnliche Rechenleistung

Nach den Voraussetzungen vergleichbare Knoten werden durch einen Clustering-Algorithmus gesucht, der anhand einer Distanzmetrik die Cluster berechnet. Die Einteilung nach Voraussetzung 1 erfolgt über einen einfachen Vergleich der Eigenschaften. Bei Übereinstimmung alle Eigenschaften sind die Knoten grundlegend vergleichbar, ansonsten nicht. Für die Eigenschaften 2 und 3 wird die Distanzmetrik gewichtet und dann anhand dieser geclustert. Die Gewichtung der Eigenschaften 2 und 3 wird zunächst auf jeweils 50 % festgelegt, ließe sich aber variieren, wenn entsprechende Messungen mit realen Daten eine Verbesserung der Erkennungsgenauigkeit versprechen. Durch die 50 %-Gewichtung

der Plattformeigenschaften zu den Rechenleistungsdaten wird dafür gesorgt, dass beide einen gleich großen Einfluss haben und dass nicht, wie im Falle ohne Gewichtung, eine große Differenz in der Anzahl der Datensätze für die jeweiligen Voraussetzungen den Einfluss der schwächeren verringert. Dies könnte dazu führen, dass eine Vielzahl an Ausführungsmesswerten eine geringe Anzahl an Plattformeigenschaften verdrängt und diese mit verschwindend geringem Anteil in die Distanz eingehen.

Nach [HK01] haben Clusteringverfahren im Allgemeinen zum Ziel, Punkte in einem mehrdimensionalen Raum, die gemeinsame Eigenschaften aufweisen, in Cluster einzuteilen, so dass die Punkte eines Clusters große Ähnlichkeit aufweisen und unterschiedlich zu den Punkten anderer Cluster sind.

Im Folgenden wird zuerst eine allgemeine Distanzmetrik für mehrdimensionale Punkte definiert, dann werden verschiedene Clusteringverfahren vorgestellt und gegeneinander abgewägt und schließlich wird der Algorithmus des gewählten Verfahrens angegeben.

4.4.1 Definition der Distanzmetrik

Als Distanzmetrik wird die „Distanzfunktion für Objekte unterschiedlicher Dimensionalität“ nach [HK01, S. 397], wie in Formel 4.1 dargestellt, verwendet.

Formel 4.1 Distanzfunktion für Objekte unterschiedlicher Dimensionalität

$$d(i,j) = \frac{\sum_{f=1}^{p} \delta_{ij}^{(f)} d_{ij}^{(f)}}{\sum_{f=1}^{p} \delta_{ij}^{(f)}}$$

Sie berechnet den Abstand $d(i,j)$ zweier Punkte i und j. Für Dimensionen beziehungsweise Plattformeigenschaften, die in einem Punkt vorhanden sind, im anderen jedoch nicht, gilt $\delta_{ij}^{(f)} = 0$, sonst $\delta_{ij}^{(f)} = 1$. Abhängig vom Typ der Dimension wird $d_{ij}^{(f)}$ unterschiedlich berechnet.

Für die hier verwendeten Datentypen *String* und *Float* ergibt sich $d_{ij}^{(f)}$ gemäß Formel 4.2 und 4.3.

Formel 4.2 Distanzfunktion für *String* Werte

$$d_{ij}^{(f)} = \begin{cases} 0, & \text{wenn } x_{if} = x_{jf} \\ 1, & \text{wenn } x_{if} \neq x_{jf} \end{cases}$$

Formel 4.3 Distanzfunktion für *Float* Werte[a]

$$d_{ij}^{(f)} = \frac{|x_{if} - x_{jf}|}{max - min}$$

[a] Gegenüber [HK01] leicht vereinfachte Notation

In den Formeln sind x_{if} und x_{jf} die Werte von i respektive j der Eigenschaft f mit max und min als Maximum und Minimum des Wertebereiches. Sie werden festgelegt, indem der höchste und niedrigste jemals betrachtete Wert für eine Eigenschaft gespeichert wird.

Nachdem die allgemeine Berechnung einer Distanzmetrik vorgestellt wurde, muss dies noch auf die drei Voraussetzungen in Formel 4.4 angewandt werden. Formel 4.4 beschreibt die Distanz $d_{ka,kb}$ zweier Knoten ka und kb. Voraussetzung 1 (d_1) ist ein hartes Ausschlusskriterium, während Voraussetzung 2 (d_2) und Voraussetzung 3 (d_3) gewichtet eingehen.

Dadurch, dass Voraussetzung 1 als hartes Kriterium verwendet wird, haben Knoten, die nicht die gleiche CPU-Architektur und das gleiche Betriebssystem haben, immer eine maximal Distanz. Es wird also abgebildet, dass diese Knoten nicht miteinander vergleichbar sind.

Die Berechnung von d_1, d_2 und d_3 erfolgt jeweils nach Formel 4.1.

Formel 4.4 Distanzfunktion für Clustering über

$$d_{ka,kb} = \max\left\{d_1, \left(\frac{1}{2}d_2 + \frac{1}{2}d_3\right)\right\}$$

4.4.2 Clusteringverfahren

Ein für das Clustering von Knoten mit ähnlicher Leistung geeignetes Verfahren muss mehrere Dinge leisten. Da Vorhersagen für alle Knoten eines Clusters anhand der integrierten Daten des Clusters getroffen werden, können Knoten, die extreme Werte haben, die Vorhersage stark verschlechtern. Zusätzlich besteht das Problem, dass die Daten des Systems nicht statisch sind, sondern die Leistungsdaten regelmäßig aktualisiert werden und somit auch das Clustering aktualisiert werden muss.

Die Anforderungen an das Clusteringverfahren sind daher

1. Einteilung in Cluster ähnlicher Leistungsfähigkeit

2. Genauigkeit
3. Geschwindigkeit

Die Anforderung 1, das Clustern von Objekten, die alle zueinander eine größere Ähnlichkeit haben als zu anderen Objekten, wird im Allgemeinen von Partitionierungsverfahren und von hierarchischen Verfahren geleistet.

Andere üblichen Verfahren wie dichtebasierte und modellbasierte Verfahren werden hier hingegen nicht weiter betrachtet, da sie für den Anwendungsfall keine Vorteile versprechen. Dichtebasierte Verfahren eignen sich besonders, um nichtkompakte Cluster zu finden, die beliebige Formen haben. Hier werden jedoch Cluster von Knoten gesucht, die alle einander sehr ähnlich sind, also räumliche Nähe aufweisen. Modellbasierte Verfahren setzen ein zuvor definiertes Modell voraus, dem die Daten angenähert werden. Ein solches Modell ist hier jedoch nicht vorhanden.

Zunächst werden Partitionierungsverfahren vorgestellt und dann hierarchische Verfahren, um daran ihre Eignung nach den Anforderungen 2 und 3 abwägen zu können.

Partitionierungsverfahren arbeiten, indem sie die Objektmenge in eine gegebene Anzahl k von Clustern aufteilen und durch Verschieben von Objekten zwischen den Clustern eine bessere Aufteilung suchen. Eine gute Partitionierung ist erreicht, wenn Objekte innerhalb einer Partition ähnlicher zueinander sind als zu Objekten anderer Partitionen.

Nachteile sind, dass heuristische Verfahren je nach Anfangsverteilung auf die Cluster zu unterschiedlichen Ergebnisclustern kommen können und dass die Anzahl der zu erzeugenden Cluster vorher festgelegt werden muss.

Als Beispiele für Partitionierungsverfahren werden der k-Means- und der k-Medoids-Algorithmus vorgestellt.

k-Means ist ein Clustering-Algorithmus, der als Eingabe eine Menge von Objekten und eine Anzahl k der Clustern bekommt. Durch Festlegung der Schwerpunkte für jedes der k Cluster durch ein zufällig gewähltes Objekt wird eine Anfangsverteilung geschaffen. In jedem weiteren Schritt werden den Clustern die ihrem Schwerpunkt nächsten Objekte zugeordnet und der Schwerpunkt der Cluster neu als Durchschnittswert alle enthaltenen Objekte berechnet. Die Schritte Zuordnung und Neuberechnung der Schwerpunkte werden wiederholt, bis entweder eine Iterationsgrenze erreicht ist oder ein Konvergenzkriterium, wie die mittlere quadratrische Abweichung in Formel 4.5, konvergiert. E ist die Summe der quadratischen Abweichung aller Objekte, p ist die Koordinate eines Objektes und m_i der Mittelpunkt des Clusters i.

Formel 4.5 Quadratische Abweichung [HK01, S. 402]

$$E = \sum_{i=1}^{k} \sum_{p \in C_i} |p - m_i|^2$$

k-Medoids Im Gegensatz zum k-Means-Verfahren werden in k-Medoids Cluster durch ein Medoid-Element, also ein reales Objekt des jeweiligen Clusters, statt durch den Durchschnittswert aller Elemente des Clusters repräsentiert. Als Konvergenzkriterium wird der absolute Fehlerwert in Formel 4.6 verwendet; o_j ist hierbei das Medoid.

Der k-Medoids-Algorithmus testet ausgehend von einer Anfangsverteilung und durch Zuteilen aller Objekte zum nächsten Medoid, ob das Ersetzen des Medoids eines Clusters durch ein anderes Objekt den absoluten Fehlerwert verringert. Die Ersetzungen werden entweder für eine festgelegte Anzahl an Iterationen fortgesetzt oder bis keine Ersetzungen mehr stattfinden, also der Fehlerwert stabil wird.

Formel 4.6 absolute Abweichung [HK01, S. 405]

$$E = \sum_{j=1}^{k} \sum_{p \in C_j} |p - o_j|$$

Der Hauptunterschied zwischen k-Means und k-Medoids ist ihre Sensibilität gegenüber Ausreißern. Objekte mit extremen Werten beeinflussen den Durchschnitt eines Clusters stark, während ihr Einfluss auf ein Medoid eher gering ist, da sie aufgrund ihrer extremen Werte nicht selbst zum Medoid werden. Die beiden Verfahren skalieren schlecht für größere Datenmengen. Häufig wird stattdessen ein Clustering für eine zufällige Untermenge der Daten, ein sogenanntes Sample, durchgeführt und der Rest der Daten den so gefunden Clustern zugeordnet.

Im Gegensatz zu den Partitionierungsverfahren arbeiten hierarchische Verfahren entweder zusammenfassend oder aufteilend statt mit einer festen Anzahl von Clustern. Die zusammenfassenden Verfahren fangen mit jedem Objekt in einem eigenen Cluster an und vereinigen sukzessive ähnliche Cluster, bis entweder nur noch ein Cluster übrig bleibt oder eine Abbruchbedingung greift. Die aufteilenden Verfahren starten mit einem einzigen Cluster, das alle Objekte enthält und in jedem Schritt aufgeteilt wird, bis alle Objekte in einzelnen Clustern sind oder wiederum eine Abbruchbedingung greift. Die beiden Verfahren werden in Abbildung 4.2 in Form eines Dendrogrammes verdeutlicht.

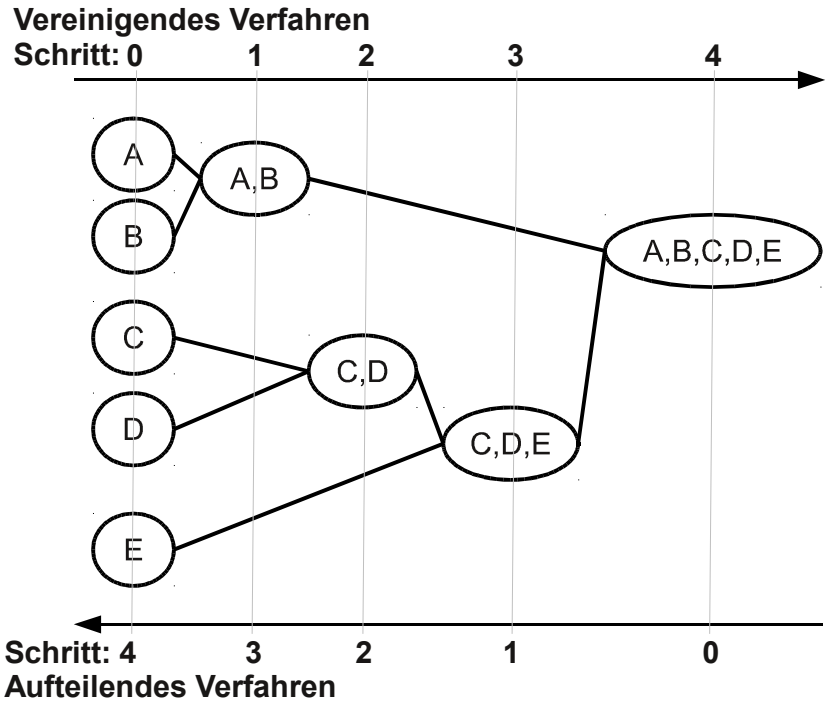


Abbildung 4.2: Beispiel Dendrogramm: Clustering von A, B, C, D, E für vereinigendes Verfahren (oben) und aufteilendes Verfahren (unten). Vergleiche [HK01]

4.4.3 Bewertung und Auswahl der Clusteringverfahren

Der Berechnungsaufwand bei hierarchischen Verfahren ist geringer als bei Partitionierungsverfahren, da einmal getroffene Vereinigungs- oder Aufteilungsentscheidungen nicht mehr revidiert werden können und daher in weiteren Schritten nicht mehr betrachtet werden müssen. Dies stellt jedoch auch den großen Nachteil dar, da nicht optimale Entscheidungen nicht mehr überarbeitet werden können. Der Nachteil der Partitionierungsverfahren ist, dass bei Start des Algorithmus die Anzahl der gewünschten Cluster bekannt sein oder aber der Algorithmus mehrfach für verschiedene Clusteranzahlen angewandt werden muss.

Im direkten Vergleich zwischen Partitionierungsverfahren und hierarchischen Clusteringverfahren ergibt sich Folgendes für die drei Anforderungen

Einteilung in Cluster ähnlicher Leistungsfähigkeit: Kann von beiden Verfahren geleistet werden; das hierarchische Verfahren ist jedoch flexibler bezüglich der Anzahl der Cluster und kann sich somit eher an aktuelle Anforderungen anpassen.

Genauigkeit: Hierarchische Verfahren unterliegen keiner Veränderung durch zufällige Anfangsverteilungen und sollten daher stabilere und genauere Ergebnisse erzeugen.

Geschwindigkeit: Hierarchische Verfahren müssen nach jedem Vereinigungsschritt weniger Elemente betrachten und ihre Laufzeit ist fest vorhersagbar, während Partitionierungsverfahren im Allgemeinen in jedem Schritt alle Elemente neu betrachten müssen und sehr lange Laufzeiten haben können, bis eine stabile Verteilung, vor allem bei großen Anzahlen von Elementen, erreicht ist.

Da alle drei Anforderungen für den Einsatz eines hierarchischen Verfahrens sprechen, wird ein solches verwendet. Das eingesetzte Verfahren ist ein vereinigendes, hierarchisches Verfahren, dass jeweils die Cluster mit dem geringsten Abstand der Schwerpunkte vereinigt. Durch dieses Vereinigungskriterium werden möglichst kompakte Cluster erzielt. Die so erzeugten Cluster entsprechen der Zielsetzung von Clustern, deren Objekte alle zueinander ähnlicher sind als zu den Objekten anderer Cluster.

Um eine möglichst hohe Flexibilität zu erhalten, wird die Clusterbildung nicht an einem bestimmten Punkt abgebrochen. Vielmehr werden alle Vereinigungsstufen durchgeführt und gespeichert, bis die Knoten vollständig vereinigt sind. Anhand der gespeicherten Clusteringlevel kann später eine beliebige Clustergranularität ausgewählt werden.

Zur Auswahl eines geeigneten Clusteringniveaus wird pro Ebene von der vollständig vereinigten Clusterung aus getestet, in wie vielen unterschiedlichen Clustern sich die in den vorherigen Schritten selektierten Knoten befinden, bis eine Ebene mit der gewünschten Granularität gefunden wurde. Die gewünschte Granularität kann von unterschiedlichen Faktoren abhängen. Da mit der Granularität und damit der Clustergröße direkt die Vorhersagequalität einhergeht, jedoch der Berechnungsaufwand für eine Vorhersage mit zu betrachtender Clusteranzahl steigt, kann so je nach verfügbarer Rechenleistung und gewünschter Vorhersagequalität ein Trade-Off getroffen werden. Zusätzlich kann auch Wissen über die Systemumgebung verwendet werden, wie zum Beispiel die Kenntnis, dass es unwahrscheinlich ist, mehr als 10 oder 100 unterschiedliche Rechnerkonfigurationen in der Menge der verfügbaren Knoten zu haben. Dementsprechend ist es nicht sinnvoll, für mehr als die Anzahl der unterschiedlichen verfügbaren Knotentypen eigene Vorhersagen zu berechnen.

4.4.4 Algorithmus

Vollständiges hierarchisches Clustern erfordert $n - 1$ Vereinigungen von Clustern [Has09, S. 523]. Für jede Vereinigung müssen die beiden Cluster mit dem geringsten Abstand bzw. der geringsten Unähnlichkeit gefunden werden. Nach [HK01] erfolgt dies im Allgemeinen durch Berechnung einer Distanzmatrix, die Ähnlichkeit oder Unähnlichkeit aller Cluster zueinander angibt. Da die Distanzen symmetrisch sind, also $\text{dist}(a, b) = \text{dist}(b, a)$, und

die Knoten zu sich selbst immer den Abstand 0 haben, ist nur der Teil unterhalb oder der Teil oberhalb der Diagonale der Matrix belegt. Die Diagonale selbst ist aufgrund der Symmetrie immer 0 und darf in der Distanzmatrix nicht belegt werden. Würden die Distanzen der Knoten zu sich selbst in der Matrix gespeichert, würde der Algorithmus zur Vereinigung der Knoten mit dem geringsten Abstand versuchen, Knoten mit sich selbst zu vereinigen.

In Abbildung 4.3 links ist eine solche Distanzmatrix für fünf Knoten zu sehen. Für n Knoten hat die Distanzmatrix $\sum_{j=1}^{n-1}(j)$ Einträge, entsprechend der Anzahl an Werten unter oder über der Diagonale.

$k1$	0	–	–	–	–
$k2$	d	0	–	–	–
$k3$	d	d	0	–	–
$k4$	d	d	d	0	–
$k5$	d	d	d	d	0
	$k1$	$k2$	$k3$	$k4$	$k5$

$k1$	0	–	–	–	–
$k2$	d	0	–	–	–
$k3$	d	d	0	–	–
$c1$	**d**	**d**	**d**	0	–
–	–	–	–	–	–
	$k1$	$k2$	$k3$	$c1$	–

Abbildung 4.3: Beispiel Distanzmatrix für die Knoten k1 - k5 und nach der Vereinigung von k4 und k5 zu c1

In jedem Vereinigungsschritt werden zwei Knoten beziehungsweise Cluster entfernt und durch ein neues Cluster ersetzt. Hieraus folgt, dass bei n Knoten nach $n-1$ Vereinigungen nur noch ein Cluster übrig bleibt. Bei einem einzigen Cluster ist die Distanzmatrix leer. Dementsprechend hat die Distanzmatrix nach s Schritten $\sum_{j=1}^{n-s}(j)$ Einträge, wenn die vollständige Matrix Schritt 1 entspricht. Abbildung 4.3 rechts zeigt eine Distanzmatrix nach dem ersten Vereinigungsschritt entsprechend $s = 2$.

Da die Vereinigung $n-1$ mal durchgeführt wird, ergibt sich die Gesamtanzahl der Vergleiche nach Formel 4.7.

Formel 4.7 Anzahl der Distanzen für alle Distanzmatrizen eines Clusterings für n Knoten

$$\text{Anzahl Distanzen} = \sum_{s=1}^{n-1} \sum_{j=1}^{n-s} j$$

Da in jedem Schritt s jedoch nur zwei Cluster verschmolzen werden, bleiben die Distanzen zwischen den anderen Clustern gleich und können direkt aus der ersten, vollständigen Distanzmatrix übernommen werden. Dementsprechend ist nur im ersten Schritt eine vollständige Distanzmatrix für n Knoten zu berechnen.

Für jeden Vereinigungsschritt sind dann die Distanzen des neu erzeugten Clusters zu allen anderen Knoten und Clustern, die in der aktuellen Matrix noch vorhanden sind, zu berechnen. Es müssen also $n-1$-mal die Distanzen eines neu erzeugten Clusters zu allen anderen Clustern berechnet werden. Da nach jedem Vereinigungsschritt die Größe der Distanzmatrix um 1 gesunken ist, müssen $n-1$-mal $n-s-1$ Distanzen neu berechnet werden, wobei s wieder den Vereinigungsschritt, beginnend bei 1 für die vollständige Distanzmatrix, angibt. Da s von 1 bis $n-1$ läuft, entspricht die Anzahl der benötigten Vergleiche Formel 4.8.

In Abbildung 4.3 entsprechen die in diesem Vereinigungsschritt neu berechneten Distanzen den fett gedruckten Distanzen der rechten Distanzmatrix.

Formel 4.8 Aufwand für vollständiges hierarchisches Clustern mit Wiederverwendung bereits gespeicherter Distanzen.

$$\text{Anzahl Distanzen} = \sum_{s=1}^{n-1} s + \sum_{s=1}^{n-2} s$$

4.4.5 Aufwandsreduzierung

Bei einem System, das laufend Änderungen erfährt, sei es durch neue und abgeschaltete Knoten oder durch aktuelle Leistungsmessdaten, ist eine ständige Aktualisierung – gleichbedeutend mit einer Neuberechnung der Cluster – sehr aufwendig. Eine Neuberechnung des gesamten Clusterings ist wiederum sehr aufwendig.

Wenn man sich die Formel 4.4 auf Seite 60 für die Distanzfunktion ansieht, stellt man fest, dass die Distanz bei unterschiedlicher CPU und unterschiedlichem Betriebssystem immer 1, also maximal ist. Diese Knoten werden also erst in den letzten Schritten des Clusterings vereinigt beziehungsweise müssen für unsere Auswertungen nie als gemeinsames Cluster betrachtet werden, da die Knoten sowieso als nicht vergleichbar gelten.

Teilclustering: Anstatt eine vollständige Distanzmatrix aufzubauen und zu vergleichen, werden unvergleichbare Knoten (nach Voraussetzung 1) fix auf den Wert `unvergleichbar` gesetzt und nicht weiter gemeinsam betrachtet. Da die Beziehung `unvergleichbar` transitiv ist, können vergleichbare Knoten unabhängig von allen Knoten, zu denen sie unvergleichbar sind, betrachtet werden. Dies entspricht einem getrennten Clustering für jede der disjunkten Mengen von kompatiblen Knoten.

Selektive Neuberechnung: Direkt aus dem Teilclustering folgt, dass bei Änderungen von Knoten eines der Teilcluster nicht alle Cluster neu berechnet werden müssen, sondern nur das Clustering für die nach Voraussetzung 1 kompatiblen Knoten.

Verzögerte Neuberechnung: Als weitere Optimierung des Aufwands wird eine an Verfahren zum Clustern großer Datenmengen angelehnte Technik verwendet. In Clustering-Algorithmen für Datenmengen, die zu groß sind, um sie geeignet mit einem Clustering-Algorithmus zu bearbeiten, der alle Elemente miteinander vergleicht, wird sogenanntes Sampling verwendet. Beim Sampling werden nicht alle Datenpunkte betrachtet, sondern zuerst nur eine möglichst repräsentative, häufig aber auch nur zufällig ausgewählte Teilmenge. Auf dieser Teilmenge wird nun ein üblicher Clustering-Algorithmus angewandt und es werden die so festgelegten Cluster verwendet, um ihnen die restlichen Datenpunkte zuzuordnen.

Um die Probleme mit geänderten oder neuen Knoten im Knotenleistungsclustering zu verringern, wird, statt bei jedem neuen oder geänderten Knoten das Clustering neu zu berechnen, das bisherige Cluster als fest betrachtet und die neuen oder geänderten Knoten werden den vorhandenen Clustern nach Ähnlichkeit hinzugefügt.

Das Hinzufügen zu den vorhandenen Clustern erfolgt von der Wurzel aus, indem für den neuen oder geänderten Knoten jeweils entschieden wird, ob der rechte oder der linke Untercluster ähnlicher ist. Da der Baum die maximale Höhe $n-1$ hat, müssen so bei zwei Unterclustern pro Ebene maximal $2(n-1)$ Vergleiche zwischen Knoten und Clustern zum Einfügen durchgeführt werden.

In regelmäßigen Abständen – wenn die Systemauslastung dies zulässt – werden die Cluster neu berechnet. Hierbei gehen dann die seit der letzten Clusterbildung neu hinzugekommen Knoten und seither veränderten Auslastungsdaten in die Berechnung der Cluster mit ein.

Um den Algorithmus inklusive der Aufwandsreduzierung durch Teilclustering zu verdeutlichen, ist er in Listing 4.2 in mengenorientiertem Pseudo-Code angegeben.

4.5 Berechnung der knoten- und operatorabhängigen Anforderungen

Die bereits in Abschnitt 3.5.1 vorgestellte Berechnung der knoten- und operatorabhängigen Anforderungen wird durchgeführt, indem die Knoten anhand des Knotenleistungsclusterings in Cluster mit Knoten ähnlicher Leistungsfähigkeit gruppiert werden. Für jedes

Listing 4.2 Algorithmus zum hierarchischen Clustern

Definitionen:

K Menge der Knoten

K_c $\subset K, \forall k_x, k_y \in K_c, k_x \neq k_y : k_x \cap k_y = \emptyset$
Disjunkte Teilmengen von K von miteinander vergleichbaren Knoten, entsprechend der Aufwandsreduzierung durch Teilclustering

```
for all K_c do          // Für jede Teilmengen K_c miteinander vergleichbarer Knoten
    cluster(K_c)        // das Clustering durchführen
end for
function CLUSTER(k: knoten)
    while |k| > 1 do                          // Bis alle Cluster vereinigt wurden
        c_1, c_2 = min{dist(k)}               // Cluster minimaler Distanz suchen
        c_m = merge(c_1, c_2)                 // Cluster minimalen Abstands vereinigen
        k = k + c_m                           // Neues Cluster hinzufügen
        k = k\{c_1, c_2}                      // Alte, vereinigte Cluster entfernen
        saveTree(c_m = {c_1, c_2})            // Update des Clustering Baumes
    end while
end function
```

dieser Cluster von Knoten ähnlicher Leistungsfähigkeit werden die statistischen Daten über Operatorausführungen zusammengeführt.

Abbildung 4.4 verdeutlicht das Vorgehen. Auf der linken Seite ist der Teil eines Clusterings für die Knoten A - G zu sehen. Das hier verwendete vereinigende Clusteringverfahren fängt mit jedem Knoten in einem eigenen Cluster an und vereinigt in jedem Schritt zwei der Cluster. In der Abbildung wurde die Clusterbildung bis zur Stufe vier durchgeführt. Diese wird hier auch für die Auswertung verwendet. Für jedes der gefundenen Cluster werden die statistischen Daten für die Ausführung des Operators aus der Datenbank, die auf der rechten Seite der Abbildung zu sehen ist, abgerufen. Wie in der Mitte der Grafik zu sehen ist, werden die Daten pro Cluster zusammengeführt und zu einem gemeinsamen Wertesatz für das Cluster ausgewertet. Hierbei wird ein Gesamtdurchschnitt über die Durchschnittswerte der einzelnen Knoten gebildet und jeweils ein Minimum und ein Maximum über alle Minima und Maxima des Clusters gebildet.

Unter der Annahme, dass die statistischen Daten über die Operatorausführungen gemäß dem Knotenleistungsclustering zwischen ähnlichen Knoten übertragbar sind, werden die aggregierten Datensätze für die Berechnung der impliziten knoten-/operatorabhängigen Anforderungen verwendet. Alle Knoten, die nach ihren aktuellen Auslastungsdaten nicht genügend freie Ressourcen verfügbar haben, um den Minimalanforderungen ihres Clusters für die Ausführung des Operators unter seiner Konfiguration zu entsprechen,

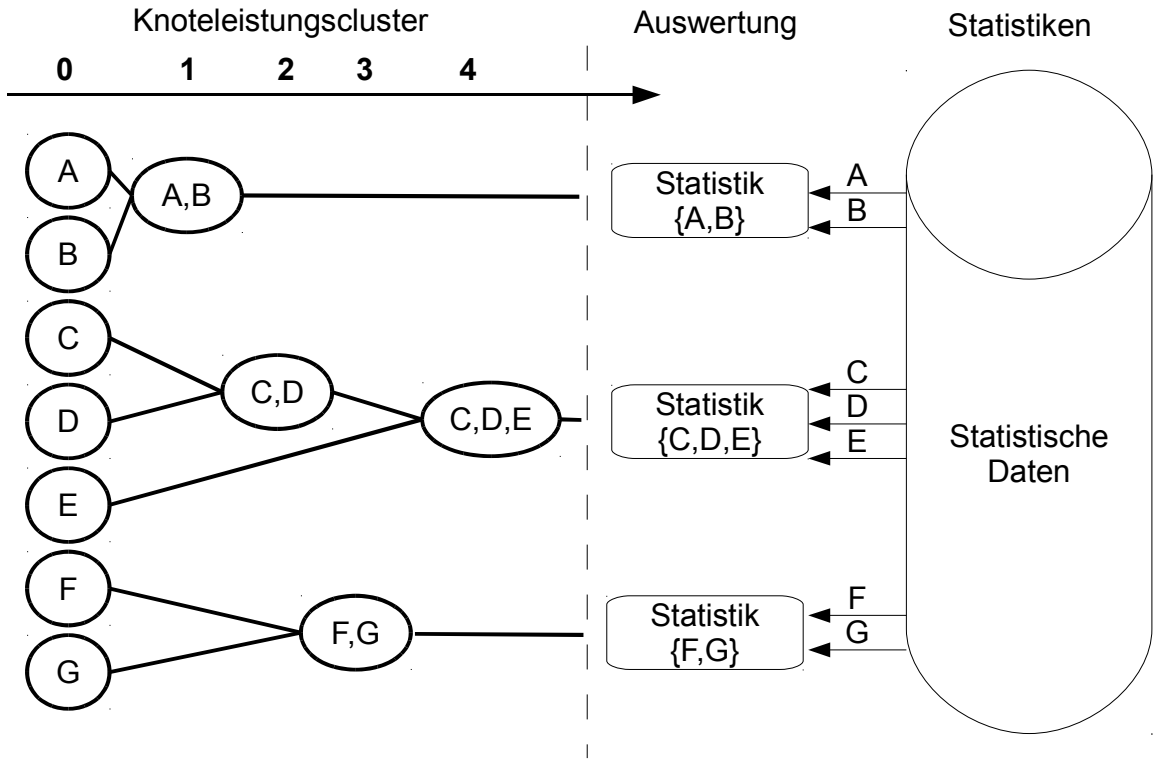


Abbildung 4.4: Veranschaulichung der Verwendung der Knotenleistungscluster zur Berechnung der knoten-/operatorabhängigen Anforderungen

werden als inkompatibel markiert. Sollte die Anzahl der in der Anfrage angeforderten Knoten durch die Anwendung der impliziten Anforderungen unterschritten werden, wird dieser Filterungsschritt verworfen.

Die Cluster **Minimal-**, **Durchschnitts-** und **Maximal-**Werte werden an die jeweiligen Datensätze der Knoten angehängt und erlauben es, auf höherer Architekturebene weitere Bewertungen vorzunehmen.

4.6 Optimierungsranking

Das Optimierungsranking erfolgt, indem die nach Anwendung der vorherigen Schritte verbleibenden als geeignet bewerteten Knoten anhand der Optimierungsreihenfolge sortiert werden.

Die Optimimierungsreihenfolge ist eine geordnete Liste von Parametern und ihrer Sortierungsrichtung. Als Beispiel hierfür ergibt die Optimierungsreihenfolge ,,`(CPU-Takt,`

`Maximieren), (RAM, Maximieren), (Latenz, Minimieren)`“, also eine absteigende Sortierung nach CPU-Takt. Alle Knoten, die den gleichen CPU-Takt haben, werden dann absteigend nach RAM sortiert. Für die Knoten mit gleichem CPU-Takt und gleicher RAM-Größe wird nochmals eine Sortierung, diesmal aufsteigend, nach Latenz angewendet.

4.7 Zusammenfassung und Visualisierung der Daten

In den Abschnitten Speicherung der Sensordaten, Abschnitt 4.2, Erzeugung und Speicherung der Kompatibilitätsmatrix, Abschnitt 4.3, und im Abschnitt Erzeugung des Knotenleistungsclusterings, Abschnitt 4.4, wurden für den jeweiligen Bereich relevante Daten erzeugt beziehungsweise gesammelt. In diesem Abschnitt werden alle bisher erbrachten Daten nochmals zur Übersicht erfasst und in einen Zusammenhang gestellt.

In Abbildung 4.5 sind die verschiedenen Daten und ihre Zusammenhänge in einem Entity-Relationship-Diagramm dargestellt. Zentraler Ausgangspunkt der Daten sind die Entitäten **Operator** und **Knoten**. Beide werden jeweils über eine eindeutige ID, die **Operator ID** beziehungsweise die **Knoten ID**, definiert. Ein Operator hat null oder mehrere **Konfigurationen**. Die Konfigurationen sind über ihren Operator und ihr Preset definiert. Jeder Operator hat null oder mehr **Plattformanforderungen**, die jeweils über einen Schlüssel, einen Vergleich und einen Wert verfügen. Ebenso hat ein Knoten null oder mehr **Plattformeigenschaften**, die jedoch nur aus Schlüssel und Wert bestehen.

Aus den Plattformanforderungen der Operatoren und den Plattformeigenschaften entsteht unter der Anwendung des Vergleichs die Relation **kompatibel**. Sie verbindet jeweils einen Operator mit einem Knoten, sofern diese zueinander kompatibel sind. Die Berechnung dieser Relation ist in Abschnitt 4.3 beschrieben. Im unteren Teil der Abbildung 4.5 befinden sie die Entitäten **Knotenmessdaten** und **Operatormessdaten**.

Die **Knotenmessdaten** sind durch einen **Knoten**, einen **Messwert** und ihr **Bucket** definiert. Sie enthalten jeweils ihr Gewicht, den Minimal-, den Maximal-, den Durchschnittswert und die Varianz. Die Operatormessdaten sind durch einen Operator, eine Konfiguration, einen Messwert, einen Knoten und ihr Bucket definiert.

Die Entität **letzter Wert** ist durch einen Knoten und einen Messwert definiert. Als Attribut trägt sie den Wert der letzten Messung des Messwertes für den Knoten.

Grau markiert auf der rechten Seite der Abbildung sind die Ergebnisse des Knotenleistungsclusterings zu sehen. Ein **Cluster**, definiert durch eine Cluster ID, enthält eine Menge der Knoten. Jedes Cluster hat ein Clustering-Level.

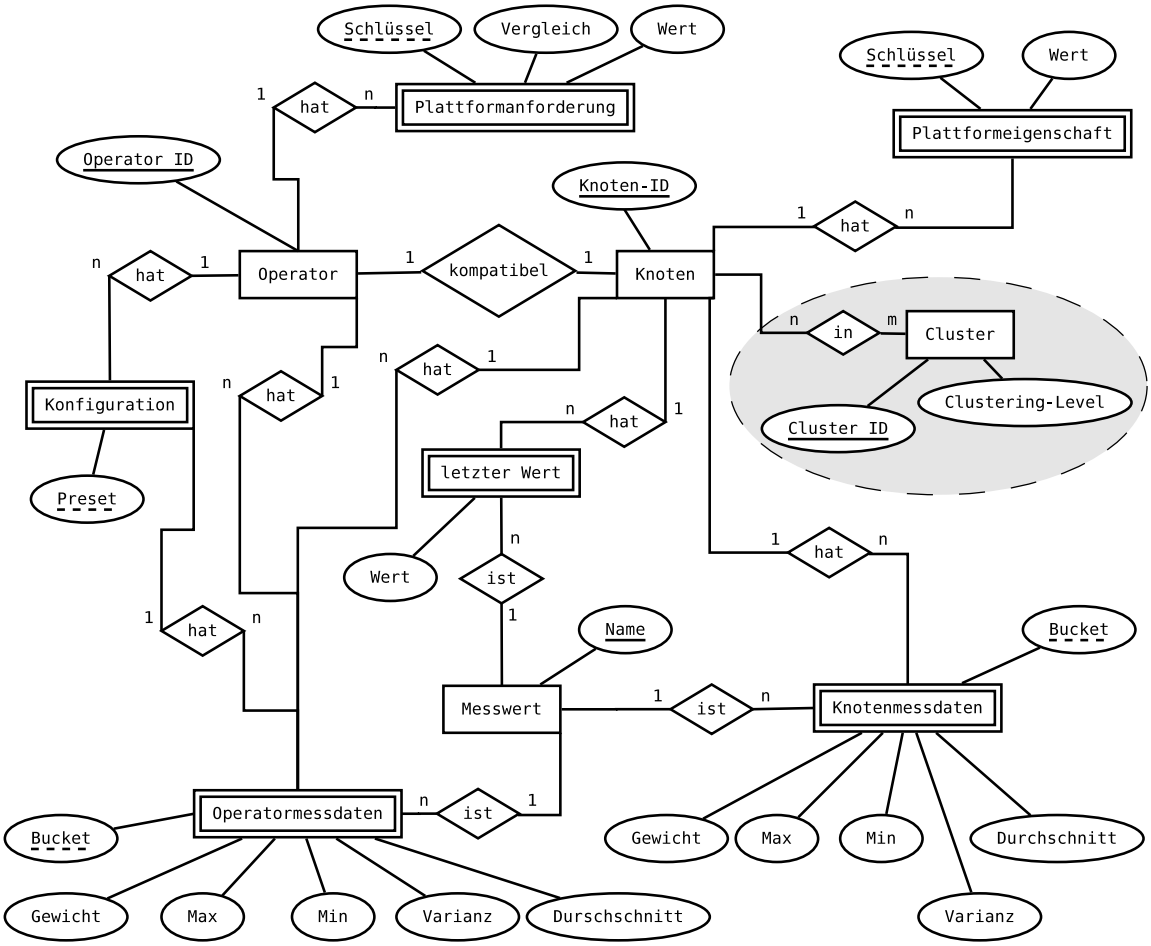


Abbildung 4.5: Übersicht über die gespeicherten Daten als ER-Diagramm

Kapitel 5

Implementierung

In diesem Kapitel werden Teile der Implementierung vorgestellt. Nachdem die vorherigen Kapitel 3 und 4 die Architektur und die verwendeten Strukturen und Algorithmen erläutert haben, wird in diesem Kapitel die tatsächliche Umsetzung besprochen. Auch in der Implementierung sind einige Designentscheidungen zu treffen, die direkten Einfluss auf die Effizienz und Wiederverwendbarkeit haben.

Die Realisierung des Auslastungsdienstes erfolgt in Java, da so ohne Schnittstellenprobleme die von NexusDS bereitgestellten Schnittstellen benutzt werden können. Da Java eine plattformunabhängige Programmiersprache ist, kann dieselbe Implementierung auf verschiedenen Plattformen verwendet werden, ohne dass Anpassungen erfolgen müssen oder die Programme entsprechend neu kompiliert werden müssen. Somit ist auch die grundlegende Anforderung an den Auslastungsdienst nach Plattformunabhängigkeit erfüllt.

In den folgenden Abschnitten sollen einige besonders interessante Konzepte der Implementierung hervorgehoben und einige Entwicklungsentscheidungen erläutert sowie verwendete externe Programme, Bibliotheken und Hilfsmittel vorgestellt werden.

Zuerst wird die Datenspeicherung für die Implementierung vorgestellt, dann wird im zweiten Abschnitt auf die Implementierung des Auslastungsdienstes eingegangen.

In Abschnitt 5.3 wird die Implementierung der Sensoren vorgestellt.

In Abschnitt 5.4 wird die Erfassung der Messwerte auf den Knoten erläutert und die hierfür verwendeten plattformabhängigen Verfahren erklärt.

Im letzten Abschnitt werden zwei Implementierungen für das Knotenleistungsclustering vorgestellt. Eine Implementierung wird lokal auf einem Knoten berechnet, während die andere Implementierung das Map/Reduce-Paradigma verwendet, um eine verteilte Berechnung der Daten zu ermöglichen.

In vielen Abschnitten der Implementierung wird Bezug auf Datentypen von Java genommen. Dies sind insbesondere die Typen `Integer`, `Float`, `String`, `Map`, `Set`, `Collection` und `Vector`. Diese Typen werden hier kurz vorgestellt, um den Text übersichtlicher zu gestalten.

Der Typ `Integer` stellt eine Ganzzahl dar. Der Typ `Float` stellt eine Gleitkommazahl dar. Der Typ `String` stellt eine Zeichenfolge dar. Der Typ `Map<Typ1, Typ2>` enthält eine Menge von Schlüssel-Wert-Paaren mit dem Schlüsseln vom Typ `Typ1` und Werten vom Typ `Typ2`. Die Schlüssel sind hierbei eindeutig. Der Typ `Set<Typ1>` enthält unsortiert Objekte vom Typ `Typ1`, die jeweils nur einmal im `Set` enthalten sind. Der Typ `Collection` ist dem Typ `Set` sehr ähnlich, kann jedoch Objekte mehrfach enthalten. Der Typ `Vector<Typ1>` enthält geordnete Daten vom Typ `Typ1`, die mehrfach vorkommen können. `Vector` ist einem Array sehr ähnlich, kann jedoch zur Laufzeit vergrößert und verkleinert werden.

5.1 Datenspeicherung

Eine grundlegende Entscheidung für die Implementierung des Auslastungsdienstes ist die Speicherung und der Zugriff auf die verwendeten Daten. In Abschnitt 4.7 wurden die verwendeten Daten bereits in Form eines ER-Diagramms zusammengefasst.

Auf die **statistischen Daten** wird vom Auslastungsdienst häufig mit selektiven und durchschnittsbildenden Anfragen zugegriffen. Die Abfrage von Datensätzen über verschiedene – auch zusammengesetzte – Indexe aus einer großen Datenmenge wird gezielt von Datenbankmanagementsystemen unterstützt. Daher werden in der Implementierung die statistischen Daten in einer SQL-basierten Datenbank gespeichert.

Die **Knotenleistungsclusterbildung** stellt besondere Ansprüche an die Datenhaltung und Abfrage. Im Verlauf des Clusteringprozesses werden, wie in Abschnitt 4.4.4 gezeigt, $\sum_{s=1}^{n-1} s + \sum_{s=1}^{n-2} s$ Distanzen berechnet. Da jede berechnete Distanz als Eingabe die statischen und statistischen Daten von zwei Knoten hat, ergeben sich, wenn jede Distanzberechnung die Daten neu anfordert, $2 * (\sum_{s=1}^{n-1} s + \sum_{s=1}^{n-2} s)$ Datenbankanfragen. Zusammengefasst ergibt dies $n^2 - 2n$ Anfragen, denen $2n - 1$ unterschiedliche Datensätze gegenüberstehen. Die $2n - 1$ ergeben sich aus n unterschiedlichen Knoten und $n - 1$ Clustern, die im Verlauf der Clusterbildung erzeugt werden.

Jeder in der Clusterbildung verwendete Datensatz besteht aus den Plattformeigenschaften der zugrundeliegenden Knoten und aus den Durchschnittswerten ihrer Operatormessdaten. Die Datenbank muss also bei jeder Anfrage Daten aus mehreren Tabellen selektieren und die Operatormessdaten aggregieren.

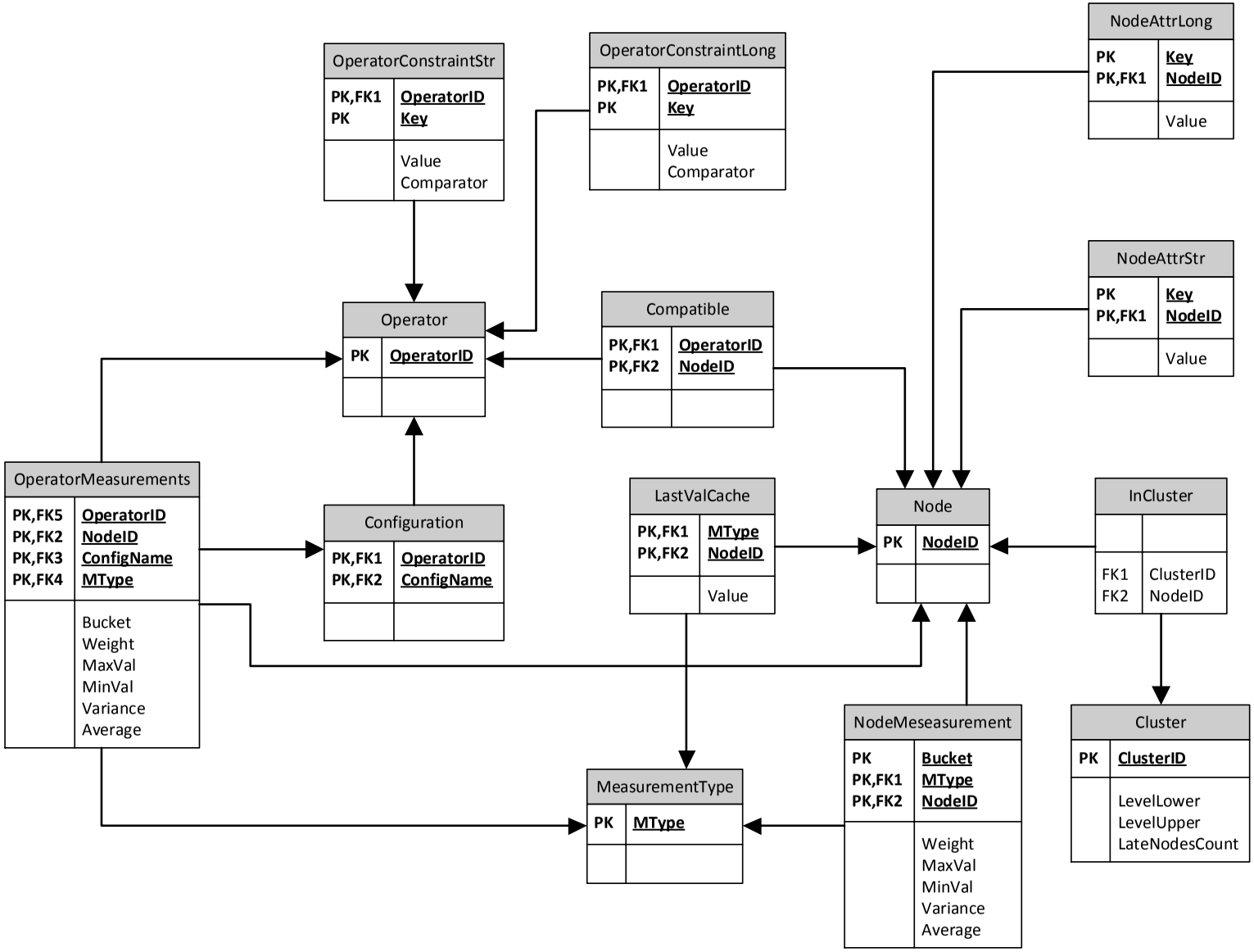


Abbildung 5.1: Übersicht Tabellen für die Datenspeicherung

Um zu verhindern, dass die Datenbank dieselben Daten wiederholt sucht und berechnet, ist es sinnvoll, für die Knotenleistungsclusterbildung einen Cache, also einen Zwischenspeicher, einzusetzen. Zu Beginn der Knotenleistungsclusterbildung werden alle hierfür benötigten Daten aus der Datenbank abgefragt und in den Cache geschrieben. Bei einer Clusterbildung, an der n Knoten beteiligt sind, werden also zu Beginn n Datensätze in den Cache geschrieben. Im Verlauf der Clusterbildung werden $n - 1$ neue Datensätze hinzugefügt und $2n - 1$ Datensätze können aus dem Cache gelöscht werden, da sie nach dem Verschmelzen zu einem Cluster nicht mehr benötigt werden.

Die Anforderungen an die Implementierung des Caches hängen stark mit der aktuellen Implementierung der Knotenleistungsclusterbildung zusammen. In Abschnitt 5.5 werden zwei unterschiedliche Implementierungen für die Knotenleistungsclusterbildung vorgestellt: Eine Implementierung, die auf einem zentralen Rechner ausgeführt wird, und eine weitere Implementierung, die mittels des Map/Reduce-Frameworks „Hadoop MapReduce“ eine verteilte Berechnung durchführt.

Bei der lokalen Implementierung ist es sinnvoll, die Daten direkt im Arbeitsspeicher vorzuhalten, sofern sie in den Arbeitsspeicher passen. Sollten die Daten die Arbeitsspeichergröße überschreiten, kann eine Auslagerung in das lokale Dateisystem erfolgen.

Bei der verteilten Implementierung müssen die Daten auf den jeweiligen Knoten verfügbar sein. Eine Diskussion der hierfür geeigneten Verfahren findet sich in Abschnitt 5.5.2.

Aus dem ER-Diagramm in Abbildung 4.5 wurde nun ein Datenbankdiagramm in Abbildung 5.1 mit den Tabellen und Spalten für die Daten entwickelt. Jede Tabelle in der Abbildung ist mit ihrem Namen im grau hinterlegten Feld gekennzeichnet. Alle Felder des Primärschlüssels einer Tabelle sind unterstrichen und mit `PK` gekennzeichnet. Spalten, die ein Fremdschlüssel aus einer anderen Tabelle sind, sind mit `FK` und einer Zahl gekennzeichnet. Alle Spalten, die eine `ID` enthalten, sind vom Typ `UUID`, sofern im DBMS verfügbar oder von einem verfügbaren Typen, der UUIDs speichern kann. Das Diagramm wird, beginnend mit der Tabelle Operator, grob im Uhrzeigersinn erläutert.

Die Tabelle `Operator` besteht aus dem Primärschlüssel `OperatorID`. Die Tabellen `OperatorConstraintStr` und `OperatorConstraintFloat` haben jeweils eine `OperatorID` als Fremdschlüssel und einen `Key` zusammen als Primärschlüssel. `Key` entspricht dem `Text` von `Key` der Operatoranforderungen, ist also ein String. Die `Value` Spalten enthalten den Wert der Operatoranforderungen und sind vom Typ String für die Tabelle `OperatorConstraintStr` und vom Typ Float für die Tabelle `OperatorConstraintFloat`. Die Spalten `Comparator` enthalten den Vergleich der Operatoranforderungen codiert als Integer, entsprechend der Definition von *compareTo* in Java.

Die Tabelle `Compatible` besteht aus den beiden Fremdschlüsseln `OperatorID` und `NodeID` aus den Tabellen `Operator` und `Node`. Sie bilden zusammen den Primärschlüssel. Die

Einträge von `Compatible` definieren, dass die so verbunden Knoten und Operatoren kompatibel sind.

Die Tabelle `Node` enthält die Spalte `NodeID`. Die Plattformeigenschaften eines Knotens werden in den Tabellen `NodeAttrFloat` und `NodeAttrStr` gespeichert. Sie enthalten jeweils die `NodeID` als Fremdschlüssel und bilden den Primärschlüssel mit dem in `Key` gespeicherten `Key` der Eigenschaft. In der Spalte `Value` wird der Wert der Eigenschaft gespeichert. Für `NodeAttrFloat` ist `Value` vom Typ Float und für `NodeAttrStr` vom Typ String.

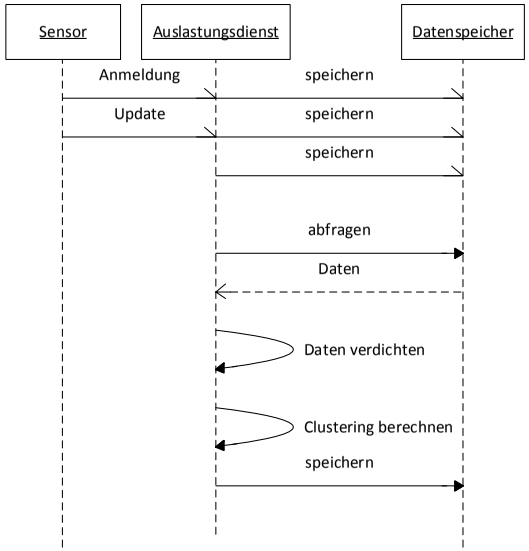
Die Tabelle `InCluster` definiert die Relation NodeIDs aus `Node` und den `ClusterIDs` der in `Cluster` gespeicherten Cluster.

5.2 Implementierung des Auslastungsdienstes

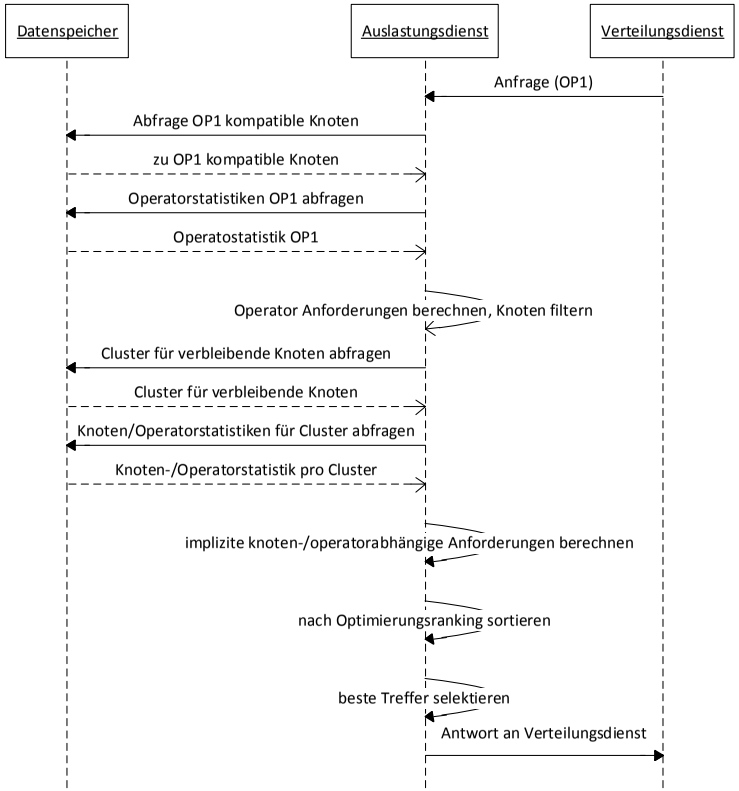
Der Auslastungsdienst wird für die Implementierung in zwei Hauptteile aufgeteilt, die **Datenerfassung** und die **Knotenanfrageverarbeitung**. Die Datenerfassung empfängt die Daten der Sensoren und bereitet sie auf. Die Datenerfassung ist also für die Pflege der Kompatibilitätsmatrix, der Statistiken und der Knotenleistungscluster zuständig. Die Knotenanfrageverarbeitung läuft asynchron zur Datenerfassung und beantwortet Knotenanfragen des Anfragedienstes auf Basis der von der Datenerfassung bereitgestellten Datenbasis. Die Schnittstelle zwischen Datenerfassung und Knotenanfrageverarbeitung ist also nur die Datenbasis, die durch die Datenerfassung geschrieben und durch die Knotenanfrageverarbeitung gelesen wird.

Abbildung 5.2 zeigt eine Übersicht über den Ablauf in Datenerfassung und Knotenanfrageverarbeitung. Die **Datenerfassung** in Abbildung 5.2a des Auslastungsdienstes erhält die Daten der Sensoren. Die Sensoren senden zuerst ihre statischen Daten und machen sich damit dem Auslastungsdienst bekannt. Nach der Anmeldung schicken sie regelmäßige Updates. Der Auslastungsdienst speichert diese Daten im Datenspeicher. Die Datenerfassung verarbeitet regelmäßig die erhaltenen Daten zu aggregierten Statistiken und berechnet beziehungweise aktualisiert das Knotenleistungsclustering.

Die **Knotenanfrageverarbeitung** in Abbildung 5.2b wartet auf Knotenanfragen durch den Anfragedienst. Sobald eine Knotenanfrage eintrifft, werden in mehreren Schritten die relevanten Knoten vom Datenspeicher abgefragt und immer weiter gefiltert. Am Ende des Filterungsprozesses wird dem Anfragedienst eine Antwort zurückgeliefert.



(a) Datenerfassung

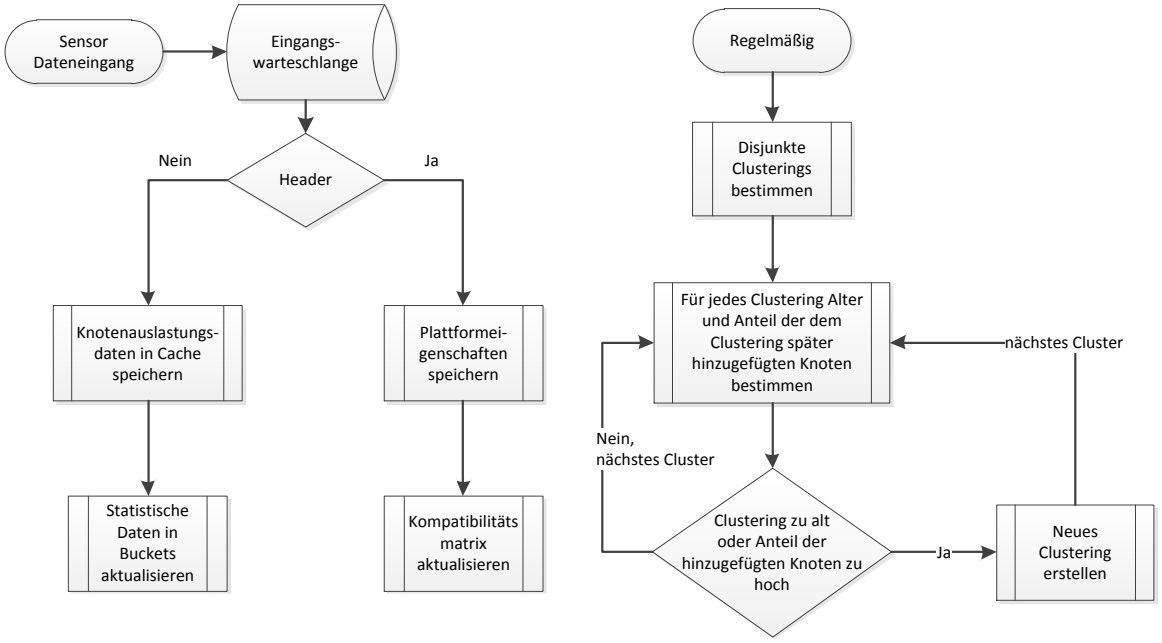


(b) Clusterbildung

Abbildung 5.2: Abläufe im Auslastungsdienst

5.2.1 Datenerfassung

Die Datenerfassung lässt sich weiter in zwei Abläufe unterteilen: in Speicherung und Clusterbildung. Die Speicherung erfasst die eingehenden Daten und speichert sie in der Statistik, während die Clusterbildung aus den statistischen Daten und den Plattformeigenschaften das Clustering erstellt und aktualisiert.



(a) Speicherung **(b)** Clusterbildung

Abbildung 5.3: Abläufe der Datenerfassung

Abbildung 5.3 stellt die beiden Abläufe dar. Der Ablauf **Speicherung** in Abbildung 5.3a wird beim Eingang von Sensordaten durchgeführt. Zuerst werden die eingegangenen Daten in eine Warteschlange eingefügt, die dann in Eingangsreihenfolge abgearbeitet werden. Der Speicherungsablauf überprüft zuerst, ob es sich bei den erhaltenen Daten um einen Header – die Plattformeigenschaften eines Knotens zu Beginn der Übertragung – oder um Messdaten handelt. Bei Empfang eines Headers wird der Knoten in die Knotentabelle (Node) eingetragen und die Plattformeigenschaften des Knotens werden in die Plattformeigenschaftentabellen `NodeAttrFloat` und `NodeAttrStr` gespeichert. Für den neuen Knoten wird dann die Kompatibilitätsmatrix aktualisiert.

Handelt es sich bei den empfangenen Daten um Messdaten, dann werden die Knotenauslastungsdaten in den Cache für die aktuelle Auslastung des Knotens geschrieben und die statistischen Daten für den Knoten und die Operatoren aktualisiert.

Der Ablauf Clusterbildung in Abbildung 5.3b wird regelmäßig aufgerufen und erstellt eine Liste von disjunkten, zueinander inkompatiblen Knotenmengen, für die, wie in Abschnitt 4.4.5 beschrieben, jeweils getrennte Clusterings berechnet werden. Der Zeitabstand für die regelmäßigen Aufrufe entspricht mindestens der Zeitdauer, die die Berechnung der Cluster benötigt, da es nicht sinnvoll ist, den Vorgang ein zweites Mal zu starten, während die Cluster neu berechnet werden. Für jede der disjunkten Knotenmengen wird überprüft, ob bereits ein Clustering erstellt wurde, wie lange dies her ist und wie viele Knoten später zu dem Clustering hinzugefügt wurden. Wenn für eine Knotenmenge kein Clustering existiert, wird dieses erstellt. Wenn das Alter eines Clusterings älter als ein per Konfiguration vorgegebener Wert ist, wird die Clusterbildung neu durchgeführt. Ebenso wird die Clusterbildung neu durchgeführt, wenn der Anteil der später zu einem Clustering hinzugefügten Knoten eine per Konfiguration vorgegebene Schwelle überschreitet. Eine solche Schwelle kann zum Beispiel 20 % nachträglich hinzugefügte Knoten sein.

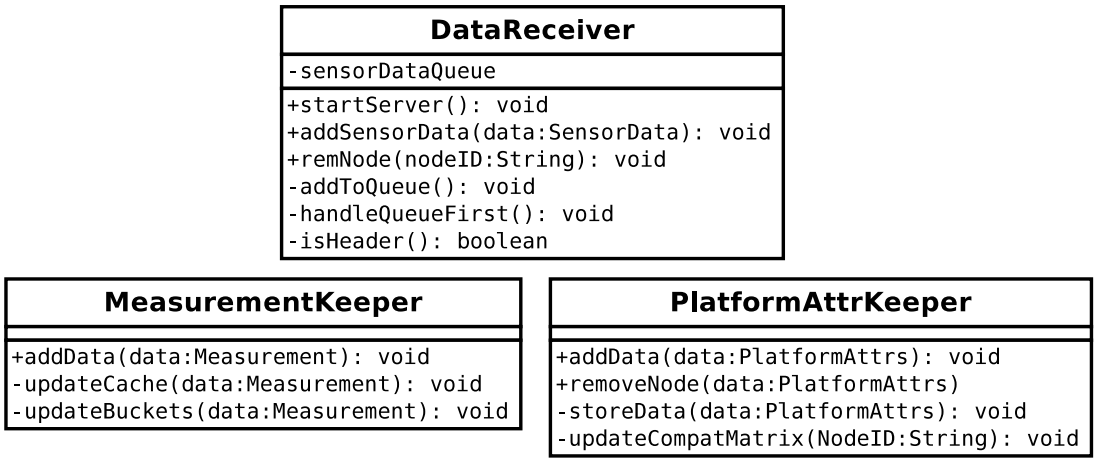


Abbildung 5.4: Klassen für den Ablauf Speicherung

Die beiden Abläufe, Speicherung und Clusterbildung, werden jeweils in getrennten Paketen implementiert und entsprechend den einzelnen Schritten der Abläufe weiter in Klassen unterteilt.

Der Ablauf **Speicherung** wird durch die in Abbildung 5.4 gezeigte Klasse `DataReceiver` implementiert. Über die Funktion `startServer` wird ein XML-RPC-Server bereitgestellt, der die Funktionen `addSensorData` und `remNode` exportiert. Die Funktion `remNode()` hat als Parameter eine NodeID und dient zum Abmelden eines Ausführungsknotens. Aufrufe von `remNode()` werden an die Funktion `removeNode()` der Klasse `NodeAttrKeeper`

weitergereicht, und von dieser Funktion wird der Knoten aus der Kompatibilitätsmatrix entfernt. Über die Funktion `addSensorData()` können Sensoren ihre Daten übertragen. Die Funktion `addSensorData()` hat einen Parameter vom Typ `SensorData`.

Der Typ `SensorData` ist in Abbildung 5.5 dargestellt. Er hat die zwei Subtypen `PlatformAttrs` und `Measurement`. `PlatformAttrs` dient zur Speicherung von Plattformeigenschaften und enthält eine Map, die die in Tabelle 4.1 aufgelisteten Eigenschaften der Plattform enthält. `Measurements` enthält eine Map `nodeMeasurements` und ein Array `operatorMeasurements`. Die Map `nodeMeasurements` enthält die in Tabelle 4.2 aufgelisteten Knotenleistungsmessdaten. Das Array `operatorMeasurements` enthält Datensätze vom Typ `OpMeasurement`, der aus einer *OperatorID* und einer Map von Messwerten für den Operator entsprechend Tabelle 4.3 besteht.

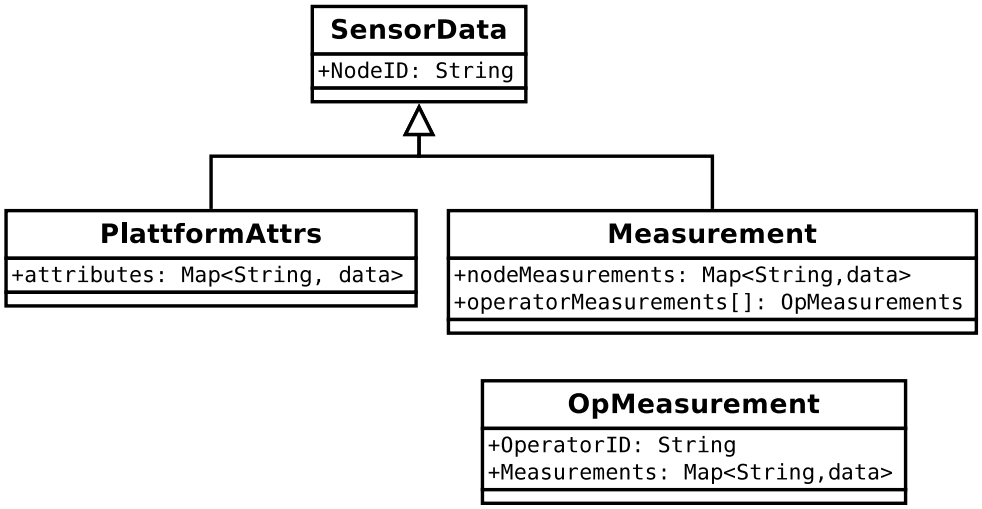


Abbildung 5.5: Datentypen für die Übertragung und Verarbeitung der Sensordaten

Die Funktion addSensorData fügt die Daten der `sensorDataQueue` hinzu. Die Funktion `handleQueueFirst` wird aufgerufen, sobald die Queue Daten enthält. Sie überprüft mit der Funktion `isHeader`, ob der vorderste Datensatz in der Queue ein Header ist. Wenn es sich um einen Header handelt, wird die Funktion `addData()` der Klasse `PlatformAttrKeeper` aufgerufen, und wenn es sich um Messwerte handelt, die Funktion `addData()` der Klasse `MeasurementKeeper`. Nach der Weitergabe der Daten wird der Datensatz aus der Queue entfernt.

Wird `addData()` der Klasse `MeasurementKeeper` mit Messdaten aufgerufen, so wird zuerst mit der Funktion `updateCache()` der Cache für die letzten Messdaten des Knotens aktualisiert. Dann werden über die Funktion `updateBuckets()` die Daten in das aktuelle Bucket verrechnet. Wenn das aktuelle Bucket seine Zeit überschritten hat, werden durch diese

Funktion auch die Buckets verschoben, ein neues hinzugefügt und das älteste Bucket entfernt.

Wird die Funktion `addData()` der Klasse `PlatformAttrKeeper` mit statischen Knotendaten aufgerufen, fügt diese mittels `storeData()` die Plattformeigenschaften zu den Attributtabellen (`NodeAttrString` und `NodeAttrFloat`) und den Knoten zur Tabelle `Nodes` hinzu. Danach wird mittels `updateCompatMatrix()` für alle Operatoren die Kompatibilität zu dem Knoten berechnet und in die Tabelle `Compatible` eingetragen.

Der in Abbildung 5.3b dargestellte Ablauf **Clusterbildung** wird durch die Klassen `ClusterMaintenance` und `Clusterer` implementiert. Die in Abbildung 5.6 dargestellte Klasse `ClusterMaintenance` fragt die getrennt zu analysierenden Knotengruppen aus der Datenbank ab und erzeugt für jede Gruppe einen `Clusterer`.

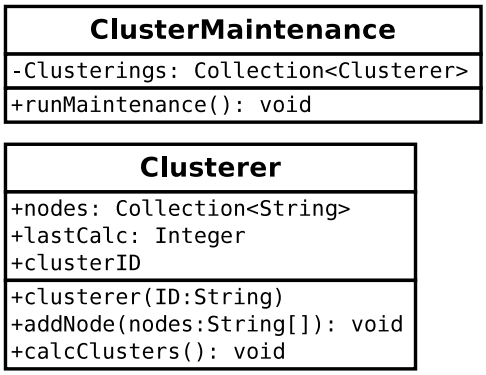


Abbildung 5.6: Definition der Klassen `ClusterMaintenance` und `Clusterer`

In `Clusterings` werden jeweils Referenzen auf die unterschiedlichen `Clusterer` verwaltet. Die Funktion `runMaintenance()` führt die Überprüfung auf Alter und Anteil der später hinzugefügten Knoten für alle `Clusterer` durch.

Die Klasse `Clusterer` speichert in der Variable `nodes` die IDs der enthaltenen Knoten. Die Variable `lastCalc` gibt den Zeitpunkt an, zu dem die Clusterbildung zuletzt durchgeführt wurde. Ein `Clusterer` bekommt über seiner Instanzierung einen eindeutigen Namen, der das von ihm verwaltete Clustering definiert. Über die Funktion `addNodes()` werden dem Clusterer neue Nodes hinzugefügt. Wird die Funktion `addNodes()` aufgerufen, wenn bereits ein Clustering berechnet wurde, dann wird der Knoten den vorhanden Clustern hinzugefügt. Wenn noch kein Clustering berechnet wurde, dann wird der Knoten nur gespeichert, bis die Funktion `calcClusters()` aufgerufen wird. Die Funktion `calcClusters()` berechnet das Clustering für die Knoten des `Clusterers` neu und schreibt sie in die Datenbank sobald die Berechnung beendet ist.

5.2.2 Knotenanfrageverarbeitung

Die Knotenanfragebearbeitung hat nach außen einen Server, der bei Eintreffen einer Knotenanfrage einen neuen Knotenanfrageverarbeiter erzeugt, der die Knotenanfrage bearbeitet und am Ende das Ergebnis ausliefert.

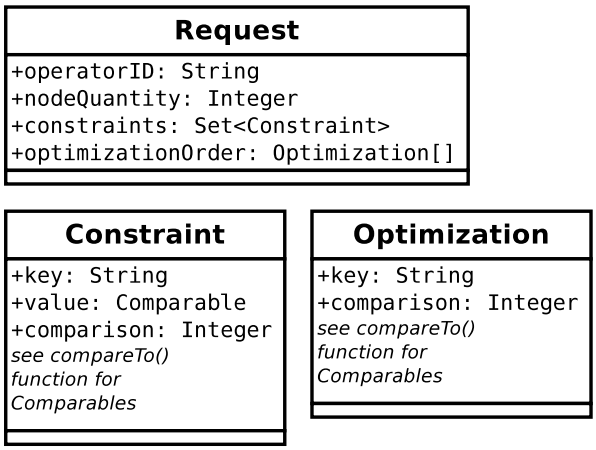


Abbildung 5.7: Definition der Klasse `Request`

Der Knotenanfrageverarbeiter bekommt als Eingabe die Knotenanfrage des Anfragedienstes und filtert in mehreren Schritten die verfügbaren Knoten anhand der Vorgaben. Jeder dieser Filterschritte ist als getrennte Funktion implementiert, so dass diese Filter einzeln erweitert, ersetzt oder entfernt werden können.

Die Knotenanfrage ist wie in Abbildung 5.7 definiert. Eine Anforderung vom Typ `Request` enthält eine `operatorID` vom Typ String, die Anzahl der zurückzuliefernden Ausführungsknoten als `nodeQuantity` vom Typ Integer, ein Set `constraints` vom Typ `Constraint` und ein Array `optimizationOrder` vom Type `Optimization`. Das Set `constraints` ist eine Sammlung, in der kein Element mehrfach vorkommt. Der Typ `Constraint` enthält einen Key vom Typ String und einen Wert vom Typ `Comparable` sowie einen Integer `comparison`, der den von der Funktion `compareTo()` in Java verwendeten Werten entspricht.

Die in `optimizationOrder` gespeicherten Elemente des Typs `Optimization` bestehen jeweils aus `key` vom Typ String und `comparison` vom Typ Integer. Negative Werte von `comparison` entsprechen einer Minimierung des Wertes und positive Werte einer Maximimierung.

Die Anfrage wird durch die in Abbildung 5.8 dargestellte Klasse `RequestHandler` bearbeitet. Ein `RequestHandler` hat als private Variablen die Variable `request` vom Typ `Request`, die die zu bearbeitende Anfrage enthält, und das Set `compatNodes`, das die IDs

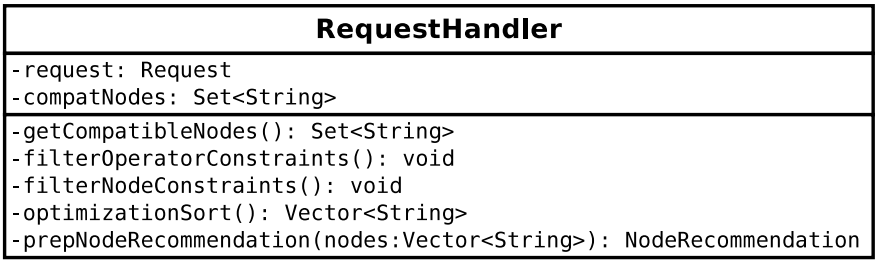


Abbildung 5.8: Definition der Klasse `RequestHandler`

der aktuellen Knoten enthält. Die Funktion `getCompatibleNodes()` liefert alle zu dem in `request` enthaltenen Operator kompatiblen Knoten. Sie wird verwendet, um die Variable `compatNodes` zu befüllen.

Nach der Ausführung von `getCompatiblesNodes()` werden die Funktionen `filterOperatorConstraints()` und `filterNodeConstraints()` nacheinander ausgeführt. Sie berechnen Anforderungen durch den Operator beziehungsweise durch die Knoten und filtern die in `compatNodes` enthaltenen Knoten. Auf die nach den Filterungsschritten noch verbleibenden Knoten wird die Funktion `optimizationSort()` auf `compatNodes` angewandt. Sie liefert einen Vector mit der in `request.nodeQuantity` geforderten Knotenanzahl entsprechend `request.optimizationOrder` sortiert zurück.

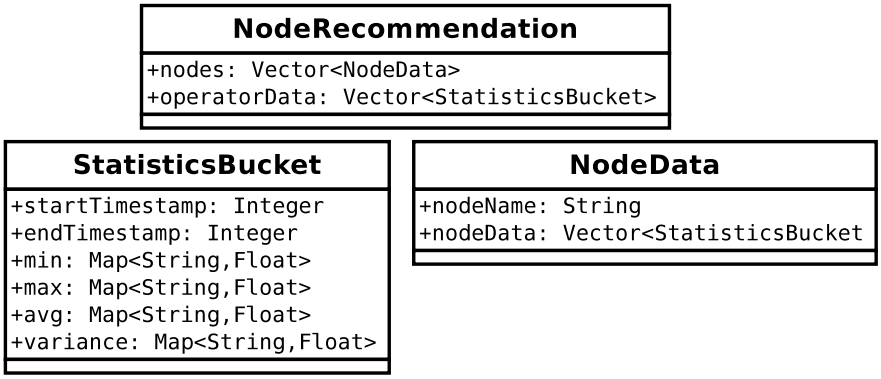


Abbildung 5.9: Definition der Klasse `NodeRecommendation`

Die Funktion `prepNodeRecommendation()` erzeugt aus dem sortierten Knotenvektor eine `NodeRecommendation` wie in Abbildung 5.9 dargestellt. Die `NodeRecommendation` enthält einen Vector `operatorData`, der Datensätze vom Typ `StatisticsBucket` enthält, und einen Vector `nodes` mit Datensätzen vom Typ `NodeData`.

Jedes `StatisticsBucket` enthält einen `startTimestamp` und einen `endTimestamp` vom Typ Integer, die den Zeitraum der Messdaten, die in dem Bucket enthalten sind, angibt. Die Variablen `min`, `max`, `avg` und `variance` enthalten jeweils eine Map, die einem String mit dem Messwertnamen den entsprechenden Statistikwert für den Zeitraum des Buckets zuordnet.

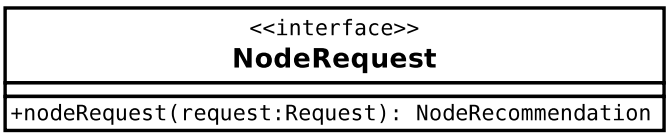


Abbildung 5.10: Definition des `NodeRequest` Interfaces

`NodeData` besteht aus den Variablen `nodeName`, `nodeData` und `operatorData`. Die Variable `nodeName` enthält einen String mit der KnotenID. Der Vector `nodeData` enthält `StatisticsBuckets`, die die statistischen Daten des Knotens enthalten. Der Vector `operatorData` enthält `StatisticsBuckets`, die die statistischen Daten für Operatorausführungen auf diesem Knoten enthalten.

Wenn die `NodeRecommendation` fertig berechnet wurde, wird sie als Antwort auf die Anfrage des Anfragedienstes zurückgegeben und der `RequestHander` wird beendet.

Die Schnittstelle des Auslastungsdienstes für den Anfragedienst wird durch das Interface `NodeRequest`, wie in Abbildung 5.10 gezeigt, beschrieben. Das Interface wird per XML-RPC exportiert und stellt die Funktion `nodeRequest()` bereit, die als Parameter einen `Request` hat und eine `nodeRecommendation` zurückliefert.

5.3 Implementierung der Sensoren

Für die Implementierung der Sensoren ist erforderlich, ihre Ausführungsform und ihre Schnittstellen festzulegen. Da die Sensoren als Dienst im NexusDS Framework ausgeführt werden sollen, müssen sie die Service-Schnittstelle implementieren. Dies erfordert, dass die in Abbildung 5.11 dargestellte Schnittstelle mit den Funktion `start()` und `stop()` implementiert wird. Die Funktion `start()` startet die Datensammlung und die Anmeldung am Auslastungsdienst, während die Funktion `stop()` die Sammlung beendet und den Sensor und somit auch den Knoten vom Auslastungsdienst abmeldet.

Die Schnittstellen des Sensors bestehen aus der bereits in Abschnitt 5.2.1 eingeführten Schnittstelle zum Auslastungsdienst und der Schnittstelle zur Abfrage der Operatormessdaten. Die Abfrage der Operatormessdaten kann entweder durch regelmäßige Abfrage, Pull-Verfahren, oder durch Empfang von durch die Operatoren versendeter Nachrichten,

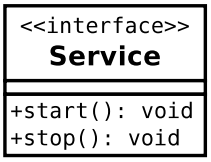


Abbildung 5.11: Definition des `Service` Interface

Push-Verfahren, erfolgen. Das Pull-Verfahren hat als Vorteil, dass die Messdaten erzeugt werden können, wenn der Sensor eine Statusmeldung absetzen will. Die Daten müssen so nicht im Sensor zwischengespeichert werden.

Hingegen hat das Push-Verfahren den Vorteil, dass die Operatoren die Freiheit haben, zu für den Operator sinnvollen Zeitpunkten Messdaten auszuliefern. So können die Messdaten zum Beispiel ereignisabhängig, wenn Daten berechnet wurden oder wenn der Operator eine Veränderung der Arbeitssituation erkennt, übertragen werden. Ist die ereignisabhängige Messdatenauslieferung nicht gewünscht, kann der Operator aber auch eine regelmäßige Messdatenauslieferung implementieren. Zusätzlich entfällt der Verwaltungsaufwand für den Sensor, welche Operatoren verfügbar sind und abgefragt werden müssen.

Aufgrund der größeren Flexibilität wird also das Push-Verfahren für die Übertragung der Messdaten von den Operatoren an den Sensor implementiert. Da Sensoren, Ausführungsdienst und Operatoren alle in einem gemeinsamen Prozess auf demselben Computer ausgeführt werden, kann die Übertragung der Daten durch einen einfachen Methodenaufruf erfolgen. Da mehrere Sensoren zu beliebigen Zeitpunkten und damit auch gleichzeitig ihre Messdaten übertragen können, ist dafür Sorge zu tragen, dass die Übertragungen entsprechend synchronisiert werden. Dies erfolgt, indem die Daten in eine Warteschlange eingefügt werden.

In regelmäßigen, konfigurierbaren Intervallen liest der Sensor die Knotenauslastungsdaten und die in der Warteschlange vorhandenen Operatormessdaten und überträgt sie an den Auslastungsdienst.

Abbildung 5.12 zeigt den Lebenslauf eines Sensors. Bei seinem Start erfasst der Sensor die Plattformeigenschaften und meldet diese an den Auslastungsdienst. Damit ist der Sensor und somit der Knoten dem Auslastungsdienst als verfügbar bekannt. Sobald Operatoren vom Ausführungsdienst (der Ausführungsdienst ist nicht abgebildet) gestartet werden, übertragen diese nach Bedarf Messdaten an den Sensor. Wenn ein Operator beendet wird, zum Beispiel weil er seine Aufgabe erfüllt hat und nicht mehr benötigt wird, überträgt er die letzten Messdaten an den Sensor.

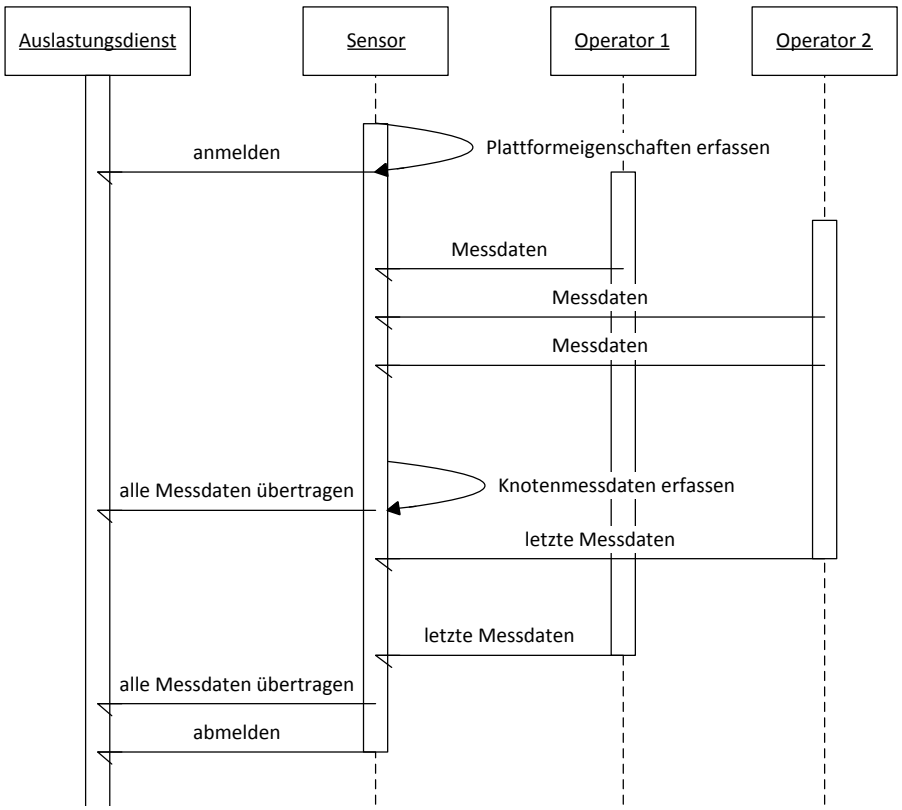


Abbildung 5.12: Sequenzdiagramm Lebenslauf eines Sensors

In regelmäßigen Abständen erfasst der Sensor die Knotenmessdaten und verschickt sie zusammen mit den gesammelten Operatormessdaten an den Auslastungsdienst. Wird die `stop()` Funktion des Sensors aufgerufen, überträgt er die verbleibenden Operatormessdaten an den Auslastungsdienst und meldet sich vom Auslastungsdienst ab.

Abbildung 5.13 zeigt die Klasse `Sensor`, ihre Subklasse `NodeDatCollector` sowie die bereitgestellten Interfaces `Service` und `PerfDataSink`. Die Klasse `Sensor` hat die Variablen `interval` und `OperatorDataQueue`. Die Variable `interval` gibt das konfigurierbare Intervall an, zu dem der Sensor Daten an den Auslastungsdienst überträgt. Der Vector `OperatorDataQueue` enthält die über die Funktion `savePerfData()` des Interfaces `PerfDataSink` übertragenen Operatormessdaten. Da die Implementierung von Vector in Java `synchronized` und damit *thread-safe* ist, also parallele Zugriffe verarbeiten kann, ist die Thread-Sicherheit für das Interface `PerfDataSink` gegeben.

Die Subklasse `NodeDataCollector` stellt die Funktionen `getAttributes()` und `getPerfData()` bereit. Die Funktion `getAttributes()` gibt die Plattformeigenschaften

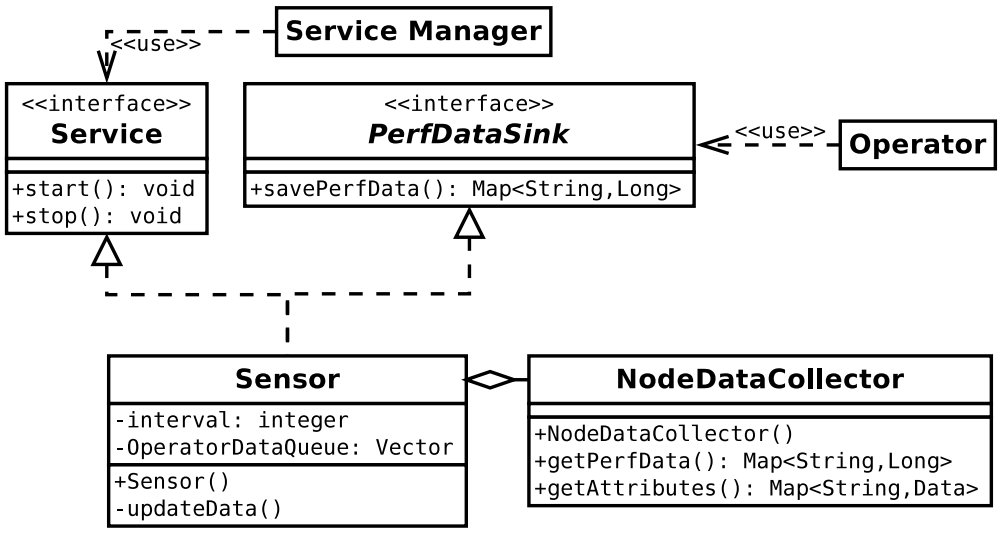


Abbildung 5.13: Klasse `Sensor` und die innere Klasse `NodeDataCollector` sowie die für Service Manager und Operatoren bereitgestellten Interfaces

des Knotens zurück und die Funktion `getPerfData()` die aktuellen Knotenauslastungsdaten.

5.4 Knotenauslastungserfassung

Die Erfassung der Knotenauslastungsdaten erfordert die Abfrage verschiedener vom System zur Verfügung gestellter Werte. Je nach Betriebssystem gibt es hierfür verschiedene Schnittstellen zum Kernel, der diese Daten erfasst. Bei unix- und linuxbasierten Systemen lassen sich die meisten Leistungsdaten aus den virtuellen Dateien des `/proc/` Dateisystems auslesen [BYS+06]. Bei Windows-Systemen hingegen werden die *Performance Counter* entweder über graphische Benutzeranwendungen oder über verschiedene Programmierschnittstellen zur Verfügung gestellt [Mic10, Fri02].

Die von unix- und linuxbasierten Systemen angebotene Schnittstelle ist mit einer Java-Anwendung relativ einfach zu benutzen. Es müssen die entsprechenden virtuellen Dateien eingelesen und nach ihrer Struktur zerlegt werden. Die Werte müssen entsprechend interpretiert werden und können dann als Messdaten fungieren. Bei Windows-Systemen hingegen ist der Zugriff auf die angebotenen Schnittstellen aufwendiger. Entweder wird das Betriebssystem zum Schreiben von Logdateien konfiguriert, die dann mit einiger Verzögerung ausgelesen und interpretiert werden können, oder aber man greift mit Java

auf die Windows-Programmierschnittstellen zu, indem über JNI[1] die entsprechenden Windows-Bibliotheken angesprochen werden. Der Zugriff über JNI ist aufwendig, da zwischen den unterschiedlichen Repräsentationen der Daten konvertiert werden muss.

Java bietet seit Version 1.6 ein eigenes Interface zum Zugriff auf Systeminformationen. Das Interface `OperatingSystemMXBean` enthält vier Funktionen: `getArch()`, `getAvailableProcessors()`, `getName()`, `getSystemLoadAverage()` und `getVersion()`. Mit diesen verfügbaren Funktionen wird also nur ein Teil der in Abschnitt 4.1 geforderten Messwerte geliefert. Hinzu kommt, dass die Funktion `getSystemLoadAverage()` bei einer Windows Java Virtual Machine keine Werte liefert, da der Windows Kernel diesen Wert nicht generiert und auch Java diese Funktion nicht implementiert.

Eine weitere Möglichkeit zur Abfrage der Leistungsdaten ist *SIGAR - System Information Gatherer And Reporter*, eine API, die für Java auf einer Vielzahl an Betriebssystemen Systeminformationen und Performance Messdaten liefert. SIGAR liefert hierfür, sofern für die jeweilige unterstützte Plattform notwendig, native Bibliotheken mit, die die entsprechenden Informationen aus dem System auslesen und über eine einheitliche Java-Schnittstelle zugänglich machen. SIGAR unterstützt eine Vielzahl an Plattformen. Eine Liste der in der aktuellen Version 1.6.3 unterstützten Plattformen befindet sich im Anhang in Abschnitt A.1.

Die von SIGAR zur Verfügung gestellten Informationen enthalten einen Großteil der in den Tabellen 4.1 (Seite 50) und 4.2 (Seite 51) beschriebenen Eigenschaften. Es fehlen jedoch Informationen zur CPU-Queue-Länge, zu den verfügbaren GPUs und zur Verfügbarkeit von Programmen.

Auf Windowsplattformen kann SIGAR jedoch zusätzliche zu den standardmäßig zur Verfügung gestellten Werte, nämlich Werte aus den *Windows Performance Countern*, auslesen. Hierüber lässt sich also auch eine Abfrage der CPU-Queue-Länge realisieren. Auf Linuxplattformen liefert das Programm `vmstat` die Information[2], wie viele Prozesse auf Aufmerksamkeit durch die CPU warten.

Die Indexierung verfügbarer Programme und ihrer Versionen und die Erkennung von GPUs und ihrer Ausstattung muss durch plattformspezifische Abfragen erfolgen oder manuell in die Eigenschaften eines Knoten eingepflegt werden, da auch hierfür keine einheitlichen Schnittstellen in den verschiedenen Betriebssystemen bestehen.

Auf verschiedenen Unix-/Linux-Systemen gibt es eine Vielzahl unterschiedlicher Paketmanagementsysteme, über die Software installiert und verwaltet wird. Auf Windowssystemen kommt zumeist keine zentrale Verwaltung installierter Software zum Einsatz.

[1] Java Native Interface [Ora], Java Schnittstelle, über die unter anderem native Bibliotheken einer Plattform angesprochen werden können.

[2] Im vmstat Manual (`man vmstat`): *r: The number of processes waiting for run time.*

Dementsprechend kann eine automatische Erkennung nur erfolgen, indem alle unterschiedlichen Paketmanagementsysteme unterstützt und abgefragt werden oder pro Softwareprodukt ein spezifischer Test zur Erkennung implementiert wird.

Die Erkennung der GPUs kann unter Windows über die DirectX-Schnittstelle erfolgen. Auf Linux- und Unix-Systemen stehen jedoch wieder verschiedene Schnittstellen zur Auswahl, die teilweise auf ein und demselben System unterschiedliche Ergebnisse in Bezug auf die GPU-Zugriffsmöglichkeiten liefern können.

Daher werden in dieser Implementierung die Plattformeigenschaften, wie GPUs und verfügbare Programme, durch eine manuelle Verwaltung in den Knoteneigenschaften gepflegt. Die Daten können jedoch später durch die Entwicklung plattformspezifischer Erkennungsmethoden automatisch eingepflegt werden.

5.5 Knotenleistungsclustering

Für das Knotenleistungsclustering wurden zwei unterschiedliche Implementierungen durchgeführt. Die erste Implementierung arbeitet in einer einzelnen Instanz und verwendet eine SQL-Datenbank zur Datenhaltung und soweit möglich auch zur Berechnung von Ergebnissen.

Die zweite Implementierung verwendet das Map/Reduce-Paradigma, um die Berechnung der Distanzen zwischen den Clustern, also die Berechnung der Distanzmatrix, verteilt durchzuführen. Für eine effiziente Verteilung der Daten werden diese nicht in Datenbanken, sondern direkt in Dateien geschrieben.

5.5.1 Lokale Berechnung

Die lokal berechnete Implementierung mit datenbankbasierter Speicherung und Berechnung verwendet für die Speicherung aller Daten eine über JDBC[3] angesprochene SQL-Datenbank. Aufgrund der Verwendung der standardisierten Schnittstelle kann die verwendete Datenbanksoftware mit relativ wenigen Änderungen am Programmcode gegen eine andere ausgetauscht werden. Für die Entwicklung wurden die eingebettet –, also ohne externen Datenbankserver – verwendbaren Datenbanken SQLite[4] und HyperSQL[5] verwendet. Beide Datenbanken können die Datenbank vollständig im Arbeitsspeicher halten, statt die Daten auf einen Festspeicher zu schreiben, und so den Zugriffsaufwand auf

[3]Standardschnittstelle für Datenbanken in Java.
[4]SQLite verfügbar unter: http://www.sqlite.org/
[5]HyperSQL verfügbar unter: http://hsqldb.org/

die Festplatte umgehen, vorausgesetzt die Datenbank ist nicht zu groß für den verfügbaren Arbeitsspeicher.

Die Speicherung der Knoteneigenschaften erfolgt in jeweils einer Tabelle für die float- und stringbasierten Eigenschaften. Jede Zeile der beiden Eigenschaftstabellen enthält ein Eigenschaftsschlüssel-Wert-Paar sowie als Index den Namen des Knotens. Durch diese Form der Speicherung entsteht ein gewisser Aufwand beim Speichern eines Knotens, da für jede Eigenschaft eines Knotens ein Insert auf der Datenbank ausgeführt werden muss. Für einen Knoten mit m Eigenschaften müssen also insgesamt m Inserts in der Datenbank ausgeführt werden. Durch die Zusammenfassung der m Inserts zu einer Transaktion verringert sich der Aufwand wieder etwas.

Vorteil dieser Speicherungsform ist, dass Berechnungen teilweise in die Datenbank ausgelagert werden können. So kann beim Verschmelzen von Clustern die Berechnung der Durchschnittswerte des neuen Clusters in der Datenbank ausgeführt werden. Durch die Berechnung in der Datenbank entfällt der Aufwand für das Umwandeln zwischen dem Datenbankformat und der Java-Repräsentation. Die Berechnung der Durchschnittswerte für das Verschmelzen erfolgt, indem die Durchschnittswert der Eigenschaften (`AVG`) per SQL-Anfrage für alle zu verschmelzenden Knoten abgefragt werden. Listing 5.1 zeigt eine solche Anfrage für die Cluster q1, q2, q3, deren Float-Eigenschaften in der Tabelle `clusterFloatAttr` gespeichert sind.

Listing 5.1 Beispiel SQL-Anfrage Durchnittsbildung beim Verschmelzen der Cluster q1 - q3

```
SELECT key, AVG(val) as val FROM clusterFloatAttr
       WHERE cluster IN("q1", "q2", "q3")
       GROUP BY key;
```

Beim Verschmelzen von String-Eigenschaften werden nur solche Eigenschaften übernommen, die für alle Knoten, die verschmolzen werden, denselben Wert haben. Die Abfrage nach Eigenschaften, die dieser Anforderung entsprechen, lässt sich wiederum durch eine SQL-Anfrage erreichen. Listing 5.2 zeigt eine Abfrage für das Verschmelzen der Cluster q1, q2, q3, die aus der Tabelle `clusterStrAttr` ausschließlich Eigenschaften abfragt, die bei allen zu verschmelzenden Knoten denselben Wert haben. Die Abfrage erfolgt durch Gruppieren nach gleichen Werten und der Auswahl aller Schlüssel-Wert-Paare, die nur einen unterschiedlichen Wert haben.

Listing 5.2 Beispiel SQL-Anfrage einheitliche String-Werte beim Verschmelzen der Cluster q1 - q3

```
SELECT key, val FROM clusterStrAttr
       WHERE cluster IN ("q1", "q2", "q3")
       GROUP BY key, val HAVING COUNT(DISTINCT val) = 1;
```

Die Clusterbildung in dieser Implementierung erfolgt, indem die Distanzmatrix für alle Knoten berechnet wird und in einer Datenbanktabelle die Entfernungen zwischen jeweils zwei Clustern gespeichert werden. Die Speicherung der Distanzmatrix erfolgt in der Tabelle `DM`, die die Einträge `(ClusterA, ClusterB, Distanz)` hat, wobei die ersten beiden Felder String-Werte für die ClusterIDs sind und die Distanz ein Float.

Da in jedem Iterationschritt der Clusterbildung zwei Cluster durch ein neues Cluster ersetzt werden, ist es sinnvoll, alle anderen, unveränderten Cluster nicht nochmals abzuspeichern. Statt für jedes Clustering-Level alle beteiligten Cluster zu speichern, erhält jedes Cluster eine obere und untere Grenze, die angibt, auf welcher Iterationsebene das Cluster erzeugt wurde und auf welcher Iterationsebene es durch Verschmelzen zu einem neuen Cluster entfernt wird. Die Lebensdauer eines Clusters wird in der Tabelle `clusters` anhand Einträgen von `(cluster, lower, upper)` gespeichert. Bei der Erzeugung eines neuen Clusters wird es in diese Tabelle mit seinem Namen und dem aktuellen Clusteringiterationsschritt als `lower`-Wert eingetragen. Der Wert von `upper` wird zunächst auf ∞, repräsentiert durch den maximalen Integer-Wert, gesetzt. Sobald ein Cluster durch Verschmelzen das Ende seiner Lebenszeit erreicht hat, wird der aktuelle Clusteringiterationsschritt in den `upper`-Wert des Clusters eingetragen.

Der eigentliche Verschmelzungsprozess, das Zusammenfügen von mehreren Knoten zu einem Cluster, wird durch eine Tabelle `clustersNodes` implementiert. In dieser Tabelle werden einem Cluster die zu ihm gehörenden Knoten zugeordnet. Es werden hierbei nicht die einzelnen Verschmelzungsschritte abgebildet, sondern nur deren Ergebnis. Dies entspricht einem Baum, in dem jeder Knoten nicht auf seine direkten Nachfolger zeigt, sondern direkt auf die Blätter. Vorteil dieser Speicherung ist, dass für ein Cluster mit einer einzigen Anfrage alle enthaltenen Knoten abgefragt werden können, statt der Verzweigung eines Baumes zu folgen. Da der Baum eine Höhe von $n - 1$ erreichen kann, müssten bei der Verfolgung der Verzweigungen bis zu $n - 1$ Elemente betrachtet werden.

Um den Aufwand beim Zugriff auf die Werte eines Clusters zu verringern, werden die Durchschnittswerte eines Clusters, die das Cluster repräsentiert, beim Verschmelzen in die Tabellen `clusterFloatAvg` und `clusterStrAvg` geschrieben.

Eine Zusammenfassung über alle verwendeten Tabellen, ihre Felder und die SQL-Typen der Felder findet sich in Tabelle 5.1. Die Tabellen `clusterStrAttr` und `clusterFloatAttr` entsprechen den bereits in Abbildung 5.1 auf Seite 75 vorgestellten Tabellen `NodeAttrFloat` und `NodeAttrStr`. Die Tabelle `clusters` entspricht der Tabelle `Cluster` und gibt die Clusteringlevel an, auf denen die Cluster existieren. Die Tabelle `clustersNodes` entspricht der Tabelle `inCluster` und ordnet die einzelnen Nodes den Clustern zu. Die Tabelle `dm` speichert die Distanzmatrix. Sie hat keine Entsprechung in Abschnitt 5.1, da die Distanzmatrix nur während der Clusterbildung benötigt wird. Die Tabellen `clusterFloatAvg` und `clusterStrAvg` speichern die Durchschnittswerte eines Clusters. Sie entsprechen daher den Tabellen `NodeAttrFloat` und `NodeAttrStr`, jedoch

Tabellenname			
clusterStrAttr	cluster VARCHAR	key VARCHAR	val VARCHAR
clusterFloatAttr	cluster VARCHAR	key VARCHAR	val Float
clusters	cluster VARCHAR	lower Integer	upper INTEGER
clustersNodes	cluster VARCHAR	node VARCHAR	
dm	clusterA VARCHAR	clusterB VARCHAR	dist FLOAT
clusterFloatAvg	cluster VARCHAR	key VARCHAR	val Float
clusterStrAvg	cluster VARCHAR	key VARCHAR	val VARCHAR

Tabelle 5.1: Tabellen für die Clusterbildung mit Feldnamen und Feldtypen

mit dem Unterschied, dass sie sich auf Cluster und nicht auf einzelne Nodes beziehen. Auch ihre Daten werden ausschließlich für die Clusterbildung verwendet.

5.5.2 Map/Reduce-basierte Berechnung

Da die Clusterbildung einen quadratischen Aufwand für die Berechnungen der Distanz aller Knoten zueinander hat, ist schon bei einigen hundert Knoten mit einem Berechnungsaufwand von mehreren Minuten zu rechnen, wenn die in Abschnitt 5.5.1 verwendete Implementierung verwendet wird. Für große Knotenanzahlen ist eine solche Clusterbildung offensichtlich kaum noch durch einen üblichen PC in einem akzeptablen Zeitraum zu berechnen. Daher wurde der Algorithmus nochmals in einer auf dem Map/Reduce-Paradigma basierten verteilbaren Version implementiert.

Das Map/Reduce-Paradigma basiert darauf, dass auf einer Menge von Schlüssel- und Wert-Paaren parallel die gleiche Funktion angewendet wird. Die Funktion erzeugt aus dem Eingabeschlüssel-Wert-Paar ein Zwischenergebnis. Das Zwischenergebnis hat wiederum die Form eines Schlüssel-Wert-Paares. In einem zweiten Schritt werden dann die Zwischenergebnisse der Funktionsanwendung anhand der Schlüssel sortiert und zusammengeführt. Die Anwendung der Funktion auf die Daten wird hierbei als Map-Schritt bezeichnet und die Zusammenführung als Reduce-Schritt [DG08].

Der Map-Schritt ist parallelisierbar, da er ausschließlich von seinem Schlüssel-Wert-Paar abhängt. Der Reduce-Schritt hingegen hängt im Allgemeinen von den Ergebnissen aller Map-Anwendungen ab und muss daher zentral durchgeführt werden. Bei manchen Algorithmen besteht die Möglichkeit, einen Reduce-Schritt auf jedem Map/Reduce-Knoten auf Teilmengen der Daten – den sogenannten Combine-Schritt – zwischen Map und Reduce durchzuführen, wenn durch eine verteilte Reduzierung von Teilmengen der Map-Ergebnisse schon ein Vorteil erzielt werden kann. Die Anwendung des Combine-Schrittes wird später anhand eines Beispiels erläutert.

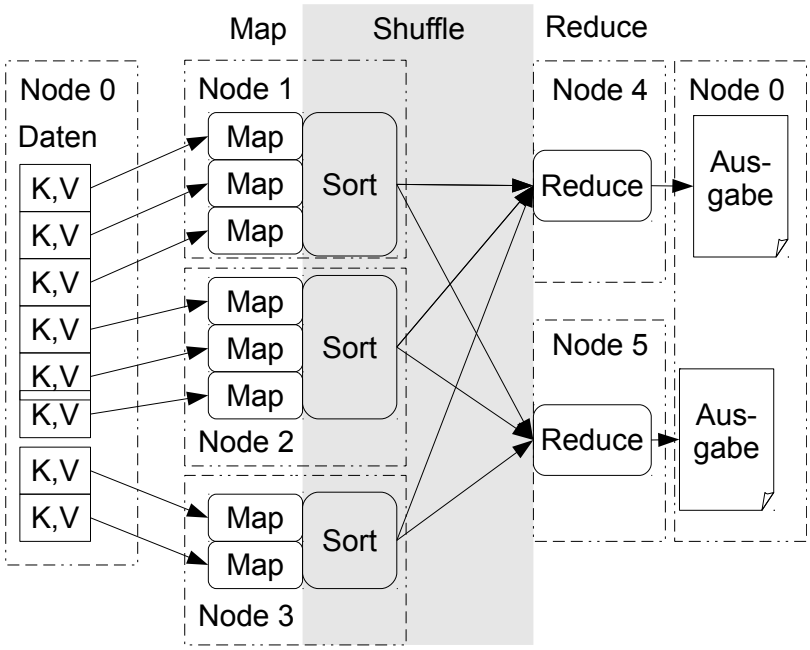


Abbildung 5.14: Map/Reduce-Ablauf

Abbildung 5.14 stellt einen Map/Reduce-Ablauf dar. Die Daten werden als Menge von Schlüssel-Wert-Paaren (`Key`, `Value`), vorgehalten. Jeder Datensatz wird von einem Map-Prozess bearbeitet. Die Map-Prozesse werden in der Abbildung von drei Rechnern, Node 1 bis Node 3, ausgeführt. Jeder der Rechner erhält die Map-Funktion und führt sie nacheinander auf allen ihm zugeteilten Daten aus. Die Ergebnisse werden anhand ihrer Schlüssel in der sogenannten *Shuffle*-Phase auf jedem Map/Reduce-Knoten sortiert und dem für den jeweiligen Schlüssel zuständigen Reducer-Prozess gesendet. Die Anzahl der Reducer hat als Obergrenze die Anzahl der verschiedenen Schlüssel und als Untergrenze 1. Alle Datensätze für einen Schlüssel werden auf demselben Reducer-Prozess bearbeitet. Ein Reducer kann jedoch Daten für mehrere Schlüssel verarbeiten. Jeder Reducer führt seine Reduzierfunktion auf den Daten eines Schlüssels aus und schreibt das Ergebnis in seine Ausgabe.

Ein Beispiel für die Anwendung von Map/Reduce gibt White [Whi09] durch die Suche nach den maximalen Jahrestemperaturen in den amerikanischen Wetteraufzeichnungen an. Da eine Vielzahl an Wetterstationen existiert, ist die Datenmenge für die Jahre 1901-2001 sehr groß. Die Daten liegen als Datensätze, die jeweils eine Beobachtung einer Vielzahl von Wetterparametern beschreiben, vor.

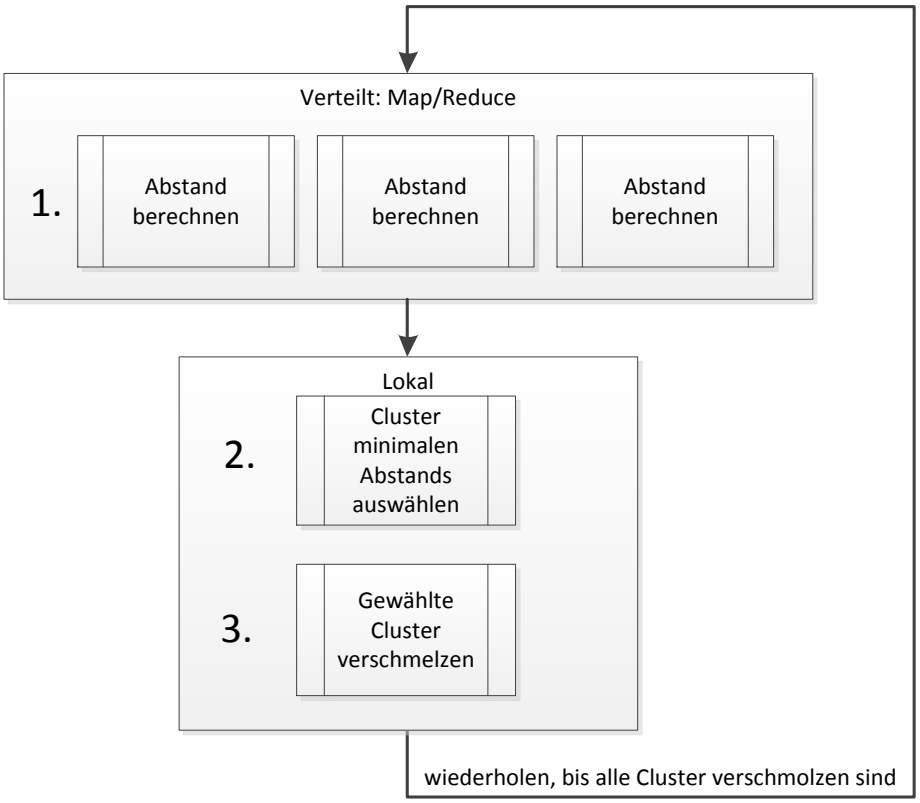


Abbildung 5.15: Ablauf des Clusterbildungs-Algorithmus mit Map/Reduce

Um nun die Maxima-Suche mittels Map/Reduce durchzuführen, wird die Map-Funktion jeweils mit einer Jahreszahl als Schlüssel und einem Datensatz als Wert aus diesem Jahr gespeist. Die Map-Funktion analysiert den Datensatz und gibt die Jahreszahl als Schlüssel und die Lufttemperatur aus dem Datensatzes als Wert zurück. Die Reduce-Funktion wird für jeden Schlüsselwert (hier den Jahreszahlen) mit den von den Map-Funktionen für diesen Schlüssel erzeugten Werten aufgerufen. Für die Maximaltemperatur muss die Reduce-Funktion also nur das Maximum der übergebenen Werte auswählen und zusammen mit der Jahreszahl zurückgeben.

Da in diesem Beispiel jeder Wert, der kein lokales Maximum ist, auch kein globales Maximum werden kann, könnte die Reduce-Funktion zusätzlich auf jedem der Rechner auf den lokal verfügbaren Ergebnissen als Combine-Funktion ausgeführt werden. Durch das Anwenden der Combine-Funktion vor dem Reduce müssen weniger Daten zwischen den Rechnern übertragen werden.

Für Map/Reduce steht ein Framework unter Betreuung der Apache Software Foundation[6] zur Verfügung. Das *Hadoop MapReduce*[7] genannte Framework stellt die Funktionalität zur Verfügung, um Berechnungen per Map/Reduce durchzuführen. Da Hadoop MapReduce in Java implementiert ist und direkte Schnittstellen für Java-Anwendungen anbietet, wird hier Hadoop zur Implementierung des Algorithmus verwendet. Für einige Plattformen enthält Hadoop MapReduce zusätzlich native Bibliotheken, um Vorgänge zu beschleunigen.

Wie bereits erläutert, wird bei der Verwendung von Map/Reduce der Map-Schritt verteilt ausgeführt. Durch eine verteilte und parallele Ausführung des aufwendigsten Schrittes im Clusterbildungs-Algorithmus kann die Zeit zur Berechnung der Cluster verringert werden. Wie in Abbildung 5.15 dargestellt kann der Clusterbildungs-Algorithmus in die Schritte 1., 2. und 3. aufgeteilt werden, die wiederholt werden, bis die Clusterbildung vollständig durchgeführt ist. Im Allgemeinen wird hier von Clustern gesprochen, auch wenn diese zu Beginn jeweils nur einen Knoten enthalten.

In der Implementierung des Algorithmus wird nun Schritt 1 verteilt ausgeführt, die Schritte 2 und 3 zentral. Zur verteilten Berechnung der Abstände zwischen allen Clustern wird jeweils der Abstand zwischen zwei Clustern durch eine Anwendung der Map-Funktion berechnet. Die Map-Funktion benötigt als Eingabe die Daten der beiden Cluster und hat als Ausgabe die Namen der Cluster und ihre Distanz.

Die Map-Funktion zur Distanzberechnung ist definiert als:

$$\text{Map} : \text{id1}, \text{id2} \rightarrow \text{id1}, (\text{id2}, \text{dist})$$

Die Eingabe zweier ClusterIDs wird also auf die gleichen ClusterIDs und den Abstand zwischen den Clustern abgebildet. Hierbei müssen jedoch die zweite ClusterID `id2` und die Distanz `dist` zu einem Objekt verschmolzen werden, da die Map-Funktion ausschließlich auf Schlüssel-Wert-Paaren arbeitet.

Der Reduce-Schritt wird in dieser Implementierung des Algorithmus nicht verwendet, da auch die Distanzen gespeichert werden, die im aktuellen Schritt nicht minimal sind. Die Distanzen, die im aktuellen Schritt nicht minimal sind, können in einem späteren Schritt, nachdem die minimale Distanz durch Verschmelzen entfernt wurde, zur minimalen Distanz werden. Das Speichern der nicht minimalen Distanzen entspricht der Wiederverwendung der Distanzmatrix aus dem vorherigen Schritt, wie in Abschnitt 4.4.5 beschrieben.

[6]The Apache Software Foundation http://www.apache.org/
[7]Hadoop MapReduce http://hadoop.apache.org/mapreduce/

Mit der Entscheidung, alle berechneten Abstände für die spätere Wiederverwendung zu speichern, folgt, dass der Map/Reduce-Prozess alle Abstände ausgeben muss. Es ist sinnvoll, aus den erhaltenen Distanzen lokal das Minimum auszulesen, da eine verteilte Ausführung den Aufwand kaum reduzieren würde. Für das lokale Auslesen müssen die Ergebnisse einmal linear gelesen werden, während für eine verteilte Ausführung die Distanzen des vorherigen Schrittes wieder in einen Map/Reduce-Prozess verteilt werden müssten. Sie müssten also gelesen und an die Map/Reduce-Knoten übertragen werden.

Ebenso ist eine verteilte Durchführung von Schritt 3 nicht sinnvoll, da immer nur zwei Cluster miteinander verschmolzen werden.

Um einen effizienten Zugriff der Map-Funktion auf die Clustereigenschaften zu gewährleisten, wurden verschiedene Konzepte evaluiert. Die zur Verfügung stehenden Abfragemöglichkeiten sind per Datenbankabfrage, durch Speicherung in einem verteilten Dateisystem, durch Verwendung der Clusterdaten als Map-Parameter (statt der IDs) und durch Verwendung des Distributed Cache von Hadoop.

Die Verwendung einer Datenbank zur Speicherung und Abfrage der Daten hat den Vorteil, dass die Daten nur an einem Ort gespeichert und verwaltet werden müssen. Als Nachteil ergibt sich jedoch, das dies dem Verteilungsgedanken widerspricht und zu einem Flaschenhals führt, da gleichzeitig viele Map-Prozesse Daten abfragen und sich so gegenseitig behindern können. Zusätzlich ist bei einer entfernten Anfrage eine gewisse Latenz für jede einzelne Anfrage nicht zu umgehen.

Ähnlich zur Speicherung in einer Datenbank verhält sich die Verwendung eines verteilten Dateisystems. Zwar ist bei entsprechender Verteilung ein Flaschenhals unwahrscheinlicher, jedoch kann es häufig zu Verzögerungen durch nicht lokal verfügbare Daten kommen. Zudem entsteht ein Verwaltungs-Overhead durch die Verteilung und Abfrage.

Bei der Übergabe der Daten als Map-Parameter ist der große Vorteil, dass die benötigten Daten in der Map-Funktion direkt vorliegen und keine weiteren Daten angefragt werden müssen. Dies entspricht dem reinen Map/Reduce-Paradigma, da so die Map-Funktion keine weiteren Daten lesen muss. Als Nachteil ergibt sich jedoch, dass dieselben Daten mehrfach übertragen werden müssen.

Formel 5.1 Overhead pro Clusterbildungs-Schritt bei Übertragung der Clusterdaten als Key und Value

$$\text{Overhead} = (2 * \sum_{i=1}^{n-1} i) - n = 2 * \frac{(n-1)(n-1+1) - n}{2} = n^2 - 2n$$

Im ersten Clusterbildungs-Schritt müssen, wie in Abschnitt 4.4.4 gezeigt, $\sum_{i=1}^{n-1} i$ Abstände für n verschiedene Knoten berechnet werden. Daraus ergibt sich, dass $\sum_{i=1}^{n-1} i$ Map-Aufrufe mit jeweils zwei Knotendatensätzen, also $2 * \sum_{i=1}^{n-1} i$ Datensätze übertragen werden müssen, obwohl es nur n verschiedene Datensätze gibt. Es entsteht also ein Daten-Overhead von $n^2 - 2n$, wie in Formel 5.1 gezeigt, der bei entsprechend großen Knoten- und damit Clusteranzahlen durchaus zu einem Problem werden kann.

Die letzte Möglichkeit ist die Verwendung des Distributed Cache, der von Hadoop MapReduce zu Verfügung gestellt wird. Hierbei handelt es sich um Dateien beziehungsweise gepackte Archive, die vor der Ausführung der Map-Funktion an alle Map/Reduce-Knoten übertragen werden. Auf den Map/Reduce-Knoten werden die Dateien lokal für die Map-Funktion verfügbar gemacht. Wenn nun alle Clusterdaten in eine einzige Datei gespeichert und vor Ausführung an alle Map/Reduce-Knoten übertragen werden, entsteht ein Daten-Overhead in Höhe von $n * (k - 1)$ mit k als Anzahl der Map/Reduce-Knoten, auf denen die Berechnung ausgeführt wird. Der Overhead ergibt sich dadurch, dass nicht bekannt ist, welchen Map/Reduce-Knoten das Map auf welche Daten ausführen, also jeder Map/Reduce-Knoten alle Daten braucht. Bei einem Map/Reduce-Knoten würde dieser den Zugriff auf alle n Knotendatensätze benötigen, bei k Map/Reduce-Knoten braucht jeder zusätzliche Map/Reduce-Knoten eine Kopie der n Knotendatensätze.

Für die Implementierung wurde nun das zuletzt vorgestellte Konzept des Distributed Cache verwendet, da hier die Kombination von Overhead und Latenz am geringsten ist. Bei jeden Aufruf der Map-Funktion werden zwei Datensätze aus dem Distributed Cache gelesen. Um die in Relation zu Arbeitsspeicherzugriffen langsamen Festplattenlesezugriffe zu minimieren, wurde zusätzlich ein kleiner Arbeitsspeichercache für die Clusterdaten entwickelt und implementiert.

Dieser Clusterdatencache macht sich zunutze, dass die Map-Funktion auf einem Map/Reduce-Knoten häufig mit aufeinander folgenden Datensätzen arbeitet. Dazu kommt es durch die blockweise Verteilung der Eingabedaten an die Map/Reduce-Knoten. Als Beispiel sei folgende Ausführung für einen Map/Reduce-Knoten gegeben: `Map(1, 2)`, `Map(1, 3)`, `Map(1, 4)`, `Map(1, 5)`, `Map(1, 6)`, `Map(2, 3)`, `Map(3, 4)`, `Map(4, 5)`, `Map(5, 6)`. Offensichtlich werden in dieser Ausführung die Daten des Clusters mit der ID 1 fünfmal in direkt aufeinander folgenden Map-Aufrufen benötigt und der Datensatz mit der ID 2 viermal.

Um nun ein wiederholtes Lesen des Datensatzes von der Festplatte zu umgehen, wurde dem Lesen ein Cache mit zwei Einträgen vorgeschaltet. Dieser Cache ist in Form einer FiFo-Queue der Länge zwei implementiert. Bei einer Leseanfrage wird immer zuerst der Cache angefragt. Ist der Datensatz im Cache verfügbar, so wird er direkt aus dem Cache zurückgeliefert und der Datensatz in der Queue an die letzte Position verschoben. Wenn ein Datensatz im Cache nicht verfügbar ist, wird er von der Festplatte gelesen und in der

Queue hinten angefügt. Er verdrängt damit den ältesten, vordersten Datensatz aus der Queue.

Der Cache macht sich zunutze, dass sich in Hadoop MapReduce über die Konfiguration das Wiederverwenden der Java Virtual Machine (JVM) für nacheinander auf einer Maschine ausgeführte Map-Anwendungen erzwingen lässt. Da die Map-Funktion als *static* definiert ist, verwenden alle in einer JVM ausgeführten Map-Funktionen dieselben Variablen. Die FiFo-Queue ist als *static*-Klassenvariable definiert.

Messergebnisse

Es wurde ein kurzer Test zu Bestätigung der Annahmen über die Parallelisierbarkeit durchgeführt. Für den Test wurde nicht das vollständige Clustering durchgeführt, sondern nur der erste und aufwendigste Clustering-Schritt. Die absoluten Werte sind nicht zuverlässig, da die Messumgebung aus zwei unterschiedlich ausgestatteten Computern bestand und der Code für die Messungen nicht optimiert wurde. Anhand der Messdaten lässt sich jedoch ablesen, dass durch die parallele Ausführung ein deutlicher Geschwindigkeitszuwachs zu erwarten ist.

Für die Messung wurde die einfach parallele Ausführung, also ein Map-Prozess, der alle Distanzen berechnet, einer vierfach parallelen Ausführung gegenübergestellt. Der Messrechner 1 ist ein PC mit einem AMD Athlon X2 Prozessor mit 2,5 GHz und Ubuntu 10.10 64 Bit. Der Messrechner 2 ist ein Notebook mit einem Intel Core2 Duo mit 2 GHz und Ubuntu 10.04 32 Bit. Messrechner 1 ist in allen Tests der Masterknoten für die Verwaltung und in den vierfach parallelen Tests zusätzlich ein Slaveknoten, der auch Berechnungen durchführt.

Hadoop MapReduce wurde so konfiguriert, dass jeder Slaveknoten bis zu zwei parallele Map-Vorgänge durchführen kann. Bei zwei Slaveknoten ergeben sich also maximal vier parallele Map-Vorgänge.

Als Tests wurden jeweils mit einfacher und vierfacher Parallelität 179700, 319600 und 499500 Abstände berechnet. Bei den Tests mit einfacher Parallelität wurde die Steuerung von Messrechner 1 übernommen und die Berechnungen wurden von Messrechner 2 in einem einzigen Map-Vorgang durchgeführt. Für die Tests wurden jeweils 600, 800 und 1000 Knoten mit 100 Attributen erzeugt. Die Attribute wurden von einem Zufallsgenerator entweder mit einem zufälligen String oder einem zufälligen Integer Wert befüllt. Die Anzahl der berechneten Abstände ergibt sich aus der Knotenanzahl n durch $((n-1)^2 + n - 1)/2$.

Die Ergebnisse sind in Abbildung 5.16 dargestellt. Auf der X-Achse ist die Anzahl der berechneten Abstände aufgeführt und auf der Y-Achse die für die Berechnungen

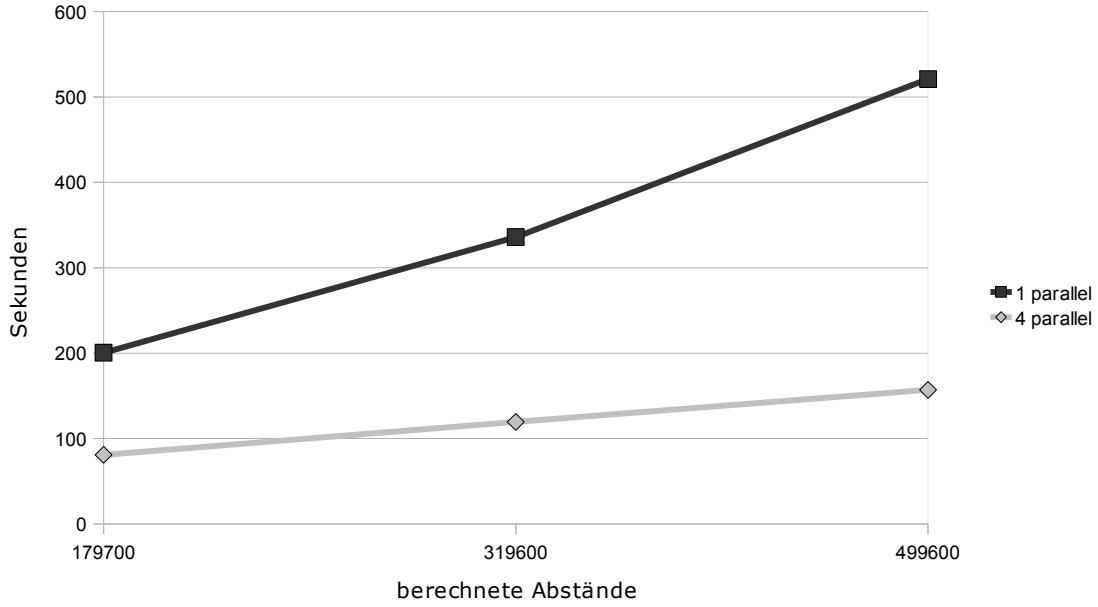


Abbildung 5.16: Messergebnisse: Map/Reduce-Beschleunigung

benötigte Zeit. Dem Diagramm ist zu entnehmen, dass die parallel berechnete Version einen deutlichen Zeitvorteil erzielt. Der Zeitvorteil wird bei größeren Knotenanzahlen noch deutlicher, da hier der Overhead für den Start des Map-Prozesses weniger ins Gewicht fällt.

Kapitel 6

Resümee

Ziel dieser Arbeit war es, NexusDS um einen Auslastungsdienst zu erweitern, der anhand von Messdaten Empfehlungen für Ausführungsknoten zur Ausführung von Operatoren gibt. Hierfür wurde NexusDS vorgestellt. Nach einer Analyse von verwandten Arbeiten zur Leistungsmessung und Vorhersage von Anwendungsperformance in verteilten und parallelen Umgebungen wurde eine Architektur für den Auslastungsdienst entwickelt.

Um den Auslastungsdienst mit Leistungsmesswerten und Daten über die Plattformeigenschaften zu versorgen, wurde das Konzept von verteilten Sensoren angewandt. Die von ihnen gesammelten Daten werden im Auslastungsdienst zu Statistiken aggregiert.

Hierfür wurde analysiert, welche Messdaten erfasst werden müssen, um aus ihnen Vorhersagen über die Leistung von Operatoren auf verschiedenen Ausführungsknoten unter bestimmten Parametrisierungen zu berechnen. Für die Statistiken wurde eine Vorgehensweise zur Historisierung von veralteten Daten vorgestellt.

Für die Verbreiterung der Datenbasis wurde das Konzept des Knotenleistungsclusterings eingeführt und es wurden verschiedene Clusteringalgorithmen evaluiert. Für den gewählten Algorithmus für hierarchisches Clustering wurden verschiedene Verfahren zur Aufwandsreduzierung vorgestellt.

Um für die Implementierung des Auslastungsdienstes ein geeignetes Verfahren für die Berechnung des Knotenleistungsclusterings zu haben, wurde das Map/Reduce-Paradigma zur verteilten Berechnung vorgestellt. Auf der Basis von Map/Reduce wurde der Algorithmus zur verteilten Berechnung der Distanzen zwischen den Clustern beim Knotenleistungsclustering entwickelt. Um die Festplattenzugriffe durch die Map-Funktion zu minimieren, wurde ein Cache entwickelt, der sich die Datenstruktur und Reihenfolge der Ausführung zunutze macht.

Kapitel 7

Zukünftige Arbeiten

An verschiedenen Stellen dieser Arbeit haben sich weitere Themen aufgezeigt, die in zukünftigen Arbeiten zu einer Verbesserung des Systems führen können oder interessante Forschungsthemen ergeben.

7.1 Bewertung der Vorhersagequalität

Der Auslastungsdienst erstellt Vorhersagen über geeignete Knoten für die Ausführung von Operatoren unter ihrer Parametrisierung. Allerdings ist bei einer Vorhersage nicht bekannt, wie gut ihre Qualität ist, also inwieweit sie der Realitität entsprechen wird. Ein spannendes Thema ist daher zum einen die Messung in einem auf allen Architekturebenen vollständigen System – das hier leider noch nicht zur Verfügung stand –, wie weit die Vorhersagen des Auslastungsdienstes von einem Optimum abweichen. Dafür könnte das Gesamtsystem im Produktivbetrieb beobachtet werden und anhand von zusätzlichen Messdaten im Nachhinein beurteilt werden, wie gut die Entscheidungen des Auslastungsdienstes tatsächlich waren.

Unabhängig von der Bewertung der Vorhersagen im Nachhinein wäre auch eine Selbstbewertung der Vorhersagen durch den Auslastungsdienst interessant. Es könnte anhand der Durchschnittswerte für die gemessenen Werte und der zugehörigen Varianz bestimmt werden, mit welcher Wahrscheinlichkeit ein Ereignis eintritt. Das Ziel wäre also, anzugeben, mit welcher Wahrscheinlichkeit ein Operator auf einem bestimmten Knoten das gewünschte Ergebnis liefert.

7.2 Heuristik für die Anzahl der zu betrachtenden Cluster

Das implementierte Clusterbildungsverfahren erlaubt es, die Granularität, für die die Knotenleistungscluster betrachtet werden, im Betrieb frei zu wählen. In der Arbeit wurde als Heuristik angegeben, dass es nicht sinnvoll ist, mehr als die Anzahl unterschiedlich ausgestatteter Systeme als getrennte Cluster zu betrachten.

Mit der Anzahl der betrachteten Cluster, für die Statistiken für die Operatorausführungen zusammengeführt werden, ändert sich nicht nur der Aufwand, sondern auch die Vorhersagequalität. Die Extrembeispiele vom oberen und unteren Ende der Skala sind das Betrachten eines einzelnen Clusters mit allen Knoten und das Betrachten aller Knoten einzeln. Im ersten Fall wird die Vorhersage durch die vielen Knoten, die nicht ähnlich sind, verfälscht. Für jeden Knoten kann es zu unrealistischen Abweichung von den Ober- und Untergrenzen sowie von den Durchschnittswerten kommen. Im anderen Fall, dass jeder Knoten für sich selbst betrachtet wird, kann es passieren, dass die Datenbasis für den Operator auf diesem Knoten sehr dünn ist. Eine einzelner Ausreißer in diesen Daten würde zu unrealistischen Vorhersagen führen.

Ziel ist, die optimale Anzahl der zu betrachtenden Cluster zu bestimmen. Wenn man das System von außen betrachtet, ist die Anzahl der zu betrachtenden Cluster offensichtlich. Sie entspricht der Anzahl von sich unterschiedlich verhaltenden Systemen in den Knotenkandidaten. Die Schwierigkeit ist jedoch, diese Anzahl innerhalb des Systems zu bestimmen. Es wäre nun zu evaluieren, wie diese Anzahl berechnet werden kann.

7.3 Map/Reduce und NexusDS

Für die Implementierung des Knotenleistungsclusterings wurde in Abschnitt 5.5 das Map/Reduce-Konzept eingeführt. Das Ziel von Map/Reduce ist ähnlich dem von NexusDS. Beide Frameworks bieten Möglichkeiten zur Berechnung mittels eines verteilten Ansatzes. Während NexusDS flexibel ist, wie die Verteilung und Abfolge von Operationen aussieht, sind die Reihenfolge und das Format bei Map/Reduce fest vorgegeben. Map/Reduce arbeitet immer auf Schlüssel-Wert-Paaren und Daten, die parallel verarbeitet werden können.

Es wäre nun interessant, das Map/Reduce-Paradigma in NexusDS verfügbar zu machen. Dafür müssten Operatoren geschaffen werden, die die Funktionen der verschiedenen Schritte von Map/Reduce übernehmen. Wie in Abschnitt 5.5.2 erläutert, werden bei Map/Reduce die Daten blockweise auf Map-Knoten verteilt, die dann für die im Block enthaltenen Key-Value-Paare nacheinander jeweils die Map-Funktion ausführen. Für diesen ersten Schritt werden zwei Operatoren in NexusDS benötigt: zum einen ein

Operator, der die Daten in Blöcke aufteilt und an verschiedene Knoten weiterreicht, und zum anderen ein Map-Operator, der die Blöcke erhält und auf die darin enthaltenen Daten die Map-Funktion anwendet.

Wird auch der Combine-Schritt implementiert, muss hierfür ein Puffer zwischen dem Map- und dem Combine-Operator auf dem Knoten existieren. Sobald ein Block abgearbeitet ist, kann die Combine-Funktion angewandt werden.

Nach dem Map- oder Combine-Schritt müssen die Daten sortiert werden und an die Reducer-Operatoren weitergegeben werden. Für die Reduce-Operation ist zu beachten, dass hierfür alle Ergebnisse für alle Datensätze aus den Datenblöcken vorliegen müssen. Ein Reduce-Operator muss also die Information bekommen, wann er alle Ergebnisse erhalten hat. Dies kann in NexusDS erfolgen, indem der Reducer per Konfiguration mitgeteilt bekommt, von wie vielen Map- beziehungsweise Combine-Knoten er Daten erhält. Er wartet dann, bis er von jedem dieser Knoten das Ende der Daten über eine *Punctation* im Datenstrom mitgeteilt bekommen hat. *Punctations* sind Zusicherungen, die im Datenstrom mit angegeben werden können (siehe auch [Dör09]).

Anhang A

Appendix

A.1 Von SIGAR unterstützte Plattformen

Die Monitoring Middleware unterstützt in Version 1.6.3 laut der Dokumentation[1] die in Tabelle A.1 aufgelisteten Plattformen.

[1] Die Dokumentation ist verfügbar unter http://support.hyperic.com/display/SIGAR/Home

Operating System	Architecture	Versions
Linux	x86	2.2, 2.4, 2.6 kernels
Linux	amd64	2.6 kernel
Linux	ppc	2.6 kernel
Linux	ppc64	2.6 kernel
Linux	ia64	2.6 kernel
Linux	s390	2.6 kernel
Linux	s390x	2.6 kernel
Windows	x86	NT 4.0, 2000 Pro/Server, 2003 Server, XP Vista, 2008 Server, Windows 7
Windows	x64	2003 Server, Vista, 2008 Server, Windows 7
Solaris	sparc-32	2.6, 7, 8, 9, 10
Solaris	sparc-64	
Solaris	x86	8, 9, 10
Solaris	x64	
AIX	ppc	4.3, 5.1, 5.2, 5.3, 6.1
AIX	ppc64	5.2, 5.3, 6.1
HP-UX	PA-RISC	11
HP-UX	ia64	11
FreeBSD	x86	4.x
FreeBSD	x86	5.x, 6.x
FreeBSD	x64	6.x
FreeBSD	x86, x64	7.x, 8.x
OpenBSD	x86	4.x, 5.x
NetBSD	x86	3.1
Mac OS X	PowerPC	10.3, 10.4
Mac OS X	x86	10.4, 10.5
Mac OS X	x64	10.5

Tabelle A.1: Von SIGAR unterstützte Plattformen

Literaturverzeichnis

[BDG+04] M. Bauer, F. Dürr, J. Geiger, M. Grossmann, N. Hönle, J. Joswig, D. Nicklas, T. Schwarz. Information Management and Exchange in the Nexus Platform. Technischer Bericht Informatik 2004/04, Universität Stuttgart: Sonderforschungsbereich SFB 627 (Nexus: Umgebungsmodelle für mobile kontextbezogene Systeme), Germany, Universität Stuttgart, Institut für Parallele und Verteilte Systeme, Verteilte Systeme; Universität Stuttgart, Institut für Parallele und Verteilte Systeme, Anwendersoftware, 2004. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=TR-2004-04&mod=0&engl=0&inst=AS. (Zitiert auf den Seiten 17 und 19)

[BWF+96] F. D. Berman, R. Wolski, S. Figueira, J. Schopf, G. Shao. Application-level scheduling on distributed heterogeneous networks. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM) - Supercomputing '96*, pp. 39–es. Pittsburgh, Pennsylvania, United States, 1996. doi:10.1145/369028.369109. URL http://portal.acm.org/citation.cfm?doid=369028.369109. (Zitiert auf Seite 23)

[BYS+06] T. Bray, F. Yergeau, C. M. Sperberg-McQueen, J. Paoli, E. Maler. Extensible Markup Language (XML) 1.0 (Fourth Edition). First edition of a recommendation, W3C, 2006. URL http://www.w3.org/TR/2006/REC-xml-20060816. (Zitiert auf den Seiten 18 und 88)

[CEB+09] N. Cipriani, M. Eissele, A. Brodt, M. Großmann, B. Mitschang. NexusDS: A Flexible and Extensible Middleware for Distributed Stream Processing. In ACM, editor, *IDEAS '09: Proceedings of the 2008 International Symposium on Database Engineering & Applications*, pp. 152–161. ACM, 2009. URL http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2009-94&engl=0. (Zitiert auf den Seiten 19 und 22)

[CLM10] N. Cipriani, C. Lübbe, A. Moosbrugger. Exploiting Constraints to Build a Flexible and Extensible Data Stream Processing Middleware. In *The Third International Workshop on Scalable Stream Processing Systems*, pp. 1–8. IEEE Computer Society, 2010. URL http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2010-08&engl=0. (Zitiert auf den Seiten 19 und 20)

[DG08] J. Dean, S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008. doi:http://doi.acm.org/10.1145/1327452.1327492. (Zitiert auf Seite 93)

[Dör09] M. Dörr. *Entwurf und Implementierung von Basisoperatoren für Nexus*. Diplomarbeit, Universität Stuttgart: Sonderforschungsbereich SFB 627 (Nexus: Umgebungsmodelle für mobile kontextbezogene Systeme), Germany, 2009. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2776&engl=0. (Zitiert auf den Seiten 41 und 105)

[Fri02] M. Friedman. *Windows 2000 performance guide*. O'Reilly, Sebastopol CA, 2002. (Zitiert auf Seite 88)

[FZ87] D. Ferrari, S. Zhou. An Empirical Investigation of Load Indices for Load Balancing Applications. Technical Report UCB/CSD-87-353, EECS Department, University of California, Berkeley, 1987. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/1987/5990.html. (Zitiert auf Seite 51)

[Geh96] J. Gehring. MARS—A framework for minimizing the job execution time in a metacomputing environment. *Future Generation Computer Systems*, 12(1):87–99, 1996. doi:10.1016/0167-739X(95)00037-S. URL http://linkinghub.elsevier.com/retrieve/pii/0167739X9500037S. (Zitiert auf Seite 25)

[Gra05] L. Grandinetti. *Grid computing : the new frontier of high performance computing*. Elsevier, 1st ed. edition, 2005. (Zitiert auf Seite 46)

[Has09] T. Hastie. *The elements of statistical learning : data mining, inference, and prediction*. Springer Series in Statistics. Springer, New York, 2nd ed. edition, 2009. (Zitiert auf Seite 64)

[HK01] J. Han, M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001. (Zitiert auf den Seiten 59, 60, 62, 63 und 64)

[IEE09] IEEE. IEEE Standard for Prefixes for Binary Multiples. *IEEE Std 1541-2002 (R2008)*, pp. c1 –4, 2009. doi:10.1109/IEEESTD.2009.5254933. (Zitiert auf den Seiten 11 und 51)

[Koc09] M. Koch. *Konzeption und Realisierung einer erweiterbaren Service-Schnittstelle in einer verteilten Umgebung*. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2009. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2913&engl=0. Published: Diplomarbeit: Universität Stuttgart, Institut für Parallele und Verteilte Systeme, Anwendersoftware. (Zitiert auf Seite 30)

[LDR03] D. Lee, J. J. Dongarra, R. S. Ramakrishna. visPerf: Monitoring Tool for Grid Computing. In *In ICCS 2003, Lecture Notes in Computer Science*, Lecture Notes in Computer Science, p. 233–243. Springer Verlag, 2003. doi: 10.1007/3-540-44863-2_24. (Zitiert auf Seite 26)

[Mic10] Microsoft. Monitoring Performance Data (Windows). Technical report, 2010. URL http://msdn.microsoft.com/en-us/library/aa392397(VS.85).aspx. (Zitiert auf Seite 88)

[Ora] Oracle. JDK 6 Java Native Interface-related APIs & Developer Guides. Technical report. URL http://download.oracle.com/javase/6/docs/technotes/guides/jni/index.html. (Zitiert auf Seite 89)

[SV08] H. Sanjay, S. Vadhiyar. Performance modeling of parallel applications for grid scheduling. *Journal of Parallel and Distributed Computing*, 68(8):1135–1145, 2008. doi:10.1016/j.jpdc.2008.02.006. URL http://linkinghub.elsevier.com/retrieve/pii/S0743731508000464. (Zitiert auf Seite 25)

[Tan06] A. Tanenbaum. *Distributed systems : principles and paradigms*. Prentice Hall, Harlow, 2nd ed. edition, 2006. (Zitiert auf Seite 48)

[WF04] P. Walmsley, D. C. Fallside. XML Schema Part 0: Primer Second Edition. W3C recommendation, W3C, 2004. URL http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/. (Zitiert auf Seite 18)

[Whi09] T. White. *Hadoop : the definitive guide*. O'Reilly, Sebastopol CA, 2009. (Zitiert auf Seite 94)

[Wol97] R. Wolski. Forecasting network performance to support dynamic scheduling using the network weather service. In *HPDC '97: Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing*, p. 316. IEEE Computer Society, Washington, DC, USA, 1997. (Zitiert auf Seite 24)

[WSP97] R. Wolski, N. Spring, C. Peterson. Implementing a performance forecasting system for metacomputing. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM) - Supercomputing '97*, pp. 1–19. San Jose, CA,

1997. doi:10.1145/509593.509600. URL http://portal.acm.org/citation.cfm?doid=509593.509600. (Zitiert auf Seite 24)

[Öz99] M. Özsu. *Principles of distributed database systems*. Prentice Hall, Upper Saddle River NJ, 2nd ed. edition, 1999. (Zitiert auf Seite 48)

Alle URLs wurden zuletzt am 31.10.2010 geprüft.

Erklärung

Hiermit versichere ich, diese Arbeit
selbstständig verfasst und nur die angegebenen
Quellen benutzt zu haben.

(Raimund Huber)