Implementation und Evaluation eines Paging-Mechanismus für ein Datenstromverwaltungssystem

Bachelorarbeit

vorgelegt von

Ortwein, Maximilian
Matrikel-Nr. 01/752703

an der

Mathematisch-Naturwissenschaftliche Sektion

Fachbereich Informatik und Informationswissenschaft

1. Gutachter: Jun.-Prof. Dr. Michael Grossniklaus
2. Gutachter: Prof. Dr. Marc H. Scholl

Konstanz, 2015

Maximilian Ortwein
Thomas-Sättele-Str. 33B
78467 Konstanz
01/752703
maximilian.ortwein@uni-konstanz.de
Implementation und Evaluation eines Paging-Mechanismus für ein Datenstromverwaltungssystem
Bachelorarbeit
Universität Konstanz, 2015.

1 Danksagung

An dieser Stelle möchte ich mich ganz besonders bei meinem Betreuer Jun.-Prof. Dr. Michael Grossniklaus für die vielen Stunden an Korrektur und die wirklich sehr ausgezeichnete Betreuung bedanken. Vielen herzlichen Dank für die Geduld und die sehr gute und konstruktive Kritik. Ebenso möchte ich mich bei Prof. Dr. Marc H. Scholl für die Zweitkorrektur bedanken.

Ein weiterer Dank gilt meiner Mutter und meinem Vater, die mir das Studium so erst ermöglicht haben und mir immer zur Seite standen. Danke für die Unterstützung von der Schulzeit bis zum ende Meines Studiums. Besonders möchte ich hier meinem Vater für die vielen Stunden Korrektur dieser Arbeit danken. Auch möchte ich meinen Geschwistern und meinen Großeltern danken.

Zuletzt möchte ich mich bei meinen Freunden bedanken, die mich während meines Studiums begleitet haben und immer für mich da sind. Danke für die unvergessliche Zeit, für die tollen Momente in meinem Studium und für die Unterstützung und Motivation bei dieser Arbeit.

2 Zusammenfassung

Die vorliegende Arbeit beschäftigt sich mit der Implementation und Evaluation eines Pagingsystems im Datenstrommanagementsystem Niagarino. Es wird dabei der Frage nachgegangen, in wieweit ein Pagingsystem eine Steigerung der Effizienz von Niahgarino bewirkt. Dabei werden andere Systeme auf die Existenz eines ähnlichen Pagingsystems hin untersucht. Des weiteren wird die Implementation des Pagingsystems in Niagarino vorgestellt und evaluiert. Im Zentrum der Evaluation steht die Frage nach dem Performanzgewinn, der durch das Paging erzielt werden konnte. Durch das Paging soll häufiges *Threadswitching* verringert werden und so die Performanz gesteigert werden. Dabei zeigte sich, dass durch das Paging ein deutlicher Performanzgewinn erzielt werden konnte ohne das der Speicherverbrauch des Systems signifikant angestiegen ist. In einem Vergleich mit NiagaraST konnte allerdings auch gezeigt werden, dass das Paging nur ein Baustein einer Reihe von Optimierungen in Niagarino ist.

Inhaltsverzeichnis

Kapitel 1

Einleitung

500 Terrabyte an Daten pro Tag, das ist das Datenvolumen das alleine auf Facebook 2012 erzeugt wurde[1]. Dazu kommen Daten bei Twitter und anderen Onlinediensten, Daten der Verkehrsüberwachung und der Flugüberwachung und Vieles mehr. All das lässt sich unter dem Begriff „Big Data“ zusammenfassen. Big Data beschreibt neben dem Datenaufkommen auch die veränderten Methoden zur Auswertung dieser immensen Datenmengen. Traditionelle relationale Datenbanken sind diesen Aufgaben mittlerweile nicht mehr gewachsen. Auch sich kontinuierlich verändernde Daten, sogenannte Streamingdaten, fallen unter den Begriff Big Data. Da Datenströme nicht mit relationalen Datenbanksystemen verwaltet werden können, wurden Datenstrommanagementsysteme (DSMS) entwickelt. Diese Systeme sind in der Lage, kontinuierliche Anfragen in Sprachen, ähnlich wie der relationaler Datenbanksysteme, zu stellen.
Mit Niagarino wurde an der Universität Konstanz ein neues Datenstrommanagementsystem entwickelt, dessen Schwerpunkt auf der Analyse von Twitterdaten liegt. Eine wichtige Eigenschaft des Systems ist es, dass die Daten möglichst performant verarbeiten werden, da es sich um ständig veränderliche und meist auch zeitkritische Daten handelt. Performanzoptimierungen sind ein wichtiger Bestandteil der Implementation eines Datenstrommanagementsystems. Diese Bachelorarbeit behandelt die Implementation eines Paging-Mechanismus zur Gewinnung von höheren Performanceeffekten. Insbesondere widmet sie sich der Frage, ob und welchen Effekt das Paging auf die Laufzeit und den Speicherverbrauch des Datenstrommanagementsystems hat. Zu diesem Zweck wurden auch andere Datenstrommanagementsysteme in Bezug auf die Behandlung von Pagingsystemen hin untersucht. Zudem wurden verschiedene Überlegungen zur Page und zur Verwaltung der Pages angestellt und diese mit der Frage nach der Effektivität verknüpft. Darüber hinaus wurde die Fragestellung untersucht, wie sich Niagarino mit Paging im Vergleich zu anderen Datenstrommanagementsystemen verhält.

Im folgenden Kapitel wird der aktuelle Stand der Forschung dargelegt. Dabei wird eine Auswahl von bestehenden Datenstrommanagementsystemen vorgestellt und untersucht. Ein besonderes Augenmerk fokussiert insbesondere die Frage, welche bestehenden Systeme ähnliche Pagingsysteme wie Niagarino implementiert haben.

Das dritte Kapitel widmet sich der Architektur von Niagarino. Insbesondere wird in diesem Zusammenhang die Frage beleuchtet, was die Bedeutung von

[1] http://t3n.de/news/facebook-big-data-gigantische-410203/

Pagingsystemen darstellt und in welchem Verhältnis es für mögliche Performancegewinne steht. Zudem widmet sich dieses Kapitel der Fragestellung, wie ein Pagingsystem in Niagarino eingebunden werden kann.

Das vierte Kapitel beschreibt die Implementation des Pagingmechanismus in Niagarino. Dabei werden zu Beginn mögliche Lösungsansätze vorgestellt. Dann folgt die realisierte Implementierung der Page. Des Weiteren wird erläutert, wie der Pagingmechanismus in das bestehende System modular eingebunden werden konnte und welche Ausnahmen bei den Operatoren beachtet werden mussten.

Kapitel 5 beschreibt die Evaluation des Pagingsystems. In diesem Kapitel wird auch die Kernfrage zum Performancegewinn durch das Paging behandelt. Dazu wird zunächst der Versuchsaufbau der Evaluation beschrieben und die Evaluationsfragen erläutert. Anschließend folgt eine erste Evaluation, um Hinweise auf eine Tendenz der Laufzeit zu bekommen. Diese ersten Evaluation dient darüber hinaus auch der Verfeinerung des Versuchsaufbaus für eine abschließende Evaluation und dem Erkennen möglicher Fehler. Dieser erste Evaluation folgt eine zweite abschließenden Evaluation, die sich insbesondere mit Fragen zur Performance durch das Paging beschäftigt. Weiter wird das Verhalten des Pagings bei sehr großen Pagegrößen untersucht und eine Analyse des Speicherverbrauchs durchgeführt. Der letzte Teil des Kapitels widmet sich dann einem Vergleich von Niagarino und NiagaraST. Hier werden die Laufzeiten und der Speicherverbrauch verglichen. Dies dient nicht zuletzt dazu, Niagarino besser in bestehende Systeme einordnen zu können.

Kapitel 2

Stand der Forschung

Im Folgenden wird zunächst der Frage nachgegangen, welche bestehenden Systeme mit Niagarino vergleichbar sind. Im nachstehenden Kapitel werden mit *STREAM*, *Aurora* und *NiagaraST* drei wichtige Vertreter aus dem Bereich der Datenstrommanagementsysteme vorgestellt. Als vierter Vertreter aus diesem Bereich wird zudem *Odysseus* als neben PIPES[2] bisher einzige Entwicklung einer deutschen Universität in diesem Bereich untersucht. Insbesondere erscheint es für diese Arbeit interessant, ob sich mit dem Paging-Mechanismus von *Niagarino* vergleichbare Mechanismen in den vorgestellten Systemen finden lassen.

2.1 STREAM

STREAM ist ein 2004 an der Stanford University entwickelter Prototyp eines *Data Stream Management Systems* [4]. Die Grundidee hinter STREAM war es, ein universelles *Data Stream Management System* zu entwickeln. Um dieses Ziel umsetzen zu können, wurde eine neue, an SQL angelehnte, Anfragesprache entwickelt. Diese als *Continous Query Language* (CQL) bezeichnete Sprache sollte die Anforderungen umsetzen, die für Anfragen an Streaming Daten notwendig sind. Die Anfragen in STREAM werden alle in CQL formuliert. Das System übersetzt diese dann in einen Queryplan [4].
Grundsätzlich kennt CQL nur zwei Datentypen. Das sind zum einen Streams und zum anderen Relationen. Streams werden dabei als Sammlung von Zeitstempel-Tupel-Paare definiert. Relationen werden dagegen als Sammlungen von Tupeln mit variablen Zeitstempeln definiert [4]. In STREAM wurden diese beiden Datentypen vereinheitlicht zu Tripeln aus *Timestamp*, dem Tupel und eines *Flag* als Markierung. Dieses *Flag* kann die Zustände „eingefügt“ oder „entfernt“ annehmen. In einem Stream können grundsätzlich nur Tupel mit der Markierung „eingefügt“ sein, während Relationen Tupel mit beiden Markierungen beinhalten können.
Die eingehenden und ausgehenden Verbindungen zwischen den Operatoren in STREAM werden durch Queues realisiert, wobei eine Queue sowohl Streams als auch Relationen beinhalten kann [4].

Für manche Operatoren, wie beispielsweise dem Join-Operator, muss es möglich sein, auf einen gewissen Zustand der Daten aus den unterschiedlichen Streams zugreifen zu können. Der Operator benötigt also die Möglichkeit, eine Art

[2] http://dbs.mathematik.uni-marburg.de/research/projects/PIPES/

Snapshot des Streams zu erstellen und möglichst im Arbeitsspeicher zu speichern. Diese Speichermöglichkeit wird im Paper [4] als *Synopsis* bezeichnet. Jeder Operator, der auf diese Möglichkeit zurückgreift, benötigt demnach eine oder mehrere Synopsen. Die konkrete Implementierung in STREAM versucht dies zu vermeiden, indem verhindert wird, dass einerseits Informationen redundant gespeichert werden und andererseits das Teilen von Synopsen zwischen Operatoren ermöglicht wird. So gibt es in einem Queryplan häufig mehrere Operatoren, die den gleichen Inhalt benötigen. Meist werden Synopsen dazu verwendet, den Inhalt eines `sliding Windows` zu speichern. Dieses Beispiel macht deutlich, weshalb diese Synopsen grundsätzlich nötig sind.
Eine weitere wichtige Optimierung, die in STREAM umgesetzt wurde, ist das Scheduling der Operatoren. Dieses kann großen Einfluss auf die Laufzeit und den Speicherverbrauch haben [4]. Die klassische Strategie ist hier das *First in First Out* Prinzip. Im Paper [4] wird auch eine sogenannte *Greedy Scheduling* Strategie vorgestellt, die das Wachstum der Queue deutlich reduziert. Die *Greedy* Strategie versucht dabei, dem Operator den Vorzug zu geben, der am meisten Elemente einer Queue verarbeiten kann und damit am meisten Speicherplatz freiräumen kann. Dadurch ist es möglich, die Queues möglichst klein zu halten. Diese Strategie funktioniert jedoch nicht immer. So kann es durchaus möglich sein, dass eine Gruppe von Operatoren in einer Zeiteinheit mehr verarbeiten kann, als ein einzelner Operator. Dies wird von der Greedy Strategie jedoch nicht berücksichtigt [4].
Für STREAM wurde deshalb der sogenannte *chain-scheduling* Algorithmus entwickelt [4]. Dieser stellt eine Kombination aus *Greedy* und *First in First out* dar. Dabei werden die Operatoren in Gruppen zusammengefasst, den sogenannten *chains*. Innerhalb dieser Gruppen wird der FIFO Algorithmus verwendet. Die Gruppen untereinander werden durch den *Greedy*-Algorithmus geschedult. Dadurch wird berücksichtigt, dass Gruppen von Operatoren mehr verarbeiten können als einzelne Operatoren.
Ein sehr interessanter Ansatz stellt die *Adaption* dar. Dieser Ansatz hat zum Ziel, dass sich die Querys selbst optimieren und somit dafür sorgen, dass sich Speicherverbrauch und CPU-Last auf ein Optimum reduzieren. Hierzu setzen drei Strukturen diese Adaption um [4]: Profiler, Reoptimizer und Executor.
Dem Profiler wird dabei vom Reoptimizer mitgeteilt, welche Werte er benötigt, um einen Operator zu optimieren. Der Profiler erstellt diese Statistiken anhand der Ausführung des Plans und liefert dem Reoptimizer die entsprechenden Statistiken. Daraufhin erstellt der Reoptimizer ein optimales Verhalten, das von den Operatoren adaptiert werden soll. Der Executor greift auf die entsprechenden Informationen des Reoptimizers zu und sorgt für eine Umsetzung in den Operatoren. In STREAM wird dieses System beispielsweise durch den sogenannten StreaMon umgesetzt [4].

Zu Monitortingzwecken wurde ein grafisches Benutzerinterface implementiert. Es stellt den Queryplan als Pipeline dar und bietet die Möglichkeit, die einzelnen Elemente des Queryplans zu untersuchen und zu überwachen. Zudem

lässt sich hier der Speicherverbrauch für die Synopsis anzeigen. Des Weiteren ist es auch möglich, über das GUI einzelne Parameter zur Laufzeit der Operatoren nachzubessern. Darüber hinaus lassen sich auch einzelne Plots zu interessanten Daten anzeigen.

2.1.1 CQL

Wie oben erwähnt, wurde mit STREAM die Anfragesprache CQL eingeführt, die sich an Streamingdaten richtet. CQL wird in STREAM zum Schreiben von Queryplänen verwendet und ist eng mit der aus relationalen Datenbanken bekannten Sprache SQL verwandt.
In CQL werden grundsätzlich dei Typen von Operatoren verwendet [4]:

- Operatoren, die eine Relation konsumieren und als Output auch eine Relation produzieren.
- Operatoren, die einen Stream konsumieren und eine Relation als Output produzieren.
- Operatoren, die eine Relation konsumieren und einen Stream als Output produzieren.

Die Tabelle 1 [4] zeigt die Operatoren die in STREAM durch CQL dargestellt werden können.

Name	Operator Type	Description
select	relation-to-relation	Filters elements based on predicate(s)
project	relation-to-relation	Duplicate-preserving projection
binary-join	relation-to-relation	Joins two input relations
mjoin	relation-to-relation	Mulitway join
union	relation-to-relation	Bag union
except	relation-to-relation	Bag difference
intersect	relation-to-relation	Bag intersection
antisemijoin	relation-to-relation	Antisemijoin of two input relations
aggregate	relation-to-relation	Perform grouping and aggregation
dublicate-eliminate	relation-to-relation	Performs dublicate elimination
seq-window	stream-to-relation	Implements time-based, tuple-based and partitioned windows
i-stream	relation-to-stream	Implements Istream semantics
d-stream	relation-to-stream	Implements Dstream semantics
r-stream	relation-to-stream	Implements Rstream semantics

Tabelle 1. Operatoren in STREAM

Für Relation-zu-Relation-Operatoren werden in CQL normale SQL-Konstrukte verwendet [4]. Da es sich um relationale Daten handelt, ist dieser Ansatz problemlos möglich.

Ein Stream-zu-Relation-Operator ist im Prinzip ein Window. In CQL wird deshalb die Spezifikation aus SQL-99 für ein `sliding Window` verwendet [4]. In der Tabelle sind die Windowtypen dargestellt.

Insgesamt stehen drei Relation-zu-Stream-Operatoren zur Verfügung: „i-stream“ (für insert Stream), wird verwendet wenn neue Tupel in die Relation eingefügt wurden, „d-stream“ (für delete Stream) findet dann seine Anwendung, wenn Tupel aus einer Relation entfernt werden müssen. „r-stream“ (relation Stream) wird zur Umwandlung einer Relation in einen Stream verwendet [4].
Es zeigt sich, dass STREAM zwei besonders interessante Konzepte beinhaltet. Zum einen ist dies die Einführung der CQL als Anfragesprache für Streams. Der zweite besonders interessante Ansatz stellt das Konzept der Adaption dar. Die Möglichkeit, dass sich Querys selbst optimieren, ist insbesondere im Bereich von großen Datenmengen, wie sie in Form von Streams häufig auftreten, besonders wichtig.

2.2 Odysseus

Odysseus ist ein an der Universität Oldenburg entwickeltes Datenstrommanagementsystem. Odysseus selbst ist ein Framework, das die Grundfunktionalitäten eines Event-Processing-Systems implementiert, zeichnet sich aber durch seinen modularen Aufbau sowie die Möglichkeit aus, sich an jeden möglichen Use-Case anzupassen [3].
Odysseus ist ähnlich wie eine Client-Server-Architektur implementiert. Bei Odysseus wird der sogenannte Core, der die Aufgaben durchführt, eine Schnittstelle zum Odysseus Studio, sowie eine Webschnittstelle unterschieden [3].

Das Odysseus Studio setzt auf das Eclipse Framework auf und zeichnet sich durch einen Editor mit Syntax-Highligting für Odysseus-Script, PQL sowie CQL aus. Des Weiteren besitzt das Odysseus Studio einen grafischen Editor, der im Hintergrund PQL Scripte erzeugt. Der grafische Editor arbeitet mit einem Pipeliningsystem und Nodes. Weiter besitzt das Odysseus Studio Visualisierungsmöglichkeiten, um die Ergebnisse grafisch darzustellen [3].

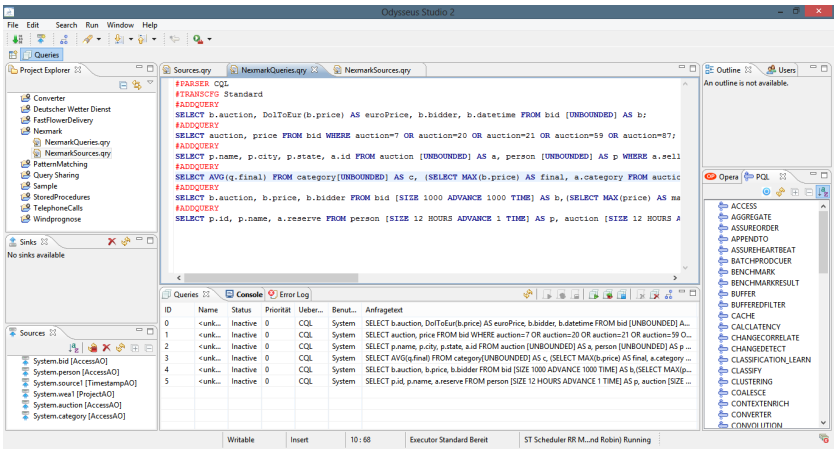

Abb. 1. Screenshot des Odysseusstudios (http://odysseus.informatik.uni-oldenburg.de/index.php?id=85)

Der Odysseus Core unterstützt drei Anfragesprachen [3]: Continous Query Language (CQL), Procedural Query Language (PQL) und die auf Graphen basierende Sprache SPARQL.

Odysseus besitzt laut Paper [3] einen *Quality of Service* Mechanismus und lässt sich auf verteilten Systemen aufsetzen.

In Odysseus ist es möglich, mit sogenannten Snapshots zu arbeiten [6]. Diese sind im Grunde *time-based- sliding Windows*, die anstatt mit Timestamps oder der jeweiligen Uhrzeit mit der Laufzeit von Odysseus in Millisekunden operieren. Die Fenstergröße wird dabei in Millisekunden angegeben. Anschließend wird auf Basis der Laufzeit jeweils ein Snapshot der Daten erstellt. Auf diesen Snapshots können dann — vergleichbar zu relationalen Datenbanksystemen — relationale Abfragen ausgeführt werden [6]. Der Vorteil dieser Snapshots gegenüber Windows anhand von Timestamps aus den Tupeln ist, dass das System resistent gegen große Zeitsprünge der Daten ist. Laut Paper [6] ist es durch diese Snapshots zudem möglich, größere Mengen an Daten mit einer höheren Frequenz durch das System verarbeiten zu lassen, ohne dass dies einen Einfluss auf das Ergebnis hat [6].

Das System ist durch seine Architektur sehr leicht erweiterbar und insbesondere auf Erweiterungen ausgerichtet. Um neue Operatoren zu schreiben, genügt eine Implementierung einer abstrakten Klasse. Ausdrücklich muss eine in der abstrakten Klasse enthaltene Methode implementiert werden. Die so entwickelten logischen Operatoren, stehen dann auch beispielsweise in PQL oder Odysseusscript zur verfügung [6].

2012 wurde Odysseus von der Universität Oldenburg zur Implementierung einer Lösung für die `Grand Challenge der Conference on Distributed Event-Based Systems` (DEBS) benutzt. Bei dieser Lösung kam die Idee des Einsatzes von Snapshots zur Bewältigung der Aufgaben.

Aussagen zu möglichen Vorteilen gegenüber der Verwendung von Datenstrommanagementsystem wie STREAM oder NiagaraST sind nicht möglich. Zum einen beziehen sich nur wenige wissenschaftliche Publikationen auf Odysseus. Zum anderen führt das System keine neuen Ansätze ein. So suggeriert schon allein die Aussage, dass es sich bei Odysseues um ein erweiterbares und modulares Framework handelt, dass Odysseus lediglich die Grundfunktionalitäten eines Datastream-Management-Systems besitzt [3]. Zudem vermittelt diese Aussage, dass das System nicht zielgerichtet ist. Es macht eher Eindruck eines Frameworks, das dazu verwendet werden kann, eigene Operatoren oder Ansätze zu implementieren.
Das Odysseus Studio kann jedoch als ein gelungener Ansatz zur grafischen Formulierung und zur Visualisierung von Ergebnissen betrachtet werden. Zusammenfassend eignet sich Odysseus weniger als ein vollwertiges Datastream-Management-System. Es handelt sich vielmehr um ein Framework. In dieser Funktion wurde es auch 2012 auf der DEBS verwendet.

2.3 AURORA

Aurora stellt ein Datenstrommanagementsystem dar, das im Jahr 2002 von der Brown University zusammen mit der Brandeis University und dem M.I.T. veröffentlicht wurde. Die Motivation zur Entwicklung von Aurora liegt einerseits in der fehlenden Möglichkeit, klassischer Datenbanksysteme mit Streaming-Daten umzugehen. Andererseits können klassische Datenbankensysteme nicht immer die Bedingungen erfüllen, die von Streams gefordert werden [5]. So handelt es sich bei Streamingdaten in der Regel um asynchrone Daten. Zudem sind klassische Datenbanksysteme nicht auf Echtzeitanfragen ausgelegt.
Operatoren und Streams werden in Aurora durch das sogenannte „Boxes and Arrows“ Paradigma [5] repräsentiert. Dabei repräsentieren Boxen die Operatoren und Pfeile die Streams zwischen den Operatoren, aber auch von einer Datenquelle zu einem Operator und von einem Operator zu einer Zielapplikation. Diese Repräsentation zeigt eine große Nähe zu den bereits vorgestellten Systemen. Des Weiteren besitzt Aurora einen Storage, um eine Datenhistorie zu halten. Damit können beispielsweise relationale Anfragen getätigt werden [5]. Dieser Storage kann mit den Synopsen von STREAM verglichen werden.
In Aurora gibt es acht Operatoren, zu denen auch einige Window-Operatoren gehören. Aurora unterstützt dabei `tumbling`und `sliding Windows` [5].
Im Wesentlichen besteht Aurora aus fünf Bestandteilen: einem Storage, einem Scheduler, einem Router, den Operatoren und dem Quality of Service. Des Weiteren besitzt es Ports für Input und Output. Auf das QoS wird innerhalb dieser Arbeit nicht weiter eingegangen.
Alle Tupel werden durch den Router sortiert. So werden eingehende Tupel an den Storage weitergeleitet. Tupel, die von einem Operator verarbeitet wurden, werden durch den Router entweder an den Output oder an den Storage weitergeleitet. Der Storage besteht aus zwei Teilen, einem Buffer und einem persistenten Storage. Im Storage werden alle Tupel zwischengespeichert bzw. auch in den

persistenten Storage verschoben, falls dies notwendig wird. Der Scheduler entscheidet dabei, welcher Operator zu welchem Zeitpunkt welche Tupel bekommt und richtet sich nach den Kapazitäten der Operatoren und der Prioritäten der Tupel.
Der persistente Storage ist für die sogenannten Connection-Points notwendig. Das sind Punkte in der Streamprocessing „Pipeline“, an welchen immer eine bestimme Historie der Daten abgerufen werden kann. An diese „Punkte“ kann ein weiterer Queryplan angefügt werden, der die bis dahin verarbeiteten Daten weiter verarbeitet. Sowohl im Buffer als auch im persistent-Store werden die Tupel in Queues organisiert. Auch der Scheduler bezieht sich auf ganze Queues.
Als Besonderheit des persistent-Storage zeigt sich, dass er — im Gegensatz zu einem Buffer, welcher im RAM liegt — auf der Festplatte abgelegt wird [5]. Wie auch schon die beiden oben beschriebenen Systeme besitzt Aurora ein grafisches Benutzerinterface [5]. In diesem Interface wird das Boxen-und-Pfeile-Prinzip umgesetzt. So kann man Boxen zu einer Superbox gruppieren und sich so den Queryplan zusammenstellen. Damit lassen sich auch sehr komplexe Querypläne übersichtlich graphisch darstellen.
Des Weiteren bietet das Interface die Möglichkeit, an bestimmten Punkten zu Debugging-Zwecken Einblick in den aktuellen Ablauf zu haben. Zudem kann die Pipeline in einzelnen Schritten durchlaufen werden [5]. Um es Aurora zu ermöglichen, auf verteilten Systemen zu laufen, wurde mit Medusa ein Projekt erschaffen, das Aurora als Streamprocessor verwendet, wodurch eine Verteilung von Querys auf mehrere Systeme über das Netzwerk möglich wird [10].
Die Ergebnisse der beiden Projekte Aurora und Medusa wurden im Projekt Borealis zusammengeführt und werden unter dieser Bezeichnung weiter entwickelt [1].
Zusammenfassend bietet Aurora zwei besondere Features. Das eine ist das Quality of Service. Das zweite Feature ist die besondere Art und Weise der Lastenverteilung und der Reduzierung von Speicherverbrauch, die zu einer optimalen Auslastung der Operatoren führen. Dabei ist insbesondere die Zuteilung der Queues durch den Scheduler, die sich an der Auslastung der Operatoren orientiert, ein spannender Ansatz. Eine ähnliche Zielsetzung wird ebenfalls mit der Einführung von Pages in Niagarino verfolgt.

2.4 NiagaraST

NiagaraST stellt ein Datenstrommanagementsystem dar, welches an der Portland State University entwickelt wurde. Es basiert auf dem Source Code des Niagara Systems, welches an der University of Wisconsin-Madison entwickelt wurde. Bei Niagara handelt es sich um ein Anfragesystem für Suchanfragen im Internet und benutzt als Anfragesprache XML bzw. XML-QL [7].
NiagaraST verwendet — wie auch Niagara — XML als Anfragesprache bzw. zum Aufbau der Querypläne. Im Grunde stellt NiagaraST eine Erweiterung des Niagara Internetquerysystems um Anforderungen für Stream-Management-Systeme dar. Dabei wurde Niagara insbesondere um sogenannte „Punctuations“

und Windows erweitert [9].
NiagaraST dient unter anderem der Auswertung von Verkehrsdaten aus dem Portland Oregon Transportation Archive Listening (PORTAL). PORTAL gewährt den Zugriff auf alle Verkehrsdaten im Großraum von Portland.
Im Gegensatz zu den bisher vorgestellten Systemen, besitzt NiagaraST kein grafisches Interface. Da die Anfragen alle in XML gestellt werden, ist dies auch nicht notwendig. Eine weitere Besonderheit von NigaraST ist im Vergleich zu den bisher vorgestellten Systemen die Implementierung von sehr vielen Operatoren, was eine große Anzahl an verschiedenen Anfragen ermöglicht.
Die Operatoren in NiagaraST sind alle über Streams verbunden. Damit ähnelt das Konzept den bereits vorgestellten Lösungen und erinnert zudem an die oben ausgeführten Pipelinesysteme.
Die Streams zwischen den Operatoren haben zwei Aufgaben. Zum einen werden darüber die Tupel von Operator zu Operator transportiert. Darüber hinaus werden aber auch Steuerbefehle (sogenannte *Control Messages*) zwischen den Operatoren ausgetauscht. Diese ermöglichen die Kommunikation der Operatoren untereinander. Ein End-of-Stream (EoS) ist eine solche Information, die einem Operator signalisiert, dass er keine weiteren Tupel mehr zu erwarten hat. Zudem werden ein Control Flag und die Punctuations zwischen den Operatoren transportiert.
NiagaraST unterscheidet verschiedene Arten von Streams, wie beispielsweise den Source-Tuple-Stream oder den Sink-Tuple-Stream. Die Basis stellt jedoch der Page-Stream dar. Wie aus der Bezeichnung schon hervor geht, werden im Page-Stream keine Tupel und Steuerbefehle verschickt, sondern nur Pages. Alle anderen Streams, die über dem Page-Stream liegen, dienen nur als Schnittstellen, die den Operatoren einen tupelbasierten Stream simulieren. Einen anderen Ansatz verfolgt Niagarino: Hier wird im Gegensatz zu NiagaraST ein genereller Stream verwendet, der Stream-Objekte transportiert, die sowohl Tupel, Pages oder auch Steuerbefehle sein können.
Ein Page-Stream in NiagaraST besteht aus zwei Page-Queues. Die eine unterstützt den Downstream, also den Stream, der von einem Operator konsumiert wird. Die zweite unterstützt den Upstream. Aus diesem Stream verarbeitet der Operator seine Tupel und Steuerbefehle. Damit besteht zwischen jedem Operator eine bidirektionale Verbindung, die vor allem für Steuerbefehle verwendet wird. Eine Page stellt einen Buffer für Tupel dar. Hierbei wird zunächst eine Anzahl an Tupel angesammelt, welche dann als Ganzes verschickt wird.
Dem besonderen Design der Pages ist geschuldet, wie Steuerbefehle in den Streams transportiert werden, da diese nur Pages akzeptieren und Pages ein Buffer für Tupel sind. Eine Page in NiagaraST besteht aus einem Array von Tupel, zwei Pointer (einen auf die aktuelle Position und einen auf den Anfang des Buffers), einem Control-Flag, einem String für den Steuerbefehl und einem Marker. Der Marker zeigt an, ob sich die Page im Schreib- oder Lesemodus befindet.
Das Design der Page ermöglicht es grundsätzlich, zwei verschiedene Pages zu erzeugen. Das eine ist die Page, die Tupel beinhaltet. Die zweite ist eine Control-Page, die nur die Steuerbefehle und Flags beinhaltet. Dazu wird über eine Metho-

de eine Control-Page erstellt, welche die Besonderheit aufweist, dass das Tupel-Array auf null gesetzt ist. Dies ermöglicht NiagaraST, auch Steuerbefehle in einem Page-Stream zu transportieren.
Diese Systematik macht die Implementation einer Page etwas komplizierter, weswegen für Niagarino ein etwas anderer Weg gewählt wurde. Dies wird besonders aus der Perspektive deutlich, dass eine Page auch in der Lage sein muss mitzuteilen, ob sie Tupel enthält oder nicht. Dass die Größe der Page eine feste Größe ist, stellt eine Gemeinsamkeit beider Systeme dar, auf die weiter unten noch einmal eingegangen wird. Die Größe der Page wird dabei durch die statische Variable $PAGE\_SIZE$ in der Page hartcodiert.
Von allen vorgestellten Systemen stellt NiagaraST das System dar, das Niagarino am ähnlichsten ist. Insbesondere weist das implementierte Pagingssystem in Niagarino eine große Nähe zu NiagaraST auf. Der entscheidende Unterschied liegt im Design der Streams. Während in Niagarino ein genereller Stream verwendet wird, kommt in NiagaraST ein stark spezialisierter Stream zum Einsatz. Interessant erscheint in diesem Zusammenhang, den Ansatz nur auf Paging zu setzen was dazu führt, dass letztendlich nur über Pages zwischen den Operatoren kommuniziert wird. Dabei mag es auf den ersten Blick etwas irritierend sein, dass eine Tupel-Page nur einen Container für Steuerbefehle darstellt und keinerlei Tupel enthält. Insgesamt wird durch dieses Design die Flexibilität der Art von Objekten, die zwischen Operatoren transportiert werden, stark eingeschränkt.

Kapitel 3

Niagarino

Niagarino stellt ein Datenstrommanagementsystem dar und wird am Lehrstuhl für Datenbanken und Informationssysteme an der Universität Konstanz entwickelt. *Niagarino* dient als Forschungssystem für Forschungsfragen rund um Datenströme und wird ständig weiterentwickelt. Eine typische Beispielanwendung für *Niagarino* sind Anfragen rund um den Onlinedienst Twitter[3]. Insbesondere wurde ein Event-Detection-System für Twitter mit Hilfe von Niagarino realisiert. Ursprünglich wurde Niagarino als Simulationsplattform für datengetriebenen Windows, sogenannte Frames, entwickelt. Frames stellen in diesem Zusammenhang Windows dar, die anhand der Werte in den Tupeln gebildet werden und nicht durch zeitbezogenen Einheiten. Ein Beispiel für diesen Zusammenhang ist die Ermittlung der Durchschnittsgeschwindigkeit von vorbeifahrenden Autos. Das Ziel von Frames ist es durch die dynamischen Windows exakter an die realen Werte zu kommen als mit herkömmlichen Windows.

3.1 Architektur von Niagarino

Niagarino ist komplett in der Programmiersprache *Java* geschrieben und benötigt zur Ausführung die Java Laufzeitumgebung (JRE)[4] in der Version 8, da einige Operatoren bereits die λ-Funktionen von Java 8 verwenden. Im Gegensatz zu den meisten vorgestellten Systemen ist keine GUI implementiert. Die Architektur von Niagarino erinnert deutlich an NiagaraST, aus welchem es auch hervor ging.
Es sind bereits eine Vielzahl an vordefinierten Operatoren implementiert, unter ihnen auch Window- und Frame-Operatoren. Ähnlich wie in einem relationalen Datenbanksystem sind auch in Niagarino Selektions- und Projektionsoperatoren implementiert. Daneben steht ein Derive-Operator zur Verfügung, mit dessen Hilfe man Funktionen auf Tupel anwenden kann. Des Weiteren stehen auch Join und Aggregationsoperatoren zur Verfügung. Für die Ausgabe sind Sink-Operatoren zuständig, für die Eingabe Source-Operatoren.
Ein Queryplan kann aus mehreren Sink- und Source-Operatoren bestehen und wird in Java umgesetzt. Querypläne in Niagarino sind nichts anderes als ein Graph $< O, S >$, der Operatoren als Knoten und Streams als Kanten darstellt. In jedem Queryplan muss dabei angegeben werden, welche Operatoren verwendet werden und welche Operatoren über Streams mit welchen verbunden sind. Im Quellcode sieht das dann zum Beispiel so aus:

[3] http://www.twitter.com
[4] http://www.oracle.com/technetwork/java/javase/downloads/index.html

```
1   // scan
2   final Operator scan = new Scan(ScanSelectPrint.SCHEMA,
        ScanSelectPrint.IN_FILENAME);
3   // selection
4   final Operator selection = new Selection(scan.
        getOutputSchema(), new TimeEqualityPredicate(1, 15, 45,
        00));
5   // print
6   final Operator print = new Print(selection.getOutputSchema
        (), printSchema, out);
7   // physical query plan
8   final PhysicalQueryPlan plan = new PhysicalQueryPlan();
9   // operators
10  plan.addOperator(scan, OperatorType.SOURCE);
11  plan.addOperator(selection);
12  plan.addOperator(print, OperatorType.SINK);
13  // streams
14  plan.addStream(scan, selection);
15  plan.addStream(selection, print);
16  return plan;
```

Zunächst werden die Operatoren erzeugt. In diesem Plan sind es ein Einlese-Operator (`scan`), ein Selektions-Operator, der in diesem Fall alle Tupel mit einem bestimmten Timestamp auswählt, und ein Print-Operator, der das Ergebnis in eine Datei schreibt. Als nächstes werden der eigentliche Queryplan erzeugt und die notwendigen Operatoren hinzugefügt. Abschließend werden noch die Streams definiert. Die Streams selbst stellen eine Queue (Warteschlange) dar in die einerseits vom Quell-Operator geschrieben wird und von der andererseits vom Zieloperator gelesen wird. Der Queryplan stellt dabei einen bidirektionalen Graph dar. Es ist zudem möglich, dass der Ziel-Operator Nachrichten an den Quell-Operator schickt.

Streams nehmen in *Niagarino* nur zwei verschiedene Typen (bzw. drei Typen mit Paging) von Stream-Elementen auf. Der erste Typ sind Tupel, also die eigentlichen Daten, die durch das System verarbeitet werden sollen. Der zweite Typ sind Steuerbefehle, wie beispielsweise ein End-Of-Stream. Beide Elementtypen leiten von `StreamElement` ab und können im Stream transportiert werden. Hier wird der Unterschied zu *NiagaraST* besonders deutlich. In *NiagaraST* können Streams nur Pages transportieren, wobei eine Page sowohl ein Container für Metadaten (Steuerbefehle) als auch ein Container für Tupel sein kann. Ein weiterer wesentlicher Unterschied zwischen *NiagaraST* und *Niagarino* ist, dass *Niagarino* nur geordnete Streams erwartet und verarbeiten kann, während *NiagaraST* auch mit ungeordneten Streams zurecht kommt.

Die folgende Grafik stellt einen etwas komplexeren Queryplan grafisch durch Boxen und Pfeile dar. Dieser Plan zeigt, dass es nicht zwingend ein sequenzieller Plan sein muss, sondern dass Operatoren auch parallel ausgeführt werden können. Dabei zeigt die Grafik, dass es zwischen den Operatoren immer zwei

gegenläufige Streams gibt und jeder Operator bis auf Scan und Print immer Streams konsumiert und produziert.

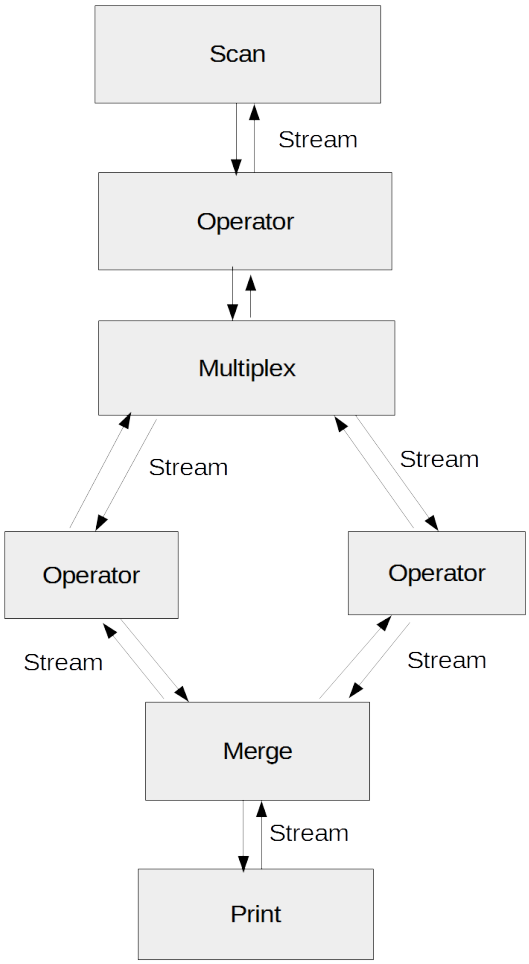

Abb. 2. Ein etwas komplexerer Queryplan als Graphen dargestellt.

Beide *Streamelemente*, sowohl Tupel als auch Control (die Steuerbefehle), müssen das Interface `StreamElement` implementiert haben, um auf dem Stream transportiert zu werden.

Die Operatoren wurden in *Niagarino* sehr generell gehalten. So muss jeder Operator (der Scan-Operator stellt hier eine Ausnahme dar) von einem Abstrakten Operator, dem `AbstractOperator` ableiten. Das macht das Erweitern des Systems um Grundfunktionalitäten relativ einfach, da nur an einer Stelle etwas geändert werden muss und nicht in jedem Operator. Der `AbstractOperator` stellt alle wichtigen Grundfunktionen zur Verfügung, die ein Operator beherrschen muss. Somit ist es auch beim Erstellen neuer Operatoren nicht notwendig, sich um die Streams und die Tupel zu kümmern. Man muss lediglich die Funktionalität des Operators implementieren. Dies geschieht dadurch, dass man die Methode `processTuple` überschreibt.

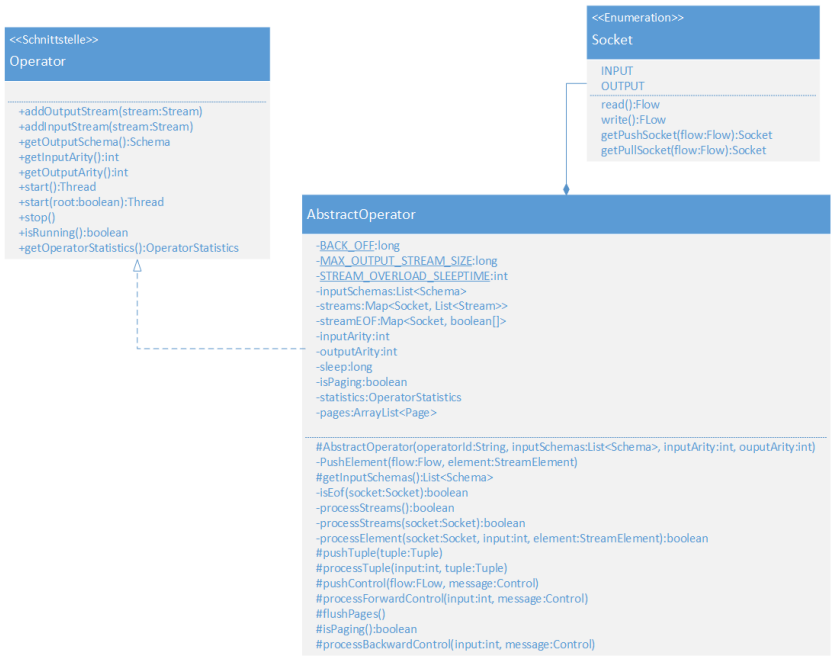

Abb. 3. Der AbstractOperator als Klassendiagramm.

Das folgende Listing zeigt beispielhaft wie sich ein Operator darstellt. Dieser ist in diesem Fall allerdings ohne jegliche Funktionalität. Dieser NoOp-Operator gibt mit der Methode `processTuple` nur Tupel weiter, diese wird dabei von ihm nicht überschrieben.

```
1  public class NoOp extends AbstractOperator {
2    private final Schema outputSchema;
3
4    public NoOp(final Schema schema) {
5      super("NoOp", Arrays.asList(schema), 1, 1);
6      this.outputSchema = schema;
7    }
8
9    @Override
10   public Schema getOutputSchema() {
11     return this.outputSchema;
12   }
13 }
```

Das folgende Listing zeigt die Methode `processTupel` wie sie im Abstract-Operator implementiert ist. Dabei wird das Tupel unverändert weiter gegeben. Aus diesem Grund lässt sich auch die NoOp-Implementierung realisieren. Dabei muss, nachdem ein Tupel verarbeitet wurde, immer ein `pushTuple(Tuple t)` erfolgen.

```
1    protected void processTuple(final int input, final Tuple t) {
2      this.pushTuple(t);
3    }
```

Querypläne werden mit der Klasse `PhysicalQuerplan` erstellt. Die Klasse hat drei Listen, von denen eine Liste die Operatoren enthält, eine enthält die *Sink* und eine gilt den *Source* Operatoren.
Für *Sink* und *Source* muss dabei jeweils der Typ der Methode `addOperator()` übergeben werden. Die Klasse sorgt auch für Verbindungen durch Streams zwischen den Operatoren. Die Methode `execute()` überprüft dabei, ob der Plan vollständig ist und führt ihn dann aus. Dazu werden die Operatoren der Reihe nach gestartet wie sie in der Liste stehen.
Technisch gesehen wird für jeden Operator in einem Queryplan ein neuer Thread erzeugt. Sobald ein Operator keine Daten mehr zu erwarten hat, wird er beendet, wodurch auch sein Thread beendet wird. Beim *Scan-Operator* wird dieser Zeitpunkt dann einfach bestimmt, wenn die Datei, aus der gelesen wird, zu Ende ist. Bei allen anderen Operatoren kommen die Control-Messages zum Einsatz. Diese werden zu dem Zeitpunkt beendet, wenn ein *EoF*[5] statt einem Tupel geschickt wird.

3.2 Warum Paging in Niagarino?

Die Architektur von Niagarino sieht vor, dass auf den Streams zwischen den Operatoren einzelne Tupel oder *Control-Messages* transportiert werden. Dies bedeutet, dass ein Operator, sobald er ein Tupel verarbeitet hat, dieses sofort

[5] End-of-File bzw. End-of-Stream

an den nächsten Operator weiter schickt. Ein großer Nachteil hierbei ist es, dass das Aktivieren und Deaktivieren von Threads ziemlich teuer ist. Dies kann deshalb gerade bei großen Queryplänen zu Performanzeinbußen führen, da ein Operator immer dann kurz aktiv wird, wenn er ein Tupel bekommt. Ab diesem Zeitpunkt wartet dieser Operator darauf, dass ein neues Tupel eintrifft. Dieser Effekt ist für kurze Querypläne nicht sonderlich relevant. Für größere Pläne mit vielen Operatoren ist dies dagegen weit problematischer zu bewerten. Auch Operatoren, wie zum Beispiel ein Aggregations-Operator oder ein Window-Operator, die nicht nur auf einem Tupel sondern auf mehreren Tupeln arbeiten, werden dadurch ausgebremst, da diese mitunter sehr lange darauf warten müssen, bis sie alle benötigten Tupel bekommen.
An diesem Punkt erhält das sogenannte *Paging* seine Bedeutung. Die Idee dabei ist, dass zwischen den Operatoren anstatt eines einzelnen Tupel eine Ansammlung von Tupeln auf dem Stream transportiert werden. Diese Ansammlungen werden als *Page* bezeichnet. Durch die Pages soll erreicht werden, dass Operatoren parallel arbeiten können und die Wartezeit auf neue Tupel auf ein Minimum reduziert wird. Die Idee dahinter ist, dass in der Zeit, in der ein Operator eine Page abarbeitet, sein Vorgänger eine neue Page befüllt und diese dann sofort oder nach einer nur kurzen Wartezeit dem nachfolgenden Operator zur Verfügung stellt. Dadurch wird auch die Anzahl an Thread-Wechseln (also das aktivieren bzw. deaktivieren von Threads) verringert. Ziel des Paging-Mechanismus ist es, die Performanz bei sehr großen Datenmengen und großen Queryplänen erheblich zu verbessern. Ein System, das einen solchen Mechanismus bereits implementiert hatte, ist das bereits oben vorgestellte *NiagaraST*.

Kapitel 4

Implementierung des Pagingmechanismuns

Im vorherigen Kapitel wurde die Motivation für die Implementierung eines Pagingmechanismus eingeführt. Zudem wurde die Umsetzung eines Pagingmechanismus bei NiagaraST vorgestellt. Dieses Kapitel widmet sich den Grundideen eines Pagingmechanismus in Niagarino und stellt für diesen Zusammenhang verschiedene Lösungsansätze vor. Der zweite Teil behandelt die konkrete Implementierung entsprechender Ansätze in Niagarino.

Die Anforderungen an einen Pagingmechanismus sind zum einen ein möglichst geringer *overhead* und schnelle Zugriffszeiten unter den Voraussetzungen, dass erwünschte Effekte durch das Paging nicht relativiert werden. Als zweite Anforderung wird definiert, dass der Pagingmechanismus möglichst generell in Niagarino als zusätzliches Modul implementiert wird. Hinzu kommt, dass keine oder kaum Änderungen an bestehenden Operatoren vorgenommen werden müssen, sodass auch beim Schreiben neuer Operatoren das Paging nicht beachtet werden muss.

4.1 Architekturideen für den Pagingmechanismus

Im Gegensatz zu NiagaraST war das Anliegen, den Stream möglichst generell zu halten und nicht auf einen Typ zu beschränken[6]. Deshalb muss eine Page auch die Klasse `StreamElement` implementieren.
Um den genannten Anforderungen zu entsprechen, wurde einer Implementierung der Page als Array von Tupeln der Vorzug gegeben. Dabei wird um das Array eine Klasse gebaut, die Informationen über die Page preisgibt und beispielsweise zu ihrer Befüllung dient. Darüber hinaus stellt sie Methoden bereit, um Tupel der Page hinzuzufügen und zu entfernen. Die Größe der Page soll hier durch den Anwender anpassbar und nicht in der Page hartgecodet sein. Dazu wurde Niagarino um eine Konfigurationsdatei erweitert, die diese Optionen für das Paging beinhaltet. Gegen eine Fixierung der Page-Größe sprach, dass diese eventuell ein wichtiger Parameter für die Effektivität des Pagingmechanismus sein könnte. Die Page-Größe muss dann eventuell, je nach Größe des Queryplans und der Datenmenge, angepasst werden. Im Kapitel über die Evaluierung des Pagingsystems wird diese Problematik noch einmal aufgegriffen werden.
Eine Idee, die zunächst verfolgt wurde, war die Anzahl an Pages zu begrenzen und die Pages wieder zu verwenden. Dazu sollte es einen Page-Pool geben,

[6] In NiagaraST kann ein Stream nur Pages transportieren.

der eine vorgegebene Anzahl an Pages bei der Initialisierung erzeugt und diese in einer `BlockingQueue` bereithält. Die Operatoren können dann leere Pages vom Page-Pool anfordern und nicht mehr benötigte wieder zurückgeben. Der Page-Pool bereitet diese dann zur Wiederverwendung auf. Abb. 4 zeigt diese Grundidee des Page-Pools.

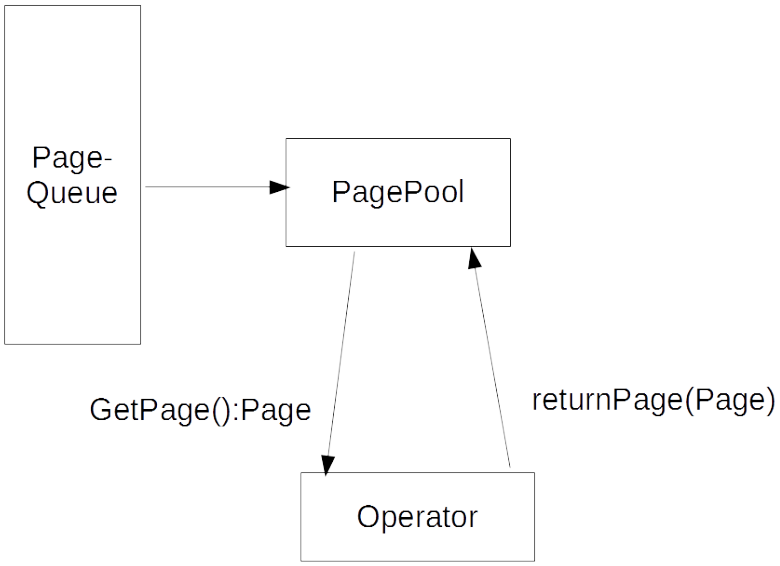

Abb. 4. Schematische Darstellung des Zusammenspiels von Page-Pool und Operatoren

Im Laufe der Evaluation des Pagingmechanismus wurde jedoch die Entscheidung getroffen, auf den Page-Pool zu verzichten, da dieser zum einen durch die `BlockingQueue` einen erhöhten Overhead mit sich bringt. Zum anderen konnten keine wirklichen Vorteile des Page-Pools festgestellt werden. Im Gegenteil: durch die Größe des Page-Pools kommt ein weiterer Parameter hinzu, der eingestellt und auch auf die Daten abgestimmt werden muss. Ist dieser zu klein konfiguriert, drohen im schlimmsten Fall *Deadlocks*, da keine Pages mehr von den Operatoren freigegeben werden können. Ist der Parameter zu groß einge-

stellt, steigt der Speicherverbrauch zu weit an, da mehr Pages erzeugt als jemals gebraucht werden. Der Vorteil der Wiederverwendung von Pages wird dadurch stark entwertet.
Die endgültige Architektur besteht nun aus Pages, die von jedem Operator nach Bedarf erzeugt werden. Sobald diese nicht mehr benötigt werden, werden sie vom *Garbage-Collector* entfernt.

4.2 Implementierung

Die Implementierung des Paging-Mechanismus besteht im Wesentlichen aus zwei Teilen. Der erste Teil stellt die Implementierung und das Design der *Page* dar. Der zweite Teil beschreibt die Einbindung des Paging-Mechanismus in *Niagarino*.
Die `Page` stellt eine Datenstruktur zum Speichern von Tupeln dar. Zudem bringt sie noch eine Reihe von Funktionen, wie beispielsweise das Abfragen der Größe, mit. Die folgende Abbildung zeigt das UML-Diagramm der `Page`.

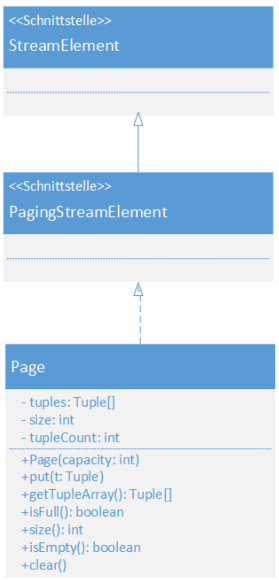

Abb. 5. Die Page

Der eigentliche Speicher für die Tupel stellt ein Array dar, dessen Größe beim Erzeugen der Page festgelegt wird. Als weitere Attribute finden sich ein Zähler für die Tupel, der gleichzeitig auch als Zeiger auf die aktuelle Position im Array fungiert, und die maximale Größe.
Das folgende Listing zeigt die Implementation der Page.

```
1  public class Page implements PagingStreamElement {
2
3    /** The Queue where the tuples are stored. **/
4    private final Tuple[] tuples;
5    /** Size of the Queue. **/
6    private final int size;
7    /**Current count of Tuples. **/
8    private int tupleCount;
9
10   public Page(final int capacity) {
11     this.size = capacity;
12     this.tuples = new Tuple[this.size];
13     this.tupleCount = 0;
14   }
15
16   public final void put(final Tuple t) {
17   this.tuples[this.tupleCount] = t;
18   this.tupleCount++;
19   }
20
21   public final Tuple[] getTupleArray() {
22     return this.tuples;
23   }
24
25   public final boolean isFull() {
26     return this.tupleCount == this.size;
27   }
28
29   public final int size() {
30     return this.tuples.length;
31   }
32
33   public final boolean isEmpty() {
34     return this.tuples.length == 0;
35   }
36
37   @Override
38   public final PagingStreamElement clone() throws
         CloneNotSupportedException {
39     super.clone();
40     // do nothing
41     return this;
42   }
43
44   public void clear() {
45      Arrays.fill(this.tuples, null);
46      this.tupleCount = 0;
47   }
48 }
```

Die Größe der Page wird zentral in einer Konfigurationsdatei für das System festgelegt. Durch den folgenden Eintrag in der *niagarino.properties* wird die Größe der Page gesetzt.

```
#Set here the number of tuples stored in one page
pageSize = 20
```

Durch das generelle Design der Page kann das *Pagingsystem* als ein zuschaltbares Modul angesehen werden. Darüber hinaus kann für jeden Queryplan definiert werden, ob das *Paging* benutzt wird oder nicht. Dazu wird in der Konfigurationsdatei die Option `isPaging = true` gesetzt.
Zum Auslesen der Konfigurationsdatei wurde ein *Properties-Reader* als Singleton implementiert, der die Konfiguration als Objekt liefert.

Bis auf zwei Ausnahmen funktioniert das Zuschalten des Pagings im `AbstractOperator`. Der Operator erzeugt sich eine Page und die verarbeiteten Tupel werden statt auf den Stream in die Page gepusht. Sobald eine Page gefüllt ist, wird diese auf den Stream gepusht. Wichtig hierbei ist die Unterscheidung, ob es sich wirklich um ein Tupel oder um einen Steuerbefehl handelt, denn diese werden gesondert behandelt. Das folgende Listing zeigt die Pushfunktion in der Methode `pushElement` im `AbstractOperator` für aktiviertes Paging.

```
1     if (this.isPaging && element instanceof Tuple) {
2        int i = 0;
3        for (final Stream stream : streams) {
4          if (this.pages.isEmpty() || this.pages.size() <=
               streams.size()) {
5            this.pages.add(new Page(this.pageSize));
6          }
7
8          final Page page = this.pages.get(i);
9          page.put((Tuple) element);
10         this.pages.set(i, page);
11         if (page.isFull()) {
12           stream.pushElement(socket.write(), page);
13           this.pages.set(i, new Page(this.pageSize));
14         }
15         i++;
16       }
17       this.statistics.incrementTupleCounter();
18    }
```

Wichtig ist für diesen Zusammenhang, dass für jeden Stream eine Page angelegt wird. Deshalb wurde der `AbstractOperator` um eine `ArrayList` mit Pages erweitert. Eine Besonderheit stellen Pages dar, die beim Deaktivieren des Operators noch nicht gepusht wurden, weil sie noch nicht ganz gefüllt sind. Hierbei muss die Page gepusht werden, bevor der End-Of-Stream-Steuerbefehl (EoF) weitergereicht wird. Dazu wurde die Methode `processForwardControl` erweitert.

```
1 if (Type.EOF.equals(message.getType())) {
2  if (this.isPaging) {
3   this.flushPages();
4  }
5 }
```

Bevor die EoF-Nachricht weiter gereicht wird, werden noch alle Pages, die der Operator beinhaltet auf die Streams gepusht. Dies wird auf folgendem Weg realisiert.

```
1  protected void flushPages() {
2   final List<Stream> streams = this.streams.get(Socket.OUTPUT);
3   int i = 0;
4   if (!this.pages.isEmpty()) {
5     for (final Stream stream : streams) {
6        stream.pushElement(Socket.OUTPUT.write(), this.pages.get(
             i));
7        this.pages.set(i, new Page(this.pageSize));
8        i++;
9     }
10   }
11 }
```

Diese Methode steht ausschließlich den Operatoren zur Verfügung, die den `AbstractOperator` erweitern.

Die erste Ausnahme für das Zuschalten des Pagings im `AbstractOperator` betrifft den *Scan-Operator*. Dieser leitet nicht vom `AbstractOperator` ab. Deshalb mussten dort die gleichen Schritte implementiert werden wie im `AbstractOperator`. Dazu musste zum einen die Methode `processFile()` dahin gehend geändert werden, dass sie die neuen Tupel in eine Page schreibt und diese auf den Stream pusht, wenn die Page voll ist. Das Behandeln von übrigen Pages zum Ende der Laufzeit des Operators stellt sich im Vergleich zum `AbstractOperator` einfach dar. Dazu musste nur in der `run()` Methode die Page auf den Stream gepusht werden bevor das *EoF* auf den Stream gepusht wird.

Die zweite Ausnahme stellt der *Merge-Operator* dar. Dieser speichert intern Tupel um das Mergen durchzuführen. Dort muss das Verschicken von vollen Pages zur Laufzeit des Operators und das Verschicken von übrigen Pages am Ende der Laufzeit gesondert behandelt werden, da der `ProgressingMerge` Operator die Methode `processForwardControl(final int input, final Control message)` überschreibt da er selbst noch vorhandene Tupel flushen muss. Der Merge-Operator ist der einzige Operator der selbst Tupel hält. Hier musste in der `processForwardControl` Methode ein `flushPages()` eingefügt werden, nachdem alle Tupel geflusht wurden.

Das Einlesen und Verarbeiten einer Page stellt sich insgesamt wesentlich einfacher dar als das Befüllen und Verschicken. Hier muss nur geprüft werden,

ob das ankommende Stream-Element vom Typ Page ist. Ist dies der Fall, lässt man sich von der Page das Tupel-Array geben. Durch eine einfache Iteration über das Array werden dann die Tupel durch den Aufruf der gleichen Methode (`processTuple(Tuple t)`) einzeln verarbeitet, wie wenn das Tupel direkt aus dem Stream kommt.

Das folgende Kapitel widmet sich der Evaluation des Pagingsystems. Die zentrale Frage ist dabei der Effektivität des Pagingsystems gewidmet.

Kapitel 5

Evaluation

Einen wesentlichen Teil dieser Arbeit stellt die Evaluation des implementierten Pagingsystems dar. Eine ausführliche Evaluation soll zeigen, dass das Pagingsystem richtig implementiert wurde und das erwartete Verhalten aufzeigt. Voraussetzung hierzu war der Aufbau einer geeigneten Versuchsumgebung und die Identifikation von interessanten und relevanten Variablen.
Im Zentrum stand die Frage, ob und wie effektiv Niagarino mit Paging arbeitet. Dazu wurden Performance-Messungen nötig, die sich insbesondere auf die benötigte Laufzeit konzentrierten. Von zusätzlichem Interesse war auch die Messung des Speicherverbrauchs. Zur Performance-Messung stellten sich folgende zentrale Fragen:

- Wie verhält sich das System mit verschiedenen Pagegrößen?
- Gibt es besondere Phänomene und wie lassen sich diese erklären?
- Welche Pagegrößen kommen überhaupt in Betracht?
- Gibt es einen optimalen Wert für die Pagegröße?
- Gibt es Zusammenhänge bei der Laufzeit zwischen Pagegröße und Windowgröße?

Neben dem Umgang von Niagarino mit Paging bot sich zudem ein Vergleich mit NiagaraST an, da dieses einen ähnlichen Pagingmechanismus implementiert hat, die Pagegröße jedoch auf 30 Tupel fixiert ist.
Die Evaluation kann damit in vier Abschnitte unterteilt werden. Im ersten Abschnitt wird ein genereller Versuchsaufbau entwickelt und anschließend einer erste Evaluation zugeführt. Diese soll vor allem Hinweise auf ungewöhnliche Phänomene liefern und dient einer Verfeinerung des Versuchsaufbaus sowie einer Identifikation interessanter Variablen. Danach folgt eine Planungsphase, um in einer weiteren Evaluationsphase möglichst genaue und aussagekräftige Ergebnisse zu erhalten. Die zweite Evaluationphase wird mit den Erkenntnissen aus dem ersten Evaluationsschritt zusammengeführt und stellt eine ausführliche Evaluation des Pagingsystems zur Klärung der zentralen Evaluationsfragen dar. Zur Veranschaulichung und besseren Auswertung werden die Ergebnisse der Evaluation grafisch aufbereitet und dargestellt. Der abschließende Teil der Evaluation widmet sich dem Vergleich zwischen Niagarino und NiagaraST. Auch hier standen wieder die Performance-Messungen zu den Laufzeiten im Vordergrund. Zusätzlich wurde ein Vergleich des Speicherverbrauchs berücksichtigt. Für diesen Evaluationsschritt mussten die Daten von NiagaraST noch einmal gesondert aufbereitet werden sowie äquivalente Querypläne in Niagarino und NiagaraST geschrieben werden.

5.1 Versuchsaufbau

Der Erfolg und die Aussagekraft der Evaluation hängen maßgeblich vom richtigen Versuchsaufbau ab, für den zwei Perspektiven bestimmend sind. Auf der einen Seite ist es die Wahl der Umgebung, in der die Evaluation durchgeführt wird. Dazu zählen sowohl die verwendete Hardware, sowie die Laufzeitumgebung und eine Anpassung des Gesamtsystems, damit die für eine Evaluation notwendigen Werte für Variablen zugänglich werden. Auf der anderen Seite sind aber auch die Parameter, mit denen das System getestet wird, von besonderer Bedeutung. So müssen zum Beispiel geeignete Querypläne geschrieben werden, um auch mögliche Randfälle abzudecken. Zudem ist für diesen Zusammenhang von Interesse, welche Operatoren durch das Paging im Besonderen beeinflusst werden können oder davon profitieren. Sämtliche hier aufgeführten Aspekte sind entscheidend für eine erfolgreiche Evaluation und wurden im Voraus gut geplant und vorbereitet.
Für diese Evaluation standen neben einem Notebook mit einer Dualcore CPU (Intel(R) Core(TM) i7-4510U 2.0GHz) und 8 Gigabyte RAM mit Windows 8.1 zwei weitere Rechner mit einer Quad Core CPU (Intel(R) Core(TM) i7 870 @ 2.93GHz) und jeweils 16 Gigabyte RAM mit Linux Mint sowie ein Mac Pro mit einer Hexa Core CPU (Intel(R) Xeon(R) CPU E5-1650 v2 @ 3.50GHz) und 64 Gigabyte RAM zur Verfügung. Als Laufzeitumgebung kam auf allen Rechner Oracle Java in der Version 8 bzw. Oracle JDK in der Version 1.8 zum Einsatz.
Die erste Evaluation wurde auf dem Notebook durchgeführt, da sie nur einer Identifizierung von Tendenzen und notwendigen Variablen diente und daher die Leistung ausreichte. Die erste Evaluation bestand zudem immer aus nur einem Durchlauf, der dann zur Verifikation der erkannten Muster wiederholt wurde.
Die Testdaten für die komplette Evaluation sind Datensätze zu Verkehrsdaten aus dem PORTAL-Archiv der Portland State University. Das folgende Listing zeigt einen Beispielausschnitt aus den Daten:

```
1 detectorid,starttime,volume,speed,occupancy,highwayid,stationid
2 1152,"2015-01-11 00:00:00-08",5,61.4,0.33,1,1016
3 1153,"2015-01-11 00:00:00-08",19,54.79,1.8,1,1016
4 1156,"2015-01-11 00:00:00-08",40,0,0,1,5016
5 1159,"2015-01-11 00:00:00-08",5,61.6,0.4,1,1017
```

Die *detectorid* ist hier die ID des Sensors, der die Geschwindigkeit gemessen hat. Die *starttime*, ist die Zeit bei der die Messung begonnen wurde. Der Wert *volume* beschreibt das Verkehrsaufkommen zum Zeitpunkt der Messung. Das Attribut *speed* zeigt dabei die Durchschnittsgeschwindigkeit im gemessenen Intervall. *Occupancy* beschreibt dabei die Verkehrsdichte. Die beiden letzten Attribute *highwayid* und *stationid* beschreiben die ID der Messtation und die ID des zugehörigen Highways. Für die erste Evaluation wurde dazu ein ca. 2 Gigabyte großer Datensatz erzeugt, um das Verhalten des Systems bei sehr großen Datenmengen zu erforschen. Bei dieser ersten Evaluation auf dem Notebook wurde insbesondere der Einfluss von parallel laufenden Prozessen auf die Evaluationsergebnisse deutlich. Aus diesem Grund wurde das Verfahren in der zweiten Eva-

luation noch einmal stark verändert. Um die notwendigen Daten zu erheben, wurden die Querypläne mit einer Funktion ausgestattet, welche die Laufzeit des Planes misst und am Ende ausgibt. Zudem wird der Speicherverbrauch des Planes gemessen und ausgegeben.
Die zweite Evaluation wurde auf dem Mac Pro und den beiden Linux Rechnern durchgeführt. Dazu war zunächst eine Anpassung von Niagarino für einen Deploy als *jar* auf einem anderen Computer nötig. Bisher lief Niagarino nur als *JUnit-Test* aus Eclipse heraus. Dies bedeutet, dass die Querypläne in Niagarino alle als JUnit-Test implementiert sind. Zu diesem Zweck wurde Niagarino um eine *Main-Klasse* erweitert um die Querypläne auch ohne Entwicklungsumgebung ausführen zu können. Um die Leistung der Testrechner optimal ausnutzen zu können, wurden der JVM folgende Argumente mit übergeben: *-xmS512M, -xmX15G -a64*, wobei *-xmS* die kleinste Heapgröße (in diesem Fall 512 Megabyte) und *xmX* die maximale Heapgröße (in diesem Fall 15 Gigabyte) angibt. Die Option *-a64* zwingt die JVM in den 64Bit-Modus, so dass der zur Verfügung stehende RAM optimal ausgenutzt werden kann.
Der Testdatensatz für die zweite Evaluation wurde mit einer Größe von knapp 100 Megabyte deutlich kleiner gewählt. Das hatte vor allem den Grund, dass mit größeren Datenmengen die Laufzeit der Tests für die zweite Evaluation unverhältnismäßig gestiegen wäre.
Für die zweite Evaluation wurde der Queryplan so verändert, dass er nach Bedarf beliebig oft ausgeführt werden kann. Dazu wurde der Queryplan in einer Schleife mit einstellbarer Durchlaufzahl ausgeführt. Am Ende jedes Durchlaufs wurde die benötigte Zeit für diesen Durchlauf gemessen und in einer Liste protokolliert. Die Liste wurde weiter nach dem letzten Durchlauf der Schleife sortiert. Danach wurde der kleinste und der größte Wert abgeschnitten, um mögliche Ausreißer zu entfernen. Abschließend wurde der Mittelwert gebildet. Dieses Verfahren sorgt dafür, dass der Einfluss der anderen Prozesse auf dem Rechner auf die Evaluationsergebnisse minimiert und eine Tendez sichtbar wird. Des Weiteren lassen sich so auch noch weitere statistische Maße, wie zum Beispiel die Standardabweichung, auf die Ergebnisse anwenden. Ausgegeben wurden die Ergebnisse mit den Spalten `Pagegröße` und `Durchschnittszeit pro Pagegröße` in eine CSV-Datei. In einer erweiterten Form kamen noch die Ermittlung von Varianz und Standardabweichung hinzu.
Die Querypläne wurde noch mit einem weiteren Iterationsschritt über die Pagegrößen ausgeführt. Dazu wurde eingestellt, bis zu welcher Pagegröße evaluiert werden soll und in welchen Schritten die Pagegröße erhöht werden sollen. Durch diese Vorgehensweise wurde es möglich, mit einem Deploy für eine vordefinierte Anzahl an Pagegrößen die Durchschnittszeit zu bestimmen. Zu diesem Zweck wurde der `PropertiesReader` modifiziert. Für die zweite Evaluation wurde nicht mehr die Konfigurationsdatei eingelesen. Auch wurde auf eine Konfigurierung von Niagarino und das entsprechende Paging verzichtet. Stattdessen wurde die Konfiguration durch den Queryplan dem `PropertiesReader` zur Verfügung gestellt. Diese Vorgehensweise hat den Vorteil, dass zu Evaluationszwecken zur Laufzeit die Einstellungen für das Paging verändert werden können.

Nur dadurch würde es möglich sein, über die Pagegröße zu iterieren.
Neben dem rein technischen Aufbau war es notwendig, geeignete Versuchsszenarien zu identifizieren und entsprechende Querypläne zu schreiben. Für diesen Zusammenhang erschien es wichtig, den Unterschied zwischen sehr kleinen und komplexen Plänen festzustellen, da sich das Paging gerade bei komplexen Plänen positiv auf die Laufzeit auswirken sollte. Als einfachster Plan wurde ein Plan (Abb. 6) entwickelt, der nur Daten einliest und diese wieder ausgibt.

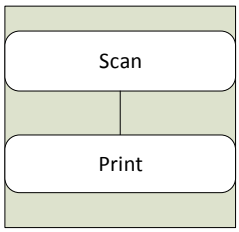

Abb. 6. Der Scan-Print Queryplan

Dieser Plan birgt allerdings einige Nachteile. So steht die Laufzeit dieses Planes in einer direkten Abhängigkeit zum I/O des Rechners, da er nur von der Festplatte einliest und dann diese Daten wieder auf die Festplatte schreibt. Als komplexerer Plan wurde ein Plan (Abb. 7) gewählt, der auch einen Window-Operator beinhaltet. Das Ziel war für diesen Zusammenhang, einen zu erzeugen, der Erkenntnisse über das Verhalten des Pagings in realen Szenarien gibt.

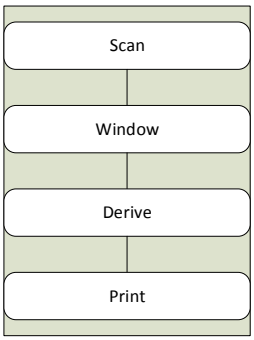

Abb. 7. Der größere Queryplan

Für die Evaluation wurden zusätzlich zwei Querypläne geschrieben. Der erste Plan (Abb. 8) zeigt sich folgendermaßen:

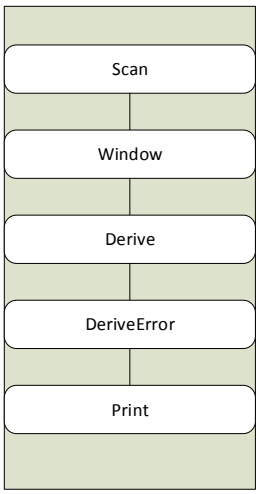

Abb. 8. Der Queryplan mit Fehlerfunktion

Der zweite Plan (Abb. 9) stellt sich folgendermaßen dar:

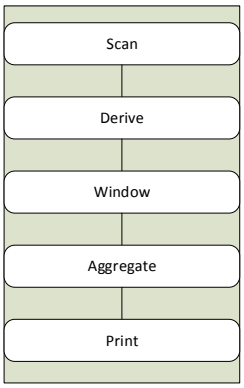

Abb. 9. Der Aggregate-Queryplan

Dieser zweite Plan (Abb. 12) soll möglichst realitätsnahe Anfragen an den Verkehrsdatenstrom abbilden.

Ein weiterer Versuch galt der Messung der Tupeldurchlaufzeiten. Hier sollte die Zeit gemessen werden, die das erste Tupel vom Scan-Operator bis zum Zeitpunkt, wenn es wieder im Printoperator auf die Festplatte geschrieben wird, braucht. Die Messung der Tupeldurchlaufzeiten soll Hinweise darauf geben, ob das Paging richtig implementiert wurde. Die Erwartung hierzu ist, das die Zeit mit der Größe der Page zu nimmt. Um diese Zeiten zu messen, wurde eine neue Klasse eingebaut, welche die Zeit vom Scan-Operator bis zum Print-Operator protokolliert und entsprechend ausgibt. Mit Hilfe dieser Klasse wurde es ermöglicht, die Zeit für jede Pagegröße, die das erste Tupel zum Durchlaufen des Queryplans benötigt, zu ermitteln.

Der vierte Evaluationsschritt ist einem Vergleich mit NiagaraST gewidmet. Da Niagarino von der Architektur und der Arbeitsweise NiagaraST sehr ähnlich ist, bietet sich dieser Vergleich an. Zudem ist NiagaraST das einzige System aus den in Kapitel 2 vorgestellten Systemen, das einen ähnlichen Pagingmechanismus implementiert hat. In diesem Evaluationsschritt wurde die Pagegröße von Niagrino fest auf 30 Tupel eingestellt. Der Grund hierfür ist, dass NiagaraST keine einstellbare Pagegröße hat und diese fix auf 30 Tupel eingestellt ist. Eine mögliche Begründung für genau diese Größe konnte die folgende Evaluation liefern. Wichtig ist in diesem Zusammenhang noch zu erwähnen, dass Niagarino im Vergleich zu NiagaraST geordnete Streams erwartet und auch nur diese akzeptiert.
Für den Vergleich mussten in Niagarino Querypläne erstellt werden, die auch in NiagaraST equivalent umgesetzt werden können. Zu diesem Zweck wurde Niagarino um ein Vergleichsprädikat erweitert, das die Operatoren *le* (kleiner als), *leq* (kleiner gleich), *eq* (gleich), *geq* (größer gleich) und *ge* (größer als) implementiert hat. Dieser Operator kann zum Beispiel im *Select-Operator* angewendet werden um folgende Queries umzusetzen:

```
1 SELECT Tuple t
2 FROM Stream s
3 WHERE t.attr >= value;
```

Des Weiteren mussten die Daten für NiagaraST vorverarbeitet werden, da NiagaraST ein „progressing attribute“ erwartet, in diesem Fall der Zeitstempel als Unix Zeitstempel (die vergangene Zeit in Millisekunden seit dem 1. Januar 1970), und zudem mit Punktierungen arbeitet. Dazu wurde ein `preprocessor` geschrieben, der den Zeitstempel von einem String in die Unix-Zeit konvertiert und nach einer bestimmten Zeit eine Punktierung einfügt. Für die Tests wurden Punktierungen nach jeder Stunde eingefügt. In dem realen Datensatz sieht eine Punktierung folgendermaßen aus:

```
1 100766,1420933200000,13,53.62,0.64,1,1011
2 *,1420933200000,*,*,*,*,*
3 100767,1420933200000,3,66,0,1,1012
```

Für den Vergleich mit NiagaraST wurden zwei Querypläne verwendet. Der eine ist wieder der einfache I/O Plan mit nur Scan und Print. Der zweite Plan ist ein etwas größerer Plan mit einer Selection. Die Tests mit NiagaraST wurden auf dem Notebook durchgeführt. Entsprechend wurden auch die Zeiten für diese beiden Querypläne in Niagarino auf dem Notebook gemessen. NiagaraST ist wie Niagarino in den meisten Teilen in Java geschrieben. Allerdings ist NiagaraST als ein *Client-Server-Modell* implementiert. Dadurch ist es sehr leicht möglich, den eigentlichen Prozess auf einem entfernten Rechner ausführen zu lassen. Das System bringt Startscripte für alle gängigen Betriebssysteme mit. Der Server lässt sich durch die Ausführung des Scripts einfach starten und nimmt dann auf einem definierten Port Verbindungen vom Client entgegen. Dem Script für den Client muss noch mit `-port <port>` der entsprechende Port und mit `-excecute <queryplan>` der entsprechende Queryplan mitgeteilt werden. Die Ausgabe erfolgt standardmäßig auf dem *Standard-Output* und dem *Error-Output* der Konsole. NiagaraST gibt die verarbeiteten Daten als XML-Dokument zurück. Dazu kann die Ausgabe beliebig in eine Datei oder auch nach *null* geleitet werden. Für die Evaluation war es wichtig, die Ausgabe in eine Datei umzuleiten, da eine Ausgabe auf der Konsole nicht in Frage kam, weil diese Ausgabe sehr langsam ist und das Ergebnis zu Gunsten von Niagarino verzerrt hätte. Für den Vergleich von NiagaraST und Niagarino wurde der gleiche Datensatz verwendet wie für die zweite Evaluation.
Der folgende Abschnitt widmet sich nun dem ersten Evaluationsschritt. Dieser sollte Aufschluss über die Effekte des Pagings geben und mögliche weitere Evaluationsfragen liefern. Wie bereits am Anfang des Kapitels erwähnt, wurde in diesem Schritt jeweils nur ein Durchlauf mit einem sehr großen Datensatz gemessen. Dieser Schritt wurde für unterschiedliche Pagegrößen durchgeführt, grafisch aufbereitet und in Relation zu einem Durchlauf ohne Paging gesetzt.

5.2 Erste Evaluation

Die zentrale Frage dieser ersten Evaluation kann folgendermaßen formuliert werden: Ist der gewünschte Effekt einer Verbesserung der Performance gegenüber dem ursprünglichen System festzustellen, der auf das Paging zurückgeführt werden kann? Zudem sollte ermittelt werden, wie sich das Paging verhält und wie sich die unterstellte Performanceverbesserung in Bezug auf einen linearen oder wieder abfallenden Verlauf darstellt.
Wie bereits im letzten Abschnitt erwähnt, besteht dieser erste Teil der Evaluation aus Einzeltests mit einzelnen Durchläufen. Der Wert der hier interessant ist, ist der Tupeldurchsatz, also die Anzahl an Tupel, die pro Sekunde verarbeitet werden. Um diesen Wert zu erhalten, wurde die Gesamtanzahl an Tupel durch die gemessene Zeit in Sekunden geteilt. Für diese Tests wurde die 2 Gigabyte

große CSV-Datei verwendet. Der erste Plan, der getestet wurde, war ein einfacher *Scan-Print-Plan*. Dieser wurde mit den Pagegrößen 1000, 850, 700 und auch ohne Paging ausgeführt.

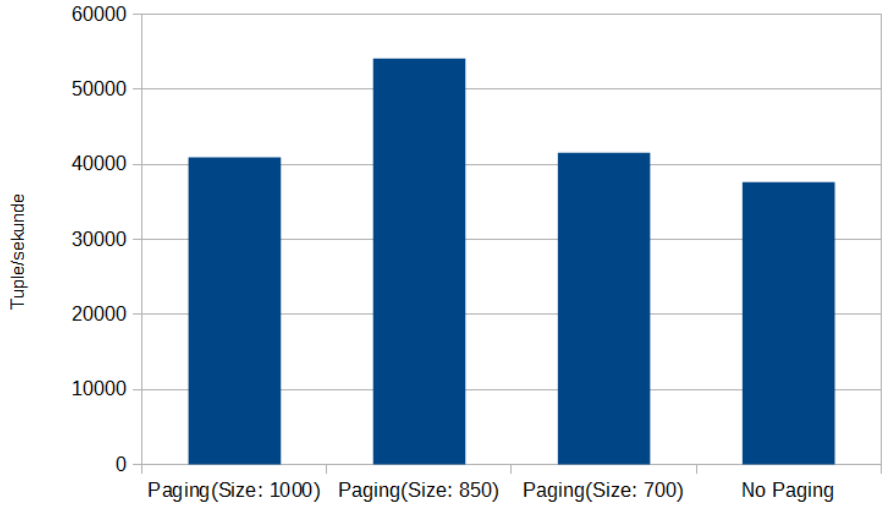

Abb. 10. Erste Evaluation mit Scan-Print-Plan

Auf den ersten Blick ist hier gut zu sehen, dass es doch einen deutlichen Unterschied zwischen Paging und keinem Paging gibt. So scheint das System mit eingeschaltetem Paging deutlich an Geschwindigkeit zu gewinnen. Allerdings ist in dieser Grafik auch zu sehen, dass der Effekt des Pagings mit größeren Pages wieder abnimmt. An dieser Stelle könnte man annehmen, dass die Abnahme der Performance bei diesem Queryplan in einer Abhängigkeit zur Auslastung des Gesamtsystems insbesondere der Festplatte steht, da der getestete Plan sehr I/O-lastig ist. Allerdings ließ sich diese leichte Kurve in mehreren Testdurchläufen reproduzieren. Sehr bemerkenswert ist zudem der große Peak bei einer Pagegröße von 850 Tupeln. Hier stellte sich die Frage, ob dies möglicherweise eine der optimalen Pagegrößen (sofern es mehrere optimale Werte gibt) oder einen Ausreißer darstellt, weil Niagarino für diesen Test unter optimalen Bedingungen lief und der Testrechner zu diesem Zeitpunkt kaum Last hatte. Diesen Fragen sollte im weiteren Verlauf der Untersuchung auf den Grund gegangen werden.
Um mögliche Beeinträchtigungen der Ergebnisse durch die I/O-Last des Testsystems auszuschließen, wurde ein weiterer größerer Queryplan geschrieben. Dieser wurde zusätzlich noch um einen Multiplexer (ein Operator der einen Stream in mehrere Streams aufteilt), einen Frame-Operator (der Operator für dynamische Windows), einen Derive-Operator und einen Merger (ein Operator der die Stre-

ams wieder zusammenfügt) erweitert. Als weiteres Ziel sollte die Kurve aus dem ersten Test reproduziert werden und die Ergebnisse dadurch einer Bestätigung zugeführt werden. Die folgende Grafik zeigt die Ergebnisse des größeren Queryplans.

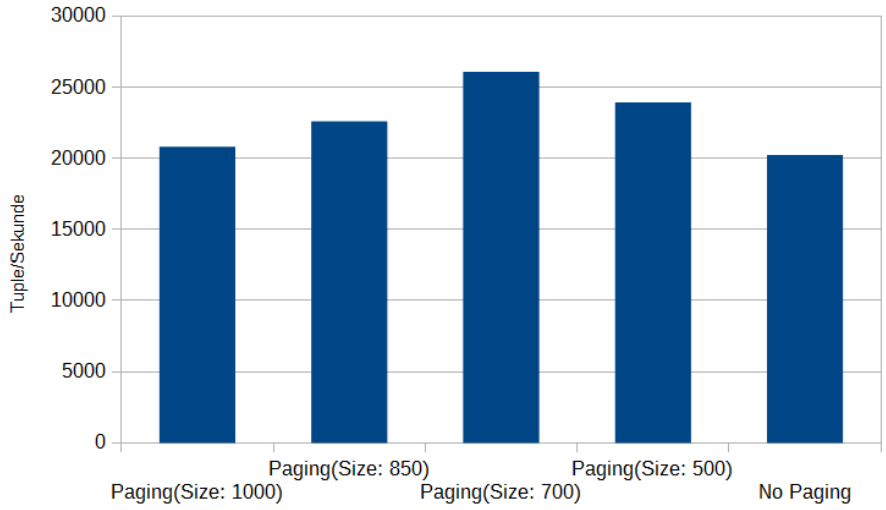

Abb. 11. Erste Evaluation mit einem umfangreicheren Queryplan

An diesen Ergebnissen fiel zunächst die diesmal sehr deutliche Kurve ins Auge. Allerdings liegt in diesem Fall das Maximum der Kurve in diesem Plan bei 700 Tupel pro Page. Im ersten Plan lag es bei 850 Tupel pro Page. Daraus lässt sich schließen, dass eine mögliche optimale Pagegröße je nach Queryplan und Datenmenge im Bereich von 650 bis 900 Tupel pro Page liegt. Auch diese Kurve ließ sich in mehreren Tests wieder reproduzieren. Grundsätzlich zeigt diese Grafik, dass mit dem Paging bessere Ergebnisse erzielt werden können als ohne Paging.
Aus den bisherigen Ergebnissen könnte der Schluss gezogen werden, dass das Paging wie vorgesehen funktioniert und die gewünschte Verbesserung der Performance liefert. Allerdings sollten zunächst noch einige offene Fragen geklärt werden, um diesen Evaluation erfolgreich abschließen zu können. Eine der offenen Fragen betraf die sehr große Pagegröße, aufgrund der diese guten Ergebnisse erzielt werden. Diese Größe ist in der Praxis kaum einsetzbar, denn häufig geht es bei Datenströmen auch um Auswertungen in nahezu Echtzeit. Dies ist bei diesen Größen nicht mehr gegeben, da die Tupel in den Operatoren gesammelt werden, bis eine Page voll ist. Dies stellt eine mögliche Erklärung dar, warum die Pagegröße bei NiagaraST fest auf 30 Tupel gesetzt wurde. Es galt hier vor allem die Frage zu klären, ob Niagarino auch bei dieser Pagegröße gute Werte

erzielen kann oder ob sich die Wirkung des Pagings aufhebt.

Des Weiteren bestand die Problematik, dass die Ergebnisse der einzelnen Durchläufe stark von der Auslastung des Rechners im Hintergrund abhängen. So kann es zum Beispiel sein, dass zwei direkt hintereinander ausgeführte Tests unterschiedlich gute Ergebnisse liefern oder auch extreme Ausreißer auftauchen. Um diese Problematik zu minimieren, sollte in einer weiteren Evaluation auf Durchschnittszeiten aus mehreren Durchläufen zurückgegriffen werden.

Der letzte wichtige Punkt bei den offenen Fragen stellte sich in Bezug auf die Kurve. Auf den ersten Blick würde man davon ausgehen, dass sich die Wirkung des Pagings durch zu große Pages wieder aufhebt, da die Operatoren dadurch eine längere Wartezeit haben. Allerdings muss man sich hier die Frage stellen, warum die Kurve genau in diesem getesteten Bereich entsteht. Weiter könnte man vermuten, dass es eine ähnliche Kurve auch in anderen Größenbereichen geben könnte.

In einer weiteren Evalutation musste zudem geklärt werden, wie sich Niagarino im Bereich von 20 bis 40 Tupel pro Page verhält. Das hat den Grund, dass NiagaraST die Tupelgröße fest auf 30 Tupel pro Page gesetzt hat. An dieser Stelle stellte sich die Frage, ob dieser Wert in diesem Bereich optimal ist. Zusätzlich sollte das Verhalten von Niagarino bei dieser Pagegröße evaluiert werden, da die bisherigen Ergebnisse, wie oben erwähnt, nur mit sehr großen Plänen erzeugt wurden und diese in der Praxis kaum relevant sind. So wurde eines der nächstliegenden Haupziele darin formuliert, einen Wert für die Pagegröße zu finden, der gute Ergebnisse liefert und gleichzeitig den Anforderungen an ein Echzeitanalysesystem gerecht werden kann.

Zusammenfassend stellt dieser erste Evaluationsschritt eine wichtige Komponente der gesamten Evaluation dar. Durch diesen Schritt konnten erste genauere Evaluationsziele identifiziert werden und auch Hinweise auf unerwartete Phänomene gefunden werden. Aufgrund dieser Ergebnisse war es nun möglich, eine zweite Evaluation durchzuführen, die sich durch eine höhere Präzision und Effektivität auszeichnet.
Auch für den Implementationsprozess des Pagingsystems stellte diese erste Evaluation eine wichtige Komponente dar. Durch diese ersten Tests konnten Fehler im System gefunden und behoben werden. Auch lieferte sie wichtige Hinweise auf mögliche Probleme in Richtung einer effektiveren und einfacheren Implementierung des Systems. Ein Beispiel hierfür ist der Wegfall des Pagepools. Diese erste Evaluation verdeutlichte die *Deadlock-Problematik* des Pagepools, welche identifiziert und ausgebessert werden konnte. Darüber hinaus konnte gezeigt werden, dass alle Tupel ihre Reihenfolge behalten und durch das Paging nicht durcheinander geraten, was fatale Folgen für den Einsatz des Datenstrommanagementsystems gehabt hätte. Auch konnte nur so zweifelsfrei sichergestellt werden, dass sowohl volle Pages als auch die nicht ganz gefüllten Pages am En-

de der Ausführung eines Operators abgeschickt werden und nicht verloren gehen.

Das nächste Unterkapitel widmet sich den Ergebnissen des zweiten Evaluationsschrittes. Diese wurde auf Basis der Ergebnisse dieser ersten Evaluation entwickelt und geplant. Auch wurden sämtliche Erkenntnisse der ersten Evaluation berücksichtigt, um eine möglichst eindeutige Aussage über die erfolgreiche Implementierung des Pagingsystems treffen zu können.

5.3 Evaluationsergebnisse

Die Ergebnisse der zweiten Evaluation sollten Aufschluss darüber geben, ob das Pagingsystem wie vorgesehen implementiert ist und funktioniert. Des Weiteren wird der Frage, woher die Kurve aus der ersten Evaluation kommt, nachgegangen. Für diese Evaluation wurden keine experimentellen Operatoren, wie zum Beispiel der Frame-Operator oder der Multiplexer, verwendet, da ein möglichst praxisrelevantes und auch mit anderen Datenstrommanagementsystemen vergleichbares Szenario eingesetzt werden sollte.
Die folgenden Ergebnisse wurden mit einem Queryplan erzeugt, der nicht mehr mit dem Frame-Operator realisiert wurde, sondern einen tupelbasierten Window-Operator verwendet. Der genaue Queryplan (Abb.12) sieht folgendermaßen aus:

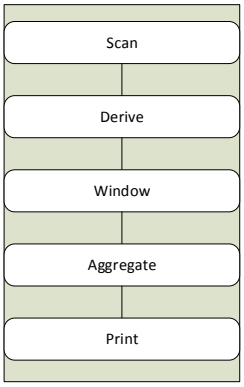

Abb. 12. Der Queryplan

Die Pagegrößen gehen in Zehner-Schritten von 10 bis 90, wobei eine Pagegröße von 0 die Ausführung ohne Paging bedeutet. Die gemessene Zeit ist die Durchschnittszeit aus 98 Werten. Für die Berechnung wurde der größte und der kleinste Wert nicht berücksichtigt.

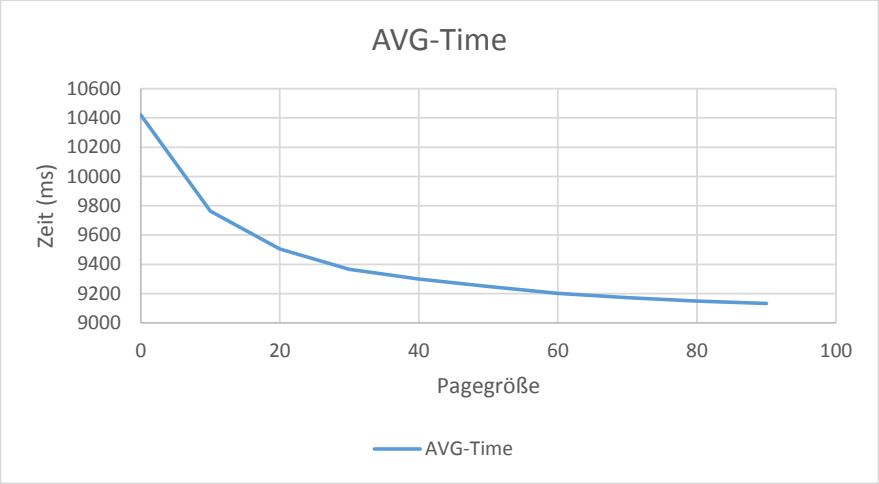

Abb. 13. Ergebnisse des ersten Queryplans

In Abb. 13 ist ab dem Zeitpunkt der Aktivierung des Paging ein deutlicher Abfall der Laufzeit festzustellen. Interessant ist für diesen Zusammenhang, dass die Kurve bis 30 Tupel pro Page relativ stark abfällt und dann langsam abflacht. Zudem wird deutlich, dass die Kurve nach 90 Tupel pro Page weiter fällt. Dieses Ergebnis zeigt, dass die Wahl von 30 Tupel pro Page als feste Pagegröße einen guten Kompromiss zwischen Performancegewinn und größtmöglicher Nähe an einer Echtzeitverarbeitung der Daten darstellt, denn ab 30 Tupel fällt die Kurve bei weitem nicht mehr so stark, dass aus dem Performancegewinn ein größerer Vorteil gezogen werden könnte. Darüber hinaus zeigt dieses Diagramm bereits erste positive Erkenntnisse zur vorgesehenen Implementierung des Paging. Die Kurve verläuft wie erwartet und entspricht damit auch dem Performancegewinn der Erwartungen.
Ein weiterer interessanter Wert stellt die Standardabweichung zur durchschnittlichen Laufzeiten pro Page dar. An ihr ist zu erkennen, wie aussagekräftig die Mittelwerte der Laufzeiten sind.

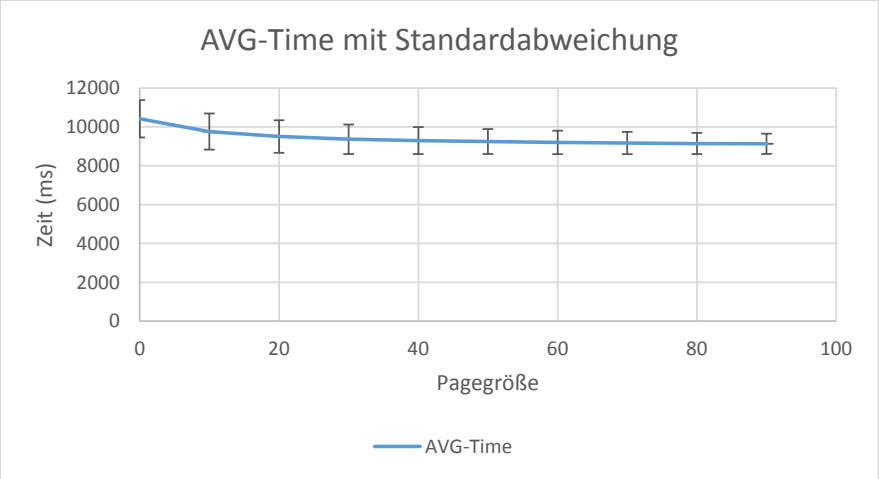

Abb. 14. Grafik mit Standardabweichung

Interessant für diesen Zusammenhang (Abb. 14) ist, dass die Größe der Pages eng mit der Standardabweichung korreliert: Je größer die Pages werden, desto kleiner wird die Standardabweichung. Daraus lässt sich schließen, dass die Laufzeiten mit größer werdenden Pages stabiler werden. Eine Begründung hierfür dürfte sein, dass durch das Befüllen und Abarbeiten der größeren Pages Schwankungen, die durch Festplattenzugriffe oder Auslastung des Systems entstehen, kompensiert werden und bei größeren Pages möglicherweise fast vollständig verschwinden können. Es könnte auch bedeuten, dass es einen Wert für die Laufzeit gibt, ab der sich Performancegewinn und die längeren Wartezeiten durch die großen Pages gegenseitig aufheben. Des Weiteren gibt es weniger Fluktuationen durch häufiges Wechseln der Threads. Dies kann dazu führen, dass ab einer bestimmten Pagegröße kaum noch Veränderungen in der Laufzeit bei wachsender Pagegröße zu verzeichnen sind. Den wichtigsten Hinweis den die Standardabweichung für diesen Fall liefert, ist darin zu sehen, dass sie mit wachsender Pagegröße abnimmt. Dies zeigt, dass das Paging wie vorgesehen implementiert werden konnte und auch wie erwartet funktioniert. Anders wäre es gewesen, wenn die Standardabweichung größer geworden wäre. In diesem Fall hätte man davon ausgehen müssen, dass die Kurve eher zufällig durch Ausreißer entstanden ist und dass grundsätzlich keine große Verbesserung der Performance besteht.
Eine weitere Bestätigung zur richtige Implementierung des Pagingsystems ergab die Messung des Zeitraums, die das erste Tupel braucht, um den Plan zu durchlaufen. Dazu wurden jeweils für eine Pagegröße die Startzeit im Scan-Operator und die Endzeit im Print-Operator gemessen. Die Differenz dieser beiden Zeiten stellt den Zeitraum dar, den das Tupel für den Queryplan benötigt. Für diesen

Zusammenhang wurde erwartete, dass die Zeit analog mit der Größe der Pages anwächst. Hier sollte die beste Zeit ohne Paging beziehungsweise mit einer Pagegröße von 10 und die schlechteste Zeit in diesem Fall mit einer Pagegröße von 90 Tupeln pro Page erreicht werden.

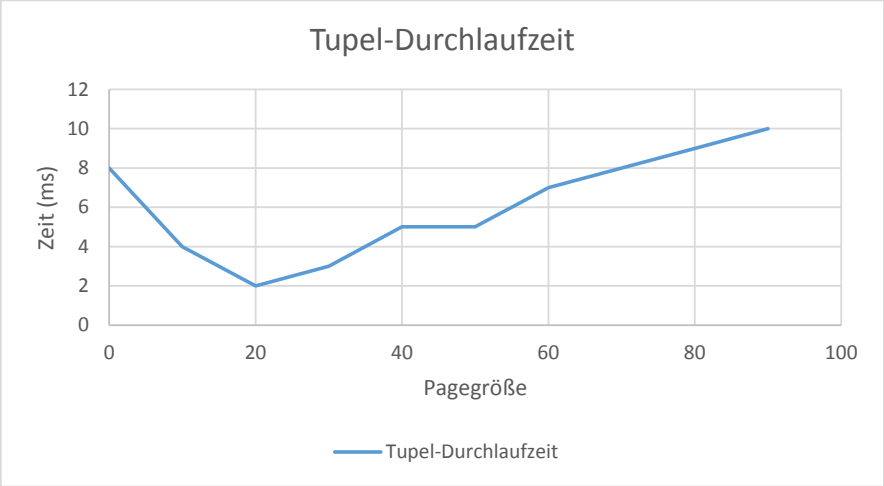

Abb. 15. Durchlaufzeit des ersten Tupels

Abb. 15 macht deutlich, dass der erwartete Effekt auch eingetreten ist. Mit 20 Tupel pro Page wurde das beste Ergebniss erzielt und die Zeit nahm dann nahezu linear zu, wenn die Pages größer wurden. Interessant erscheint allerdings die Tatsache, dass der Wert für den Queryplan mit deaktiviertem Paging und der Wert für eine Pagegröße von 10 Tupeln entgegen den Erwartungen relativ hoch ist. Eine mögliche Erklärung für dieses Phänomen könnte sein, dass es sich um Ausreißer handelt, die durch die Initialisierung des Systems entstanden sind. Insgesamt sind diese beiden Werte allerdings vernachlässigbar, da ab einer Pagegröße von 20 Tupeln exakt das erwartete Verhalten zu erkennen ist, was die Vermutung, dass es sich hier um Ausreißern handelt, stützt. Dieser Test liefert abschließend den Beweis, dass das Pagingsystem wie vorgesehen implementiert ist und auch wie erwartet funktioniert. Somit lässt sich auch zeigen, dass durch das Paging eine Performanceverbesserung erzielt wurde.
Zur Verifikation der Ergebnisse wurde noch ein weiter Queryplan (Abb. 16) geschrieben und evaluiert. Der Queryplan sieht folgendermaßen aus:

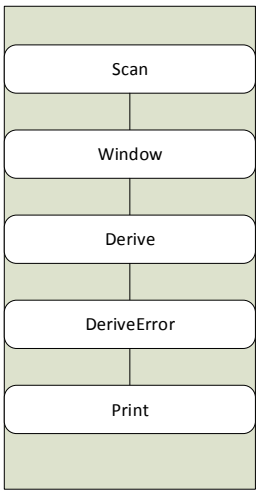

Abb. 16. Queryplan mit Fehlerfunktion

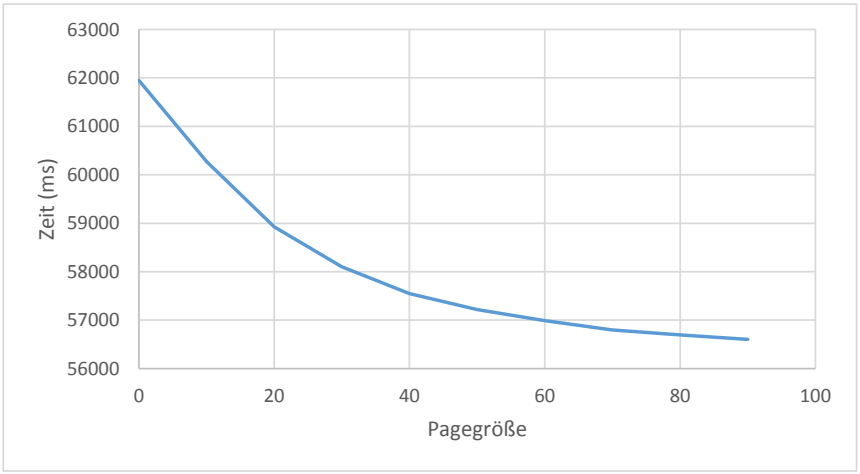

Abb. 17. Ergebnisse des zweiten Plans

Auch auf Abb. 17 sieht man eine ähnliche Kurve, die auch das vorherige Diagramm zeigte, auch wenn diesmal die Zeiten um das sechsfache größer sind.

Man sieht wieder sehr deutlich die absteigende Kurve, die eine Performanceverbesserung darstellt. Damit bestätigt dieser Test die Ergebnisse aus dem vorangegangenen Test. Auch wenn in diesem Fall die Differenz zwischen der Zeit ohne Paging und der Zeit mit einer Pagegröße von 60 bei 500 Millisekunden liegt. Bei dem vorangegangenen Test lag die Differenz bei 1200 Millisekunden. Diese Differenz lässt sich dadurch begründen, dass mit dem `deriveError` Operator ein weiterer Rechenoperator in diesem Queryplan eingesetzt wurde, der den Fehler berechnet. Dieser Operator verursacht möglicherweise Wartezeiten und dies insbesondere beim Übergang von zwei Pages. Der Effekt des Pagings kann somit durchaus in Abhängigkeit vom verwendeten Operator unterschiedlich stark ausfallen. Insgesamt lässt sich aber immer eine Verbesserung der Performance mit eingeschaltetem Paging verzeichnen.
Die Frage nach der Pagegröße warf jedoch weitere Fragen auf. Zwar lässt sich durch die bisherigen Ergebnisse die Wahl von 30 Tupel pro Page bei NiagaraST erklären, aber es gibt durchaus Gründe, andere Größen zu wählen. So könnte es für optimale Ergebnisse durchaus sinnvoll sein, die Pagegröße je nach Queryplangröße und Datenmenge zu bestimmen und einzustellen.
Eine weitere Fragestellung in Bezug auf die Pagegröße bezog sich auf den Einfluss der Pagegröße auf die Windows. So galt es herauszufinden, wie sich die Zeiten darstellen, wenn die Pagegröße gleich der Windowgröße ist. Aber auch was sich verändert, wenn man für die Page die halbe Windowgröße oder die 1,5-fache Windowgröße wählt. Auch sollte evaluiert werden, ob es einen Unterschied macht, wenn die Pagegröße in keinem geraden Verhältnis zur Windowgröße steht. Zu diesem Zweck wurde die Pagegröße fest auf 30 Tupel pro Page gesetzt. Zudem wurde eine Funktion eingebaut, die die Windowgrößen für jeden Durchlauf einstellt. Dabei wurden Windowgrößen von 13, 15, 30, 60, 73 evaluiert. Diese Größen entsprechen den oben genannten Anforderungen an die Evaluation.

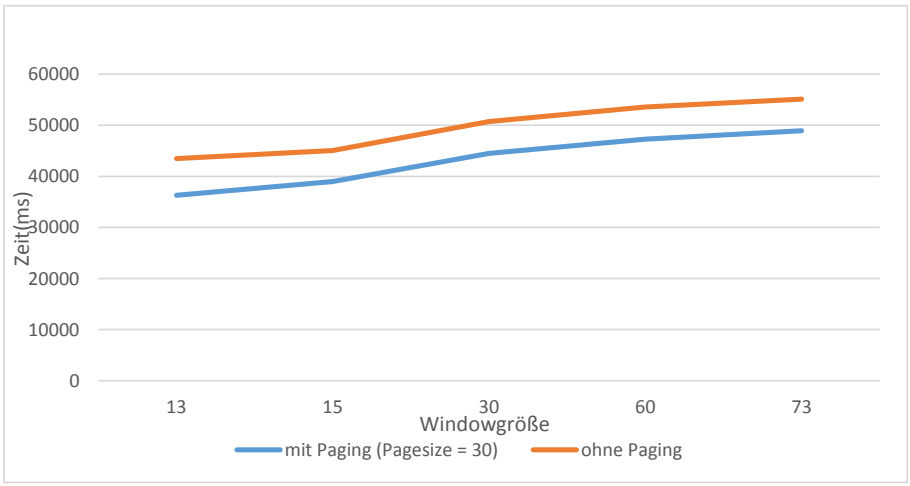

Abb. 18. Ergebnisse mit unterschiedlichen Windowgrößen

Abb. 18 zeigt die Ergebnisse des Tests. Man kann klar erkennen, dass die Zeit leicht zunimmt, wenn die Windows größer werden. Es lässt sich aber kein Unterschied im Wachstum der Kurve feststellen, wenn die Pagegröße in einem geraden Verhältnis steht. Die relativ konstante Steigung scheint nur von der Windowgröße abzuhängen. Daraus lässt sich schließen, dass das Paging keinerlei Einfluss auf die Windows hat. Dies ist ein wichtiges Ergebnis, denn es zeigt, dass bei der Wahl der optimalen Pagegröße die Windowgröße nicht berücksichtigt werden muss. Zudem stellt dies eine wichtige Erkenntnis für weitere Forschungen in Richtung tupelbasierter Windows sowie den bereits vorgestellten Frames dar, da hierbei sich die Windowgröße nicht bestimmen lässt und somit veränderlich bleibt. Dieses Ergebnis lässt insgesamt den Schluss zu, dass auch die Performance bei der Verwendung von Frames nicht von den Pagegrößen abhängig ist.

Offen ist nach wie vor die Frage zu den Kurven aus der ersten Evaluation. Um dieser Frage nachzugehen, wurde ein größerer Test entwickelt. Dieser Test geht über Pagegrößen von 10 bis 10000 Tupel pro Page, wobei die Pagegröße immer um 10 Tupel erhöht wird. Das Ziel dieser Evaluation war es, durch die große Anzahl an Werten heraus zu finden, ob und wo diese Kurve wiederzufinden ist. Da für diesen Test eine relativ lange Laufzeit erwartet wurde, wurde er mit einem kleinen Plan, dem Scan-Print-Plan, durchgeführt, denn wie aus der ersten Evaluation zu sehen war, tauchte auch bei diesem Plan bereits diese Kurve auf. Des Weiteren sollte die Frage geklärt werden, ob sich die Werte ab einer bestimmten Pagegröße fest einpendeln. Mit dieser Evaluation sollte es möglich sein, einen Überblick über das Verhalten des Systems mit aktiviertem Paging zu bekommen.

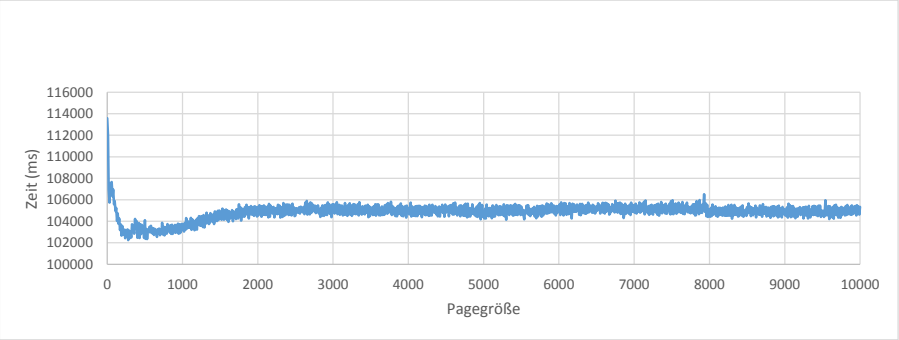

Abb. 19. Großer Scan-Print Test

An dieser Visualisierung (Abb. 19) der Evaluation sieht man, dass die Ergebnisse zwischen einer Pagegröße von 20 und 2000 Tupel pro Page am besten sind. Ab ca. 2000 Tupel pro Page scheint sich die Laufzeit auf einen konstanten Wert einzupendeln. So lässt sich auch erklären, warum die Laufzeit mit größeren Pages wieder zunimmt. Der Effekt des Pagings hebt sich durch die längere Wartezeit teilweise auf. Auch auf die Kurven aus der ersten Evaluation lässt sich ein Rückschluss ziehen: Mit größeren Pages nimmt die Laufzeit wieder zu. Allerdings ist die Laufzeit ohne Paging noch deutlich größer als mit Paging an beliebiger Stelle. Dieses Verhalten lässt den Schluss zu, dass die Kurve im Bereich von 20 bis 2000 Tupel pro Page an beliebiger Stelle zu finden ist. Weitere Schwankungen in den Kurven lassen sich durch die Durchführung der ersten Evaluation erklären, denn diese basiert, wie bereits beschrieben, nicht auf den Mittelwerten mehrerer Messungen sondern auf den Werten aus Einzelmessungen. So lassen sich durchaus Schwankungen in den Ergebnissen erklären. Auffällig an dieser Grafik ist, dass es keine linearen Verläufe gibt, sondern die Ergebnisse stark zu schwanken scheinen. So lässt sich zwar durchaus eine Tendenz erkennen, die das erwartete Ergebnis zeigt, aber es sind keine klaren Linien zu erkennen. Eine Erklärung hierfür ist die Verwendung des Scan-Print-Plans, da dieser Plan stark von der Eingabe und Ausgabe des Systems abhängt, im vorliegenden Fall vom I/O des Testrechners. So lassen sich diese Schwankungen vermutlich komplett auf den I/O zurückführen. Um dies aufzuzeigen, wurde ein zweiter Test mit möglichst vielen Pagegrößen durchgeführt. Dieser Test beinhaltete mehrere Operatoren, was die Auswirkungen des I/O möglichst weit reduzieren soll oder diese sogar insgesamt entfernt, so dass auf der Grafik keine Schwankungen mehr zu erkennen sind. Des Weiteren sollte dieser Test auch dazu dienen, die Ergebnisse aus dem ersten Test zu bestätigen. So sollte wiederum ab ca. 2000 Tupel pro Page zu erkennen sein, dass sich die Laufzeit auf einen festen Wert einpendelt. Dieser Test arbeitet in einem Bereich von 10 bis 4000 Tupel pro Page, da eine Obergrenze

von 4000 Tupel pro Page zur Beobachtung des zu erwarteten Effekts genügen dürfte, höhere Werte die Laufzeit dagegen ins Unermessliche steigern würden.

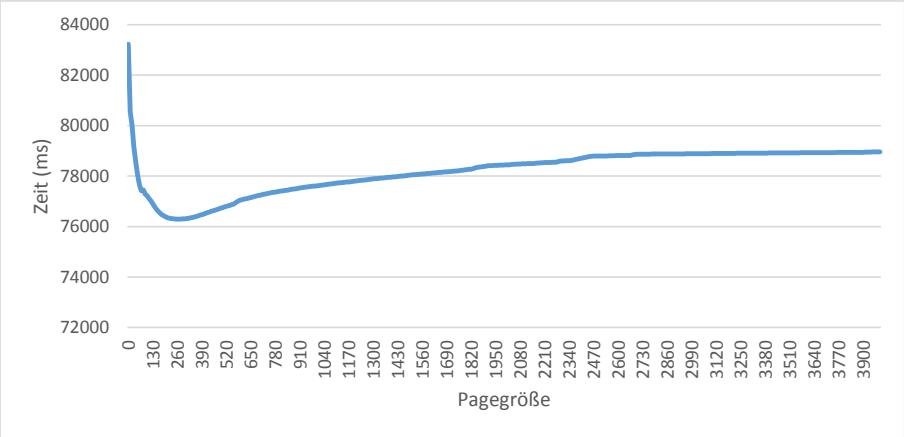

Abb. 20. Zweiter Test mit großen Pages

Auf den ersten Blick fällt an Abb. 20 auf, dass die Kurve sich der vorherigen Kurve stark ähnelt. Allerdings sind nun keine Schwankungen zu sehen. Dadurch bestätigt sich, dass die Schwankungen durch den Input und Output des Rechners entstehen. Auch diese Kurve zeigt deutlich, dass sich die Laufzeit in einem Bereich um 2000 Tupel pro Page stabilisiert. Damit lassen sich die Ergebnisse aus dem vorangegangenen Test mit diesem Test bestätigen. Beide Tests zeigen eindeutig, dass Werte kleiner 500 Tupel pro Page die besten Performance-Ergebnisse liefern. Insgesamt wird gezeigt, dass mit kleinen Pagegrößen um die 40 Tupel pro Page sehr gute Ergebnisse erzielt werden können und trotzdem verhindert wird, dass das Paging zu großen Einfluss auf zeitrelevante Analysen hat.

Neben der Laufzeit wurde auch der Speicherverbrauch mit und ohne Paging und mit unterschiedlichen Pagegrößen analysiert. Dazu wurde nach jedem Durchlauf der Speicherverbrauch protokolliert. Auf den ersten Blick würde man annehmen, dass durch das Paging der Speicherverbrauch um ein vielfaches steigt, da die Pages Tupel speichern und im Verlauf mehrere Pages im Umlauf sind. Die Auswertung der Daten ergab allerdings, dass der Speicherverbrauch durch das Paging nicht signifikant steigt. Eine minimale Steigerung ist bei sehr großen Pages zu verzeichnen. Allerdings sind von diesen deutlich weniger im Umlauf als von kleinen Pages und so gleicht sich der Effekt aus. Es ist entgegen der Erwartungen deutlich zu sehen, dass das Paging keinen entscheidenden Einfluss auf den Speicherverbrauch hat.

Die Evaluation zeigt abschließend, dass das Pagingsystem wie vorgesehen implementiert werden konnte. Des Weiteren wurde deutlich erkennbar, dass das Paging den erwarteten Effekt der Performance-Verbesserung bringt. Daneben ergab die Analyse des Speicherverbrauchs, dass sich aus dem Paging keine signifikanten Nachteile ergeben. Als letzter Evaluationsschritt steht der Vergleich mit NiagaraST aus. Diesem widmet sich der folgende Abschnitt.

5.4 Vergleich von Niagarino mit NiagaraST

NiagaraST wurde an der Portland State University entwickelt. Es ist das Datenstromverwaltungssystem, das Niagarino von der Architektur her am ähnlichsten ist. Da es wie Niagarino ein Pagingsystem implementiert hat, bot sich hier ein Vergleich der Laufzeiten und des Speicherverbrauchs an. Da NiagaraST mit dieser Pagegröße arbeitet, musste für einen Vergleich in Niagarino die Pagegröße fest auf 30 Tupel pro Page gesetzt werden. Des Weiteren mussten geeignete Querypläne identifiziert und implementiert werden. Für NiagaraST mussten dazu zusätzlich noch die Querypläne in XML übersetzt werden, da in NiagaraST der Queryplan als XML File mit übergeben wird. Für diesen Vergleich wurden zwei Querypläne verwendet: Zum einen der Scan-Print Plan und zum anderen ein Queryplan mit einem Selections-Operator, der nur Tupel in die Ergebnismenge mit aufnimmt, wenn eine bestimmte Geschwindigkeit gefahren wird. Als Datensatz kamen wieder die Verkehrsdaten aus der vorangegangenen Evaluation zum Einsatz. Das folgende Beispiel zeigt den Queryplan in XML.

```
1  <?xml version="1.0"?>
2  <!DOCTYPE plan SYSTEM "queryplan.dtd">
3
4  <plan top="cons">
5
6  <!-- Internalize stream -->
7
8  <csvstream id="data"
9    file_name="C:\Users\Maximilian\workspace\Preprocessing\prepro.
         csv"
10   attr_names="detectorid starttime volume speed occupancy
         highwayid stationid"
11   attr_types="double TS double double double double double"
12 />
13
14 <select id="sel" input="data" log="no">
15 <pred op="gt">
16      <var value="$speed"></var><number value ="60"></number>
17 </pred>
18 </select>
19
20 <construct id="cons" input="sel">
21 <![CDATA[
22 <result>
```

```
23  $detectorid
24  $starttime
25  $volume
26  $speed
27  $occupancy
28  $highwayid
29  $stationid
30 </result>
31 ]]>
32 </construct>
33 </plan>
```

Nach dem Kopf der XML-Datei wird dem Plan zunächst der Input mitgegeben. In diesem Fall handelt es sich um eine CSV-Datei, weshalb `csvstream` verwendet wird. Der Stream bekommt eine ID und ihm wird der Pfad zur Datei mitgegeben. Im nächsten Schritt muss dem Plan noch die Struktur der Daten mitgegeben werden, sprich die Kopfzeile der CSV und die dazugehörigen Datentypen. Das TS stellt das sogenannte Progressing Attribut dar, das in diesem Plan der Timestamp ist. Nach der Definition des Inputstreams folgen die Operatoren. In diesem Plan gibt es nur den `select` Operator. Dieser wird wieder mit einer ID definiert. Des Weiteren benötigt er einen Input, im vorliegenden Fall der Stream. Diesem Operator wird noch ein *Größer als (gt)*-Prädikat mitgegeben. Bei der Definition des Prädikats muss das Attribut (in diesem Fall *speed*) und der Grenzwert (in diesem Fall 60) mitgegeben werden. Abschließend wird noch die Ergebnismenge definiert. Das bedeutet, dass alle Attribute angegeben werden müssen, die im Ergebnis enthalten sein sollen. Neben den ursprünglichen Attributen werden auch alle von Operatoren neu erzeugten Attribute angegeben. Hier zeigt sich ein deutlicher Unterschied zu den in Java implementierten Queryplänen in Niagarino. In NiagaraST müssen die Streams nicht explizit angegeben werden. Stattdessen bekommt jeder Operator die ID seines Vorgängers mit. Auch werden die Datenquellen direkt im Queryplan angegeben, damit es auch ohne Programmierkenntnisse möglich bleibt, für NiagaraST Querypläne zu erstellen. So müssen in NiagaraST auch keine Scan- und Print-Operator angegeben werden, da diese implizit durch die Angabe der Datenquelle und der Definition des Outputs festgelegt werden. Ein weiteres Unterscheidungsmerkmal zeigt sich dadurch, dass beim Output in Niagarino jeder Operator sein neues Attribut anhängt und dieses dann im Output auftaucht.
Der erste Vergleichstest ist wieder der Scan-Print-Plan. Er soll einen ersten Anhaltspunkt dafür liefern, inwieweit sich die Laufzeiten von Niagrino und NiagaraST unterscheiden. Des Weiteren bekommt man Informationen zu den Einflüssen der I/O des Computers auf die Laufzeiten. Wichtig erschien hierfür, dass die Ausgabe des NiagaraST-Clients in eine Datei umgeleitet werden, da es alle Ergebnisse auf dem *Standardoutput* ausgibt und die Konsole extrem langsam ist. Auch für diesen Zusammenhang wird ein wesentlicher Unterschied zu Niagarino deutlich, da im Queryplan ein Outputfile definiert werden muss, in das der Output geschrieben wird. Es ist aber auch möglich, die Ergebnisse direkt auf den *Standardoutput* zu leiten.

Für diesen Test wurden jeweils für NiagaraST und Niagarino zehn Durchläufe durchgeführt und der Mittelwert der Laufzeit gebildet. In NiagaraST wird die Laufzeit am Ende der Ausführung auf dem *Erroroutput* ausgegeben. So bekommt man die Laufzeit, auch wenn die Ausgabe in eine Datei umgeleitet wurde.

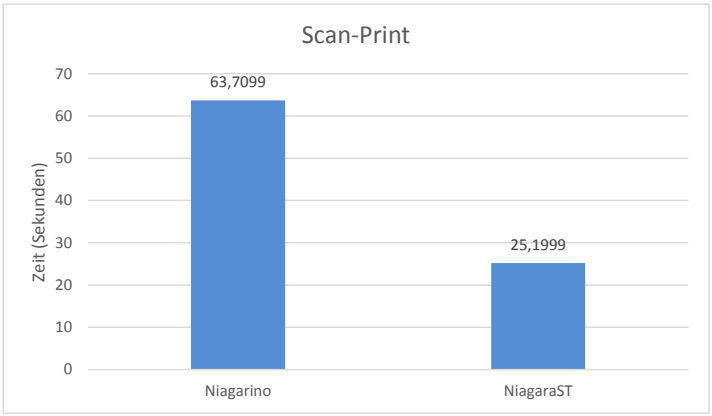

Abb. 21. Scan-Print Vergleich

Aus Abb. 21 wird deutlich, dass NiagaraST weniger als die Hälfte der Laufzeit von Niagarino benötigt. Das deutet darauf hin, dass NiagraST trotz seiner Ähnlichkeit zu Niagarino um einiges effizienter implementiert ist als Niagarino. Allerdings ist dieses Ergebnis auch der I/O-Lastigkeit des Queryplans zuzuschreiben. Ein besseres Ergebnis sollte der nächste Vergleich bringen. Dazu wurde der Queryplan aus obigem Listing ausgeführt. Auch bei diesem Test wurden wieder 10 Zeiten gemessen und der Mittelwert der Laufzeiten gebildet.

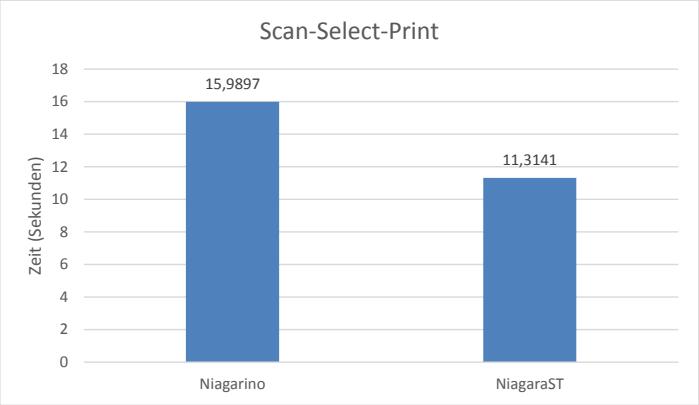

Abb. 22. Scan-Select-Print Vergleich

Abb. 22 zeigt den Einfluss des I/O bei dem ersten Vergleich. Daraus wird hier deutlich, dass der Abstand der Laufzeiten von NiagaraST und Niagarino deutlich kleiner ist. Niagarino benötigt nur ca. 30 Prozent mehr Laufzeit als NiagaraST und nicht, wie in dem ersten Test, über 100 Prozent. Aber es bestätigt auch, das NiagaraST ein sehr effizientes System ist und trotz ähnlicher Implementierung deutlich weniger Laufzeit benötigt als Niagarino. Die Ergebnisse zeigen aber auch, dass Niagarino mit Paging deutlich näher an die Laufzeit von NiagaraST herangerückt ist. Zusammenfassend sieht man also noch einen ausgeprägten Performanz-Unterschied. Dies deutet darauf hin, dass NiagaraST neben dem Paging noch weitere Optimierungsmechanismen implementiert hat. Möglicherweise ist das auch mit in den intern verwendeten Datenstrukturen begründet. Ein weiteres interessantes Vergleichsmoment stellt der Speicherverbrauch dar, da er weitere Rückschlüsse auf die Effizienz des Systems zulässt. Dazu wurde die Heapgröße mit dem Programm VisualVM gemessen. Für beide Tests wurde einmalig der Scan-Select-Print-Queryplan ausgeführt. Wichtig war hier, den Speicherverbrauch des NiagaraST-Servers zu messen, da dieser für die Auswertung der Daten zuständig ist. Der interessante Wert stellt hier der tatsächlich verwendete Heapplatz dar. Dieser entspricht dem tatsächlichen Speicherverbrauch des Systems, während die gesamte Heapgröße nur den reservierten Speicher für das System angibt. Die Grafik (Abb. 23) zeigt die Belegung des Heaps der *Java Virtual Machine* während der Ausführung des Scan-Select-Print-Queryplans in NiagaraST.

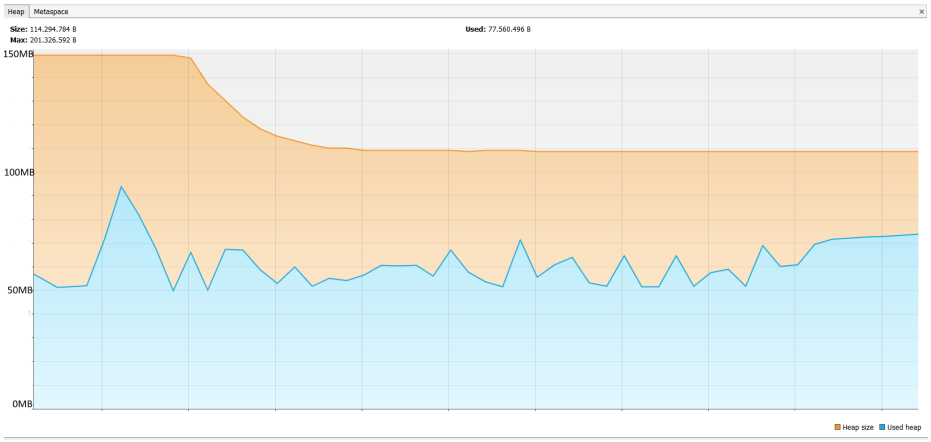

Abb. 23. Heapverbrauch von NiagaraST

Zu beobachten ist hier eine konstante Größe des Heaps von mindestens 50 Megabyte. Gerade am Anfang der Ausführung ist ein größerer Peak zu sehen. Dann stabilisiert sich die Heapgröße auf einen Bereich zwischen 50 und 70 Megabyte. Interessant ist hier ein Anstieg des Speicherverbrauchs bzw. des Heaps auf über 70 Megabyte am Ende der Ausführung. Die nächste Grafik (Abb. 24) zeigt den Speicherverbrauch von Niagarino. Mit Niagarino wurde der gleiche Queryplan ausgeführt und auch hier war die Pagegröße in Niagarino wieder auf 30 Tupel pro Page festgelegt (der gleiche Wert wie in NiagaraST).

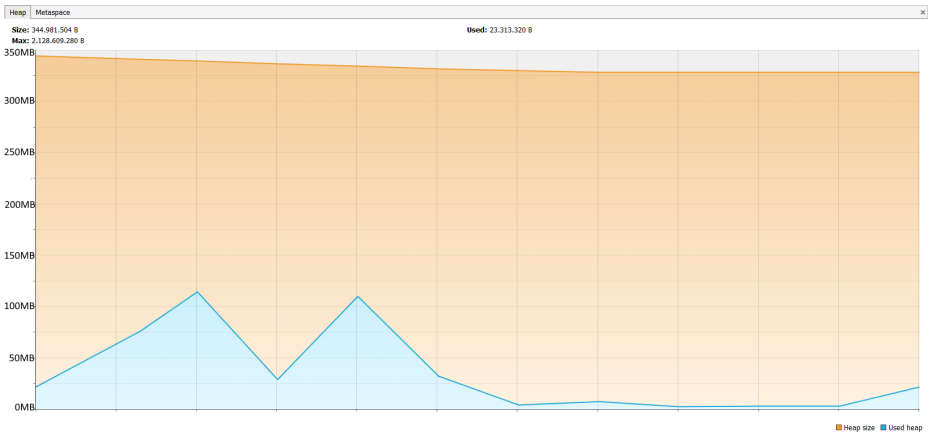

Abb. 24. Heapverbrauch von Niagarino

Auf den ersten Blick sieht man hier zu Beginn der Ausführung zwei deutliche Peaks und dann einen sehr starken Abfall des Speicherverbrauchs. Diese Peaks sind sogar deutlich größer als der Peak, der bei der Grafik von NiagaraST zu sehen war. Doch dann sinkt der Speicherverbrauch auf unter 25 Megabyte. Damit verbraucht Niagarino zu diesem Zeitpunkt weniger als die Hälfte an Speicher als NiagaraST. Das ist vor allem damit zu begründen, dass NiagaraST intern deutlich komplexere Datenstrukturen verwendet als Niagarino. Überraschenderweise kann allerdings die Größe der Streams bzw. der verwendeten Queues ausgeschlossen werden, da diese bei fünf Pages beginnt und nur maximal 100 Pages groß werden kann. Des Weiteren ist ein großer *Overhead* durch die Implementierung als Client-Server System vorhanden. Dieser benötigt noch weitaus mehr Objekte, die für das Herstellen und Verwalten der Verbindungen notwendig sind. Zudem ist in NiagaraST auch eine Webschnittstelle für die Kommunikation unter mehreren NiagaraST-Servern implementiert. Dadurch ist es möglich, dass NiagaraST verteilt auf mehrere Systeme ausgeführt werden kann. Auch diese Schnittstelle benötigt viele Objekte und damit Speicher. All diese Funktionen sind in Niagarino nicht implementiert. Aus diesem Grund ist es deutlich schlanker und verbraucht weniger Speicher.

Es stellte sich jedoch die Frage, weshalb der Speicherverbrauch am Ende der Ausführung von NiagaraST noch einmal sehr deutlich ansteigt. Auch bei Niagarino ist dieser Anstieg zu verzeichnen, hier ist er aber bei weitem nicht so ausgeprägt. Zur Klärung dieser Frage lohnt sich ein Blick auf das Verhalten des Speicherverbrauchs von NiagaraST im Leerlauf. Dazu wurde wieder mit VisualVM der Speicherverbrauch für eine gewisse Zeit protokolliert.

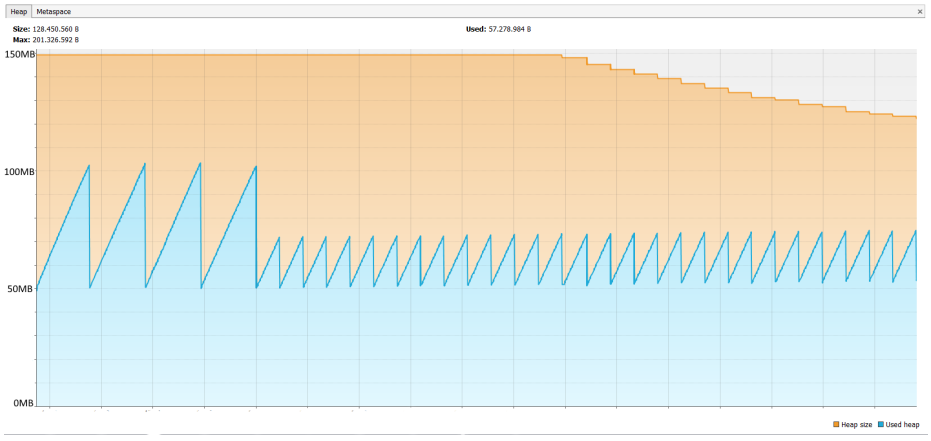

Abb. 25. Heapverbrauch von NiagaraST im Leerlauf

Aus dieser Abbildung (Abb. 25) wird ein Anstieg des Speicherverbrauchs im Leerlauf und dann ein plötzlicher Abfall deutlich, wodurch ein Sägeblattmuster in der Grafik entsteht. Interessant ist in diesem Zusammenhang, dass diese Peaks nach kurzer Zeit kleiner werden. Auch der maximale Heap wird nach einer gewissen Zeit kleiner. Dieses sehr regelmäßige Muster lässt den Schluss zu, dass es sich hier um Aktivitäten des *Garbage-Collectors* der JVM handelt. Auch scheint der *Garbage-Collector* seine Aktivität anzupassen und nach einer gewissen Zeit öfter aktiv zu werden. Dies sorgt dafür, dass der Heap möglichst klein gehalten wird. Es stellte sich jedoch die Frage, weshalb der *Garbage-Collector* im Leerlauf so häufig einspringt. Ein Blick in den Quelltext des NiagaraST-Servers zeigt, welche Objekte erzeugt werden und weshalb diese wieder aufgeräumt werden.

```
1  public void run() {
2  [...]
3      try {
4         // Calls to accept unblock every 500 msecs
5         // to check for stop requests
6               queryEngineSocket.setSoTimeout(500);
7        } catch (SocketException e) {
8          System.err.println("Could not set socket timeout!");
9        }
10
11       do {
12             try {
13                   // Listen for the next client request
14                   Socket clientSocket = null;
15                   clientSocket = queryEngineSocket.accept();
16 [...]
```

An diesem Auszug sieht man deutlich, dass innerhalb einer Schleife ein neues `Socket`-Objekt erzeugt wird um eine eingehende Verbindung zu akzeptieren. Normalerweise blockiert die Funktion `Socket.accept()` die Ausführung, bis tatsächlich eine eingehende Verbindung zustande kommt. Durch diese Methode `setSoTimeout(500)` wird allerdings ein Timeout auf 500 Millisekunden für diesen Block gesetzt. Dies sorgt dafür, dass alle 500 ms ein neues Objekt erzeugt wird und die Referenz auf das alte Objekt verloren geht. Diese Objekte ohne Referenz erzeugen die Peaks im Speicherverbrauch. Der *Garbage-Collector* entfernt diese nach einer gewissen Zeit, wodurch das Sägezahnmuster zustande kommt. Das Schrumpfen der Peaks während der Ausführung ist auf das Verhalten des standardmäßig verwendeten *Linear-Garbage-Collectors* zurück zu führen. Dieser optimiert sich selbst und springt nach einer gewissen Zeit häufiger ein, wenn über einen längeren Zeitraum viele Objekte des gleichen Typs aufzuräumen sind[7].

[7] http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

Kapitel 6

Fazit und Ausblick

Wie in der Einleitung erläutert zielt die vorliegende Arbeit auf eine Performanzoptimierung des noch jungen Datenstromverwaltungssystems Niagarino ab. Dazu wurde zur Steigerung der Performanz ein Paging-Mechanismus implementiert. Das sogenannte Paging ist eine Zusammenfassung von mehreren Tupeln zu einer Page, welche an Stelle einzelner Tupel übertragen wird.

In einem ersten Schritt wurden die Datenstromverwaltungssysteme Aurora, STREAM, Odysseus und NiagaraST als etablierte Systeme betrachtet und insbesondere auf ähnliche Optimierungsmechanismen hin untersucht. Dabei zeigte sich, dass alle Systeme eine Kombination aus verschiedenen Mechanismen zur Performanzoptimierung einsetzen. Das an der Portland State University entwickelte NiagaraST stellt in diesem Zusammenhang das einzige System dar, das einen Pagingmechanismus implementiert hat.

Aufgrund der Architektur von Niagarino stellt Paging eine wirksame Vorgehensweise zur Steigerung der Performance dar. Dabei ist jeder Operator in Niagarino ein eigener Thread. Durch die Übertragung von Pages statt einzelnen Tupeln wird eine häufiges Aktivieren und Deaktivieren der Operatoren verhindert, da ein jeder Operator mit der Abarbeitung einer Page so lange aktiv ist, bis eine neue Page eintrifft. Werden nur einzelne Tupel übertragen, so deaktivieren sich Operatoren häufig nach der Verarbeitung des Tupels, da nicht sofort ein neues Tupel eintrifft. Dieses Deaktivieren und Reaktivieren kostet viel Performance und wird durch das Paging reduziert. Ein erster Ansatz zur Implementierung des Pagings stellte die Implementation eines PagePools dar. Dieser Pool diente der Verwaltung einer vordefinierten Anzahl an Pages und teilte diese zu, wenn sie gebraucht werden. Dahinter stand die Idee zur Wiederverwendung von Pages, um einen häufigen Einsatz des *Garbage-Collectors* zu verhindern. Dieser Ansatz stellte sich jedoch als nicht praktikabel heraus. In der finalen Implementierung besteht der Pagingmechanismus nun aus einer Page-Klasse. Sie beinhaltet ein Array aus Tupeln und hat eine feste Größe. Des Weiteren verfügt die Page über Methoden zur Befüllung und für das Auslesen der Daten aus der Page. Die Pagegröße wird über einen Parameter in einer Java-Properties-Datei festgelegt und kann zur Laufzeit nicht mehr verändert werden. Diese Implementierung stellte sich im Rahmen von Tests als die effizienteste Variante heraus. Insgesamt wurde der Pagingmechanismus als ein zuschaltbares Modul implementiert. So wird in der Properties-Datei festgelegt, ob das Paging verwendet wird oder nicht. Dies hat den entscheidenden Vorteil, dass es dadurch möglich ist, Evaluationen über die Effekte des Pagings zu erstellen. Das Paging wurde so modular implemen-

tiert, so dass eine Erweiterung oder Ergänzung um weitere Pagetypen leicht realisiert werden kann.

Neben der Implementation war insbesondere eine ausführliche Evaluation des Pagingsystems Gegenstand dieser Bachelorarbeit. Dazu musste eine geeignete Versuchsumgebung mit möglichst realitätsnahen Versuchsszenarien erstellt werden. Das beinhaltete insbesondere die Implementation verschiedenster Querypläne und das Beschaffen geeigneter Datensätze. In einem ersten Evaluationsschritt zeichnete sich bereits eine deutliche Performanzverbesserung ab. So war ein deutlicher Geschwindigkeitszuwachs zu verzeichnen. Es offenbarten sich durch diese erste Evaluation aber auch einige Schwächen. Bedingt durch die unterschiedliche Auslastung der Testsysteme wurden mit dem gleichen Queryplan zu verschiedenen Zeitpunkten sehr unterschiedliche Ergebnisse gemessen, was eine weitere Evaluation notwendig machte.
In dieser abschließenden Evaluation wurde aus einer vordefinierten Anzahl von Durchläufen der Mittelwert der Laufzeiten gebildet. Dies sorgt für ein realitätsnahes Ergebnis und dokumentiert die Performanzverbesserung. So konnte durch diese Evaluation gezeigt werden, dass das Paging den gewünschten Effekt bringt und für eine deutliche Performanzverbesserung sorgt. Es konnte auch gezeigt werden, dass sich die Laufzeit ab einer Pagegröße von ca. 2000 Tupel pro Page auf einen festen Wertebereich einpendelt. Auch die Frage nach einer möglichst optimalen Pagegröße konnte beantwortet werden. Dabei galt es nicht nur, eine Verbesserung der Performance zu erreichen. So musste darauf geachtet werden, dass die Page nicht zu groß gewählt wurden, um eine Auswertung der Daten in Echtzeit zu gewährleisten. Es zeigte sich, dass ein Größenbereich von 25–40 Tupeln für eine effiziente Ausführung die geeigneten Werte darstellen. Dies bestätigt auch die feste Pagegröße von 30 Tupeln pro Page im Vergleichssystem NiagaraST.

Entgegen der Erwartungen konnte bei der Analyse des Speicherverbrauchs keine signifikante Zunahme durch das eingeschaltete Paging gezeigt werden. Dies stützte die Entscheidung des Verzichts auf den Pagepool und der Wiederverwertung der Pages.

Der letzte Teil dieser Bachelorarbeit befasste sich mit einem Vergleich von Niagarino mit NiagaraST. Für diesen Vergleich mussten die Daten für NiagaraST aufbereitet und mit Punktierungen versehen werden. Die Auswertung des Vergleichs ergab, dass NiagaraST immer noch effizienter arbeitet als Niagarino. Es zeigte sich aber auch, dass im Vergleich zu NiagaraST Niagarino mit nur ca. 30 Prozent längerer Laufzeit durch das Paging deutlich an Performance gewonnen hat. Eine Analyse des Speicherverbrauchs ergab, dass Niagarino nur knapp die Hälfte des Speichers von NiagaraST benötigt. Dies ist vor allem der schlanken Implementierung von Niagarino zu verdanken.

In einem zukünftigen Projekt könnte man die feste Größe der Pages durch eine dynamische Größe ersetzen. Durch dieses Verfahren würde man ein sich selbst optimierendes System schaffen. Realisiert werden könnte dies durch eine Rückmeldung der benötigten Verarbeitungszeit der Operatoren an das Sys-

tem. Dies würde dann die Pagegröße zur Laufzeit optimieren und anpassen, so dass möglichst wenige Deaktivierungen von Operatoren notwendig wären. Möglicherweise lässt sich durch dieses Verfahren ein zusätzlicher Performanzgewinn realisieren.

Rückblickend zeigen die guten Ergebnisse der Evaluation des Pagingsystems, das dieses ein wichtiger Teil einer notwendigen Reihe von Optimierungen an Niagarino darstellt. Dabei zeigt der Vergleich mit NiagaraST und den anderen Datenstromverwaltungssystemen, dass eine Kombination von verschiedenen Performanzoptimierungen die besten Ergebnisse liefert. Das Paging in Niagaraino stellt hierzu einen wichtigen Bestanteil dieser Kombination bereit.

Literatur

1. Daniel J Abadi, Yanif Ahmad, Hari Balakrishnan, Magdalena Balazinska, Mitch Cherniack, Jeong hyon Hwang, Samuel Madden, Anurag Maskey, Er Rasin, Mike Stonebraker, Nesime Tatbul, and Ying Xing. The aurora and borealis stream processing engines. In *Data Stream Management: Processing High-Speed Data Streams*. Springer, 2007.
2. Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Uğur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the borealis stream processing engine. In *Proc. Biennial Conf. on Innovative Data Systems Research (CIDR 2005)*, Asilomar, CA, January 2005.
3. H.-Jürgen Appelrath, Dennis Geesen, Marco Grawunder, Timo Michelsen, and Daniela Nicklas. Odysseus: A highly customizable framework for creating efficient event stream management systems. In *Proc. Int. Conf. on Distributed Event-Based Systems (DEBS '12)*, pages 367–368. ACM, 2012.
4. Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. Stream: The stanford data stream management system. Technical Report 2004-20, Stanford InfoLab, 2004.
5. Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams: A new class of data management applications. In *Proc. Int. Conf. on Very Large Data Bases (VLDB '02)*, pages 215–226. VLDB Endowment, 2002.
6. Dennis Geesen and Marco Grawunder. Odysseus as platform to solve grand challenges: Debs grand challenge. In *Proc. Int. Conf. on Distributed Event-Based Systems (DEBS '12)*, pages 359–364. ACM, 2012.
7. Jeffrey Naughton, David Dewitt, David Maier, Ashraf Aboulnaga, Jianjun Chen, Leonidas Galanis, Jaewoo Kang, Rajasekar Krishnamurthy, Qiong Luo, Naveen Prakash, Ravishankar Ramamurthy, Jayavel Shanmugasundaram, Feng Tian, Kristin Tufte, and Stratis Viglas. The niagara internet query system. *IEEE Data Engineering Bulletin*, 24:27–33, 2001.
8. Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. Exploiting punctuation semantics in continuous data streams. *Knowledge and Data Engineering, IEEE Transactions on*, 15(3):555–568, May 2003.
9. Kristin Tufte, Jin Li, David Maier, Vassilis Papadimos, Robert L. Bertini, and James Rucker. Travel time estimation using niagarast and latte. In *Proc. Int. Conf. on Management of Data (SIGMOD '07)*, pages 1091–1093, 2007.
10. Stanley B. Zdonik, Michael Stonebraker, Mitch Cherniack, Ugur Çetintemel, Magdalena Balazinska, and Hari Balakrishnan. The aurora and medusa projects. *IEEE Data Engineering Bulletin*, 26, 2003.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur mit erlaubten Hilfsmitteln angefertigt habe.

Konstanz, 14. Juni 2015

Maximilian Ortwein