Institut für parallele und verteilte Systeme

Abteilung Anwendersoftware

Universität Stuttgart
Universitätsstraße 38
D – 70569 Stuttgart

Diplomarbeit-Nr.: 2776

Entwurf und Implementierung von Basisoperatoren für Nexus

Markus Dörr

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. Bernhard Mitschang
Betreuer:	Dipl. Inf. Matthias Großmann
begonnen am:	23.06.08
beendet am:	07.01.09
CR-Klassifikation:	C.2.4, H.2.4 , H.3.4

Inhaltsverzeichnis

1. Einleitung

Wenn man sich der Datenverarbeitung und -speicherung heutzutage zuwendet, so sind die bisher meist genutzten Datenquellen die Datenbanken. Diese zeichnen sich vor allem dadurch aus, dass sie einen Datenbestand effizient und konsistent verwalten. Der Datenbestand der Datenbanken ist dabei, aufgrund des beschränkten Speicherplatzes eines Computers, endlich. Natürlich ist es möglich riesige Datenbanken zu erstellen, die sich über Gigabytes von Daten erstrecken. Da die Menge der Informationen aber absehbar ist, können ohne Probleme jegliche Funktionen auf diesen Daten ausgeführt werden, die für eine Verwaltung und Nutzung dieser Daten von Nöten ist, wobei die Effizienz solcher Datenbanken im Zusammenhang mit ihrer Größe steht.

Neben den Informationen aus Datenbanken treten immer häufiger auch andere Informationen und Datenquellen in den Vordergrund, die Datenströme. Diese neuen Datenquellen und Informationen resultieren unter anderem aus der Notwendigkeit und Nützlichkeit, verschiedene Systeme und Gegenstände kontinuierlich zu beobachten oder zu überwachen. Dies geschieht unter anderem mittels Sensoren. Durch diese Kontinuität der Datenerhebung ergibt sich aber gleichzeitig die größte Diskrepanz zur bisherigen Datenverarbeitung mittels Datenbanken: Die Massen an Daten, von denen laufend neue produziert werden, können nicht mehr gespeichert werden. Die Speicherung von Daten ist bisher jedoch für einige Operationen der Datenverarbeitung ein essentieller Vorgang, ohne diesen ein weiteres Vorgehen teilweise nicht möglich ist.

Dieser Schritt, Daten zu verarbeiten, die alleine auf Datenbanken liegen, hin zur Verarbeitung von Datenströmen soll auch im NEXUS-Projekt vollzogen werden. Basierend auf dem NEXUS-Projekt sollen nun diese, für Datenströme inkompatiblen Operationen, kompatibel gemacht, sowie andere Operationen für Datenströme angepasst und optimiert werden.

Einen Überblick über die aktuelle Architektur, sowie über die geplanten Änderungen bezüglich der datenstromverarbeitenden Fähigkeiten des NEXUS-Projekts, wird in Kapitel **2** gegeben. In Kapitel **3** werden die, für die zentrale Problemstellung dieser Diplomarbeit, verschiedenen Lösungsansätze und deren Brauchbarkeit, beziehungsweise deren Einfluss auf die Lösung der Problemstellung, vorgestellt. Stromeigenschaften von Datenströmen in Kombination mit den so genannten Punctations bilden den ersten Lösungsansatz, um ein datenstromkompatibles Operatorenkonzept für das NEXUS-System zu erstellen. In Kapitel **4** wird näher auf diese Stromeigenschaften von Datenströmen eingegangen und wie man sie zum Vorteil nutzen kann. Neben der Funktion der Punctations werden in Kapitel **5** auch die Unterschiede zwischen Punctations und Stromeigenschaften beschrieben.

Der zweite ausgewählte Lösungsansatz wird in Kapitel **6** beschrieben und in wie weit er in das Konzept einfließt. In Kapitel **7** wird das eigentliche Operatorenkonzept vorgestellt. Es werden weiter die einzelnen Operatoren in ihrer Funktionsweise, sowie in ihren Vor- und Nachbedingungen, erläutert. Diese Erläuterung wird durch verschiedene Beispielanfragen unterstützt. Angaben zur Implementierung der Operatoren, sowie für die Implementierung getroffene Annahmen, befinden sich in Kapitel **8**. Des Weiteren befinden sich in diesem Kapitel eine konkrete Realisierung einer Beispielanfrage, sowie Aussagen eines Performanztests über die Operatoren. Abschließend werden verschiedene Möglichkeiten für zukünftige Erweiterungen und Verbesserungen in Kapitel **9** aufgeführt. In Kapitel **10**, dem Anhang, befinden sich verschiedene Code-Beispiele, auf die in Kapitel **8** verwiesen wird.

2. Überblick über das NEXUS - Projekt

Das NEXUS-Projekt ist ein offenes Pervasive-Computing – System. Hierbei steht im Mittelpunkt die Idee, einen bestehenden realen und globalen Kontext in die digitale Welt zu überführen. Das

bedeutet, dass erklärende Informationen für alltägliche Dinge, wie zum Beispiel Türaufschriften, die beschreiben, was hinter einer Türe ist, oder Verkehrszeichen, die den Verkehr regeln, von der realen Welt in die digitale Welt portiert werden können, um sie zum Beispiel direkt über einen PDA digital verfügbar zu machen. Hierbei spielt vor allem die Verarbeitung und Bereitstellung von räumlichen Daten eine große Rolle. Für die Darstellung der Daten der realen Welt wurde ein objektorientiertes Informationsmodell entworfen, das Augmented World Model (AWM). Repräsentiert werden diese Informationen mittels einer Sprache in XML-Notation, der Augmented World Model Language (AWML). Analog dazu wurde eine Anfragesprache entworfen, die ebenfalls auf diesem Datenmodell und der XML-Notation basiert, die Augmented World Query Language (AWQL).

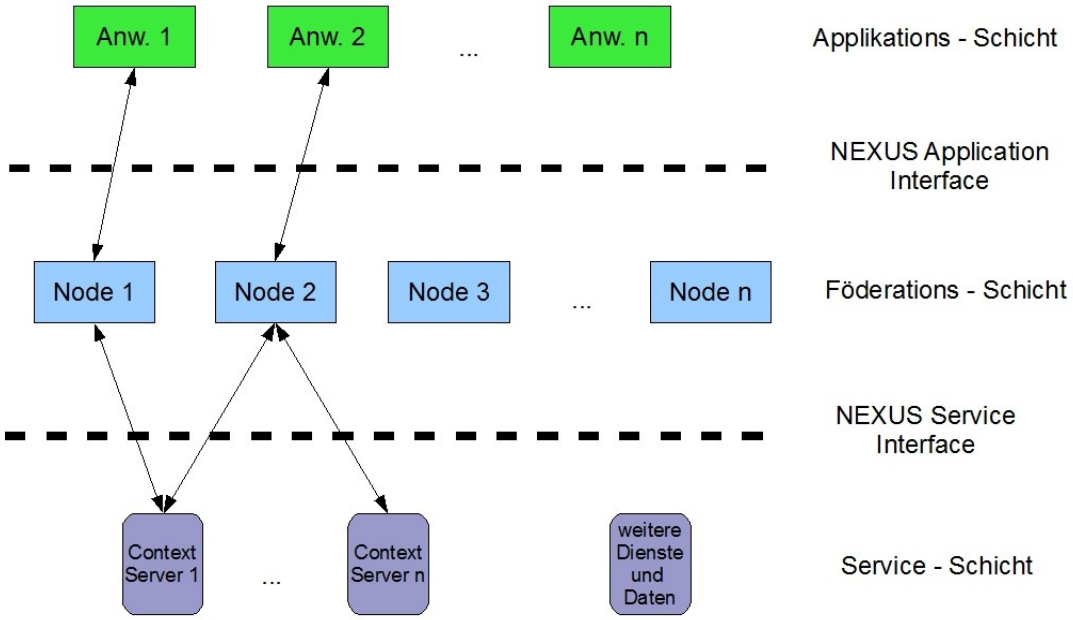

Abbildung 2.1: Aktuelle 3-schichtige Architektur der NEXUS-Plattform

2.1 Aktuelle Architektur

Die gesamte NEXUS-Plattform gliedert sich dabei in ein 3-schichtiges System, wie sie vereinfacht in Abbildung **2.1** beschrieben ist. Alle für eine Anfrage relevanten Daten liegen auf verschiedenen, verteilten Servern, den Context-Servern. Diese Datenanbieter werden in der Service-Schicht zusammengefasst. Diese Datenhaltung in der Service-Schicht ist vergleichbar mit den Servern einer Client-Server Architektur, wie sie zum Beispiel im Internet zu finden ist.. Um nun alle diese Daten zu erhalten, verbindet sich eine NEXUS-Software in der Applikation-Schicht, vergleichbar mit einem Client der Client-Server Architektur, nicht direkt mit den Servern, sondern mit der so genannten Föderation, welche einer verteilten Middleware-Schicht gleichkommt. Die Föderation, oder Middleware-Schicht, besteht aus mehreren Servern, den Nodes. Diese sammeln alle für die entsprechenden Anfragen notwendigen Daten aus der Service-Schicht, bereiten sie auf und geben sie an die Applikation-Schicht zurück.

Die Schnittstelle zwischen der Applikation-Schicht und der Föderation-Schicht ist das NEXUS-Application Interface, wie in Abbildung **2.1** dargestellt. Die Schnittstelle zwischen der Föderation- und der Service-Schicht ist das NEXUS-Service Interface. Diese Schnittstellen unterscheiden sich praktisch nur durch ihren Namen, da aus der Service-Schicht auf die gleiche Art auf die Nodes zugegriffen wird, wie aus der Applikation-Schicht. Genauere Informationen über die NEXUS-Plattform Architektur sind [1] zu entnehmen.

Daten und Informationen, die Gegenstände und andere erfassbare Dinge beschreiben, werden in der NEXUS-Plattform von Datenobjekten repräsentiert. Diese Datenobjekte werden durch eine XML-ähnlichen Notation, der *Augmented World Modelling Language (AWML)* repräsentiert. Die Datenhaltung spielt dabei eine untergeordnete Rolle, da jede Art der Datenhaltung angebunden werden kann, wenn sie zusätzlich einen Wrapper besitzt, der die Konvertierungen ins AWML Format durchführt. Durch diese XML-ähnliche Darstellungsart der Daten ergibt sich weiter die Möglichkeit, mehrere verschiedene Datentypen zu so genannten Multitypen zu kombinieren. So besteht zum Beispiel für eine Kirche die Möglichkeit, sowohl vom Typen „Gebäude“ als auch vom Typen „Sehenswürdigkeit“ zu sein.

Das NEXUS-Metadaten-Modell, beschrieben in [2], bietet die Möglichkeit, Informationen über die verschiedenen Attribute und Daten der Objekte im NEXUS-System, also Daten über Daten, einzubringen. Basierend darauf können auch verschiedene Aussagen und Forderungen über Daten, Objekte und die Relationen zwischen Objekten getroffen werden (siehe auch Kapitel **4**), so zum Beispiel wie genau ein Sensor misst oder in welchem Abstand die Daten gesendet werden. Eines der wichtigsten Anwendungsgebiete dieser Metadaten ist zum Beispiel die Nutzung einer Gültigkeitszeit von Daten. So kann bestimmt werden, ob bestimmte Daten, wie Positionsdaten, noch aktuell sind, oder ob sie verworfen werden können, beziehungsweise, welche Daten noch genutzt werden können.

Derzeit können nur statische Anfragen an das System gestellt werden. So ist zum Beispiel ein Verfolgen von Positionen bestimmter Personen in einem definierten Umkreis derzeit nicht möglich. Deshalb sollen verschiedene Änderungen an der NEXUS-Plattform durchgeführt werden, um diese zu erweitern ([3] und Kapitel **2.2**).

2.2 Geplanten Veränderungen des Systems

Zur Verbesserung und Steigerung der Leistungsfähigkeit, sowie der Flexibilität des NEXUS-Systems, sollen verschiedene Änderungen durchgeführt werden ([3]). Diese sollen das Ziel haben, nicht nur die Leistungsfähigkeit der Plattform zu steigern, sondern auch die Optionen der Verarbeitung von Daten zu erweitern.

So wird eine Homogenisierung verschiedener Verarbeitungskonzepte, wie zum Beispiel der Verarbeitung von klassischen Tupelmengen, aber auch von datenstrombasierten Daten oder Ereignissen, durch ein zu entwickelndes Operatorenkonzept (wie in Kapitel **7** beschrieben) erfolgen. Dieses soll dann in einem Virtualisierungsschritt auf die jeweils zugrunde liegende Systemstruktur möglichst optimal abgebildet werden. Dadurch können unter Anderem anwendungsspezifische und flexible Verarbeitungskonzepte realisiert werden.

Bei diesem Virtualisierungsschritt werden Anfragen an das System nicht mehr nur von einzelnen Nodes in der Föderation-Schicht bearbeitet. Anfragen an das System erhält weiterhin ein Node, dieser verteilt allerdings die Aufgaben, wie das Durchführen einer Selektion auf verschiedenen Daten, an andere Knoten und überwacht die Durchführung. Es erfolgt also eine virtuelle Abbildung der Operationen und Operatoren auf verschiedene Knoten in der Föderation-Schicht. Diese Verteilung der Anfrage führt dazu, dass die gesamte Arbeitslast nicht mehr nur von einem Knoten getragen werden muss, sondern dass sie auf verschiedene Knoten verteilt wird, was wiederum zu einer besseren Skalierung des Systems führt.

Neben der Virtualisierung, die nur die Föderation-Schicht betrifft, werden auch

Verbesserungen und Veränderungen in der Service-Schicht durchgeführt. So werden Daten in Datenbanken nicht länger zwangsweise zusammen, nach Objekten geordnet, gespeichert und verwaltet. Sie werden vielmehr nach ihrer Beschaffenheit und Ausprägung verschiedenen Servern zugeordnet. Es werden zum Beispiel alle räumlichen Daten verschiedener Objekte in einem Server verwaltet, getrennt von ihrem Namen oder anderen Ausprägungen der Attribute. Identifiziert und zusammengefügt werden diese Objekte durch ihren Object-Identifier (OID). Auf die Virtualisierung oder die Datenverteilung wird in dieser Arbeit jedoch nicht näher eingegangen.

Die Föderation- sowie die Service Schicht wird dahingehend verändert, dass dem System die Fähigkeit hinzugefügt wird, neben Daten aus Datenbanken, auch Datenströme verarbeiten zu können. Dies jedoch führt zu verschiedenen Problemen, die in Kapitel **3.1** erläutert werden.

Durch diese verschiedenen Veränderungen sollen neue, dynamische und somit mächtigere Anfragen ermöglicht werden, die die Leistungsfähigkeit der NEXUS-Plattform steigern.

3. Related Work

3.1 Problemstellung

Wie schon weiter oben erwähnt, ist das eigentliche Problem die Verarbeitung der Daten, die bei kontinuierlichen Datenströmen anfallen. Da eine komplette Speicherung der Daten auf Grund der unendlich großen Masse von Daten auszuschließen ist, müssen Operationen beziehungsweise Operatoren erstellt werden, die eine direkte Verarbeitung der Daten sicherstellen. Für einige Operationen, die bereits auf Datenbanken angewendet werden, ist dies auch bei kontinuierlichen Datenströmen generell kein Problem. Hier sei zum Beispiel die Selektion zu nennen, da für diese theoretisch nur ein Vergleich zu bewältigen ist. Im Gegensatz dazu gibt es auch Operationen, die einen internen Speicher nutzen, um ihre Aufgaben zu erfüllen. Diese Operationen lassen sich entweder den zustandsbehafteten oder den blockierenden Operationen zuteilen.

Zustandsbehaftete Operationen besitzen einen inneren Speicher, der verschiedene Zustände für die Daten hält, so zum Beispiel beim kartesischen Produkt. In diesem inneren Speicher wird ein Abbild der Elemente angelegt, die mit anderen Elementen kombiniert werden müssen. Für zwei unendliche Datenströme müsste dieser interne Speicher, der die Abbilder verwaltet, ebenfalls unendlich groß sein. Kontinuierliche Datenströme führen hier also zu einem Überlaufen des internen Speichers.

Bei blockierenden Operatoren wie der Sortierung von Daten, oder dem Group-by - Operator, ist ebenfalls ein Wissen über alle zu sortierenden / gruppierenden Daten notwendig. Es werden alle zu sortierenden / gruppierenden Elemente im Speicher zwischengespeichert, bis das notwendige Wissen, also zum Beispiel die zusammen zu gruppierenden Elemente oder die Reihenfolge der Sortierung der Daten, feststeht. Bis alle Elemente gespeichert wurden, wird die weitere Verarbeitung der Daten blockiert. Auch hier kommt es somit zu einem Überlaufen des internen Speichers, falls dieser Operator zur Verarbeitung eines unendlichen Datenstroms benutzt werden soll.

3.2 Lösungsansätze

Um nun die eigentliche Problemstellung bearbeiten zu können, wurden einige Ideen und Abhandlungen, die sich auf den Umgang mit Datenobjekten und Datenbanken, sowie einer intelligenten Verarbeitung von kontinuierlichen Datenströmen beziehen, erfasst und analysiert. Daten und Informationen, die Gegenstände und andere erfassbare Dinge beschreiben, werden in der NEXUS-Plattform von Datenobjekten repräsentiert (siehe Kapitel **2.1**).

Als eine hilfreiche Grundlage für die Interaktion von Benutzer und Datenobjekt, sowie zwischen den Datenobjekten, ist die dritte Spezifikation für einen Objekt-Daten-Standard, der Objekt Data Management Group (ODMG) [4], anzusehen. Hier wird beschrieben, wie durch diese Spezifikation (kurz: ODMG 3.0) ein Datenmodell für den Umgang mit Datenobjekten definiert werden kann. Dieses Buch bildet also quasi die Grundlage für ein Verständnis im Umgang mit Datenobjekten und objektbasierten Datenbanksystemen. Weiter wird in diesem Buch die eigenständige Sprache Objekt Query Language (OQL) eingeführt und beschrieben. Diese Sprache besitzt eine ähnliche Syntax wie SQL, bei der sich die Operatoren ebenfalls nach Belieben frei miteinander kombinieren lassen. Zusätzlich bietet diese Sprache aber die Möglichkeit, innerhalb dieser SQL-ähnlichen Anfragen objektspezifische Methoden auszuführen, und deren Ergebnisse in die Anfragen mit einzubinden. Dies führt dazu, dass sehr mächtige Anfragen gestellt werden können. Jedoch bietet die Sprache OQL derzeit keine expliziten Möglichkeiten der Datenstromverarbeitung.

Im Gegensatz zum Lösungsansatz mit der Sprache OQL, bietet der Ansatz von C. Zaniolo et al. in [5] und [6] die Möglichkeit, sich direkt mit Datenströmen auseinander zu setzen. Diese Abhandlungen sprechen dabei vor allem die oben genannten Hauptprobleme an, dass zustandsbehaftete und blockierende Operationen nicht mit kontinuierlichen Datenströmen vereinbar sind. Diese Abhandlungen beschreiben die Anfragesprache ESL (Expressive Stream Language). Diese wiederum basiert selbst auf der Anfragesprache SQL. Die Hauptunterschiede zwischen den beiden Sprachen beziehen sich auf die so genannten Aggregate. Aggregate entsprechen Funktionen, wie zum Beispiel das „avg“- Aggregat in SQL, das den Durchschnitt verschiedener Elemente in Datenbanksystemen berechnet. In ESL werden nun diese Aggregate aus SQL so verändert, dass sie kompatibel zu Datenströmen werden. Diese Kernidee ist das so genannte User-Defined Aggregates-Konzept (UDA-Konzept). Eine genau Erläuterung der Veränderungen erfolgt in Kapitel **6.** Des Weiteren werden in ESL Timestamps genutzt, Datenobjekten zugeordnete Zeitstempeln kombiniert, um so eine Ordnung für Datentupel zu erhalten. Weiter ist in ESL die Ausführung von blockierenden und zustandsbehafteten Operationen nur über Fensteroperatoren, den so genannten Window Aggregates, möglich. Dabei werden Elemente aus dem eingehenden Datenstrom, oder den Datenströmen, mittels einem Puffer zwischengespeichert. Je nach Größe des Puffers können so eine Anzahl von Elementen zwischengespeichert und verarbeitet werden. Dies bedeutet also, dass der potenziell unendliche Datenstrom in zwar unendliche viele, aber endlich große Teile unterteilt wird. Auf diesen endlich vielen Daten im Zwischenspeicher können nun die blockierenden und zustandsbehafteten Operationen durchgeführt werden. Nicht-blockierende Aggregate und Operationen sind direkt ausführbar. Das Konzept der ESL bietet noch weitere interessante Ausführungen, die den Arbeiten zu entnehmen sind.

Eine ebenfalls neue Anfragesprache wird in [7] mit der Sprache CQL (Continuous Query Language) geschaffen. CQL ist ebenfalls eine komplett neue Sprache, die auch auf der Anfragesprache SQL basiert. Die Kernidee hinter CQL ist, die Verarbeitung von Datenströmen in drei Phasen zu gliedern. Im Allgemeinen werden dabei die Datenströme durch Operatoren in Relationen überführt, anschließend mittels Operatoren verarbeitet und wieder zurück überführt. Die Operatoren dieser drei Phasen können jedoch beliebig miteinander kombiniert werden.
Man unterscheidet bei CQL zwischen zwei verschiedenen Klassen von Datenströmen. Die eine Klasse bilden die so genannten *base-streams*, welches alles eingehende Ströme sind. Alle ausgehenden und berechneten Datenströme, die *derived-streams*, bilden die zweite Klasse.

Bei der Überführung eines solchen *base-streams* (eingehender Datenstrom) in Relationen mittels eines Operators, also der ersten Phase, spricht man vom *stream-to-relation* Mapping. Dies wird mittels eines Fensteroperators durchgeführt. Dabei verwendet der Operator einen Puffer, der Elemente aus dem eingehenden Datenstrom zwischenspeichert. Hierbei ist die Puffergröße variabel

und kann bei jeder Verwendung des Operators als Parameter neu bestimmt werden. Auf diesem Puffer können dann verschiedene Operationen in der zweiten Phase durchgeführt werden. Hierbei werden Relationen mit normalen SQL-Anfragen wiederum in Relationen überführt (*relation-to-relation* Mapping). Hier findet die eigentliche Verarbeitung der Daten aus dem Datenstrom statt. Dies geschieht auf gleiche Weise, wie es in SQL der Fall ist.

Die dritte Phase ist die Überführung der Relationen zurück in Datenströme, also in die zweite Klasse der Datenströme, die *derived-streams*. Hierbei spricht man vom *relation-to-stream* Mapping. Dies wird mittels drei verschiedenen Stream-Operatoren durchgeführt, die verschiedene Arten von Datenströmen erzeugen, die jedoch alle zur Klasse der *derived-streams* gehören.

Einer der drei datenstromerzeugenden Operatoren ist der Insert-Stream-Operator, oder *ISTREAM*-Operator. Mit dem *ISTREAM*-Operator wird aus einer Relation ein Datenstrom erstellt, bei dem immer, wenn ein Datensatz in die Relation eingefügt wird, ebenfalls eine Kopie in den Datenstrom eingefügt wird. Dies geschieht jedoch nur dann, wenn der in die Relation einzufügende Datensatz nicht bereits in der Relation vorhanden ist. Ist dies bereits der Fall, der Datensatz also bereits in der Relation vorhanden, so wird er nicht in den ausgehenden Datenstrom eingefügt. Der zweite Operator ist der Delete-Stream-Operator, oder *DSTREAM*-Operator. Mit dem *DSTREAM*-Operator wird aus einer Relation ein Datenstrom erstellt, in den immer dann ein Datensatz eingefügt wird, wenn dieser Datensatz aus einer Relation entfernt wurde. Der letzte Operator ist der Relation-Stream-Operator, oder *RSTREAM*-Operator. Der *RSTREAM*-Operator wandelt eine komplette Relation, zum Beispiel eine komplette Datenbanktabelle, in einen Datenstrom um. Dabei werden die Daten der Relation zum aktuellen Zeitpunkt verwendet.

Der Grund für die Einteilung der Verarbeitung von Daten in die drei Phasen ist, dass Implementierungstechniken und das bereits vorhandene Wissen über das *relation-to-relation* Mapping wiederverwendet werden können. Die verschiedenen Arten der Operatoren und deren Einsatz werden in [7] näher beschrieben.

Mit PIPES (Public Infrastructure for Processing and Exploring Streams) wird in [8] eine komplette Infrastruktur vorgestellt, die ein integraler Bestandteil von XXL ist. Dies wiederum ist eine Java-Bibliothek für komplexe Anfrageverarbeitung über heterogene und relationale Datenquellen. In PIPES wird für die Verarbeitung von Daten ein komplettes, anwendungsspezifisches Datenstrom-Management-System (DSMS) erstellt, das intern drei Typen von Knoten für die Modellierung eines Operatorengraphs bereitstellt: Die Quellen, Operatoren und Senken. Eine Quelle transferiert dabei ihre Daten an die auf ihr subskribierten Senken. Eine Senke wiederum kann auf mehreren Quellen subskribiert sein. Der Operator steht zwischen der Quelle und Senke und verarbeitet die von der Quelle gesendeten Daten und gibt die Ergebnisse an die Senke weiter. Als Operator steht in PIPES der so genannte Fensteroperator zur Verfügung, der entweder mit einem fixen oder einem gleitenden Fenster benutzbar ist. Das Prinzip des Fensteroperators, wie es weiter oben schon beschrieben wurde, wird hier durch fixe oder gleitende Fenster modifiziert. Der fixe Fensteroperator lädt hierbei Elemente in den Puffer, verarbeitet sie und löscht anschließend den Puffer. Der Operator mit den gleitenden Fenstern hingegen entfernt verarbeitete Daten sofort aus dem Speicher und lädt ein neues Objekt nach. Anschließend wird auf dem neuen Pufferzustand die Operationen durchgeführt. Durch das System wird jedes Element mit einem zeitbezogenen Gültigkeitsintervall versehen. Dieses Intervall bestimmt, wie lange ein Fensteroperator ein bestimmtes Element berücksichtigen muss. Auf diese Weise werden kontinuierliche Datenströme in endliche Subströme unterteilt. Dadurch ist es möglich, die üblichen Operatoren auf die endlich vielen Elemente anzuwenden. Verschiedene Aggregate, wie die Durchschnittsbildung, können nur mittels annähernden Algorithmen erfüllt werden, was zu einer Ungenauigkeit des Ergebnisses führt.

Ein weiterer interessanter Ansatz ergibt sich aus [9] durch die so genannten Punctations. Mittels dieser Punctations, die direkt in den Datenstrom eingefügt werden, werden verschiedene

Informationen über diese kontinuierlichen Datenströme eingefügt. Durch diese Informationen können kontinuierliche Datenströme in virtuell-endliche Datenströme aufgeteilt werden, das heißt, die kontinuierlichen Datenströme werden intern in endliche Subströme geteilt. Dies lässt sich leicht anhand eines Beispiels verdeutlichen:

Es soll kontinuierlich die stündliche Durchschnittstemperatur eines Raumes gemessen werden. Ein Sensor sendet die Daten mit Zeitstempeln versehen. Zu jeder vollen Stunde fügt der Sensor eine Punctation ein, die angibt, dass alle Messungen der letzten Stunde gesendet wurden. Die Durchschnittsbildung kann also abgeschlossen werden, sobald die Punctation erhalten wird. Alle Elemente einer Stunde bilden hier also einen virtuellen, endlichen Substrom. Weitere Informationen befinden sich in Kapitel **5**.

3.3 Auswahl der Ansätze

Für die Lösung der eigentlichen Aufgabenstellung, ein Operatoren-Konzept für Anfragen an kontinuierliche Datenströme zu erstellen, und die damit verbundenen Probleme, wie bereits unter Kapitel **3.1** aufgezeigt, wurden diese verschiedenen Ansätze analysiert. Ausschlaggebend dafür, ob die verschiedenen Ansätze in die Lösung mit einfließen könnten, waren ihre jeweilige Anwendbarkeit, ihre Komplexität und natürlich ihre Problemlösungsfähigkeit, das heißt in wie weit sich ihr Vorgehen, hinsichtlich der Behandlung der Unendlichkeit von Datenströmen, mit der NEXUS-Plattform vereinbaren ließen. Dabei sollte beachtet werden, dass eine einfache, effektive aber auch erweiterbare Lösung für das Operatoren-Konzept gefunden wird, die in die NEXUS-Plattform eingebettet werden sollte.

Da die eigentliche Infrastruktur, beziehungsweise das Rahmenwerk, durch die NEXUS-Plattform bereits vorgegeben ist, mussten nur Operatoren für das Ausführungsrahmenwerk, sowie unterstützende Hilfsklassen, entwickelt werden, die Anfragen an kontinuierliche Datenströme stellen können. Dabei wird unterschieden zwischen elementaren Operatoren und Aggregaten. Es wurden zwei Ansätze ausgewählt. Diese können effektiv und erweiterbar in die NEXUS-Plattform eingebettet werden, und gleichzeitig auf einfache Art und Weise an die notwendigen Rahmenbedingungen angepasst werden. Hierbei handelt es sich zum einen um das UDA-Konzept in abgewandelter Form. Zum anderen handelt sich um die Punctations, mittels denen unendliche Datenströme in endliche Teilströme getrennt werden können.

Das UDA-Konzept

Dem UDA-Konzept liegt normalerweise die komplette Sprache ESL zugrunde, in deren Rahmenwerk die Anfragen verarbeitet werden. Dadurch, dass für die NEXUS-Plattform aber direkt in Java implementiert wird und das eigene Ausführungsrahmenwerke besitzt, konnten nur die Kernideen des UDA-Konzeptes übernommen werden und für die NEXUS-Plattform angepasst werden. Das UDA-Konzept wird hierbei für die Erstellung von Aggregaten, wie zum Beispiel einem COUNT-Operator und weiteren eigenen einfacheren Funktionen genutzt, die für die Verarbeitung von kontinuierlichen Datenströmen angewandt werden sollen. Dieses Konzept bietet die Möglichkeit einfach und erweiterbar in der NEXUS-Plattform integriert zu werden, und falls notwendig, zu einem späteren Zeitpunkt auf einfache Art und Weise neue Aggregate zur NEXUS-Plattform hinzu zu fügen.

Punctations

Generell sind für die Aufteilung eines kontinuierlichen Datenstroms in endliche Teile zwei Möglichkeiten geboten. Zum einen die Verarbeitung über einen Fensteroperator, der entweder

über variable oder fixe Fenster Elemente eines Datenstromes erfasst und verarbeitet. Hierbei tritt aber häufig das Problem zu Tage, dass keine Informationen über weitere Elemente zur Verfügung stehen, und somit das Ergebnis einer Verarbeitung nur approximativ ist. Im Gegensatz dazu bietet die Kombination aus Punctations und Aussagen über Stromeigenschaften die Möglichkeit, diese Ungenauigkeiten zu begrenzen, indem die verschiedenen Stromeigenschaften mit in die Verarbeitung einfließen können. Im Gegensatz zu Fensteroperatoren unterteilen Punctations einen kontinuierlichen Datenstrom in endliche Subströme, nicht nur für die Dauer der direkten Verarbeitung der Daten, das heißt nicht nur für die Zeit der Zwischenspeicherung der Daten in einem temporären Zwischenspeicher eines Operators, der anschließenden Verarbeitung und der Freigabe dieser Daten, sondern auch darüber hinaus. Dies geschieht dadurch, dass spezielle Datenobjekte in den Datenstrom eingefügt werden können, die die einzelnen Aussagen über Elemente des Datenstromes beinhalten. Solche Information können sich zum Beispiel darauf beziehen, dass eine gewisse Relation zwischen Elementen im Datenstrom vorherrscht, wie zum Beispiel, dass alle Elemente, die nach der Punctation auftreten, einen bestimmten Wert eines definierten Attributs nicht überschreiten.

Der Lösungsansatz der Punctations wird mit einem weiteren, eigenen Konzept erweitert. Zusätzlich können bei der Verbindung auf einen Datenstrom zwischen Operator und Datenquelle verschiedene Stromeigenschaften ausgehandelt werden. Die Kombination zwischen Punctations und Verwendung von Stromeigenschaften ist effektiv und einfach zu integrieren und bietet ebenso die Option der späteren Erweiterbarkeit. Weiter ist sie kein monolithisches Rahmenwerk, das Anpassungen an sich und der NEXUS-Plattform erzwingt oder notwendig macht.

ODMG und OQL

Das ODMG Datenmodell hat verschiedene Parallelen zum derzeitigen Datenmodell des NEXUS-Systems. Es ist durch OQL sogar zu mächtigeren Anfragen fähig als NEXUS, bietet jedoch explizit keine datenstromverarbeitenden Operatoren an. Es müsste also das ODMG Datenmodell für die Verarbeitung von Datenströmen an das bereits existierende Datenmodell der NEXUS-Plattform angepasst werden, und zusätzlich in sofern erweitert werden, dass die Möglichkeit bestünde, Datenströme direkt zu verarbeiten.

Die Anpassung an das NEXUS-Datenmodell würde diverse Schwierigkeiten mit sich bringen, wie die weitere Verwendung des Metadatenmodells oder der Multitypen, die bereits weiter oben in Kapitel **3.2** beschrieben wurden. Während das NEXUS-System mit verschiedenen Datentypen und den Multitypen umgehen kann, akzeptiert OQL nur exakt definierte Datentypen. Dies würde bei der Anpassung an die Multitypen erhebliche Probleme mit sich bringen. Des Weiteren würde dies einen größeren Aufwand bedeuten, als dies die Anpassung der NEXUS-Plattform mittels anderen, einfacheren Mitteln, wie den Lösungsansätzen, die ausgewählt wurden, darstellt.

CQL

Die Anfragesprache CQL wurde nicht als Problemlösungsansatz für das NEXUS-Projekt ausgewählt. Sie besitzt zwar den Vorteil, dass vorhandene SQL-Anfragen wiederverwendet werden können, jedoch werden zum Zwecke des *stream-to-relation* Mappings Fensteroperatoren genutzt. Fensteroperatoren teilen kontinuierliche Datenströme temporär, also nur für die Zeit der Verarbeitung in Operatoren, in endliche Bereiche ein. Auch machen sie keine Aussagen über Elemente, die jenseits dieser Fenstergrenze liegen. In diesem Bereich sind Punctations die bessere Wahl.

PIPES

Wie NEXUS ist auch PIPES in Java implementiert und bietet sogar die Möglichkeit bereits vorbereitete Bibliotheken und Klassen zu nutzen. Da PIPES eine eigenständige Infrastruktur mit sich bringt, müsste diese und die NEXUS-Plattform ineinander integriert werden. Dies würde aber, verglichen mit den gewählten Lösungsansätzen, einen unnötigen Mehraufwand bedeuten.

4. Stromeigenschaften

Datenströme können von verschiedenen Datenquellen ausgehen. Von der Art der Datenquelle hängen auch die Eigenschaften des jeweiligen ausgehenden Datenstromes direkt ab. Stromeigenschaften sind wichtige Elemente, um vor allem kontinuierliche Datenströme verarbeiten zu können. Die Eigenschaften wirken sich direkt auf die Vorgehensweise, sogar auf die generelle Möglichkeit der Verarbeitung von Datenströmen, aus.

So sind hier in dieser Arbeit zwei unterschiedliche Stromeigenschaften zu unterscheiden. Die Endlichkeit, beziehungsweise die Unendlichkeit, sowie die Sortiertheit von Datenströmen. Diese Stromeigenschaften treffen Aussagen über die Beziehungen zwischen den einzelnen Datenstromelementen. Ergänzend zu diesen Stromeigenschaften kommen die Punctations hinzu, welche unter Anderem Aussagen über Attribute von Elementen in Datenströmen machen (siehe Kapitel **5**).

4.1 Endlichkeit und Unendlichkeit von Datenströmen

Datenbanken können nur einen eingeschränkten Datenbestand beinhalten. Dies führt zu der Tatsache, dass vor allem bei Datenbanken endliche Datenströme auftreten. Bei der Behandlung von endlichen Datenströmen müssen bei der Nutzung von beliebigen Operatoren keine besonderen Vorkehrungen getroffen oder Schwierigkeiten beachtet werden. Das Verhalten der Operatoren mit endlichen Datenströmen ist ähnlich wie bei Standard-Datenbankanfragen, wie zum Beispiel in SQL.

Im Gegensatz zu den Datenbanken geht üblicherweise von Sensoren, die eine kontinuierliche Datenerhebung ausführen und die Daten nicht speichern müssen, ein unendlicher Datenstrom aus. Im Gegensatz zu endlichen Datenströmen ergibt sich bei unendlichen Datenströmen als Hauptproblem die Unmöglichkeit der Speicherung aller Daten, beziehungsweise die Ausführung zustandsbehafteter oder blockierender Operatoren auf den kontinuierlichen Datenströmen. Bei zustandsbehafteten Operatoren findet sich ebenfalls ein Problem mit dem begrenzten Speicherplatz. Bei blockierenden Operatoren ergibt sich ein Problem dadurch, dass erst alle betroffenen Daten vorhanden sein müssen, die bearbeitet werden sollen. Dies ist jedoch bei unendlichen Datenströmen logischerweise unmöglich, beziehungsweise nie der Fall. Um diese Operatoren auf unendlichen Datenströmen trotzdem ausführen zu können, müssen diese in ihrer Funktionsweise abgeändert werden.

4.2 Sortiertheit von Datenströmen

Da die Stromeigenschaft der Sortiertheit in Verbindung mit der Endlichkeit keine Probleme darstellt wird auf diese hier nicht weiter eingegangen.

Neben diesen sortierten, endlichen Datenströmen, die vor allem auf Datenbanken basieren, gibt es auch sortierte, unendliche Datenströme, die zum Beispiel auf Sensoren basieren können. Diese Datenströme können zum Beispiel eine Sortiertheit bezüglich ihrer Zeitstempel besitzen. Aufgrund der unendlichen Menge von zu sortierenden Datenstromelementen sind Aussagen oder

Zusicherungen über die Sortierung nicht leicht zu treffen.

5. Zusicherungen für Datenströme (Punctations)

Zusicherungen für Datenströme, so genannte Punctations, sind eine besondere Form von Stromeigenschaften. Punctations sind zusätzlich eingefügte Stromeigenschaften, die verschiedene Teile von Daten der Datenobjekte in unterschiedlichen Ausprägungen betreffen können. So können Punctations Aussagen machen über Dimensionen und Werte von Daten in einem kontinuierlichen Datenstrom. Hier liegt auch der Hauptunterschied zu Fensteroperatoren. Es können durch Punctations Aussagen gemacht werden, die immer über die Puffergrenzen hinaus Gültigkeit besitzen. Dies kann bei verschiedenen Operationen, wie zum Beispiel der Sortierung, zu einer genaueren Aussage bezüglich der eigentlich gewünschten oder angestrebten Semantik der Operation, sowie deren Ergebnisse führen.

Der Zweck der Punctations ist die Erleichterung der Verarbeitung von Datenobjekten in kontinuierlichen Datenströmen. Hierfür wird der kontinuierliche Datenstrom in logische, endliche Subströme unterteilt. Dies geschieht dadurch, dass durch die eingefügten Punctations Aussagen über alle folgenden Elemente getroffen werden. Fensteroperatoren hingegen teilen einen Datenstrom nur in endliche Teile auf, um die Verarbeitung gewährleisten zu können, ohne Informationen bereit zu stellen. Jeder endliche Substrom wird durch anführende Punctations angezeigt. Abgeschlossen wird ein solcher endlicher Substrom durch eine neue Punctation für den nächsten Substrom. Die Zusicherungen aller folgenden Punctations ergänzen sich dabei.

5.1 Aufbau der Punctations

Punctations werden als anführende Datenobjekte jedes zu unterteilenden Substromes gesendet. Anschließend folgen alle davon betroffenen Elemente des Datenstromes. Punctations bestehen dabei aus einer Liste von Bedingungen für die im Datenstrom folgenden Elemente. Diese Liste wiederum besteht aus Elementen mit zwei Teilen.

- Der erste Teil gibt den Namen der Attribute an, auf die sich die Bedingungen beziehen. Beispiel AttrName = „Größe“.
- Der zweite Teil besteht aus der eigentlichen Bedingung für die vorher definierten Attribute. Beispiel: AttrCondition = „<12“
- Da jedem Attribut im System ein Datentyp zugewiesen wird, kann dieser einfach bestimmt werden.

Es können in der Liste mehrere Bezüge auf ein Attribut gemacht werden. Bedingungen ergänzen sich logisch, wobei darauf geachtet werden muss, dass keine unerfüllbaren Forderungen entstehen. Im obigen Beispiel ergibt sich somit für die Elemente, die im Datenstrom nach der Punctation folgen die Zusicherung, dass die Datenobjekte alle in der Größe kleiner als zwölf sind.

5.2 Unterschied zwischen Punctations und Stromeigenschaften

Punctations und Stromeigenschaften treffen beide Aussagen über unendliche Datenströme durch Zusicherungen. Dabei beziehen sie sich auf verschiedene Teile des Datenstromes.
Stromeigenschaften gelten für den gesamten Strom, beziehen sich also auf die einzelnen Elemente und ihre Relationen zueinander, wie zum Beispiel bei der Sortiertheit. Hierbei sind alle Elemente nach einem Attribut sortiert.

Dagegen beziehen sich Punctations nur auf die Ausprägungen der verschiedenen Elemente eines Teils des Datenstroms. Punctations teilen einen kontinuierlichen Datenstrom in verschiedene endliche Subdatenströme auf und treffen dabei Aussagen über die Attributwerte dieses endlichen Teils des gesamten Datenstromes. So zum Beispiel, dass alle Elemente eine Größe haben, die größer als 12 ist. Diese Regel gilt für alle folgenden Elemente.

Die in Kapitel **7** beschriebenen Operatoren führen auf den Attributen der Datenobjekte nur solche Operationen durch, die diese Daten nicht verändern. Anordnungen und Relationen zwischen den verschiedenen Datenobjekten können allerdings, zum Beispiel durch den Selektions-Operator, verändert werden. Dadurch, dass Punctations Aussagen über die Werte der Attribute der Elemente dieses endlichen Teildatenstromes treffen, diese aber nicht durch einen Operator verändert werden, werden auch die Zusicherungen, die von den Punctations gemacht werden, nicht von den Operatoren beeinflusst. Im Gegensatz dazu kann es sein, dass Aussagen über Relationen zwischen Datenstromelementen, die durch Stromeigenschaften beschrieben werden, so durch Operatoren verändert werden, dass diese Stromeigenschaften für ausgehenden Datenströme nicht mehr zutreffend sind.

6. Das UDA - Konzept

Das User-Defined-Aggregate (UDA) Konzept, beschrieben in [5] und [6], beinhaltet als wesentliches Kernkonzept die Möglichkeit, Aggregate für Anfragesprachen wie SQL selbst zu definieren und somit für kontinuierliche Datenströme kompatibel zu machen. Die meisten vordefinierten Aggregate in solchen Sprachen sind nicht für kontinuierliche Datenströme geeignet, da zuerst alle Elemente der Datenmenge zwischengespeichert werden, um anschließend verarbeitet zu werden (vgl. Durchschnittsbildung über „avg“-Aggregat in SQL). UDA-Konstrukte können mit Schleifendurchläufen verglichen werden.

Für Aggregate gelten drei Phasen:

- Die INITIALIZE-Phase
  Diese Phase beschreibt alle Vorgänge, die zu Anfang des Aggregataufrufs durchgeführt werden. Sie ist mit den vorbereitenden Initialisierungen bei Schleifendurchläufen vergleichbar.

- Die ITERATE-Phase
  Diese Phase beschreibt das laufende Vorgehen während des Aggregataufrufs. Sie ist mit dem eigentlichen Schleifendurchlauf vergleichbar.

- Die TERMINATE-Phase
  Diese Phase beschreibt alle Vorgänge, die notwendig sind, um die gesammelten Daten wieder zurückzugeben, meist in Form von Listen.

Bei dem UDA-Konzept geht es grundsätzlich darum, die Ausgabevorgänge der TERMINATE-Phase in die ITERATE-Phase zu verlagern und so eine stetige Ausgabe zu schaffen. Dadurch lässt sich erreichen, dass die Ausgabe nicht erst am Ende der Berechnungen, sondern direkt nach jedem Element durchgeführt wird.
Die TERMINATE-Phase bleibt dabei ungenutzt. Auf diese Weise erhält man Funktionen, die auch kontinuierliche Datenströme behandeln können.

Mittels dieses Konzepts können verschiedene Anfragefunktionen, wie zum Beispiel die Count-Funktion, für kontinuierliche Datenströme realisiert werden. Dies wird dadurch

bewerkstelligt, dass der ursprüngliche Ablauf der Funktion an die Anforderungen von kontinuierlichen und unendlichen Datenströmen angepasst wird. Das generelle Vorgehen soll nun anhand dieser Count-Funktion erläutert werden:

Die ursprüngliche Count-Funktion führt, während alle zu betrachtenden Elemente durchlaufen werden, sämtliche Berechnungen und Überprüfungen bis zum Ende der Elementliste aus. Das Ergebnis wird erst anschließend zurückgegeben. Dies ist offensichtlich für kontinuierliche Datenströme nicht praktikabel, da der interne Speicher aufgrund der Datenfülle irgendwann überlaufen würde. Durch das UDA-Konzept wird die Ausgabe der Auswertungen und Berechnungen nach jeder Überprüfung jedes einzelnen Elements durchgeführt. Dies führt zu einer Vielzahl an Teilergebnissen, garantiert jedoch die Beherrschbarkeit der kontinuierlichen Datenströme.

Mit dem UDA-Konzept können generell beliebige Aggregate / Funktionen erstellt werden, die für kontinuierliche Datenströme geeignet sind. Dabei sollte beachtet werden, dass nach dem UDA-Konzept die einzigen Daten, die dem Aggregat übergeben werden, bzw. die das Aggregat zurück gibt, nur die zu behandelnde Elemente als Eingabe und das (Zwischen-)Ergebnis, als Ausgabe sein sollten. Zudem sollten keine komplizierten Java-Konstrukte eingebaut werden, die weit über dies hinausgehen, nicht zuletzt, um die Wartbarkeit und Übersichtlichkeit des Aggregates zu erhalten bzw. um verschiedene Seiteneffekte zu verhindern.

7. Operatoren-Konzept

Bei einigen Operatoren können Informationen aus mehreren Datenströmen miteinander verarbeitet werden. Sind alle eingehenden Datenströme endlich, so werden sie nicht unterschieden. Falls einer von ihnen unendlich ist, so wird dieser als Hauptdatenstrom bezeichnet, denn er gibt die zu bearbeitenden Objekte vor (über die OID). Nur dieser Hauptdatenstrom darf kontinuierlich sein. Die übrigen Datenströme werden als eine Art Nebendatenströme angesehen, sie müssen endlich sein.

Für Operatoren gelten verschiedene Vor- und Nachbedingungen, um die korrekte Ausführung der Operationen sicherstellen zu können. Damit Operatoren die eingehenden Datenströme auch bearbeiten können, müssen bei der Verbindung mit den Datenquellen die entsprechenden Vorbedingungen verhandelt werden. Falls diese nicht zugesichert werden können, kann keine Verbindung zustande kommen. Die Bedingungen werden durch die Datenquellen auf die Datenströme übertragen. Zu diesem Zweck werden diese Bedingungen formalisiert. Da Operatoren sich auch mit ausgehenden Datenströmen von vorhergehenden Operatoren verbinden können, muss bei Nachbedingungen von Operatoren entsprechend der Verbindung zu Datenquellen verfahren werden.

7.1 Operatoren

Formalisierung für Vor- und Nachbedingungen:

Für die Vor- und Nachbedingungen wurden Formeln für die Formalisierung entworfen:

$\Sigma_{x\uparrow}()$: Die Sortierheitsfunktion gibt an, dass ein Datenstrom nach Attribut x aufsteigend (↑) sortiert ist. Die Angabe der Sortierungsrichtung ist nicht essentiell.

$E()$: Die Endlichkeitsfunktion gibt an, ob ein Datenstrom endlich ist.

σ_x : Zeichen für die Selektion, wobei x ein definierter, logischer Ausdruck ist

ε^* : Eingehender Hauptdatenstrom

ε : Ein beliebiger eingehender Datenstrom

α : Ausgehender Datenstrom

$\gg_x$: Zusatz zu einem Datenstrom, der angibt, dass der Datenstrom in endliche Subströme unterteilt ist. Der Index x steht dabei für die Aussagen, die über die folgenden Elemente getroffen wer den.
Bsp: »Sitze > 12 = Repräsentiert die Punctation mit der Regel, dass die Anzahl der Sitze immer größer als 12 ist.

7.1.1 Assemble - Operator

Der Assemble-Operator wird dazu genutzt, Informationen, die logisch zu einem Datenobjekt gehören, wieder zu diesem Datenobjekt zusammen zu setzen. Dabei wird die Identifizierung, zu welchem Datenobjekt eine Information gehört, über die OID durchgeführt. Ein Datenstrom, der Hauptdatenstrom, gibt die Objekte vor, die mittels der Informationen aus den anderen Datenströmen zusammengesetzt werden sollen. Der Hauptdatenstrom darf als einziger Datenstrom kontinuierlich sein. Die grafische Darstellung der Funktion des Operators zeigt die Abbildung **7.1**. Der Operator wird in Operatorengraphen durch das Symbol ***Assemble*** repräsentiert.

Vorbedingung:

Bezüglich Daten:

- Zusammenzubauende Datenobjekte sind registriert und besitzen eine OID

Bezüglich Stromeigenschaften:

- Nur einer der eingehenden Datenströme, der Hauptdatenstrom, darf unendlich sein
  Für alle ε gilt: $\neg E(\varepsilon) \rightarrow \varepsilon = \varepsilon^*$

Ausführung:

Der Operator ergänzt ein unvollständiges Datenobjekt z.B. aus einem Datenstrom zu einem kompletten Datenobjekt mit allen dazugehörigen Daten.

- Erhält ein Datenobjekt / OID aus einem unendlichen Datenstrom
- Erweitert das Datenobjekt mit Daten aus endlichen Datenströmen
- Gibt das vollständige Datenobjekt aus

Nachbedingung:

- Der ausgehender Datenstrom ist unendlich, wenn ein eingehender Datenstrom unend-

lich ist. Für alle ε gilt: $E(\alpha) := \Lambda\,(E(\varepsilon))$,
weiter gilt: $\neg E(\varepsilon) \rightarrow \varepsilon = \varepsilon^*$

- Der ausgehende Datenstrom ist dann sortiert, wenn der eingehende „Hauptdatenstrom“ sortiert ist.
  $\Sigma_x(\alpha) := \Sigma_x(\varepsilon^*)$

- Wenn andere eingehenden Datenströme sortiert sind, so wirkt sich das auf den ausgehenden Datenstrom nicht aus.

Grafik

Die graphische Abbildung des Operators wird durch Abbildung **7.1** beschrieben.

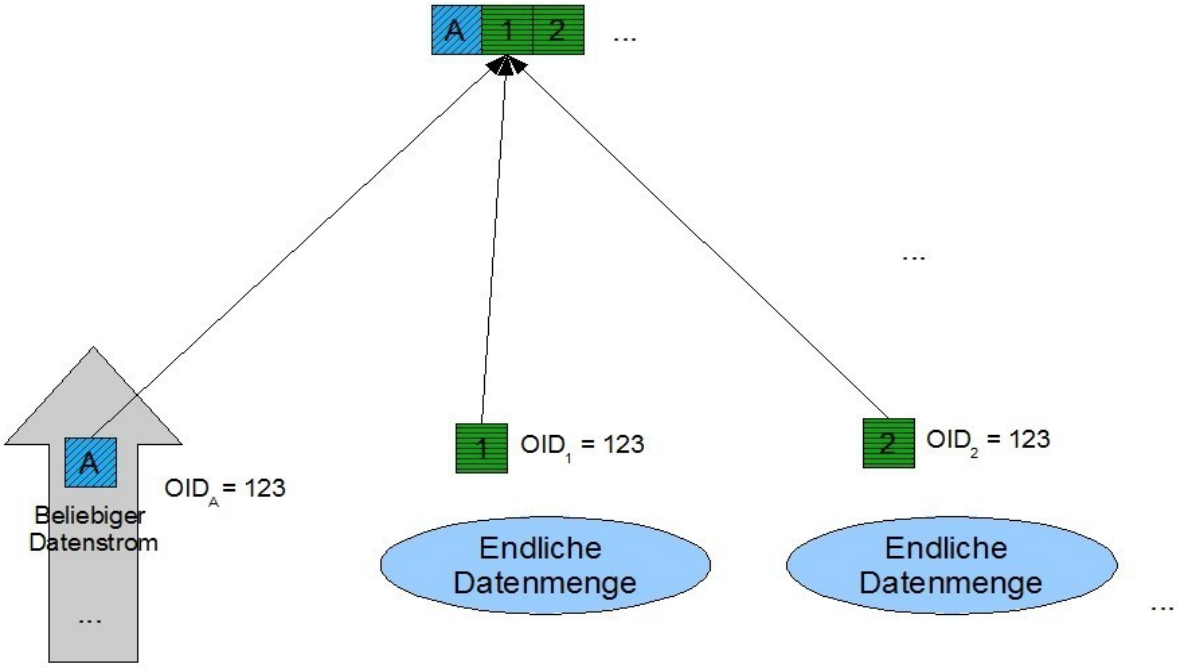

Abbildung 7.1: Assemble-Operator

7.1.2 Selektions - Operator

Beim Selektions-Operator steht der Vergleich von Attributwerten von Objekten im Vordergrund. Hierbei kann man zwei verschiedene Typen des Vergleichens unterscheiden. Zum einen den Vergleich gegen eine Konstante und zum anderen den Vergleich zwischen zwei Attributwerten, zum Beispiel zweier Objekte, die durch Relationenobjekte verbunden sind. Dies wird mittels entsprechenden Vergleichsoperatoren, wie „größer“ oder „kleiner“ durchgeführt. Dem Selektions-Operator wird dazu ein logischer Ausdruck übergeben, der die eigentliche durchzuführende Operation beschreibt, gegen die verglichen werden soll. Je nach Art des Vergleichs entscheidet der Operator über die entsprechende Vorgehensweise.

Für den ersten Typ des Vergleichens gilt, dass der komplette logische Ausdruck erfüllt werden muss, damit die Selektion erfolgreich ist. Diese Art von Vergleichen bezieht sich nur auf einzelne Datenobjekte.

Der zweite Typ bezieht sich vor allem auf zwei Datenobjekte. Dies bringt eine weitere Voraussetzung für diese Art des Zugriffes mit sich. So muss der eingehende Datenstrom, der untersucht werden soll, Relationenobjekte beinhalten. Relationenobjekte besitzen, neben anderen, als Attribute die OIDs der Datenobjekte, auf die sie sich beziehen, beziehungsweise für die sie eine Relation darstellen. Der Zugriff auf diese OID-Attribute, die hier als „Obj1" und „Obj2" bezeichnet werden, geschieht dabei dadurch, dass der Operator durch die definierten Anweisungen, wie zum Beispiel „Obj1.temp" bei dem Vergleich „Obj1.temp = Obj2.temp", über die entsprechend definierten OIDs (hier „Obj1", also die erste OID, die in dem Relationenobjekt festgehalten wird) auf die entsprechenden Objekte und deren Attribute (hier: „temp") zugreift. Für den zweiten Typen besteht auch die Möglichkeit, verschiedene Attributwerte des selben Datenobjektes zu vergleichen. Hierfür ist es nicht notwendig, dass der eingehende Datenstrom Relationenobjekte beinhaltet. Auch hier muss der logische Ausdruck erfüllt sein.

Die logischen Ausdrücke können bei beiden Typen auch zu komplexeren Anfragen kombiniert werden.

Die grafische Darstellung der Funktion des Operators befindet sich in Abbildung **7.2**. Der Operator wird in Operatorengraphen durch das Symbol σ repräsentiert. Hierbei können verschiedene Prädikate im Index angegeben werden, die die Selektion definieren.

Vorbedingung:

- keine

Ausführung:

- Bei dem ersten Vergleichstyp hat der Operator direkten Zugriff, zum Beispiel über einen internen Speicher, auf einen logischen Ausdruck wie zum Beispiel „Sitzanzahl < 20" oder „Farbe = rot", gegen den verglichen werden soll. Ein Datenobjekt des eingehenden Datenstromes wird gelesen.
- Der Vergleich wird so ausgeführt, dass ein bestimmtes Attribut des Objekts gegen den logischen Ausdruck ausgewertet wird. Sollte dieses Attribut nicht mit diesem logischen Ausdruck übereinstimmen, so wird das zu untersuchende Objekt verworfen.

- Die zweite Version vergleicht auf gleiche Weise zum Beispiel zwei Attribute von zwei verschiedenen Objekten miteinander, die über ein Relationenobjekt in Verbindung stehen.
- Falls hier ein Vergleich erfolgreich durchgeführt werden konnte, wird das Relationenobjekt auf den ausgehenden Datenstrom gelegt.

Nachbedingung:

Für die Nachbedingung ergeben sich zwei verschiedene Fälle:

- **1. Fall:**
  Der ausgehende Datenstrom ist bei unendlichem Eingangsdatenstrom endlich, falls der eingehende Datenstrom bezüglich des zu selektierenden Elements sortiert ist, und mittels Zusicherungen und Monotonie der Sortierung auf Folgeelemente geschlossen wer-

den kann.

$\neg E(\varepsilon)$

$\Sigma_{x\uparrow}(\varepsilon)$

$\sigma_{x\leq...}$ (Selektion betrifft Attribut x und hat einen endlichen Wertebereich, das heißt es wird keine Selektion durchgeführt der Form x>...)

$\rightarrow E(\alpha)$

- **Sonst**:
  Die Stromeigenschaften des ausgehenden Datenstroms entsprechen denen des eingehenden Datenstroms.

$$E(\alpha) := E(\varepsilon)$$
$$\Sigma_x(\alpha) := \Sigma_x(\varepsilon)$$

Grafik

Die graphische Abbildung des Operators wird durch Abbildung **7.2** beschrieben.

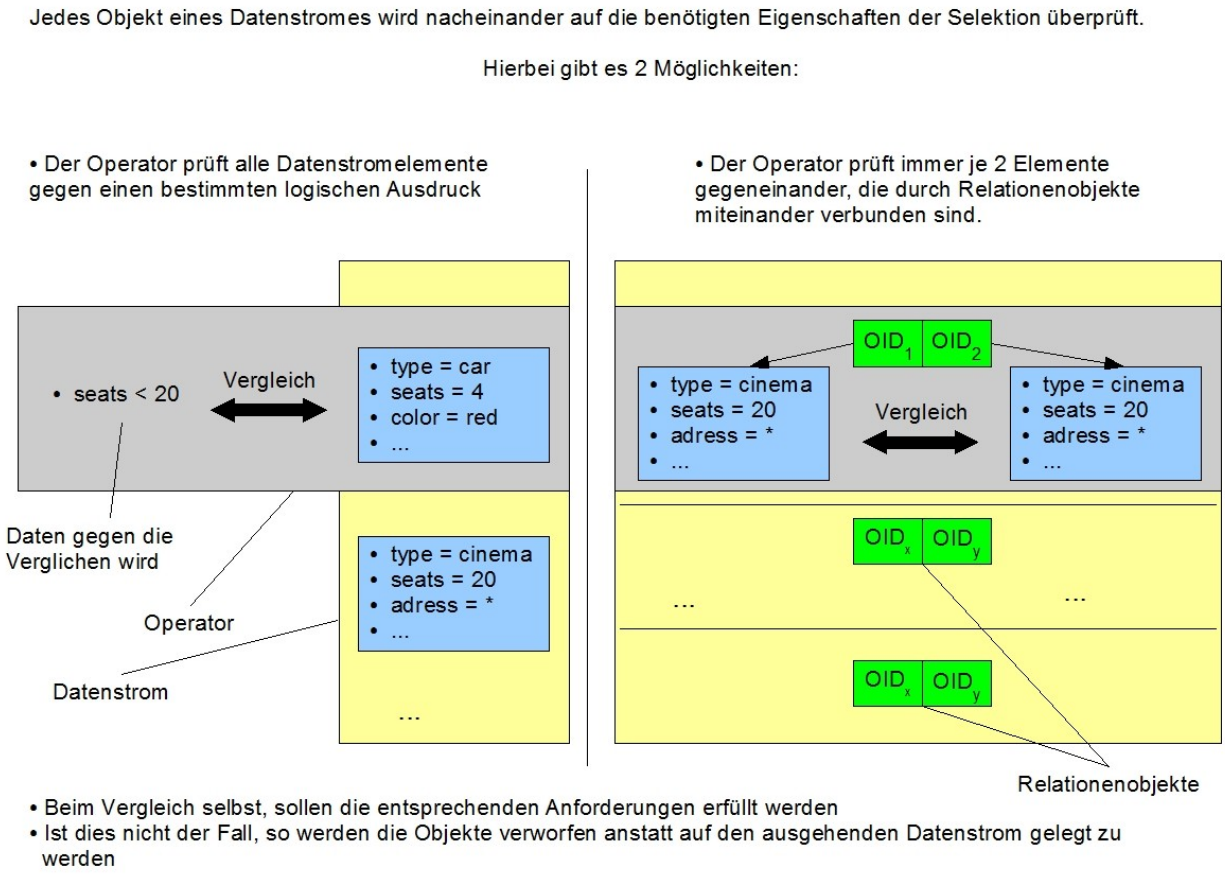

Abbildung 7.2: Selektions-Operator

7.1.3 Kreuzprodukt - Operator

Für den Kreuzprodukt-Operator sind zwei Datenströme zugelassen. Falls mehrere Datenströme verschränkt werden sollen, kann der Operator mehrfach eingesetzt werden, wobei der Ausgangsstrom wieder als eingehender Datenstrom benutzt wird. Auch hier wird der Datenstrom, der zur Ergänzung gedacht ist, als Nebendatenstrom betrachtet. Der andere ist der potentiell unendliche Hauptdatenstrom. Die grafische Darstellung der Funktion des Operators ist in Abbildung **7.3** abgebildet. Der Operator wird in Operatorengraphen durch das Symbol $x_{//}$ repräsentiert. Hierbei können verschiedene Attribute in den Klammern angegeben werden, die in die Erstellung der Relationenobjekte einfließen, was wiederum durch die Klammern ausgedrückt wird.

Vorbedingung:

- Maximal ist ein unendlicher Datenstrom erlaubt, dieser kann nur der Hauptdatenstrom sein
  Für alle ε gilt: $\neg E(\varepsilon) \rightarrow \varepsilon = \varepsilon^*$

Ausführung:

Zu allen Objekten einer endlichen oder unendlichen Datenmenge A (Hauptdatenstrom) und den Elementen einer endlichen Datenmenge B (z.B. aus einer Datenbank) wird das Kreuzprodukt mittels so genannter Relationenobjekte gebildet.

- Da die Anzahl der Datenobjekte der Datenmenge B endlich ist, werden diese am Anfang des ausgehenden Datenstromes versendet und beim Empfänger zwischengespeichert.
- Zu jedem Objekt des Hauptdatenstromes A wird anschließend ein endlicher Substrom, der aus dem einzelnen Datenobjekt des Datenstromes A, sowie Relationenobjekten und einer Punctation besteht, gebildet und versendet.
- Die Relationenobjekte werden gebildet, indem die OID des beteiligten Objektes des kontinuierlichen Datenstroms A, sowie die OID jedes Datenobjekts der Datenmenge B als Relationen beibehalten werden. Hinzu kommen verschiedene Attribute, die das Relationenobjekt charakterisieren und durch Berechnungen erzeugt werden; zum Beispiel die Distanz zweier Objekte, die durch eine distanzberechnende Funktion erzeugt wurde.
- Das erste Element des endlichen Substreams ist die Punctation, die den Anfang des neuen Substreams anzeigt. Sie wird durch ein spezielles Objekt repräsentiert. (siehe dazu Kapitel **5**). Dies vermittelt die Zusicherung, dass sich keines der folgenden Elemente mehr auf Objekte des Vorgänger-Substromes bezieht. Das zweite Element des Substreams is das Objekt aus dem unendlichen Datenstrom A, gefolgt von allen Relationenobjekten.

Nachbedingung:

- Durch die getrennte Behandlung der Objekte des kontinuierlichen Datenstromes A in den endlichen Substreams, ist die Punctation eine Zusicherung, dass keine weiteren Bezüge auf das jeweilige aktuell verarbeitete Datenobjekt des Datenstromes A folgen.
- Ausgehender Datenstrom ist unendlich, falls der eingehenden Hauptdatenstrom un-

endlich ist. Analoges gilt für die Endlichkeit, da nur der Hauptdatenstrom unendlich sein darf.

$E(\alpha) := E(\varepsilon^*)$

- Die Sortiertheit beider eingehenden Datenströme bleibt erhalten

  Für alle x und ε gilt: $\Sigma_x(\alpha) := \Sigma_x(\varepsilon)$

Grafik

Die graphische Abbildung des Operators wird durch Abbildung **7.3** beschrieben.

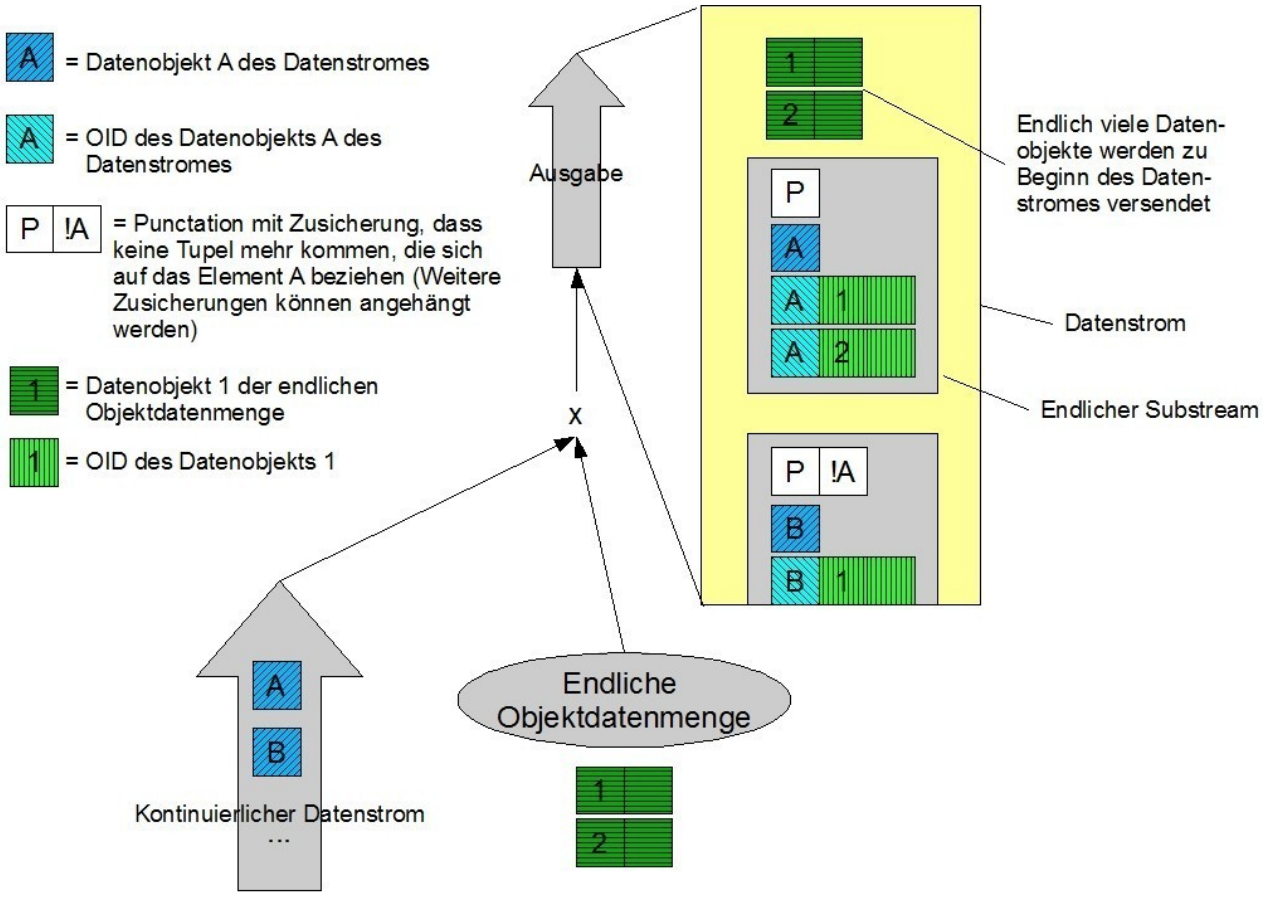

Abbildung 7.3: Kreuzprodukt-Operator

7.1.4 „Fetch first X“ - Operator

Bei diesem Operator sind leicht verschiedene Ausprägungen möglich. So können über einen Parameter die Anzahl der auszugebenden Elemente bestimmt werden. Die grafische Darstellung der Funktion des Operators zeigt die Abbildung **7.4**. Der Operator wird in Operatorengraphen durch das Symbol ***Fetch*** repräsentiert. Im Index des Operatoren wird angegeben, wie viele Elemente zurückgegeben werden sollen.

Vorbedingung:

- Keine

Ausführung:

- Der Operator erhält eine Menge von Objekten durch den eingehenden Datenstrom, sowie einen Parameter, der die Anzahl von auszugebenden Objekten definiert.
- Der Operator gibt die Anzahl von Elementen auf den ausgehenden Datenstrom, die restlichen können verworfen werden
- Die Anzahl der zurückgegebenen Elemente hängt nur vom übergebenen Parameter ab.

Nachbedingung:

- Sortierung des Eingangsdatenstromes bleibt erhalten

  Für alle x und ε gilt: $\Sigma_x(\alpha) := \Sigma_x(\varepsilon)$
- Der ausgehende Datenstrom ist endlich

  $E(\alpha)$

Grafik

Die graphische Abbildung des Operators wird durch Abbildung **7.4** beschrieben.

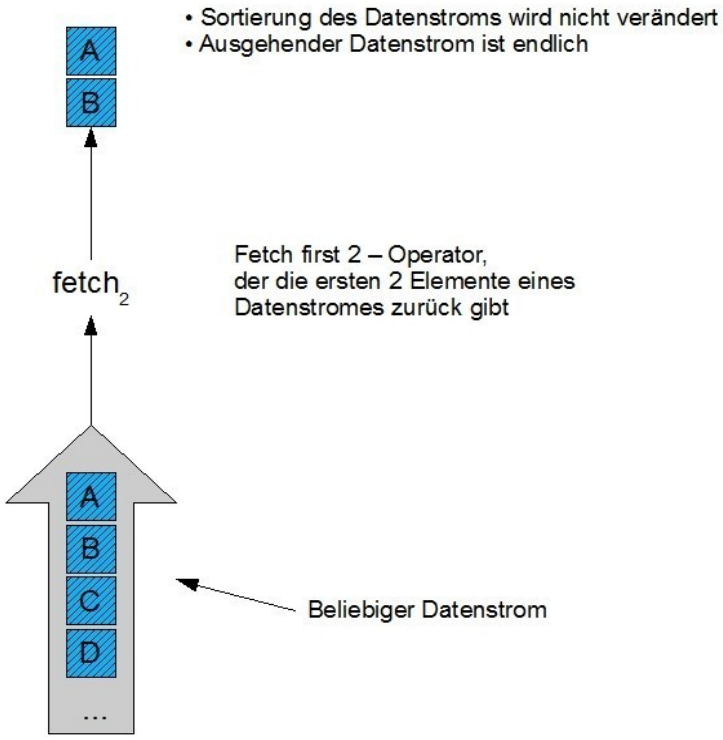

Abbildung 7.4: Fetch-Operator

7.1.5 Sortierungs - Operator

Die Schwierigkeit beim Sortierungs-Operator liegt darin, kontinuierliche Datenströme überhaupt zu sortieren, denn für eine Sortierung müssen normalerweise Informationen über alle Elemente vorhanden sein. Dies ist bei einer unendlichen Datenmenge nicht möglich. Bei der Sortierung von kontinuierlichen Datenströmen wird dabei versucht, möglichst genaue Aussagen über die Sortierung eines Teiles der Datenmenge zu treffen. Um diese Sortierung möglichst genau durchführen zu können, werden Punctations mit einbezogen, deren Informationen über eine Vorsortierung mit einfließen. Es wird also nur immer ein Teil der Datenmenge sortiert. Die grafische Darstellung der Funktion des Operators ist in Abbildung **7.5** abgebildet. Der Operator wird in Operatorengraphen durch das Symbol ***Sort*** repräsentiert. Dabei wird im Index das Attribut angegeben, nach dem sortiert werden soll.

Vorbedingung:

Die Datenmenge, die sortiert werden soll, muss endlich sein. Dies wird erreicht durch:

- die Endlichkeit des eingehenden Datenstroms

oder

- Aufteilen in endliche Subströme mittels Punctations (siehe Kreuzprodukt). Die Punctations werden von den datenstromerzeugenden Datenquellen eingefügt.

  $E(\varepsilon) \quad v \quad (\neg E(\varepsilon) \ \& (\varepsilon = »_x))$

  wobei x für diverse logische Ausdrücke, wie zum Beispiel „Größe > 12“, steht.

Ausführung:

- Die Sortierung für einen endlichen Datenstrom wird wie in SQL durchgeführt und stellt keine großen Probleme dar.

- Die Sortierung für einen unendlichen Datenstrom wird mittels Punctations durchgeführt. In dem eingehenden Datenstrom beschränken dabei die Punctations die Anzahl der Elemente auf einen endlichen Substrom. Somit ist eine Sortierung der verschiedenen Submengen der Daten möglich.
- Die Punctations können dabei ebenso in die Sortierung logisch mit einbezogen, wobei Informationen über eine eventuelle Vorsortierung in der Punctation sich dazu direkt auf das zu sortierende Attribut beziehen müssen. Die Punctations nehmen praktisch eine grobe Vorsortierung des Datenstroms vor.
- Als Beispiel kann so für einen Substrom gelten, dass die Größe der enthaltenen Elemente immer kleiner ist als ein festgelegter Wert ist. Nach der Sortierung des Substromes werden die Punctations zusammen mit den sortierten Elementen in den Ausgangsstrom gelegt.

Nachbedingung:

- Der ausgehende Datenstrom ist unendlich, falls der eingehende Datenstrom unendlich ist.

  $E(\alpha) := E(\varepsilon)$

- Die Sortierung eines eingehenden Datenstrom bleibt nicht erhalten. Es entsteht eine neue Sortierung.

  $\Sigma_x(\alpha)$

Grafik

Die graphische Abbildung des Operators wird durch Abbildung **7.5** beschrieben.

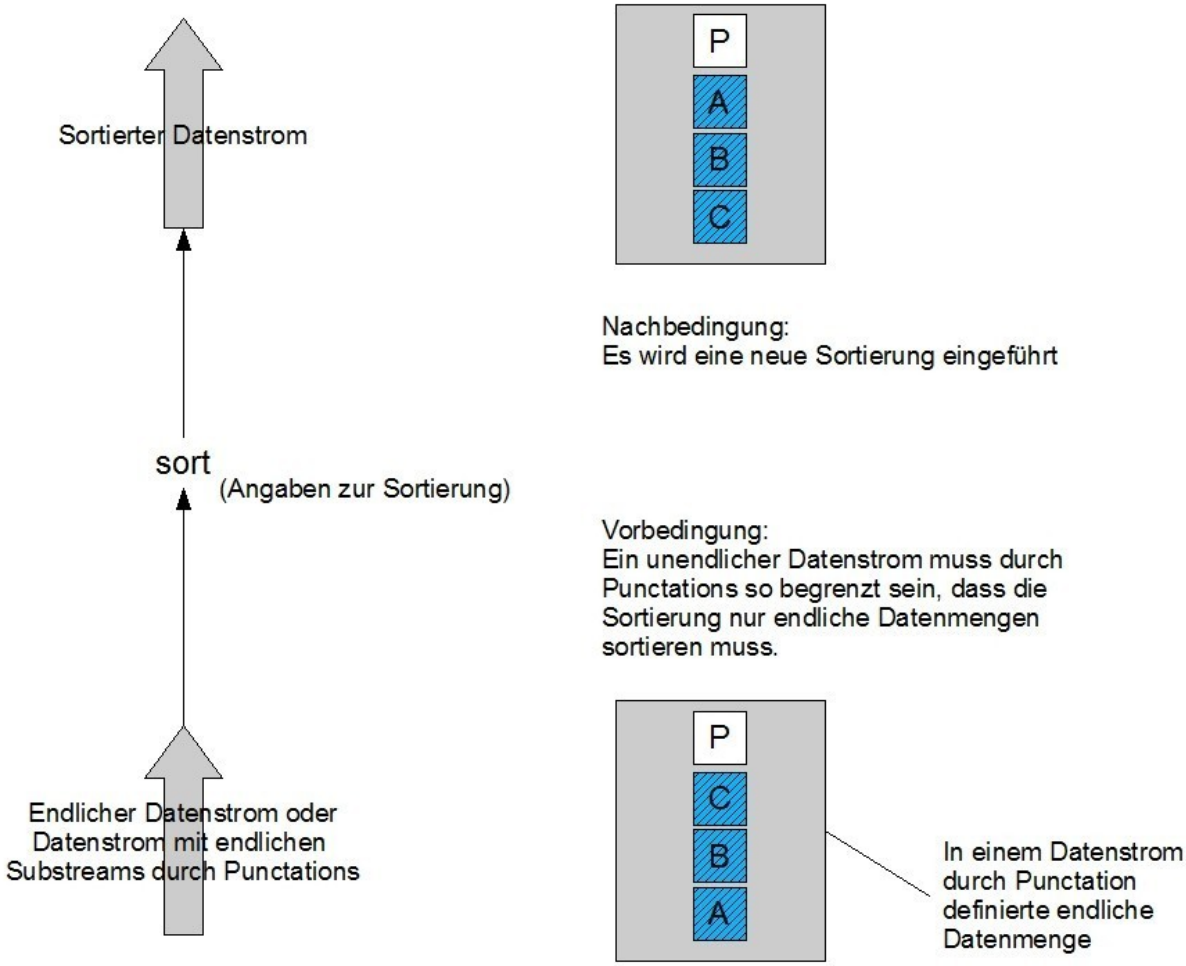

Abbildung 7.5: Sortierungs-Operator

7.1.6 Gruppierungs - Operator

Durch diesen Operator werden Objekte miteinander zu Relationenobjekte gruppiert. Generell können die Relationenobjekte, die durch diesen Operator erzeugt werden, eine beliebige Anzahl von Elementen gruppieren. Als Beispiel wird hier die Ausprägung mit zwei Elementen erläutert. Die grafische Darstellung der Funktion des Operators ist in Abbildung **7.6** abgebildet. Der Operator wird in Operatorengraphen durch das Symbol ***group*** repräsentiert. Im Index wird angegeben, wie viele Elemente in einem Relationenobjekt gruppiert werden sollen.

Vorbedingung

- keine

Ausführung

- Der Operator erhält einen beliebigen Datenstrom und puffert immer zwei Elemente

zwischen.

- Das erste der gepufferten Elemente wird anschließend auf den Ausgangsstrom gegeben.
- Der Operator erzeugt als nächstes ein Relationenobjekt, das zwei OIDs als Referenzen beinhaltet: Die erste OID des ersten gepufferten Elements, das bereits auf den Ausgangsstrom gelegt wurde, und die zweite OID des zweiten gepufferte Elements.
- Das Relationenobjekt wird anschließend auf den Ausgangsstrom gelegt.
- Der Operator wechselt das zweite gepufferte Element auf den ersten Platz und puffert auf den nun freien zweiten Platz ein neues, drittes Element. Das nicht mehr benötigte erste Element wird verworfen
- Nun wird das neue erste Element auf den Ausgangsstrom gelegt und analog weiter verfahren.

Nachbedingung

- Stromeigenschaften, die der eingehende Datenstrom besitzt, werden beibehalten. $\Sigma_x(\alpha) := \Sigma_x(\varepsilon)$ und $E(\alpha) := E(\varepsilon)$

Grafik

Die graphische Abbildung des Operators wird durch Abbildung **7.6** beschrieben.

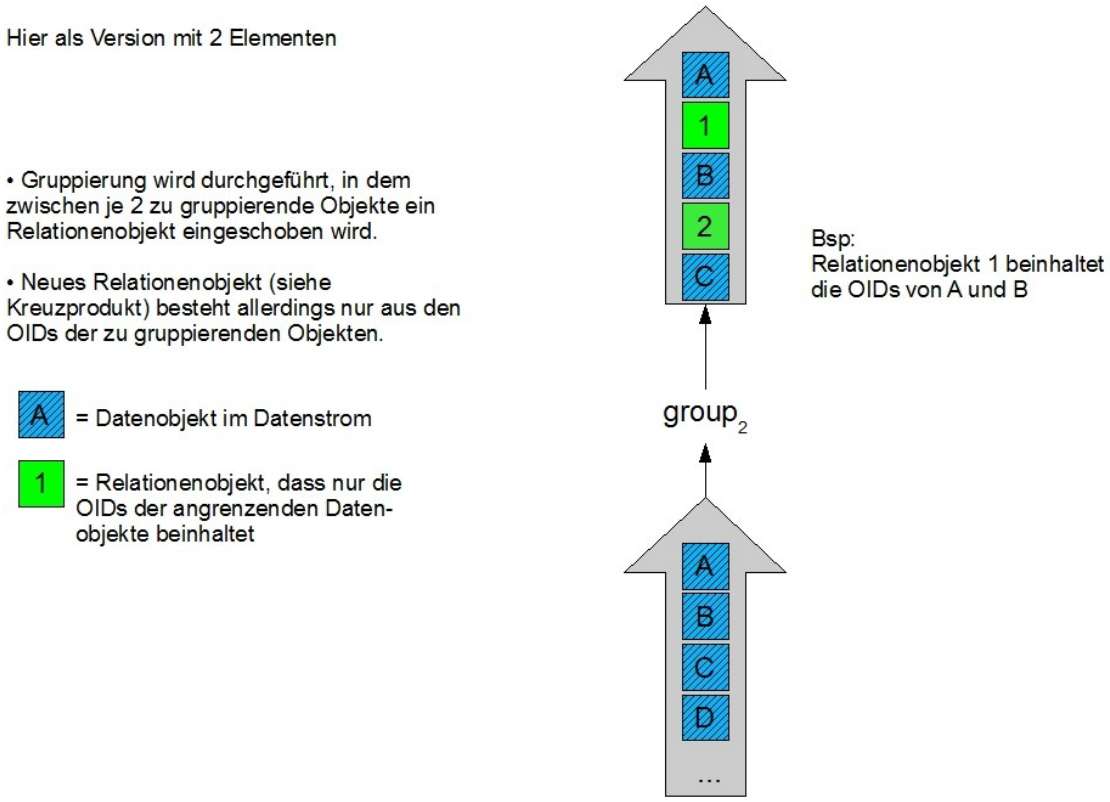

Abbildung 7.6: Gruppierungs-Operator

7.1.7 User Defined Aggregate (UDA) - Operator

Trotz der Tatsache, dass es sich bei UDAs nicht um atomare Operatoren, wie sie hier benutzt und definiert werden, sondern eher um Funktionen handelt, wird ein Operator für UDAs definiert. Außerdem macht die Implementierung die Nutzung eines solchen Operators notwendig. Der eigentliche UDA-Operator bildet nur ein Ausführungsgerüst für verschiedene Arbeiten. Diese müssen dem UDA Operator übergeben werden, wodurch seine jeweiligen Verhaltensweisen festgelegt werden. Es müssen sowohl die Verhaltensweisen für die INITIALIZE – Phase, als auch für die ITERATE – Phase übergeben werden. Als Symbol genutzt wird

$UDA_{Name}(init, iter)$: Repräsentiert ein User-Defined-Aggregate (UDA) – Konstrukt.
Diesem werden als Parameter Informationen zu den INITIALIZE (init) und ITERATE - Phasen (iter) zur Verfügung gestellt. Der Index (Name) gibt den Name des Konstruktes an.

Vorbedingung:

- Da die Vorbedingungen von den Verhaltensweisen des Operators abhängen, und diese nicht vorgegeben werden, lassen sich hier keine direkten Vorbedingungen angeben.

- Da allerdings das UDA-Konstrukt und somit auch der UDA-Operator dafür gedacht ist, blockierende und zustandsbehaftete Funktionen auf unendlichen Datenströmen auszuführen, lässt sich über die eingehenden Datenströme sagen, dass diese kontinuierlich sind, da sonst dieser UDA-Operator unnötig wäre.

Ausführung:

- Die Ausführung ist abhängig von den Verhaltensweisen, aber im Allgemeinen gilt:

- Die INITIALIZE – Anweisungen werden ausgeführt. Im Üblichen sind dies vorbereitende Maßnahmen, wie die Initialisierung eines Arrays oder Ähnlichem, in dem der innere Zustand gespeichert werden kann.
- Die ITERATE – Anweisungen werden ausgeführt. Hierbei gilt es zu beachten, dass nach jedem Durchlauf dieser Anweisungen eine Ausgabe getätigt wird, da diese auch die TERMINATE-Phase ersetzt, in der die Ausgabe für „normale“ Aggregate stattfindet.

Nachbedingung:

- Auch hier gilt, dass die Nachbedingung von der definierten Verhaltensweise des UDA-Operators abhängt.

7.1.8 Return - Operator

Bei dem Return -Operator handelt es sich um einen Hilfsoperator, der zu einer gelieferten OID das korrespondierende Datenobjekt sucht und es zurück gibt. Hier gibt es verschiedene Möglichkeiten dies zu nutzen. Die OID kann eine, in den Attributen gespeicherte OID eines Relationenobjektes sein. Dieser Operator kann über die Parameter bei der Erstellung des Operators konfiguriert werden. Als Parameter wird dabei definiert, in welchem Attribut eines Datenobjektes die OID des Objektes,

das gefunden werden soll, gespeichert ist. Der Operator erhält zwei eingehende Datenströme. Der Hauptdatenstrom liefert dabei die Datenobjekte, die die zu suchenden OIDs enthalten. Der zweite, endliche Datenstrom enthält die Objekte, in denen nach diesen OIDs gesucht werden soll. Dargestellt wird der Operator mittels:

Return $_x$ (Object): Die Return-Funktion schlägt ein Datenobjekt zu einer gegebenen OID nach und gibt es zurück. Das Attribut, das die OID speichert wird hier als x angegeben.

Vorbedingung:

- keine

Ausführung:

- Der Operator erhält ein Objekt aus dem Hauptdatenstrom und entnimmt die zu suchende OID.
- Der Operator durchsucht den zweiten Datenstrom nach Objekten mit diesen OIDs.

Nachbedingung:

- Stromeigenschaften, die der eingehende Datenstrom besitzt, werden beibehalten.
  $\Sigma_x(\alpha) := \Sigma_x(\varepsilon)$ und $E(\alpha) := E(\varepsilon)$

7.2 Mögliche Beispielanfragen:

Im Folgenden werden einige mögliche Beispielanfragen aufgezeigt, um die Kombinations- und Anwendungsmöglichkeiten der Operatoren zu demonstrieren.

7.2.1 „Nächster Nachbar“ - Anfrage

Die „Nächster Nachbar“-Anfrage wird genutzt, um zu einem gegebenen Datenobjekt, also einer Konstante, wie zum Beispiel der eigenen Person und ihrem Standpunkt, bestimmte, räumlich nächstgelegene Objekte zu ermitteln. Dazu werden entsprechende Datenobjekte mit dem Selektions-Operator gesucht und mit dem Kreuzprodukt-Operator untereinander kombiniert. Die Distanz zwischen dem gegebenen und den ermittelten Objekten, wird dabei durch eine entsprechende Funktion berechnet. Mittels dem Sortierungs- und „Fetch first X“ - Operator werden anschließend die nächsten Objekte bestimmt. Da alle Anfragen an Datenbanken gestellt werden, ergibt sich von vorne herein, dass alle Datenströme endlich sind und somit keine weiteren Einschränkungen oder Vorkehrungen getroffen werden müssen, um die für die Operatoren jeweils geforderten Strombedingungen zu erfüllen.

Eine entsprechende Verkettung der Operatoren hierfür wäre:

$$\alpha = \text{Return}_{\text{Obj1.OID}}\ (\text{Fetch}_1\ (\text{Sort}_{\text{Dist}\uparrow}\ (x)))$$

$$x = \sigma_{\text{Type=Rel}}\ (x_{[\text{Dist}(\ \text{Obj1.pos}\ ,\ \text{Obj2.pos})]}\ (\sigma_{\text{pos...,\ type=...}}\ (\varepsilon), (c)))$$

Dabei stellt **c** eine Konstante dar, die interpretiert werden kann als ein Datenstrom aus nur einem Element. Dieses Element ist hier zum Beispiel das Datenobjekt, das den Benutzer darstellt.
Erläuterung:

- σ ist der Selektions-Operator, wobei gegen eine bestimmte Position und einen bestimmten Typ verglichen wird.
- ε ist ein eingehender Datenstrom
- c ist eine endliche Datenmenge (zum Beispiel das Datenobjekt, das den Benutzer darstellt)
- x ist der Kreuzproduktoperator, der bei der Ausführung hier die Distanz (Dist(...)) der beiden zu verbindenden Elemente berechnet. Hierfür werden die für die Distanzberechnung notwendigen Attribute „pos“ der Datenobjekte „Ob1“ und „Obj2“ der Funktion übergeben. Hierbei beziehen sich „Obj1“ beziehungsweise „Obj2“ auf die jeweiligen Objekte des Kreuzprodukt-Operators. Weiter wird ein neues Attribut in ein Relationenobjekt eingefügt. Die Erzeugung dieses Relationenobjekt wird mittels [ ] dargestellt.
- Der zweite Selektions-Operator wählt nun nur Relationenobjekte mittels „Type=Rel.“ aus
- $Sort_{Dist\uparrow}$ sortiert diese Objekte nach der Distanz in aufsteigender Reihenfolge
- $Fetch_1$ gibt das erste Element, also das Relationenobjekt mit der kleinsten Distanz zurück.
- $Return_{Obj1.OID}()$ wurde hier so konfiguriert, dass es ein Datenobjekt zurück liefert, welches durch die erste OID des durch $Fetch_1$ erhaltenen Relationenobjekts definiert ist.
- α ist der sich ergebende Ausgangsstrom der gesamten Anfrage

Die Strombedingungen für die Operatoren sind:

- Für die Selektion σ ergibt sich hier: $E(\varepsilon) \rightarrow E(\alpha)$
- Für das Kreuzprodukt x ergibt sich: $E(\varepsilon_1)$ und $E(\varepsilon_2) \rightarrow E(\alpha)$
- Für die zweite Selektion ergibt sich: $E(\varepsilon) \rightarrow E(\alpha)$, wobei nur Objekte mit einem Distanzattribut vorhanden sind, also Relationenobjekte
- Für die Sortierung Sort ergibt sich: $E(\varepsilon) \rightarrow E(\alpha)$ und $\Sigma_{Dist}(\alpha)$
- Für die Operation Fetch ergibt sich: $E(\varepsilon) \rightarrow E(\alpha)$, da nur ein Element vorhanden ist.
- Ebenso für Return

7.2.2 Kino-Haltestellen – Problem

Das Kino-Haltestellen-Problem bezieht sich auf eine Anfrage, bei der Paare aus Kinos und Haltestellen gefunden werden sollen, die einen bestimmten Abstand voneinander nicht überschreiten. Die Menge der Haltestellen und Kinos einer Stadt sind dabei begrenzt, also endlich.

$\alpha_{Kino} = Return_{Obj1.OID}(x)$
$\alpha_{Haltestellen} = Return_{Obj2.OID}(x)$
$x = \sigma_{type=Rel,\ dist <= 500}(x_{[Dist(Obj1.pos, Obj2.pos)]}(\sigma_{type=“Kino“,\ pos=...}(\varepsilon_1),\ \sigma_{type=“Haltestelle“,\ pos=...}(\varepsilon_2)))$

- σ sind Selektions-Operatoren, die Kinos, bzw. Haltestellen aus Datenströmen herausfiltern, falls die Datenströme nicht schon nur aus diesen Objekten bestehen. Ihre mögliche Lage kann dabei durch die Angabe eines Wertes für „pos=...“ eingeschränkt werden.
- ε_1 und ε_2 sind eingehende, endliche Datenströme
- x ist der Kreuzproduktoperator, der bei der Ausführung hier die Distanz (Dist(...)) der beiden zu verbindenden Elemente berechnet. (siehe dazu auch Kapitel **7.2.1**)
- Der zweite Selektions-Operator wählt nun nur Relationenobjekte mittels „type=Rel.“, sowie von diesen wiederum nur diese, deren Distanzattribut kleiner oder gleich z.B. hier 500 Meter ist.
- $\text{Return}_{\text{Obj1.OID}}()$ wurde hier so konfiguriert, dass es ein Datenobjekt zurück liefert, welches durch die erste OID des erhaltenen Relationenobjekts definiert ist. Entsprechendes gilt für den anderen Return-Operator.
- α_{Kino} ist der sich ergebende Ausgangsstrom der Anfrage. Er enthält alle Kinos des Ergebnisses. $\alpha_{\text{Haltestellen}}$ ist der sich ergebende Ausgangsstrom der Anfrage. Er enthält alle Haltestellen des Ergebnisses. Diese ausgehenden beiden Datenströme enthalten alle Kinos und Haltestellen in geordneter und paarweise, d.h. dass das z.B. 7. Element des einen Datenstroms zum 7. Element des anderen Datenstroms als Ergebnis gehört,

Die Strombedingungen für die Operatoren sind:

- Für die erste Selektionsstufe ergibt sich hier: $E(\varepsilon_1) \rightarrow E(\alpha_1), E(\varepsilon_2) \rightarrow E(\alpha_2)$
- Für das Kreuzprodukt x ergibt sich: $E(\varepsilon_1)$ und $E(\varepsilon_2) \rightarrow E(\alpha)$
- Für die zweite Selektionsstufe ergibt sich: $E(\varepsilon) \rightarrow E(\alpha)$, wobei nur Objekte mit einem Distanzattribut vorhanden sind, also Relationenobjekte
- Für Return ergibt sich: $E(\varepsilon) \rightarrow E(\alpha_{\text{Haltestellen}})$, gleiches gilt für die Kinos

7.2.3 Event-erzeugende Anfrage

Eine event-erzeugende Anfrage wird dazu genutzt, um Datenströme zu beobachten und Veränderungen in diesen über Datenobjekte mitzuteilen. Event-erzeugende Anfragen können aus beliebigen Operatoren zusammengestellt werden, je nach gewünschter Funktion, die erfüllt werden soll. Als ein Beispiel wird hier eine solche Anfrage gestellt, bei der ein Datenstrom aus Temperaturwerten dahingehend beobachtet wird, dass ein Event ausgelöst werden soll, wenn aufeinander folgende Temperaturwerte nicht gleich sind, also sich die Temperatur verändert.

Dazu werden über den Gruppierungs-Operator Datenobjekte so gruppiert, dass jedes Datenobjekt mit seinem direkten Nachfolger kombiniert wird. Für einen Vergleich dieser kombinierten Objekte wird die Anforderung gestellt, dass die Temperaturen der aufeinander folgenden Datenobjekte nicht gleich sein sollen. Wird diese Anforderung bei einem Vergleich erfüllt, wird mittels einer Konstruktorfunktion des NEXUS-Systems ein neues Eventobjekt erzeugt, und auf den Ausgangsstrom gegeben. Der Event wird nur durch das Auftreten des Objektes im ausgehenden Datenstrom repräsentiert. Der hier genutzte Selektions-Operator ist eine Variante des Basis-Selektions-Operators, wie er in **7.1.2** beschrieben wurde. Dieser Operator hier erzeugt ein Eventobjekt und gibt es über den ausgehenden Datenstrom weiter, anstatt übereinstimmende

Objekte weiter zu geben. Eine entsprechende Verkettung der Operatoren hierfür wäre:

$$\alpha = \sigma_{[Obj1.temp \neq Obj2.temp]} (group_2 (\varepsilon))$$

Erläuterung:

- ε ist der eingehende Datenstrom, für den die Event-erzeugende Anfrage verwendet werden soll
- $group_2$ gruppiert je 2 Objekte des Datenstromes überlappend miteinander
- $\sigma_{[Obj1.temp \neq Obj2.temp]}$ ist die Selektion, die hier als Beispiel das Temperaturattribut (temp) von zwei gruppierten Elementen miteinander vergleicht. Dabei wird über Obj1 auf das erste Objekt des Relationenobjektes zugegriffen. Entsprechendes gilt für Obj2. Falls wie in diesem Beispiel die Temperaturen der beiden Datenobjekte übereinstimmen, wird ein neues Eventobjekt erzeugt. Die Erzeugung eines neuen Eventobjektes mittels einer Konstruktorfunktion wird durch [ ] dargestellt.
- α ist der sich ergebende Ausgangsstrom der Anfrage

Die Strombedingungen für die Operatoren sind:

- Für die Gruppierung ergibt sich hier: $E(\varepsilon) \rightarrow E(\alpha)$
- Für die Selektion ergibt sich: $E(\varepsilon) \rightarrow E(\alpha)$

7.2.4 Count - Anfrage

Die Count-Anfrage wird zur Zählung verschiedener Objekte benutzt. So können Datenströme durch den Selektions-Operator so bearbeitet werden, dass nur die zu beachtenden Datenobjekte im ausgehenden Datenstrom vorhanden sind. Diese Elemente werden über eine entsprechende Count-Funktion, die über das UDA-Konzept realisiert wird (siehe Kap. **6**), gezählt. Hierbei werden auch die drei Phasen des Aggregates genutzt.

- In der INITIALIZE-Phase wird hierfür ein Counter erzeugt, der die aktuelle Zahl der zu zählenden Elemente festhält.

- In der ITERATE-Phase wird nach jedem betrachteten Element der aktuelle Zustand des Counter ausgegeben. So erhält man eine kontinuierliche Ausgabe, statt einer am Ende der Ausführung. Somit ist die Count-Anfrage für kontinuierliche Datenströme nutzbar.

- Die TERMINATE-Phase bleibt, wie im UDA-Konzept beschrieben, leer, das heißt, bei der Terminierung der Anfrage sind keine zusätzlichen Schritte und Verarbeitungen notwendig.

In dem folgenden Beispiel soll die Anzahl der in dem Datenstrom enthaltenen Objekte gezählt werden, deren Sitzanzahl größer als 10 ist.

Es ergibt sich somit für die Count-Anfrage folgende Operator-Schreibweise:

$$\alpha = UDA_{Count}(\sigma_{Sitze > 10}(\varepsilon))$$

- ε stellt einen eingehenden Datenstrom dar.
- $\sigma_{Sitze > 10}$ ist der Selektions-Operator, wobei nur Objekte selektiert werden, deren Sitzanzahl größer als 10 ist.
- Das Count-UDA erhält alle Elemente, die der Selektions - Operator ausgibt.
  Als Konfiguration erhielt Count-UDA Angaben, wie das gewünschte Vorgehen ist, das heißt hier, dass alle erhaltenen Elemente fortlaufend gezählt werden sollen, und nach jedem Element eine Ausgabe auf den ausgehenden Datenstrom α gegeben werden soll, die das aktuelle Zwischenergebnis repräsentiert.

Als Strombedingung ergibt sich für die Selektion: $E(\varepsilon) \rightarrow E(\alpha)$

Für den Ergebnisstrom des UDAs ergibt sich, dass immer dann die ITERATE-Phase durchlaufen und ein Zwischenergebnis ausgegeben wird, wenn die Bedingungen der Selektion auf ein Element zutrifft. Dies führt dazu, dass die gleiche Strombedingung, die für die Selektion gilt, für das ganze UDA gilt.

8. Realisierung

Die Realisierung des Operatorenkonzeptes, das in dieser Arbeit beschrieben wird, wurde, basierend auf [10], durchgeführt. Im Folgenden wird das Wissen aus dieser Masterthesis vorausgesetzt. Für diese Realisierung mussten verschiedene Bestandteile umgeändert und/oder erweitert werden.

Die in [10] erwähnte Sprache, die für die Erstellung, Modifikation und Beendigung aller Bestandteile einer so genannten `ProcessLine` zuständig ist, besitzt derzeit keinen eigenen Namen im System, da die Ausarbeitung dieser Sprache noch nicht vollendet wurde. Sie wird hier als „Command-Sprache“ bezeichnet, eine Befehlseinheit dieser Sprache wird dementsprechend „Command“ genannt, optional mit dem jeweiligen Zusatz ihres Einsatzgebietes. So zum Beispiel für eine instantiierende Befehlseinheiten die Bezeichnung „SetupCommand“.

Da durch das vorhandene System bestimmte Begrifflichkeiten bereits vorhanden sind, müssen die systembezogenen Begriffe auf die, die in dieser Arbeit verwendet wurden, angeglichen, bzw. eine Verbindung zwischen den Begriffen geschaffen werden. So wird im System zwischen Operationen und Operatoren unterschieden. Die Operatoren dieser Arbeit werden im vorhandenen System durch die Operationen repräsentiert und umgesetzt. In ihnen werden, genau wie hier beschrieben, die eigentlichen Verarbeitungen auf Datenobjekten durchgeführt. Um allerdings eine möglichst gute Wiederverwendung von Strukturen zu ermöglichen, werden die atomaren Vergleiche, wie zum Beispiel ein einfacher „<“-Vergleich in eine eigene Klasse ausgelagert. Diese Klassen wiederum werden im System als Operatoren bezeichnet. Jedem Operator werden des weiteren verschiedene Vergleichsarten zugewiesen. So können zum Beispiel durch einen so genannten `ComparisonOperator` bzw. durch eine Subklasse desselben, alle einfachen Vergleiche durchgeführt werden (wie zum Beispiel der „<“-Vergleich). Ein `BooleanOperator` hingegen verknüpft zwei Wahrheitswerte zu einem.

Da zum Zeitpunkt der Bearbeitung die Umstellung auf ein datenstrombasiertes System noch nicht abgeschlossen ist und das System in verschiedenen Bereichen noch diverse „Lücken“

aufweist, werden verschiedene Änderungen und Anpassungen mit Ausblick auf eine temporäre Nutzung vorgenommen; so zum Beispiel die Nutzung von kommaseparierten Werten als Parameter. Vereinzelt kann allerdings auch die Funktionalität der Operationen durch das Fehlen verschiedener, kleinerer Systemkomponenten, die nicht Teil dieser Arbeit sind, aber nachträglich Implementiert werden können (siehe Kapitel **9**), beeinträchtigt sein.

Alle Klassen die im Rahmen dieser Arbeit erstellt wurden oder größeren Überarbeitungen unterlagen, wurden im Projekt `nexus--federation--streamFederation--operators--basic` in verschiedenen Paketen zusammengefasst.

8.1 Annahmen

Für die Entwicklung der vorliegenden Prototypen und auf Grund dessen, dass verschiedene Teile des Systems noch nicht vorhanden sind oder noch abgeändert werden müssen, werden folgende Annahmen getroffen:

- Die Operationen werden auf lokalen StreamNodes realisiert, das heißt, dass eine Anbindung an andere StreamNodes nicht vollzogen oder getestet wurde.

- Die in dieser Arbeit definierten Stromeigenschaften können nicht genutzt werden, da die Struktur des Systems diese derzeit nicht unterstützt. Eine Realisierung der Stromeigenschaften geht über den Rahmen dieser Arbeit hinaus und wird hier nicht weiter behandelt.

- Weiter können Punctations und Relationenobjekte, wie sie hier definiert wurden, ebenfalls nicht genutzt werden, da ihre Struktur und Eigenschaften erst noch in das System eingefügt werden müssen. Da dies nach Absprache nicht Gegenstand dieser Arbeit ist, müssen diese durch zukünftige Arbeite in das System eingefügt werden. Es wird in den Implementierungen der Komponenten auf die jeweilige Nutzung der Komponenten hingewiesen, so dass ein Einfügen und die Nutzung dieser erleichtert wird.

- Es wird ebenso angenommen, dass alle Funktionen, die durch das System angeboten werden, genau wie definiert funktionieren und genutzt werden können, da einige dieser Methoden und Klassen in dieser Implementierung nicht oder nur leicht abgeändert wiederverwendet werden. Von Teilen, die noch nicht im System vorhanden und nicht Gegenstand dieser Arbeit sind, wird angenommen, dass sie hinzugefügt werden, um die Funktionalität der Operationen zu gewährleisten und die definierte Funktionalität erfüllen. (z.B. Punctations, Relationenobjekte oder Stromeigenschaften)

- Des Weiteren wird ein ursprünglich temporäres Systemkonstrukt, die `PossibleObjectExcerptList`, weiterhin genutzt, da eine Ersetzung dessen nicht im Rahmen dieser Arbeit liegt. Dieses Konstrukt stellt eine Liste von Objekten dar, die die notwendigen Attribute besitzen, die für die Vergleiche in der Datenobjektverarbeitung notwendig sind.

8.2 Implementierung und Wiederverwendung

Um eine Wiederverwendung und bestmögliche Kompatibilität zu bereits bestehenden Systemteilen zu gewährleisten, wurden einige bereits vorhandene Klassen, für die Anpassung an das obige Operatorenkonzept, herangezogen und überarbeitet. Es sollte ein möglichst großer Teil des

bestehenden Systems übernommen werden, mit der Ausnahme der AWQL-Bestandteile, da diese in den kommenden Vorhaben von anderen Sprachkonstrukten abgelöst werden sollen. So wurden die weiterhin notwendigen Klassen dahingehend überarbeitet, dass keine AWQL-Bezüge mehr vorhanden sind.

Die folgende Liste soll einen Überblick geben, was bei Abschluss dieser Arbeit bereits alles implementiert, beziehungsweise was an vorhandenen Systemkomponenten bereits überarbeitet und erweitert werden konnte:

Operatoren:

An Operatoren konnten bereits alle bisher vorhandenen Operatoren überarbeitet und an die neue Klasse `ObjectTarget`, die die Klasse `AWQLTarget` ersetzen soll, angepasst werden. Des Weiteren wurde der zusätzliche Operator `ComplexCompOperator` implementiert und an die bereits bestehenden Operatoren, bezüglich Verwendung und Verhalten, angeglichen, damit er ebenfalls in einem Operatorenbaum genutzt werden kann. Diese Operatoren werden in dem Paket `de.uni_stuttgart.nexus.streamFederation.operators.basic.operators` zur Verfügung gestellt.

Weiter wurde die abstrakte Klasse `UDAOperator` und eine zugehörige Subklasse `CountUDAOperator`, als mögliches Beispiel für die Nutzung des UDA-Konzepts, in das Paket `de.uni_stuttgart.nexus.streamFederation.operators.basic.operations.uda` eingefügt. Diese Klassen werden näher in Kapitel **8.2.2.8** erläutert.

Operationen:

Im Bereich der Operationen konnten alle unter Kapitel **8.2.3** beschriebenen Operationen realisiert werden. Da Relationenobjekte und Punctations vom derzeitigen NEXUS-System nicht unterstützt werden, sind einzelne Methoden der Operationen, die diese nutzen, in ihrer Funktionalität eingeschränkt. Die Operationen sind in den entsprechenden Unterpaketen von `de.uni_stuttgart.nexus.streamFederation.operators.basic.operations` zu finden. So befindet sich zum Beispiel die Select – Operation in de.uni_stuttgart. nexus.streamFederation.operators.basic.operations.select.

Hilfsklassen:

Die Überarbeitung und Neuimplementierung im Bereich der Hilfsklassen und Werkzeuge umfasst einige Klassen, die weitestgehend in die Pakete `de.uni_stuttgart.nexus. streamFederation.operators.basic.util` bzw. `de.uni_stuttgart.nexus. streamFederation.operators.basic.awml`, oder entsprechende Unterpakete eingefügt wurden:

In `de.uni_stuttgart.nexus.streamFederation.operators.basic.awml`:

- `AWMLFactory`
- `AWMLrefimplFactory` (in Unterpaket `.refimpl`)

In `de.uni_stuttgart.nexus.streamFederation.operators.basic.util`:

- `ObjectTarget` (ehemals `AWQLTarget`)
- `RestrictionReader` (vergleiche `AWQLReader`)
- `StringReader`

Als Hilfsklasse gilt auch die Klasse `OperatorFactory` (ehemals `AWQLFactory`). Diese wurde ebenfalls in das Paket `de.uni_stuttgart.nexus.streamFederation.operators.basic.operators` eingefügt.

Um die Performanz-Tests durchführen zu können, wurden zwei weitere Klassen angepasst. Hierbei handelt es sich um die Klassen `GenericObjectSink` und `GenericObjectSource`.

Die Klasse `GenericObjectSink`, die sich in `de.uni_stuttgart.nexus.streamFederation.operators.basic.sink` befindet, entspricht einer minimal modifizierten `GenericObjectSink`-Klasse aus `de.uni_stuttgart.nexus.streamFederation.sinks.extended.vispipe.genericObjectSink`.

Die Klasse `GenericObjectSource`, die sich in `de.uni_stuttgart.nexus.streamFederation.operators.basic.source` befindet, entspricht zu großen Teilen der Klasse `ResultSetSource` in `de.uni_stuttgart.nexus.streamFederation.sources.extended.vispipe.resultSetSource`. Allerdings wurde sie insofern angepasst, dass sie keine ganzen `ResultSet`-Objekte auf den Datenstrom gibt, sondern einzelne Objekte der Klasse `GenericObject`.

Des Weiteren musste zusätzlich die Klasse ProcessLine in de.uni_stuttgart.nexus.streamFederation.sandbox so angepasst werden, dass die Methode deserialize() auch direkt Strings einlesen kann.

8.2.1 Befehlseinheiten der Command-Sprache (Command)

Das Verhalten von Operationen wird durch Parameter bestimmt. Diese Parameter werden gewöhnlich bei der Erstellung (setup) der Operationen durch die `ProcessLine` den Operationen übergeben und in diesen jeweils gespeichert. Eine Ausprägung eines solchen Parameters wäre zum Beispiel die Angabe für eine Selektions-Operation, welche Datenobjekte ausgewählt werden sollen und welche nicht.

Die Möglichkeit der Definition von Parametern in einem solchen Command beschränkt sich zum gegenwärtigen Zeitpunkt vorerst auf die Angabe eines Key-Value – Paares. Diese Daten werden zu den Eigenschaften einer Operation, den so genannten „Properties“, hinzugefügt. Die Daten bestehen derzeit nur aus einer Zeichenkette, deren verschiedene Werte durch Kommata und ähnliche Zeichen getrennt sind. So sind für die verschiedenen Operationen auch verschiedene Schlüsselwörter (Keys) der Key-Value – Paare zu benutzen. Um zu erkennen, für welche Operation die definierte Konfiguration bestimmt ist, muss als `parameterKey` ein entsprechender Wert für die Parameter in den Commands eingetragen werden.

Die erste Konfiguration von Operationen wird über die Parameter der Commands bei der Initialisierung der Operationen geladen. Anschließende Änderungen der Konfigurationen von Operationen werden nicht über Commands bewerkstelligt, sondern, aus Performanzgründen, über spezielle Datenobjekte in Datenströmen. Eine nachträgliche Änderung der Konfiguration wird allerdings nicht von allen Operationen unterstützt.

8.2.1.1 Command-Parameter für Assemble

Die Parameter für die Konfiguration der Operation `Assemble` bestehen hier nur aus der Angabe der Anzahl der eingehenden Datenströme. Diese Datenströme werden zur Ergänzung der Hauptdatenstromobjekte herangezogen.

Hierfür muss als `parameterKey` „assemble“ angegeben werden, um eine korrekte Identifikation der Operationskonfiguration und Zuweisung dieser durch die Methode `setProperty()` zu gewährleisten. Als `parameterValue` muss die Anzahl der unterstützenden

Datenströme inklusive des Hauptdatenstromes angegeben werden. Wenn also 2 ergänzende Datenströme genutzt werden sollen, muss als Anzahl die Zahl 3 definiert werden, da der Hauptdatenstrom mit eingerechnet werden muss.

Eine Konfiguration für die `Assemble` – Operation in einen Parameter eingefügt, könnte zum Beispiel so aussehen:

```
<eas:parameter xmlns:eas="...">
<nsas:value>
     <eas:blockId xmlns:eas="...">
          xy
     </eas:blockId>
     <eas:parameterKey xmlns:eas="...">
          assemble
     </eas:parameterKey>
     <eas:parameterValue xmlns:eas="...">
          3   (Anzahl der eingehenden Datenströme, inklusive des Hauptdatenstromes)
     </eas:parameterValue>
     <eas:parameterClass xmlns:eas="...">
          urn:java:java.lang.Integer
     </eas:parameterClass>
</nsas:value>
</eas:parameter>
```

Hierbei muss der Name der Operation, für die die Parameter bestimmt sind, unter `eas:blockId` anstatt „xy“ angegeben werden.

8.2.1.2 Command-Parameter für Select

Eine Restriktion für die `Select`-Operation kann entweder einfach oder komplex sein, das heißt aus einem oder mehreren `RestrictionOperators` bestehen. Um Kombinationen aus diesen `RestrictionOperators`, also eine komplexe Restriktion, zu erstellen, können zwei beschreibende Zeichenketten für jeweils einen Operator zu diesen komplexeren Anfragen mittels Semikolons (';') und logischen, boolschen Verbindungsoperatoren (and, or) kombiniert werden (zum Beispiel: Beschreibung1 ; and; Beschreibung2). Diese Verbindungsoperatoren werden durch `BooleanOperators` repräsentiert. Die beschreibenden Zeichenketten können jeweils wieder aus komplexen, zusammengesetzten Zeichenketten bestehen. Auf diese Weise können Baumstrukturen erstellt werden, die verschachtelten Anfragen entsprechen. Die Baumstruktur kann mittels Klammerungen ('(' und ')') dargestellt werden. Diese werden wie die Verbindungsoperatoren „and“, „or“ oder „not“ mittels des Semikolons (';') eingebunden. Für den Verbindungsoperator „not“ gilt zusätzlich, dass der zu negierende Teil, selbst wenn er nur aus einer Beschreibung eines Operators besteht, in Klammern stehen muss. (Bsp: not; (; Beschreibung1;)) (Siehe auch Beispiel unten)

Die einfachen Restriktionen werden definiert durch eine Zeichenkette, welche aus drei Hauptteilen besteht, die jeweils voneinander durch ein Komma (',') getrennt sind. Diese drei Teile beschreiben die linke Seite eines Operators, den Operator selbst und die rechte Seite. Hierbei sollte beachtet werden, dass der Operator der Zeichenkette hier der Klasse `Operator` des Systems entspricht, also einer atomaren Verarbeitungseinheit. Der Operator selber wird durch eine einfache Beschreibung der Verarbeitungsart repräsentiert, wie etwa durch den Ausdruck „less“, der angibt, dass es sich um einen „kleiner als“-Operator handelt. Es werden die Vergleichsoperatoren des Systems unterstützt. Die linke und rechte Seite bilden die jeweiligen, zu verarbeitenden Informationen. Sie bestehen aus einem so genannten Descriptor-Value – Paar. Ein Descriptor wird

von dem Value-Teil durch einen Punkt ('.') getrennt.

- Der Descriptor-Teil gibt an, auf welche Art von Daten sich der jeweilige Operand des Operators bezieht. Mögliche Werte sind:

  - **Constant**: Dieser Descriptor definiert den Operanden als einen konstanten Wert gegen den zum Beispiel verglichen werden soll.

  - **Attribute**: Dieser Descriptor definiert den Operanden als ein Attribut eines Datenobjektes, das verarbeitet werden soll.

  - **Reference**: Dieser Descriptor definiert den Operanden als ein Attribut eines entfernten Datenobjektes, auf das nur über die vorhandene OID zugegriffen werden kann. (Nutzung bei Relationenobjekten, die u.a. die OIDs der entsprechenden Objekte beinhalten.)

Der Value-Teil selbst kann aus mehreren, verschachtelten Value-Teilen bestehen, die ebenfalls durch Punkte getrennt sind. Diese folgen den Attribut-Definitionen des Nexus-Systems.

- Der erste der Value-Teile eines Operanden wird auch Instance-Teil genannt. Alle weiteren Teile sind die Part-Teile, angelehnt an die Namensgebung für Attribute des Systems. Der Instance-Teil definiert eine AttributeInstance inklusive des zugehörigen Namespaces, der vom Attribut-Namen durch einen Doppelpunkt (':') getrennt wird. Ein Part-Teil definiert einen AttributePart, also zum Beispiel einen Value- oder Meta-Teil (vgl. Nexus-System). Die AttributeInstances sowie die AttributeParts besitzen zusätzlich den zugehörigen Namespace, in dem sie definiert sind. Die Tiefe der Verschachtelungen der Value- und Meta-Teile, die hier definiert werden können, ist, genau wie im Nexus-System, theoretisch unbegrenzt.
  (Bsp: attribute.nsas:person.nsas:meta.nsas:adress.nsas:zipCode)

  - Für den Descriptor „**Attribute**“ besteht der Value-Teil aus einer AttributeInstance (Instance-Teil) und einem oder mehreren dazugehörigen AttributeParts (Part-Teil), getrennt durch einen Punkt ('.').
    (Bsp: attribute.nsas:seats.nsas:value)

  - Der Value-Teil des Descriptors „**Reference**“ besteht aus zwei Instance-Part – Kombinationen. Die erste gibt den Pfad zum Speicherort der OID des gesuchten Objektes an; die zweite gibt, wie bei Descriptor „Attribute“, das zu verarbeitende Attribut an. Beide werden durch einen Stern ('*') getrennt.
    (Bsp: reference.nsas:nol.nsas:value.*.nsas:seats.nsas:value)

  - Für den Descriptor „**Constant**“ besteht der Value-Teil aus einem Datentypen der Konstanten (z.B. string), sowie der zugehörige Namespace (z.B. „nsat“), und der Konstanten selbst(z.B. 'red'); beide Teile sind ebenfalls getrennt durch einen Punkt ('.').
    (Bsp: constant.nsat:string.red)

    Für geometrische Objekte besteht die Möglichkeit, diese über WKT anzugeben. Dies geschieht auf folgende Weise: Als Datentyp muss „nsat:wkt“ angegeben werden. Als Part-Teil des geometrischen Objekts wird die Kombination aus Code, für das gewünschte Referenzsystem, und die Beschreibung des geometrischen Objektes, wie bereits in AWQL und dem bisherigen System beschrieben, angegeben.

(Bsp: constant.nsat:wkt.'srs-code':POLYGON((4.123...))

Für temporale Objekte besteht die Möglichkeit, diese zu definieren, indem als Datentyp „nsat:time“ angegeben wird. Diesen Angaben folgen die zwei Zeit-Zeichenketten, die für die Klasse `CTime` notwendig sind. Diese sind durch einen Stern ('*') voneinander getrennt. Unterstützt wird derzeit das Format der ISO-8601.
(Bsp: constant.nsat:time.2005-11-22T10:05:00*2005-11-22T10:05:00)

Bei temporalen und geometrischen Objekten ist die Angabe des Namespaces „nsat“ bei dem Datentypen nur der Vollständigkeit wegen anzugeben.

Eine komplexe Restriktion in einen Parameter eingefügt, kann zum Beispiel so aussehen:

```
<eas:parameter xmlns:eas="...">
  <nsas:value>
     <eas:blockId xmlns:eas="...">
          xy
     </eas:blockId>
     <eas:parameterKey xmlns:eas="...">
          restriction
     </eas:parameterKey>
     <eas:parameterValue xmlns:eas="...">
       (; attribute.nsas:seats.nsas:value, greater, constant.nsat:integer.12;
       and; attribute.nsas:seats.nsas:value, less, constant.nsat:integer.15; );
       or; attribute.nsas:color.nsas:value, equal, constant.nsat:string.red
     </eas:parameterValue>
     <eas:parameterClass xmlns:eas="...">
          urn:java:java.lang.String
     </eas:parameterClass>
  </nsas:value>
</eas:parameter>
```

Hierbei muss der Name der Operation, für die die Parameter bestimmt sind, unter `eas:blockId` anstatt „xy“ angegeben werden. Der `parameterKey` „restriction“ muss hier angegeben werden, damit dem Restriktions-Parser die Zeichenkette unter `parameterValue` durch die Methode `setProperty()` übergeben werden kann. Da es sich bei der Zeichenkette um einen String handelt, muss dies unter `parameterClass` mittels `urn:java:java.lang.String` angegeben werden.

8.2.1.3 Command-Parameter für CartesianProduct

Die Parameter für die Konfiguration der Operation `CartesianProduct` bestehen hier nur aus den Angaben der zusätzlichen Verarbeitungsschritte, deren Ergebnisse als Attribute dem Relationenobjekt hinzugefügt werden sollen, wie zum Beispiel die Berechnung der Distanz der zu verarbeitenden Datenobjekte.

Hierfür muss als `parameterKey` „cartesianProduct“ angegeben werden, um eine korrekte Identifikation der Operation und Zuweisung der Konfiguration zu gewährleisten. Als `parameterValue` müssen für jedes zu berechnende Attribut Vorgänge definiert werden, die durch die Operation umgesetzt werden. So zum Beispiel die `parameterValue` – Angabe „distance“, die die Distanz der Datenobjekte als Attribut des Relationenobjekts speichert. Zusätzliche Angaben werden mittels Kommata (',') aufgelistet. Die Liste der zusätzlichen Verarbeitungen ist frei erweiterbar, falls die entsprechenden Methoden für die jeweiligen Zeichenketten ebenfalls entsprechend erweitert werden. Die Verarbeitungen sowie die entsprechenden Schlüsselwörter

müssen also der Operation `CartesianProduct` kenntlich gemacht werden.

Eine Konfiguration für die `CartesianProduct` – Operation in einen Parameter eingefügt, könnte zum Beispiel so aussehen:

```
<eas:parameter xmlns:eas="...">
<nsas:value>
     <eas:blockId xmlns:eas="...">
          xy
     </eas:blockId>
     <eas:parameterKey xmlns:eas="...">
          cartesianProduct
     </eas:parameterKey>
     <eas:parameterValue xmlns:eas="...">
          distance
     </eas:parameterValue>
     <eas:parameterClass xmlns:eas="...">
          urn:java:java.lang.String
     </eas:parameterClass>
</nsas:value>
</eas:parameter>
```

Hierbei muss der Name der Operation, für die die Parameter bestimmt sind, unter `eas:blockId` anstatt „xy“ angegeben werden.

8.2.1.4 Command-Parameter für FetchFirst

Die Angaben der Parameter für die Konfiguration der Operation `FetchFirst` beschränken sich auf die Angabe der Anzahl der Elemente, die ausgegeben werden sollen, als Integerwert.

Eine Konfiguration für die `FetchFirst` – Operation in einen Parameter eingefügt, könnte zum Beispiel so aussehen:

```
<eas:parameter xmlns:eas="...">
<nsas:value>
     <eas:blockId xmlns:eas="...">
          xy
     </eas:blockId>
     <eas:parameterKey xmlns:eas="...">
          fetch
     </eas:parameterKey>
     <eas:parameterValue xmlns:eas="...">
          4
     </eas:parameterValue>
     <eas:parameterClass xmlns:eas="...">
          urn:java:java.lang.Integer
     </eas:parameterClass>
</nsas:value>
</eas:parameter>
```

Hierbei muss der Name der Operation, für die die Parameter bestimmt sind, unter `eas:blockId` anstatt „xy“ angegeben werden. Der `parameterKey` „fetch“ muss hier angegeben werden, damit die verarbeitende Methode `setProperty()` den Parameter identifizieren und verarbeiten kann. Da es sich bei der Zeichenkette des eigentlichen Parameterwertes um einen Integerwert handelt, muss dies unter `parameterClass` mittels `urn:java:java.lang.Integer` angegeben werden.

8.2.1.5 Command-Parameter für Sort

Die Parameter für die Konfiguration der Operation `Sort` bestehen hier aus den Angaben des Attributes eines Datenobjektes, nach dem sortiert werden soll, sowie der Sortierungsrichtung. Das Attribut wird mit dem vollständigen Namen des Attributpfades, wie er im System genutzt wird, beschrieben. Für die Sortierungsrichtung sind nur zwei Richtungen möglich: „up“ für eine aufsteigende und „down“ für eine absteigende Sortierungsrichtung. Diese Daten werden als eine Zeichenkette als `parameterValue` angegeben, wobei zuerst die Sortierungsrichtung und dann das Sortierungsattribut angegeben wird. Getrennt werden die Werte durch einen Punkt ('.'). Als `parameterKey` wird „sort“ angegeben, um die Konfiguration der Operation zuzuordnen. Des Weiteren wird als `parameterClass` mit `urn:java:java.lang.String` angegeben, dass es sich bei dem `parameterValue` um eine Zeichenkette handelt.

Eine Konfiguration für die `Sort` – Operation könnte zum Beispiel so aussehen:

```
<eas:parameter xmlns:eas="...">
<nsas:value>
     <eas:blockId xmlns:eas="...">
          xy
     </eas:blockId>
     <eas:parameterKey xmlns:eas="...">
          sort
     </eas:parameterKey>
     <eas:parameterValue xmlns:eas="...">
          up.nsas:seats.nsas:value
     </eas:parameterValue>
     <eas:parameterClass xmlns:eas="...">
          urn:java:java.lang.String
     </eas:parameterClass>
</nsas:value>
</eas:parameter>
```

Hierbei muss der Name der Operation, für die die Parameter bestimmt sind, unter `eas:blockId` anstatt „xy“ angegeben werden.

8.2.1.6 Command-Parameter für Group

Ähnlich wie bei den Command-Parametern für die Operation `Fetch`, wird bei den Command-Parametern für die Operation `Group` nur der `parameterKey`, sowie die Anzahl der zu gruppierenden Elemente definiert.

Eine Konfiguration für die `Group` – Operation in einen Parameter eingefügt, könnte zum Beispiel so aussehen:

```
<eas:parameter xmlns:eas="...">
<nsas:value>
     <eas:blockId xmlns:eas="...">
          xy
     </eas:blockId>
     <eas:parameterKey xmlns:eas="...">
          group
     </eas:parameterKey>
     <eas:parameterValue xmlns:eas="...">
          2
     </eas:parameterValue>
```

```
     <eas:parameterClass xmlns:eas="...">
          urn:java:java.lang.Integer
     </eas:parameterClass>
</nsas:value>
</eas:parameter>
```

Hierbei muss der Name der Operation, für die die Parameter bestimmt sind, unter `eas:blockId` anstatt „xy“ angegeben werden. Der `parameterKey` „group“ muss hier angegeben werden, damit die verarbeitende Methode `setProperty()` den Parameter identifizieren und verarbeiten kann. Da es sich bei der Zeichenkette des eigentlichen Parameterwertes um einen Integerwert handelt, muss dies unter `parameterClass` mittels `urn:java:java.lang.Integer` angegeben werden.

8.2.1.7 Command-Parameter für UDA

Die Informationen, die für die Konfiguration der `UDA` – Operation angegeben werden müssen, sind Informationen über die INITALIZE – und ITERATE – Phase. Beide werden, getrennt durch ein Semikolon (';'), als Zeichenkette in `parameterValue` angegeben. Der erste der beiden Teile ist die Information für die INITIALIZE – Phase. Um dies anzuzeigen, wird die Teilzeichenkette durch „init:“ angeführt, gefolgt von einem Datentypen, der durch den internen Speicher unterstützt werden soll. Die zweite Teilzeichenkette wird angeführt durch „iter:“ gefolgt von dem Namen eines Operators, der die komplette gewünschte Funktionalität für die UDA – Operation zur Verfügung stellt. Der hier angegebene Operator muss vor der Benutzung allerdings bereits erstellt, sowie unter `de.uni_stuttgart.nexus.streamFederation.operators.basic.operations.uda` eingefügt worden sein. Er wird dann zur Laufzeit geladen. Als Vorlage für diese Operatoren dient die abstrakte Klasse `UDAOperator` oder deren Subklasse `CountUDAOperator`, die ebenfalls dort zu finden ist.

Eine Konfiguration für die `UDA` – Operation könnte zum Beispiel so aussehen:

```
<eas:parameter xmlns:eas="...">
<nsas:value>
     <eas:blockId xmlns:eas="...">
          xy
     </eas:blockId>
     <eas:parameterKey xmlns:eas="...">
          uda
     </eas:parameterKey>
     <eas:parameterValue xmlns:eas="...">
          init:integer; iter:CountUDAOperator
     </eas:parameterValue>
     <eas:parameterClass xmlns:eas="...">
          urn:java:java.lang.String
     </eas:parameterClass>
</nsas:value>
</eas:parameter>
```

Hierbei muss der Name der Operation, für die die Parameter bestimmt sind, unter `eas:blockId` anstatt „xy“ angegeben werden. Der `parameterKey` „uda“ muss hier angegeben werden, damit die verarbeitende Methode `setProperty()` den Parameter identifizieren und verarbeiten kann. Da es sich bei der Zeichenkette des eigentlichen Parameterwertes um eine Zeichenkette handelt, die später geparst werden muss, muss dies unter `parameterClass` mittels `urn:java:java.lang.String` angegeben werden.

8.2.1.8 Command-Parameter für Return

Für die Konfiguration der Operation `Return` muss neben dem `parameterKey` auch das Attribut definiert werden, an dessen Stelle eine OID gefunden werden kann, zu der ein Datenobjekt gefunden und ausgegeben werden soll.

Der `parameterKey` wird hier durch „return“ definiert, um der `setProperty()` Methode der `Return` – Operation anzuzeigen, dass es sich hierbei um ihre Konfiguration handelt. Als Werte des Parameters wird ein vollständiges Attribut angegeben, dass die zu benutzende OID enthält. Als Beispiel soll hier ein Datenobjekt gesucht werden, dessen OID als Bezugsinformation in einem Relationenobjekt als „obj1“ gespeichert wurde.

Eine Konfiguration für die `Return` – Operation in einen Parameter eingefügt, könnte zum Beispiel so aussehen:

```
<eas:parameter xmlns:eas="...">
<nsas:value>
     <eas:blockId xmlns:eas="...">
          xy
     </eas:blockId>
     <eas:parameterKey xmlns:eas="...">
          return
     </eas:parameterKey>
     <eas:parameterValue xmlns:eas="...">
           nsas:obj1.nsas:value
     </eas:parameterValue>
     <eas:parameterClass xmlns:eas="...">
          urn:java:java.lang.String
     </eas:parameterClass>
</nsas:value>
</eas:parameter>
```

Hierbei muss der Name der Operation, für die die Parameter bestimmt sind, unter `eas:blockId` anstatt „xy“ angegeben werden.

8.2.2 Operatoren

8.2.2.1 RestrictionOperator (abstrakt)

Alle Operatoren werden von der abstrakten Klasse `RestrictionOperator` abgeleitet. Diese Klasse lag schon zu einem früheren Zeitpunkt im System vor, wurde aber, um unabhängig von AWQL zu sein, überarbeitet. Weiter beziehen sich die Änderungen auf Namensänderungen von Variablen, sowie auf den Verzicht auf die Methode `serialize`. Diese Änderung wurde durchgeführt, da die Methode bereits in anderen Operatoren als „deprecated“ deklariert wurden.

8.2.2.2 BooleanOperator

Dieser Operator wird vor allem als Verbindungsglied zwischen anderen Operatoren genutzt, um so komplexere Anfragen und Restriktionsbäume zu ermöglichen. Dabei werden die Ergebnisse der anderen Operatoren mittels boolschen Operatoren (z.B. „and“) verknüpft. Weiter kann der Operator

für die Verarbeitung von einfacheren boolschen Werten verwendet werden. Bei diesem Operator wurden Namensänderungen vor allem der Variablen durchgeführt, sowie die Methode `serialize()` entfernt, die im ursprünglichen `BooleanOperator` mit „deprecated“ gekennzeichnet wurde.

8.2.2.3 ComparisonOperator (abstrakt)

Von diesem abstrakten Operator, der selbst von `RestrictionOperator` abgeleitet wurde, werden alle Operatoren abgeleitet, die Verarbeitungen durchführen mittels zum Beispiel „größer“ oder „kleiner“ Vergleichen. Auch hier wurden nur Namensänderungen und Änderungen bezüglich der Verwendung der Klasse `AWQLTarget` durchgeführt. Stattdessen wird die Klasse `ObjectTarget` verwendet.

8.2.2.4 SimpleCompOperator

Von der abstrakten Klasse `ComparisonOperator` abgeleitet, dient der bereits vorhandene `SimpleCompOperator` dazu, einfache Vergleiche zwischen einem Attribut, repräsentiert durch die Klasse `ObjectTarget`, und einer Konstanten durchzuführen. Es wurden verschiedene Änderungen und Anpassungen an dieser Klasse vorgenommen. So wurden alle Bezüge zu AWQL (z.B. `AWQLTarget`) entfernt und ersetzt durch angepasste Klassen, Methoden oder Namen (z.B. `ObjectTarget`). Ebenso wurden alle mit „deprecated“ gekennzeichneten Methoden, wie `addCompValues()` und `serialize()` entfernt.

8.2.2.5 SpatialCompOperator

Ebenfalls von der abstrakten Klasse `ComparisonOperator` abgeleitet, dient der bereits vorhandene `SpatialCompOperator` dazu, örtliche Vergleiche zwischen einem Attribut, repräsentiert durch die Klasse `ObjectTarget`, und einer Raumkonstanten durchzuführen. Es wurden auch verschiedene Änderungen und Anpassungen an dieser Klasse vorgenommen. So wurden auch hier alle Bezüge zu AWQL (z.B. `AWQLTarget`) entfernt und ersetzt durch angepasste Klassen, Methoden oder Namen (z.B. `ObjectTarget`). Ebenso wurden alle mit „deprecated“ gekennzeichneten Methoden, wie `serialize()` entfernt.

8.2.2.6 TemporalCompOperator

Ebenfalls von der abstrakten Klasse `ComparisonOperator` abgeleitet, dient der bereits vorhandene `TemporalCompOperator` dazu, zeitliche Vergleiche zwischen einem Attribut, repräsentiert durch die Klasse `ObjectTarget`, und einer Zeitkonstanten durchzuführen. Auch hier wurden verschiedene Änderungen und Anpassungen an der Klasse vorgenommen. So wurden auch hier alle Bezüge zu AWQL (z.B. `AWQLTarget`) entfernt und ersetzt durch angepasste Klassen, Methoden oder Namen (z.B. `ObjectTarget`). Ebenso wurden alle mit „deprecated“ gekennzeichneten Methoden, `serialize()` entfernt.

8.2.2.7 ComplexCompOperator

Diese Klasse wird von der abstrakten Klasse `RestrictionOperator` abgeleitet und dient dazu, komplexere Vergleiche, also zum Beispiel Vergleiche zwischen zwei Datenobjekten, zu realisieren. Dieser Operator führt selbst keine Vergleiche durch, sondern delegiert die Vergleiche an einfachere Operatoren, wie den `SimpleCompOperator`, weiter. Dazu werden die Operatoren erstellt und anschließend zur Datenverarbeitung genutzt. Die Ableitung von `RestrictionOperator` erfolgt deshalb, damit der `ComplexCompOperator` wie die anderen Operatoren, in einen Operatorenbaum eingebunden werden kann.

Der `ComplexCompOperator` kann für Attribute-Attribute – Vergleich, zwei Attribute des gleichen Datenobjekts, und für Reference-Reference – Vergleiche, zwei Attribute von zwei verschiedenen Datenobjekten, genutzt werden. Hierfür wird unter anderem eine boolesche Variable bei der Erstellung des Operators übergeben, die angibt, ob es sich um einen Reference-Reference – Vergleich handelt. Falls dies der Fall ist, wird zusätzlich zu den zwei `ObjectTargets`, über die die Zielattribute für den Vergleich angesprochen werden können, über weitere zwei `ObjectTargets` auch das Attribut angegeben, in dem die OIDs der zu verwendenden Datenobjekte gespeichert wird.

Die von der Klasse `RestrictionOperator` vorgegebenen, zu implementierenden Methoden wurden von der Klasse `ComparisonOperator` übernommen oder leer gelassen, da der `ComplexCompOperator` hauptsächlich von `RestrictionOperator` abgeleitet wurde, damit er in einen Operatorenbaum eingefügt werden kann. Neben verschiedenen Konstruktoren und diversen setter- und getter-Methoden, sind in diesem Operator folgende Methoden vorhanden:

Methoden:

Einer der vorhandenen Konstruktoren ist zum Beispiel folgender:

- `ComplexCompOperator(int, ObjectTarget[], ObjectTarget[], boolean)`

Dieser Konstruktor, der in verschiedenen Versionen verfügbar ist, erhält Informationen über den durchzuführenden Vergleichstyp und ob es sich um einen Reference-Reference – Vergleich handelt. Ist dies der Fall, erhält der Konstruktor zwei `ObjectTarget`-Arrays. Hierbei repräsentiert das jeweils erste Element der Arrays den Speicherort der zu nutzenden OID. Die jeweils zweiten `ObjectTargets` repräsentieren die Attribute, die zum Vergleich herangezogen werden sollen. Falls es sich jedoch um einen Attribute-Attribute – Vergleich handelt, so werden nur zwei `ObjectTargets` übergeben, die nur die miteinander zu vergleichenden Attribute definieren. Das erste `ObjectTarget`-Array repräsentiert die linke Seite des Operators, das zweite die rechte Seite.

- `evaluateOperator(GenericObject) : boolean`

Diese Methode wird direkt von einer Operation (wie zum Beispiel `Select`) aufgerufen, um ein erhaltenes `GenericObject` zu verarbeiten. Ziel dieser Methode ist es, je nach Vergleichstyp, einen entsprechenden, einfachen `ComparisonOperator` mit zu extrahierenden Informationen zu erzeugen, der den Vergleich durchführt.

Falls es sich bei den Vergleichen um einen Reference-Reference – Vergleich handelt, werden die entsprechenden, zu verwendenden Datenobjekte ermittelt. Dies geschieht durch die Methode `getReferenceObject()`. Da derzeit allerdings eine solche Schnittstelle für OID-Anfragen an einen entsprechenden Server noch nicht vom System bereit gestellt wird, ist die Funktionalität dieser Methode noch nicht gegeben. Falls es sich um Attribute-Attribute – Restriktionen handelt,

wird dieser Schritt übersprungen, und das übergebene Datenobjekt genutzt.

Anschließend wird von einem der entsprechenden Zielobjekte, ein Zielattribut als `GenericAttributePart` mittels `getTargetAttributePart()` extrahiert. Nun wird mit den ermittelten Teilen ein entsprechender `ComparisonOperator` erstellt, d.h. zum Beispiel einen `SimpleCompOperator`, der den eigentlichen Vergleich vornimmt. Das Ergebnis dieses Operators wird anschließend zurückgegeben.

- `getReferenceObject(GenericObject, ObjectTarget) : GenericObject`

Diese Methode ermittelt ein `GenericObject` anhand einer OID, deren Speicherort durch das `ObjectTarget` auf dem übergebenen `GenericObject` definiert ist. Da derzeit, wie bereits oben geschildert, eine entsprechende Schnittstelle fehlt, ist die Funktionalität der Methode eingeschränkt.

- `getTargetAttributePart(GenericObject, ObjectTarget) : GenericAttributePart`

Diese Methode ermittelt zu einem `GenericObject` und einem `ObjectTarget` ein Attribut (genauer: `GenericAttributePart`) mit dem ein `ComparisonOperator` erstellt werden kann, der dann anschließend den gewünschten Vergleich durchführt.

8.2.2.8 Operatoren für UDA – Operationen

Diese Operatoren unterscheiden sich insofern von den anderen, oben aufgeführten Operatoren, dass diese Operatoren nur einfache Funktionalitäten erfüllen, und einen einfachen internen Speicher besitzen, wie zum Beispiel ein einfacher Zähloperator, der `CountUDAOperator`. Der Zähloperator besitzt nur eine Integer-Variable als internen Speicher. Als Funktionalität zählt der Operator alle eingehenden Objekte und gibt nach jeder Aktivierung des Operators ein Zwischenergebnis zurück. Es können aber auch Operatoren erstellt werden mit einem flexibleren internen Speicher. Die Operatoren erhalten Datentypangaben, mittels denen z.B. Listen, die evtl. die internen Speicher darstellen, konfiguriert werden können. Die Unterstützung der jeweiligen Datentypen hängt vom benutzten Operator und seinen Spezifikationen ab.

Alle Operatoren, die für UDA-Operationen erstellt werden, müssen von der abstrakten Klasse `UDAOperator` abgeleitet werden. Es kann entschieden werden, ob bei der Erstellung eines Operators die zu benutzenden Datentypen des internen Speichers über die Konfiguration bestimmt werden soll, oder ob die Variablen fix sein sollen. Die abstrakte Oberklasse bietet nur die abstrakte Methode `iterate()`, die bei jedem Aufruf ausgeführt werden muss. Weiter werden Setter- und Getter-Methoden zur Verfügung gestellt.

Methoden:

- `getUsedType() : String`
- `setUsedType(String) : void`
- `getAWMLFac() : AWMLFactory`
- `setAWMLFac(AWMLFactory) : void`

Diese Methoden bearbeiten u.a eine Variable, in der gespeichert wird, welcher Datentyp genutzt wird. So kann auch zum Beispiel mittels der Setter-Methoden der zu benutzende interne Speicher (die Variable), falls gewünscht, initialisiert werden. Weiter kann über diese Methoden die `AWMLFactory` bearbeitet werden, die für die Erstellung des Ergebnis-Objekts benötigt wird.

- `iterate(GenericObject) : GenericObject (abstrakt)`

Jede spezifische Ausprägung dieser abstrakten Klasse, muss diese Methode mit der jeweils gewünschten Funktion implementieren. Diese Methode verarbeitet das übergebene Datenobjekt und generiert wiederum ein Datenobjekt, das ein Abbild des aktuellen, internen Speichers repräsentiert, und gibt dieses zurück, damit es über den ausgehenden Datenstrom versendet werden kann. Derzeit liegt im System aber keine Definition für ein solches Datenobjekt, das ein Zwischenergebnis repräsentiert vor.

Als Beispiel eines `UDAOperators` wurde der einfache `CountUDAOperator` erstellt. Er implementiert auf einfache Weise einen objekt-zählenden Operator mit fixem internen Speicher. Er ist ebenfalls in `de.uni_stuttgart.nexus.streamFederation.operators.basic.operations.uda` zu finden. Derzeit ist allerdings ein Datenobjekt für die einfachen, zurückzugebenden Zwischenergebnisse noch nicht im System vorgesehen, weshalb die Funktionalität des Operators eingeschränkt ist.

8.2.3 Operationen

Alle Operationen werden von der abstrakten Klasse `AbstractUnlinkedInputsOperation` abgeleitet. Hierbei handelt es sich um Operationen, deren eingehende Datenströme (inputs) entsprechend frei definiert werden können und nicht synchronisiert sind. Das heißt, dass alle Daten, die über einen beliebigen eingehenden Datenstrom empfangen werden an die Methode `process()` weitergegeben werden. Jede von dieser Oberklasse abgeleitete Operation, muss die Methode `process()` implementieren, in der die Verarbeitung der zugeführten Datenobjekte geregelt wird. Des Weiteren hat jede Operation standardmäßig zwei Konstruktoren, die aber nicht weiter angegeben werden. Diese setzen die Input- und Output-Streams. Sie unterscheiden sich nur darin, dass eine Operation entweder mit oder ohne individuellen Namen initialisiert werden können.

Alternativ dazu können Operationen von der im System bereits vorhandenen abstrakten Klasse `AbstractLinkedInputsOperation` abgeleitet werden, in der die eingehenden Datenströme synchronisiert werden. Das heißt, dass die Datenobjekte erst dann zur Bearbeitung freigegeben werden, wenn aus jedem eingehende Datenstrom ein Objekt vorliegt. Soweit nichts anderes vermerkt ist, sind die Operationen von der abstrakten Klasse `AbstractUnlinkedInputsOperation` abgeleitet.

Die Grundkonfiguration für Operationen werden durch das SetupCommand über Parameter definiert. Falls Änderungen an der Konfiguration einer Operation notwendig werden sollten, so werden diese aus Performanzgründen nicht wieder über die Parameter eines entsprechenden UpdateCommands definiert, sondern über ein Datenobjekt, das direkt in den Datenstrom eingespeist wird. Die Operationen prüfen die eingehenden Objekte in der Methode `isUpdateCommand()` darauf, ob es sich um Konfigurationsänderungen handelt. Falls dies der Fall ist, so werden die Änderungen verarbeitet. Dies wird jedoch nicht von allen Operationen unterstützt. Weiter greifen Operationen, die Punctations verwenden, auf die Methode `isPunctation()` zurück, um ein Datenobjekt dahingehend zu untersuchen, ob es sich um eine solche Punctation handelt.

- `isUpdateCommand(GenericObject) : boolean`
- `isPunctation(GenericObject) : boolean`

Diese Methoden überprüfen, ob es sich bei einem übergebenen Datenobjekt um ein UpdateCommand-Objekt bzw. eine Punctation handelt. Die Methode `isPunctation()` ist derzeit, da noch keine Definitionen für Punctations in das System eingefügt wurden, in der Funktionalität eingeschränkt.

8.2.3.1 Assemble (entspricht: Assemble – Operator)

Die `Assemble` – Operation kombiniert die Datenobjekte mehrerer eingehender Datenströme zu komplexeren Datenobjekten. Die Anzahl der eingehenden Datenströme ist dabei beliebig, wobei allerdings nur ein eingehender Datenstrom kontinuierlich sein darf. Dieser wird als Hauptdatenstrom bezeichnet. Die Anzahl der eingehenden Datenströme kann über die Parameter in der Konfiguration gespeichert werden. Diese Informationen werden durch die Methode `setProperty()` ausgelesen und gesetzt. Da eine dynamische Bindung oder Trennung von Datenströmen durch das System nicht unterstützt wird, wird hier eine Aktualisierung der Konfiguration zur Laufzeit nicht unterstützt.

Die Funktion dieser Operation bezieht sich dabei darauf, dass über den Hauptdatenstrom eingehende Datenobjekte eine Art Grundelement des zusammenzufügenden Datenobjekts bilden. Es gibt die OID des Objektes vor, mittels der die zugehörigen Elemente aus den anderen, endlichen Datenströmen bestimmt werden.

Erhält die Operation aus dem Hauptdatenstrom ein Datenobjekt, so werden weitere Annahmen von Elementen aus diesem Datenstrom abgelehnt, bis das Objekt fertig zusammengesetzt und weitergesendet wurde. Dies wird durch die Methode `process()` geregelt. Zu jedem solchen Hauptdatenobjekt werden alle endlichen, eingehenden Datenströme nach zugehörigen Datenobjektteilen durchsucht. Hierfür ruft die Methode `process()` für jedes neue Element aus einem endlichen Datenstrom die Methode `checkObject()` auf. Falls mittels dieser Methode ein Objekt gefunden wurde, ruft `process()` anschließend die Methode `assembleObjects()` auf, die den eigentlichen Zusammenbau durchführt. Die Daten der endlichen Datenströme werden immer wieder gesendet, wobei jeder Durchgang der endlichen Datenmengen über eine Punctation angezeigt wird. Wurde das Hauptdatenobjekt mit allen Daten zusammengesetzt, so wird es auf den ausgehenden Datenstrom gelegt.

Außer den unten aufgeführten Methoden wird bei dieser Operation auf die Methode `isPunctation()` zurückgegriffen. (Siehe **8.2.3**)

Methoden:

- `process(int, Object) : boolean`

Diese Methode ist die Standardverarbeitungsmethode, die aufgerufen wird, falls ein Objekt an den Operator geschickt wurde. Als Eingaben erhält sie die Nummer des eingehenden Datenstroms, von welchem das Objekt geschickt wurde, sowie das Objekt selbst. Der erste eingehende Datenstrom wird hierbei als kontinuierlicher Hauptdatenstrom definiert, der die zu ergänzenden Datenobjekte liefert. Es wird immer nur dann ein Datenobjekt aus dem Hauptdatenstrom angenommen, falls nicht schon eines zur Verarbeitung zwischengespeichert wurde. Falls Datenobjekte aus den anderen eingehenden Datenströmen an die Operation gesendet werden, werden diese an `checkObject()` weitergegeben. Falls in dieser Methode festgestellt wird, dass dieses Datenobjekt zum zu ergänzenden Datenobjekt passt, so wird es weiter an die Methode `assembleObjects()`

weitergegeben. Falls in allen Ergänzungsdatenströmen alle Elemente der endlichen Datenmenge überprüft wurden, wird das zusammengesetzte Datenobjekt auf den ausgehenden Datenstrom gegeben.

- `checkObject(GenericObject) : boolean`

Diese Methode überprüft anhand der OID des zu ergänzenden, zwischengespeicherten und des neu empfangenen Datenobjektes, ob sie übereinstimmen, und so das neue Datenobjekt einen hinzuzufügenden Teil des Datenobjektes des Hauptdatenstromes, das zu ergänzen ist, ausmacht. Ist dies der Fall, wird ein entsprechender boolescher Wert zurückgegeben.

- `assembleObjects(GenericObject) : boolean`

Falls beide Datenobjekte, das übergebene eines Ergänzungsstromes und das zwischengespeicherte des Hauptdatenstromes, miteinander kompatibel sind, werden die Attributinstanzen des hinzuzufügenden, neu-erhaltenen Objektes denen des Basisdatenobjekts hinzugefügt. Es wird ein entsprechender boolescher Wert zurückgegeben.

- `setProperty(String, Object) : boolean`

Diese Methode liest die zu dieser Operation gehörende Konfiguration aus den Parametern aus. (Siehe hierfür auch **8.2.1.1**) Diese Informationen beziehen sich auf die Anzahl der eingehenden Datenströme. Wenn die Informationen ermittelt wurden, wird mittels diesen die eingehenden Datenströme für diese Operation definiert.

8.2.3.2 Select (entspricht: Selektions-Operator)

Obwohl die `Select` - Operation zu gewissen Bestandteilen bereits vorhanden war, wurde das Verhalten der Operation notwendigen Anpassungen unterzogen. Da nur ein geringer Teil des Verhaltens der Operation wiederverwendet wurde, wird die Realisierung dieser Komponente als Neu-Implementierung angesehen.

Nach der Erstellung der `Select`-Operation durch eine `ProcessLine`, wird der Operation ein Parameter übergeben. In einer eigenen `setProperty()` – Methode, die die ursprüngliche Methode der Oberklasse überschreibt, werden die Werte der Restriktion, die kommasepariert in einer Zeichenkette vorliegen, durch einen externen `RestrictionReader` geparst. Mittels diesen geparsten Werten können die benötigten Operatoren bestimmt und gespeichert werden. Der Aufbau der Zeichenkette ist Kapitel **8.2.1.2** zu entnehmen.

Diese Klasse besitzt einen eingehenden und zwei ausgehende Datenströme. Der erste der ausgehenden Datenströme führt alle Datenobjekte, für die die definierten Restriktionen zutreffen, der zweite führt die restlichen Datenobjekte. Ebenso wird hier die Methode `isUpdateCommand()` (siehe **8.2.3**) genutzt.

Methoden:

- `process(int, Object) : boolean`

Diese Methode ist die Standardverarbeitungsmethode, die aufgerufen wird, falls ein Objekt an den

Operator geschickt wurde. Als Eingaben erhält sie die Nummer des eingehenden Datenstroms, von welchem das Objekt geschickt wurde, sowie das Objekt selbst. Hier wird nun die Methode `checkData()` mit dem Objekt als Parameter aufgerufen. Falls die Methode für dieses Objekt einen positiven booleschen Wert zurück erhält, wird das Objekt in den ersten ausgehenden Datenstrom gegeben, andernfalls in den zweiten.

- `checkData(GenericObject) : boolean`

In dieser Methode werden zuerst der definierte `RestrictionOperator`, bzw. `RestrictionOperator`-Baum, welche in den Properties gespeichert sind, aufgerufen. Anhand diesen wird das erhaltene Datenobjekt geprüft und es wird zurückgegeben, ob das Datenobjekt die Bedingungen erfüllt oder nicht.

- `setProperty(String, Object) : boolean`

Mittels dieser Methode werden die gewünschten `RestrictionOperators` aus der Konfiguration für die Operation geparst und in den Properties gespeichert. Die Methode erhält über die Parameter einen Property-Schlüssel, anhand dessen die Konfiguration für die Operation identifiziert werden kann. Handelt es sich hierbei um eine solche Restriktion, so wird einem `RestrictionReader` die Zeichenkette mit den definierten Bedingungen übergeben. Anschließend erhält die Methode einen fertig definierten `RestrictionOperator`, bzw. einen Operatoren-Baum zurück und speichert diesen in den Properties.

8.2.3.3 CartesianProduct (entspricht: Kreuzprodukt – Operator)

Die `CartesianProduct` – Operation besitzt zwei eingehende und einen ausgehenden Datenstrom. Der erste eingehende Datenstrom, der Hauptdatenstrom, darf kontinuierlich sein. Der zweite eingehende Datenstrom muss endlich sein, um die Durchführbarkeit der Operation zu gewährleisten.

Erhält die Methode `process()` aus dem kontinuierlichen Datenstrom ein Datenobjekt, so wird es zwischengespeichert und anschließend mit allen Elementen aus dem zweiten, endlichen Datenstrom kombiniert. Dazu werden alle Elemente aus dem zweiten Datenstrom zwischengespeichert, aber gleichzeitig auch ausgegeben (vergleiche Kapitel **7.1.3**). Für die Verarbeitung wird die Methode `createPunctation()` aufgerufen, anschließend übergibt die Methode die jeweils zu kombinierenden Datenobjekte an die Methode `combineObjects()`, die wiederum die Berechnung der zusätzlich zu erzeugenden Attribute mittels der Methode `addAttributes()` vornimmt. Von `combineObjects()` erhält die Methode `process()` ein Relationenobjekt zurück, das auf den ausgehenden Datenstrom gegeben wird. Nachdem alle Relationenobjekte erstellt und ausgegeben wurden, wird mit dem nächsten Element aus dem kontinuierlichen Datenstrom weiter verfahren.

Die Methode `addAttributes()` führt alle über die Konfiguration definierten Funktionen zur Berechnung der zusätzlichen Attribute aus, und fügt zum aktuellen Relationenobjekt die entsprechenden Attribute hinzu. Hierfür muss in dieser Methode für jedes Schlüsselwort eine Methode angegeben werden, die genutzt werden soll. Ebenso muss ein entsprechendes Attribut vom Relationenobjekt unterstützt werden.

Die Konfigurationsparameter, die bei der Erstellung der Operation der Methode `setProperty()` übergeben werden, werden dadurch für die Methode kenntlich gemacht, dass der `parameterKey` die Zeichenkette „cartesianProduct“ enthält. Der `parameterValue` dagegen

enthält die Informationen, die für die Verhaltensweise der Operation zuständig sind. So wird hier angegeben, welche Funktionen ausgeführt und deren Ergebnisse als zusätzliche Attribute dem entstehenden Relationenobjekt beigefügt werden. Der Aufbau der Parameterzeichenkette für die Konfiguration der Operation ist Kapitel **8.2.1.3** zu entnehmen. Die Operation greift außerdem auf die Methode `isPunctation()` und `isUpdateCommand()` zurück, die in **8.2.3** bereits aufgeführt wurden. Es werden Änderungen der Konfiguration durch UpdateCommands, die sich im Hauptdatenstrom befinden, unterstützt. Hierfür werden die Daten ausgelesen und `setProperty()` übergeben.

Methoden:

- `process(int, Object) : boolean`

Diese Methode ist die Standardverarbeitungsmethode, die aufgerufen wird, falls ein Objekt an den Operator geschickt wurde. Als Eingaben erhält sie die Nummer des eingehenden Datenstroms, von welchem das Objekt geschickt wurde, sowie das Objekt selbst.

Diese Methode überwacht vor allem den Empfang der erhaltenen Objekte. Zu jeder Zeit darf nur ein Datenobjekt des Hauptdatenstroms verarbeitet werden. Zu Beginn erhält die Methode alle Elemente des zweiten, endlichen Datenstroms, speichert sie zwischen und gibt sie aber gleichzeitig jeweils einmal aus. Die Elemente werden immer wieder wiederholt, wobei jeder Durchgang durch eine anführende Punctation bestimmt wird. Falls alle Elemente des zweiten Datenstroms eingelesen wurden, beginnt die eigentliche Datenverarbeitung. Damit kein Pufferüberlauf stattfindet, wird intern eine Grenze bestimmt, die nicht überschritten werden darf.

Als erstes wird ein Element aus dem Hauptdatenstrom empfangen und die Methode `createPunctation()` aufgerufen, die eine Punctation mit Informationen über die aktuell zu verarbeiteten Daten erzeugt. Diese wird auf den ausgehenden Datenstrom gegeben, um den Anfang des aktuellen Subdatenstroms zu signalisieren. Anschließend wird das Datenobjekt des Hauptdatenstroms auf den Datenstrom gegeben, gefolgt von dem Aufruf der Methode `combineObjects()`. Dieser werden je zwei der zu kombinierenden Datenobjekte übergeben. Zum einen das aktuelle Datenobjekt des Hauptdatenstroms, zum anderen eines der zwischengespeicherten Datenobjekte, bis alle gepufferten Objekte des zweiten Datenstroms genutzt worden sind. Zurück erhält man je ein Relationenobjekt, das auf den ausgehenden Datenstrom gegeben wird. Nachdem alle Datenobjekte verarbeitet wurden, wird das aktuelle Datenobjekt des kontinuierlichen Datenstromes verworfen und ein neues empfangen.

- `createPunctation() : GenericObject`

Diese Methode erstellt eine Punctation, die verschiedene Zusicherungen bezüglich der verwendeten Datenobjekte macht. So wird zum Beispiel, nachdem das erste Datenobjekt des kontinuierlichen Datenstroms (Datenobjekt A) verarbeitet wurde, der zweite Subdatenstrom durch eine Punctation angeführt, die angibt, dass das eventuell zwischengespeicherte Element A nun verworfen werden kann, da alle folgenden Datenobjekte sich nicht mehr auf dieses beziehen. Da noch keine Punctations vom System unterstützt werden, ist die Funktionalität dieser Methode noch eingeschränkt.

- `combineObjects(GenericObject, GenericObject) : GenericObject`

Die Methode benutzt je zwei Datenobjekte; das aktuelle des kontinuierlichen und eines des

endlichen Datenstromes, und kombiniert sie in einem Relationenobjekt. In diesem werden die OIDs der einzelnen Datenobjekte eingefügt. Nachdem das Relationenobjekt angelegt wurde, wird die Methode `addAttributes()` aufgerufen, die zusätzliche Attribute dem Relationenobjekt hinzufügt, die durch die Konfiguration bestimmt werden. Es wird derzeit für Relationenobjekte das nicht optimale `MultirepresentationalRelation` – Objekt genutzt. Auch hier ist die Funktionalität dieser Methode eingeschränkt, da Relationenobjekte vom System noch nicht verwendet werden.

- `addAttributes(GenericObject, GenericObject, GenericObject) : GenericObject`

Diese Methode erhält aus den Properties Strings, die spezifizieren, welche Funktionen genutzt und deren Ergebnisse dem übergebenen Relationenobjekt als Attribute hinzugefügt werden sollen. Diese Spezifizierung lässt sich erweitern, um Attribute durch eigene Funktionen berechnen zu lassen. Standardmäßig wird nur mittels des Strings „distance“ die Distanzberechnung des Systems angeboten. Für die Berechnung werden die beiden anderen `GenericObjects` (das Haupt- und Nebendatenstromobjekt) herangezogen.

- `setProperty(String, Object) : boolean`

Diese Methode erhält eine Zeichenkette, in der, durch Kommata getrennt, verschiedene Schlüsselwörter angegeben sind, mittels denen bestimmt werden kann, welche Attribute den Relationenobjekten hinzugefügt werden sollen. Die Methode fügt in die Properties die Schlüsselwörter ein, die in der Zeichenkette vorhanden sind. Die Verarbeitung der Schlüsselwörter wird in `addAttributes()` durchgeführt.

Da das System derzeit keine, für diese Zwecke optimal geeigneten Relationenobjekte zur Verfügung stellt, werden vorerst `GenericObjects` vom Typen `MultirepresentationalRelation` verwendet.

8.2.3.4 FetchFirst (entspricht: „Fetch first X“ - Operator)

Als Parameter für die Erstellung der Operation durch die `ProcessLine` wird für die Operation die Anzahl an Elementen übergeben, die ausgegeben werden sollen. Diese Anzahl wird mittels der Methode `setProperty()` geparst und gespeichert.

Diese Klasse besitzt nur einen eingehenden und zwei ausgehende Datenströme. Der erste ausgehende Datenstrom erhält die definierte Anzahl von Objekten des eingehenden Datenstromes, die restlichen Elemente werden auf den zweiten ausgehenden Datenstrom gelegt.

Der Aufbau der Parameterzeichenkette für die Konfiguration der Operation ist Kapitel **8.2.1.4** zu entnehmen. Außer den unten definierten Methoden wird auf die Methode `isUpdateCommand()` (siehe **8.2.3**) zurückgegriffen.

Methoden:

- `process(int, Object) : boolean`

Diese Methode ist die Standardverarbeitungsmethode, die aufgerufen wird, falls ein Objekt an den

Operator geschickt wurde. Als Eingaben erhält sie die Nummer des eingehenden Datenstroms, von welchem das Objekt geschickt wurde, sowie das Objekt selbst.

Zu Beginn der Verarbeitung wird die Anzahl der weiterzuleitenden Elemente aus den Properties gelesen. Anschließend wird immer, wenn ein Element an die Operation geschickt wird, kontrolliert, ob bereits genügend Elemente auf den ersten ausgehenden Datenstrom ausgegeben wurden. Falls dem so ist, werden alle weiteren Elemente auf den zweiten ausgehenden Datenstrom gegeben.

- `setProperty(String, Object) : boolean`

Diese Methode parst und definiert die Anzahl an Elementen, die ausgegeben werden sollen. Anhand des als `PropertyKey` übergebenen Strings wird überprüft, ob es sich um die Konfiguration handelt. Ist dies der Fall, wird aus dem übergebenen Objekt die Information über die Anzahl an Elementen geholt und in den Properties gespeichert.

8.2.3.5 Sort (entspricht: Sortierungs-Operator)

Der Operation Sort werden, als Parameter bei der Erstellung, Informationen über die Sortierung, wie das zu sortierende Attribut oder die Sortierungsrichtung, übergeben. Die Informationen werden durch die Methode `setProperty()` aus den Parametern gelesen und gespeichert. Der Aufbau der Parameterzeichenkette für die Konfiguration der Operation ist Kapitel **8.2.1.5** zu entnehmen. Weiter wird in der Operation intern eine feste maximale Puffergröße von maximal 1000 Elementen festgelegt. Diese Puffergröße dient lediglich dazu, eine maximale Grenze an Elementen zu definieren, nach deren Überschreitung die Pufferung abgebrochen werden soll, um einen Pufferüberlauf vorzubeugen.

Datenobjekte, die durch den eingehenden Datenstrom erhalten werden, werden solange durch die Methode `process()` gepuffert, bis ein eingehendes Objekt erneut einer Punctation entspricht, oder bis die definierte Puffergröße erreicht wird. Falls die Puffergröße überschritten wird, wird die Verarbeitung der gepufferten Elemente abgebrochen. Falls eine weitere Punctation erhalten wird, so werden die in einem Array gepufferten Elemente der Methode `sort()` übergeben und dort, je nach Sortierungsrichtung, nach dem definierten Attribut sortiert.

Die den zu verarbeitenden Substrom anführende Punctation wird, nachdem sie ebenfalls zwischengespeichert wurde, durch die Methode `updatePunctation()` um die Sortierungsinformationen erweitert und wieder als anführendes Element, gefolgt von den sortierten Datenobjekten, auf den ausgehenden Datenstrom gegeben.

Da das Wechseln der Sortierungsrichtung oder des Sortierungsattributs während der Verarbeitung von Datenobjekten nicht effektiv ist, bzw. wenig Sinn ergibt, wird eine Neukonfiguration der Operation nicht unterstützt. Allerdings wird hier die Methode `isPunctation()` (siehe **8.2.3**) genutzt.

Diese Operation besitzt einen eingehenden und einen ausgehenden Datenstrom.

Methoden:

- `process(int, Object) : boolean`

Diese Methode ist die Standardverarbeitungsmethode, die aufgerufen wird, falls ein Objekt an den Operator geschickt wurde. Sie puffert die eingehenden Datenobjekte eines, durch eine Punctation angeführten Subdatenstromes solange, bis ein eingehendes Objekt wiederum eine Punctation ist – die des nächsten Subdatenstromes -, oder bis die definierte Puffergröße erreicht wurde. Falls die

Puffergröße überschritten wurde, wird die Verarbeitung der gepufferten Elemente abgebrochen und keine Elemente werden zurückgegeben. Falls erneut eine Punctation erhalten wurde, so werden die gepufferten Elemente durch die Methode `sort()` sortiert. Das Ergebnis ist ein sortierter Puffer, welcher anschließend versendet wird, nachdem die, durch `updatePunctation()` aktualisierte, vorangegangene Punctation als erneut anführendes Element auf den ausgehenden Datenstrom gegeben wurde. Die zuletzt erhaltene Punctation wird nun als neue, anführende Punctation für den nächsten Substrom genutzt.

- `sort(int, int) : void`

Diese Methode sortiert die gepufferten Elemente anhand des, über die Konfiguration definierten Attributes. Dabei wird der QuickSort-Algorithmus genutzt, um eine effektive Sortierung zu erhalten. Übergeben werden der Methode der jeweilige Anfang und das Ende des zu sortierenden Teil-Arrays, da der Algorithmus rekursiv genutzt wird. Die Methode greift dabei auf die Methode `partition()` zurück. Der Algorithmus wurde [11] entnommen.

- `partition(int, int) : int`

Diese Methode ist Teil des QuickSort-Algorithmus und bestimmt das dort genutzte Pivot-Element indem die Methode `check()` genutzt wird.

- `check(GenericObject, GenericObject) : boolean`

Diese Methode ist Teil des QuickSort-Algorithmus und führt die eigentlichen Vergleiche in diesem Algorithmus aus. Dabei werden die Datentypen der zu vergleichenden Attribute verglichen, und je nach Basisdatentyp die entsprechende Vergleichsmethode genutzt.

- `updatePunctation(GenericObject) : GenericObject`

Da jeder Subdatenstrom von einer Punctation angeführt wird, wird diese zwischengespeichert. Nach der Durchführung der Sortierung wird diese Methode durch `process()` aufgerufen und die zwischengespeicherte Punctation übergeben. Die Methode aktualisiert dann die Sortierungsinformationen für den entsprechenden Subdatenstrom, die in der Punctation gespeichert sind. Abschließend wird die aktualisierte Punctation wieder an die Methode `process()` zurückgegeben.

- `setProperty(String, Object) : boolean`

Diese Methode erhält Informationen über die Sortierung, beispielsweise das zu sortierende Attribut oder die Sortierrichtung, wie in **8.2.1.5** beschrieben. Anhand der Informationen wird ein `ObjectTarget` erstellt, mittels dessen die für die Sortierung notwendigen Attributsvergleiche durchgeführt werden können. Die Sortierungsrichtung, sowie das `ObjectTarget` werden anschließend in den Properties gespeichert.

Da Punctations vom System noch nicht unterstützt werden, ist diese Operation in ihrer Funktionalität eingeschränkt. Des Weiteren können die Informationen der Punctations noch nicht

mit in die Sortierung einfließen.

8.2.3.6 Group (entspricht: Gruppierungs-Operator)

Bei der Erstellung dieser Operation wird, mittels der Methode `setProperty()` und dem übergebenen Parameter, die Anzahl von Objekten übergeben, die in einem neuen Relationenobjekt gruppiert werden sollen. Dazu wird diese Anzahl von Datenobjekten gepuffert und anschließend verarbeitet. Der Aufbau der Parameterzeichenkette für die Konfiguration der Operation ist Kapitel **8.2.1.6** zu entnehmen.

Die Operation besitzt einen eingehenden und einen ausgehenden Datenstrom. Die Reihenfolge der ausgehenden Elemente ist so, wie oben unter Kapitel **7.1.6** definiert wurde. Es werden immer die eingegangenen Elemente verschickt, auf die sich kein zu erzeugendes Relationenobjekt mehr bezieht. Dabei ist zu beachten, dass sich mehrere Relationenobjekte auf Datenobjekte beziehen können. Die Datenobjekte werden sozusagen überlappend gruppiert. Eine Prüfung der eingehenden Elemente auf ein Konfigurationsupdate wird bei dieser Operation nicht durchgeführt, da eine Änderung der Anzahl der zu gruppierenden Elemente zu Problemen in der internen Objektpufferun führt. Für eine Änderung muss eine neue Operation erstellt werden.

Methoden:

- `process(int, Object) : boolean`

Diese Methode ist die Standardverarbeitungsmethode, die aufgerufen wird, falls ein Objekt an den Operator geschickt wurde. Die eingehenden Datenobjekte werden an die Methode `groupData()` weitergegeben, die die Gruppierung vornimmt. Wenn diese Verarbeitung erfolgreich durchgeführt wurde, erhält die Methode einen entsprechenden boolschen Wert zurück.

- `groupData(GenericObject) : boolean`

Diese Methode erhält als Parameter das zu gruppierende Datenobjekt. Sie prüft, ob genügend Objekte zwischengespeichert wurden. Falls dies der Fall ist, erstellt sie das Relationenobjekt mittels `createRelationalObject()` und fügt in dieses die OIDs der zu gruppierenden Objekte ein. Weiter versendet sie gegebenenfalls die nicht mehr benötigten Elemente, wie in **7.1.6** beschrieben. Als Indikator für den Erfolg der Verarbeitung des aktuellen Objektes gibt die Methode einen booleschen Wert zurück.

- `createRelationalObject(GenericObject[]) : GenericObject`

Diese Methode erstellt mit dem übergebenen `GenericObject`-Array das Relationenobjekt, das anschließend zurückgegeben wird. Derzeit liegt im System allerdings kein optimales Objekt vor, das als Relationenobjekt genutzt werden kann und es wird der `GenericObject`-Type `MultirepresentationalRelation` genutzt. Dies beschränkt jedoch die Funktionalität dieser Methode.

- `setProperty(String, Object) : boolean`

Diese Methode parst und definiert die Anzahl an Elementen, die gruppiert werden sollen. Anhand des übergebenen Strings wird überprüft, ob es sich um die Konfiguration handelt. Ist dies der Fall,

wird aus dem übergebenen Objekt diese Information geholt und in den Properties gespeichert. Die Anzahl der zu gruppierenden Elemente ist, aufgrund des zu nutzenden `GenericObject`-Typen `MultirepresentationalRelation` auf nur 2 zu beschränken.

Da das System derzeit keine, für diese Zwecke optimal geeigneten Relationenobjekte zur Verfügung stellt, werden vorerst `GenericObjects` vom Typen `MultirepresentationalRelation` verwendet. Diese bieten nur die Möglichkeit zwei Relationen zu definieren und schränkt somit die Funktionalität der Operation ein.

8.2.3.7 UDA (entspricht: UserDefinedAggregate - Operator)

Diese Operation dient gewissermaßen als Ausführungsrahmenwerk für die UDA-Phasen, beschrieben in Kapitel **6**, um einfachste Aufgaben, wie die Zählung von Elementen, zu erfüllen. Durch diese Gegebenheiten werden UDAs nicht mit einer neuen Konfiguration modifiziert, sondern müssen beendet und mit einer neuen Konfiguration gestartet werden.

Die Operation bekommt als Konfiguration die für die Phasen notwendigen Informationen übergeben. Diese Informationen werden mittels der `setProperty()` Methode verarbeitet. Dies geschieht dadurch, dass der entsprechend für die ITERATE – Phase gewählte Operator mit dem ebenfalls definierten Datentyp initialisiert und gespeichert wird. Der Aufbau der Parameterzeichenkette für die Konfiguration der Operation ist Kapitel **8.2.1.7** zu entnehmen.

Über den eingehenden Datenstrom erhaltene Datenobjekte werden dann, wie ebenfalls in Kapitel **6** beschrieben, nach der definierten Verhaltensweise verarbeitet, indem sie von der Methode `process()` an die Methode `iterate()` des jeweiligen `UDAOperators` übergeben werden. Welcher Operator genutzt werden soll, kann durch die Konfiguration definiert werden.

Zwischenergebnisse werden nach jedem verarbeiteten Datenobjekt ausgegeben, indem ein eigenes Objekt auf den ersten ausgehenden Datenstrom gegeben wird. Alle eingegangenen und verarbeiteten Datenobjekte werden auf den zweiten ausgehenden Datenstrom gelegt, um die Möglichkeit zu bieten, dass diese wiederverwendet werden können.

Methoden:

- `process(int, Object) : boolean`

Diese Methode ist die Standardverarbeitungsmethode, die aufgerufen wird, falls ein Objekt an den Operator geschickt wurde. Weiter wird nun das erhaltene Datenobjekt an die Methode `iterate()` des zu verwendenden `UDAOperators` weitergeleitet. Zurück erhält sie ein Datenobjekt, dass das aktuelle Zwischenergebnis repräsentiert. Dieses wird auf den ersten ausgehenden Datenstrom gegeben, genau wie anschließend auch das verarbeitete Objekt auf den zweiten ausgehenden Datenstrom gegeben wird.

- `setProperty(String, Object) : boolean`

Hier wird der zu verwendende `UDAOperator` initialisiert und in den Properties der Operation gespeichert. Die Informationen für den `UDAOperator` werden aus dem übergebenen Objekt geparst, falls, mittels des ebenfalls übergebenen Strings, die Parameter als Konfiguration identifiziert werden können.

8.2.3.8 Return (entspricht: Return-Operator)

Diese Operation ist vor allem dazu gedacht, Datenobjekte, deren OID als einziger Bezug zu dem gesuchten Datenobjekt in einem beliebigen Datenobjekt gespeichert ist, zu ermitteln und zurückzugeben.

Die Konfiguration dieser Operation definiert das genaue Attribut eines Datenobjektes, in welchem die OID des zu ermittelnden Objektes gespeichert ist. Diese Konfiguration wird mittels der Methode `setProperty()` in die Eigenschaften (Properties) der Operation gespeichert. Der Aufbau der Parameterzeichenkette für die Konfiguration der Operation ist Kapitel **8.2.1.8** zu entnehmen.

Die Operation besitzt zwei eingehende und einen ausgehenden Datenstrom. Der erste eingehende Datenstrom führt die Objekte, die die OIDs enthalten, deren jeweils zugehörige Objekte ermittelt werden sollen. Der zweite eingehende Datenstrom führt alle Objekte, die einem verbundenen Server bekannt sind. Die Menge der Objekte ist endlich, es werden aber die Objekte wiederholend auf den Datenstrom gegeben, wobei jeder Durchgang durch eine Punctation angeführt wird.

Die Methode `process()` wird aufgerufen, falls ein Objekt des ersten Datenstromes an den Operator geschickt wurde. Sie ruft `retrieveID()` auf, um die OID eines zu überprüfenden Objektes zu erhalten. Das Attribut, das die OID beinhalten soll, wurde in den Properties gespeichert. Anschließend wird die ermittelte OID gegen die OID der Objekte des zweiten Datenstromes verglichen. Falls ein Objekt gefunden wurde, so wird es auf den ausgehenden Datenstrom gegeben. Sonst, falls das gesuchte Objekt nicht gefunden wurde, wird das gesuchte Element selbst auf den ausgehenden Datenstrom gelegt.

Ebenso wird hier die Methode `isUpdateCommand()` (siehe **8.2.3**) genutzt.

Methoden:

- `process(int, Object) : boolean`

Falls aus dem ersten Datenstrom, dem Hauptdatenstrom, Datenobjekte eingehen, wird in der Methode `process()` überprüft, ob bereits ein Objekt gesucht wird oder nicht. Ist dies der Fall, wird die Annahme verweigert. Sonst wird die zu prüfende OID im Objekt mittels `retrieveID()` gesucht und mit dem Objekt zusammen zwischengespeichert.

Aus dem zweiten Datenstrom eingehende Datenobjekte werden auf die zwischengespeicherte OID überprüft (ebenfalls unter der Verwendung der Methode `retrieveID()`). Stimmen die OIDs überein, so wird das gefundene Objekt auf den ausgehenden Datenstrom gegeben und ein neues Datenobjekt aus dem ersten eingehenden Datenstrom kann überprüft werden. Stimmen diese nicht überein, wird dieses Objekt ignoriert. Falls keines der Objekte des zweiten Datenstroms das gesuchte ist, wird das zwischengespeicherte Datenobjekt des ersten eingehenden Datenstroms auf den ausgehenden Datenstrom gegeben.

- `retrieveID(GenericObject, ObjectTarget) : NEL`

Diese Methode erhält ein Datenobjekt. Mittels der vorher definierten NEL-Lokation innerhalb dieses Objektes wird der Identifier ermittelt und zurückgegeben. Die Lokation wird als `ObjectTarget` übergeben.

- `setProperty(String, Object) : boolean`

Diese Methode parst und definiert den Attributspfad innerhalb eines Datenobjektes, am Ende dessen ein Identifier eines Datenobjektes gespeichert sein soll, falls die Konfiguration mittels des übergebenen Strings identifiziert werden kann. Es wird hierfür mittels der Methode `getAttributeTarget()` des `StringReaders` ein `ObjectTarget` erzeugt, das für den Zweck der OID-Ermittlung genutzt wird und in den Properties gespeichert wird.

8.2.4 Hilfsklassen und Werkzeuge

Hier werden alle neuen Klassen aufgeführt, die weder Operationen, noch Operatoren sind, aber für die Funktion des System und des Operatorenkonzepts notwendig sind.

8.2.4.1 AWMLFactory und AWMLrefimplFactory

Diese Klassen aus `de.uni_stuttgart.nexus.augmentedWorld2v2.awml2`, und dem Unterpaket `de.uni_stuttgart.nexus.augmentedWorld2v2.awml2.refimpl`, wurden nur insofern verändert, dass anstatt `AWQLTargets` nun die `ObjectTargets` unterstützt werden, die sich von den `AWQLTargets` im wesentlichen nur durch die geänderten Namensgebung unterscheiden.

8.2.4.2 OperatorFactory

Diese Klasse entspricht der Klasse `AWQLFactory`. Sie wurde jedoch so umgebaut und angepasst, dass sie unabhängig von AWQL funktioniert und sich in keiner Weise mehr darauf bezieht. Es wurden vorrangig Umbenennungen von Variablen vorgenommen. Des Weiteren wurden alle `createAWQLQuery()` - Methoden entfernt, da die Klasse `AWQLQuery` nicht mehr genutzt wird. Ebenso wurden alle `NearestNeighbourOperator`-erzeugende Methoden entfernt, da diese Funktionalität nun nicht mehr durch nur einen Operator, sondern durch eine Verknüpfung von Operatoren realisiert wird (siehe Kapitel **7.2.1**).

Da sich die Klasse `AWQLFactory` des Öfteren auf die Klasse `AWQLTarget` bezieht bzw. nutzt, wurden die Beziehungen auf die neue Klasse `ObjectTarget` geändert und bereits bestehende, operatorenerzeugende Methoden überarbeitet. So zum Beispiel die Methode:

- `createSimpleCompOperator(int, ObjectTarget, Integer) : SimpleCompOperator`

Weitere Methoden wurden hinzugefügt, die einen `ComplexCompOperator`, ebenfalls nach dem Vorbild der bereits vorhandenen Methoden, erstellen können. Eine dieser Methoden ist `createComplexCompOperator()`, die, wie andere operatorerzeugende Methoden, in verschiedenen Versionen vorhanden ist. So zum Beispiel:

- `createComplexCompOperator() : ComplexCompOperator`
- `createComplexCompOperator(int, ObjectTarget, ObjectTarget, boolean) : ComplexCompOperator`

- `createComplexCompOperator(int, ObjectTarget[], ObjectTarget[], boolean) : ComplexCompOperator`

Diese Methoden verhalten sich wie die anderen bereits vorhandenen und operatorerzeugenden Methoden. (Siehe hierfür auch den Konstruktor des `ComplexCompOperators`)

8.2.4.3 ObjectTarget

Diese Klasse entspricht der Klasse `AWQLTarget` und erfüllt den gleichen Nutzen wie diese: Sie repräsentiert ein Zielattribut eines Datenobjektes, um darauf zum Beispiel Vergleiche durchzuführen. Alle Änderungen an der Klasse beziehen sich auf Abwandlungen der Namensbeziehungen zu AWQL (zum Beispiel von Variablen), sowie auf die Entfernung von Methoden (`serialize()` und `serializeQName()`), die in der Klasse `AWQLTarget` als „deprecated“ deklariert wurden.

8.2.4.4 RestrictionReader

Der `RestrictionReader` ist eine Hilfsklasse, die dazu gedacht ist, die Zeichenketten, die als Parameter vor allem für die Operation `Select` genutzt werden, zu erfassen und auszuwerten. Anhand der Informationen der Zeichenketten wird ein fertig-konfigurierter `Restriction-Operator` (oder Operatorenbaum) zurückgeliefert. Neben den standardmäßigen Basisdatentypen, werden auch geometrische und temporale Datentypen unterstützt.

Methoden:

- `readRestriction(String) : RestrictionOperator`

Die Methode `readRestriction()` des `RestrictionReaders` bekommt eine Restriktionszeichenkette, wie unter Kapitel **8.2.1.2** beschrieben, als Eingabe. Diese Zeichenkette wird an jedem Semikolon (';') aufgetrennt. Die so entstandenen String-Teile werden daraufhin alle an `parsePartRestriction()` übergeben, um aus diesen den entsprechenden `Restriction Operator`, beziehungsweise Operatorenbaum, zu generieren.

- `parsePartRestriction(String[], int) : RestrictionOperator`

Diese Methode wiederum parst die einzelnen Restriktionen rekursiv für den Fall, dass verschachtelte Restriktionen vorliegen. Falls nur noch die einzelnen Beschreibungen der Operatoren vorliegen, werden diese an `parseOperator()` weitergeleitet, um dort das eigentliche Parsen der Operatorkonfigurationen durchzuführen.

- `parseOperator(String) : RestrictionOperator`

Diese Methode benutzt die Methode `parseSide()`, um die jeweiligen Seiten der Beschreibung eines Operators zu parsen. Diese wurden vorher durch die Auftrennung des übergebenen Strings an den Kommas (',') erhalten. Sind alle Informationen, einen Operator betreffend, ausgelesen worden, so werden sie mittels `createRestrictionOperator()` genutzt, um einen `RestrictionOperator` zu erstellen.

- `parseSide(String) : String[]`

Diese Methode bekommt als Eingabe eine der beiden Seiten des Zeichenketten-Operators. Diese wird zuerst mittels des Punkt-Symbols ('.') aufgetrennt und anschließend zurückgegeben. Der erste der mindestens 3 erhaltenen Teile bildet bereits den Descriptor, der zweite Teil repräsentiert eine AttributeInstance oder den Datentyp einer Konstanten. Die folgenden Teile stellen den Wert einer Konstanten oder den Pfadnamen eines AttributeParts dar.

- `createRestrictionOperator(String[], String, String[]) : RestrictionOperator`

Diese Methode liest alle Informationen aus den übergebenen Strings und bestimmt anhand dieser Informationen, die zur Definition eines solchen Operators notwendig sind, die Art des gewünschten `RestrictionOperators`. Die dafür notwendigen `ObjectTargets` bzw. `GenericAttribute-Parts` werden mittels den Methoden `getReferenceTarget()`, `getAttributeTarget()` etc. der Klasse `StringReader` erzeugt. Zur Erzeugung des `RestrictionOperators` greift `createRestrictionOperator()` auf die `OperatorFactory` und deren Methoden zur Erstellung von Operatoren zurück.

8.2.4.5 StringReader

Diese Klasse stellt vor allem für die Klasse `RestrictionReader` Methoden zur Verfügung, die für die Informationsextraktion von Parametern notwendig sind. Allerdings nutzen auch andere Klassen diese Methoden, um aus einem String, der Pfade in Attributen definiert, zum Beispiel ein `GenericAttributePart` zu generieren.

Methoden:

- `getReferenceTargets(String[]) : ObjectTarget[]`
- `getAttributeTarget(String[]) : ObjectTarget`
- `getConstant(String[], ObjectTarget) : GenericAttributePart`
- `getConstantGeo(String[], ObjectTarget) : GenericAttributePart`
- `getConstantTemp(String[], ObjectTarget) : GenericAttributePart`

Diese Methoden erhalten die Informationen einer der Seiten eines Komparators einer Restriktion, um `ObjectTargets` oder `GenericAttributeParts` zu erstellen und diese zurückzugeben. Die Methoden werden je nach Descriptor aufgerufen, das heißt, dass eine Seite eines Komparators, die mit „`attribute`“ beginnt, die Methode `getAttributeTarget()` nutzt.

Die Methode `getConstant()` erhält, neben diesen Informationen, ein `ObjectTarget`, das das zu vergleichende Attribut der anderen Komparatorseite repräsentiert. Anhand dieses `ObjectTargets` wird der Datentyp der Konstanten ermittelt.

Die Methode `getConstantGeo()` erhält ebenfalls die Informationen der einen Seite des Komparators, sowie ein `ObjectTarget`, das das zu vergleichende Attribut der anderen Komparatorseite repräsentiert. Über das String-Array wird die, für die Erzeugung des

GenericAttributeParts notwendige CGeometry über den *srs-Code* und die Angaben über die Dimensionen ermittelt. Für die Erzeugung der CGeometry wird die Klasse WKTReader genutzt.

Die Methode getConstantTemp() verfährt ähnlich wie die Methode getConstantGeo(). Sie erhält Informationen zu einer Seite eines Komparators, sowie ein ObjectTarget. Dabei werden allerdings alle Informationen zur Erzeugung eines CTime-Objektes ermittelt. Anschließend wird auch hier ein GenericAttributePart erzeugt und zurückgegeben.

Die Methode getReferenceTargets() liefert ein ObjectTarget-Array zurück, wobei das erste Element den Ort der zu nutzenden OID repräsentiert, und das zweite das eigentliche, zu vergleichende AttributePart. Dabei nutzt die Methode getAttributeTarget(), um die ObjectTargets des Arrays zu erzeugen.

Die Methode getAttributeTarget() nutzt das übergebene String-Array, um mittels der Informationen ein ObjectTarget zu erstellen, welches dann zurückgegeben wird.

8.2.4.6 GenericObjectSink

Diese Klasse entspricht GenericObjectSink in de.uni_stuttgart.nexus.streamFederation.sinks.extended.vispipe.genericObjectSink. Sie wurde vor allem für Testzwecke mit erweiterten Ausgaben ergänzt. Sonst entspricht sie ihrer Vorbild-Klasse. Sie wurde in das Projekt dieser Arbeit eingefügt, damit gewährleistet werden kann, dass alle in dieser Arbeit verwendeten Klassen, die nicht zum eigentlichen, grundlegenden System gehören, zusammengefasst werden können. Das Verhalten diese Klasse beschränkt sich darauf, Objekte der Klasse GenericObject entgegen zu nehmen und in der Konsole auszugeben.

8.2.4.7 GenericObjectSource

Diese Klasse entspricht zum Großteil der Klasse ResultSetSource in de.uni_stuttgart.nexus.streamFederation.sources.extended.vispipe.resultSetSource. Sie wurde nur für Testzwecke unter anderem mit weiteren Ausgaben ergänzt. Des Weiteren liest sie, ebenfalls wie ResultSetSource, über eine Zeichenkette in den Properties, eine AWML-Datei als ResultSet ein. Im Gegensatz zu ihrer Vorbild-Klasse wird aber nicht dieses ResultSet direkt ausgegeben, sondern über einen Iterator in GenericObject-Objekte aufgetrennt und anschließend ausgegeben. Auch sie wurde in das Projekt dieser Arbeit eingefügt, damit gewährleistet werden kann, dass alle in dieser Arbeit verwendeten Klassen, die nicht zum eigentlichen, grundlegenden System gehören, zusammengefasst werden können.

8.3 Realisierung der Beispielanfrage „Nächster Nachbar“

Die unter Kapitel **7.2** beschriebenen Beispielanfragen können mit den vorhandenen Operationen nun auf die folgende Weise beschrieben werden. Hierzu werden die Befehlseinheiten, die Commands, die für die Erstellung der ProcessLines zuständig sind, also die SetupCommands, genauer beschrieben.

Die Kombination der Operatoren wurde unter **7.2.1** wie folgt dargestellt:

$$\alpha = \mathrm{Return}_{\mathrm{Obj1.OID}} \left(\mathrm{Fetch}_{1} \left(\mathrm{Sort}_{\mathrm{Dist}\uparrow} (x)\right)\right)$$

$$x = \sigma_{\mathrm{Type=Rel}} \left(x_{[\mathrm{Dist}(\mathrm{Obj1.pos} , \mathrm{Obj2.pos})]} \left(\sigma_{\mathrm{pos...,\ type=...}} (\varepsilon), (c)\right)\right)$$

Im SetupCommand dieser Anfrage müssen, wie sonst auch, zuerst je eine Datenquelle für die eingehenden Datenströme ε und c, und eine Datensenke für die die Anfrage durchgeführt wird, hier α, erstellt werden. Für jede Operation ist ebenfalls ein Eintrag notwendig, sowie für die Verknüpfung der einzelnen Komponenten. Falls die Anfrage über mehrere Knoten ablaufen soll, muss für jede Übermittlung zwischen Knoten jeweils eine Senke und eine Quelle angegeben werden. Dies ist jedoch hier nicht der Fall.

Die Einträge teilen sich hierbei in drei Hauptbereiche: Zum ersten in die Definition der so genannten Blöcke, die Datenquellen, -senken und die verarbeitenden Operationen repräsentieren; zum zweiten in die Definition der Verknüpfungen zwischen den Blöcken; und zum Dritten in die Definition der Konfigurationen der einzelnen Operationen.

Die SetupCommand-Einträge der obigen Beispielanfrage sind Anhang A zu entnehmen. Hierbei wurden die Blöcke und ihre jeweiligen Angaben der Konfigurationen zusammen gefasst. Die Positionen der einzelnen Teile innerhalb des Commands können dabei beliebig gewählt werden. Die Zusammenfassung wurde hier nur zur besseren Lesbarkeit vollzogen.

8.4 Performanz-Test

Um die Leistungsfähigkeit für zukünftige Anwendungen mit dem erarbeiteten Operatorenkonzept zu testen und zu analysieren, wurden die folgenden Performanz-Tests hier durchgeführt.

Es wurde mehrmals die Gesamtdauer von jeweils gleichen Selektionen über einer bestimmten Anzahl von gleichbleibenden Datenobjekten gemessen, um eine durchschnittliche Gesamtverarbeitungsdauer zu ermitteln.

Wie bereits in den Annahmen unter Kapitel **8.1** beschrieben wurde, wurde der Test nur auf einem lokalen `Streamnode` durchgeführt. Das Testsystem, auf dem die Tests durchgeführt wurden, bestand aus einem `Intel Core 2 Duo` mit `2,4Ghz` mit `2 x 2MB` CPU-Cache, sowie `4 GB` Hauptarbeitsspeicher. Das benutzte Betriebssystem war `Windows Vista Home Premium 64-bit` mit der Java Virtual Machine des `JRE 1.6.0_07`. Des Weiteren nutzt das Testsystem einen Festplatten-Raid-Verbund.

Damit die Tests durchgeführt werden konnten, musste zuerst die Klasse `ProcessLine` in `de.uni_stuttgart.nexus.streamFederation.sandbox` so angepasst werden, dass die Methode `deserialize()` auch Strings einlesen kann. Diese Anpassung war aber nur von geringem Ausmaß.

Für die Tests wurde weiter ein `SetupCommand` erstellt, das eine einfache Anfrage an einen Datenstrom darstellte. Es wurde eine Datenquelle, eine Datensenke und eine Selektions-Operation erstellt. Als Datenquelle wurde die Klasse `GenericObjectSource` in `de.uni_stuttgart.nexus.streamFederation.operators.basic.source` genutzt, die einen Datenstrom aus einer einfachen AWML-Datei erzeugt. Dazu werden die in dieser Datei gespeicherten Datenobjekte ausgelesen und als `GenericObjects` über den Datenstrom versendet. Somit konnte sichergestellt werden, dass die Datenobjekte von gleichbleibender Qualität waren und somit Schwankungen bei unterschiedlichen Messungen hier ausgeschlossen werden konnten.

Die genutzte Datensenke des `SetupCommands` wurde dabei von einer einfachen Senke repräsentiert, die die entsprechenden Ausgaben von Datenobjekten vornahm. Genutzt wurde hierfür die Klasse `GenericObjectSink` in `de.uni_stuttgart.nexus.streamFederation.operators.basic.sink`. Diese Klasse gibt die einzelnen Datenobjekte aus, die sie erhält.

Die Selektion des Performanz-Tests stellt sich dabei so dar, dass hierfür die Operation `SelectTest` genutzt wurde. Diese Klasse entspricht der Operation `Select` (siehe **8.2.3.2**), die für Testzwecke um einige Ausgaben und Zeitmessungen erweitert wurde. Sie ist zu finden in

`de.uni_stuttgart.nexus.streamFederation.operators.basic.operations.selectTest`. Die Konfiguration der Operation ist dabei so definiert, dass ein einfaches Attribut der jeweiligen Datenobjekte mit einer Konstanten verglichen wird. Die Konfiguration ist dem folgenden Ausschnitt des genutzten `SetupCommands` zu entnehmen:

```
<eas:parameter xmlns:eas="...">
<nsas:value>
     <eas:blockId xmlns:eas="...">
          op1
     </eas:blockId>
     <eas:parameterKey xmlns:eas="...">
          restriction
     </eas:parameterKey>
     <eas:parameterValue xmlns:eas="...">
       attribute.nsas:type.nsas:value, equal, constant.nsat:type.nscs:Building
     </eas:parameterValue>
     <eas:parameterClass xmlns:eas="...">
          urn:java:java.lang.String
     </eas:parameterClass>
</nsas:value>
</eas:parameter>
```

Die Messungen selbst wurden so realisiert, dass die Zeit ab Erhalt des ersten Datenobjektes bis nach dem Weiterversenden des letzten Datenobjektes, also die gesamte Verarbeitungsdauer für alle Datenobjekte, gemessen wurde. Es wurde hier die durchschnittliche Gesamtverarbeitungsdauer erfasst, da die einzelnen Verarbeitungszeitspannen für jeweilige Messungen zu kurz waren, um Aussagen über diese zu treffen.

Getestet wurde mittels 20 verschiedenen Datenobjekten, von denen 12 die obige Bedingung erfüllten. Diese 20 Objekte wurden erneut in die zu lesende AWML-Datei eingefügt (kopiert), bis 500 Elemente pro Durchlauf geschickt werden konnten. Auf 300 von diesen 500 Datenobjekte traf nun die oben definierte Restriktion zu. Insgesamt wurden 10 Durchläufe durchgeführt. In allen 10 Durchläufen wurden alle 300 Datenobjekte korrekt selektiert und ausgegeben.

Aus der Abbildung 8.1 lässt sich die, zu jedem Durchlauf zugehörige Gesamtverarbeitungszeit entnehmen. Somit ergibt sich für diese 500 Objekte im Durchschnitt eine Gesamtverarbeitungsdauer von 113,5 ms (rote Linie in der Abbildung unten). Wie in der unten stehenden Abbildung auch zu erkennen ist, liegt in den Testergebnissen ein stärker abweichendes Ergebnis vor. Dieser „Ausreißer“ war auf nebenläufige Aktivitäten des Testrechners zurückzuführen. Bei diesen Verarbeitungszeiten ist zu beachten, dass sie auf dem weiter oben beschriebenen, genutzten Testrechner durchgeführt wurden, und die Berechnungszeiten für andere Rechner von diesen Werten abweichen können.

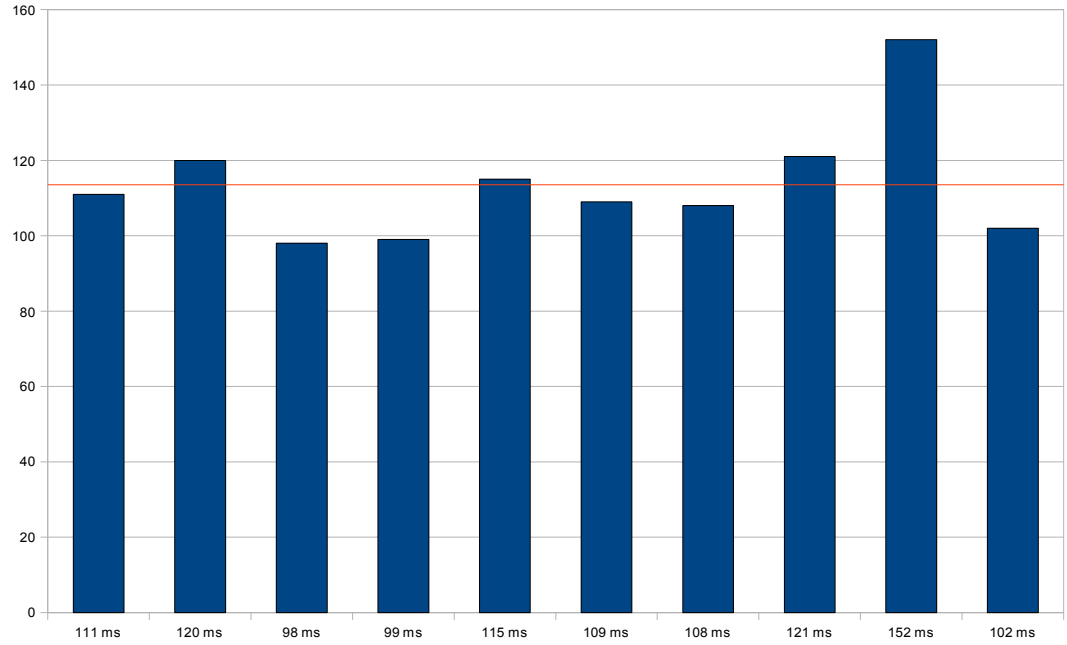

Abbildung 8.1: Gesamtzeitspannen der Verarbeitungen

Es ist anzunehmen, dass sich für komplexere Anfragen diese Zeitspannen vergrößern.

9. Ausblick und zukünftige Entwicklungsmöglichkeiten

In dieser Arbeit wurde ein Konzept erarbeitet, mittels dessen die Problemstellung von Basisoperatoren für kontinuierliche Datenströme gelöst werden kann. Hierfür wurden zuerst verschiedene Möglichkeiten, diese Basisoperatoren für Datenströme zu definieren, vorgestellt und analysiert. Aus diesen verschiedenen Möglichkeiten wurden die ausgewählt, die am besten zum vorhandenen NEXUS-System passen. Hierbei wurden die Vorarbeiten, beschrieben in [10], mit in die Abwägungen einbezogen. Neben diesen Vorarbeiten, waren auch die Multidatentypen des Systems die ausschlaggebenden Faktoren für die Auswahl des Lösungskonzepts. Als Lösungskonzept wurden schließlich die Punctations zusammen mit den User-Defined-Aggregates ausgewählt, um die unter Kapitel **7.1** beschriebenen Operatoren und ihr Verhalten zu ermöglichen. Anschließend wurden diese Operatoren entworfen und implementiert. Diese erarbeiteten Operatoren bieten einen Grundstock an Funktionen, die vom derzeitigen System unterstützt werden. Mit der Weiterentwicklung des NEXUS-Systems sollten natürlich auch diese Komponenten weiterentwickelt und erweitert werden.

Im Folgenden sollen Möglichkeiten dargelegt werden, in welchen Bereichen des hier erstellten Operatorenkonzeptes noch offene Fragen vorliegen oder Möglichkeiten für Weiterentwicklungen bestehen.

9.1 Befehlseinheiten (Commands) und die Command-Sprache

Eine der zukünftigen Verbesserungsmöglichkeiten des Systems ist die Erweiterung der Möglichkeiten für die Parametrisierung der Operationen in der Command-Sprache. Da derzeit nur kommaseparierte Zeichenketten als Key-Value-Paare übergeben werden können, die später aufwendig geparst werden müssen um das Verhalten genau zu spezifizieren, bestünde hier die Möglichkeit, die erweiterte Konstrukte für die Commands zu definieren, mittels denen die

derzeitige Lösung verbessert und vereinfacht werden könnte.

So sollte durch die neuen Konstrukte bei der Konfiguration für die `Select` – Operation der zu benutzende Operator, sowie die beteiligten Attribute, definiert werden können, ebenso wie die Möglichkeit, einen entsprechenden Operatorenbaum darzustellen. Des Weiteren kann die Unterstützung von Datentypen weiter ausgebaut werden.

Für die `CartesianProduct` – Operation sollte ebenso die Möglichkeit gegeben werden, die Methoden zur Berechnung der hinzuzufügenden Attribute, auf eine einfachere und bessere Weise angeben zu können.

Die Konfiguration für die `Sort` – Operation sollte so definierbar sein, dass unter anderem die Sortierungsrichtung sowie das zu sortierende Attribut in einer komfortableren und wohldefinierten Form angegeben werden können.

Auch besteht die Möglichkeit, erweiterte und angepasste Strukturen der Commands für UDA-Konstrukte zu entwickeln. Hierbei sollte es möglich sein, Informationen für die INITIATE- und ITERATE-Phasen getrennt und wohldefiniert angeben zu können.

Für die Operationen `FetchFirst`, `Assemble`, `Group` und `Return`, die zwar nur jeweils einen Wert für die Konfiguration benutzen, sollten diese Werte, anstatt eines einfachen Key-Value – Paars, aussagekräftigere Tags besitzen.

9.2 Komponenten des Operatorenkonzepts

Da an dem NEXUS-System derzeit größere Änderungen zur Anpassung an die Streamfähigkeit durchgeführt werden, besteht die Möglichkeit, dass Überarbeitungen bzgl. der kommenden Systemänderungen an den, in dieser Arbeit entwickelten Komponenten notwendig werden. So müsste zum Beispiel bei einer Änderung der Komponente `PossibleObjectExcerptList` auch ein großer Teil der Operatoren unter Kapitel **8.2.2** überarbeitet werden. Das Operatorenkonzept sollte weitestgehend kompatibel zu dem Vorhaben der Virtualisierung des Systems sein.

Weitere Erweiterungsmöglichkeiten der Komponenten des Operatorenkonzepts beziehen sich auf Erweiterungen im Bereich der unterstützten Datentypen und Vergleichsoperatoren. Es wurden die notwendigsten Vergleichsoperatoren und Datentypen unterstützt. Falls zu einem späteren Zeitpunkt weitere erforderlich sein sollten, so könnten diese einfach in der jeweiligen Klasse eingefügt werden, indem die vorhandenen Methoden und Konstrukte als Vorbild genommen werden.

In der Klasse `ComplexCompOperator` sollen Datenobjekte per OID von einem Server abgefragt werden. Diese Anfrage geschah bisher nur mit AWQL. Eine stream-basierte Anfrage (z.B. mittels der Return-Operation) ist nicht praktikabel, da nicht von vorne herein bei einer Selektion klar ist, ob ein `ComplexCompOperator` genutzt wird. Falls ein solcher genutzt wird, müsste zur Laufzeit eine komplette, eigenständige `ProcessLine` hierfür erstellt werden. Eine direkte Anfrage an den Server über eine möglicherweise noch zu erstellende Schnittstelle, losgelöst von AWQL, ist hier effektiver, aber noch nicht im System verfügbar.

Die `UDAOperatoren` geben für jedes verarbeitete Datenobjekt ein Zwischenresultat zurück. Diese sollen anschließend über die ausgehenden Datenströme einer UDA-Operation weitergeleitet

werden. Hierfür müssten die Zwischenergebnisse in Datenobjekte verpackt werden. Solche Datenobjekte sind allerdings derzeit von der Typisierung her, d.h. die Typen der jeweiligen `GenericObjects` betreffend, nicht vorgesehen. So könnten die Objekttypen um diese einfachen Resultatobjekte erweitert werden.

Derzeit werden noch keine Relationenobjekte, wie sie hier in dieser Arbeit beschrieben werden, vom System zur Verfügung gestellt. Es werden stattdessen auf die nicht optimalen `GenericObjects` vom Typen `MultirepresentationalRelation` zurückgegriffen. Um die Funktionalitäten der Operationen, die auf Relationenobjekte zurückgreifen, voll nutzen zu können, sollten optimale Relationenobjekte in das System eingefügt werden.

Gleiches gilt für Punctations. Hier kann allerdings nicht temporär auf eine Ausweichklasse zurück gegriffen werden. Auch hier muss eine entsprechende Komponente in das System eingefügt werden, um die entsprechenden Funktionalitäten der punctation-nutzenden Operationen zu gewährleisten. Das Einfügen von Relationenobjekten und Punctations ist, nach Absprache, nicht Gegenstand dieser Arbeit, da diese Konstrukte später möglichst dauerhaft genutzt werden sollen.

9.3 Cross-StreamNode - Kommunikation

Alle entwickelten Komponenten wurden entsprechend den Angaben in [10] entwickelt. Auch wenn eine Kommunikation dieser Komponenten über mehrere StreamNodes theoretisch funktionieren müsste, wurde dies, wie in den Annahmen vermerkt, nie getestet oder überprüft. Es sollte also die Funktionstüchtigkeit dieser Art der Kommunikation sichergestellt werden.

9.4 Stromeigenschaften

Wie bereits in den Annahmen erwähnt wurde, können die in dieser Arbeit definierten Stromeigenschaften nicht genutzt werden, da die Kommunikationsstruktur des Systems diese derzeit nicht unterstützt. Es könnten die Stromeigenschaften realisiert und in das bestehende Operatorenkonzept integriert werden. Dies bedeutet auch die Anpassung der Operationen, die sich auf die Stromeigenschaften beziehen. Des Weiteren müsste in Zusammenhang mit den Stromeigenschaften eine überwachende Instanz realisiert werden, die die Verbindungen zwischen den Operationen bezüglich der Stromeigenschaften prüft, herstellt und reguliert.

9.5 Ersetzung der `PossibleObjectExcerptList`

Die im System vor allem durch AWQL genutzte `PossibleObjectExcerptList` wird bei der Implementierung weiter genutzt, da eine Überarbeitung dessen nicht im Rahmen dieser Arbeit liegt. Es bietet sich also die Möglichkeit, dieses Konstrukt im Rahmen einer eventuellen Loslösung von AWQL, neu zu überarbeiten und zu ersetzen, da dieses Konstrukt nur für einen temporären Einsatz konzipiert war.

10. Anhänge

10.1 Anhang A

In diesem Anhang ist die komplette SetupCommand-Datei für die Beispielanfrage „Nächster Nachbar“, bezüglich ihrer Blöcke, Parameter und Verbindungen, dokumentiert. Sie ist in 2 Bereiche gegliedert. Zum einen in die Definition der Blöcke und ihrer jeweiligen Konfigurationen, zum anderen in die Verbindungen zwischen den einzelnen Blöcken.

10.1.1 Definition der Blöcke und Konfigurationen

Command – Eintrag der Datenquelle von ε

```
<eas:block xmlns:eas="...">
   <nsas:value>
      <eas:type xmlns:eas="...">
         source
      </eas:type>
      <eas:classURI xmlns:eas="...">
         urn:java:de...  (Angabe des Pfades zu einer Datenquelle für kontinuierliche Datenströme)
      </eas:classURI>
      <eas:id xmlns:eas="...">
         datastreamSource
      </eas:id>
   </nsas:value>
</eas:block>
```

Command – Eintrag für die Datenquelle von c

```
<eas:block xmlns:eas="...">
   <nsas:value>
      <eas:type xmlns:eas="...">
         source
      </eas:type>
      <eas:classURI xmlns:eas="...">
          urn:java:de... (Angabe des Pfades zu einer Datenquelle, die eine Konstante liefert)
      </eas:classURI>
      <eas:id xmlns:eas="...">
         constantSource
      </eas:id>
   </nsas:value>
</eas:block>
```

Command – Eintrag für die Datensenke von α

```
<eas:block xmlns:eas="...">
   <nsas:value>
      <eas:type xmlns:eas="...">
         sink
      </eas:type>
      <eas:classURI xmlns:eas="...">
         urn:java:de...   (Pfad zur zu benutzenden Datensenke)
```

```
      </eas:classURI>
      <eas:id xmlns:eas="...">
         mainSink
      </eas:id>
   </nsas:value>
</eas:block>
```

Command – Einträge für die erste Select – Operation

```
<eas:block xmlns:eas="...">
   <nsas:value>
      <eas:type xmlns:eas="...">
         operation
      </eas:type>
      <eas:classURI xmlns:eas="...">
         urn:java:de.uni_stuttgart.nexus.streamFederation.operators.basic.
          operations.select.Select
      </eas:classURI>
      <eas:outputSlotId xmlns:eas="...">
         0
      </eas:outputSlotId>
      <eas:outputSlotId xmlns:eas="...">
         1
      </eas:outputSlotId>
      <eas:inputSlotId xmlns:eas="...">
         0
      </eas:inputSlotId>
      <eas:id xmlns:eas="...">
         select1
      </eas:id>
   </nsas:value>
</eas:block>
```

sowie

```
<eas:parameter xmlns:eas="...">
<nsas:value>
   <eas:blockId xmlns:eas="...">
      select1
   </eas:blockId>
   <eas:parameterKey xmlns:eas="...">
      restriction
   </eas:parameterKey>
   <eas:parameterValue xmlns:eas="...">
      attribute.nsas:type.nsas:value, equals, constant.nsat:type.nsas:building;
      and; attribute.nsas:pos.nsas:value, within, constant.nsat:wkt.'srs-
      code':Polygon((...))
```

(Siehe hierfür auch Kapitel **8.2.1.2**)

```
   </eas:parameterValue>
   <eas:parameterClass xmlns:eas="...">
      urn:java:java.lang.String
   </eas:parameterClass>
</nsas:value>
</eas:parameter>
```

Command – Einträge für die Kreuzprodukt – Operation

```
<eas:block xmlns:eas="...">
   <nsas:value>
      <eas:type xmlns:eas="...">
```

```
         operation
      </eas:type>
      <eas:classURI xmlns:eas="...">
          urn:java:de.uni_stuttgart.nexus.streamFederation.operators.basic.
         operations.cartesianProduct.CartesianProduct
      </eas:classURI>
      <eas:outputSlotId xmlns:eas="...">
         0
      </eas:outputSlotId>
      <eas:inputSlotId xmlns:eas="...">
         0
      </eas:inputSlotId>
      <eas:inputSlotId xmlns:eas="...">
         1
      </eas:inputSlotId>
      <eas:id xmlns:eas="...">
         cartesianProduct
      </eas:id>
   </nsas:value>
</eas:block>
```

sowie

```
<eas:parameter xmlns:eas="...">
<nsas:value>
   <eas:blockId xmlns:eas="...">
      cartesianProduct
   </eas:blockId>
   <eas:parameterKey xmlns:eas="...">
      cartesianProduct
   </eas:parameterKey>
   <eas:parameterValue xmlns:eas="...">
      (Angabe zu den zu berechnenden Attribute wie „distance“)
   </eas:parameterValue>
   <eas:parameterClass xmlns:eas="...">
      urn:java:java.lang.String
   </eas:parameterClass>
</nsas:value>
</eas:parameter>
```

Command – Einträge für die zweite Select – Operation

```
<eas:block xmlns:eas="...">
   <nsas:value>
      <eas:type xmlns:eas="...">
         operation
      </eas:type>
      <eas:classURI xmlns:eas="...">
         urn:java:de.uni_stuttgart.nexus.streamFederation.operators.basic.
          operations.select.Select
      </eas:classURI>
      <eas:outputSlotId xmlns:eas="...">
         0
      </eas:outputSlotId>
      <eas:outputSlotId xmlns:eas="...">
         1
      </eas:outputSlotId>
      <eas:inputSlotId xmlns:eas="...">
         0
```

```
      </eas:inputSlotId>
      <eas:id xmlns:eas="...">
         select2
      </eas:id>
   </nsas:value>
</eas:block>
```

sowie

```
<eas:parameter xmlns:eas="...">
<nsas:value>
   <eas:blockId xmlns:eas="...">
      select2
   </eas:blockId>
   <eas:parameterKey xmlns:eas="...">
      restriction
   </eas:parameterKey>
   <eas:parameterValue xmlns:eas="...">
      attribute.nsas:type.nsas:value, equals, constant.nsat:type....
```

(Angabe des Relationenobjekt-Typs, der allerdings im System noch nicht vorhanden ist...)

```
   </eas:parameterValue>
   <eas:parameterClass xmlns:eas="...">
      urn:java:java.lang.String
   </eas:parameterClass>
</nsas:value>
</eas:parameter>
```

Command – Einträge für die Sortierungs – Operation

```
<eas:block xmlns:eas="...">
   <nsas:value>
      <eas:type xmlns:eas="...">
         operation
      </eas:type>
      <eas:classURI xmlns:eas="...">
         urn:java:de.uni_stuttgart.nexus.streamFederation.operators.basic.
         operations.sort.Sort
      </eas:classURI>
      <eas:outputSlotId xmlns:eas="...">
         0
      </eas:outputSlotId>
      <eas:inputSlotId xmlns:eas="...">
         0
      </eas:inputSlotId>
      <eas:id xmlns:eas="...">
         sort
      </eas:id>
   </nsas:value>
</eas:block>
```

sowie

```
<eas:parameter xmlns:eas="...">
<nsas:value>
   <eas:blockId xmlns:eas="...">
      sort
   </eas:blockId>
   <eas:parameterKey xmlns:eas="...">
      sort
   </eas:parameterKey>
```

```
   <eas:parameterValue xmlns:eas="...">
      (Angabe der Sortierungseigenschaften gemäß Kapitel 8.2.1.5,
           hier z.B. in der Art von: up, nsas:distance.nsas:value)
   </eas:parameterValue>
   <eas:parameterClass xmlns:eas="...">
      urn:java:java.lang.String
   </eas:parameterClass>
</nsas:value>
</eas:parameter>
```

Command – Einträge für die FetchFirst – Operation

```
<eas:block xmlns:eas="...">
   <nsas:value>
      <eas:type xmlns:eas="...">
         operation
      </eas:type>
      <eas:classURI xmlns:eas="...">
         urn:java:de.uni_stuttgart.nexus.streamFederation.operators.basic.
          operations.fetch.FetchFirst
      </eas:classURI>
      <eas:outputSlotId xmlns:eas="...">
         0
      </eas:outputSlotId>
      <eas:inputSlotId xmlns:eas="...">
         0
      </eas:inputSlotId>
      <eas:id xmlns:eas="...">
         fetch
      </eas:id>
   </nsas:value>
</eas:block>
```

sowie

```
<eas:parameter xmlns:eas="...">
<nsas:value>
   <eas:blockId xmlns:eas="...">
      fetch
   </eas:blockId>
   <eas:parameterKey xmlns:eas="...">
      fetch
   </eas:parameterKey>
   <eas:parameterValue xmlns:eas="...">
      (Anzahl der zurückzugebenden Elemente, hier zum Beispiel „1“)
   </eas:parameterValue>
   <eas:parameterClass xmlns:eas="...">
      urn:java:java.lang.Integer
   </eas:parameterClass>
</nsas:value>
</eas:parameter>
```

Command – Einträge für die Return – Operation

```
<eas:block xmlns:eas="...">
   <nsas:value>
      <eas:type xmlns:eas="...">
         operation
```

```
      </eas:type>
      <eas:classURI xmlns:eas="...">
         urn:java:de.uni_stuttgart.nexus.streamFederation.operators.basic.
          operations.returnOp.Return
      </eas:classURI>
      <eas:outputSlotId xmlns:eas="...">
         0
      </eas:outputSlotId>
      <eas:inputSlotId xmlns:eas="...">
         0
      </eas:inputSlotId>
      <eas:id xmlns:eas="...">
         return
      </eas:id>
   </nsas:value>
</eas:block>
```

sowie

```
<eas:parameter xmlns:eas="...">
<nsas:value>
   <eas:blockId xmlns:eas="...">
      return
   </eas:blockId>
   <eas:parameterKey xmlns:eas="...">
      return
   </eas:parameterKey>
   <eas:parameterValue xmlns:eas="...">
      (Beschreibung des zu nutzenden Attributs z.B. „nsas:obj1.nsas:value“)
   </eas:parameterValue>
   <eas:parameterClass xmlns:eas="...">
      urn:java:java.lang.Integer
   </eas:parameterClass>
</nsas:value>
</eas:parameter>
```

10.1.2 Definition der Verknüpfungen

Verknüpfung von der Datenquelle von ε zur ersten Select – Operation

```
<eas:link xmlns:eas="...">
   <nsas:value>
      <eas:inputBlockId xmlns:eas="...">
         datastreamSource
      </eas:inputBlockId>
      <eas:inputBlockOutputSlotId xmlns:eas="...">
         0
      </eas:inputBlockOutputSlotId>
      <eas:outputBlockId xmlns:eas="...">
         select1
      </eas:outputBlockId>
      <eas:outputBlockInputSlotId xmlns:eas="...">
         0
      </eas:outputBlockInputSlotId>
   </nsas:value>
</eas:link>
```

Verknüpfung von der ersten Select – Operation zur Kreuzprodukt – Operation

```
<eas:link xmlns:eas="...">
   <nsas:value>
      <eas:inputBlockId xmlns:eas="...">
         select1
      </eas:inputBlockId>
      <eas:inputBlockOutputSlotId xmlns:eas="...">
         0
      </eas:inputBlockOutputSlotId>
      <eas:outputBlockId xmlns:eas="...">
         cartesianProduct
      </eas:outputBlockId>
      <eas:outputBlockInputSlotId xmlns:eas="...">
         0
      </eas:outputBlockInputSlotId>
   </nsas:value>
</eas:link>
```

Verknüpfung von der Datenquelle von c zur Kreuzprodukt – Operation

```
<eas:link xmlns:eas="...">
   <nsas:value>
      <eas:inputBlockId xmlns:eas="...">
         constantSource
      </eas:inputBlockId>
      <eas:inputBlockOutputSlotId xmlns:eas="...">
         0
      </eas:inputBlockOutputSlotId>
      <eas:outputBlockId xmlns:eas="...">
         cartesianProduct
      </eas:outputBlockId>
      <eas:outputBlockInputSlotId xmlns:eas="...">
         1
      </eas:outputBlockInputSlotId>
   </nsas:value>
</eas:link>
```

Verknüpfung von der Kreuzprodukt – Operation zur zweiten Select – Operation

```
<eas:link xmlns:eas="...">
   <nsas:value>
      <eas:inputBlockId xmlns:eas="...">
         cartesianProduct
      </eas:inputBlockId>
      <eas:inputBlockOutputSlotId xmlns:eas="...">
         0
      </eas:inputBlockOutputSlotId>
      <eas:outputBlockId xmlns:eas="...">
         select2
      </eas:outputBlockId>
      <eas:outputBlockInputSlotId xmlns:eas="...">
         0
      </eas:outputBlockInputSlotId>
   </nsas:value>
</eas:link>
```

Verknüpfung von der zweiten Select – Operation zur Sortierungs – Operation

```
<eas:link xmlns:eas="...">
   <nsas:value>
      <eas:inputBlockId xmlns:eas="...">
         select2
      </eas:inputBlockId>
      <eas:inputBlockOutputSlotId xmlns:eas="...">
         0
      </eas:inputBlockOutputSlotId>
      <eas:outputBlockId xmlns:eas="...">
         sort
      </eas:outputBlockId>
      <eas:outputBlockInputSlotId xmlns:eas="...">
         0
      </eas:outputBlockInputSlotId>
   </nsas:value>
</eas:link>
```

Verknüpfung von der Sortierungs – Operation zur FetchFirst – Operation

```
<eas:link xmlns:eas="...">
   <nsas:value>
      <eas:inputBlockId xmlns:eas="...">
         sort
      </eas:inputBlockId>
      <eas:inputBlockOutputSlotId xmlns:eas="...">
         0
      </eas:inputBlockOutputSlotId>
      <eas:outputBlockId xmlns:eas="...">
         fetch
      </eas:outputBlockId>
      <eas:outputBlockInputSlotId xmlns:eas="...">
         0
      </eas:outputBlockInputSlotId>
   </nsas:value>
</eas:link>
```

Verknüpfung von der FetchFirst – Operation zur Return – Operation

```
<eas:link xmlns:eas="...">
   <nsas:value>
      <eas:inputBlockId xmlns:eas="...">
         fetch
      </eas:inputBlockId>
      <eas:inputBlockOutputSlotId xmlns:eas="...">
         0
      </eas:inputBlockOutputSlotId>
      <eas:outputBlockId xmlns:eas="...">
         return
      </eas:outputBlockId>
      <eas:outputBlockInputSlotId xmlns:eas="...">
         0
      </eas:outputBlockInputSlotId>
   </nsas:value>
</eas:link>
```

Verknüpfung von der Return – Operation zur Datensenke von α

```
<eas:link xmlns:eas="...">
   <nsas:value>
      <eas:inputBlockId xmlns:eas="...">
         return
      </eas:inputBlockId>
      <eas:inputBlockOutputSlotId xmlns:eas="...">
         0
      </eas:inputBlockOutputSlotId>
      <eas:outputBlockId xmlns:eas="...">
         mainSink
      </eas:outputBlockId>
      <eas:outputBlockInputSlotId xmlns:eas="...">
         0
      </eas:outputBlockInputSlotId>
   </nsas:value>
</eas:link>
```

Die Erläuterungen zu den jeweiligen vollständigen Einträgen sind [10] zu entnehmen.

Für die übrigen zwei Beispielanfragen wird in entsprechend ähnlicher Weise verfahren. Sie unterscheiden sich hauptsächlich nur in den zu definierenden Konfigurationsparameter, die in Kapitel **8.2.1** beschrieben sind.

11. Literaturverzeichnis

1: Nicklas, D.; Großmann, M.; Schwarz, T.; Volz, S.; Mitschang, B.: *A Model-Based, Open Architecture for Mobile, Spatially Aware Applications*, Universität Stuttgart, Institut für parallele und verteilte Systeme (IPVS), 2001.
2: Hönle, N.; Käppeler, U.P.; Nicklas, D.; Schwarz, T.; Grossmann, M.: *Benefits of Integrating Meta Data into a Context Model*, Universität Stuttgart, Institut für parallele und verteilte Systeme (IPVS), 2004.
3: Institut für parallele und verteilte Systeme (IPVS): *Vorhaben im 2. Förderungszeitraum*, http://as.informatik.uni-stuttgart.de/as/nexus/fp2/tpb1.html , Stand: 29.12.08.
4: Cattrell, R.G.G.; Barry, Douglas K.: *The Object Data Standard: ODMG 3.0*. o.O.: Academic Press 2000.
5: Law, Y.-N.; Wang, H.; Zaniolo, C. : *Query Languages and Data Models for Database Sequences and Data Streams*. 30th VLDB Conference. Toronto (Kanada). 2004.
6: Bai, Y.; Thakkar, H.; Luo, C.; Wang, H.; Zaniolo, C.: *A Data Stream Language and System Designed for Power and Extensibility*. CIKM '06. Arlington (USA). 5.-11. November 2006.
7: Arasu, A.; Babu, S.; Widom, J.: *The CQL continuous query language: semantic foundations and query execution*. The VLDB Journal 15.2, Seiten 121-142, 2005 .
8: Cammert, M.; Heinz, C.; Krämer, J.; Seeger, B.: *Anfrageverarbeitung auf Datenströmen*. Datenbankspektrum 1, , 2005 .
9: Tucker, P.; Maier, D.; Sheard, T.; Fegaras, L.: *Exploiting Punctation Semantics in Continuous Data Streams*. IEEE Transactions on knowledge and data engineering 15.3, Seiten 555-568, 2003 .
10: Sardina, D.G. : *Framework for Distributed Data Processing*, Institut für parallele und verteilte Systeme, Universität Stuttgart, 2008.
11: Altenbernd, P.: *Sortieren*, http://www.fbi.h-da.de/fileadmin/personal/p.altenbernd/pgII_files/3_Sortieren.pdf Seiten 23ff., Stand: 31.12.2008.

Erklärung

Ich versichere hiermit, dass ich diese Arbeit selbständig verfasst und nur die angegebenen Hilfsmittel und Quellen verwendet habe.

Stuttgart, den 07.01.2009

_____

Markus Dörr