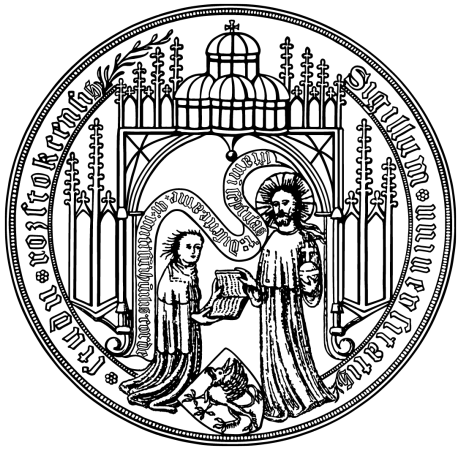

Masterarbeit

Stromdatenverarbeitung für Data-Science-Anwendungen basierend auf Sensordaten

vorgelegt von Mark Lukas Möller
Matrikelnummer 212204377
am 04. September 2017

am Lehrstuhl für Datenbank- und Informationssysteme
der Universität Rostock

Erstgutachter
Prof. Dr. rer. nat. habil. Andreas Heuer
Zweitgutachterin
PD Dr.-Ing. habil. Meike Klettke

Betreuer
Dr.-Ing. Holger Meyer
Hannes Grunert

Abstract

Zur Anwendung von Operationen der relationalen Algebra auf strömende Daten sind Methodiken nötig, die Unterschiede in den Paradigmen der Verarbeitung von kontinuierlichen Daten gegenüber materialisierten Daten beachten. Im Rahmen dieser Arbeit wird untersucht, wie sich Selektions-, Projektions-, Extremwert- und Verbundoperationen auf Datenströme anwenden lassen. Es wird ein Konzept entwickelt, welches spezifische Eigenschaften wie Ressourcenbeschränkungen und ungenaue Daten, insbesondere bei Verbundattributen, berücksichtigt. Auf Basis dieser konzeptuellen Betrachtung wird ein Demonstrator unter Verwendung der Programmiersprache Python entwickelt, der die Funktionsfähigkeit der entwickelten Algorithmen darlegen soll.

Abstract

For the application of operations of the relational algebra on streaming data, methodologies are needed that take differences of the paradigms of processing continuous data versus materialized data into account. This master thesis examines how selection, projection, extreme value determination, and join operations can be applied to data streams. A concept is developed which considers specific characteristics such as resource restrictions and inaccurate data, especially in the context of j attributes. Based on this conceptual consideration, a demonstrator will be developed using the programming language Python, which will demonstrate the functionality of the algorithms.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ACID	Atomicity, Consistency, Isolation, Durability
CQL	Continuous Query Language
DBMS	Datenbank-Managementsystem
DML	Data Manipulation Language
DSMS	Data Stream Management System
ETL	Extraction, Transformation, Load
GODESS	Gotland Deep Environmental Sampling Station
GPIO	General Purpose Input Output
HDFS	Hadoop Distributed File System
IOW	Leibniz-Institut für Ostseeforschung Warnemünde
SDW	Stream-Data-Warehouse
SQL	Structured Query Language

1 Einführung

Seit Anfang des Jahrtausends sehen sich Informatiker immer stärker mit dem Trend der Verarbeitung wachsender und hochdimensionaler Datenmengen konfrontiert. In Fachkreisen etablierte sich dafür das Fachwort *Big Data*. 2001 stellte der *Gartner* Analyst Doug Laney drei Dimensionen für Eigenschaften und deren Problematiken von Big Data vor, für welche im Umgang und bei der Verarbeitung von Daten Lösungen gesucht werden müssen [Sic1313, Lan01]. Diese wurden unter den *Drei V's von Big Data* bekannt: *Volume*, *Velocity* und *Variety* [Lan01]. Die drei Dimensionen sind in Abb. 1 veranschaulicht und sollen nachfolgend erklärt werden, wobei der Schwerpunkt hauptsächlich auf Velocity liegen wird.

Volume bezeichnet die Dimension von Big Data, die sich mit der Menge von Daten auseinander setzt. In datenintensiven Systemen ist es üblich, dass Datenmengen in einem Tera- oder gar Petabytebereich produziert werden: Das *CERN* gab 2012 an, dass bei ihrem Teilchenbeschleuniger jährlich etwa 30 Petabytes produziert werden, die durch Wissenschaftler ausgewertet werden müssen [Eur12] und die Analysten von *IDC* gaben an, dass bis 2020 weltweit etwa 40 000 EB[1] an Daten produziert sein werden [GR12]. Datenbanksysteme können hierfür zwar eingesetzt werden, allerdings werden diese schnell teuer hinsichtlich Speicherplatz und Rechenkapazität (vgl. [KTGH13]). Daher muss entschieden werden, welchen Wert Daten haben bzw. wie relevant diese sind und ob es lohnt, diese zu speichern (vgl. [KTGH13]).

Variety beschäftigt sich mit der Homogenität von Daten. Daten können strukturiert, semi-strukturiert oder unstrukturiert vorliegen, wie in Abb. 1 ersichtlich. Während strukturierte Daten wie etwa eine Datenbankrelation mit explizitem Schema von einem Datenbanksystem effizient verarbeitet werden können, sind unstrukturierte Daten, wie etwa Video- oder Audiodateien, schwieriger zu analysieren und zu verarbeiten. Lösungsansätze dafür werden in Forschungsthematiken bezüglich der Dimension der Variety gesucht (vgl. [KTGH13]).
Das dritte Kernproblem nach Laney stellt Velocity, die „Schnelligkeit“ bzw. die Datenfrequenz dar [Lan01]. Velocity beschreibt die Problematik der schnellen oder gar Echtzeitverarbeitung von Daten, die im Kontext von Data Science nicht als abgeschlos-

[1] Exabyte, 10^{18} Bytes.

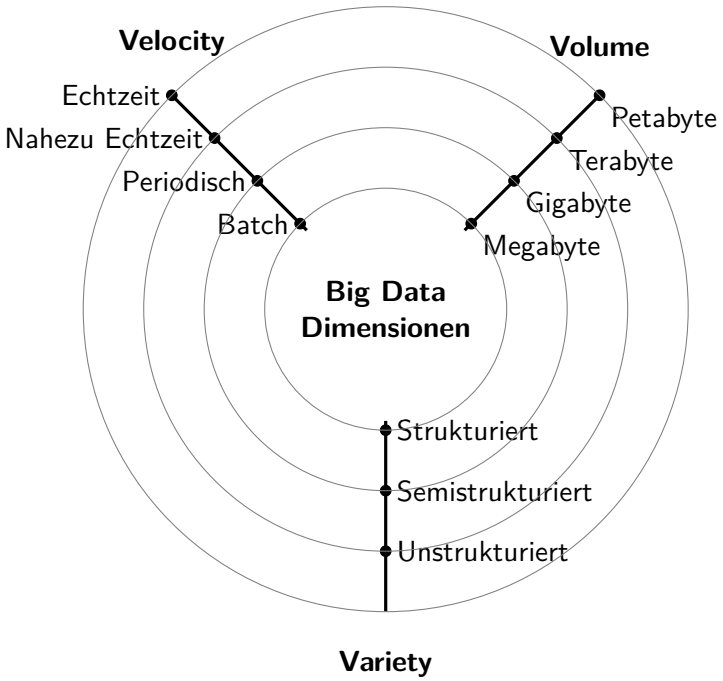

Abb. 1: Dimensionen von Big Data nach [KTGH13]

sene Datensätze, sondern als Datenströme vorliegen können. Datenströme können als nicht-persistente Datensätze angesehen werden, deren Daten *kontinuierlich*, *inkrementell* generiert werden und im Allgemeinen nicht absehbar ist, wann ein Datensatz abgeschlossen sein wird (vgl. [GÖ10, S.1]). Eine genauere Betrachtung von Datenströmen wird im Abschnitt 2.1 vorgenommen. Würden blockierende Operationen – also jene Operationen, welche die gesamte Eingabe konsumieren müssen, bevor sie Berechnungen durchführen und eine Ausgabe generieren können (vgl. [CHK+04, S. 2]) – in Umgebungen mit strömenden Daten auf konventionellem Wege ausgeführt werden, d.h. auf das Ende der Eingabe warten und erst dann die Berechnung durchführen, würde dies zu Problemen führen, da durch das Blockieren die Eigenschaft der Velocity unerfüllbar werden kann. Es müssen Alternativen zu konventionellen, auf abgeschlossenen Datenmengen stattfindenden Berechnungen gefunden werden, die für Datenströme praktikabel sind.

Im Jahre 2013 führte *IBM* den Begriff *Veracity* als viertes *V* ein, welches die Qualität bzw. die Zuverlässigkeit von Daten beschreibt [KTGH13] [ZPD+13]. Insbesondere beschreibt diese Dimension den Umgang mit Daten von unterschiedlichen Quellen. Diese können das Problem der Ungenauigkeit von Daten bergen, wenn Daten für Analysen nicht oder nicht rechtzeitig bereinigt werden können (vgl. [KTGH13]).

Velocity und Veracity sind die Hauptproblemdimensionen von Big Data, dessen Not-

wendigkeiten des effektiven Umganges im Kontext strömender Daten betrachtet wird. Warum eine nahezu Echtzeitverarbeitung von strömenden Daten in Stromdatenumgebungen notwendig ist, soll in den nachfolgenden Kapiteln erläutert werden. Das Suchen von Lösungen für Teilprobleme der Verarbeitung strömender Daten, insbesondere das Anwenden von Operationen der relationalen Algebra, soll Gegenstand dieser Arbeit sein. Veracity spielt eine Rolle, da insbesondere bei der Verbundoperation ersichtlich sein wird, dass eine hohe Datenqualität nicht immer gegeben sein wird. Ein Verbund wird nicht als *Exact-Match* sondern nur über ähnliche Attributwerte durchgeführt, sodass sich diese Ungenauigkeiten im System fortpflanzen können. Entfernt hat bei Veracity auch die Tatsache eine Bedeutung, dass Sensoren ausfallen können. Dieser Punkt wird aber im Rahmen der Masterarbeit nicht betrachtet.

1.1 Motivation

Die Verarbeitung strömender Daten ist Bestandteil vieler Anwendungsszenarien. Eines davon ist das Live-Monitoring von Daten, bei dem man bestimmte Veränderungen von Messwerten feststellen und so auf bestimmte Ereignisse reagieren kann.

Beispielhaft soll ein solches Szenario für das *Leibniz-Institut für Ostseeforschung Warnemünde* (IOW) beschrieben werden: Das IOW misst mittels einer Sonde die Veränderungen von Parametern wie Salzgehalt oder Sauerstoff in der Ostsee. Dafür nutzen sie die *Gotland Deep Environmental Sampling Station* (GODESS), eine Plattform, die eine Sonde mit einer Vielzahl von Sensoren besitzt. Diese wird jeweils über einen Zeitraum von etwa drei Monaten im Gotlandbecken abgesenkt und die gemessenen Daten nach der anschließenden Bergung im IOW ausgewertet und interpretiert[IOW01, PSB11].

Dabei ist ein Nachteil erkennbar: Nach dem Absetzen der GODESS dauert es im Regelfall ein Vierteljahr, bis die Daten zur Verarbeitung zur Verfügung stehen. Den Forschern des IOW ist es dabei nicht möglich, die Daten in Echtzeit abzurufen und auf spontane Ereignisse, wie etwa einem seltenen Salzwassereinbruch von der Nordsee in die Ostsee, zu reagieren (vgl. [Mö16]). Solche Umwelteffekte werden erst im Nachhinein bei der Datenauswertung sichtbar.

An dieser Stelle ist auch ein zweites Motiv ableitbar: Die Vorverarbeitung von Daten. Es ist nicht immer gewünscht oder notwendig, gemessene Rohdaten vollständig zu speichern oder weiterzuverarbeiten. Mittels Sensoren ist es möglich, Daten vorzufiltern, wenn diese nicht relevant sind. Möglicherweise sind für das IOW nur Änderungen des

Salzgehaltes interessant und damit relevant zu speichern, wenn sich diese in einem bestimmten Zeitraum um einen bestimmten Wert ändern. Ähnliches Beispiel wäre der genannte Anwendungsfall beim CERN: Hier ist es fraglich, ob wirklich alle 30 Petabyte an Daten pro Jahr relevant für die nachfolgenden Auswertungen sind oder ob bestimmte Daten im Vorfeld filterbar gewesen wären. Sollen solche Vorauswertungen und Vorfilterungen auf unterster Ebene – der Sensorebene – geschehen, so müssen dafür effektive Algorithmen entwickelt werden, die den üblicherweise geringen Speicher und die geringe Leistung von Sensoren berücksichtigen und trotzdem den Analyseanforderungen der Forscher gerecht werden.

Auch wenn das Leibniz-Institut für Ostseeforschung Warnemünde (IOW) zum gegenwärtigen Zeitpunkt noch über keine Infrastruktur im Kontext der Stromdatenverarbeitung verfügt, die es erlaubt, Daten sofort nach ihrer Generierung zu verarbeiten, so soll mit dieser Arbeit die Grundlage für eine solche Datenverarbeitungsweise geschaffen werden. Dabei soll der Blick weniger auf der Infrastruktur, sondern auf der Algorithmik von Stromdatenverarbeitung liegen.

1.2 Zielsetzung

Ziel dieser Arbeit ist es, ein Konzept zu entwickeln, welches es ermöglicht, Stromdatenoperationen auf ressourcenbeschränkten Sensoren auszuführen. Im Gegensatz zur GODESS, die über einen externen Speicher zur mittelfristigen Speicherung der Messdaten verfügt, soll als Voraussetzung für diese Arbeit davon ausgegangen werden, dass die Sensoren über einen minimalen internen Zwischenspeicher für Halten von Daten verfügen und Daten während des Strömens verarbeitet werden, d.h. ein *Extraction, Transformation, Load* (ETL)-Prozess, bei dem die Daten erst aus den Sensoren extrahiert werden, auf ein bestimmtes Schema angepasst und dann persistent gespeichert werden, findet vor der Datenverarbeitung bzw. Datenanalyse nicht statt. Es soll direkt auf den Daten gearbeitet werden, die in den durch Sensoren generierten Datenströmen vorhanden sind. Entsprechend effizient müssen Daten verarbeitet werden, da Sensoren üblicherweise nur eine geringe Anzahl an Daten puffern können.

Als Stromdatenoperationen werden hier insbesondere Operationen der relationalen Algebra angesehen, die auf strömenden Daten arbeiten. Es sollen typische Anfragen der relationalen Algebra so transformiert werden, dass sie auch auf kontinuierlichen Datenströmen anwendbar sind. Als typisch gelten hier insbesondere die Selektions-, Projektions-, Extremwertbestimmungs- und Verbundoperationen. Hierbei steht nur die Implementierung von Stromdatenoperatoren im Vordergrund. Die automatische Zerlegung

und Transformation von entsprechenden Anfragen in Stromdatenoperatoren ist nicht Bestandteil dieser Arbeit. Es werden Modelle, Operationen und Systeme betrachtet, die sich zur Nutzung in Data-Science-Anwendungen nutzen lassen. Beispielhaft wird dabei ein Data-Science-Szenario vorgestellt, an welchem die Eignung des Konzeptes durch eine prototypische Implementierung der entsprechenden Algorithmen nachgewiesen wird.

1.3 Aufbau der Arbeit

Diese Arbeit gliedert sich in folgende Teile: Neben der erfolgten Einleitung, die die Problematik bei der Datenverarbeitung auf Sensoren und die Rolle von Velocity und Veracity einleitend erklären soll, werden im nachfolgenden Kapitel Grundlagen vermittelt, die für das Verständnis im Umgang mit strömenden Daten oder für die verwendeten Konzepte im Konzeptions- und Umsetzungsteil notwendig sind.

In den beiden darauffolgenden Kapiteln wird eine Übersicht zum aktuellen Stand der Technik und zum Stand der Forschung gegeben. Es wird gezeigt, welche Techniken und Technologien bereits in existierenden Lösungen für die Datenstromverarbeitung existieren und welche Ansätze diesbezüglich Gegenstand der Forschung sind.

Im fünften Kapitel wird ein Konzept entwickelt, das die Zielsetzung der Anwendung von Operationen der relationalen Algebra auf strömenden Daten aus Abschnitt 1.2 erfüllt. Im anschließenden Kapitel wird eine konkrete Umsetzung beschrieben und prototypisch gezeigt, dass die konzeptionelle Ausarbeitung funktioniert.

Letztlich werden gewonnene Erkenntnisse und erzielte Ergebnisse zusammengefasst und die Umsetzung bewertet. Es wird ein Ausblick auf weitere Thematiken gegeben, die im Rahmen dieser Arbeit nicht betrachtet werden konnten, aber weitere Fragestellungen zur Problematik darstellen, die es zu lösen gilt.

2 Theoretische Grundlagen

In diesem Kapitel werden theoretische Grundlagen vermittelt, die nötig sind, um ein allgemeines Verständnis für strömende Daten zu entwickeln oder die dazu dienlich sind, die Zielsetzung zu erreichen. Es wird gezeigt, warum die Grundfunktionalitäten eines *Data Stream Management System* (DSMS) vonnöten sind und welche Techniken zur Datenverarbeitung benötigt werden. Insbesondere werden Eigenschaften strömender Daten beschrieben, diese materialisierten Daten verglichen und weiterhin relevante Techniken zur Fensterbildung beschrieben.

2.1 Datenströme

Liegen zu verarbeitende Daten nicht als materialisierter Datensatz vor, sondern treffen Daten kontinuierlich ein, so spricht man von *strömenden Daten*[2] bzw. von Daten, die sich in einem *Datenstrom* befinden (vgl. [BBD+02, S. 2], [Wik17a]). Während bei einem relationalen DBMS die Persistentmachung und die anschließende, möglicherweise zeitaufwändige Verarbeitung im Vordergrund steht, stellt das Operieren auf Datenströmen ein Paradigma dar, in welchem die persistente Speicherung im Allgemeinen nicht möglich ist und eine *datengetriebene* Verarbeitung im Vordergrund steht [CHK+04]. Daten treffen dabei mit einer sehr hohen Datenrate[3] ein (vgl. [GG07, S. 25]). Im Folgenden sollen Eigenschaften herausgestellt werden, die später als charakteristisch für Datenströme angesehen werden können und darlegen, warum strömende Daten anders behandelt werden müssen, als nichtströmende Daten.

Aus der bereits genannten Charakteristik des kontinuierlichen Eintreffens von Daten lassen sich Eigenschaften, Anforderungen und Herausforderungen ableiten. Das Ende eines Datenstroms ist im Allgemeinen unbekannt, d.h. es ist nicht voraussagbar, wann bzw. ob ein Datenstrom endet [BBD+02, S. 2] [GG07, S. 25]. Daraus wiederum ergibt sich die Problematik, dass Algorithmen, die zwingend auf der Gesamtmenge von Daten arbeiten müssen, im ungünstigsten Falle unendlich lange blockieren würden, da sie auf

[2] In der Arbeit sprachlich gleichgesetzt mit *Stromdaten*.
[3] In der Arbeit sprachlich gleichgesetzt mit *Datenfrequenz* und *Datengeschwindigkeit*.

das Ende des Datenstroms warten. Insbesondere holistische[4] Algorithmen, wie etwa die Medianbestimmung (vgl. [GM04]), die sich nicht parallelisieren lassen, sind davon betroffen.

Es muss die Problematik betrachtet werden, dass die Verknüpfung von Daten über ein gemeinsames Attribut – der *Verbund* (*Join*) – im Allgemeinen nicht über die Gesamtmenge realisierbar ist. Ein Verbund verknüpft zwei Datensätze über ein Attribut genau dann, falls beide Datensätze dabei den gleichen Attributwert aufweisen (vgl. [SSH11, S.10]). Dies bedeutet, dass auch bei allen zukünftig eintreffenden strömenden Daten die Attributwerte untersucht werden müssen. Somit würde die Verbundoperation möglicherweise unbegrenzt lange blockieren[5, 6]. Ebenfalls muss hier die Randbedingung des begrenzten Speichers von Sensoren bedacht werden. Eine beliebige Anzahl von Daten kann nicht vorgehalten werden. Dieses Problem kann durch eine Kombination zweier Techniken umgangen werden: Das Anwenden einer Verbundoperation nur innerhalb eines Fensters und die Verknüpfung über das Attribut in Form eines Zeitstempels. Die konkreten Techniken dafür werden ab Abschnitt 2.6 (S. 25 ff.) und im Konzeptionskapitel ab S. 38 vorgestellt.

Eine weitere Charakteristik eines Datenstromes ist, dass die Datenrate bzw. Messfrequenz der Sensoren oder damit auch die Frequenz der strömenden Daten in einem Strom pro Zeiteinheit variieren kann (vgl. [Wik17a]). Ebenso möglich ist, dass die Reihenfolge der strömenden Daten nicht zwingend erhalten bleiben muss (vgl. [BBD+02, S. 2]). Auf diese Eigenschaften muss bei der Implementierung von Algorithmen Rücksicht genommen werden. Kommen Daten etwa unsortiert an oder liegen diese innerhalb eines Windows, das längst abgearbeitet wurde, so muss entschieden werden, was mit diesen Daten geschieht.

2.2 Klassifikation von Aggregatfunktionen

Für die Ausführung von Aggregatfunktionen auf Datenströmen muss deren Komplexität betrachtet werden. Es wird dabei zwischen dabei zwischen *distributiven*, *algebraischen* und *holistischen* Aggregatfunktionen unterschieden.

Eine Aggregatfunktion f ist genau dann distributiv, wenn für zwei disjunkte Multimengen X und Y gilt: $f(X \cup Y) = f(X) \cup f(Y)$. Distributive Aggregatfunktionen sind inkrementell berechenbar und konstant hinsichtlich der Raum- und Zeitkomplexität pro

[4] s. Abs. 2.2, S.16 f.

[5] Eine Operation ist blockierend, wenn diese die komplette Eingabe konsumieren muss, um eine Ausgabe zu produzieren und vorher kein Ergebnis propagiert werden kann (vgl. [CHK+04, S. 2]).

[6] Gültig für den allgemeinen Fall, wenn Daten nicht sortiert vorliegen. Dann sind zwingend alle Tupel zu betrachten, bevor der Verbundalgorithmus terminiert.

Tupel [GÖ10, S. 15]. Beispiele für distributive Aggregate sind etwa die Summenbildung (`SUM`), die Berechnung von Maximal- und Minimalwerten (`MAX`, `MIN`) oder das Zählen aller Tupel ohne Duplikateliminierung (`COUNT`) [GÖ10, S. 15].

Eine Aggregatfunktion f ist genau dann algebraisch, wenn sie aus zwei distributiven Aggregaten berechnet werden kann. Auch algebraische Aggregatfunktionen sind inkrementell berechenbar und konstant hinsichtlich Raum- und Zeitkomplexität [GÖ10, S. 15]. Beispiel für distributive Aggregatfunktionen sei die Durchschnittsberechnung (`AVG`), da sie aus den zwei distributiven Aggregatfunktionen – die der Summenbildung und die der Zählung der Tupel – realisiert werden kann (`AVG/COUNT`) [GÖ10, S. 15].

Die komplexeste Klasse von Aggregatfunktionen stellen die holistischen Funktionen dar. Eine Aggregatfunktion f ist genau dann holistisch, wenn für zwei Multimengen X und Y die Berechnung $f(X \cup Y)$ exakt so viel Speicher benötigt, wie die Vereinigung von X und Y, $X \cup Y$ [GÖ10, S. 15]. Dies bedeutet, dass man für die Berechnung einer holistischen Aggregatfunktion zwingend die gesamte Eingabe konsumieren muss, bevor eine Ausgabe generiert werden kann und dass dieser Prozess nicht parallelisierbar ist. Beispiel für holistische Aggregatfunktionen sind TOP-k-Berechnungen (`TOP-k`) oder die Zählung aller Tupel mit Duplikateliminierung (`COUNT DISTINCT`) [GÖ10, S. 15].

Insbesondere holistische Aggregatfunktionen stellen eine Herausforderung für Velocity dar. Dadurch, dass holistische Algorithmen nicht parallelisierbar sind, ist eine effiziente Berechnung holistischer Probleme nicht oder nur in Teilen möglich. Für diese Komplexitätsklasse müssen andere Lösungen gefunden werden, wie etwa Näherungsverfahren oder erlaubter Unschärfe bzw. erlaubter Ungenauigkeit.

2.3 Data Stream Management Systeme

Wie im Abschnitt 2.1 erwähnt, sind klassische DBMS nur gering geeignet, um den Anforderungen zur Verarbeitung von Datenströmen gerecht zu werden. Die Beschränktheit solcher Systeme soll vergleichend genauer erläutert und die Notwendigkeit von *Data Stream Management Systemen* (DSMS) eingehend dargelegt werden. Anschließend werden Kernkonzepte von DSMS erläutert.

2.3.1 Gegenüberstellung von DBMS und DSMS

Der Kernunterschied zwischen einem DBMS und DSMS besteht im Paradigma der Anfrageverarbeitung. Ein DBMS arbeitet *pull-basiert* bzw. *query-driven*. Eine Anfrage wird verarbeitet, indem der aktuelle Zustand der Datenbank gelesen („*gepulled*“) wird und die Berechnung dann auf diesem Zustand ausgeführt wird. (vgl. [GÖ10, S. 2]) Es liegt also eine materialisierte, zum Zeitpunkt der Anfrage abgeschlossene Menge an Daten vor, auf der eine beliebige, sofern gültige[7], Berechnung ausführt werden kann. Ein DSMS arbeitet dagegen *push-basiert* bzw. *data-driven*. Berechnungen auf kontinuierlichen Daten finden bei ihrer Ankunft im DSMS statt und das Ergebnis wird weiterverarbeitet, etwa, indem es als Datensatz in einen weiteren Stream geschickt wird. Die ursprünglichen, strömenden Daten werden danach im Allgemeinen verworfen und nicht materialisiert (vgl. [GÖ10, S. 2]), d.h. eine persistente Datenhaltung, wie es eine Kerneigenschaft in einem DBMS ist[8, 9], findet nicht statt. Die sofortige Verarbeitung setzt im Gegensatz zum DBMS voraus, dass die Anfrage schon im Voraus definiert wurde. Eine weitere beliebige Anfrage nach dem Verarbeiten der Daten ist aufgrund des Verwerfens dieser nicht möglich. Folglich ist der Zugriff auf Daten in einem DBMS *mehrfach* möglich, während die Daten in einem DSMS *einmalig* verarbeitet werden können (vgl. Tab. 1, [GÖ10, S. 2 f.]).

Ein DBMS findet häufig dann Anwendung, wenn in einem Anwendungsszenario Daten selten aktualisiert werden (vgl. [GÖ10, S. 3]). Es herrscht in diesen System eine geringe Aktualisierungsfrequenz[10] vor. In einem DSMS dagegen ist die Aktualisierungsfrequenz sehr hoch, da ständig neue Daten eintreffen und sich damit der auszuwertende Datenbestand kontinuierlich verändert (vgl. Tab. 1, [GÖ10, S. 2 f.]). Auch die Art der Aktualisierung ist verschieden. Während in einem DBMS Daten beliebig eingefügt, aktualisiert und gelöscht werden können, erfolgt das Einfügen in DSMS prinzipiell *append-only*, d.h. strömende, neu eintreffende Daten werden einfach nur an den bestehenden Datenbestand angehängt (vgl. Tab. 1, [GÖ10, S. 2 f.]). Die aus DBMS bekannte UPDATE-Operation ist in DSMS nicht notwendig, da Daten nur für einen kurzen Zeitpunkt gehalten, verarbeitet und anschließend gelöscht werden.

Die Einsatzzwecke und Funktionsweisen zwischen einem DBMS und DSMS sind vollkommen unterschiedlich, wie dargelegt wurde. Es wurde herausgestellt, warum Ersteres nicht zur Verarbeitung von Datenströmen geeignet ist. An dieser Stelle soll erwähnt werden, dass es auch *Stream-Data-Warehouses* (SDW) gibt, die das Pendant zum traditionellen

[7] Gültig im Sinne von syntaktisch und semantisch korrekt.

[8] „*Daten sind* persistent*, wenn sie über die Laufzeit des Programmes hinaus, in dem sie verarbeitet werden, weiter existieren. Anderenfalls spricht man von* transienten *Daten.*“ [Kud15].

[9] „*DBMSe verwalten persistente (langfristig zu haltende) Daten, die einzelne Läufe von Anwendungsprogrammen überstehen sollen.*“ [SSH08, S. 9].

[10] *INSERT*-, *UPDATE*- und *DELETE*-Anweisungen.

Eigenschaft	DBMS	DSMS
Daten	Persistente Relationen	Datenströme, Fenster
Datenzugriff	Zufällig	Sequentiell, einmalig
Updates	Beliebig	Append-Only
Aktualisierungsfrequenz	Gering	Sehr hoch
Verarbeitungsmodell	Query-driven (pull-based)	Data-driven (push-based)
Anfragen	One-Time	Kontinuierlich
Antwortgenauigkeit	Exakt	Exakt oder Approximiert
Latenz	Relativ hoch	Gering

Tabelle 1: Unterschiede zwischen DBMS und DSMS nach [GÖ10, S. 3]

Data-Warehouse darstellen. Bei diesen liegt der Fokus auf dem Speichern von aktuellen und historischen Daten mit einer hohen Aktualisierungsrate mit einem schnellen, leichtgewichtigen ETL-Prozess (vgl. [GÖ10, S. 4 f.]). Da diese Arbeit sich mit dem Operieren von Daten auf Sensorebene beschäftigt und die Speicherung in diesem Kontext weniger Priorität hat, werden SDW nicht weiter ausgeführt.

2.3.2 Rahmenarchitektur eines DSMS

Im Folgenden soll die Rahmenarchitektur eines DSMS nach [GÖ10] vorgestellt werden. Diese besteht, wie in Abb. 2 ersichtlich, aus mehreren Komponenten: Der Buffer/Input-Monitor ist dafür zuständig, die eingehenden Datenströme zu verwalten und zwischenzuspeichern. Außerdem ist er dafür zuständig, zu entscheiden, was geschieht, wenn das System nicht alle Daten halten kann. Beispielsweise können zufällige Tupel verworfen werden („*Random Sampling*“) [GÖ10, S. 4]. Der Arbeitsspeicher (*Working Storage*) speichert kleine Teile des Streams und Datenstrukturen für das Ergebnis, die für die einzelnen Anfragen benötigt werden [GÖ10, S. 4]. Der lokale Speicher (*Local Storage*) dient zur Metadatenspeicherung und als Übersetzungstabelle, etwa von einer Sensor-ID zu einem für den Menschen lesbaren Namen (vgl. [GÖ10, S. 4]). Dem Nutzer ist es möglich, die Metadaten durch Updates zu ändern. Das Repository hält kontinuierliche Anfragen vor und erstellt Anfragepläne zur Abarbeitung [GÖ10, S. 4]. Der Anfrage-Prozessor ist für die Abarbeitung der Anfragepläne zuständig und kann auch mit dem Input-Monitor kommunizieren, etwa um bei bestimmten Situationen, etwa bei Änderung der Datenfrequenz, die Anfragepläne zu adaptieren [GÖ10, S. 4]. Letztendlich werden die Ergebnisse weiter gestreamt.

Ähnlich wie bei relationalen Anfragen werden auch hier Interoperator-Queues und Scheduling-Algorithmen benötigt [GÖ10, S. 4]. Konzeptuell wird ein Datenstrom von einem Operator konsumiert, von dem Operator entsprechend abgewandelt und als veränderter

Datenstrom zu einem möglichen nächsten Operator weitergegeben oder zu einer Applikation weitergeleitet, die etwa ein Monitoring durchführt. Auch das anschließende Speichern in einem Stream-Data-Warehouse ist möglich [GÖ10, S. 4].

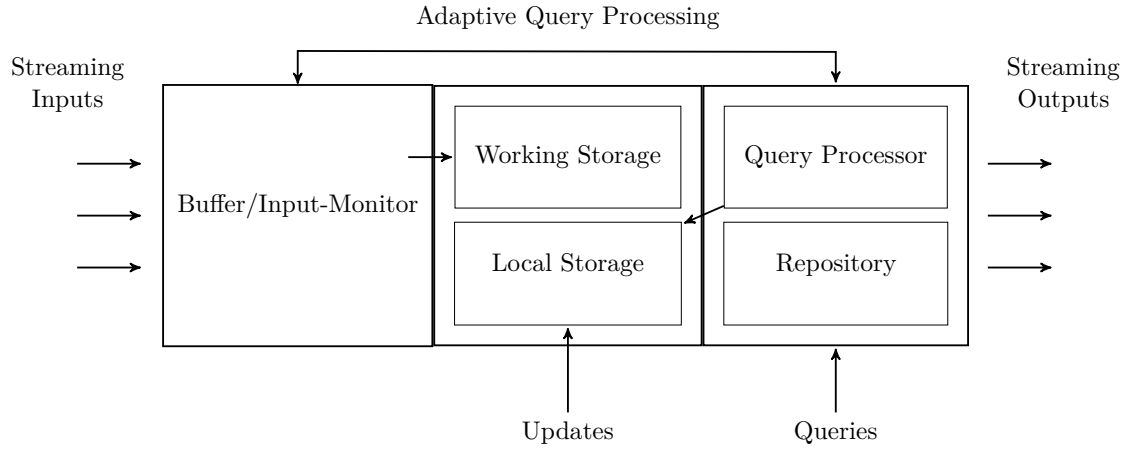

Abb. 2: Referenzarchitektur eines DSMS nach [GÖ10, S. 4]

2.4 Continuous Query Language

Die *Continuous Query Language* (CQL) ist eine deklarative Anfragesprache, die im Rahmen des *STREAM* DSMS-Projektes der Universität Stanford entstand. Gemäß den Entwicklern ist CQL als kleine, syntaktische Erweiterung zu SQL anzusehen [ABB+04, S. 3]: Sie erweitert die Anfragesprache SQL um Datentypen für Streams und behandelt Anfragen auf diesen entsprechend [Gö14]. Trotz der ursprünglichen Entwicklung von CQL für STREAM findet die Anfragesprache auch in anderen DSMS Anwendung (vgl. Abschnitt 3.1, S. 31 f.).

Bei Anfragen mittels SQL in einem DBMS werden die Daten, auf denen operiert werden soll, als *statisch* bzw. *isoliert* angesehen. Werden also während einer Berechnungsoperation neue Daten eingefügt oder verändert, so wird trotzdem auf den Daten operiert, die zum Zeitpunkt des Beginns der Berechnung vorhanden waren. Während der Operation eingefügte, modifizierte oder gelöschte Daten werden für die Berechnung ignoriert. Dies entspricht auch der *Isolation* des ACID[11]-Prinzips[12]. CQL-Anfragen in einem DSMS sehen Anfragen dagegen als *kontinuierlich* an, d.h. eine Anfrage wird registriert und läuft gegebenenfalls

[11] *Atomicity, Consistency, Isolation, Durability* (ACID): Vier Grundkonzepte des Transaktionsmanagements.

[12] Der Standard-Isolationsgrad der Serialisierbarkeit sei vorausgesetzt.

unendlich lange. Realisiert wird dies z. B. mit Hilfe von Fenstertechniken (vgl. [Gö14]), die in Abschnitt 2.6 erläutert werden. Hierbei wird der Snapshot der Elemente im Fenster eines Stream in eine statische Relation umgewandelt, auf der dann operiert wird [Gö14]. Verschiebt sich das Fenster, wird ein neuer Snapshot des neuen Windows angelegt, auf welchem dann erneut operiert wird.

In CQL wird ein Stream S als (möglicherweise unendliche) Multimenge (Bag) von Elementen $\langle s, \tau \rangle$ angesehen, wobei s ein Tupel ist, dass dem Schema des Streams S entspricht und $\tau \in \mathcal{T}$ der Timestamp des Elementes ist [ABW06, S. 124]. Die Zeitdomäne $\mathcal{T}$ ist dabei anwendungsspezifisch und kann beispielsweise konkrete Werte wie nichtnegative ganzzahlige Werte, etwa $\{0, 1, \ldots\}$, oder Werte vom Datentyp *Datetime* repräsentieren [ABW06, S. 124]. Eine Relation R ist in diesem formalen Modell eine ungeordnete Multimenge von Tupeln, die dem Schema von R entsprechen. im Gegensatz zum standardmäßigen relationalen Modell spielt bei CQL die zeitliche Komponente eine Rolle. Eine Relation $R(\tau)$ wird immer über eine Zeitinstanz $\tau \in \mathcal{T}$ definiert [ABW06].

CQL bietet die Möglichkeit, durch bestimmte Schlüsselwörter in der *Data Manipulation Language* (DML) kontinuierliche Anfragen zu erstellen. Die Relation wird als Stream angesehen, wenn beispielsweise die Fenstergröße hinter der Relation angegeben wird. Es sei der Stream `BoughtArticles` und die Relation `Pricelist` gegeben. Um etwa ein Sliding Window der Größe 5 auf `BoughtArticles` zu realisieren, definiert man diese bei der Angabe des Streams. Das Beispiel

```
SELECT MAX(P.price)
FROM BoughtArticles[Rows 5] as B, Pricelist as P
WHERE B.ISBN = P.ISBN;
```

gibt jeweils den höchsten Preis der fünf zuletzt gestreamten Artikel aus (vgl. [ABW06]). `[Rows 5]` ist dabei ein Schlüsselwort aus CQL, welches ein Sliding Window über jeweils 5 Elemente definiert (vgl. [ABW06, S. 121]).

2.4.1 Abstrakte Semantik von CQL

Die Semantik der CQL basiert auf drei Klassen von Operatoren auf Streams und Relationen [ABW06, S. 124]:

- *Stream-To-Relation*-Operatoren, die einen Stream S konsumieren und eine Relation R produzieren. Zu jedem Zeitpunkt τ soll $R(\tau)$ aus S berechenbar sein [ABW06, S. 124 f.].

- *Relation-To-Relation*-Operatoren, die eine oder mehrere Relationen $R_1, \ldots, R_n$ konsumieren und eine Relation R produzieren. Zu jedem Zeitpunkt τ soll $R(\tau)$ dabei aus $R_1(\tau), \ldots, R_n(\tau)$ berechenbar sein [ABW06, S. 124 f.].
- *Relation-To-Stream*-Operatoren, die einen Relation konsumieren R und einen Stream S produzieren. Zu jedem Zeitpunkt τ soll dabei S aus R berechenbar sein [ABW06, S. 124 f.].

Die drei Klassen von Operatoren sind in Abb. 3 grafisch verdeutlicht. *Stream-To-Stream*-Operatoren werden nicht explizit angegeben, da diese aus den obigen Operatorenklassen konstruiert werden können (vgl. [ABW06, S. 124]).

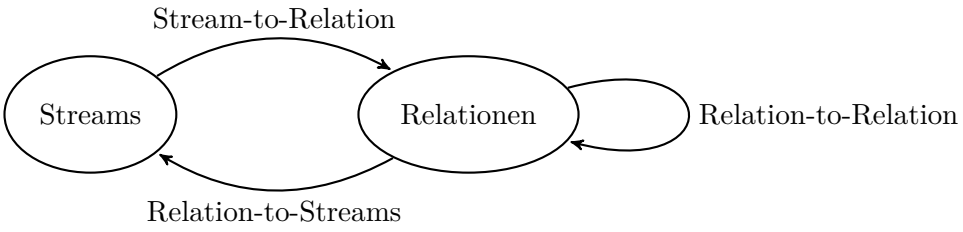

Abb. 3: Operatorklassen und deren Beziehungen nach [ABW06, S. 125]

2.4.2 Relation-To-Stream Operatoren

CQL bietet drei Arten von Streams: Den *Istream* (Insert-Stream), den *Dstream* (Delete-Stream) und den *Rstream* (Relation-Stream). Diese sollen genauer betrachtet werden.

Der Istream-Operator, welcher auf eine Relation R angewendet wird, enthält ein Stream-Element $\langle s, \tau \rangle$ mit dem Tupel s und dem Zeitpunkt τ, wenn das Tupel s während des Zeitpunktes τ hinzugekommen ist, d.h. wenn $s \in R(\tau) - R(\tau - 1)$ (vgl. [ABW06, S. 128]). Formal bedeutet dies:

$$Istream(R) = \bigcup_{\tau \geq 0} ((R(\tau) - R(\tau - 1)) \times \{\tau\}) \quad \text{[ABW06, S. 128].}$$

Der DStream-Operator, der auf eine Relation R angewendet wird, enthält alle Elemente, die zum Zeitpunkt $\tau - 1$ in R vorkamen, nicht aber zum Zeitpunkt τ (vgl. [ABW06, S. 128]). Die formale Definition dafür ist:

$$Dstream(R) = \bigcup_{\tau > 0} ((R(\tau - 1) - R(\tau)) \times \{\tau\}) \quad \text{[ABW06, S. 128].}$$

Der RStream-Operator, der auf eine Relation R angewendet wird, enthält ein Stream-Element $\langle s, \tau \rangle$, wenn ein Tupel s zum Zeitpunkt τ in R vorkommt (vgl. [ABW06, S. 128]).

Formal:

$$Rstream(R) = \bigcup_{\tau \geq 0} (R(\tau) \times \{\tau\})$$ [ABW06, S. 128].

Beispielhaft würde die Abfrage

```
SELECT Istream(*)
FROM TemperatureSensor [Range Unbounded]
WHERE temperature > 25
```

alle Elemente ausgeben, bei denen die Temperatur größer als 25 °C ist und die bisher noch nicht gestreamt wurden (vgl. [ABW06, S. 128]).

2.4.3 Stream-To-Relation Operatoren

Alle Stream-To-Relation Operatoren basieren auf Sliding-Window-Techniken, die auf einem Stream angewendet werden. Das Window wird wie im Abschnitt 2.4 genannt durch Snapshots von Streams realisiert [ABW06, S. 126]. CQL stellt dabei drei Arten von Sliding Windows bereit: Ein *zeitbasiertes (time-based) Sliding Window*, ein tupelbasiertes (tuple-based) Sliding Window und ein *partitioniertes (partitioned) Sliding Window* [ABW06, S. 126 f.]. Weiterhin kann CQL etwa um *Tumbling Windows*, *Fixed Windows* und *wertebasierte (value-based) Windows* erweitert werden [ABW06, S. 126]. Die konkrete Erläuterung der Window-Arten findet im Abschnitt 2.6 statt.

2.5 Operator-Parallelismen

Um Anfragen parallel ausführen zu können, sind zwei Techniken der Parallelität innerhalb einer Anfrage – beim *Intra-Query-Parallelismus* – ausnutzbar. Diese Techniken sind *Interoperatorparallelität* und *Intraoperatorparallelität* der [ÖV11, S. 513]. Der Unterschied zwischen den Parallelitätsarten soll aufgezeigt und im Laufe der Arbeit soll dargelegt werden, warum insbesondere Interoperatorparallelität für die Verarbeitung von Sensordaten relevant ist.

2.5.1 Intraoperatorparallelität

Die Intraoperatorparallelität basiert auf der Dekomposition eines Operators in mehrere Teiloperatoren. Dies geschieht durch die Dekomposition von Relationen. Auf jeder der Teilrelationen – den Buckets – wird dann der Operator angewendet [ÖV11, S. 513]. So lässt sich dann beispielsweise eine Selektion parallelisieren, wie in Abb. 4 grafisch veranschaulicht.

Eine Relation R wird in die Buckets $R_1, \ldots, R_n$ partitioniert. Auf jeder der Partitionen wird entsprechend die Selektionsoperation $Sel._1, \ldots, Sel._n$ ausgeführt, die die Teilergebnisse $S_1, \ldots, S_n$ generieren (vgl. [ÖV11, S. 513]). Die Teilergebnisse – in Abb. 4 $S_1, \ldots, S_n$ – werden letztendlich zum Gesamtergebnis S vereinigt.

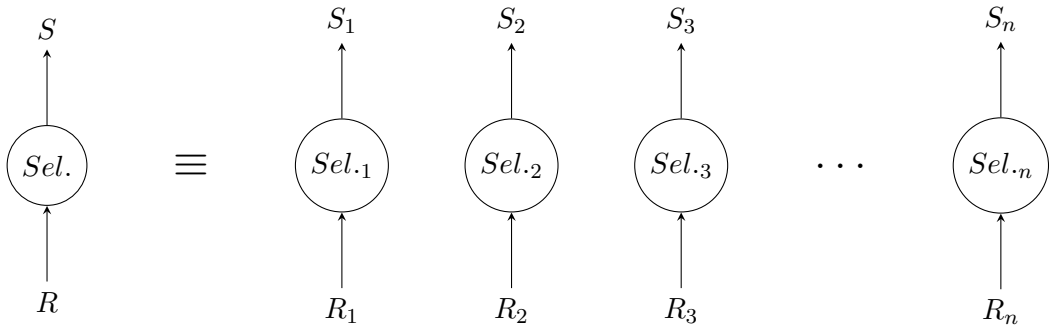

Abb. 4: Visualisierung des Intraoperatorparallelismus nach [ÖV11]

2.5.2 Interoperatorparallelität

Bei der *Interoperatorparallelität* können zwei Anfragen parallel ausgeführt werden und von nachfolgenden Operationen genutzt werden, wie in Abb. 5 ersichtlich: Hier existieren zwei Select-Anfragen, die parallel ausgeführt werden und deren Ergebnisse von einer Join-Operation konsumiert werden. Im Gegensatz zur Intraoperatorparallelität handelt es sich hier nicht um eine Dekomposition von Anfragen, sondern um jeweils eigenständige und vollständige Anfragen.

Interoperatorparallelität unterteilt sich in zwei Varianten. Die erste Variante ist *Pipeline-Parallelismus*. Dieser basiert auf dem *Erzeuger-Verbraucher-Pattern*, bei der die Erzeuger Operationen parallel berechnen können und Zwischenergebnisse erzeugen. Die Verbraucher konsumieren dann die Zwischenergebnisse und verarbeiten sie gegebenenfalls weiter (vgl. [ÖV11, S. 513]). Abb. 5 stellt so einen Pipeline-Parallelismus dar. Vorteil hierbei ist, dass Zwischenergebnisse nicht materialisiert werden müssen und dadurch hinsichtlich Speicher eine hohe Performance erreicht wird (vgl. [ÖV11, S. 513]).

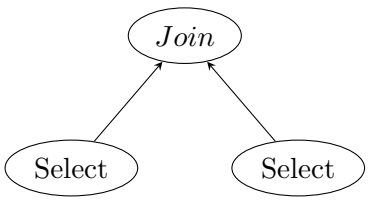

Abb. 5: Visualisierung des Interoperatorparallelismus nach [ÖV11]

Gibt es keinen Zusammenhang zwischen parallel ausgeführten Operatoren, wird von *unabhängiger Parallelität* gesprochen. Die Operatoren können parallel ausgeführt werden und zeichnen sich dadurch aus, dass die Prozessoren, auf denen die Operatoren laufen, unabhängig voreinander arbeiten können (vgl. [ÖV11, S. 513]). Im Laufe der Arbeit wird ersichtlich, warum insbesondere Interoperatorparallelität für die Verarbeitung von Sensordaten im Vergleich zu Intraoperatorparallelität relevant ist.

2.6 Windowing

Bereits genannt wurde die Schwierigkeit, bestimmte Algorithmen auf Datenströmen anzuwenden, etwa weil sie auf dem gesamten Datenbestand operieren müssen, bevor sie ein Berechnungsergebnis produzieren können. Nach [CHK+04, S. 2] bestehen zur Lösung dieses Problems verschiedene Lösungsansätze. Eine davon ist das Windowing, also die Bildung von Fensteranfragen. Beim Windowing wird jeweils nur ein bestimmter Ausschnitt der Daten betrachtet. Der aktuell betrachtete Ausschnitt wird dabei *Window* oder *Fenster* genannt. Die Berechnungen werden jeweils nur auf der Teilmenge der Daten im Fenster durchgeführt (vgl. [TM11, S. 626]) und nicht auf der – auf Grund der Kontinuität möglicherweise nie zustande kommenden – Gesamtheit der strömenden Daten.

Gemäß [GÖ10, S. 11 f.] sind die Window-Arten nach drei Kriterien unterteilbar, die in Abb. 6 dargestellt sind:

- Nach der *Bewegungsrichtung des Windows*, also wann und wie weit sich das Fenster verschiebt. Hierzu gehören etwa Sliding Windows oder Landmark Windows.
- Nach *Definition des Inhaltes*: hierzu gehören beispielsweise zeitbasierte oder logische Windows.
- Nach *Häufigkeit der Verschiebung* des Windows. Hierzu gehören etwa Jumping Windows oder Tumbling Windows.

Anzumerken ist, dass diese drei Kriterien nicht disjunkt sein müssen. Es kann etwa ein *Count-Based Sliding Window* implementiert werden. Im Folgenden werden einige ausgewählte Window-Arten vorgestellt und entsprechend charakterisiert.

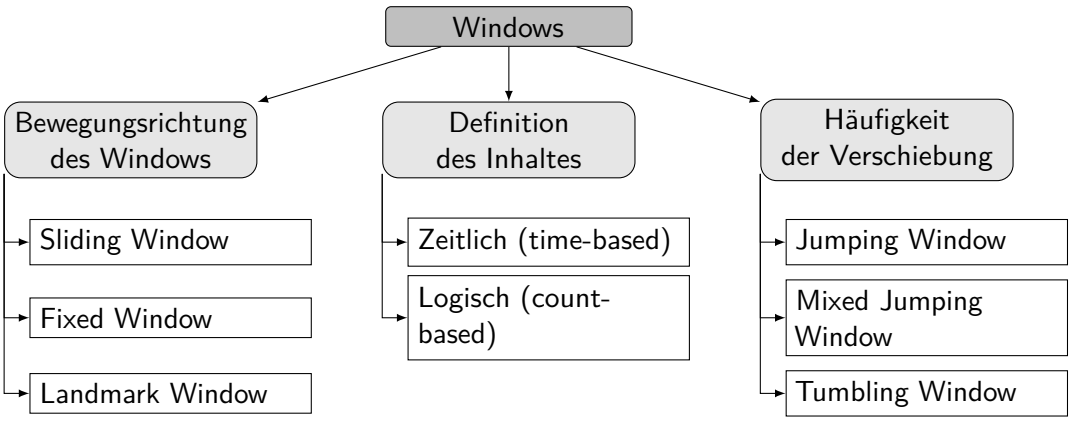

Abb. 6: Klassifikation von Window-Arten nach [GÖ10]

2.6.1 Sliding Window

Das Sliding Window ist eines der Windows, das nach *Bewegungsrichtung* charakterisierbar ist. Die Fenstergröße ist hierbei fest vorgegeben und das Fenster „gleitet“ über die Daten hinweg (vgl. [GÖ10]). Dabei werden Daten jeweils in verschiedenen Fensterpositionen zur Berechnung herangezogen, wie in Abb. 7 ersichtlich.

In der Abbildung seien neun Messwerte im Puffer eines Datenstroms geschrieben worden, über denen mittels eines Sliding Windows Berechnungen durchgeführt werden sollen. Die Grafik sei so zu verstehen, dass Daten in einem Datenstrom von links nach rechts strömen. Entsprechend sind rechts die zeitlich ältesten Daten, während die Daten links die zeitlich neusten Daten im Datenstrom darstellen. Analog sind die Grafiken zu den anderen Window-Arten zu verstehen.

Im ersten Berechnungsschritt werden beim Sliding Window die ältesten drei Messwerte `{1, 2, 3}` zur Berechnung herangezogen. Danach verschiebt sich das Fenster um einen Datenpunkt, d.h. der nächstjüngste Datenpunkt wird in die Berechnung mit einbezogen, während der älteste Datenpunkt des Fensters nicht mehr zur Berechnung herangezogen wird. Im Beispiel enthält das nächste Fenster nun also die Messwerte `{2, 3, 4}`. Charakteristisch für ein Sliding Window ist die Eigenschaft, dass man zwischen zwei Fenstern nur um jeweils einen Datenpunkt verschiebt. Ist die Weite der Verschiebung zwischen zwei Fenster größer als 1, so spricht man von einem *Jumping Window* (vgl. Abs. 2.6.4, S. 29)[13].

[13]Dies gilt im Rahmen dieser Masterarbeit, in der zwischen einem Sliding Window und einem Jumping Window unterschieden wird. In der Literatur wird dies unterschiedlich gehandhabt. In manchen Artikeln wird ebenfalls diese Unterscheidung vorgenommen, während in anderen Artikeln lediglich ein Sliding Window mit einer Verschiebungsweite definiert wird.

Das Sliding-Window hat die Eigenschaft, dass das Window während des Gleitens *das gleiche* Fenster bleibt („Fenster 1“) und sich über verschiedene Zeiträume (t_0, t_1, t_2) über andere Daten legt. Dies ist Kernunterscheid zu den in Abb. 8 erläuterten Fenstern, bei denen zwischen den verschiedenen Zeitpunkten aus Sicht der Implementation neue Fenster gebildet werden.

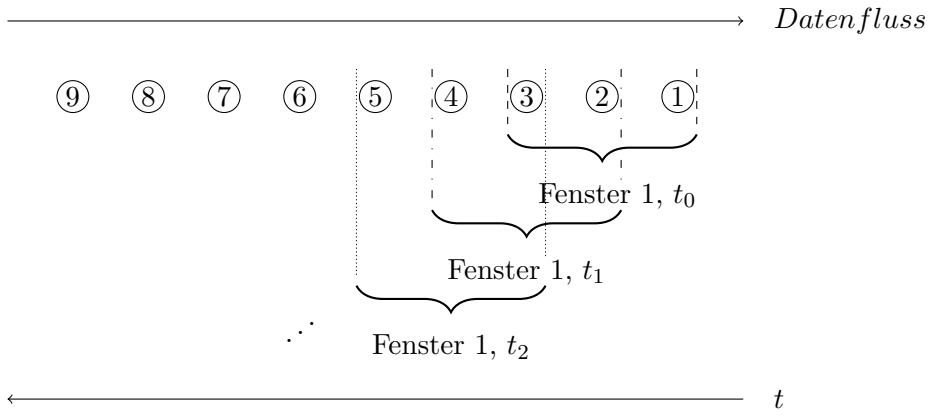

Abb. 7: Sliding Window mit Fenstergröße 3

2.6.2 Fixed Window

Bei einem *Fixed Window* werden die Start- und Endpunkte durch den Anwender fest definiert. Die Idee des Fixed Windows soll durch einen Vergleich mit dem Sliding Window erfolgen. Ein Unterschied zwischen den beiden Fensterarten ist in den Abbildungen 7 und 8 ersichtlich. Während ein Sliding Window aus *einem* Fenster besteht, welches sich über die Daten schiebt, bestehen ein Fixed Window aus einem Fenster, was über die Laufzeit des Programmes hinweg definiert ist. Es können mehrere Fixed Windows definiert werden, die ein Sliding Window simulieren, aber über einen Vorteil verfügen: Während bei einem Sliding Window Daten nicht wieder aufgerufen werden können, über die das Window bereits geschoben wurde[14], so sind bei einem Fixed Window die Start- und Endpunkte des Windows fest definiert, d.h. sofern die Fenster und die Daten, die in den Fenster liegen, nicht de-initialisiert werden, sollen die Daten zu beliebigen Zeitpunkten[15] erneut lesbar sein (vgl. [GÖ10]).

[14] Unter der Annahme, dass ein Sliding Window nicht zurückgeschoben wird.
[15] Beliebiger Zeitpunkt nach Initialisierung des Fixed Windows.

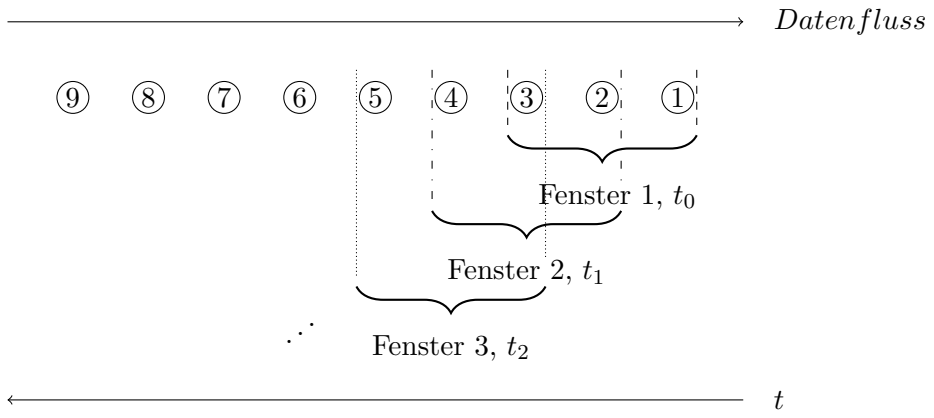

Abb. 8: Fixed Window mit drei vorher definierten Fenstern

2.6.3 Landmark Window

Das Landmark Window hat die Kerneigenschaft, dass dieses sich nur in eine Richtung weiterverschiebt. Es zählt nach Abb. 6 zu den Windows, die nach Bewegungsrichtung charakterisiert sind [GÖ10]. Während der Startpunkt des Fensters fix bleibt, verschiebt sich der Endpunkt des Fensters mit dem Eintreffen neuer Datenpunkte. Das Landmark Window betrachtet also alle Daten von Beginn der Messung bis zum aktuellen Zeitpunkt [GÖ10], wie in Abb. 9 ersichtlich: Das erste Window nutzt den Wert `{1}` für die Berechnung, das zweite Window die Werte `{1, 2}` und das n-te Window die n ersten Werte des Datestroms `{1, 2, ..., n}`.

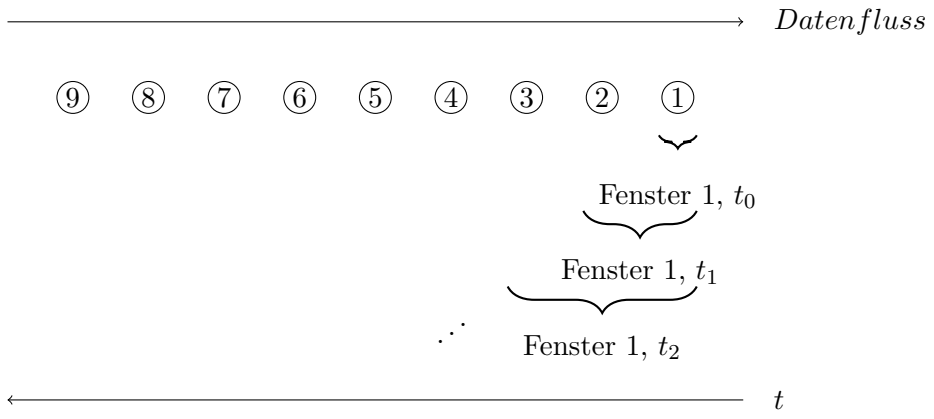

Abb. 9: Landmark Window

2.6.4 Jumping Window

Das *Jumping Window* ist nach Abb. 6 den Fenstern zuzuordnen, dessen Charakteristik die Häufigkeit der Verschiebung ist. Während sich etwa bei einem Count-Based Window das Fenster bei jedem Eintreffen von neuen Daten aktualisiert, geschieht dies beim Jumping Window nur alle n Ticks bzw. beim Eintreffen jedes n-ten Datensatzes [GÖ10]. In Abb. 10 ist dies dargestellt. Hier ist ein Jumping Window mit einer Fenstergröße von 3 Datensätzen und einer Aktualisierungsrate von 2 Ticks[16] definiert worden. Die Daten `{1, 2, 3}` seien der Inhalt des aktuellen Windows. Nun trifft der Datensatz Daten `{4}` ein. Da eine Aktualisierungsrate von 2 Ticks definiert wurde, geschieht an der Stelle nichts. Trifft der Datensatz Daten `{5}` ein, so ist die Aktualisierungsrate von 2 Ticks erfüllt und das Fenster schiebt sich auf die drei neusten Datenwerte. Der Inhalt des Windows beträgt nun Daten `{3, 4, 5}`.

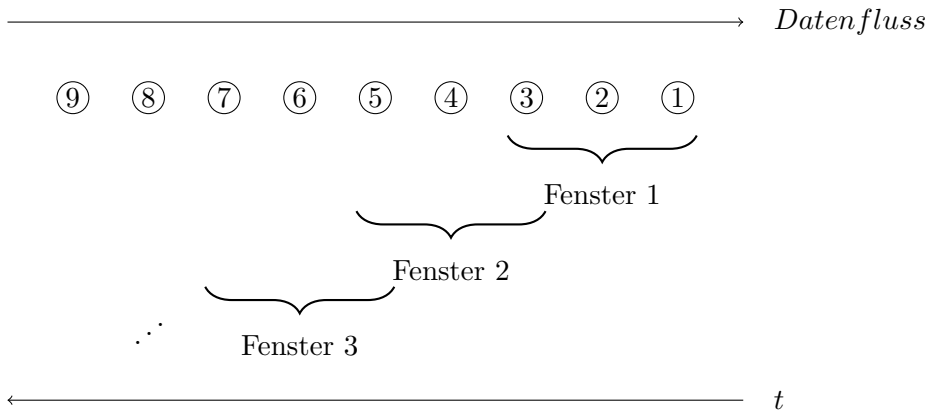

Abb. 10: Jumping Window mit Aktualisierungsrate von 2 Ticks

2.6.5 Tumbling Window

Das *Tumbling Window* ist ein Spezialfall des Jumping Window. Bei dieser Art des Fensters ist die Verschiebungsweite genauso groß wie die Fenstergröße. Daraus folgt, dass zwei Tumbling Windows jeweils disjunkt sind [GÖ10]. Dies ist in Abb. 11 nachvollziehbar: Die Fenstergröße und die Verschiebungsweite beträgt jeweils 3. Im ersten Fenster sind die Datenpunkte `{1, 2, 3}` vorhanden. Dann verschiebt sich das Fenster um drei Datenpunkte und es werden nun zur Berechnung die Datensätze `{4, 5, 6}` betrachtet. Es folgt eine erneute Verschiebung, nachdem drei weitere Datenpunkte eingetroffen sind und für die letzte Berechnung werden nun `{7, 8, 9}` herangezogen.

[16] Im Sinne von eintreffenden Daten. Ein neuer Datensatz entspricht einem Tick.

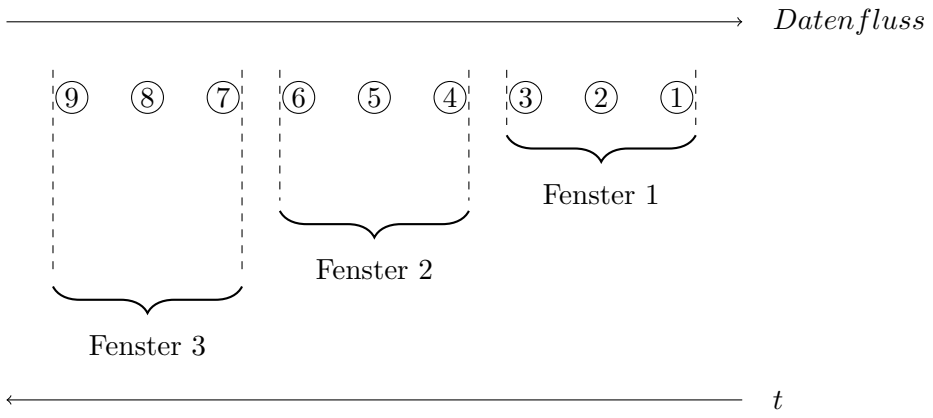

Abb. 11: Tumbling Window mit Verschiebungsweite 3

2.7 Einordnung der theoretischen Grundlagen

Die vorgestellten theoretischen Grundlagen sollen grob eingeordnet werden. Die Paradigmen von DBMS und DSMS wurden gegenübergestellt, um die Notwendigkeit eines DSMS zu zeigen. Der im Rahmen dieser Masterarbeit entwickelte Demonstrator wird Grundfunktionalitäten eines DSMS beinhalten. Datenströme bilden die Quelle für Daten, auf denen Operationen ausgeführt werden. Aufgrund der Kontinuität der Daten ist es von Bedeutung, diese im Kontext von darauf auszuführenden Operationen von materialisierten Daten abzugrenzen und Eigenschaften von Datenströmen bzw. strömenden Daten zu verdeutlichen. CQL wurde vorgestellt und dient als höhere Anfragesprache, die in einem Gesamtkonzept die Schnittstelle zum Anwender des Demonstrators darstellen kann. Anfragen in CQL sollen in Operationen des Demonstrators transformiert werden. Diese Operationen könnten in einem Gesamtsystem so geschachtelt werden, dass Interoperatorparallelität zustande kommt. Diese Transformation ist allerdings nicht Teil der Arbeit. CQL eignet sich perspektivisch sehr gut als Anfragesprache, da Datenströme bzw. Fensterkonzepte in der DML unterstützt werden. Die Komplexität umzusetzender Operationen, insbesondere von Aggregatfunktionen, wurde erläutert, um ein Verständnis zu entwickeln, welche Auswirkungen begrenzte Ressourcen und die Kontinuität von Daten auf Algorithmen, insbesondere Aggregationen, haben. Im Konzeptions- und Umsetzungskapitel wird auf Basis der theoretischen Grundlagen ein Konzept für die Anwendung von Operationen auf Datenströmen entwickelt und umgesetzt. Ebenfalls werden Detailfragestellungen geklärt. Zunächst werden allerdings existierende Systeme und Forschungsthemen vorgestellt, die sich mit der Verarbeitung strömender Daten beschäftigen. Es wird geklärt, welcher der bestehenden Technologien sich für diese Masterarbeit verwenden lassen und welche eigenen Anteile notwendig sind.

3 Stand der Technik

3.1 Data Stream Management Systeme

Um Datenströme zu verarbeiten, gab es in der Vergangenheit verschiedene Ansätze. 2006 wurde an der Stanford-Universität das DSMS-Projekt „*Standford Stream Data Manager*“ („*STREAM*“) entwickelt. Besonderheit des Systems ist das Vorhandensein einer eigenen deklarativen, bereits vorgestellten Programmiersprache, der CQL, in der ein Stream als eigener Datentyp betrachtet wird und Operationen entsprechend abgewandelt verarbeitet werden [Gö14]. Anstatt eine Operation auf einer statischen Datenmenge durchzuführen, wie es in einem DBMS der Fall wäre, können bei STREAM die Daten bei Bedarf als kontinuierlich angenommen werden [Gö14]. CQL wurde im Abschnitt 2.4 (S. 20) detailliert betrachtet. Ähnliche Systeme wie STREAM sind *Aurora* von der Brown Universität[17], das *TelegraphCQ* System der Berkeley Universität[18], das als PostgreSQL-Erweiterung entwickelt wurde und ebenfalls eine eigene deklarative Sprache besitzt (vgl. [KCC+03]), sowie das *PIPES* System der Phillips Universität Marburg, das bezüglich seiner temporalen Operationen kompatibel zur CQL des STREAM Systems ist, allerdings zusätzlich Mechanismen implementiert, um die Stromrate zu reduzieren (vgl. [KS04]).

Die vier genannten Systeme STREAM, Aurora, PIPES und TelegraphCQ sind als Vorreitersysteme für heutige Produkte anzusehen (vgl. [MW13, S. 13]). Die Techniken von STREAM werden in *Oracle Complex Event Processing (CEP)*[19] verwendet [MW13, S. 13]. CEP ist ein auf Java aufbauendes System, das die Möglichkeit bietet, auf Basis von *Oracle CQL* – dem aus STREAM übernommenen CQL-Pendant – Berechnungen auf Datenströmen durchzuführen. Oracle CEP bietet eine Vielzahl an Features. Neben der Abfrage auf strömenden Daten mittels Oracle CQL ist es möglich, das System als *Publish-Subscribe*-System zu nutzen. Es ist also möglich, Datenströme Dritter zu abonnieren („subscribe“), sowie selbst seine strömenden Daten bereitzustellen („publish“). Eine systemeigene *Event-Processing-Language* (EPL) bietet die Möglichkeit, Daten zu filtern,

[17] http://cs.brown.edu/research/aurora/ [25.06.2016].
[18] http://telegraph.cs.berkeley.edu/telegraphcq/v0.2/ [25.06.2016].
[19] http://www.oracle.com/technetwork/middleware/complex-event-processing/overview/index.html [25.06.2016].

zu aggregieren und zu mergen (vgl. [Ora09]). Weitere Features sind unter [Ora09] ersichtlich.

Das System *StreamBase* (ehem. *StreamBase LiveView*) ist ein auf Aurora basierenden System (vgl. [MW13, S. 13]). Das System ist im Gegensatz zu Oracle CEP kommerziell und bietet keine weitere wissenschaftliche Literatur.

Cisco Connected Streaming Analytics (CSA) basiert auf *TruCQ* [van15, S. 10], ein von *Truviso* entwickeltes und auf TelegraphCQ basierendes System (vgl. [Con07]). Truviso wurde 2012 von Cisco aufgekauft. TruCQ basierte wie TelegraphCQ auf PostgreSQL. Neben einer modifizierten Version von PostgreSQL bietet das System auch ein Integrationsframework, sowie die Möglichkeit einer Visualisierung der Komponenten [Con07]. TruCQ – und damit auch Cisco CSA – erlaubt es, strömende Daten zu analysieren, Datenströme zu kombinieren oder sogar historische Daten mit einem Datenstrom zu verknüpfen. Standard-PostgreSQL Funktionen können auf Datenströme angewendet werden. [van15, S. 10 ff.]. Bei der Big Data Dimension der Velocity nach Abb. 1 (S. 11) ist Cisco CSA unter „nahezu Echtzeit“ anzusiedeln [van15, S. 12].

Projekte der *Apache Software Foundation* wie *Apache Flink*[20], *Apache Storm*[21] oder *Apache Spark Streaming*[22] sind weitere Softwareprojekte, die in diesem Kontext erwähnenswert sind. Apache Flink ist dabei ein Framework für In-Memory-Streaming, Apache Storm ein Framework für Stream Processing und Apache Spark ein System für In-Memory-Processing (vgl. [Sei15, S. 36 f.]), die von ihrem Funktionsumfang für die Analyse von Stromdaten geeignet wäre. Die Flink-Produkte eignen sich dennoch aufgrund ihrer Architektur nicht für die Zielsetzung dieser Masterarbeit, da sie auf dem Hadoop-Ökosystem[23] aufbauen und viele Systemabhängigkeiten wie etwa das *Hadoop Distributed File System* (HDFS) benötigen (vgl. [Sei15, S. 36 f.]). Weitere zu nennende State-of-the-Art Systeme sind etwa *Esper*[24], *Microsoft StreamInsight*[25] und *IBM InfoSphere Streams*[26] (vgl. [MW13, S. 13], [Hed17][27]).

Die genannten DSMS-Systeme sind aufgrund ihres Funktionsumfangs für die Nutzung in leistungsfähigen Rechnerclustern praktikabel, nicht aber für Sensoren mit eingeschränkten

[20] https://flink.apache.org/ [25.06.2016].
[21] http://storm.apache.org/ [25.06.2016].
[22] https://spark.apache.org/streaming/ [25.06.2016].
[23] http://hadoop.apache.org/ [25.06.2016].
[24] http://www.espertech.com/products/esper.php [25.06.2016].
[25] https://msdn.microsoft.com/de-de/library/ee362541(v=sql.111).aspx [25.06.2016].
[26] http://www-03.ibm.com/software/products/de/ibm-streams [25.06.2016].
[27] Durch die weitgehende Nutzung von Esper im Buch [Hed17], welches 2017 erschienen ist, wird auf die anhaltende Relevanz dieses Systems geschlossen, auch wenn der erste Release über 10 Jahre zurückliegt.

Ressourcen, insbesondere hinsichtlich der gewünschten Minimalität an Programmcode. Im Rahmen dieser Arbeit wird daher kein solches DSMS verwendet, sondern eine eigene minimale Lösung mithilfe einer Programmiersprache entworfen.

3.2 Sensorumgebungen[28]

Heutige Sensorumgebungen setzen sich aus einem modularen Aufbau zusammen, welcher in Abb. 12 visualisiert wurde (vgl. [Tor12]). Auf unterster Ebene finden sich die eigentlichen Sensoren wieder, die Parameter aus der Umgebung wie etwa Lichtstärke oder Luftfeuchtigkeit messen können. Diese bestehen häufig aus Mikrosystemen wie etwa MEMS. Ein oder mehrere Sensoren sind dabei über ein Bussystem an einen *Sensor Hub* angebunden welcher die Rohdaten der einzelnen Sensoren sammelt und sie über eine Schnittstelle (API) bereitstellt. Der Sensor Hub ist ein Mikrocontroller, der je nach Modell auch durch den Benutzer programmierbar sein kann. Er kann zur Vorfilterung von Daten dienen und unterstützt beim Zusammenführen der Daten verschiedener Sensoren („*Sensorfusion*“), etwa um aus einem Beschleunigungssensor und einem Gyroskop einen virtuellen Bewegungssensor zu erzeugen, aus dem sich eine Bewegung in X-, Y- und Z-Richtung ableiten lässt (vgl. [Tor12, Abb. 3]). Weiterhin bewältigt der Sensor Hub Probleme wie etwa das Sammeln von Daten von Sensoren mit unterschiedlicher Messfrequenz, sodass der eigentliche Prozessor auf modular höherer Ebene, energiesparend arbeiten kann (vgl. [Tor12]). Die Kombination aus den Sensoren und dem Sensor Hub mit Schnittstelle (in Abb.12 gestrichelt umrandet) ist das, was der Nutzer als eigentlichen Sensor wahrnimmt.

Die Sensordaten werden vom Sensor Hub üblicherweise von einem Application Processor abgerufen und auf diesem verarbeitet. Dies kann durch eine Programmiersprache auf einem Rechner geschehen, der mittels der vom Sensor Hub bereitgestellten Schnittstelle die Sensordaten abruft und auf seinen Prozessoren weiterverarbeitet. In dem Schema in Abb. 12 ist der Application Processor hierbei etwa als Bibliothek einer Programmiersprache zu verstehen, die durch die Nutzung einer höheren Sprache, wie etwa CQL, eingebunden wird.

Heutige Anwendungsumgebungen für solche Sensor-Hub-Infrastrukturen wären insbesondere die Sensorarchitekturen in Smartphones. Bei Google ist dies etwa der „*Android Sensor Hub*“-Chip, der in den hauseigenen Smartphones verbaut ist. Bei Geräten von Apple sind dies etwa die *Apple Motion*-Coprozessoren wie etwa der Apple M10 [Wik17c].

[28] Dieser Abschnitt wurde maßgeblich durch Gespräche mit Martin Kasparick und Sebastian Stieber, Institut für Angewandte Mikroelektronik und Datentechnik, Universität Rostock geprägt.

Außerhalb von Mobilfunkgeräten bieten Hersteller wie Bosch Sensoren und Sensor Hubs an. Dies sind beispielsweise Bauteile aus der *Bosch SensorTec* Serie[29]. Diese Sensor Hubs, wie etwa der *BMF055*, bieten einen Beschleunigungssensor, ein Gyroskop und ein Magnetometer sowie einen integrierten Sensor, der aus den genannten drei Sensoren die absolute Position bestimmen kann [Bos15]. Der Mikrocontroller des BMF055 ist ein *SAM D20* [Bos15, S. 14], der mit einer Maximalgeschwindigkeit von 48 MHz operiert [Atm16] und 256 kB Flash-Speicher für Anwendungslogik und 32 kB SRAM-Speicher für flüchtige Daten [Bos15, S. 14] bietet. Diese Kenngrößen sind auch etwa die Standardkenngrößen, die heutige vergleichbare Sensor Hubs und deren Mikroprozessoren bieten[30]. Weitere Sensoren in diesem Gebiet wären etwa die der Firma TDK InvenSense[31] oder die Produktlösungen von ST Microelectronics[32].

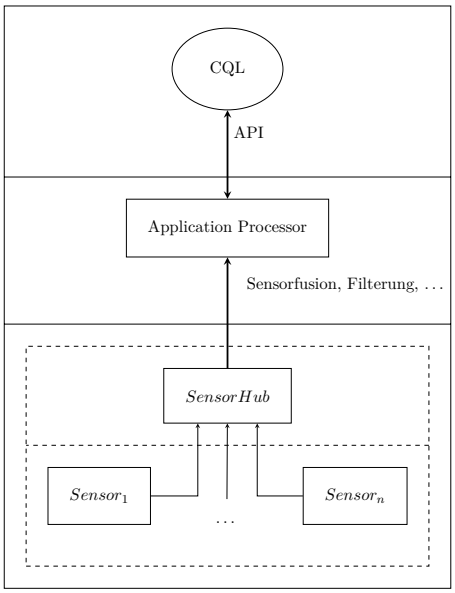

Abb. 12: Modularisierte Sensorumgebung (nach M. Kasparick und [Tor12])

[29] https://www.bosch-sensortec.com/bst/products/motion/hubs_nodes/overview_hubs_nodes [11.07.2017].
[30] Angaben entnommen aus einem Gespräch mit M. Kasparick, S. Stieber am 26.06.2017.
[31] https://www.invensense.com/solutions/ [11.07.2017].
[32] http://www.st.com/en/mems-and-sensors.html [11.07.2017].

4 Stand der Forschung

Dieses Kapitel gibt einen einen Überblick über aktuelle Forschungsthemen im Bereich der Datenstromverarbeitung. Ferner sollen konkrete Teilfragestellungen im Bereich Data Stream Processing betrachtet werden, zu denen derzeit nach Lösungen geforscht wird. Selten bezieht sich Literatur auf konkrete Forschungsumgebungen für die Datenstromverarbeitung. Es sind häufig nur zwei Kernthematiken: Artikel zu leistungsfähigen DSMS-Systemen, die auf großen Clustern arbeiten und Artikel zu konkreten Algorithmen, die auf kontinuierliche Daten angewendet werden. Insbesondere Forschungsliteratur zum letzteren Themenkomplex soll in diesem Kapitel beleuchtet werden.

4.1 Forschung im Bereich der Datenstromverarbeitung

Im Artikel „*Grand Challenge: Real-time High Performance Anomaly Detection over Data Streams*“ greifen die Autoren Jankov, Sikdar, Mukherjee et al. das Problem der Echtzeitverarbeitung von kontinuierlichen Daten auf. Insbesondere legen die Autoren Fokus auf die Möglichkeit der parallelen Verarbeitung und auf das Thema der Anomalien- und Ausreißerdetektion. Dazu beschreiben sie eine Systemarchitektur für ein auf diese Thematik zugeschnittenes DSMS, stellen Anomaliedetektionsmethoden vor und geben einen groben Überblick über ihre Implementierung in Java und C++ [JSM+17]. Motivation der Autoren war, ein Lösung für die Grand Challenge 2017 der ACM International Conference of Distributed and Event-Based Systems (DEBS) zu suchen[33] [JSM+17].

Laut den Autoren des Artikels [RTG14] waren Kang, Naughton und Viglas mitunter die ersten Forscher, die versucht haben, den Verbund auf Datenströmen zu beschreiben [RTG14, S. 710]. Die Grundidee wurde von den drei Autoren im Artikel „*Evaluating Window Joins over Unbounded Streams*“ veröffentlicht. Die Grundidee des Verbundes wurde hier in den drei nachfolgenden Schritten beschrieben [KNV03]:

- Ein Datenstrom wird nach Verbundpartnern durchsucht, Ergebnisse werden propagiert.

[33] http://www.debs2017.org/call-for-grand-challenge-solutions/ [11.07.2017].

- Im zweiten Datenstrom wird ein Tupel eingefügt.
- Im zweiten Datenstrom werden alle abgelaufenen Tupel gelöscht.

Auf dieser Idee aufbauend entstanden auch die Konzepte des Fuzzy-Merge-Joins und des Bufferlosen Fuzzy-Merge-Joins in dieser Masterarbeit.

Teubner und Mueller beschreiben im Artikel „*How Soccer Players Would do Stream Joins*“ eine Verbundoperation namens *Handshake Join*, deren Grundidee darin besteht, dass Daten in zwei Datenströmen entgegengesetzt zueinander fließen und innerhalb eines Windows geprüft wird, ob Tupelpaare innerhalb eines Windows zueinander passen, wie in Abb. 13 dargestellt ist [TM11]. Der Name Handshake Join und der Titel des Artikels leiten sich von der Geste ab, wie Fußballspieler sich vor einem Spiel gegenseitig die Hand geben. Der Handshake Join wurde ursprünglich mit dem Ziel der Ermöglichung eines hohen Datendurchsatzes konzipiert, die Latenz wurde in diesem Artikel ignoriert (vgl. [RTG14]). Als Latenz wird hier die Dauer definiert, die zwischen dem Eintreffen des zweiten Tupels und der Verbundbildung liegt. Wird ein Tupel direkt beim Eintreffen in einen Datenstrom in den Verbund aufgenommen, wäre die Latenz optimal (vgl. *Latency characteristics*, [RTG14, S. 710]) Teubner und Mueller betrachtet im Artikel „*Low-Latency Handshake Join*“, welches auf dem Artikel [TM11] aufbaut, wie die Latenz minimierbar ist ([RTG14], vgl. [TM11]).

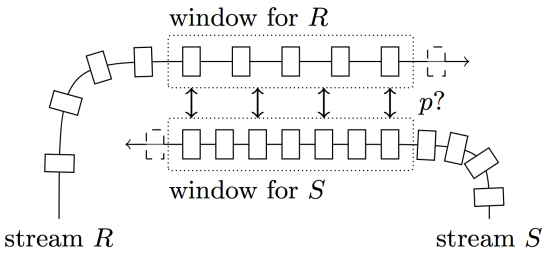

Abb. 13: Handshake Join nach [TM11]

Die Autoren De Francisci Morales und Gionis untersuchen in ihrem Artikel „*Streaming similarity self-join*“, wie man ähnliche Elemente in einem Datenstrom finden kann. Dazu definieren die Autoren ein Ähnlichkeitsmaß. Elemente im Datenstrom werden als hochdimensionale Featurevektoren angesehen, deren Ähnlichkeit durch die Cosinus-Ähnlichkeit gemessen werden kann. Weiterhin wird ein Dämpfungsfaktor für die zeitliche Komponente eingeführt, sodass Elemente, die inhaltlich gleich sind, nach einem bestimmten Zeithorizont dennoch als unterschiedlich angesehen werden. Je größer die zeitliche Differenz zwischen zwei Elementen ist, desto unterschiedlicher werden diese angesehen. Elemente, die hinter diesem Zeithorizont liegen, können aus dem Puffer des Datenstroms freigegeben werden

[DFMG16]. Die Autoren führen darauf aufbauend erweiterte Techniken mithilfe von Indexbildungen ein, die für den Rahmen der Masterarbeit nicht mehr relevant sind [DFMG16].

Gulisano, Papadopoulos, Nikolakopoulos et al. beschreiben im Artikel „*Performance Modeling of Stream Joins*“ eine Frameworkmodellierung, um Durchsatz und Latenz eines Datenstromes abschätzen zu können. Das Modell beschreibt, wie diese beiden genannten Parameter von der Anzahl der Datenströme und von dem Grad der Parallelisierung abhängen. Für diese Masterarbeit insbesondere von Interesse ist das Einleitungskapitel, das Verbundalgorithmen als Grundlage für das darauf aufbauende Framework vorstellt. Hier werden tupel- und zeitbasierte Algorithmen vorgestellt, mit denen ein Verbund auf Basis ähnlicher Zeitstempel möglich wird [GPN+17]. Diese Algorithmen dienen als Ausgangspunkt für den später vorgestellten Minimal-Delta-Join (s. Abs. 5.4.3, S. 51 ff.)[34].

4.2 Forschung im Bereich Sensorik

Bosch SensorTec bietet in einer der neueren Produktfamilien (BMA4XY) Sensoren mit „Embedded Intelligence“ an[35, 36]. Die dazugehörigen Produkte sind noch nicht erhältlich. Eine genaue Beschreibung von der eingebetteten Intelligenz gibt Bosch SensorTec nicht, allerdings ist davon auszugehen, dass diese Intelligenz vermutlich aus einem Mikrocontroller bestehen wird, der noch vor dem Sensor Hub auf den Sensoren sitzt und kleinere Operationen – wie etwa das Vorfiltern von Sensordaten im Sinne der Selektion – auf unterster Ebene erlaubt. Die konkreten Spezifikationen und tatsächlichen Features sind jedoch noch unklar.

[34] Tatsächlich wurde der Minimal-Delta-Join in dieser Masterarbeit unabhängig von diesem Artikel konzipiert. Das im Juni 2017 erschienene und im August 2017 entdeckte Artikel stellt aber einen sehr ähnlichen Algorithmus bereit, sodass es für die wissenschaftliche Redlichkeit den Ausgangspunkt für den beschriebenen Minimal-Delta-Join darstellen muss.

[35] https://ae-bst.resource.bosch.com/media/_tech/media/others/Bosch_Sensortec_Product_overview.pdf, S. 4 [11.06.2017].

[36] https://www.bosch-sensortec.com/bst/products/all_products/homepage_1___ohne_marginalspalte_69 [11.06.2017].

5 Konzeption

Das Konzeptionskapitel besteht aus zwei Teilen. Im ersten Teil soll die Darstellung der Idee erfolgen, wie man algebraische Operationen konkret auf Stromdaten umsetzen kann. Es werden die vorgestellten Eigenschaften aufgegriffen, welche es nicht erlauben, Anfragen der relationalen Algebra ohne Abwandlung auf Stromdaten auszuführen und Lösungen für diese Teilprobleme konzipiert. Dafür werden insbesondere die Operationen *Selektion*, *Projektion*, *Join* und *Extremwertbildung* (*Minimum*, *Maximum*) betrachtet. Der zweite Teil skizziert grob den Aufbau eines Frameworks, in welchem Operatoren auf Stromdaten umgesetzt werden.

5.1 Formale Semantik zur Abbildung algebraischer Operationen auf Stromdaten

Aufgrund der im Kapitel 2 herausgestellten Eigenschaften von strömenden Daten ist es nicht möglich, alle algebraische Operationen auf Stromdaten zu übertragen, ohne spezifische Problematiken zu betrachten. In diesem Abschnitt sollen Stromdatenoperatoren eingeführt und formal definiert werden, die entsprechende Probleme lösen. Angelehnt an die Semantiken der Artikel [ABW06] und [GPN+17] soll dafür folgende Semantik für den Rahmen dieser Masterarbeit eingeführt werden:

$S(\tau)$ ist ein von einem Sensor[37] SEN_S erzeugter Datenstrom, der eine Menge von Tupeln enthält, die bis einschließlich des Zeitpunktes τ produziert wurden. Das Element s_τ sei das Tupel des Streams S, das zum Zeitpunkt τ produziert wurde. τ ist dabei ein Element der Zeitdomäne $\mathcal{T}$, die anwendungsspezifisch definiert sein kann. Dies kann etwa ein Timestamp einer logischen Uhr oder ein Wallclock[38]-Timestamp sein. Für die nachfolgenden Abschnitte wird angenommen, dass $\tau \in \{0, \ldots, n\}$ sei, wobei n die Anzahl der vergangenen Millisekunden seit Anfang der Messung darstellt. Die erste Messung wird

[37] Analog gilt die formale Spezifikation für Daten, die nicht direkt vom Sensor erzeugt wurden, etwa von Streams aus einem Sensor Hub oder Daten in einem Stream, die das Ergebnis einer Stromdatenoperation darstellen.

[38] Wallclock-Time ist der vom Menschen wahrgenommener Zeitpunkt wie etwa 1. September 2017 12:47:11.

zum Zeitpunkt $\tau = 0$ erhoben. Eine diskrete Wertzuweisung – beispielsweise, in der der erste gemessene Wert den Zeitstempel $\tau = 0$, und zweite gemessene Wert den Zeitstempel $\tau = 1$ zugewiesen bekommt – ist ungünstig, da verschiedene Sensoren eine unterschiedliche Abtastfrequenz haben können und dies mit diskreten Werten schwieriger darzustellen wäre.

s_τ bezeichnet das Element s, welches zum Zeitpunkt τ produziert wurde. τ_s bezeichnet den Zeitstempel des Elementes s zum Zeitpunkt τ. Besteht ein Tupel T aus den Elementen s_1 bis s_m, so wird dafür die Schreibweise $S[\tau, s_1, \ldots, s_m]$ verwendet, wobei τ den aktuellen Zeitstempel des Tupels enthält, gefolgt von den m Datenwerten. Der Aufbau eines Tupels heißt angelehnt an relationale Datenbanken im *Schema*. Die einzelnen Elemente eines Tupels heißen *Attribute* (vgl. [SSH08, S. 11]).

Für alle Tupel s aus S gilt folgende Totalordnung:

$$s_{\tau_{-1}} < s_\tau < s_{\tau_{+1}}, \quad \nexists s\tau' : s_{\tau_{-1}} < s'_\tau < s_\tau, \quad \nexists s''_\tau : s_\tau < s''_\tau < s_{\tau_{+1}}$$

Dies soll formal ausdrücken, dass $s_{\tau-1}$ das Vorgängertupel von s_τ ist es und es kein Tupel im Datenstrom S gibt, welches zu einem Zeitpunkt zwischen τ und τ_{-1} produziert wurde. τ_{-1} bedeutet explizit *nicht*, dass etwa der Zeitstempel vom zugehörigen Tupel betrachtet und mit 1 subtrahiert wird! Analog gilt dies für τ und τ_{+1}.

Diese Semantik ist auch auf die Datenströme anwendbar. $S(\tau)$ sei der Datenstrom, der alle bis zum Zeitpunkt τ produzierten Tupel enthält. Der zeitlich gesehene Vorgänger-Datenstrom $S(\tau_{-1})$ beinhaltet alle Tupel zum Zeitpunkt τ_{-1}, also zum Zeitpunkt, als das Vorgängerelement von s_τ das aktuellste Element im Datenstrom war. Daraus folgt, dass $S(\tau) = S(\tau_{-1}) \cup s_\tau$ gilt. Diese Definition sagt nichts über zwei verschiedene Datenströme bzw. Tupel zweier verschiedener Datenströme aus. Seien etwa zwei Sensoren S und R gegeben, so wäre $\tau_S - 1 < \tau_R < \tau_S$ möglich. Dies wäre etwa der Fall, wenn die Sensoren zeitlich versetzt oder mit einer unterschiedlichen Frequenz messen.

Messen mehrere Sensoren gleichzeitig, so haben diese – auch wenn diese einen zeitlich differenten Messbeginn haben – zum gleichen Zeitpunkt[39] alle das gleiche τ. Es existiert also eine globale, logische Uhr für alle Datenströme[40].

Wird ein Tupel oder ein Datenstrom ausgewertet, das bzw. der zeitlich vor dem Beginn des Datenstromes liegt, so wird die leere Menge als Ergebnis geliefert. Es gilt also sowohl für ein einzelnes Tupel, wie auch für einen Datenstrom: $\forall \tau < 0 : s_\tau = \emptyset, S(\tau) = \emptyset$.

[39] Gleicher Zeitpunkt in Wallclock-Time.

[40] Diese globale, logische Uhr kann zum Beispiel auf Ebene des Sensor Hubs implementiert werden, indem beim Eintreffen eines Messwerttupels das Tupel um die Systemzeit des Sensor Hubs erweitert wird.

Die Abbildung algebraischer Operationen auf Stromdaten wird im Folgenden immer im Vergleich zu materialisierten Datensätzen gezeigt, wie sie etwa in Relationen von Datenbanken vorkommen. Es werden Schwierigkeiten aufgezeigt, die bei der Anwendung auf strömenden Daten entstehen und Lösungen für die Problematiken aufgezeigt.

Im Rahmen der nachfolgend vorgestellten Algorithmen werden Fenster benutzt. Für alle Fenster gilt, dass diese tupelbasiert und nicht zeitbasiert sind. Es ist Ziel, Daten so schnell wie möglich zu verarbeiten und so schnell wie möglich aus dem Puffer freizugeben. Bei zeitbasierten Fenstern bestünde das Problem, dass möglicherweise zu viele im Puffer gehalten werden würden, bevor eine Operation ausgeführt wird und somit ein Pufferüberlauf entstehen würde.

5.2 Selektion

Bei der Selektionsoperation auf Tupel einer Relation, wie sie in einer Datenbank vorkommen kann, wird für alle in der Relation enthaltenen Tupel geprüft, ob ein Attribut ein bestimmtes Prädikat erfüllt. Ist dies der Fall, so wird das Tupel in die Zielrelation übernommen.

S

τ	**Data**
0	10
10	12
20	14

$\sigma_{Data \geq 12}(S)$

S'

τ	**Data**
10	12
20	14

Abb. 14: Selektion auf einer materialisierten Relation

Eine Selektionsoperation auf Tupel einer Relation wird in Abb. 14 dargestellt: Die materialisierte Relation S auf der linken Seite besteht aus den zwei Attributen τ und $Data$ und enthält drei Tupel. Durch Anwenden einer Selektionsoperation auf das Attribut $Data$ der Relation S mit der Selektionsbedingung $Data \geq 12$, entsteht die Zielrelation S', in der das Tupel $S[0, 10]$ der Ausgangsrelation fehlt, da das Tupel das einzige ist, das die Selektionsbedingung nicht erfüllt.

Auf strömenden Daten ist die Selektion ähnlich anwendbar. Zu jedem Zeitpunkt τ soll gelten:

$$S(\tau) = \begin{cases} S(\tau_{-1}) \cup s_\tau, & \sigma_f \text{ erfüllt,} \\ S(\tau_{-1}), & \text{sonst.} \end{cases}$$

wobei σ_f das Selektionsprädikät der Selektionsbedingung σ sei. Veranschaulicht werden soll die Selektionsoperation auf strömenden Daten in Abb. 15. Hier sei ein Sensor Hub dargestellt, der den Datenstrom S den Sensors SEN_S konsumiert. In diesem Datenstrom strömt alle 10 ms ein neuer Messwert ein.

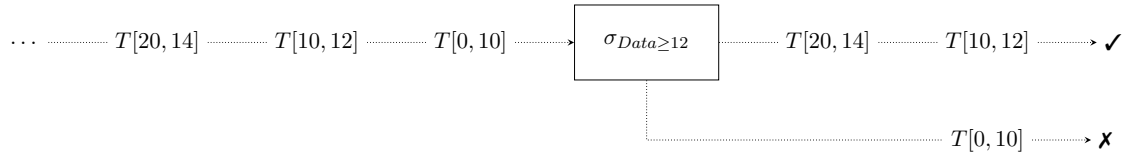

Abb. 15: Selektion auf strömenden Daten

Für den ersten Datensatz zum Zeitpunkt $\tau = 0$ wird nach genannter Semantik der Datenstrom $S(-1)$ ausgewertet, welcher bereits als leere Menge $\emptyset$ definiert wurde. Der durch die Selektion entstanden Ergebnisdatenstrom ist initial also leer. Es trifft zum Zeitpunkt $\tau = 0$ der Wert 10 ein, zum Zeitpunkt $\tau = 10$ der Wert 12 ein und zum Zeitpunkt $\tau = 20$ der Wert 14 ein. Der Sensor Hub wendet eine Selektionsoperation an, die besagt, dass der Attributwert des *Data*-Attributes eines Tupels ≥ 12 sein muss, um weiterverarbeitet zu werden. Das erste Element mit dem Wert 10 trifft ein. Da σ nicht erfüllt ist, gilt laut obiger Formel, dass $S(\tau) = S(\tau_{-1})$ gelten soll, also $S(0) = S(-1)$, was gemäß der Definition $\emptyset$ entspricht. Zum Zeitpunkt $\tau = 0$ wird durch den Sensor Hub entsprechend kein Element in den entstandenen Ergebnisdatenstrom gegeben – das Element mit dem Messwert 10 wurde gefiltert. Zum Zeitpunkt $\tau = 10$ trifft das Element mit dem Wert 12 ein. Da 12 das Selektionsprädikat erfüllt, gilt also $S(\tau) = S(\tau_{-1}) \cup s_\tau$, im, konkreten Fall $S(10) = S(0) \cup T[10, 12]$. Der Zeitpunkt τ_{-1} entspricht in diesem Falle laut formaler Semantik dem Wert 0, da dies der letzte Zeitpunkt war, an dem vor $\tau = 10$ Daten gemessen wurden. Der Datenstrom R(10) besteht aus dem Tupel 12 mit dem Zeitstempel 10. Analog wird der Wert 14 zum Zeitpunkt $\tau = 20$ verarbeitet.

Nach Ende der Operation wurden die Werte 12 und 14 vom Sensor Hub im Stream weitergegeben und der Wert 10 wurde aufgrund des Nichterfüllens des Selektionsprädikates gefiltert.

5.3 Projektion

Eine Projektion findet Anwendung, wenn bei mehrelementigen Tupeln nur Elemente an bestimmten Tupelindexen weiterverarbeitet werden soll. Ein n-Tupel T bestehe im relationalen Falle aus n Attributwerten $a_1, \ldots, a_n$ über dem Schema $A_1, \ldots A_n$. Jeder Attributwert a_i entspricht dabei dem Attribut A_i über dem entsprechenden Schema. Wird

die Projektion auf das Attribut A_i eines Tupels angewendet, so werden alle Attributwerte außer a_i nach der Projektion verworfen. Es ist möglich, auf mehr als ein Element zu projizieren.

S

τ	**Data**
0	10
10	12
20	14

$\pi_{Data}(S)$ S'

Data
10
12
14

Abb. 16: Projektion auf einer materialisierten Relation

Verdeutlicht wird dies in Abb. 16, in der eine Projektion auf die Relation S stattfindet. Dabei wird auf das Attribut *Data* projiziert. In der Zielrelation S' tauchen also nur die entsprechenden Attributwerte des Attributes *Data* auf, nicht aber die Attributwerte des Attributes τ.

Analog anwendbar ist dies auf strömende Daten, die in der Datenstruktur eines Tupels mit mehreren Elementen vorliegen. Formal bedeutet dies:

$$\pi_{P_0,\ldots,P_m}(S(\tau)) = \{T_0[\tau_0, a_{0_0}, \ldots, a_{0_k}, \ldots, a_{0_n}], \ldots, T_\tau[\tau_\tau, a_{\tau_0}, \ldots a_{\tau_k}, \ldots, a_{\tau_n}]\},$$
$$\forall A_{i_k} : A_{i_k} \in \{P_0, \ldots P_m\}, \forall \tau_i : \tau_i \notin \{P_1, \ldots P_m\}, i \in \{0, \ldots, \tau\}, k \in \{0, \ldots, n\}.$$

Der Stream S enthält zum Zeitpunkt τ die Elemente: $T_0, \ldots T_\tau$. Die Elemente sind dabei $n+1$-Tupel, die jeweils den Zeitpunkt[41] τ des Elementes enthalten, sowie n Datenattribute. Das Element τ über dem Schema der Zeitdomäne $\mathcal{T}$ wird im Gegensatz zur Projektion auf einer materialisierten Relation gesondert behandelt, da τ das einzige Attribut ist, das nicht herausprojiziert werden darf. Die zwingende Notwendigkeit des Vorhandenseins von τ wird im Abschnitt 5.4 (ab S. 43) erläutert.

Es findet eine Projektion auf $S(\tau)$ statt, bei der auf die Projektionsattribute $P_0, \ldots P_m$ projiziert wird. Dabei werden von den Tupeln $T_0, \ldots, T_\tau$ diejenigen Elemente a_{i_k} über dem Schema A_{i_k} in den Zielstream übernommen, bei denen die Projektionsbedingung $A_{i_k} \in \{P_0, \ldots, P_m\}$ erfüllt sind. Alle anderen Tupelelemente werden pro Tupel verworfen, also herausprojiziert. Die Bedingung $\forall \tau_i : \tau_i \notin \{P_0, \ldots P_m\}$ soll formal garantieren, dass kein Wert über dem Schema der Zeitdomäne herausprojiziert wird.

[41] Dies kann je nach Implementierung etwa der Generierungszeitpunkt des Tupels sein (τ wird auf Sensorebene generiert) oder der Zeitpunkt, an dem das Tupel am Sensor Hub eintrifft (τ wird auf Sensor-Hub-Ebene generiert).

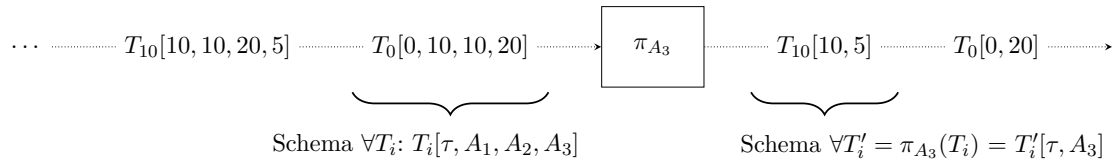

Abb. 17: Projektion auf strömenden Daten

Beispielhaft lässt sich die Projektion an Abb. 17 erläutern. Es existieren die zwei Tupel T_0 und T_{10}, die entsprechend zu den Zeiten $\tau = 0$ bzw. $\tau = 10$ generiert wurden. Schematisch handelt es sich um Viertupel, die jeweils als erstes Element den Zeitstempel τ sowie drei Datenattribute A_1, A_2 und A_3 tragen. Vor der Projektion ist erkennbar, dass im Tupel alle vier Attribute vorhanden sind. Nach der Projektion auf das Attribut A_3 sind nur noch die Attribute τ und A_3 vorhanden und die Attribute A_1 und A_2 wurden verworfen. Aufgrund der getroffenen Festsetzung, dass τ bei der Projektion nicht elimiert werden darf, bleibt es im Tupel, auch wenn τ nicht explizit in der Projektionsbedingung angegeben wurde.

Soll die Komponente eines Fensters mit einbezogen werden, etwa weil die Projektion nicht für den gesamten Stream angewendet werden kann oder soll, so ist die formale Definition für die Projektion abzuwandeln. Gegeben sei ein Sliding Window, welches in den Grenzen $\tau_{Beginn} = y$ und $\tau_{Ende} = z$, $y < z$ liegt. Folgende formale Definition gilt:

$$\pi_{P_0,\ldots,P_m}(R(z)) = \{T_y[\tau_y, a_{y_0}, \ldots, a_{y_k}, \ldots, a_{y_n}], \ldots, T_z[\tau_z, a_{z_0}, \ldots a_{z_k}, \ldots, a_{z_n}]\},$$
$$\forall A_{i_k} : A_{i_k} \in \{P_0, \ldots P_m\}, \forall \tau_i : \tau_i \notin \{P_1, \ldots P_m\}, i \in \{y, \ldots, z\}, k \in \{0, \ldots, n\}.$$

5.4 Verbund (Join)

Die *Verbundoperation* (*Join*) beschreibt die Verknüpfung zweier Tupel über eine Attributmenge, bei welcher die Attribute jeweils die gleichen Attributwerte besitzen. Diese Operation ist erheblich komplexer und eingeschränkter realisierbar auf strömenden Daten im Vergleich zu materialisierten Daten. Die Begründung, Problematiken und Lösungen dieser Komplexität sollen im Folgenden dargelegt werden.

Gegeben seien zwei materialisierte Relationen R und S, wie sie in einer Datenbank auftauchen können. Diese seien wie in Abb. 18 definiert.

R

τ_R	$\mathbf{Data}_R$
0	10
10	12
20	14

S

τ_S	$\mathbf{Data}_S$
0	10
10	14
20	14

$R \bowtie_{Data_R = Data_S} S$

T

τ_R	$\mathbf{Data}_R$	τ_S	$\mathbf{Data}_S$
0	10	0	10
20	14	10	14
20	14	20	14

Abb. 18: Verbund zweier materialisierter Relationen

Findet eine Verbundoperation über R und S mit der Verknüpfungsbedingung $Data_R = Data_S$ statt, so entsteht die in Abb. 18 dargestellte Relation T. Es wird geprüft, welche Attributwerte in $Data_R$ und $Data_S$ identisch sind. Bei Tupelkandidaten, bei denen diese Bedingung erfüllt ist, bildet die Verknüpfung beider Tupel das Ergebnistupel in T. Die formale Definition dafür ist nach [SSH08, S. 99 ff.]:

$$r_1 \bowtie r_2 := \{t | t(R_1 \cup R_2) \wedge \exists t_1 \in r_1 : t_1 = t(R_1) \wedge \exists t_2 \in r_2 : t_2 = t(R_2)\}$$

Dabei sind r_1 und r_2 die Relationen über dem Relationenschema R_1 bzw. R_2, die verbunden werden sollen und t_1 und t_2 die Tupel in den entsprechenden Relationen.

Für die Realisierung von Joins werden in Datenbanksystemen Implementierungen wie der *Nested-Loop-Join*, der *Merge-Join* oder der *Hash-Join* genutzt (vgl. [SSH11, S. 367 ff.]). Diese Operationsmöglichkeiten haben alle eine Gemeinsamkeit: Während der Berechnung lesen sie die gesamten Relationen, über die eine Verbundoperation ausgeführt werden soll, d.h. es findet eine blockierende[42] Operation statt, bis alle Daten auf mögliche Verbundpartner geprüft wurden. Im Abschnitt 2.1 (S. 15 f.) wurde bereits herausgestellt, dass blockierende Operationen auf Datenströmen aufgrund ihrer potentiell unbegrenzten Länge im Allgemeinen nicht ausführbar sind und eine alternative Algorithmik für ebensolche Operationen gefunden werden muss. Entsprechend sind die genannten Verknüpfungstechniken, welche in Datenbankensystemen vorkommen, im Allgemeinen nicht unadaptiert anwendbar.

Um die Eigenschaft von Sensoren, Daten nur in begrenzter Menge speichern zu können, zu beachten und um die angestrebte Verarbeitung in nahezu Echtzeit zu bewältigen, werden unter anderem bereits vorgestellte Techniken der Fensterbildung eingesetzt. Die Fenstergröße kann dabei je nach Leistungsfähigkeit bezüglich der Speicherung eines Sensors variieren. Ziel soll es sein, Sensordaten zweier Datenströme zu verknüpfen.

Aufgrund der geringen Datenmenge, die in einem Fenster gehalten werden kann und aufgrund der möglichen Heterogenität von Sensordaten (der Messwert des Luftdruckes, etwa

[42] Im allgemeinen Fall, wenn sortierte Relationen nicht vorausgesetzt sind.

1000 hPa im Vergleich zur Temperatur, etwa 20 °C) ist es im Allgemeinen nicht möglich, über Parametermesswerte[43] mehrerer Sensoren einen Verbund zu bilden. In Spezialfällen funktioniert dies, etwa wenn garantiert ist, dass zwei Sensoren den gleichen Wert mit leichten Differenzen messen, im Allgemein seien Messwerte jedoch nicht voraussagbar. Lediglich über das Zeitattribut τ kann eine Verknüpfung geschaffen werden, da τ im Optimalfall bei gleichzeitig gemessenen Sensordaten identisch oder nahezu identisch ist. Dies leitet sich aus der getroffenen formalen Definition ab, dass alle Sensoren zum Zeitpunkt τ den gleichen Wert für τ haben[44]. Diese Eigenschaft stellt die Begründung für das offen gelassene Problem dar, weshalb τ niemals in der Projektion eliminiert werden darf (vgl. Abs. 5.3). Das τ dient als einzig zuverlässiges Verbundattribut mit Einschränkungen, die nachfolgend gezeigt werden sollen.

Es sollen im Folgenden zwei Algorithmen erläutert werden, durch die der Join von Sensordaten innerhalb eines Fensters möglich ist. Diese Techniken sollen *Fuzzy-Merge-Join* und *Minimal-Delta-Join* heißen. Zusätzlich wird je eine Abwandlung der Techniken vorgestellt, die je nach Einsatzzweck Anwendung finden können.

5.4.1 Fuzzy-Merge-Join

Der Fuzzy-Merge-Join ähnelt der Idee, die in [KNV03] vorgestellt wurde. Für den Fuzzy-Merge-Join seien zwei Streams S mit Tupeln über dem Schema $S[\tau_S, A_S]$ und T mit Tupeln über dem Schema $T[\tau_T, A_T]$ gegeben[45], bei denen über das jeweilige Zeitattribut τ ein Verbund durchgeführt werden soll. Folgende Grafik soll beispielhaft veranschaulichen, wie der Algorithmus arbeitet.

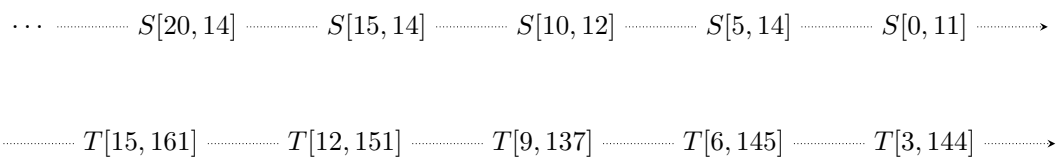

Abb. 19: Zwei Datenströme, die mittels Fuzzy-Merge-Join verknüpft werden sollen

Es soll ein Verbund über die die Zeitattribute von S und T – dies sind τ_S und τ_T – vollzogen werden, also $S \bowtie_{\tau_S=\tau_T} T$. Dabei soll gelten, dass S der Sensor ist, der mit einer niedrigeren Abtastfrequenz misst, d.h. $(\tau_S - (\tau_S - 1)) > (\tau_T - (\tau_T - 1))$ soll gelten. Die Begründung dieser Festlegung wird später ersichtlich. Diese Randbedingung gilt im Beispiel in Abb. 19, da im Stream S alle fünf Zeiteinheiten und im Stream T alle drei Zeiteinheiten ein Tupel

[43] Mit Parameter sind Eigenschaften gemeint wie Luftdruck, Helligkeit, Temperatur, etc. Als Messwerte sind hier die Attributwerte der Parameter gemeint.

[44] Etwa die gleiche Wallclock-Time.

[45] Der Einfachheit halber haben die Streams im Schema nur ein Attribut. Die Attributzahl ist je Stream im Allgemeinen beliebig.

durch den Datenstrom strömt.

Der Stream mit der geringeren Messfrequenz heiße dabei *dominierender Stream*, da der Join von diesem Stream ausgehen wird. Im Beispiel in Abb. 19 ist dies der Stream S.

Der Algorithmus geht wie folgt vor: Für das erste unverarbeitete Element von S wird ein passender Verbundpartner gesucht. Dabei wird aus T das Tupel verknüpft, bei dem das τ den gleichen Wert hat oder – falls ein solches τ nicht vorhanden ist – das nächstgrößere τ darstellt. Da die Werte als zeitlich geordnet vorliegend vorausgesetzt werden, ist es nicht notwendig, alle Tupel zu betrachten. Die genaue Anzahl der zu betrachtenden Tupel wird später dargelegt.

Im obigen Beispiel soll das erste Tupel von S – $S[0, 11]$ – mit einem Tupel aus T verknüft werden. Der erste Tupelkandidat ist $T[3, 144]$. Da zum Zeitpunkt $\tau = 3$ $\tau_T > \tau_S$, konkret $3 > 0$, gilt und kein Tupel im Stream von T mit $\tau_T < 3 \wedge \tau_T > 0$ existiert, werden diese beiden Tupel verknüpft, wie in Abb. 20 ersichtlich.

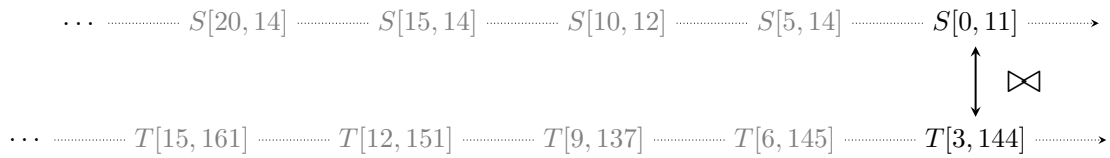

Abb. 20: Fuzzy-Merge-Join mit gefundenem Tupelpaar

Der Zeitstempel ist nicht identisch, daher handelt es sich auch nur um einen Join, der als *fuzzy* bezeichnet wird, d.h., mögliche Abweichungen der beiden Zeitstempelwerte werden geduldet. An dieser Stelle können nun $S[0, 11]$ und $T[3, 144]$ aus dem Speicher freigegeben werden, da sie verarbeitet wurden.

Im nächsten Schritt wird das nächste Tupel von S ausgewertet und es wird analog mit dem nächsten Tupeln von T verknüpft: $S[5, 14] \bowtie T[6, 145]$. Der dritte Join von $S[10, 12]$ bildet dabei einen Sonderfall, wie in Abb. 21 ersichtlich: in T wird ein Tupelpaar – $T[9, 137]$ – übersprungen, zu dem kein passendes Tupel zur Verknüpfung gefunden wird. Daher kann es nach dem – insgesamt dritten – Verbund von $S[10, 12]$ und $T[12, 151]$ gelöscht werden.
An dieser Stelle ist die Bedeutung des dominanten Streams mit geringerer Abtastfrequenz ersichtlich: Wäre T an dieser Stelle der dominante Stream, so würden τ_S und τ_T nach einiger Zeit bezogen auf das τ deutlich auseinander driften und es würde sich kein verlässlicher

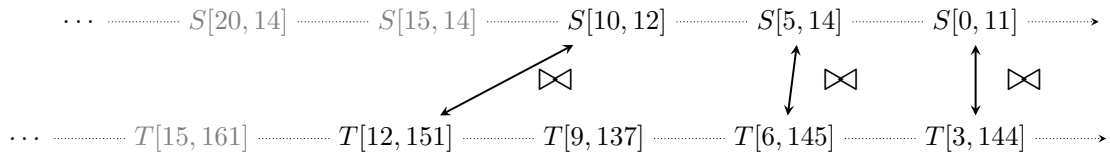

Abb. 21: Fuzzy-Merge-Join mit drittem gefundenen Tupelpaar

zeitlicher Zusammenhang mehr bilden lassen. Wie in Abb. 22 dargestellt, würde nach dem Algorithmus $T[12, 151]$ etwa mit $S[20, 14]$ verknüpft werden und damit schon eine zeitliche Differenz von acht Zeiteinheiten aufweisen.

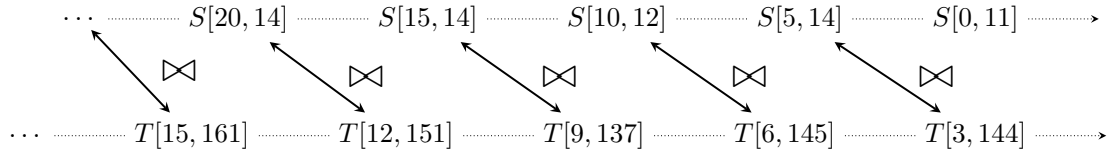

Abb. 22: Fuzzy-Merge-Join mit auseinanderdriftendem τ

Während der Ausführung des Algorithmus wurde eine implizite Annahme getroffen. Es darf ein Tupel nur einmal mit einem anderen Tupel verknüpft werden. Dies kann je nach Nutzungszweck Vor- und Nachteile haben. Würde ein Sensor S_1 mit einer Abtastfrequenz alle 100 Zeiteinheiten und ein Sensor S_2 mit einer Abtastfrequenz alle zwei Zeiteinheiten messen, so würde dies nach einer Verknüpfung bedeuten, dass man ein Tupel von S_2 bis zu 50 Mal mit S_1 joinen könnte und dadurch die Relevanz von S_2 deutlich erhöht. An dieser Stelle wurde die Relevanz des dominanten Streams, also die des Streams mit geringerer Abtastrate über die des nicht-dominanten Streams gestellt und es ist valide, zu behaupten, dass damit die Relevanz der Daten des nicht-dominanten Streams künstlich verringert wurde. Durch die Tatsache, dass Sensor Hubs eine begrenze Speicherkapazität haben und Daten so schnell wie möglich verarbeitet werden sollen, fiel die Entscheidung auf diese Variante, um Daten so früh wie möglich zu verarbeiten und Speicher freizugeben und daher die Verknüpfung mit nur einem Tupel zu erlauben.

Aus Implementierungssicht ist diese Technik als Sliding-Window über den dominanten Stream S mit der Fenstergröße 1 anzusehen. Über den nicht-dominanten Stream T wird ebenfalls ein Sliding-Window gelegt, der sich wie folgt berechnen lässt:

$$\text{Windowsize}_T = \left\lceil \frac{S_\tau - S_{\tau-1}}{T_\tau - T_{\tau-1}} \right\rceil$$

Hier wird die Abtastfrequenz des dominanten Streams durch die Abtastfrequenz des nicht-dominantdominanten Streams dividiert und dann aufgerundet. Im laufenden Beispiel wäre dies $\left\lceil \frac{5}{3} \right\rceil = 2$. Im nicht-dominanten Stream mussten also maximal zwei Elemente

betrachtet werden, wobei das erste davon das erste unverarbeitete Element ist, gefolgt vom nächsten Element. Im Allgemeinen sind dies immer die w unverarbeiteten Elemente, über dem das Sliding Window liegt, wobei $w = Windowsize_T$.

Für den Fuzzy-Merge-Join soll anhand diese Algorithmus eine formale Semantik entworden werden. Diese sieht wie folgt aus:

$$S(\tau) \bowtie T(\tau) = \forall s \in S(\tau), t \in T(\tau) :$$
$$ST'[\tau_S, s_0, \ldots, s_n, t_0, \ldots, t_m],$$
$$S(\tau) = S(\tau) - S[\tau_S, s_0, \ldots, s_n],$$
$$T(\tau) = T(\tau) - T[\tau_{T_i}, t_{0_i}, \ldots, t_{m_i}], \forall i : i \in \mathcal{T}_t \wedge i \leq \tau$$
$$\tau_s \leq \tau_t \wedge \forall \tau_T \in \mathcal{T}_T : \nexists \tau'_T \in \mathcal{T}_T : \tau_S < \tau'_T < \tau_T$$

Die Datenströme S und T werden betrachtet. Für alle Tupel s in S und t in T soll gelten: Das Ergebnis des Verbundes sei das Tupel ST', welches als Schema den Verbund beider Schemata der Datenströme S und T, mit Ausnahme des wegfallenden Zeitattributes des nicht-dominanten Datenstroms, besitzt. Für den Datenstrom S zu jedem beliebigen Zeitpunkt τ soll gelten: Wurde ein Verbundpartner gefunden, so kann das Tupel nicht erneut als Verbundpartner auftreten. Der Algorithmus wartet außerdem so lange, bis ein Verbundpartner gefunden wurde. Es gilt also für $S(\tau)$ nach dem Join: $S(\tau) = S(\tau) - S[\tau_S, s_0, \ldots, s_n]$. Per Definition ist dies immer die leere Menge[46]. Für $T(\tau)$ gibt es eine ähnliche Bedingung. Hier muss nur zusätzlich beachtet werden, dass alle Tupel freigegeben werden, die entweder in einer Verbundoperation verknüpft wurden oder die während der Laufzeit des Algorithmus ignoriert wurden (vgl. beispielhaft Tupel T[9, 137] in Abb. 21, S. 47). Dies wird durch $T(\tau) = T(\tau) - T[\tau_{T_i}, t_{0_i}, \ldots, t_{m_i}], \forall i : i \in \mathcal{T}_t \wedge i \leq \tau$ beschrieben: Alle Tupel in T, die einen kleineren Timestamp $\tau_T \in \mathcal{T}$ tragen als das τ, zu dem der Verbund ausgeführt wird, werden aus $T(\tau)$ entfernt. Die Randbedingung $\tau_s \leq \tau_t \wedge \forall \tau_T \in \mathcal{T}_T : \nexists \tau'_T \in \mathcal{T}_T : \tau_S < \tau'_T < \tau_T$ garantiert, dass S mit dem nächstmöglichen Element aus T nach dem Algorithmus verbunden wird – und nicht ein kleineres Tupel existiert, mit dem der Verbund hätte durchgeführt werden können und müssen.

Für den Fuzzy-Merge-Join wurde ein datengetriebenes Window definiert, das im dominanten Stream die Größe 1 und im nicht-dominanten Stream eine Fenstergröße besitzt, dessen Berechnung gezeigt wurde. Aufgrund der Arbeitsweise des Algorithmus ist ein zeitgetriebenes Window nicht einsetzbar. Es wurde bei der Konzeption dieses Algorithmus

[46] Dies gilt, da der Algorithmus immer so lange wartet, bis ein Verbundpartner gefunden wird. Nach dem Verbund wird dann das Tupel s_τ zum Zeitpunkt τ eliminiert. Das bedeutet, $S(\tau)$ entspricht nach jedem Verbund der leeren Menge. Dies funktioniert nur, da S eine langsamere Messfrequenz hat (dominierender Stream) und daher zu jedem Zeitpunkt τ ein Verbundpartner garantiert ist.

Wert darauf gelegt, dass jedes Tupel so schnell wie möglich verarbeitet und anschließend aus dem Speicher freigegeben werden kann. Entsprechend wurde die Variante verworfen, dass ein Tupel mehrfach verknüpft werden kann. Durch die Entscheidung, dass der Datenstrom mit der geringeren Frequenz als Ausgangspunkt des Verbundes gilt, wird garantiert, dass zu jedem Zeitpunkt, bei dem beim dominierenden Stream ein Tupel strömt, ein passender Verbundpartner gefunden wird. Veränderte Implementierungsmöglichkeiten des Fuzzy-Merge-Joins sind denkbar.

5.4.2 Bufferloser Fuzzy-Merge-Join

Der im vorherigen Abschnitt konzeptuell beschriebene Fuzzy-Merge-Join ist hinsichtlich bestimmter Punkte effizient: Die Elemente des dominierenden Streams müssen nicht gepuffert werden, da aufgrund der geringeren Datenfrequenz gegenüber dem nicht-dominanten Streams zu jedem Zeitpunkt ein Verbundpartner garantiert ist. Im nicht-dominanten Stream muss nur eine bestimmte Anzahl an Elementen zwischengespeichert werden, die sich wie in Abschnitt 5.4.1 beschrieben aus dem Quotienten der Frequenzen beider Datenströme berechnen lassen.

Der bufferlose Fuzzy-Merge-Join stellt eine Abwandlung des Fuzzy-Merge-Joins dar, der garantiert, dass zu jedem Zeitpunkt maximal ein Tupel in jedem Stream gepuffert werden muss. Dieses Tupel wird als potentieller Verbundkandidat dienen, darüber hinaus ist ein Vorhalten weiterer Tupel in keinem der Streams vonnöten. Diese Eigenschaft sei als *bufferlos* bezeichnet[47]. Der Nachteil der bufferlosen Fuzzy-Merge-Join Variante wird im Gegensatz zum Fuzzy-Merge-Join sein, dass nicht für jedes Tupel des dominanten Datenstroms ein Verbundpartner garantiert werden kann. Ebenso wie der Fuzzy-Merge-Join ist dieser Verbundalgorithmus datengetrieben.

Wie beim Fuzzy-Merge-Join sind zwei Datenströme S und T gegeben, bei denen gilt, dass S der Stream mit geringerer Messfrequenz sei. Der Unterschied im Algorithmus im Vergleich mit dem Fuzzy-Merge-Join besteht darin, dass der Datenstrom mit der höheren Messfrequenz, hier T, der dominierende Stream sei. Ein Verbund wird über dem Tupel des nicht-dominierenden Streams realisiert, der den gleichen oder nächstkleineren Timestamp trägt. Durch die geringere Messfrequenz von S gilt zudem unabhängig vom Timestamp, dass es zu jedem Zeitpunkt τ maximal ein Element im nicht-dominanten Stream gibt,

[47] Der Zwang des Pufferns des einzigen Tupels in jedem Stream ist unvermeidlich, da ein zu verarbeitendes Element immer im Puffer des Sensor Hubs liegen muss. Eine Verarbeitung des Tupels außerhalb des Puffers ist nicht möglich.

welcher der Verbundpartner sei[48]. Gibt es zu einem beliebigen Zeitpunkt τ für ein Tupel aus dem dominierenden Stream T keinen Verbundpartner, so wird das Tupel aus dem Puffer von T gelöscht.

Zur Visualisierung des Ergebnisses seien folgende Datenströme in Abb. 23 mit ihren zum Zeitpunkt τ auftauchenden Elementen gegeben.

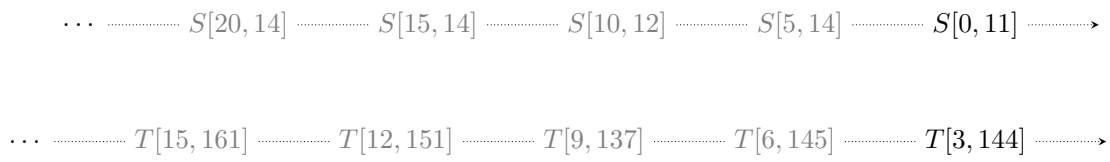

Abb. 23: Zwei Datenströme, die mittels bufferlosen Fuzzy-Merge-Join verknüpft werden sollen

Zum Zeitpunkt $\tau = 3$ ist im dominanten Datenstrom T das Tupel $T[3, 144]$ und im nicht-dominanten Datenstrom S das Tupel $S[0, 11]$ vorhanden. Es soll nun für das Tupel aus T ein Verbundpartner gefunden werden. Dies ist das Tupel, welches zum genannten Zeitpunkt in S vorhanden ist. Es gilt also, dass über die Tupel $T[3, 144]$ und $S[0, 11]$ der Verbund realisiert wird, wie in Abb. 24 ersichtlich.

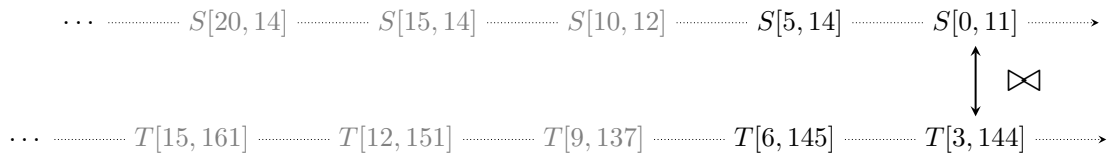

Abb. 24: Bufferloser Fuzzy-Merge-Join mit gefundenem Tupelpaar

Ferner ist in Abb. 24 die Zeit auf $\tau = 6$ fortgeschritten. Zu diesem Zeitpunkt soll für das Tupel $T[6, 145]$ ein Verbundpartner gefunden werden. Der passende Partner ist gemäß des Algorithmus das Tupel $S[5, 14]$, da dies das Tupel aus dem nicht-dominanten Datenstream ist, das den gleichen oder nächstkleineren Zeitstempel bezogen auf das Tupel aus T hat. Entsprechend findet auch hier ein Join zwischen den Tupeln $T[6, 145]$ und $S[5, 14]$ statt, wie in Abb. 25 dargestellt.
In Abb. 25 tritt weiterhin der Fall ein, dass die Zeit auf $\tau = 9$ fortschreitet. Zu diesem Zeitpunkt existiert im Stream S noch kein passender Tupelkandidat. Das Tupel $T[9, 137]$ wird deswegen verworfen und nicht in einen Verbund mit aufgenommen. Der Algorithmus

[48] Dies gilt nur, wenn der nicht-dominante Datenstrom zum Zeitpunkt der Verbundoperationen keine Tupel beinhaltet. Sind im Datenstrom schon Tupel vorhanden, etwa weil der Stream schon einige Zeit Messwerte strömen lässt, so gelten die bis zur Verbundoperation existenten Tupel im Stream als nicht qualifiziert für den Verbund und werden nicht betrachtet.

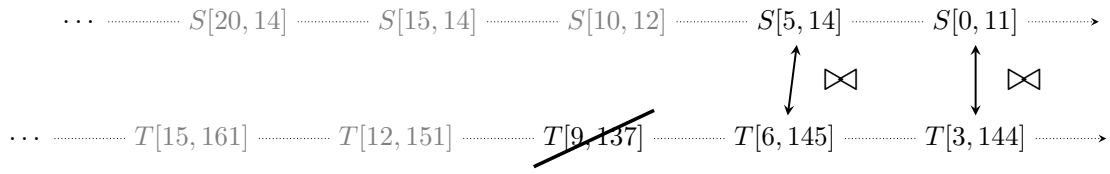

Abb. 25: Fuzzy-Merge-Join mit zweitem gefundenen Tupelpaar ohne Joinkandidat

arbeitet entsprechend weiter und verbindet zum Zeitpunkt $\tau = 12$ die Tupel $T[12, 151]$ und $S[10, 12]$ und zum Zeitpunkt $\tau = 15$ die Tupel $T[15, 161]$ und $S[15, 14]$. Das Ergebnis des Algorithmus nach dem Zeitpunkt $\tau = 15$ ist in Abb. ersichtlich.

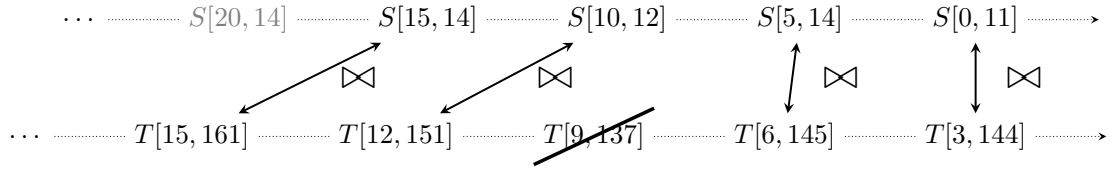

Abb. 26: Fuzzy-Merge-Join mit abgearbeitetem dominanten Stream nach $\tau = 15$

Mit dem bufferlosen Fuzzy-Merge-Join wurde eine Abwandlung des Fuzzy-Merge-Joins gefunden, die eine effizientere Nutzung des Puffers möglich macht. Durch die Notwendigkeit des Vorhaltens von genau einem Tupel pro Puffer – die durch die Verbundoperation zu verbindenden Tupel – wurde ein datengetriebener Algorithmus erläutert, der für Komponenten mit starker Ressourcenbeschränkung geeignet ist.

5.4.3 Minimal-Delta-Join

Beim Fuzzy-Merge-Join und beim bufferlosen Fuzzy-Merge-Join entsprach der Verbundpartner strikt dem Tupel, das immer den nächstgrößeren bzw. den nächstkleineren Zeitwert aufwies. Dies ist nicht immer sinnvoll, wie in Abb.26 ersichtlich: Hier wurde das Tupel $T[3, 144]$ mit dem Tupel $S[0, 11]$ verbunden, da etwa im Algorithmus für den bufferlosen Fuzzy-Merge-Join strikt festgelegt wurde, dass das Tupel mit dem nächstkleineren oder gleichen Zeitstempel den Verbundkandidaten darstellt. Intuitiver wäre es an dieser Stelle gewesen, abweichend von den beiden genannten Algorithmen die Tupel $S[5, 14]$ mit $T[3, 144]$ zu verbinden anstatt das Tupel $S[0, 11]$, da die Zeitwerte der beiden Tupel näher zusammenliegen. Durch die potentiell unterschiedlichen Werte der beiden Datenströme wurde bisher versucht, den Verbund durch einen Ähnlichkeitsverbund („fuzzy“) zu konzipieren, wobei die Ähnlichkeit durch das nächste Tupel im Datenstrom definiert war. Dies geschah nach statischen Regeln. Es wurden nie *mehrere* Tupel *innerhalb* eines Streams miteinander verglichen, wodurch ein Verlust der Optimalität in der Kandidatenfindung für den Verbund in Kauf genommen wurde, wie an den gerade genannten Tupeln beispielhaft ersichtlich.

Es soll ein Ähnlichkeitsmaß definiert werden, welches zur Lösung der optimalen Kandidatenfindung beiträgt. Die absolute Differenz der Zeitwerte zweier Tupel sei als *Delta* definiert. Für die Berechnung sei eine Funktion $\Delta(T_1, T_2)$ definiert[49], die das entsprechende Delta berechnet und zurückgibt. Für ein Tupel t aus dem Datenstrom T ist ein Tupel s aus dem Datenstrom S genau dann der optimale Verbundpartner, wenn für die gesamte Laufzeit des Algorithmus gilt, dass das Delta für s mit einem anderen Tupel t minimal ist. Es wird also das Tupel aus T bestimmt, welches die geringste absolute zeitliche Differenz zu s hat.

Konzeptionell besitzt jeder Stream ein Sliding Window mit der Mindestfenstergröße 2. Außerdem besitzt jeder Stream einen Zeiger, der auf das nächste Tupel zeigt, das in einen Verbund aufgenommen werden soll. Soll ein Tupel s aus S mit einem Tupel t aus dem Datenstrom T verbunden werden – an dieser Stelle gilt, dass der Zeiger von S auf s zeigt – so gilt, dass das Sliding Window des Streams T die zwei Tupel umfasst, sodass ein Zeitstempel eines Tupels t_1 aus T im Fenster einen kleineren oder gleichen Zeitstempel hat, als bzw. wie s und das andere Tupel t_2 aus T im Fenster einen größeren Zeitstempel hat als s. Zwischen s und t_1 sowie zwischen s und t_2 werden die Deltas gebildet. Die Tupel, die zueinander das kleinere Delta haben, werden in den Verbund aufgenommen. Ist das Delta identisch, so wird das Tupel aus t mit dem kleineren Zeitstempel verbunden.

Gibt es mehrere Tupel mit dem gleichen Zeitstempel in einem Datenstrom, so werden alle Tupel mit diesem Zeitstempel, sofern mindestens eins davon im einem Window eines Datenstroms liegt, mit in das Window einbezogen. Dies ist die Begründung dafür, warum das Sliding Window eine Mindestgröße hat. Anwendungsbeispiel wäre hierbei eine vorausgegangene *Union*-Operation, bei der mehrere Tupel eines Datenstroms den gleichen Zeitwert haben können. Dies wäre bei der Semantik des (bufferlosen) Fuzzy-Merge-Joins nicht erlaubt, bei Minimal-Delta-Join dagegen schon. Entsprechend werden gegebenenfalls alle entsprechenden Tupel mit diesem Zeitstempel verknüpft, sofern alle Elemente mit diesem Zeitstempel im Datenstrom optimale Tupelkandidaten sind.

Es werden die beiden Elemente, auf denen die Zeiger von S und T zeigen, verglichen und für das Element mit dem kleineren Zeitstempel wird der nächste Verbundpartner gesucht. Haben die beiden Elemente, auf die die Zeiger zeigen, den gleichen Zeitstempel, so gilt, dass das Element aus dem Datenstrom Vorrang hat, dessen Datenstrom zuletzt Ausgangspunkt für den Verbund war. Die Dominanz der Datenströme wechselt also nach bestimmten Regeln.

[49] Mathematisch ist die Delta-Funktion definiert als $\Delta(T_1, T_2) = |\tau_{T_1} - \tau_{T_2}|$.

Nach einem Verbund wird der Zeiger vom Datenstrom, von dem der letzte Verbund ausging, auf das nächste Element gesetzt. Dabei gibt es folgende Randbedingungen[50]:

- Haben die Tupel s und t, auf die die Zeiger von S und T zeigen, den gleichen Zeitwert, so werden *beide* Zeiger auf das nächste Element gesetzt. *(Randbedingung 1)*
- Gilt die eben genannte Bedingung, geht der Verbund vom Datenstrom S aus und es existieren mehrere Elemente in T, die den gleichen Zeitwert haben, wie das Tupel aus S, so wird der Zeiger auf T auf das Element gesetzt, das den nächstgrößeren Zeitwert hat, als das Tupel aus S, das in den Verbund aufgenommen wurde. *(Randbedingung 2)*
- Wurde s mit t verbunden, so darf t nicht erneut mit s verbunden werden, wenn die Dominanz der Datenströme wechselt. Ein Verbundpaar soll nur einmal gefunden werden. *(Randbedingung 3)*

Diese Bedingungen sind notwendig, um einen mehrfachen Verbund gleicher Tupel zu unterbinden[51]. Tupel, über die ein Sliding Window bereits herübergeglitten ist, werden aus dem entsprechenden Puffer entfernt.

An folgendem Beispiel soll der Algorithmus veranschaulicht werden. Der Übersichtlichkeit halber werden hier nur die Zeitstempel betrachtet. Der Verbund der anderen Elemente des Tupels (Nutzdaten) erfolgt wie beim Fuzzy-Merge-Join.

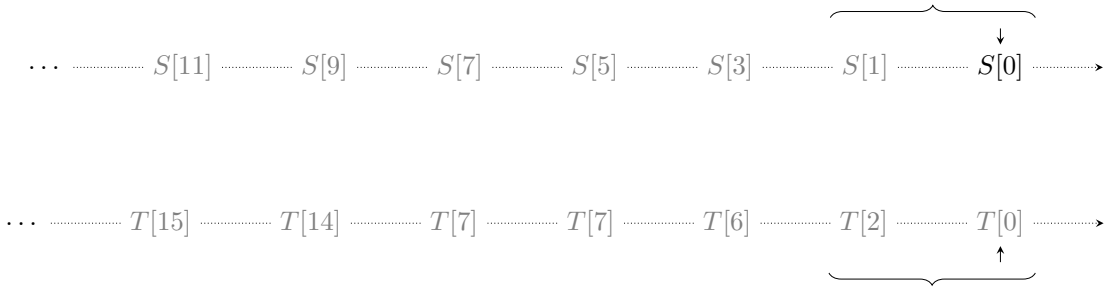

Abb. 27: Zwei Datenströme, die mittels Minimal-Delta-Join verknüpft werden sollen

In Abbildung 27 sind zwei Datenströme dargestellt, die jeweils sieben Tupel beinhalten. Initial wurden die Zeiger beider Streams – symbolisiert durch Pfeile über bzw. unter dem

[50] Die Randbedingungen werden aufgrund späterer Referenzierung benannt.
[51] Dies wird später an einem Beispiel erläutert.

entsprechenden Tupel – auf jeweils das älteste Element im Stream gesetzt und die Sliding Windows – symbolisiert durch eine Klammer über bzw. unter den beiden entsprechenden Tupeln – auf die beiden jeweils ältesten Elemente gesetzt. Ohne Beschränkung der Allgemeinheit ist $S[0]$ im Beispiel das Tupel, mit dem als erstes der Verbund durchgeführt wird[52].

Für $S[0]$ wird ein passender Verbundkandidat gesucht. Das Sliding Window über den Stream T ist bereits an der richtigen Position. Es wurde definiert, dass, sofern ein Tupel in T den gleichen Zeitstempel hat wie das zu verknüpfende Element in S, das erste Element des Sliding Windows das mit dem geringsten Zeitstempel darstellt. Dies ist hier der Fall. Das andere Element im Sliding Window ist das nächstgrößere Element in T, hier $T[2]$. Es werden die Deltas der beiden Kandidaten gebildet. $\Delta(S[0], T[0]) = 0$ und $\Delta(S[0], T[2]) = 2$. Das Delta von $S[0]$ und $T[0]$ ist das minimale; entsprechend werden diese beiden Tupel verbunden. Der Pointer von S wird auf das nächste Element, hier $S[1]$, gesetzt. Da $S[0]$ und $T[0]$ die gleichen Zeitwerte hatten, wird nun auch der Pointer von T auf $T[2]$ gesetzt, da ansonsten der Verbund nun von $T[0]$ ausgehen würde und $S[0]$ erneut und damit fehlerhaft doppelt in den Verbund aufgenommen werden würde (vgl. Randbedingung 1, S. 53). Die Verbundoperation befindet sich nun im Zustand, der in Abb. 28 dargestellt ist.

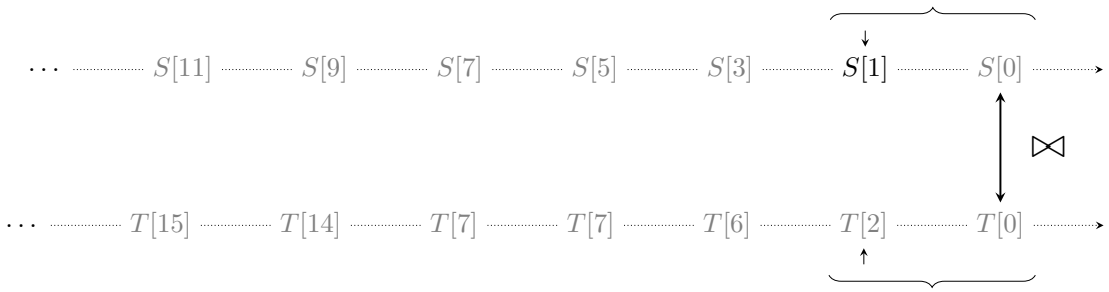

Abb. 28: Minimal-Delta-Join mit gefundenem Tupelpaar

Es werden nun die Zeitwerte der Tupel verglichen, auf die die beiden Zeiger verweisen. Da der Zeitwert von $S1$ geringer ist, soll nun ein Verbundkandidat für dieses Tupel gefunden werden. Das Fenster über T muss nicht verschoben werden, da der kleinere Zeitwert aus T 0 und der größere Zeitwert aus T 2 ist. Da $\Delta(S_1, T_0) = \Delta(S_1, T_2)$ gilt, wird per Definition das Element mit dem kleinen Zeitstempel verwendet. $T[0]$ ist also erneut Tupelkandidat und wird in den Verbund mit $S[1]$ aufgenommen. Der Zeiger von S verschiebt sich auf $S[3]$. Es werden erneut die Zeitwerte der Tupel verglichen, auf die die beiden Zeiger verweisen. $T[2]$ qualifiziert sich als Ausgangspunkt für den nächsten Verbund und das Fenster von S verschiebt sich über die Datenwerte $S[1]$ und $S[3]$, da der Zeitwert von $T[2]$ zwischen den

[52] Ein Start mit $T[0]$ wäre an dieser Stelle ebenfalls möglich.

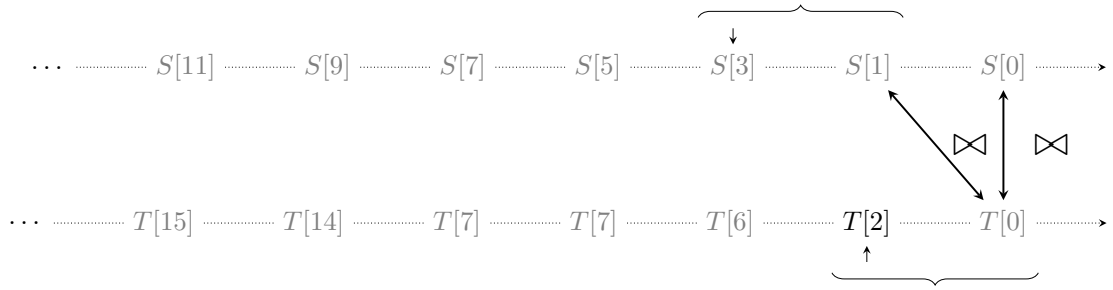

Abb. 29: Minimal-Delta-Join mit zweitem gefundenen Tupelpaar

Zeitwerten der beiden Tupel liegt (vgl. Abb. 29).

Die weitere Vorgehensweise des Algorithmus kann im Appendix ab S. 100 ff. (Abbildungen A.1 – A.7) nachvollzogen werden. Der Algorithmus endet bei gegebenem Datenstromausschnitt an folgender Stelle, die in Abb. 30 dargestellt ist[53]. Es ist ersichtlich, dass der Algorithmus für jedes Tupel den optimalen Verbundpartner gefunden hat.

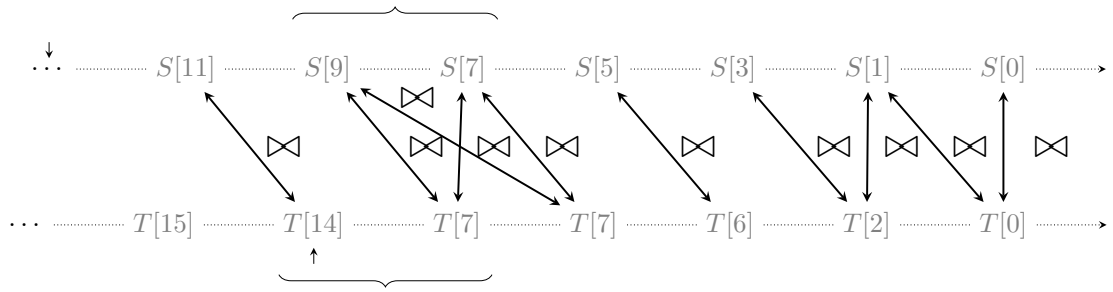

Abb. 30: Minimal-Delta-Join mit letztem gefundenen Tupelpaar

5.4.4 Predictive Minimal-Delta-Join

Der Minimal-Delta-Join puffert die Daten so lange, bis das Sliding Window über die Daten herübergeglitten ist. Danach ist das Löschen der Daten aus dem Puffer sicher, da sich diese Daten nicht mehr als Verbundkandidaten qualifizieren können. Der Minimal-Delta-Join birgt allerdings Probleme, wenn Sensoren mit einer deutlich unterschiedlichen Messfrequenz messen, da Daten dann mitunter lange im Puffer liegen müssen. Zur Darstellung des Problems soll Abb. 31 dienen.

[53] An diesere Stelle wurde das Fenster für $T[14]$ noch nicht verschoben, da das letzte Tupel aus T, von dem der Verbund ausging, das zeitlich spätere $T[7]$ Tupel war.

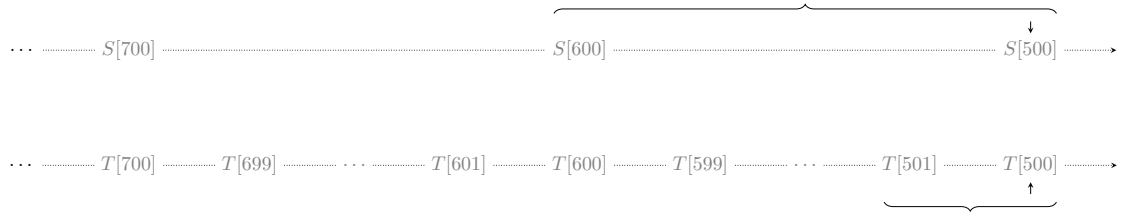

Abb. 31: Datenströme mit stark unterschiedlichen Frequenzen

Der Verbund in dieser Grafik soll an dieser Stelle vom Tupel $T[500]$ ausgehen[54]. Alle Tupel von $[T500]$ bis $T[550]$ würden mit $S[500]$ verknüpft werden, während alle Tupel von $T[551]$ bis $T[650]$ mit $S[600]$ verknüpft werden müssten. Die Tupel mit einem kleineren Zeitstempel als 600 könnten aber vor $\tau = 600$ nicht aus dem Puffer von T entfernt werden, da sich das Window über T erst aktualisiert, wenn $S[600]$ Ausgangspunkt für den Verbund wird. Es werden Daten also unnötig lange im Puffer gehalten[55].

Eine Möglichkeit, Daten von T früher freizugeben, kann darin bestehen, den Timestamp des nächsten Tupels von S durch mitgeführte Statistiken über bisherige Verbundzeitpunkte abzuschätzen und alle Elemente freizugeben, die vermutlich nicht mit dem Tupel $S[500]$ verbunden werden. In Abb. 31 ist berechenbar, dass die Messfrequenz des Sensors, der den Datenstrom S generiert, bei 100 Zeiteinheiten liegt. Dafür lässt sich etwa der Mittelwert der Zeitintervalle zwischen den Daten nutzen. Für das Intervall [500, 600] lässt sich an dieser Stelle für die Daten von T vermutlich voraussagen („*predictive*“), dass alle Tupel von T im Zeitbereich von 500 – 550 mit dem Tupel $S[500]$ in einen Verbund aufgenommen werden und alle Tupel von 551 – 600 in einen Verbund mit $S[600]$ aufgenommen werden. Der Algorithmus kann also schätzen, dass er zum Zeitpunkt 551 alle Daten aus T mit einem Zeitstempel ≤ 550 aus dem Puffer entfernen kann, auch wenn das Tupel $S[600]$ noch nicht im Datenstrom von S vorhanden ist. Es lässt sich annehmen, dass die Verbünde, die in Abb. 32 dargestellt sind, zustande kommen.
Nicht betrachtet wurde die Einbeziehung einer möglichen Fehlerwahrscheinlichkeit, etwa wenn die Messfrequenz des langsameren Sensors sich auf eine Zeitspanne von $\tau = 120$ erhöht. Dies würde etwa im Zeitraum von $\tau = 601$ bis $\tau = 700$ bedeuten, dass nicht nur alle Tupel von T von $T[601]$ bis $T[650]$ mit $S[600]$ einen Verbund eingehen müssten, sondern

[54] Ohne Beschränkung der Allgemeinheit, würde der Join von $S[500]$ ausgehen, wäre das Ergebnis nicht anders.

[55] Anmerkung: Das Problem würde sich vollständig vermeiden lassen, wenn man zum Zeitpunkt des Versetzens des Zeigers von S das Window über T gleichzeitig mitaktualisiert. Da aber nicht nach dem Verbund von $S[500]$ (und anschließendem Weitersetzen des Zeigers) ausgegangen werden kann, dass die Werte in T schon so weit vorliegen, dass ein optimaler Join für $S[600]$ garantiert werden kann, wird diese Möglichkeit hier nicht in Betracht gezogen.

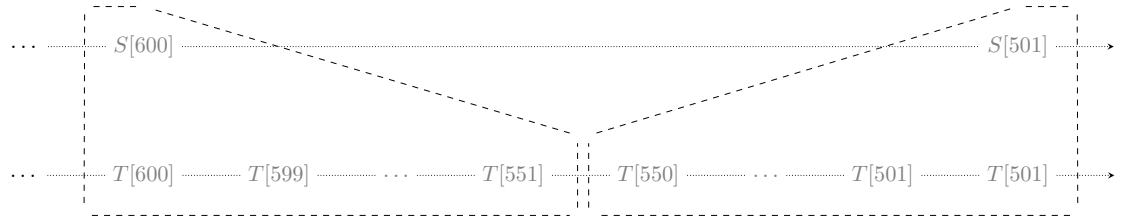

Abb. 32: Predictive Minimal-Delta-Join, der Verbundpartner abschätzt

alle Tupel bis $T[660]$. Würden ab $\tau = 651$ alle Tupel zwischen $T[601]$ und $T[650]$, sowie $S[501]$ prädiktiv gelöscht werden, so würden die Tupel zwischen $T[651]$ und $T[660]$ einen nicht optimalen Verbundpartner finden.

Analog gilt dies, wenn sich die Messfrequenz des langsameren Sensors verringert. Dann gilt, dass für den schnelleren Sensor für einige Tupel bereits ein Verbund zustande gekommen ist, obwohl der spätere Wert des langsameren Datenstroms der bessere Verbundpartner gewesen wäre. Die genauen Auswirkungen sollen an dieser Stelle nicht evaluiert werden. Für ein optimales Verbundergebnis ist eine konstante Frequenz beider Datenströme zwingend erforderlich.

5.4.5 Bewertung der Verbundalgorithmen

Abschließend sollen die Verbundalgorithmen gegenübergestellt und bewertet werden. Es werden Vor- und Nachteile der Algorithmen herausgestellt und gezeigt, in welchen Anwendungsfällen sie praktikabel sind und in welchen Fällen nicht. Eine übersichtsartige Zusammenfassung lässt sich in Tab. 2 (S. 60) finden.

Alle vier Algorithmen basieren auf dem gleichen Paradigma: Sie sind *data-driven*, d.h. sie arbeiten Daten dann ab, wenn neue Daten dazugekommen sind. Das Gegenstück, *time-driven*, wurde hier nicht betrachtet, da das Hauptaugenmerk bei der Konzeption der Verbundalgorithmen darauf liegt, minimale Ressourcen hinsichtlich Speicher zu verbrauchen. Bei einem Algorithmus, der auf einem time-driven Paradigma basiert – beispielsweise indem alle zwei Sekunden der Join gebildet wird – könnte man nicht abschätzen, ob der Puffer in einen Überlaufzustand gerät, etwa weil die Daten des Puffers nicht rechtzeitig verarbeitet und aus dem Puffer freigegeben wurden, bevor neue Daten in den Puffer eingefügt werden. Daten zu verarbeiten, sobald sie eintreffen, ist in diesem Fall die bessere Variante.

Der Ausgangsdatenstrom mit seinem Ausgangselement des Verbundes ist eines der Kernmerkmale, in denen sich die Verbundalgorithmen unterscheiden. In den beiden Vari-

anten des Fuzzy-Merge-Joins gibt es jeweils einen Stream, der als dominant gekennzeichnet wird und von denen der Join ausgeht. In den beiden Varianten des Minimal-Delta-Joins gibt es dagegen keine Dominanz. Hier findet zwischen den Datenströmen eine Alternation der Dominanz genau dann statt, wenn im Datenstrom, der nicht Ausgangspunkt des letzten Verbundes war, das Element mit dem kleinsten Zeitstempel einen kleineren Timestamp hat als das Element mit dem kleinsten Zeitstempel im Datenstrom, von dem der Verbund zuletzt ausging. Beim Fuzzy-Merge-Join ist immer das Element mit dem kleinsten Timestamp im Stream mit der geringeren Frequenz der Ausgangspunkt für den nächsten Verbund. Ähnlich gestaltet sich der bufferlose Fuzzy-Merge-Join, bei dem der Ausgangspunkt für den Verbund immer das Element mit dem kleinsten Timestamp im Stream mit der höheren Frequenz ist. Durch die jeweiligen Implementierungen ist nur für den Fuzzy-Merge-Join garantiert, dass zu jedem Zeitpunkt τ ein Verbundpartner vorliegt[56]. Dies leitet sich daraus ab, dass der dominante Stream der Stream mit der geringeren Frequenz ist, d.h. zu jedem Zeitpunkt τ muss im nicht-dominanten Stream mind. ein Element generiert worden sein.

Während bei beiden Varianten des Minimal-Delta-Joins die Änderung der Frequenz möglich ist – hiermit ist gemeint, dass der Stream mit der geringeren Frequenz der Stream mit der höheren Frequenz wird und vice versa – so gilt dies bei den Varianten des Fuzzy-Merge-Joins nicht. Hier muss im Voraus definiert werden, welches der dominante Stream ist. Dafür ist zusätzliches Anwender- bzw. Entwicklerwissen nötig, um den dominanten Stream korrekt zu bestimmen. Im Falle einer so starken Schwankung der Frequenz beider Datenströme, sodass der dominante Datenstrom der nicht-dominanten Datenstrom werden müsste und vice versa, könnte es im schlimmsten Falle zu einem Pufferüberlauf kommen.

Während bei den Fuzzy-Merge-Join Varianten nach dem Verbund zweier Elemente diese aus dem Puffer gelöscht werden, geschieht dies bei den Minimal-Delta-Join Algorithmen erst, nachdem das Window über die Elemente herübergezogen ist. Dies hat den Vorteil, dass in den Minimal-Delta-Join Varianten ein Mehrfachjoin eines Elementes möglich ist, während dies bei den anderen beiden Verbundvarianten nicht der Fall ist. Weiterhin hat das Window den Vorteil, dass auch Elemente mit gleichen Zeitstempeln in einem Stream auftauchen können, wie sie etwa nach einer Vereinigungsoperation (`UNION`) auftreten können. Die Minimal-Delta-Join Varianten dürfen also als Input den Output einer solchen Vereinigungsoperation nutzen, während das für die Fuzzy-Merge-Join Algorithmen verboten ist.

Der bufferlose Fuzzy-Merge-Join ist der einzige vorgestellte Algorithmus, für den garantiert ist, dass zu jedem Zeitpunkt τ maximal ein Element im Puffer liegt. Die Begründung dafür

[56] Ausfälle eines Algorithmus, das Ende eines Streams etc. ausgenommen.

ist, dass im Datenstrom mit der schnelleren Frequenz maximal ein Tupel im Puffer liegt. Im nicht-dominanten Stream wurde bis dahin maximal ein Tupel produziert, mit dem der Verbund ausgeführt wird. Existiert kein solches Tupel, so wird das Element des dominanten Streams nicht in einen Verbund aufgenommen und gelöscht. Zu jedem Zeitpunkt τ haben die Streams maximal ein Element im Puffer, was als bufferlos bezeichnet wird. Für die anderen drei Algorithmen gilt dies nicht. Beim Fuzzy-Merge-Join besitzt der Puffer des dominanten Streams zwar auch nur ein Element, der Puffer für den nicht nicht-dominanten Stream enthält aber eine Anzahl an Elementen, die wie beschrieben durch den Quotienten der Streamfrequenzen errechnet werden kann. Bei den Varianten des Minimal-Delta-Joins kann aufgrund der Möglichkeit, mehrere Tupel mit gleichem Timestamp in einem Stream zu haben, keine Aussage zu der Anzahl der Elemente im Puffer getroffen werden.

Aufgrund des einfachen Prüfens beim Fuzzy-Merge-Join und des bufferlosen Fuzzy-Merge-Join, ob ein Element im nicht-dominanten Datenstrom vorhanden ist, ist es möglich, dass der Verbundkandidat nicht optimal ist und dennoch in den Verbund aufgenommen wird. Die Varianten des Minimal-Delta-Joins dagegen garantieren zu jedem Tupel einen optimalen Verbundkandidaten.

Aufgrund des geringen Speichers eines Sensors und der damit verbundenen Möglichkeit, nur eine geringe Anzahl an Elementen zu speichern, wurde hier nur der Verbund über den Timestamp untersucht. Ein Verbund über ein arbiträres Attribut ist mit den genannten Methoden nicht angedacht, auch wenn ein Verbund über ein beliebiges Attribut durch Sortieren nach ebendiesem Attribut und das Ausführen einer der genannten Join-Operationen möglich wäre. Für Verbünde über beliebige Attribute gilt es aber, nach effizienteren Algorithmen zu suchen.

Eigenschaft	Fuzzy-Merge-Join	Bufferloser Fuzzy-Merge-Join	Minimal-Delta-Join	Predictive Minimal-Delta-Join
Paradigma	Data-driven	Data-driven	Data-driven	Data-driven
Verbund über	Timestamp	Timestamp	Timestamp	Timestamp
Ausgangsstream des Verbundes	Stream mit langsamerer Frequenz	Stream mit höherer Frequenz	Alternierend	Alternierend
Ausgangselement des Verbundes	Tupel mit kleinstem Timestamp im Stream langsamerer Frequenz	Tupel mit kleinstem Timestamp im Stream höherer Frequenz	Tupel mit kleinstem Timestamp aus beiden Streams	Tupel mit kleinstem Timestamp aus beiden Streams
Verbundpartner zu jedem τ garantiert	✓	✗	✗	✗
Änderung der Datenstromdominanz	Keine Unterstützung Dominanz gleichbleibend, sonst Timestamps auseinanderdriftend	Keine Unterstützung Dominanz gleichbleibend, sonst Timestamps auseinanderdriftend	Unterstützt Dominanz darf sich ändern	Keine Unterstützung aufgrund Abschätzung des nächsten Verbundpartners
Tupel im Puffer	Dominanter Stream: 1 Tupel Nicht-dominanter Stream: $\left\lceil \frac{S_\tau - S_{\tau-1}}{T_\tau - T_{\tau-1}} \right\rceil$ Tupel	Pro Stream max. 1 Tupel	Jeweils abhängig von Anzahl der Tupel mit gleichem Timestamp	Wie Minimal-Delta-Join verfrühtes Löschen der Hälfte der Tupel möglich
Verbundkandidat	Suboptimal Tupel aus anderem Stream mit größeren oder gleichen Zeitstempel	Suboptimal Tupeö aus anderem Stream mit kleinerem oder gleichen Zeitstempel	Optimal Tupel aus anderem Stream mit geringster zeitlicher Differenz	Optimal Tupel aus anderem Stream mit geringster zeitlicher Differenz
Mehrfachverbund von Elementen	Nicht unterstützt	Nicht unterstützt	Unterstützt	Unterstützt
Tupel mit gleichen Zeitstempeln in einem Stream	Nicht unterstützt Folglich nicht für interoperable Operationen mit vorausgehender `Union`-Operation geeignet	Nicht unterstützt Folglich nicht für interoperable Operationen mit vorausgehender `Union`-Operation geeignet	Unterstützt	Nicht unterstützt
Erwartete Implementierungskomplexität	Gering Hauptsächlich Prüfung des Vorhandenseins von Tupeln im nicht-dominanten Datenstrom	Gering Hauptsächlich Prüfung des Vorhandenseins von Tupeln im nicht-dominanten Datenstrom	Komplex Timestampvergleich, zusätzliche Betrachtung von Zeigern und Fenstern mit änderbarer Größe	Komplex Wie Minimal-Delta-Join Zusätzliche Statistikbetrachtung

Tabelle 2: Gegenüberstellung der Verbundalgorithmen des Konzeptionskapitel

5.4.6 Ausblick: Mehrfach-Verbund-Algorithmen

Für den Verbund zweier Datenströme wurden Algorithmen konzipiert und miteinander verglichen. Nicht betrachtet wurden hier Mehrfach-Verbünde, die drei oder mehr Datenströme gleichzeitig in einen Verbund aufnehmen. Für solche Verbünde lassen sich zwei Grundparadigma beschreiben:

Das erste Paradigma beschreibt die Realisierung über Interoperatorparallelität mit dem Spezialfall Pipelineparallelität (vgl. [ÖV11, S. 514]). Dabei handelt es sich um hintereinandergeschachtelte Verbundoperationen, die auf höheren Ebenen eine Verbundoperation auf das Teilergebnis des letzten Verbundes und einem weiteren Datenstrom anwenden. Die Pipelineparallelität ist in Abb. 33 dargestellt: Zuerst werden die Datenströme S und T in einen Verbund aufgenommen. Der Ergebnisdatenstrom heiße ST. Der zweite Verbund nimmt als Eingabeparameter die Zwischenergebnisse, welche sich im Datenstrom ST befinden und den Datenstrom U an. Analog geschieht das eine Ebene höher, in der das Zwischenergebnis des Verbundes aus S, T und U mit dem Datenstrom V kombiniert wird. Die Eigenschaft, dass die Verbünde der höheren Ebenen nicht unabhängig von den Verbünden der unteren Ebenen ausgeführt werden können, erfüllt das Kriterium der Pipelineparallelität (vgl. [ÖV11]).

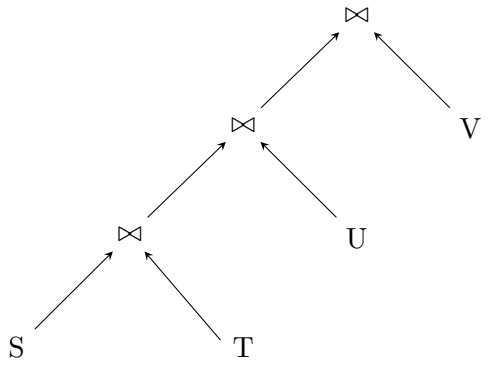

Abb. 33: Pipelineparallelität im Multi-Join-Verfahren nach [ÖV11, S. 514], Grafik adaptiert von [ÖV11, Abb. 9.3, S. 320]

Das zweite Paradigma beschreibt die Realisierung des Mehrfach-Joins durch das Erlauben von mehr als zwei Eingabeströmen pro Verbundoperation. Die Verbundoperation muss daraufhin mehr als zwei Datenströme gleichzeitig verbinden. Schematisch ist eine solche Operation in Abb.34 dargestellt.
Ohne formale Semantik soll für die vier erläuterten Verbundoperationen beschrieben werden, ob und wie es möglich sei, diese für einen Mehrfach-Verbund zu adaptieren. Beim Fuzzy-Merge-Join werden die Streams nach aufsteigender Frequenz geordnet. Der Stream mit der geringsten Frequenz sei auch hier wieder der dominante Stream, von

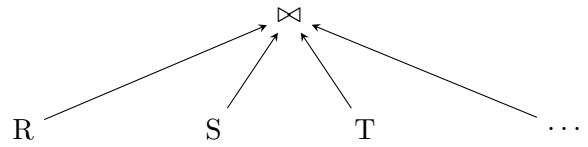

Abb. 34: Multi-Join-Verfahren mit n Datenquellen

dem der Join ausgeht. Die Algorithmus lässt sich angepasst ausführen, indem für jedes Element des dominanten Streams ein Element aus jedem anderen nicht-dominanten Datenströme gesucht wird. Da die nicht-dominanten Datenströme eine gleiche oder höhere Frequenz haben, ist zu jedem Zeitpunkt τ ein Verbundpartner aus allen Streams garantiert. Allerdings entsteht daraus auch ein Nachteil: Bereits in Abb. 21 (S. 47) war ersichtlich, dass nicht alle Elemente des nicht-dominanten Streams in den Verbund einbezogen werden. Diese Eigenschaft pflanzt sich hier fort, da beobachtbar ist, dass je höher die Frequenz des nicht-dominanten Datenstroms ist, desto weniger Elemente dieses Datenstroms in den Verbund einbezogen werden. Entsprechend werden, wenn man den Verbund als Baum betrachtet, wie in Abb. 33 dargestellt, in Richtung des Wurzelknotens immer weniger Tupel pro Verbundoperation in den Verbund aufgenommen.

Etwas abgewandelt lässt sich die Argumentation für den bufferlosen Fuzzy-Merge-Join führen. Hier geht der Join von den Tupeln des Datenstroms mit der höchsten Frequenz (dominanter Stream) aus. Gibt es für ein Tupel eines dominanten Streams keinen Verbundpartner, so wurde dieses Tupel gelöscht und geschaut, ob für das nächste Tupel ein Verbundpartner existiert. Dies ist auch für den Mehrfach-Verbund so möglich. Ohne weiteren Beweis gilt, dass sofern der undominanteste Datenstrom – der Datenstrom mit der geringsten Frequenz – ein Tupel beinhaltet, für das Tupel des dominanten Datenstroms ein Verbundpartner in jedem Datenstrom vorhanden ist. Zusätzlich muss die Thematik des kaskadierenden Löschens betrachtet werden. Gegeben seien die drei Datenströme S, T und U mit $Frequenz(S) > Frequenz(T) > Frequenz(U)$, d.h. S sei der dominante Datenstrom. Sucht S einen passenden Verbundpartner in T und existiert kein Partner, so wird S gelöscht. Das gleiche muss aber gelten, wenn für T noch kein Partner in U existiert, ansonsten würde die *Bufferlos-Eigenschaft* verletzt sein. Dies könnte etwa durch einen Puffer pro Stream realisiert werden, der nur ein Tupel hält und bei Eintreffen eines neuen Tupels dieses überschreibt. Wird ein Tupel in den Puffer des undominantesten Datenstroms gelegt, so gilt aufgrund der Frequenzen, dass für alle Datenströme ein Verbundpartner existiert. Es wird ein Verbund über die Tupel aller Puffer, die je nur ein Tupel – das aktuellste Tupel – enthalten, durchgeführt und anschließend werden alle Puffer geleert.

Für die Varianten des Minimal-Delta-Joins im Mehrfach-Verbund ist die Ausführung ähnlich des Einfach-Verbundes anwendbar. Es muss über jedem Datenstrom ein Fenster pro anderem Datenstrom existieren, d.h. bei n Datenströmen liegen über jedem Datenstrom $n-1$ Fenster. Ansonsten ist die Ausführung ähnlich zur Variante mit zwei Datenströmen: Es wird das Tupel mit dem geringsten Zeitstempel gesucht und für dieses Tupel wird in allen anderen Datenströmen jeweils das Tupel mit dem minimalen Delta gesucht. Entsprechend werden diese in den Verbund aufgenommen. Es muss entschieden werden, was geschieht, wenn zwei Datenströme je ein Element mit dem nächsten global kleinsten Zeitstempel besitzen, d.h. von welchem Datenstrom der Verbund ausgeht. Im Einfach-Verbund war dies klar geregelt: Ausgangspunkt war der Datenstrom, der zuletzt Ausgangspunkt für den Join war. Wechselt im Mehrfach-Verbund aber nun der Datenstrom, in dem der nächste Verbundkandidat liegt und haben zwei Datenströme jeweils einen potentiellen nächsten Kandidaten, von dem der Verbund ausgehen kann, so muss entschieden werden, wie fortgefahren wird. Es werden auch hier ideale Tupelpartner gefunden, Trade-Off ist jedoch die große Metadatenstruktur[57], insbesondere $n-1$ Fenster über jeden Datenstrom bei n Eingangsdatenströmen, die schnell zum Problem werden kann, wenn nur begrenzter Speicher vorhanden ist.

Ähnlich wie beim Minimal-Delta-Join wird die Predictive Minimal-Delta-Join Variante angepasst. An dieser Stelle muss der Anwender entscheiden, für welche Datenströme er eine Statistik für die Voraussage des nächsten Tupels speichern will. Auch hier sind für die Statistik weitere Metadaten notwendig, die gespeichert werden müssen – etwa die Summe und der Count pro Datenstrom, um den Mittelwert zu berechnen. Jedoch sind im Idealfall Daten vorzeitig aus dem Puffer freigebbar, wie in Abs. 5.4.4 (S. 55 f.) beschrieben. Der Anwender muss daher entscheiden, ob es sich lohnt, die prädiktive Variante anzuwenden und falls ja, für welche Datenströme.

Gemeinsamkeit aller (Mehrfach-)Verbund-Algorithmen ist, dass kein Verbund ausgeführt werden kann, sofern nicht mindestens ein Tupel in jedem Datenstrom vorhanden ist. Ein vorzeitiger Teilverbund ist nicht möglich.

[57] Dies sind nicht Metadaten im eigentlichen Sinne, dennoch müssen diese Daten als Datenstruktur mitgeführt werden, um einen korrekten Verbund zu ermöglichen.

5.5 Extremwertbestimmung

Die letzte Operation, die vorgestellt wird, ist die Extremwertbestimmung. Sie dient zur Bestimmung eines Minimal- oder Maximalwertes. Es gibt zwei Varianten der Extremwertbestimmung: Die Berechnung über ein Fenster oder die Berechnung über den gesamten Stream bis zum aktuellsten Zeitpunkt. Beide Varianten werden im Folgenden vorgestellt.

5.5.1 Extremwertbestimmung innerhalb eines Fensters

Gegeben sei ein Stream S und ein Sliding Window mit einer definierten Fenstergröße ws. Findet nun die Berechnung eines Extremwertes statt, so werden nur die Elemente im aktuellen Fenster betrachtet und ausgewertet. Für eine Extremwertbestimmung mit der Fenstergröße ws gilt zu jedem Zeitpunkt τ:

$$min_{ws}(\tau) = min\{(s_\tau),\ \ldots,\ (s_{\tau-(ws-1)})\}.$$
$$max_{ws}(\tau) = max\{(s_\tau),\ \ldots, (s_{\tau-(ws-1)})\}.$$
$$\forall s_\tau \in S : \tau < 0 : min\{s_\tau\} = +\infty,\ max\{s_\tau\} = -\infty.$$

Die ersten beiden Vorschriften zur Bildung vom Minimal- und Maximalwert entsprechen einer trivialen Bestimmung der entsprechenden Extrema mit Ergänzung um die Einschränkung der Suche innerhalb eines Fensters. Bei einer Fenstergröße von ws werden zum Zeitpunkt τ alle Elemente im Zeitraum von τ bis $\tau - (ws - 1)$ analysiert[58] und das entsprechende Extremum ermittelt. Das Fenster ist hier ein data-driven Sliding-Window, d.h. das Sliding-Window betrachtet immer ws Tupel, unabhängig von deren Timestamps.

Die Randbedingung $\forall \tau < 0 : min_{ws}(\tau) = +\infty,\ max_{ws}(\tau) = -\infty$ ist insbesondere für die Initialisierung des Algorithmus wichtig: Gegeben sei ein Stream s und ein Fenster mit der Fenstergröße 3. Wird zum Zeitpunkt τ das Extremum gebildet, so würden die Elemente s_0, s_{-1} und s_{-2} ausgewertet werden, die nicht vorhanden sind. Um damit umzugehen, wird definiert, dass Werte mit einem τ kleiner bei der Minimalbildung als $+\infty$ und bei der Minimalbildung als $-\infty$ definiert seien und somit irrelevant sind. Es wird also mit dem „Anti-Extremum“ initialisiert.

In Abb. 35 sind die ersten vier Schritte einer Maximalwertberechnung ersichtlich. Zum Zeitpunkt $\tau = 0$ werden die Elemente s_τ, $s_{\tau_{-1}}$ und $s_{\tau_{-2}}$ ausgewertet. Die letzten beiden Werte ergeben gemäß obiger Randbedingung $-\infty$, d.h. s_τ ist das einzig verbleibende

[58] Zur Erinnerung: τ_{-1} entspricht dem Vorgängerelement von τ und *nicht* dem Zeitpunkt von τ abzüglich einer Zeiteinheit. $\tau - (ws - 1)$ bestünde bei einer Fenstergröße von 10 aus dem neunten Vorgänger von s_τ ($=s_{\tau-9}$).

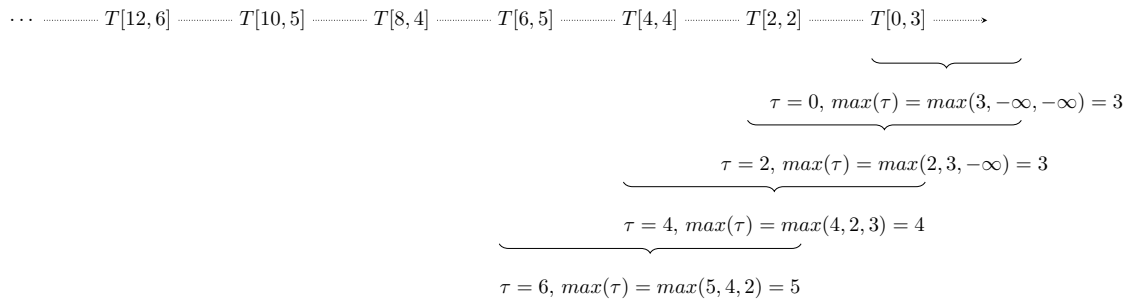

Abb. 35: Erste vier Schritte der Maximalwertbestimmung im Fenster mit Fenstergröße 3

relevante Element, dessen Datenwert 3 auch Maximalwert wird. Im nächsten Schritt verschiebt sich das Sliding Window um ein Tupel und die Berechnung wird erneut ausgeführt. Hier wird das Maximum aus $T[2,2]$, $T[0,3]$ und $T[-1,-\infty]$ gebildet. Auch hier bleibt der Maximalwert weiterhin 3. Analog setzt sich der Algorithmus fort. Tupel, über die das Fenster bereits herübergeglitten ist, können aus dem Puffer freigegeben werden.

Je nach Anwendung kann es sinnvoll sein, anstatt einem Sliding Window ein Tumbling Window zu verwenden. Hierfür funktioniert der Algorithmus analog.

5.5.2 Extremwertbestimmung im Stream

Als Alternative zur Berechnung des Extremwertes pro Window ist es möglich, nur ein Tupel mit dem Extremwert genau dann in den Ausgangsdatenstrom zu produzieren, wenn der Extremwert sich ändert. Es wird jedes Tupel bei Ankunft im Datenstrom untersucht, geschaut, ob das Tupel einen neuen Extremwert darstellt und anschließend gelöscht. Diese Variante benötigt nur einen geringen Puffer, im besten Falle mit der Speichergröße von einem Tupel, in dem das aktuell zu untersuchende Tupel liegt. Weiterhin muss es einen weiteren Puffer mit der Größe von einem Eintrag geben, der über die gesamte Laufzeit den bisher bestimmten Extremwert hält. Dieser Puffer wird beim Finden eines neuen Extremwertes aktualisiert.

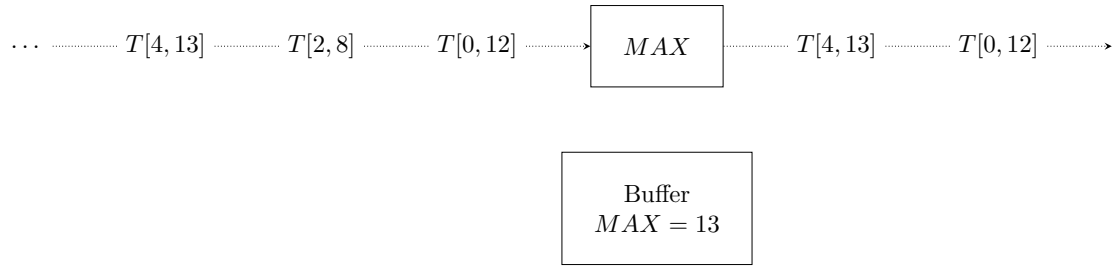

Abb. 36: Ablauf der Maximalwertbestimmung im Stream nach dem Zeitpunkt $\tau = 4$

Beispielhaft ist dieses Vorgehen in Abb. 36 dargestellt: Zu Beginn sei der Puffer, der den aktuellen Maximalwert speichert, leer. Zum Zeitpunkt 0 strömt das Tupel $T[0, 12]$ im Datenstrom auf. Das Tupel wird initialer Maximalwert, da noch kein bisheriger Maximalwert existiert. Der Buffer wird mit dem aktuellen Datenwert des Tupels aktualisiert und das Tupel wird in den Ausgangsdatenstrom gelegt. Das zweite Tupel $T[2, 8]$ taucht im Datenstrom auf. Da $8 > 12$ nicht gilt, wird das Tupel nicht als neuer Extremwert gewertet und verworfen. Das dritte Tupel $[4, 13]$ taucht im Datenstrom auf. Da hier die Bedingung $13 > 12$ gilt, ist der Datenwert 13 des Tupels neuer Extremwert. Der Puffer, das aktuelle Extremum hält, wird aktualisiert und ein Tupel im Ausgangsdatenstrom produziert. Dieser Zustand ist in Abb. 36 dargestellt.

Implementierungstechnisch ließe sich die Extremwertbestimmung – hier Maximalwertbestimmung – auch als Selektion implementieren, die sich bei einem neuen gefundenen Maximalwert auf den Datenwert des gefundenen Tupels adaptiert. Ist die Bedingung etwa $\sigma_{Data>n}$ und es wird ein Tupel im Stream mit einem Datenwert m mit $m > n$ gelesen, so wird dieses Tupel in den Ausgabestrom gelegt und die Selektionsbedingung auf $\sigma_{Data>m}$ adaptiert. Analog gilt dies auch für den Minimalwert.

5.6 Konzeption des Frameworks

In Abb. 2 (S. 20) wurde der Aufbau eines DSMS nach [GÖ10] beschrieben. Diese Architektur ist allerdings vom Autor vermutlich für leistungsfähige DSMS entworfen worden, denn es ist fraglich, ob man diese umfangreiche Architektur auf ressourcenmäßig eingeschränkte Systeme wie Sensor Hubs abbilden kann. Daher muss eine eigene Architektur beschrieben werden.

Für die Umsetzung der vorgestellten Algorithmen des Konzeptionskapitels auf einem Sensor Hub, werden diese als Klassen in einer Bibliothek bereitgestellt. Der Anwender ist für die Programmierung des Sensor Hubs im Sinne der Zuweisung der anzuwendenden Operationen auf Datenströme und deren Kommunikationsfluss mithilfe der entwickelten Bibliothek verantwortlich, während die Logik der Operationen und tieferliegende Komponenten wie Speichermanagement bereits implementiert ist.

Im Vergleich zu Abb. 2 gilt Folgendes: Der Buffer/Input-Monitor existiert abgewandelt weiterhin. Jeder Datenstrom besitzt seinen eigenen Puffer und Operationen zum Einfügen in und zum Lesen aus dem Puffer müssen bereitgestellt werden. Der Nutzer muss sich über die Pufferverwaltung – etwa beim konkurrierenden Zugriff – keine Gedanken machen. Die dafür notwendige Logik ist implementiert und wird bereitgestellt.

Ein Working-Storage, der Teile des gepufferten Input-Streams kopiert und nur auf dieser Teilmenge der Daten arbeitet, existiert nicht mehr. Es wird direkt auf den Puffern der Eingangsdatenströme gearbeitet und die Ergebnisse werden in die Puffer der Ausgangsdatenströme (in Abb. 2 Streaming Outputs) geschrieben.

Ein Local Storage zur Speicherung von Metadaten existiert nicht. Der Nutzer muss beispielsweise bei der Projektion wissen, auf welche Indexwerte eines Tupels er projiziert und was diese semantisch bedeuten. Metadateninformationen – etwa Spaltenbezeichnungen für Tupel, ähnlich dem Schema einer Tabelle in relationalen Datenbanken – werden nicht gespeichert.

Ein Repository, in dem Anfragen gespeichert werden, existiert so nicht. Der Nutzer hat die Möglichkeit, den Sensor dahingehend zu programmieren, dass verschiedene algebraische Anfragen durch eine hintereinander geschachtelte Anwendung von bereitgestellten Operationen abgebildet werden. Eine automatische Übersetzung von Anfragen – etwa von CQL in die entsprechenden Operationen mit entsprechender Reihenfolge – findet nicht automatisch statt. Die Zerlegung einer Anfrage stellt ein eigenes Forschungsthema dar, das nicht im Rahmen dieser Masterarbeit behandelt wird. Literatur zur Zerlegung von Anfragen findet sich etwa bei [GH17], auch wenn die Intention der Autoren hier die Wahrung von Privatsphäreaspekten ist. In diesem Artikel wurde eine Anfrage in Teilanfragen zerlegt. Diese müssten auf unterster Ebene an dieser Stelle in Operationen umgewandelt werden, die als Bibliothek für den Sensor Hub bereitgestellt werden.

Die Konzeption der Algorithmen bzw. der Bibliothek geht mit der Umsetzung einher. Einige Detailfragen werden daher im nachfolgenden Umsetzungskapitel im Rahmen der Implementierungsbeschreibung erläutert.

6 Umsetzung

In diesem Kapitel soll die Umsetzung der Algorithmen beschrieben werden, die im vorherigen Kapitel konzeptionell vorgestellt wurden. Dabei werden Randbedingungen wie Hard- und Softwareumgebung beschrieben und die grundlegende Softwarearchitektur vorgestellt. Es werden Implementierungskonzepte beschrieben und eine Analyse der Effizienz, insbesondere des Minimal-Delta-Joins, durchgeführt. Probleme, die während der Umsetzung auftraten, werden vorgestellt.

6.1 Hardwareumgebung

Die Umsetzung erfolgt hardwaremäßig auf einem *Raspberry Pi* der dritten Version. Der Raspberry Pi ist ein Einplatinencomputer [Wik17b], der unter anderem mit einer 1.2 GHz CPU, 1 GB RAM und 40 *General Purpose Input Output* (GPIO)-Pins ausgestattet ist [Rasa]. Die GPIO-Pins können dazu dienen, Ein- oder Ausgabegeräte anzuschließen und eignen sich daher für den Anschluss von Sensoren zum Erlangen von Sensordaten (vgl. [Wik17b]).

Neben den Raspberry Pi wird eine Komponente namens *Sense HAT* verwendet. Der Sense HAT ist ein zusätzliches Board für den Raspberry Pi, das über die GPIO-Ports angeschlossen wird (s. Abb. 37) und einen Neigungssensor (Gyroskop), einen Beschleunigungssensor (Accelerometer), einen Sensor für die Messung der magnetischen Flussdichte (Magnetometer), einen Temperatorsensor, einen Luftdrucksensor (Barometer) und einen Luftfeuchtigkeitsmesser besitzt. Zusätzlich besitzt der Sense HAT weitere Ein- und Ausgabekomponenten, wie etwa einen Vier-Wege-Joystick oder eine 8x8-LED-Matrix [Rasb]. Über eine Bibliothek für die Programmiersprache Python wird der Zugriff auf die entsprechenden Sensorwerte und die Steuerung der LED-Matrix ermöglicht. Die Sensordaten des Sense HAT werden verwendet, um die Algorithmen des Konzeptionskapitels zu testen.

Da es nicht möglich ist, den Sensor Hub des Sense HAT zu programmieren, wird wie folgt vorgegangen. Der Sense HAT wird als Sensor (bzw. als mehrere Sensoren) angesehen, dessen Datenströme genutzt werden. Es werden nur die reinen Daten der Sensoren verwendet,

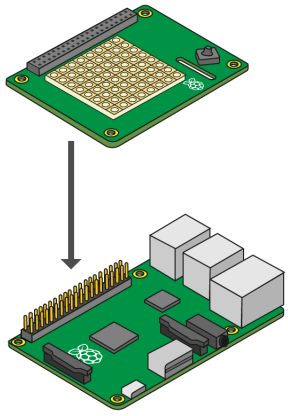

Abb. 37: Raspberry Pi (unten) mit Sense HAT (oben).
Entnommen aus: https://www.raspberrypi.org/learning/astro-pi-guide/images/sense-hat-assembly.png [12.08.2017]

ohne diese in irgendeiner Art und Weise durch Funktionen, die durch den Sense HAT bereit gestellt werden, vorzufiltern oder zu modifizieren. Der Demonstrator, der im Rahmen dieser Umsetzung entstehen soll, implementiert die konzeptionelle Logik eines Sensor Hubs: Er stellt den Zugriff auf die Sensordaten und die Operatoren aus dem Konzeptionsteil bereit und übernimmt grundlegende Aufgaben wie das Speichermanagement für konkurrierende Zugriffe. Weitere Operationen des Sense HATs werden nicht genutzt. Er dient lediglich als eine Art „Proxy“ für die Sensordaten.

Beim Vergleich der Ressourcen des Raspberry Pi mit denen eines Sensor Hubs (vgl. Abs. 3.2, S. 33 f.) fällt auf, dass die Kapazitäten hinsichtlich Rechenleistung und Speicher beim Raspberry Pi deutlich höher sind. Um die Umsetzung an die Ressourcen von Sensor Hubs anzupassen, werden bei der Implementierung bestimmte Aspekte beachtet, die die Nutzung der Ressourcen des Raspberry Pi einschränken.

6.2 Softwareumgebung

Für die Implementierung des Demonstrators wird die Programmiersprache Python in der Version 2.7.9 verwendet. Hauptargument für die Verwendung dieser Sprache stellt die Tatsache dar, dass die Bibliothek für den Zugriff auf den Sense HAT ausschließlich in Python implementiert ist[59]. Es gibt zwar Wrapper für andere Programmiersprachen wie Java,

[59] http://pythonhosted.org/sense-hat/ [12.08.2017].

etwa Java Sense HAT[60], allerdings übersetzen diese die Programmieranweisungen erst in Python und führen diese dann aus, sodass hier eine schlechtere Performance zu erwarten ist.

Python hat den Vorteil, dass es eine leichtgewichtige, portable Programmiersprache ist, die auf der Programmiersprache C basiert[61] und damit auch für die Programmierung auf low-level Ebene geeignet ist[62]. Für eingebettete System gibt es eine Abwandlung von Python namens *Embedded Python*, welche eine minimale Version von Python bereitstellt und sich somit gut für ressourcenbeschränkte Systeme eignet[63]. Eine weitere minimale Variante stellt MicroPython dar, die eine effiziente und leichtgewichtige Implementierung einer Teilmenge der Python-Standardbibliothek enthält und lediglich 256 kB persistenten Speicher für den Code und 16 kB Hauptspeicher benötigt[64].

6.3 Softwarearchitektur

Im Folgenden wird die Softwarearchitektur beschrieben, die als Ausgangspunkt für alle Algorithmen dient. Es werden Kernprobleme der Umsetzung aufgezeigt und entsprechende Lösungen beschrieben.

6.3.1 Datenströme

Grundbaustein des Demonstrators stellt ein *Datenstrom* dar. ein Datenstrom besteht konzeptionell aus einem Puffer für die Zwischenspeicherung von Daten und einer Grundmenge an Operationen, die auf diesem Puffer aufgeführt werden können. Bei der Grundmenge an Operationen handelt es sich nicht um die Algorithmen, etwa zur Extremwertbestimmung, sondern um grundlegende Operationen zum Lesen und Schreiben des Puffers. Hierbei wurde auf Randbedingungen paralleler Systeme bzw. paralleler Programmierung geachtet, etwa um einen gemeinsamer Zugriff zweier Threads konkurrierend auf einen Puffer zu ermöglichen[65].

In der Umsetzung mit Python wird ein Datenstrom durch die Klasse `Stream` repräsentiert.

[60] https://github.com/junthehacker/java-sense-hat [12.08.2017].

[61] vgl. https://github.com/python/cpython [12.08.2017].

[62] „*Die grundlegenden Programme aller Unix-Systeme und die Systemkernel vieler Betriebssysteme sind in C programmiert.*“. https://de.wikipedia.org/w/index.php?title=C_(Programmiersprache)&oldid=167805952 [12.08.2017].

[63] „*Embedded Python. Python can be used in embedded, small or minimal hardware devices, depending on how limiting the devices actually are.*“. https://wiki.python.org/moin/EmbeddedPython [30.08.2017]

[64] https://micropython.org/ [30.08.2017].

[65] Insbesondere wird dies notwendig, wenn ein Thread in einen Puffer ein Ergebnis schreibt und ein anderer Thread diesen Eintrag lesen möchte.

sentiert, die nach dem Klassendiagramm in Abb. 38 aufgebaut ist. Das Klassendiagramm wird nicht an dieser Stelle erläutert, sondern die Bestandteile der Klasse `Stream` wird mit Verweis auf das Klassendiagramm in den nachfolgenden Abschnitten geklärt. Prinzipiell gilt, dass alle Variablen und Methoden, die mit einem „-“ gekennzeichnet sind, mit dem privaten Zugriffsmodifikator versehen sind („private“), während mit „+“ gekennzeichnete Variablen und Methoden als öffentlich gelten („public“). In der Umsetzung mit Python werden private Definitionen hier mit einem Unterstrich eingeleitet.

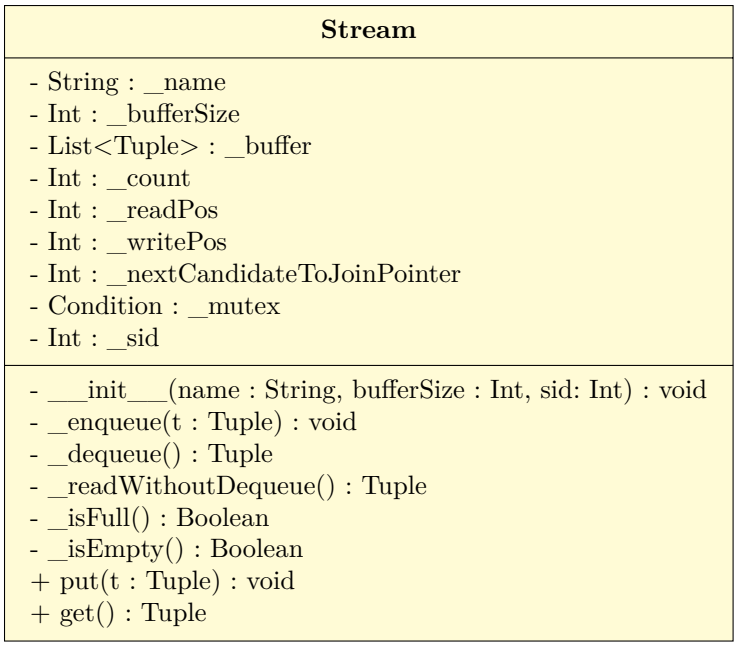

Abb. 38: Klassendiagramm der Klasse `Stream` im Demonstrator

6.3.2 Internes Speichermanagement

Wie im Klassendiagramm in Abb. 38 ersichtlich, besteht ein Stream unter anderem aus der Kompontente eines Puffers (`_buffer`). Der Puffer ist für die Zwischenspeicherung der Elemente eines Streams verantwortlich und besitzt eine fest definierte Anzahl an Elementen, die er speichern kann (`_bufferSize`). Durch die Operationen `_enqueue(t)` und `_dequeue()` ist es möglich, Elemente zum Puffer hinzuzufügen und zu entfernen. Um eine Reorganisation der Daten nach dem Entfernen eines Elementes zu vermeiden, wie es etwa bei der Speicherung in einem Array der Fall wäre (Element $n + 1$ müsste Element n werden, wenn das Element mit dem Index 0 entfernt werden würde), wird der Puffer als Ringpuffer implementiert. Durch die Nutzung zweier Variablen `_writePos` und `_readPos` werden zwei Zeiger implementiert, die aufzeigen, an welche Position als nächsten geschrieben werden darf (`_writePos`) bzw. an welcher Stelle das älteste Element steht (`_readPos`). Wird ein Element in den Puffer geschrieben, so erhöht sich nach dem Schreiben

die Variable `_writePos` um den Wert 1 (modulo Puffergröße aufgrund des Ringpuffers), um die nächste Position zum schreiben anzuzeigen. Wird ein Element gelesen, so erhöht sich nach dem Schreiben die Variable `_readPos` um den Wert 1 (modulo Puffergröße aufgrund des Ringpuffers), um das nächste zu lesende Element anzuzeigen. In beiden Fällen wird die Anzahl der insgesamt enthaltenen Elemente (`_count`) aktualisiert. Die grundlegende Implementierung der `_enqueue(t)` und `_dequeue()` erfolgt wie folgt:

```
def _enqueue(self, t):
    self._count += 1
    self._buffer[self._writePos] = t
    self._writePos = (self._writePos + 1) % self._bufferSize

def _dequeue(self):
    t = self._buffer[self._readPos]
    self._buffer[self._readPos] = None  # frees the object
    self._readPos = (self._readPos + 1) % self._bufferSize
    self._count -= 1
    return t
```

Haben `_readPos` und `_writePos` den gleichen Wert, so lässt sich mithilfe der Operationen `_isFull()` bzw. `_isEmpty()`, die jeweils die Variable `_count` nutzen, prüfen, ob der Ringpuffer vollständig gefüllt oder vollständig leer ist. Im Falle des Einfügens eines Elementes in den Puffer, obwohl dieser gefüllt ist, muss gewartet werden, bis im Puffer ein Element entfernt wurde. Dies geschieht durch die Operationen `put(t)` bzw. `get()`, die die `_enqueue(t)`- bzw. `_dequeue()`-Operationen abhängig vom Zustand des Ringpuffers und unter Nutzung eines Mutex kapseln. Die genaue Funktionsweise wird im nachfolgenden Kapitel erläutert.

Zur Visualisierung des Puffers wurde eine Komponente implementiert, die bei einer Puffergröße von maximal acht Tupeln pro Puffer eine Zeile des 8x8-LED-Displays des Sense HAT nutzt und durch Aufleuchten einer LED signalisiert, ob ein Puffereintrag belegt ist. In Abb. 39 wurden für ein Beispielprogramm vier Datenströme definiert. Für den zweiten

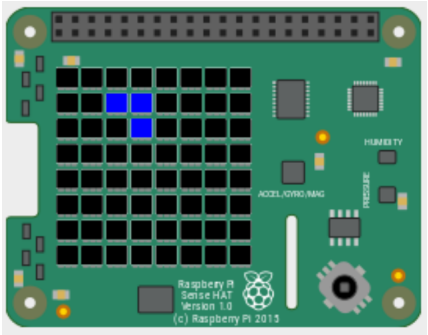

Abb. 39: Anzeige der Pufferbelegung auf dem Sense HAT

Datenstrom ist ersichtlich, dass zwei Elemente im Puffer an der Ringpufferstelle drei und vier liegen. Im Puffer des dritten Datenstroms liegt ein Tupel an der vierten Stelle des Ringpuffers. Alle anderen Puffereinträge der beiden Datenströme sind aktuell nicht belegt. Die Puffer des ersten und des vierten Datenstromes sind an dieser Stelle vollständig leer.

6.3.3 Interdatenstromkommunikation

Bei Operationen, wie etwa der Selektion, werden die Tupel des Puffers eines Streams gelesen, die Operation angewendet und die Ergebnisse dieser als Tupel in den Puffer eines anderen Streams geschrieben. Es besteht also eine Kommunikation zwischen zwei oder mehreren Datenströmen. Wie diese Kommunikation genau vonstattengeht, soll im Nachfolgenden betrachtet werden.

Pro Operation auf einem Datenstrom wird ein neuer Thread erzeugt, der ein (Selektion, Projektion, Extremwertbestimmung) oder mehrere (Verbund) Eingabeströme konsumiert, nach Anwendung der entsprechenden Operation Ergebnistupel berechnet und diese in den Puffer eines Ausgabestromes einfügt. Dies ist in Abb. 40 dargestellt: Hier wurde eine Selektionsoperation in einem eigenen Thread dargestellt. Der Selektion werden zwei Parameter übergeben: Der Input Stream (`iStream`), von welchem die Daten für die Selektion gelesen werden sollen und der Output Stream (`oStream`), in welchen die Ergebnisse der Selektion geschrieben werden.

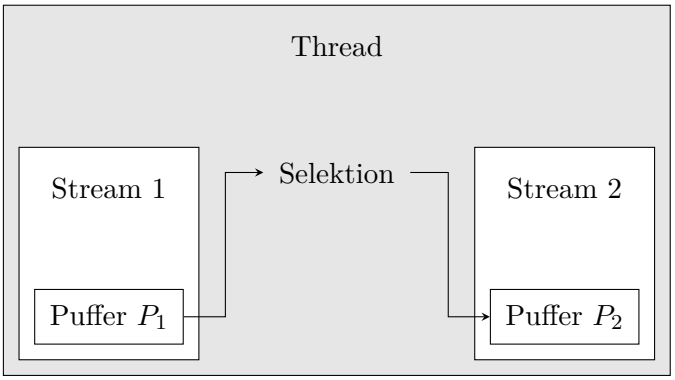

Abb. 40: Kommunikationsfluss im Framework am Beispiel der Selektion

An dieser Stelle muss das Problem gelöst werden, wie ein konkurrierender Zugriff auf den Puffer geschieht. In Abb. 40 ist abgebildet, wie ein Selektionsprozess in den Puffer P_2 schreibt. Soll eine Operation – beispielhaft eine Projektionsoperation – in einem weiteren Thread existieren, die aus dem Puffer P_2 liest und diese Daten weiterverarbeitet, so muss gewährleistet werden, dass der Zugriff auf diese Daten nach bestimmten Regeln abläuft: Ist der Puffer S_2 voll, so darf die Selektionsoperation nicht in diesen Puffer schreiben. Die

Selektionsoperation muss warten, bis ein Element aus dem Puffer entfernt und Platz für ein neues Element geschaffen wurde. Dies geschieht, indem die nachfolgende Operation – hier die Projektion – beim Lesen das Tupel aus dem Puffer P_2 entfernt. Weiterhin gilt, dass eine spätere lesende Operation so lange warten muss, bis mindestens ein Element im Puffer des Input Streams existiert.

Dieses Problem des konkurrierenden Zugriffs auf den Puffer wird durch die Nutzung eines *Mutex* pro Stream bzw. pro Puffer gelöst. Ein Mutex – für mutual exclusion – regelt über Sperr- und Freigabemechanismen, wann ein Thread von einem Puffer lesen bzw. in einen Puffer schreiben darf. Das Lesen von und das Schreiben in einen Puffer wird als *kritische Abschnitt* bezeichnet. Soll eine Operation im kritischen Abschnitt durchgeführt werden, so sperrt man den Puffer, indem man ein Mutex anfordert. Wird dieser Mutex genehmigt – dies ist der Fall, wenn zum aktuellen Zeitpunkt noch kein anderer Thread ein Mutex für diesen Puffer angefordert und genehmigt bekommen hat – so darf man auf dem Puffer arbeiten. Anschließend muss der Mutex wieder freigegeben werden.

Für das Arbeiten auf dem kritischen Abschnitt wurden zwei Operationen – `put(t)` und `get()` eingeführt, die das Operieren auf einem Stream mithilfe eines Mutex regeln:

```
def put(self, t):
    self._mutex.acquire()
    if self._isfull():
        logging.debug("WAITING Thread blocked by full buffer of output
            ↪ stream = %s" % str(self))
        self._mutex.wait()  # wait for tuples dequeued
        logging.debug("WORKING Buffer of output stream not full anymore")
    self._enqueue(t)
    self._mutex.notify()
    self._mutex.release()

def get(self):
self._mutex.acquire()
if self._isempty():
    logging.debug("WAITING Thread blocked by empty buffer
                of input stream")
    self._mutex.wait()  # wait for tuples enqueued
    logging.debug("WORKING Buffer of input stream not empty anymore. Buffer
        ↪  is now = %s" % str(self))
t = self._dequeue()
self._mutex.notify()
self._mutex.release()
return t
```

Die `put(t)`-Operation eines Streams wird für das Hinzufügen eines Tupels in den Puffer des entsprechenden Streams genutzt. Bei Aufruf wird als Parameter das einzufügende Tupel übergeben. Die Operation fordert einen Mutex für den Zielpuffer an und überprüft danach, ob dieser voll ist. Ist dies der Fall, wird der gesamte Thread mit Hilfe der `wait()`-Operation des Mutex suspendiert, also vorläufig angehalten und seine zugewiesene Rechenleistung entzogen. Die Suspendierung wird dann aufgehoben, wenn durch einen anderen Thread die `signal()`-Operation des Mutex ausgeführt wird. Danach findet das Schreiben des Tupels in den Puffer durch die bereits vorgestellte `_enqueue(t)`-Operation statt. Durch den Mutex der `put(t)`-Operation in Kombination mit den `readPos`- und `writePos`-Zeigern des Puffers, wird gewährleistet, dass an der Stelle, an der das Tupel eingefügt wird, kein bisheriges Tupel überschrieben wird.

Analog zur `_put(t)`-Operation funktioniert die `_get()`-Operation zum Lesen des ältesten Tupels aus dem Puffer eines Streams. Dabei wird nach Setzen des Mutex geprüft, ob der Puffer leer ist oder nicht. Falls ja, wird der Thread so lange suspendiert, bis ein Tupel in den Puffer des Streams eingefügt wurde. Danach wird die `_dequeue()`-Operation ausgeführt, um das älteste Tupel im Puffer zu lesen und aus dem Puffer zu löschen.

Im angegebenen Quellcode ist ersichtlich, dass die `put(t)`- und die `get()`-Operationen einen Algorithmus für den Zugriff auf den gemeinsam genutzten Puffer realisieren: Da die Bedingungen `_isFull()` in der `put(t)`-Operation und `_isEmpty()` in der `get()`-Operation niemals gleichzeitig wahr sein kann, wird die `notify()`-Operation des Mutex bei gleichzeitigem Lese- und Schreibzugriff immer mindestens einmal aufgerufen. Entsprechend ist gesichert, dass kein Deadlock beim Zugriff auf den Puffer entstehen kann. Es wurde mithilfe der vorgestellten Operationen ein Konzept vorgestellt, mit der ein gleichzeitiger Lese- und Schreibzugriff durch zwei Threads auf einen Puffer erfolgen kann, ohne, dass ein Tupel überschrieben wird oder aus einem leeren Puffer gelesen wird. Durch die mögliche Suspendierung der Threads geschieht dies effizient[66].

6.3.4 Klassen-/Implementierungshierarchie

Die vorgestellte Klasse `Stream` ist die Basisklasse für alle Datenströme im Programm. Entsprechend erben andere Klassen, die auf dem kritischen Abschnitt operieren, von der Oberklasse `Stream`. Die Unterklassen repräsentieren intern den bereits vorgestellten allgemeinen Stream mit Pufferverwaltung und stellen weitere, algorithmusspezifische Operationen bereit, die im kritischen Bereich arbeiten, wie etwa die Bestimmung des

[66] Die Realisierung ist effizienter, als etwa eine Realisierung mithilfe einer While-Schleife, in der andauernd geschaut wird, ob ein Puffer voll oder leer ist, da im Gegensatz zur Nutzung einer While-Schleife bei der Nutzung eines Mutex die Rechenleistung des Threads entzogen wird.

minimalen Deltas für den Minimal-Delta-Join[67]. Für Operationen, für die die allgemein definierten `put(t)`- und `get()`-Operationen ausreichen – das sind Selektion, Projektion, Bestimmung dee globalen Extrema – ist keine Vererbung notwendig.

Folgende Unterklassen wurden entsprechend implementiert:

- Eine Klasse für den Fuzzy-Merge-Join, bei dem über den Puffer des Streams zur Findung eines Join-Partners iteriert wird. Eine eigene Klasse wäre nicht zwingend notwendig, ermöglicht aber Ermittlung des Verbundkandidaten das *einmalige* Setzen eines Mutex, das Iterieren über die Elemente und das Freigeben des Mutex anstatt *pro iteriertem Element* ein Mutex setzen zu müssen.
- Eine Klasse für den bufferlosen Fuzzy-Merge-Join. Aufgrund der Funktionsweise des bufferlosen Fuzzy-Merge-Joins kann hierfür auch eine `Stream`-Klasse verwendet werden. Die Klasse dient nur der Konsistenz der Vererbungsoperationen, sodass jede Verbundoperation eine eigene Klasse besitzt. Auch ist so eine Fehlerbehandlung besser implementierbar
- Eine Klasse für den Minimal-Delta-Join mit einer Operation, bei der das minimale Delta bestimmt wird, einer Operation, bei der die Verbundkandidaten mit dem minimalen Delta ermittelt werden und eine Operation, die alle Tupel löscht, über die das Fenster bereits herübergezogen ist und die daher nicht mehr Verbundkandidat sein können.

Die in Abb. 41 Klassen- und Hierarchierungimplementierung leitet sich entsprechend daraus ab[68].
Die konkrete Implementierung ist im Appendix in den Abschnitten A.7 ab S. 111 ersichtlich. Die Klassen tragen `Source` im Namen, da diese Klassen in den entsprechenden Operationen den Ausgangspuffer darstellen, auf welchen die entsprechenden Operationen angewandt werden. In Anlehnung an Abb. 40 (S. 73) würde beispielsweise *Stream 1* als Ausgangspunkt („*Source*“) für die Selektion dienen. Würde in Abb. 40 etwa ein Fuzzy-Merge-Join anstatt einer Selektion ausgeführt werden sollen, so muss *Stream 1* ein Stream der Klasse `StreamFMJSource` sein, in der zusätzlich die Funktion `getFMJCandidate(dominantTime)` zur Suche des Verbundkandidaten bereitgestellt wird (vgl. Klassendiagramm in Abb. 41). Für den Zielstream (in Abb. 40: `Stream 2`) existieren keine Vorgaben, welche Stream-Klasse dies sein muss[69], da die Elemente einfach in den Puffer des Zielstreams geschrieben werden und keine weiteren Operationen vonnöten sind.

[67] Das Operieren im kritischen Abschnitt wäre hier das Lesen des Puffers.
[68] Variablen und Funktionen, die nicht Bestandteil der Algorithmen sind, sondern lediglich für die Erstellung von Statistiken/Plots für diese Masterarbeit genutzt wurden, sind nicht im Klassendiagramm aufgeführt.
[69] Hinweis: Es muss die Klasse `Stream` oder eine der Unterklassen gemäß Klassendiagramm in Abb. 41 sein. Welche konkrete Klasse benutzt wird, ist nicht relevant.

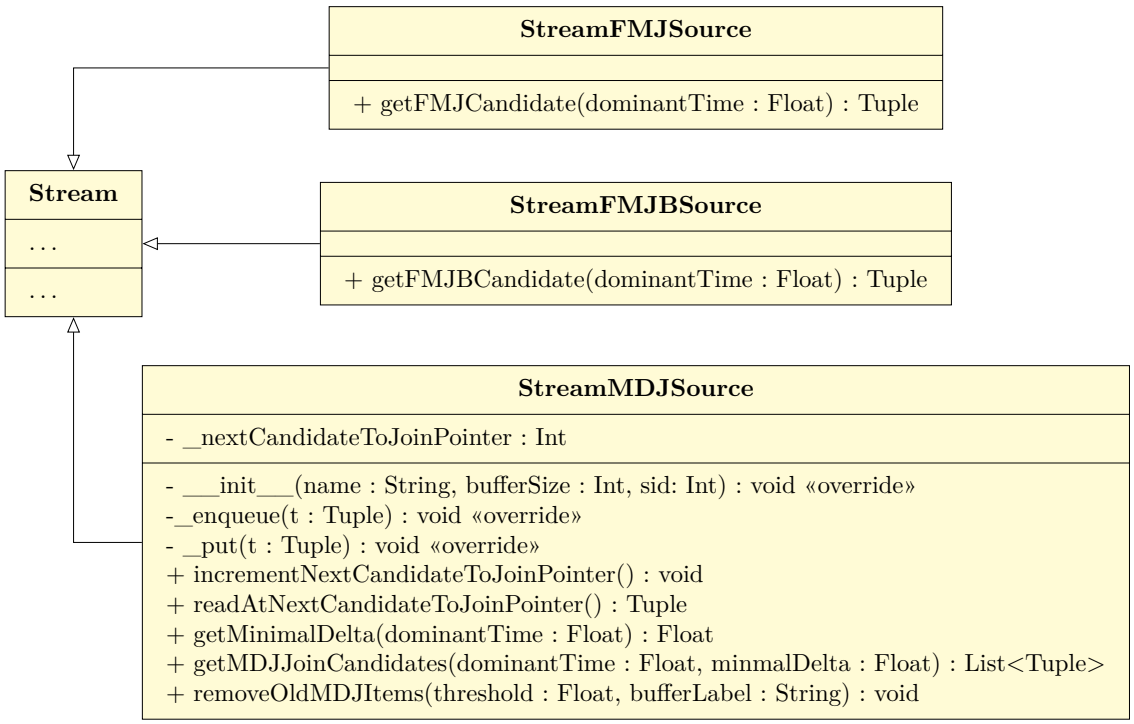

Abb. 41: Vollständiges Klassendiagramm des Demonstrators

6.4 Datenquellen

Als Datenquelle für den Demonstrator werden die Sensordaten des Sense HAT genutzt. Dabei kann der Nutzer zwischen einem Emulator oder zwischen echten Daten unterscheiden. Die Python-Bibliothek des Sense HAT bietet eine graphische Umgebung an, um eine Sensorplattform zu emulieren. Diese verhält sich genau so wie der Sense HAT, bietet allerdings die Möglichkeit, die Messwerte mit exakt den selben Schnittstellen wie beim physischen Sense HAT zu emulieren. Dies bietet sich für den Rahmen der Masterarbeit insofern an, dass Dritte ohne Zwang, im Besitz eines Sense HATs sein zu müssen, die Algorithmen nachvollziehen bzw. den Demonstrator ausführen können. Der Sense HAT Emulator ist in Abb. 42 abgebildet.
Für den Wechsel zwischen Werten des Emulators und den echten Werten der Sensoren des Sense HATs lässt sich in der Präambel[70] des Demonstrators die Variable `vm` auf `True` oder `False` setzen. Hat diese Variable den Wert `True`, so wird der Sense HAT Emulator verwendet, andersfalls der physische Sense HAT. Der entsprechende Quellcode dafür ist in den Abschnitten A.7.1 ab S. 111 und A.7.15 ab S. 126 ersichtlich.

[70]Mit Präambel ist der Bereich des Quellcodes gemeint, der für die Einrichtung des Programmes dient und vor dem eigentlichen Quellcode auftaucht. Üblicherweise werden hier `import`-Anweisungen ausgeführt.

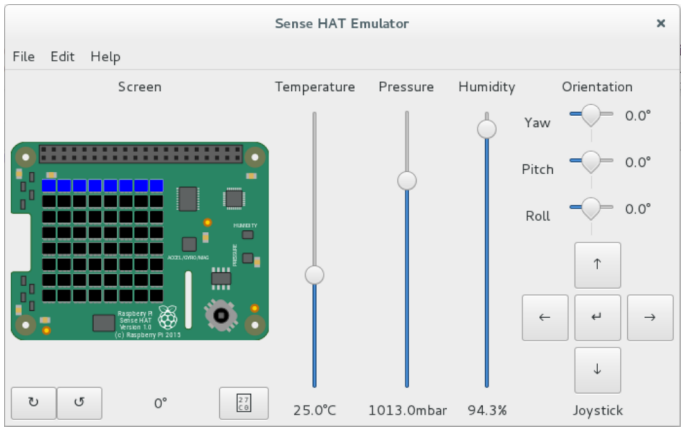

Abb. 42: Sense HAT Emulator zum Emulieren von Messwerten

Als Datenquellen wurden in der Umsetzung die Sensoren für Luftfeuchtigkeit und der Luftdruck verwendet. Gemäß Konzeption besteht ein Tupel aus einem Timestamp und den entsprechenden Messwerten (vgl. formale Semantik, Abs. 5.1). Der Zeitstempel wird im Programm dem Tupel beim Lesen des Sensordatenwertes an erster Stelle durch den Demonstrator hinzugefügt. Das Tupel wird dann in den Puffer eines Datenstroms geschrieben, wie im folgenden Beispielcode ersichtlich ist:

```
def fetchHumidityData(oStream):
    while True:
        humidity = sensorSource.get_humidity()
        humidity = [time.time(), float(humidity)]
        logging.info("[==>] Pressure[%f, %f]", float(humidity[0]), float(
            ↪ humidity[1]))
        oStream.put(humidity)
        time.sleep(DECAY_HUMIDITY)
```

Die Operation `fetchHumidityData(oStream)` dient zum Lesen von Werten des Luftfeuchtigkeitssensors und wird mit dem Parameter `oStream` aufgerufen, der den Zielstream darstellt. In den Puffer dieses Streams werden die Tupel geschrieben. In der Präambel des Programmes ist es möglich, einen Dämpfungsparameter `DECAY_HUMIDITY` zu setzen, der dazu dient, die Datenrate des Lesens des Sensors zu begrenzen. Nimmt dieser Dämpfungsparameter den Wert 0 an, so treffen Daten kontinuierlich ohne Dämpfung der Datenrate ein.

Neben den Operationen `fetchHumidityData(oStream)` und `fetchPressureData(oStream)` existieren die beiden Operationen `fetchHumidityDataLC(oStream)` und `fetchPressureDataLC(oStream)`. Diese funktionieren analog zu der eben beschriebenen Operation, mit der Ausnahme, dass statt eines Timestamps in Wallclock-Time eine logische Uhr genutzt wird, wie sie auch in den Beschreibungen der Algorithmen im Konzeptionsabschnitt ab S. 5 Anwendung findet.

6.5 Beispielprogramm

Bisher wurden einzelne Komponenten wie Datenquellen, Datenstromklassen und Operatoren betrachtet. Um das Verständnis zu stärken, wie diese Komponenten zusammenarbeiten, wird nachfolgend ein Beispielsprogramm vorgestellt, in dem zwei Datenquellen ausgelesen werden, eine Selektionsoperaton auf einen Datenstrom angewendet wird und beide Datenströme dann eine Verbundoperation ausgeführt wird. Letztendlich sollen die Ergebnistupel ausgegeben bzw. in eine Datensenke geschrieben werden. Das Beispielprogramm wurde in Abb. 43 visualisiert. Dabei wurden die Operationen als Rechtecke gekennzeichnet, in denen jeweils oben die Namen der Datenströme stehen und darunter die dafür verwendete Klasse. Die Pfeile zwischen den Operationen stellen Datenflüsse dar.

Zunächst ist es nötig, die Datenströme im Programm zu definieren. In diesem Fall sollen Luftfeuchtigkeit und der Luftdruck als Quelle dienen. Dafür werden die Klassen genutzt, die im Abschnitt 6.3.4 erläutert wurden:

```
humidityStream = Stream("HumidityStream", 8, sid=0)
pressureStream = StreamMDJSource("PressureStream", 8, sid=1)
selectionStream = StreamMDJSource("SelectionStream", 8, sid=2)
minimalDeltaJoinStream = Stream("MinimalDeltaJoinStream", 8, sid=3)
sinkStream = Stream("Print Stream", 8, sid=4)
```

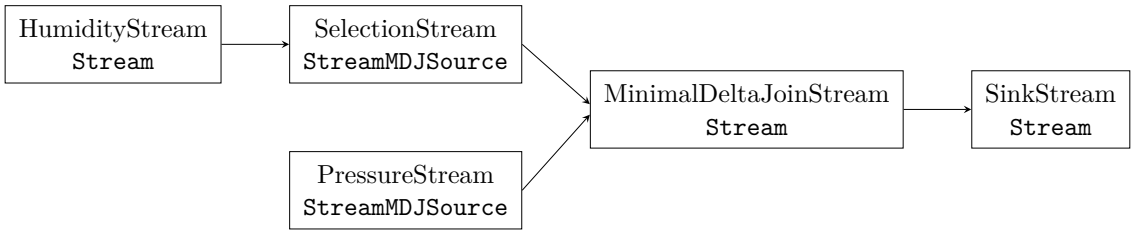

Abb. 43: Datenfluss des Beispieldatenprogramms

Mit dem genannten Codestück wurden fünf Datenströme der Klasse `Stream` erzeugt, wobei `pressureStream` und `selectionStream` eine Spezialisierung der Klasse `Stream` sind und der Klasse `StreamMDJSource` angehören. Dies ist notwendig, weil diese beiden Datenströme die Eingangsströme für den nachfolgenden Minimal-Delta-Join darstellen (vgl. Abb. 43) und diese Klasse erweiterte Funktionen bereitstellt, wie etwa die Suche nach dem minimalen Delta und das Ermitteln der Verbundkandidaten. Die Klasse `Stream` und ihre Spezialisierung nehmen dabei im Konstruktur zwei oder drei Parameter an: Den Namen des Streams, die Größe des Puffer des Streams und optional die Stream-ID (`sid`). Wird die Stream-ID gesetzt und ist die Puffergröße maximal acht, so wird auf dem 8x8 LED-Display des Sense HAT in der `sid`-*ten* Zeile visualisiert, welche der Pufferpositionen belegt sind.

Dadurch lässt sich der Speicherverbrauch der Algorithmen visualisieren. Erläutert wurde dieses in Abs. 6.3.2 ab S. 71.

Als dieser Stelle wird pro Datenstrom ein Thread erzeugt. Dabei wird auch festgelegt, welche Operationen auf welchem Datenstrom ausgeführt werden[71]. Dabei werden für die einzelnen Operationen die Input- und Outputströme, falls vorhanden, definiert und damit der Datenfluss fixiert.

```
humidityThread = threading.Thread(name='humidity', target=
    ↪ fetchHumidityDataLC, args=(humidityStream,))
selectionThread = threading.Thread(name='selection', target=
    ↪ selectionOperation, args=(humidityStream, selectionStream, 1, ">",
    ↪ 40))
pressureThread = threading.Thread(name='pressure', target=
    ↪ fetchPressureDataLC, args=(pressureStream,))
minimalDeltaJoinThread = threading.Thread(name='md join', target=
    ↪ minimalDeltaJoin, args=(selectionStream, pressureStream,
    ↪ minimalDeltaJoinStream))
sinkThread = threading.Thread(name='sink', target=sink, args=(
    ↪ minimalDeltaJoinStream,))
```

Es werden fünf Threads definiert, die jeweils mit dem Parameter `name` benannt werden. Der Name ist in den späteren Programmausgaben sichtbar, hat darüber hinaus allerdings keine Relevanz und ist entsprechend wahlfrei. Durch den Parameter `target` wird festgelegt, welche Operation aufgerufen wird und durch den Parameter `args` wird beschrieben, welche Parameter der Operation, die im `target`-Parameter gesetzt wurde, übergeben werden.

Am Beispiel von `humidityThread` bedeutet dies, dass die Operation `fetchHumidityDataLC` in einem eigenen Thread aufgerufen wird und den Parameter `humidityStream` übergeben bekommt. In der Definition dieser Operation (vgl. Abs. A.7.6, S. 119) ist ersichtlich, dass der Parameter von `fetchHumidityDataLC` der Output-Datenstrom ist. In der genannten Funktion werden Luftfeuchtigkeitsdaten vom Sensor ausgelesen und mit einem Timestamp verknüft. Insgesamt wird hier im Thread `humidityThread` kontinuierlich die Luftfeuchtigkeit ausgelesen und in den Puffer des Streams `humidityStream` geschrieben.

Parallel liest etwa der Thread `selectionThread` in der Operation `selectionOperation` aus diesem Thread `humidityStream` die Luftfeuchtigkeitsdaten aus, wendet die Selektion an und schreibt Zieltupel in den Puffer des Datenstromes `selectionStream`. Analog arbeiten auch andere Operationen parallel. Die zu threadenden Funktionen wie `fetchHumidityDateLC`, `selectionOperation`, `minimalDeltaJoin` oder `sink` liegen im-

[71] Bisher fand nur eine beliebige Benennung statt, die unabhängig von den ausgeführen Operationen ist. Man könnte auch auf dem `selectionStream` eine Projektion durchführen.

plementierungstechnisch außerhalb einer Klasse.

Um das Minimalbeispiel zu beenden, müssen die einzelnen Threads noch gestartet werden. Dies geschieht durch das folgende Codestück:

```
sinkThread.start()
minimalDeltaJoinThread.start()
pressureThread.start()
selectionThread.start()
humidityThread.start()
```

6.6 Implementierung unärer Operationen

Die Implementierung der unären Operationen, d.h. Operationen, die nur *einen* Datenstrom konsumieren müssen [Dat12], wird aufgrund der geringen Komplexität in einem Abschnitt zusammengefasst. Die unären Operationen sind die Selektion, die Projektion und die Extremwertbestimmung.

6.6.1 Implementierung der Selektionsoperation

Die Implementierung der Selektionsoperation des Demonstrators benötigt drei Argumente. Diese sind der Eingangsdatenstrom, auf dem die Selektionsoperation ausgeführt werden soll, der Ausgangsdatenstrom, in welchen die Tupel des Eingangsdatenstromes eingefügt werden, die die Selektionsprädikate erfüllen, sowie die Liste der Selektionsprädikate. Die Implementierung gestaltet sich, wie im Abs. A.7.7 auf S. 120 ersichtlich.

Die Liste der Selektionsprädikate besteht aus Einträgen mit Dreiertupeln über dem Schema `[index, op, threshold]`. Das Beispielselektionsprädikat $[1, " \geq ", 20]$ würde bei jedem Tupel prüfen, ob das Tupel an der Indexstelle 1 einen größeren oder gleichen Wert als 20 hat. Ist dies der Fall, wird die nächste Selektionsbedingung ausgewertet, ansonsten wird das Tupel gefiltert und nicht in den Ergebnisdatenstrom eingefügt. Wurde eine Liste von Selektionsbedingungen als Parameter übergeben, wird das Tupel in den Ergebnisdatenstrom eingefügt, wenn *alle* Selektionsbedinungen erfüllt sind. Die Selektionsbedingungen sind an dieser Stelle also *konjunktiv*. Bei einer Liste von Selektionsbedingungen mit $[[0, " < ", 10], [1, " \geq ", 20]]$ wird pro Tupel geprüft, ob der Eintrag an der Indexstelle 0 einen kleineren Wert als 10 besitzt *und* ob der Eintrag an der Indexstelle 1 einen größeren oder gleichen Wert als 20 hat. Nur dann wird das Tupel in den Ergebnisdatenstrom geschrieben. Als Operationsparameter (`op` in der Selektionsbedingung) werden `<`, `<=`, `=`, `>=`, `>` und `!=` unterstützt.

6.6.2 Implementierung der Projektionsoperation

Die Projektionsoperation des Demonstrators nimmt drei Argumente an: Den Eingangsdatenstrom, aus dem die Tupel gelesen werden, den Ausgangsdatenstrom, in den die Tupel nach Anwenden der Projektionsoperation geschrieben werden und eine Liste von Indexwerten, nach denen projiziert werden soll. Die Implementierung ist im Abs. A.7.8, S. 120 ersichtlich. Besitzt ein Tupel des Eingangsdatenstroms das Schema $T[a_0, a_1, a_2, a_3, a_4]$ und wird die Projektionsliste $[0, 3]$ übergeben, so wird in den Zieldatenstrom das Tupel mit dem Schema $T[a_0, a_3]$ geschrieben. Das Tupel wurde um die Attribute a_1, a_2 und a_4 vermindert. Taucht in der Projektionsliste nicht der Index 0 auf, so wird es automatisch der Projektionsliste hinzugefügt. Dies gilt aufgrund der Bedingung, dass an Indexstelle 0 der Zeitstempel vorhanden ist und nicht herausprojiziert werden darf (vgl. 5.3, S. 41). Im Gegensatz zu relationalen Datenbanken ist hier kein Relationenschema mit Attributnamen nötig, sondern es werden die auf die zu projizierenden Indexwerte des Tupels als Projektionsattribute genutzt. Dafür müssen Aufbau und Semantik der Tupel bekannt sein, auf denen die Projektionsoperation angewendet wird.

6.6.3 Implementierung der Extremwertoperationen

Im Demonstrator wurden zwei Varianten der Operation zum Bestimmen des Extremwertes implementiert. Die erste Variante bestimmt den Extremwert eines Datenstromes über die gesamte Laufzeit. Wird ein neuer Extremwert gefunden, so wird das Tupel in den Ausgabedatenstrom eingefügt. Die zweite Variante bestimmt den Extremwert über ein Fenster. Das Fenster ist an dieser Stelle der gesamte Pufferbereich eines Datenstroms. Wird ein neues Tupel im Puffer des Eingangsdatenstroms gespeichert, so wird die Extremwertoperation erneut ausgeführt und das Extremum – unabhängig davon, ob das Extremum das gleiche ist, wie das des vorherigen Fensters – in den Ausgabedatenstrom geschrieben.

Beide Operationen benötigen vier Parameter: Den Eingangsdatenstrom, den Ausgangsdatenstrom, den Tupelindex, die angibt, an welcher Stelle des Tupels nach dem Extremwert gesucht werden soll, sowie die konkrete Extremwertoperation (Maximum, Minimum). Soll nach dem Maximum gesucht werden, so muss als vierter Parameter `>` übergeben werden, für die Suche nach dem Minimum entsprechend `<`. Die Implementierungen der Extremwertoperationen sind in Abs. A.7.9 ab S. 121 bzw. in Abs. A.7.10 ab S. 121 aufgeführt.

6.7 Implementierung der Verbundalgorithmen

Bereits in Abb. 41 auf Seite 77 war im Klassendiagramm ersichtlich, dass alle Verbundalgorithmen eine Spezialisierung der Klasse `Stream` darstellen. Die drei Verbundalgorithmen des Fuzzy-Merge-Joins, des bufferlosen Fuzzy-Merge-Joins und des Minimal-Delta-Joins sollten von der implementierungstechnischen Seite vorgestellt werden. Der Predictive Minimal-Delta-Join wurde nicht implementiert, da diese Variante nur einen Ausblick im Konzeptionskapitel dargestellt hat und nicht weiter betrachtet wurde.

Sowohl der Fuzzy-Merge-Join als auch der bufferlose Fuzzy-Merge-Join besitzen jeweils eine private Operation zum Bestimmen des nächsten Verbundkandidaten (`getFMJCandidate (dominantTime)` bzw. `getFMJBCandiate(dominantTime)`) und kommen aufgrund ihrer formalen Definition ohne zusätzliche Datenstrukturen wie etwa ein Window oder ein Pointer, der auf das nächste, in den Verbund aufzunehmende, Element zeigt, aus. Bei der Definition eines Threads ist darauf zu achten, dass beim Fuzzy-Merge-Join und beim bufferlosen Fuzzy-Merge-Join die Dominanz der Datenströme eine Rolle spielt. Erstellt man einen Thread, der für die Ausführung eines Fuzzy-Merge-Joins verantwortlich ist, sind drei Parameter anzugeben:

```
fuzzyMergeJoinThread = threading.Thread(name='fm join', target=
    ↪ fuzzyMergeJoin, args=(humidityStream, pressureStream,
    ↪ fuzzyMergeJoinStream))
```

Der erste Parameter – in diesem Falle `humidityStream` – stellt den *dominanten* Datenstrom dar. Der zweite Parameter – hier `pressureStream` – stellt den nicht-dominanten Datenstrom[72] und der dritte Parameter – `fuzzyMergeJoinStream` – stellt den Ergebnisdatenstrom dar, in dessen Puffer die Tupel nach der Verbundoperation eingefügt werden.

Bei der Nutzung des Minimal-Delta-Joins ist es beliebig, welcher der Streams zuerst angegeben wird, da die Dominanz bzw. der Ausgangsdatenstrom basierend auf deren enthaltenen Elementen dynamisch wechselt. Jeder Eingangsdatenstrom eines Minimal-Delta-Joins gehört der spezialisierten Klasse `StreamMDJSource` and un verfügt über zusätzliche Datenstrukturen wie einer Variablen, die auf das Ausgangselement für den nächsten Verbund zeigt. Dies ist immer das kleinste Element im Puffer des Datenstromes, von dem noch kein Verbund ausging[73]. Zusätzlich verfügen Datenströme der Klasse `StreamMDJSource` über ein Fenster, in dem potentielle Kandidaten liegen (vgl. Abs. 5.4.3,

[72] Im Quellcode wird dieser, abgeleitet von der Genetik in der Biologie, als rezessiver Stream bezeichnet (vgl. http://www.transgen.de/lexikon/1797.dominant-rezessiv.html [24.08.2017]).

[73] Es werden beide Elemente verglichen, auf die je ein Zeiger der beiden Eingangsdatenströme zeigt. Das kleinere Element stellt den Ausgangspunkt für den Verbund dar. Sind beide Elemente gleich, gelten Regeln, die im Abs. 5.4.3 (S. 51 ff.) beschrieben wurden.

S. 51 ff.). Das Fenster wird intern über zwei Variablen, `_windowStart` und `_windowEnd`, des Datentypes *Integer* realisiert, die auf die entsprechenden Positionen im Puffer zeigen. Nur Tupel innerhalb des Fensters werden bei der Suche nach dem minimalen Delta und nach den möglichen Kandidaten betrachtet.

Der Algorithmus des Minimal-Delta-Joins läuft in mehreren Phasen ab. Zuerst wird aus beiden Datenströmen das Element bestimmt, von dem der Verbund ausgeht. Dieses sei für diesen Durchlauf das dominante Element. Anschließend wird für den nicht-dominanten Stream innerhalb der Hilfsmethode `_getMDJMinimalDelta(dominantTime)` geprüft, ob es bereits ein Element gibt, das bereits einen größeren Zeitstempel hat als das dominante Element. Ist dies nicht der Fall, wartet der Algorithmus so lange. Ansonsten kann nicht gewährleistet werden, dass das minimale Delta gebildet wird. Würden nur schon existente Tupel im Puffer des Datenstroms geprüft werden, so kann es sein, dass zum nächst kleineren Tupel des nicht-dominanten Datenstroms das minimale Delta gebildet wird, die Differenz zum nächsten, im nicht-dominanten Datenstrom eintreffende und zeitlich größere Element als das dominante Element aber ein minimaleres Delta darstellen könnte.

Ist das minimale Delta gefunden, so werden innerhalb des Fenster über dem nicht-dominanten Datenstrom alle Tupel bestimmt, die zu dem dominanten Tupel dieses minimale Delta besitzen. Dies geschieht durch Anwendung der Operation `getMDJJoinCandidates (dominantTime, minimalDelta)`. Diese werden in die Datenstruktur einer Liste zurückgegeben und der Verbund des dominanten Tupels mit den Tupeln in der Liste wird nun vorgenommen. Letztendlich werden alle Tupel gelöscht, die im nicht-dominanten Datenstrom hinter dem Beginn des Fensters liegen, da durch die zeitliche Ordnung der Tupel gewährleistet wird, dass diese Tupel nicht mehr Verbundkandidat werden können. Anschließend beginnt der Algorithmus erneut.

In der theoretische Konzeption wurden die konkreten Grenzen eines Puffers in Kombination mit seiner konkurrierenden Eigenschaft nicht betrachtet. Die Konkurrenz spielt in der Implementierung eine bedeutende Rolle. Bisher wurde mittels Nutzung eines Mutex ein Puffer implementiert, mit dem zwei Threads auf einen Puffer zugreifen konnten. Ist ein Puffer voll, so bleibt dieser Puffer so lange für das Schreiben in diesen blockiert, bis ein anderer Thread aus diesem Puffer liest und damit das Element entfernt.

Diese Implementierung ist zwar threadsicher, hat aber einen entscheidenden Nachteil, der an der Abb. 44 aufgezeigt wird. Gegeben sei ein Datenstrom S, in den mit einer

Frequenz von einer Sekunde neue Tupel eingefügt werden und ein Datenstrom T, in den mit einer Frequenz von zehn Minuten neue Tupel eingefügt werden. Beide Puffer können acht Elemente speichern.

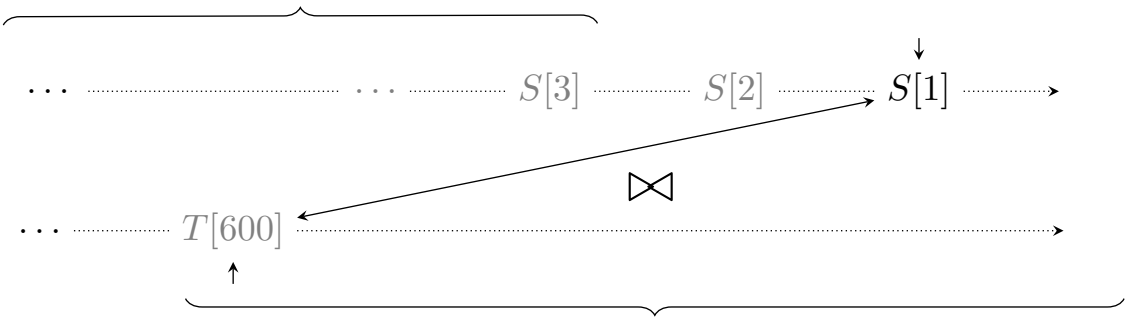

Abb. 44: Blockierender Buffer S nach drei Tupeln

Wie in Abb. 44 ersichtlich, würden in den Puffer vom Datenstrom S drei Tupel zu den Zeitpunkten $\tau = 1$, $\tau = 2$ und $\tau = 3$ eingefügt werden. Ab dann würde der Puffer von S blockieren, bis eine andere Operation die *get*()-Operation dieses Datenstroms aufruft und damit einen freien Platz im Puffer schafft. An dieser Stelle ist das Problem ersichtlich, dass gemäß des Algorithmus nun das Tupel $S[1]$ mit $S[600]$ vereinigt werden würde (minimales Delta ist 599). Gleiches geschieht mit den Tupeln $S[2]$ und $S[3]$. Es wird für diese drei Tupel also ein zum Zeitpunkt $\tau = 600$ ein Verbundpartner gefunden, allerdings kann der Puffer des Datenstromes nur drei Elemente halten. Dies bedeutet, dass $S[4]$ bis $S[599]$ nicht im Puffer liegen können.

Wesentlich optimer wäre es, die ältesten Elemente des Puffers zu überschreiben, sofern im anderen Stream noch kein Element für den Verbund existiert. Dies ist für den Minimal-Delta-Join möglich indem dies in den überschriebenen[74] Operationen `_enqueue(t)` und `put(t)` in der Klasse `StreamMDJSource` berücksichtigt wurde. Ist der Puffer voll, so wird nicht blockiert und gewartet, bis eine andere Funktion eine `get()`- und damit eine `_dequeue()`-Operation auf diesen Datenstrom bzw. diesen Puffer anwendet und damit ein Tupel aus dem Puffer löscht, sondern die ältesten Elemente im Puffer werden überschrieben. Damit wird sichergestellt, dass die Tupel in zwei Datenströmen zueinander ein geringeres Delta haben. Für das gegebene Beispiel in Abb. 44 würde dies bedeuten, dass nach dem Einfügen der drei Tupel $S[1]$ durch $S[4]$ überschrieben werden würde, danach würde $S[2]$ durch $S[5]$ überschrieben werden. Dies setzt sich solange fort, bis zum Zeitpunkt $\tau = 600$

[74] Override der entsprechenden Funktionen der Elternklasse.

die Tupel $S[598]$, $S[599]$ und $S[600]$ im Puffer befinden. Für alle Elemente mit einem kleineren Zeitstempel als $\tau = 597$ wird aufgrund der Beschränktheit des Puffers kein Verbundpartner gefunden.

6.8 Analyse des Minimal-Delta-Joins

An dieser Stelle soll analysiert werden, wie sich Parameter wie die Puffergröße und die Frequenz der Datenströme auf die Effizienz und auf die Abarbeitung des Algorithmus auswirken. Dazu werden verschiedene Puffergrößen und verschiedene Messfrequenzen gewählt und die Pufferauslastung sowie die gefundenen Tupel nach jedem gefundenen Verbundpartner gespeichert. Außerdem wird erfasst, wie viele Tupel überschrieben werden, etwa aufgrund eines zu kleinen Puffers. Die jeweiligen Ergebnisse werden geplottet und interpretiert.

Auf der x- und auf der ersten y-Achse der Plots wurden die Zeitstempel der Tupel dargestellt, die in einen Verbund aufgenommen wurde. In jedem der nachfolgenden Diagramme wurde zusätzlich ein Graph mit der Funktion $f(x) = y$ gezeichnet, der den Idealwert nach jedem Verbund darstellt. Punkte, die auf dieser Geraden liegen, stellen ein *Exact-Match* dar, da diese den gleichen Zeitstempel haben. Bei Punkten, bei denen das minimale Delta größer als Null ist, wird dies durch eine Abweichung von dieser Geraden in der x-Koordinate ersichtlich. Ein perfekter Verbund aller Tupel ist in Abb. 45 ersichtlich, bei dem die Zeitstempel der x-Werte mit den Zeitstempeln der y-Werte zu jedem Zeitpunkt übereinstimmen und damit einen Exact-Match darstellen. Dabei wurden in beiden Datenströmen Tupel mit einer Frequenz von einer Sekunde erzeugt. Beide Puffer konnten je acht Elemente speichern und besaßen entsprechen der Algorithmik – außer zu Beginn – dauerhaft jeweils zwei Elemente.

Für die Untersuchung, wie sich die Puffergröße auf die Effizienz des Algorithmus auswirkt, wurden jeweils 1000 Verbundoperationen durchgeführt, bei dem ein Datenstrom alle 0.1 s Sekunden ein neues Tupel enthielt und ein weiterer Datenstrom alle 1 s ein neues Tupel enthielt. Der Puffer wurden in vier verschiedenen Testläufen auf fünf, zehn, 15 oder 20 Elemente begrenzt. Dabei wurden neben den Zeitstempeln der Verbünde auch die Pufferbelegungen zum Zeitpunkt eines Verbundes sowie die kumulierte Anzahl aller bisher überschriebenen Tupel erfasst. In dieser Grafik sowie in allen nachfolgenden Grafiken werden die x- und die y-Achse für die Zeitstempel jeweils die gleiche Skala haben, da die Dominanz der Datenströme wechselt. Ähnlich gilt dies für den Puffer. Ein Datenstrom kann für die Erzeugung eines Datenpunktes sowohl in der Achse „*Pufferbelegung*

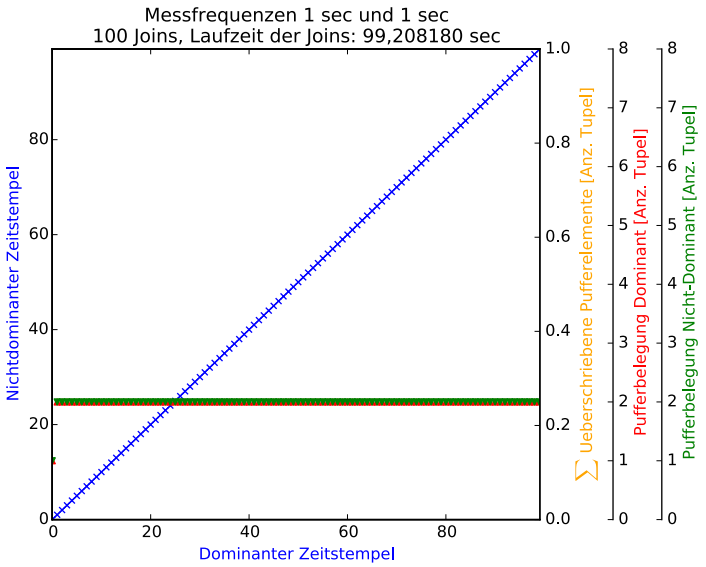

Abb. 45: Optimales Ergebnis des Minimal-Delta-Joins

Dominant“ als auch für die Erzeugung eines Datenpunktes in der Achse „*Pufferbelegung Nicht-Dominant*“ verantwortlich sein. Entsprechend entspricht die obere Skalengrenze der möglichen Pufferelemente im Plot immer dem größeren Puffer. Für alle Messungen wurden beide Puffer gleich groß definiert.

Während der Messreihen wurde die Beobachtung gemacht, dass die Geschwindigkeit der Durchführung von 1000 Minimal-Delta-Joins von der Größe der Eingangspuffer abhängt. Dies ist in den Abbildungen 46 – 49 erkennbar. Die genannten Abbildungen sind im Appendix A.8 – A.11 (S. 104 ff.) noch einmal größer dargestellt.

Bei der Generierung von Tupeln alle 0.1 s und alle 1 s ist zu erwarten, dass ein Puffer zehn Elemente und der andere Puffer zwei Elemente[75] im Puffer enthält, bevor es zum Verbund kommt. Dass dies tatsächlich gilt, ist in den Abbildungen 47 – 49 ersichtlich. In allen drei Diagrammen enthält der dominante Puffer zehn Elemente, bevor es zu einem Verbund kommt. Auf die ersichtlichen Abweichungen des Puffers in Abb. 48 und Abb. 49 auf neun Pufferelemente wird zu einem späteren Zeitpunkt in Abs. 6.9 ab S. 89 eingegangen.

Kann der Puffer für das gegebene Beispiel mehr als zehn Elemente enthalten, so ist in Abb. 48 und Abb. 49 ersichtlich, dass keine Tupel im Puffer überschrieben werden. Auf die Nötigkeit der Betrachtung der Problematik des Überschreibens von Puffereinträgen wurde bereits in Abs. 6.7 ab S. 83 eingegangen. Kann der Puffer nur zehn oder weniger Elemente

[75] Ausgehend vom dominanten Element je ein größeres und ein kleineres Tupel.

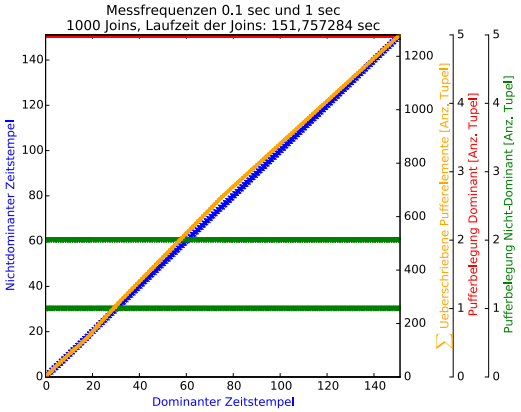

Abb. 46: Messergebnis mit Puffergröße = 5

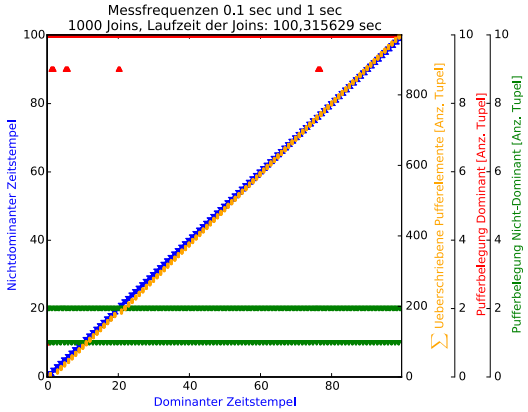

Abb. 47: Messergebnis mit Puffergröße = 10

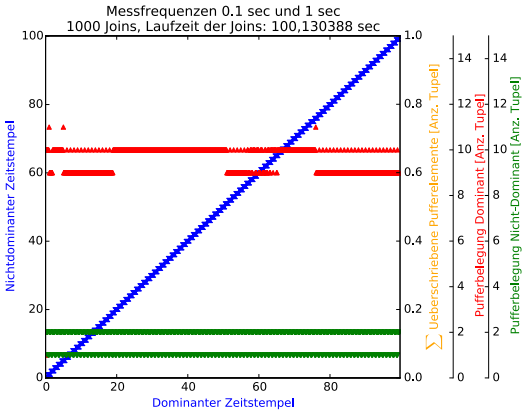

Abb. 48: Messergebnis mit Puffergröße = 15

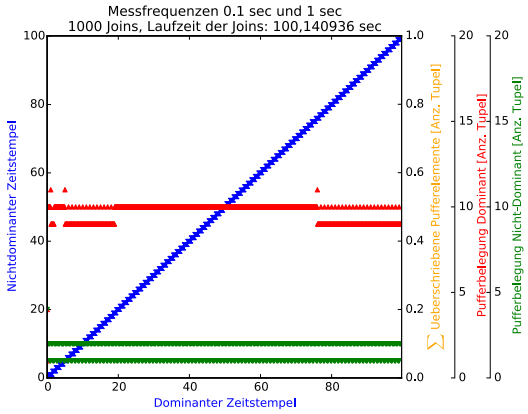

Abb. 49: Messergebnis mit Puffergröße = 20

halten, so tritt je nach Puffergröße ein Überschreibungsvorgang von Tupeln im Puffer auf. Bei weniger als zehn Elementen, so verlangsamt sich die Durchführung einer bestimmten Anzahl an Verbünden.

Es soll erklärt werden, warum in Abb. 47 bei 1000 Verbünden eine Überschreibung von 1000 Tupeln erfolgt. Dies lässt sich aus den Ausgaben des Algorithmus ableiten. Zum Zeitpunkt $\tau = 1$ enthält einer der Puffer den Wert `0.0` und der andere Puffer die Werte[76] `{[0.0], [0.1], [0.2], [0.3], [0.4], [0.5], [0.6], [0.7], [0.8], [0.9]}`. An dieser Stelle tritt ein Problem des Threadings auf, das später in Abs. 6.9 ab S. 89 erläutert wird. Das Resultat dieses Problems äußert sich darin, dass zum Zeitpunkt $\tau = 1$ das Tupel `[1.0]` zuerst im Puffer des Datenstroms eintrifft, das nur alle 1 s ein neues Tupel enthält und danach im Puffer des Datenstromes, durch welchen alle 0.1 s ein Tupel strömt. Da der Algorithmus wartet, bis der nicht-dominante Datenstrom mindestens ein größeres Tupel

[76] Der Einfachheit halber werden hier nur die Zeitstempel als Tupel angegeben. Im Demonstrator besteht das Tupel aus Zeitstempel und Datenwert.

enthält und das Tupel `[0.1]` eintrifft, werden zu diesem Zeitpunkt 10 Joins durchgeführt. Trifft im Puffer mit der Speicherfähigkeit von zehn Elementen nun auch das Tupel `[1.0]` ein, so ist der Puffer überfüllt und das Tupel [0.0] wird mit dem Tupel [1.0] überschrieben. Das gleiche passiert mit den neun anderen Tupeln `{[1.1], ..., [2.0]}`, bis das Tupel `[2.0]` im nicht-dominanten Datenstrom eintrifft. Der Wechsel der Dominanz und das entfernen der Tupel haben in diesem Fall aufgrund der spezifischen Implementierung keine Auswirkung. Den detaillierten Ablauf und das Auftreten ist im Appendix A.6 ab S. 108 ersichtlich. An dieser Stelle handelt primär um ein Phänomen, welches aufgrund der gewählten Variante der Implementierung auftritt.

Aufgrund einer analogen Argumentation werden Tupel bei einer Puffergröße von fünf Elementen überschrieben, wie in Abb. 46 dargestellt. Auffällig ist hier die Tatsache, dass insgesamt 1500 Tupel produziert werden müssen[77], damit 1000 Verbünde durchgeführt werden können. Es werden also 34 % der Tupel unverbunden ignoriert[78]. Dies ist aufgrund des kleinen Puffers ableitbar. Ein dominantes Tupel wird nur verbunden, wenn im nichtdominanten Datenstrom ein größeres Tupel auftaucht, da ansonsten kein minimales Delta berechnet werden kann. Zum beispielhaften Zeitpunkt $\tau = 1$ gilt, dass alle Tupel von `{[0.5], ..., [0.9]}` in den Verbund mit `[1.0]` einbezogen werden. Die `[0.5]` wird aufgrund des gleichen Deltas zusätzlich mit `[0.0]` verbunden. Die nachfolgenden fünf Tupel `{[1.0], ..., [1.4]}` werden aufgrund der Tatsache, dass während ihres Verbleibes im Puffer kein Tupel mit höherem Zeitstempel im Puffer des nicht-dominanten Datenstroms auftaucht, in keinen Verbund aufgenommen und von den Tupeln `{[1.5], ..., [1.9]}` überschrieben, die wiederum einen Verbund mit dem Tupel `[2.0]` eingehen. Pro zehn Tupel werden also nur sechs Verbünde durchgeführt.

Zusammenfassend ist durch die Verkleinerung eines Puffers Speicherplatz einsparbar. Trade-Off ist jedoch, dass innerhalb eines bestimmten Zeitraums deutlich weniger Tupelpartner gefunden werden.

6.9 Threading-Problematik des Demonstrators

Während der Implementierung des Demonstrators sind einige Problematiken aufgetreten, die abschließend benannt werden sollen. Insbesondere waren dies Problematiken des Multithreadings. Es war an einigen Stellen der Ausgabe des Demonstrators ersichtlich,

[77] Erkennbar an der logischen Uhr der x-achse 150 und dem Wissen, dass pro Sekunde 10 Tupel in einen Verbund aufgenommen werden müssten.

[78] 1500 Tupel dividiert durch 1000 Verbünde = Verbundrate von $0.6\overline{6}$.

dass sich einige Threads überholt haben. Insbesondere bei der Generierung der Diagramme in Abb. 48 und Abb. 49 (S. 88) trat das Problem auf, dass durch einen Datenstrom (Messfrequenz = 0.1 s) vor einem Verbund korrekterweise zehn Tupel erzeugt wurden und dann der Verbund durchgeführt wurde, da im anderen Datenstrom (Messfrequenz = 1 s) ein Verbundtupel generiert wurde. Nach etwa jeweils 150 generierten Tupeln trat dann das Problem auf, dass nur noch neun Tupel im ersten Datenstrom generiert wurden, bevor der zweite Datenstrom das entsprechende Verbundtupel generiert hat. Der erste Datenstrom wurde alle 150 Tupel langsamer.

Diese Problematik konnte behoben werden. Der Grund für dieses Phänomen lag in der Implementierung der Funktion, die die logische Uhrzeit mit dem Messwert des Sensors verknüpft hat. Folgende Implementierung wurde genutzt:

```
DECAY_HUMIDITY = 0.1

...

def fetchHumidityDataLC(oStream):
clock = float(0.0)
while True:
    humidity = sensorSource.get_humidity()
    humidity = [clock, float(humidity)]
    clock = clock + float(DECAY_HUMIDITY)
    logging.info("[==>] Humidity[%f, %f]", float(humidity[0]), float(
        ↪ humidity[1]))
    oStream.put(humidity)
    time.sleep(DECAY_HUMIDITY)
```

Die Implementierung dieser Funktion bestand aus dem Lesen des Datenwertes, der Verknüpfung mit der logischen Uhrzeit und dem Pausieren des Threads um 0.1 s. Dieser Ablauf wurde repetitiv in einer `while`-Schleife ausgeführt. Anfangs wurde hierbei nicht beachtet, dass das Erzeugen des Tupels ebenfalls eine bestimmte Zeit t braucht. Die gesamte Funktion `fetchHumidityDataLC(oStream)` benötigt als $t + 0.1\,\text{s}$, $t > 0$. Da die Funktion für die Bestimmung der Luftfeuchtigkeit zehn Mal in einer Sekunde aufgerufen wurde und die Funktion der Luftfeuchtigkeit nur einmal in der Sekunde aufgerufen wurde, gilt also $10 * t + 10 * 0.1s > t + 1s$. Dadurch entstand ein zeitlicher Drift zwischen den Funktionen, der ein ungenaues Bild der Pufferbelegung und des Verbundes gezeichnet hat. Wie stark dieser vermeintlich kleine zeitliche Drift den Plot beeinflusst hat, ist in Abb. 50 sichtbar. Möglicherweise ist dieser Effekt auch für die genannte Pufferabweichung in Abb. 48 und Abb. 49 verantwortlich.

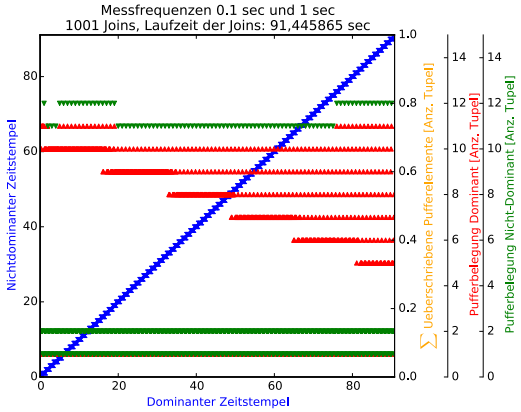

Abb. 50: Plot mit driftender logischer Uhr

Behoben wurde diese Abweichung durch eine Anpassung des `DECAY_HUMIDITY`-Parameters. Dieser wurde mit einem Dämpfungsfaktor multipliziert, sodass dieser Drift minimiert wird. Die letzte Zeile des obigen Codes wurde durch

```
time.sleep(DECAY_HUMIDITY * 0.995)
```

ersetzt. Der Faktor 0.995 wurde dabei explorativ ermittelt, sodass in der Ausgabe des Programmes und im Plot dieser Drift nicht mehr nachvollziehbar war. Dieser Dämpfungsfaktor ist systemspezifisch. Auf dem Raspberry Pi muss dieser Parameter kleiner sein, als auf der virtuellen Maschine, in der der Algorithmus programmiert und getestet wurde. Es ist zu vermuten, dass die Höhe der Faktors von der Systemleistung abhängig ist. Wie man die benötigte Größe dieses Dämpfungsparameters ermitteln kann muss bzw. welche Alternative es zu dieser Implementierung gibt, sei als Ausblick dahingestellt.

6.10 Hinweise zur Referenzumgebung

Die Implementierung der Algorithmen sowie die Generierung der Plots wurden auf einer virtualisierten Linux-Umgebung mit dem Betriebssystem Debian 8 durchgeführt. Die virtuelle Maschine besaß dabei einen Prozessor mit 2.90 GHz und einen Hauptspeicher von 2048 MB Das Hostsystem für die virtuelle Maschine war ein MacBook Pro mit dem Betriebssystem macOS Sierra 10.12.6, einem Intel Core i5 Prozessor und 16 GB Arbeitsspeicher.

Die Implementierung wurde nach Fertigstellung auf den Raspberry Pi portiert, dessen Spezifikationen in [Rasa] ersichtlich sind. Auf dem Raspberry Pi ist es ebenfalls

möglich, die Messwerte ebenfalls durch den Sense HAT Emulator generieren zu lassen oder tatsächliche Messwerte der Umgebung zu nutzen. Um zwischen emulierten Daten und echten Daten umzuschalten, wird auf Abs. 6.4 ab S. 77 verwiesen.

7 Zusammenfassung, Bewertung, Ausblick

Für die Anwendung von algebraischen Operationen auf Datenströmen wurde ein Demonstrator entwickelt, der Rücksicht auf die Eigenschaften strömender Daten nimmt. Insbesondere wurde Augenmerk auf die Tatsache gelegt, dass Datensätze im Allgemeinen nie als abgeschlossen angesehen werden und dass bei Verbundoperationen ein Exact-Match nicht immer möglich ist. Im Bezug auf Sensordaten wurde eine begrenzte Speichermöglichkeit von Sensordaten bei der Implementierung beachtet.

Auf konzeptueller Ebene wurde eruiert, wie man die Selektionsoperation, die Projektionsoperation, die Operation zum Bestimmen von Extremwerten (`MIN`, `MAX`) und die Verbundoperation auf kontinuierliche Daten anwenden kann. Die Selektions- und Projektionsoperation wurde analog zu den Operationen auf materialisierten Daten realisiert, indem eintreffende Tupel nach einem Attribut gefiltert wurden bzw. indem die Dimension des Tupels nach bestimmten Regeln reduziert wurde. Bei der Selektion sind aktuell nur konjunktive Anfragen möglich. Es gilt zu evaluieren, wie man eine allgemeinere Variante implementieren kann. Allerdings verlässt dies dieses Teilproblem der Verarbeitung strömender Daten und ist eher im Bereich der Datenbanktheorie (Query Rewriting, Query Containment) anzusiedeln. Es wurden zwei Varianten für die Extremwertbestimmung vorgestellt, bei der ein Extremum innerhalb eines Fensters oder eines ganzen Datenstroms gefunden werden kann. Für die Verbundoperationen wurden mehrere Varianten vorgestellt, mit der ein Join auch ohne Exact-Match realisiert werden kann. Insbesondere wurden dafür der (bufferlose) Fuzzy-Merge-Join und der Minimal-Delta-Join entwickelt, vorgestellt und implementiert und ein Ausblick auf eine möglicherweise ressourcensparendere Variante – der Predictive Minimal-Delta-Join – gegeben, der abschätzt, wann Tupel gelöscht werden können.

Während der Umsetzung hat sich insbesondere beim Minimal-Delta-Join gezeigt, dass diese Variante eine gute Variante für den Verbund darstellt, wenn die Messfrequenzen beider Sensoren etwa gleich schnell sind. Für sehr unterschiedliche Messfrequenzen und kleine Puffer hat sich in der Analyse gezeigt, dass der Puffer der Datenströme mehr Tupel zwischenspeichern müsste, als möglich ist und somit Tupel überschrieben werden. Als Ausblick muss hierfür eine Lösung gesucht werden, die dieses Problem vermeidet, falls ein

solches Verhalten nicht gewünscht ist.

Verbünde wurden bei den drei Verbundarten ausschließlich über ein Zeitattribut realisiert. Ein Verbund über andere Attribute wurde nicht betrachtet, da die Problematik besteht, dass der Verbund nur über einen Teilausschnitt der Daten – den Daten im Puffer – realisiert werden kann. Es ist zukünftig zu prüfen, ob hierfür eine Lösung gefunden werden kann, die einen Verbund über ein beliebiges Datenattribut zulässt. Theoretisch lassen sich die vorgestellten Algorithmen auf beliebige Attribute anwenden, allerdings fordern Operationen wie der Minimal-Delta-Join ein Vorliegen der Tupel in sortierter Reihenfolge, welche man bei Sensordaten nicht voraussetzen kann.

Bei allen Algorithmen wurde davon ausgegangen, dass alle Tupel rechtzeitig ankommen, d.h. nicht außerhalb des Fensters, in dem sie verarbeitet werden. Für Realbedingungen muss definiert werden, was in Fällen geschieht, indem Tupel zu spät eintreffen. Im einfachsten Fall werden sie gelöscht und für die Verarbeitung ignoniert.

Bei der Nutzung der vorgestellten Algorithmen muss insbesondere noch die Rolle von Veracity betrachtet werden. Werden bei einem Verbund zwei Tupel zusammengefügt, so enthält das Zieltupel nur einen der beiden – im Allgemeinen unterschiedlichen – Zeitstempel der Ausgangstupel. Wird dieses Zieltupel an weitere Komponenten, z.B. an eine weitere Verbundoperation, als Eingangstupel propagiert, so tritt das Problem auf, dass auch hier wieder nur einer der Zeitstempel ins Zieltupel übernommen wird. Insofern besteht die Möglichkeit, dass sich hier eine Fortpflanzung der Ungenauigkeit insbesondere der Zeitstempel der Tupel ergibt, die in einen Verbund aufgenommen wurde. Ob dies tatsächlich ein Problem darstellt und wie stark diese Ungenauigkeit Einfluss auf den korrekten Ablauf eines Programmes hat, bleibt zu ermitteln.

Als Stromdatenoperatoren wurden im Rahmen dieser Arbeit hauptsächlich Operationen der relationalen Algebra angesehen, die so transformiert wurden, dass diese auf strömenden Daten arbeiten. Fokus wurde dabei insbesondere auf die Selektion, die Projektion, die Extremwertbestimmung und den Verbund gelegt. In einer aufbauenden bzw. ergänzenden Arbeit gilt es perspektivisch zu untersuchen, wie auch andere Operationen der relationalen Algebra, wie etwa die Schnittmenge oder die Differenz, auf strömenden Daten anwendbar sind. Auch für spezielle Probleme, etwa dem holistischen `Top-k`-Problem, gilt es zu untersuchen, ob diese nur innerhalb eines Fensters ausgeführt werden können oder ob es optimalere Varianten gibt, die im Kontext der Verarbeitung strömender Daten anwendbar sind.

8 Literaturverzeichnis

[ABB+04] Arasu, A.; Babcock, B.; Babu, S. et al.: STREAM: The Stanford Data Stream Management System / Stanford InfoLab. Version: 2004. http://ilpubs.stanford.edu:8090/641/. Stanford InfoLab, 2004 (2004-20). – Technical Report

[ABW06] Arasu, Arvind; Babu, Shivnath; Widom, Jennifer: The CQL Continuous Query Language: Semantic Foundations and Query Execution. In: *The VLDB Journal—The International Journal on Very Large Data Bases* 15 (2006), Nr. 2, S. 121–142. – DOI 10.1007/s00778–004–0147–z

[Atm16] Atmel: *SMART ARM-Based Microcontrollers – SAM D20E / SAM D20G / SAM D20J*. September 2016. – Data Sheet

[BBD+02] Babcock, Brian; Babu, Shivnath; Datar, Mayur et al.: Models and Issues in Data Stream Systems / Stanford InfoLab. Version: 2002. http://ilpubs.stanford.edu:8090/535/. Stanford InfoLab, 2002 (2002-19). – Technical Report

[Bos15] Bosch SensorTec: *BMF055 – Custom programmable 9-axis motion sensor*. https://ae-bst.resource.bosch.com/media/_tech/media/datasheets/BST_BMF055_DS000_01.pdf. Version: November 2015, Abruf: 11.07.2017. – Data Sheet

[CHK+04] Cammert, Michael; Heinz, Christoph; Krämer, Jürgen et al.: Anfrageverarbeitung auf Datenströmen. In: *Datenbank-Spektrum* 4 (2004), Nr. 11, S. 5–13

[Con07] Conway, Neil: An Introduction To Data Stream Query Processing / Amalgated Insight, Inc. Version: 2007. https://www.pgcon.org/2007/schedule/attachments/17-stream_intro.pdf, Abruf: 18.06.2017. 2007. – Forschungsbericht. – PGCon 2007

[Dat12] Datenbanken Online Lexikon: *Relationale Algebra (RA)*. http://wikis.gm.fh-koeln.de/wiki_db/Datenbanken/Relationale-Algebra. Version: 2012, Abruf: 01.09.2017. – FH Köln

[DFMG16] De Francisci Morales, Gianmarco; Gionis, Aristides: Streaming similarity self-join. In: *Proceedings of the VLDB Endowment* 9 (2016), Nr. 10, S. 792–803

[Eur12] European Organization for Nuclear Research (CERN): *Computing*. http://cds.cern.ch/record/1997391. Version: August 2012, Abruf: 18.06.2017. – Cern Document Server

[GG07] Gama, Joao (Hrsg.); Gaber, Mohamed M. (Hrsg.): *Learning From Data Streams. Processing Techniques in Sensor Networks*. Springer, 2007. – ISBN 9–783–54073678–3

[GH17] Grunert, Hannes; Heuer, Andreas: Rewriting Complex Queries from Cloud to Fog under Capability Constraints to Protect the Users' Privacy. In: *Open Journal of Internet Of Things (OJIOT)* 3 (2017), Nr. 1, 31–45. http://nbn-resolving.de/urn/resolver.pl?urn=urn:nbn:de:101:1-2017080613421. – URN urn:nbn:de:101:1–2017080613421. – ISSN 2364–7108. – Special Issue: Proceedings of the International Workshop on Very Large Internet of Things (VLIoT 2017) in conjunction with the VLDB 2017 Conference in Munich, Germany.

[GM04] Gehrke, Johannes; Madden, Samuel: Query Processing in Sensor Networks. In: *IEEE Pervasive computing* 3 (2004), Nr. 1, S. 46–55. – DOI 10.1109/MPRV.2004.1269131

[GÖ10] Golab, Lukasz; Özsu, M. Tamer: *Data Stream Management*. Morgan & Claypool Publishers LLC, 2010. – ISBN 9–781–60845273–6

[GPN+17] Gulisano, Vincenzo; Papadopoulos, Alessandro V.; Nikolakopoulos, Yiannis et al.: Performance Modeling of Stream Joins. In: *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems* ACM, 2017, S. 191–202

[GR12] Gantz, John; Reinsel, David: The Digital Universe in 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East. In: *IDC iView: IDC Analyze the future* 2007 (2012), Nr. 2012, S. 1–16

[Gö14] Götz, Johannes: *Data Stream Processing*. http://wwwlgis.informatik.uni-kl.de/cms/fileadmin/courses/ws1314/Seminar/johannes/Data_Stream_Processing.pdf. Version: 2014, Abruf: 18.06.2017. – Seminararbeit Universität Kaiserslautern

[Hed17] Hedtstück, Ulrich: *Complex Event Processing: Verarbeitung von Ereignismustern in Datenströmen (eXamen.press) (German Edition)*. Springer Vieweg, 2017. – ISBN 9–783–66253451–9

[IOW01] Institut für Ostseeforschung Warnemünde: *Profilierende Verankerung im Gotlandbecken*. http://www.io-warnemuende.de/Profilierende_Verankerung.html, Abruf: 12.07.2017

[JSM+17] Jankov, Dimitrije; Sikdar, Sourav; Mukherjee, Rohan et al.: Real-time High Performance Anomaly Detection over Data Streams: Grand Challenge. In: *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems* ACM, 2017, S. 292–297

[KCC+03] Krishnamurthy, Sailesh; Chandrasekaran, Sirish; Cooper, Owen et al.: TelegraphCQ: An Architectural Status Report. In: *IEEE Data Eng. Bull.* 26 (2003), Nr. 1, S. 11–18

[KNV03] Kang, Jaewoo; Naughton, Jeffrey F.; Viglas, Stratis D.: Evaluating Window Joins over Unbounded Streams. In: *Data Engineering, 2003. Proceedings. 19th International Conference on* IEEE, 2003, S. 341–352

[KS04] Krämer, Jürgen; Seeger, Bernhard: PIPES – A Public Infrastructure for Processing and Exploring Streams. In: *Proceedings of the 2004 ACM SIGMOD international conference on Management of data* ACM, 2004, S. 925–926

[KTGH13] Klein, Dominik; Tran-Gia, Phuoc; Hartmann, Matthias: *Big Data*. https://www.gi.de/service/informatiklexikon/detailansicht/article/big-data.html. Version: 2013, Abruf: 18.06.2017. – Gesellschaft für Informatik, Informatiklexikon

[Kud15] Kudrass, Thomas: *Taschenbuch Datenbanken*. 2., neu bearbeitete Auflage. Carl Hanser Verlag GmbH Co KG, 2015. – ISBN 9–783–44643508–7

[Lan01] Laney, Doug: 3D Data Management: Controlling Data Volume, Velocity, and Variety. In: *META Group Research Note* (2001). https://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf, Abruf: 15.05.2017

[MW13] Meyer-Wegener, Klaus: *A Federated Data-Stream-Processing System*. Vortragsfolien, Juni 2013. – Kolloquiumsvortrag, Universität Rostock

[Mö16] Möller, Mark L.: *Aufbau einer Forschungsdatenverwaltung für chemische und physikalische In-Situ-Daten aus der Ostsee*. 2016. – Bachelorarbeit

[Ora09] Oracle Corporation: *Oracle CEP Getting Started*. https://docs.oracle.com/cd/E16764_01/doc.1111/e14476/overview.htm. Version: Oktober 2009, Abruf: 18.06.2017. – Onlinedokumentation

[ÖV11] Özsu, M. Tamer; Valduriez, Patrick: *Principles of Distributed Database Systems*. Springer Science & Business Media, 2011 (3. Auflage). – ISBN 9–781–44198833–1

[PSB11] Prien, Ralf D.; Schulz-Bull, Detlef E.: The Gotland Deep Environmental Sampling Station in the Baltic Sea. Version: 2011. In: IEEE (Hrsg.): *Oceans 2011 (Santander, Spain)*. New York: Curran Associates, Inc., 2011. – DOI 10.1109/Oceans–Spain.2011.6003606. – ISBN 9–781–45770086–6, S. 1624–1629

[Rasa] Raspberry Pi Foundation: *Raspberry Pi 3 Model B – Raspberry Pi*. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/, Abruf: 11.08.2017

[Rasb] Raspberry Pi Foundation: *Sense HAT – Raspberry Pi*. https://www.raspberrypi.org/products/sense-hat/, Abruf: 11.08.2017

[RTG14] Roy, Pratanu; Teubner, Jens; Gemulla, Rainer: Low-Latency Handshake Join. In: *Proceedings of the VLDB Endowment* 7 (2014), Nr. 9, S. 709–720

[Sei15] Seiler, Uwe: Zoo voller Gehege – Die wichtigsten Projekte der Hadoop-Community. In: *Heise iX Developer – Big Data* (2015), Nr. 2/2015. ISBN 4–192–04531290–6

[Sic1313] Sicular, Svetlana: *Gartner's Big Data Definition Consists of Three Parts, Not to Be Confused with Three „V"s*. https://www.forbes.com/sites/gartnergroup/2013/03/27/gartners-big-data-definition-consists-of-three-parts-not-to-be-confused-with-three-vs/#42f68c2b42f6. Version: 2013, Abruf: 15.05.2015
Abruf: 15.05.2017. Sorry, erst nach dem Druck bemerkt.

[SSH08] Saake, Gunter; Sattler, Kai-Uwe; Heuer, Andreas: *Datenbanken: Konzepte und Sprachen*. 3. Auflage. mitp Verlags GmbH & Co. KG, 2008. – ISBN 9–783–826616648

[SSH11] Saake, Gunter; Sattler, Kai-Uwe; Heuer, Andreas: *Datenbanken: Implementierungstechniken*. 3. Auflage. mitp Verlags GmbH & Co. KG, 2011. – ISBN 9–783–82669156–0

[TM11] Teubner, Jens; Mueller, Rene: How Soccer Players Would do Stream Joins. In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data* ACM, 2011, S. 625–636

[Tor12] Torres, Daniel: *Using sensor controllers to reduce power consumption in mobile computing*. http://www.embedded.com/design/power-optimization/4396126/Using-sensor-controllers-to-reduce-power-consumption-in-mobile-computing. Version: September 2012, Abruf: 11.07.2017. – Texas Instruments

[van15] van der Lans, Rick F.: Combining Data Streaming with Data Virtualization for Real-Time Analytics / R20/Consultancy. 2015. – Forschungsbericht. – Technical Whitepaper

[Wik17a] Wikipedia: *Datenstrom*. https://de.wikipedia.org/w/index.php?title=Datenstrom&oldid=165291196. Version: 2017, Abruf: 02.07.2017

[Wik17b] Wikipedia: *Raspberry Pi*. https://de.wikipedia.org/w/index.php?title=Raspberry_Pi&oldid=167427525. Version: 2017, Abruf: 11.08.2017

[Wik17c] Wikipedia: *Sensor hub*. https://en.wikipedia.org/w/index.php?title=Sensor_hub&oldid=789914549. Version: 2017, Abruf: 11.07.2017

[ZPD+13] Zikopoulos, Paul; Parasuraman, Krishnan; Deutsch, Thomas et al.: *Harness the Power of Big Data – The IBM Big Data Platform*. McGraw Hill Professional, 2013 ftp://public.dhe.ibm.com/software/pdf/at/SWP10/Harness_the_Power_of_Big_Data.pdf. – ISBN 9–780–07180818–7

Die englische Version des Abstracts wurde mithilfe von *DeepL* erstellt, einem KI-Übersetzungsdienst auf Basis eines neuronalen Netzwerkes (https://www.deepl.com/ [29.08.2017]).

A Appendix

A.1 Weitere Schritte des Minimal-Delta-Joins

Fortsetzung des Minimal-Delta-Join Algorithmus, der im regulären Teil ab S. 54 beendet wurde.

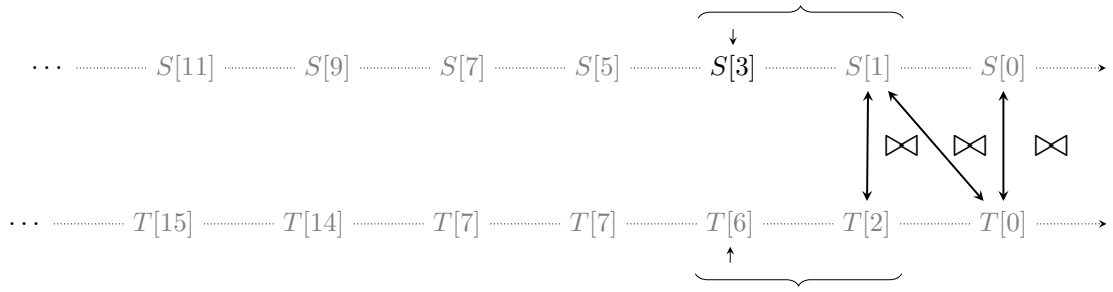

Abb. A.1: Minimal-Delta-Join mit drittem gefundenen Tupelpaar

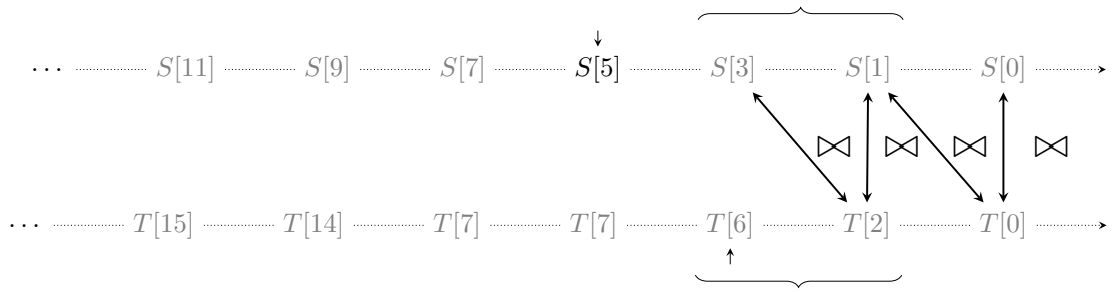

Abb. A.2: Minimal-Delta-Join mit viertem gefundenen Tupelpaar

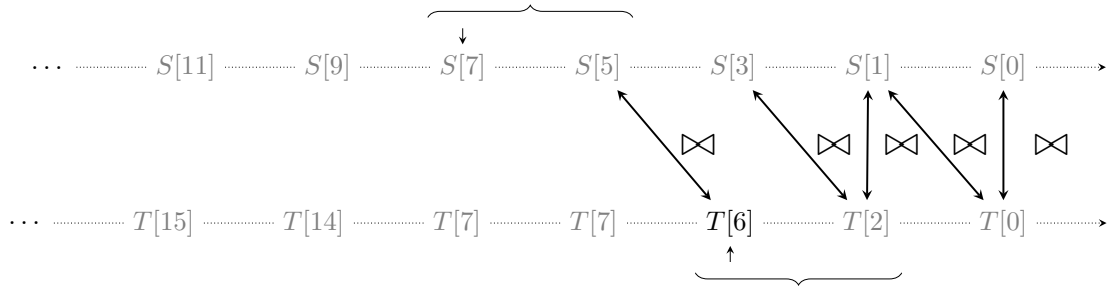

Abb. A.3: Minimal-Delta-Join mit fünftem gefundenen Tupelpaar

Gemäß Randbedingung 3 (vgl. S. 53 wird der von $T[6]$ ausgehenden Verbund hier nicht durchgeführt

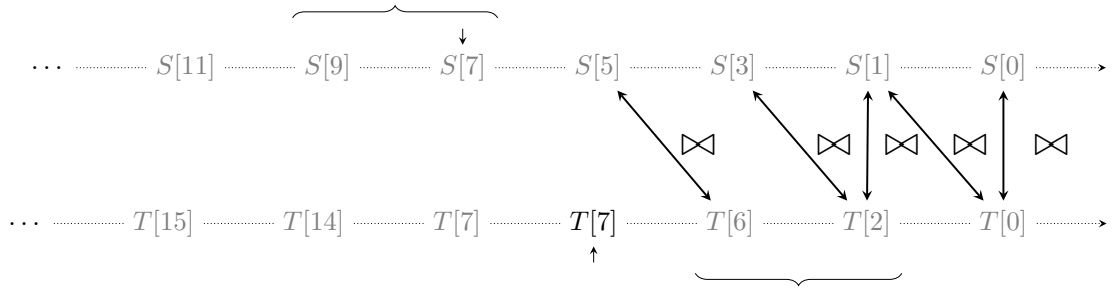

Abb. A.4: Minimal-Delta-Join mit übersprungenem Tupelpaar

Es wurde beschrieben, dass, sofern die beiden Elemente, auf die die Zeiger zeigen, den gleichen Zeitstempel, gilt, dass das Element aus dem Datenstrom Vorrang hat, dessen Datenstrom zuletzt Ausgangspunkt für den Verbund war. Dies ist an der Stelle $T[7]$, auch wenn im letzten Schritt kein Verbund erfolgt ist, da der potentielle Verbund von $T[7]$ ausging.

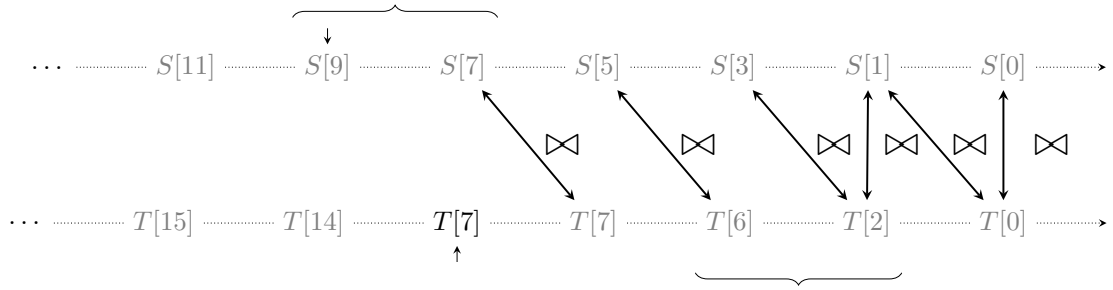

Abb. A.5: Minimal-Delta-Join mit sechstem gefundenen Tupelpaar

Der Zeiger von S wurde gemäß Randbedingung 1 (vgl. S. 53) nach dem Verbund ebenfalls weitergesetzt, da $S[7]$ und $T[7]$ die gleichen Zeitwerte hatten.

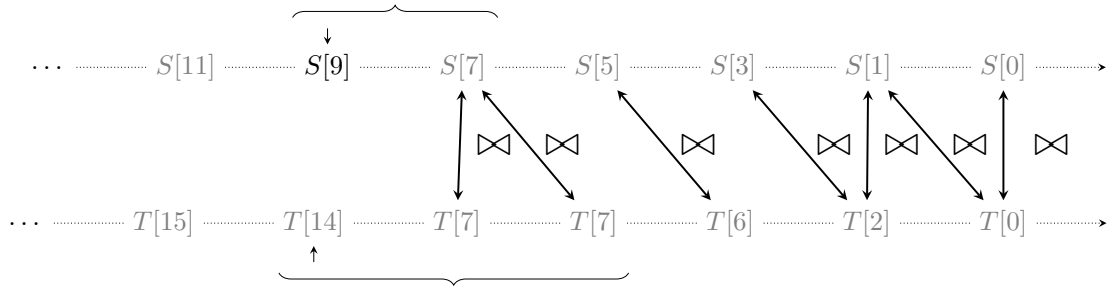

Abb. A.6: Minimal-Delta-Join mit siebtem gefundenen Tupelpaar

In Abb. A.6 erstreckt sich das Fenster von T gemäß das Algorithmus über drei Tupel, da zwei der Tupel den gleichen Zeitstempel haben. Entsprechend wird $S[9]$ mit beiden $T[7]$ Tupel in den Verbund aufgenommen, wie in Abb. A.7 ersichtlich.

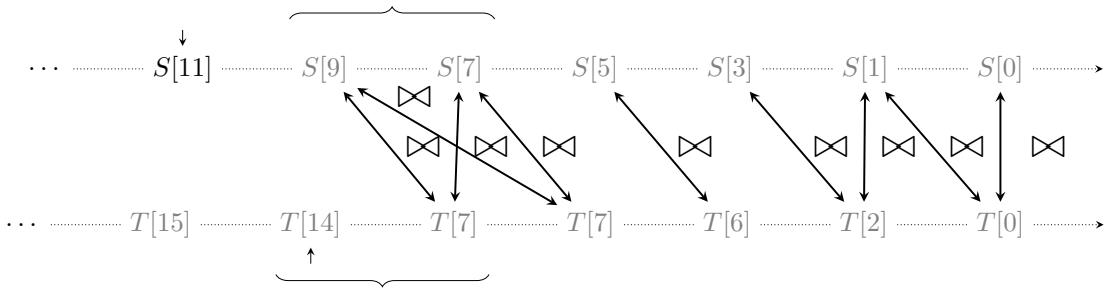

Abb. A.7: Minimal-Delta-Join mit achtem gefundenen Tupelpaar

A.2 Plot der Pufferanalyse mit bis zu fünf möglichen Pufferelementen

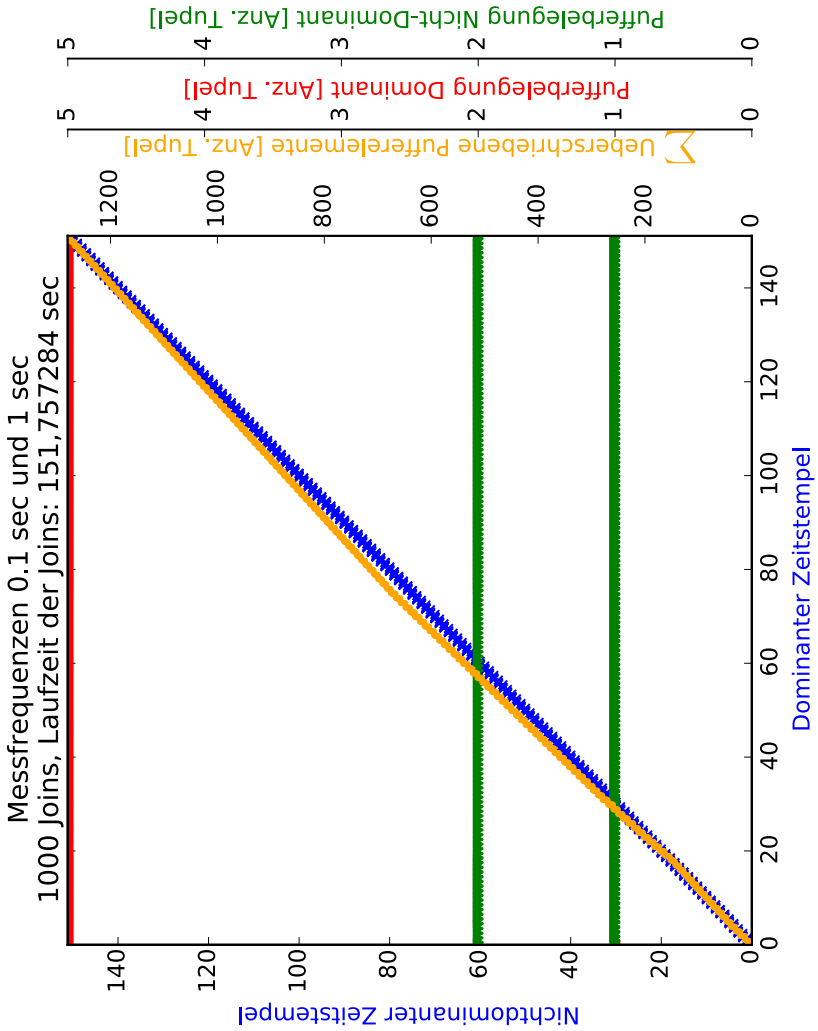

Abb. A.8: Plot der Pufferanalyse mit Puffergröße von fünf Elementen, Messfrequenzen 0.1 Sekunden und 1 Sekunde

A.3 Plot der Pufferanalyse mit bis zu zehn möglichen Pufferelementen

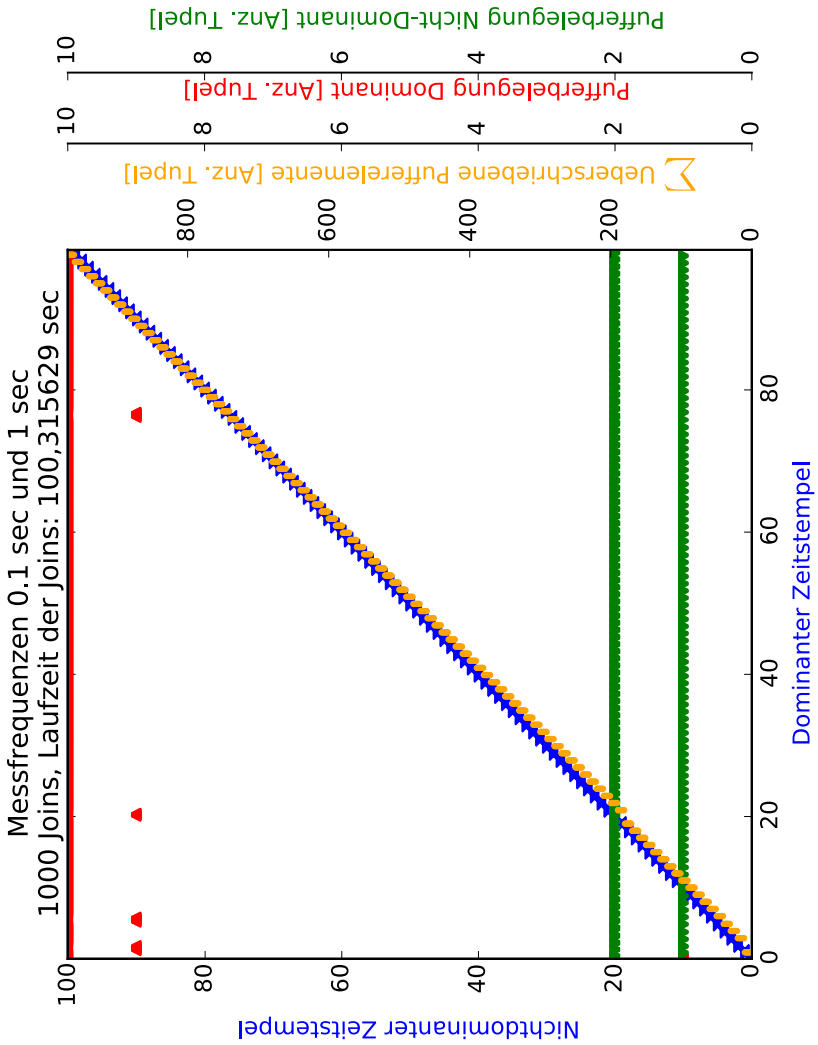

Abb. A.9: Plot der Pufferanalyse mit Puffergröße von zehn Elementen, Messfrequenzen 0.1 Sekunden und 1 Sekunde

A.4 Plot der Pufferanalyse mit bis zu 15 möglichen Pufferelementen

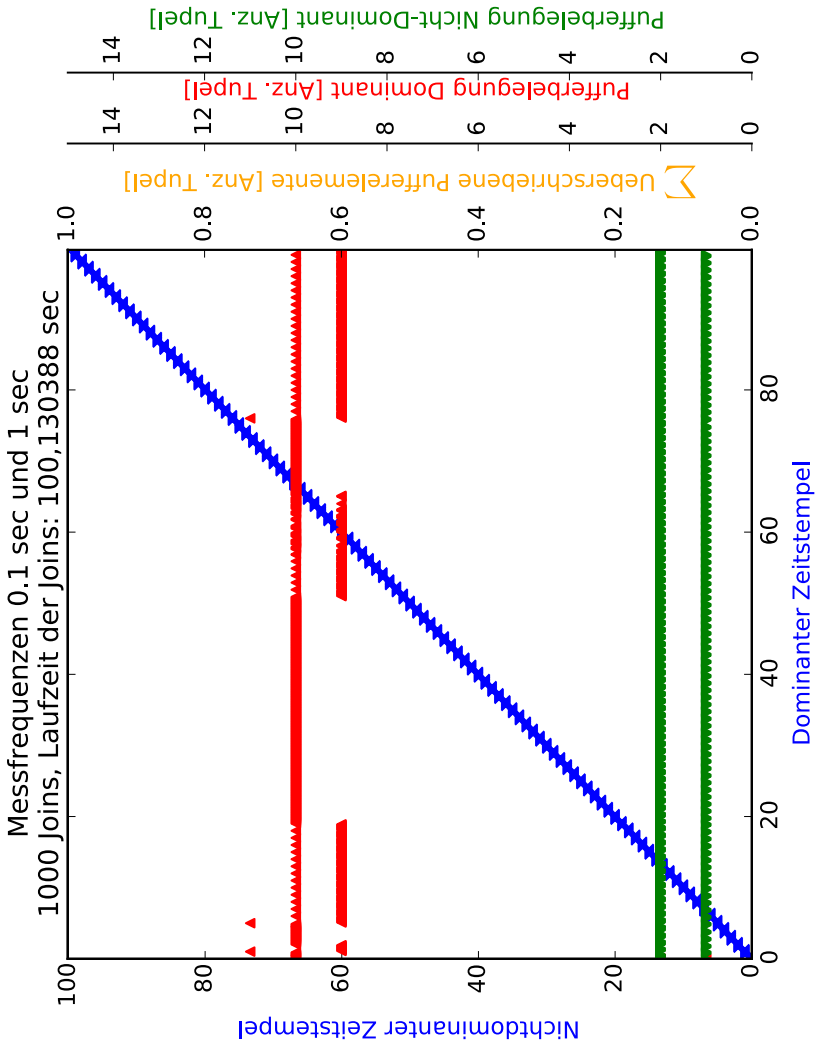

Abb. A.10: Plot der Pufferanalyse mit Puffergröße von 15 Elementen, Messfrequenzen 0.1 Sekunden und 1 Sekunde

A.5 Plot der Pufferanalyse mit bis zu 20 möglichen Pufferelementen

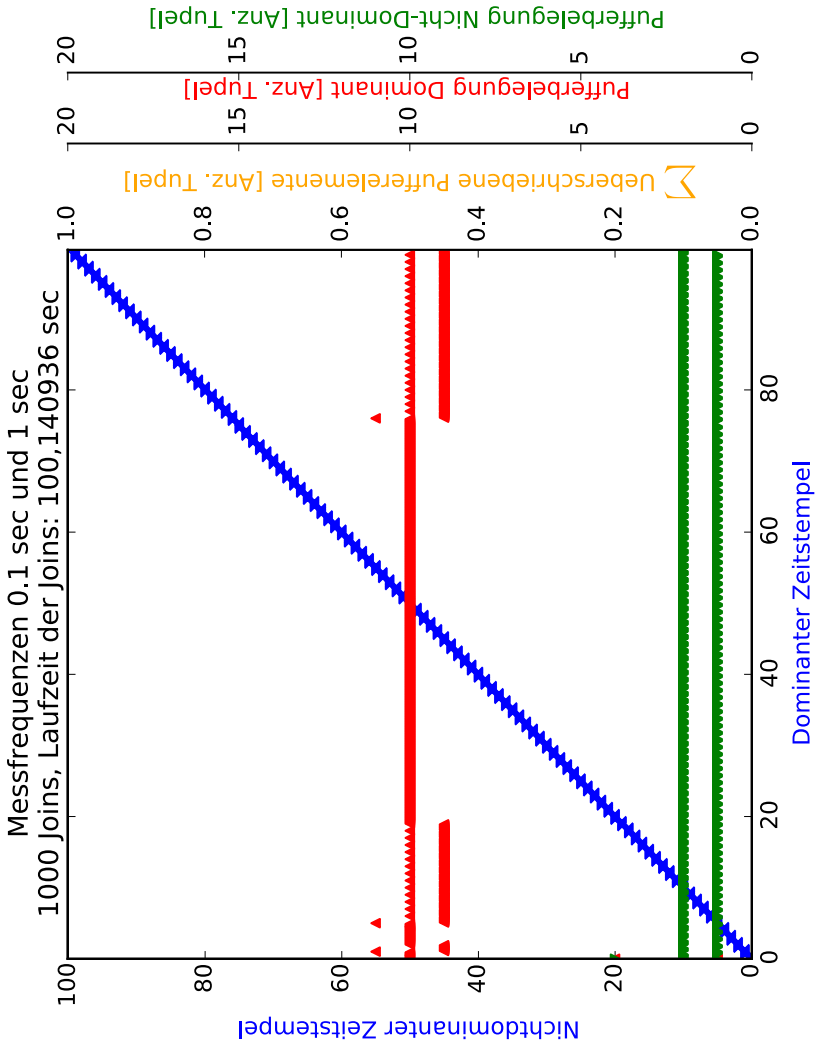

Abb. A.11: Plot der Pufferanalyse mit Puffergröße von 20 Elementen, Messfrequenzen 0.1 Sekunden und 1 Sekunde

A.6 Programmausgabe des Minimal-Delta-Join Programmes

Auszug der Ausgabe des Minimal-Delta-Joins für die Problematik des Verbundes von Elementen in Datenströmen mit kleinem Puffer, das in Abs. 6.8 ab S. 86 beschrieben wird.

```
[INFO   ] (md join   ) +++++ Beging New Minimal Delta Join Phase +++
[INFO   ] (md join   ) BUFFER DOM: stream(name=Humidity, q=[[0.0,
    ↪ 45.421875], [0.1, 45.421875], [0.2, 45.3671875],
    ↪ [0.30000000000000004, 45.3671875], [0.4, 45.328125], [0.5,
    ↪ 45.328125], [0.6, 45.4296875], [0.7, 45.4296875],
    ↪ [0.7999999999999999, 45.19921875], [0.8999999999999999,
    ↪ 45.19921875]], cnt=10, rpos=0, wpos=0, jp=9)
[INFO   ] (md join   ) BUFFER REC: streinfoam(name=Pressure, q=[None, [1.0,
    ↪  1013.00244140625], None, None, None, None, None, None, None],
    ↪  cnt=1, rpos=1, wpos=2, jp=1)
[INFO   ] (md join   ) Looking for a partner for [0.8999999999999999,
    ↪ 45.19921875]
[INFO   ] (md join   ) Looking for min-delta according to tuple with
    ↪ timestamp 0.900000
[INFO   ] (md join   ) Sliding Window of REC stream has a range of [1] (
    ↪ windowStart=1, windowEnd=1)
[INFO   ] (md join   ) Minimal delta found: 0.1
[INFO   ] (md join   ) New MDJ tuple found: [0.8999999999999999,
    ↪ 45.19921875, 1013.00244140625], derived from tuples
    ↪ [0.8999999999999999, 45.19921875] and [1.0, 1013.00244140625] (min-
    ↪ delta was 0.1)
[INFO   ] (md join   ) Removing all tuples with a smaller timestamp than
    ↪ 1.000000 from buffer of recessive stream
[INFO   ] (md join   )   BUFFER REC: stream(name=Pressure, q=[None, [1.0,
    ↪ 1013.00244140625], None, None, None, None, None, None, None],
    ↪ cnt=1, rpos=1, wpos=2, jp=1)
[INFO   ] (md join   ) +++++ End Of Minimal Delta Join Phase +++
[INFO   ] (md join   ) +++++ Beging New Minimal Delta Join Phase +++
[INFO   ] (humidity  ) [==>] Humidity[1.000000, 44.980469]
[WARNING] (humidity  ) Buffer is full! Overwriting oldest tuple [0.0,
    ↪ 45.421875] with new tuple [0.9999999999999999, 44.98046875]! (wpos=
    ↪ 0, rpos= 0, joinpos= 0)
[INFO   ] (humidity  ) [==>] Humidity[1.100000, 44.980469]
[WARNING] (humidity  ) Buffer is full! Overwriting oldest tuple [0.1,
    ↪ 45.421875] with new tuple [1.0999999999999999, 44.98046875]! (wpos=
    ↪ 1, rpos= 1, joinpos= 1)
[INFO   ] (humidity  ) [==>] Humidity[1.200000, 45.042969]
[WARNING] (humidity  ) Buffer is full! Overwriting oldest tuple [0.2,
    ↪ 45.3671875] with new tuple [1.2, 45.04296875]! (wpos= 2, rpos= 2,
    ↪ joinpos= 2)
[INFO   ] (humidity  ) [==>] Humidity[1.300000, 45.042969]
```

```
[WARNING] (humidity  ) Buffer is full! Overwriting oldest tuple
   ↪ [0.30000000000000004, 45.3671875] with new tuple [1.3, 45.04296875]!
   ↪  (wpos= 3, rpos= 3, joinpos= 3)
[INFO   ] (humidity  ) [==>] Humidity[1.400000, 45.011719]
[WARNING] (humidity  ) Buffer is full! Overwriting oldest tuple [0.4,
   ↪ 45.328125] with new tuple [1.4000000000000001, 45.01171875]! (wpos=
   ↪ 4, rpos= 4, joinpos= 4)
[INFO   ] (humidity  ) [==>] Humidity[1.500000, 45.011719]
[WARNING] (humidity  ) Buffer is full! Overwriting oldest tuple [0.5,
   ↪ 45.328125] with new tuple [1.5000000000000002, 45.01171875]! (wpos=
   ↪ 5, rpos= 5, joinpos= 5)
[INFO   ] (humidity  ) [==>] Humidity[1.600000, 45.187500]
[WARNING] (humidity  ) Buffer is full! Overwriting oldest tuple [0.6,
   ↪ 45.4296875] with new tuple [1.6000000000000003, 45.1875]! (wpos= 6,
   ↪ rpos= 6, joinpos= 6)
[INFO   ] (humidity  ) [==>] Humidity[1.700000, 45.187500]
[WARNING] (humidity  ) Buffer is full! Overwriting oldest tuple [0.7,
   ↪ 45.4296875] with new tuple [1.7000000000000004, 45.1875]! (wpos= 7,
   ↪ rpos= 7, joinpos= 7)
[INFO   ] (humidity  ) [==>] Humidity[1.800000, 44.878906]
[WARNING] (humidity  ) Buffer is full! Overwriting oldest tuple
   ↪ [0.7999999999999999, 45.19921875] with new tuple
   ↪ [1.8000000000000005, 44.87890625]! (wpos= 8, rpos= 8, joinpos= 8)
[INFO   ] (pressure  ) [==>] Pressure[2.000000, 1012.994873]
[INFO   ] (humidity  ) [==>] Humidity[1.900000, 44.878906]
[WARNING] (humidity  ) Buffer is full! Overwriting oldest tuple
   ↪ [0.8999999999999999, 45.19921875] with new tuple
   ↪ [1.9000000000000006, 44.87890625]! (wpos= 9, rpos= 9, joinpos= 9)
[INFO   ] (md join   ) BUFFER DOM: stream(name=Humidity, q
   ↪ =[[0.9999999999999999, 44.98046875], [1.0999999999999999,
   ↪ 44.98046875], [1.2, 45.04296875], [1.3, 45.04296875],
   ↪ [1.4000000000000001, 45.01171875], [1.5000000000000002,
   ↪ 45.01171875], [1.6000000000000003, 45.1875], [1.7000000000000004,
   ↪ 45.1875], [1.8000000000000005, 44.87890625], [1.9000000000000006,
   ↪ 44.87890625]], cnt=10, rpos=0, wpos=0, jp=0)
[INFO   ] (md join   ) BUFFER REC: stream(name=Pressure, q=[None, [1.0,
   ↪ 1013.00244140625], [2.0, 1012.994873046875], None, None, None, None,
   ↪  None, None, None], cnt=2, rpos=1, wpos=3, jp=1)
[INFO   ] (md join   ) Looking for a partner for [0.9999999999999999,
   ↪ 44.98046875]
[INFO   ] (md join   ) Looking for min-delta according to tuple with
   ↪ timestamp 1.000000
[INFO   ] (md join   ) Sliding Window of REC stream has a range of [1, 2] (
   ↪ windowStart=1, windowEnd=2)
[INFO   ] (md join   ) Minimal delta found: 1.11022302463e-16
```

```
[INFO   ] (md join   ) New MDJ tuple found: [0.9999999999999999,
    ↪ 44.98046875, 1013.00244140625], derived from tuples
    ↪ [0.9999999999999999, 44.98046875] and [1.0, 1013.00244140625] (min-
    ↪ delta was 1.11022302463e-16)
[INFO   ] (md join   ) Removing all tuples with a smaller timestamp than
    ↪ 1.000000 from buffer of recessive stream
[INFO   ] (md join   )   BUFFER REC: stream(name=Pressure, q=[None, [1.0,
    ↪ 1013.00244140625], [2.0, 1012.994873046875], None, None, None, None,
    ↪  None, None, None], cnt=2, rpos=1, wpos=3, jp=1)
[INFO   ] (md join   ) +++++ End Of Minimal Delta Join Phase +++
[INFO   ] (md join   ) +++++ Beging New Minimal Delta Join Phase +++
[INFO   ] (md join   ) BUFFER DOM: stream(name=Humidity, q
    ↪ =[[0.9999999999999999, 44.98046875], [1.0999999999999999,
    ↪ 44.98046875], [1.2, 45.04296875], [1.3, 45.04296875],
    ↪ [1.4000000000000001, 45.01171875], [1.5000000000000002,
    ↪ 45.01171875], [1.6000000000000003, 45.1875], [1.7000000000000004,
    ↪ 45.1875], [1.8000000000000005, 44.87890625], [1.9000000000000006,
    ↪ 44.87890625]], cnt=10, rpos=0, wpos=0, jp=1)
[INFO   ] (md join   ) BUFFER REC: stream(name=Pressure, q=[None, [1.0,
    ↪ 1013.00244140625], [2.0, 1012.994873046875], None, None, None, None,
    ↪  None, None, None], cnt=2, rpos=1, wpos=3, jp=1)
[INFO   ] (md join   ) [<->] MDJ Dominance swap! 1.000000 is now smaller
    ↪ than 1.100000
[INFO   ] (md join   ) Already joined candidate over here!. [1.0,
    ↪ 1013.00244140625] of the DOM stream was already a join partner when
    ↪ it was in the REC stream. Adapting logic and joining next element
[INFO   ] (md join   ) Removing all tuples with a smaller timestamp than
    ↪ 1.000000 from buffer of REC stream
[INFO   ] (md join   )   Removed [0.9999999999999999, 44.98046875] from REC
    ↪  buffer (threshold was 1.000000)
[INFO   ] (md join   )     Adapted lower end of sliding window of REC
    ↪ stream (windowEnd=1)
[INFO   ] (md join   )   BUFFER REC: stream(name=Humidity, q=[None,
    ↪ [1.0999999999999999, 44.98046875], [1.2, 45.04296875], [1.3,
    ↪ 45.04296875], [1.4000000000000001, 45.01171875],
    ↪ [1.5000000000000002, 45.01171875], [1.6000000000000003, 45.1875],
    ↪ [1.7000000000000004, 45.1875], [1.8000000000000005, 44.87890625],
    ↪ [1.9000000000000006, 44.87890625]], cnt=9, rpos=1, wpos=0, jp=1)
[INFO   ] (md join   )
```

A.7 Quellcode des Demonstrators (ohne PyDoc)

A.7.1 Programmpräambel

```
#!/bin/env python

vm = True

import sys
import threading
import time
import logging
import operator
import matplotlib.pyplot as plt
from datetime import datetime

if vm:
    from sense_emu import SenseHat
else:
    from sense_hat import SenseHat

DECAY_HUMIDITY = 1
DECAY_PRESSURE = 1 # bigger value if plotting
DECAY_COMPASS = 1.1
COLOR_BLUE = (0,0,255)

STAT_PLOT_STATISTICS = True
STAT_AMOUNT_OF_JOINS = 1000
STAT_UPPER_BORDER = DECAY_PRESSURE * STAT_AMOUNT_OF_JOINS
```

A.7.2 Implementierung der Klasse `Stream`

```
class Stream:
    def __init__(self, name, bufferSize, sid=None):
        self._name = name
        self._bufferSize = bufferSize
        self._buffer = [None] * bufferSize  # array with winsize None
            ↪ elements
        self._count = self._readPos = self._writePos = self.
            ↪ _nextCandidateToJoinPointer = 0
        self._mutex = threading.Condition()
        self._sid = sid

    def __len__(self):
        return len(self._buffer)  # or self._winsize
```

```
    def __str__(self):
        return "stream(name=%s, q=%s, cnt=%d, rpos=%d, wpos=%d, jp=%d)" % (
            self._name, str(self._buffer), self._count, self._readPos, self
                ↪ ._writePos, self._nextCandidateToJoinPointer)

    def _isfull(self):
        return self._readPos == self._writePos and self._count == self.
            ↪ _bufferSize

    def _isempty(self):
        return self._readPos == self._writePos and self._count == 0

    def getBufferSize(self):
        return self._bufferSize

    def _clearbuffer(self):
        self._mutex.acquire()
        self._buffer = [None] * self._winsize
        self._count = self._readPos = self._writePos = self.
            ↪ _nextCandidateToJoinPointer = 0
        self._mutex.notify()
        self._mutex.release()

    def _enqueue(self, t):
        self._count += 1
        self._buffer[self._writePos] = t
        if self._sid is not None and self._bufferSize <= 8:
            sensorSource.set_pixel(self._writePos, self._sid, COLOR_BLUE)
        self._writePos = (self._writePos + 1) % self._bufferSize

    def _dequeue(self):
        t = self._buffer[self._readPos]
        self._buffer[self._readPos] = None  # frees the object
        if self._sid is not None and self._bufferSize <= 8:
            sensorSource.set_pixel(self._readPos, self._sid, (0, 0, 0))
        self._readPos = (self._readPos + 1) % self._bufferSize
        self._count -= 1
        return t

    def _readWithoutDequeue(self):
        return self._buffer[self._readPos]

    # public methods, the stuff above should only be called withing the
        ↪ guarded commands acquire()/release() to be thread-safe
    def put(self, t):
        self._mutex.acquire()
        if self._isfull():
```

```
            logging.debug("WAITING Thread blocked by full buffer of output
                ↪ stream = %s" % str(self))
            self._mutex.wait()  # wait for tuples dequeued
            logging.debug("WORKING Buffer of output stream not full anymore
                ↪ ")
        self._enqueue(t)
        self._mutex.notify()
        self._mutex.release()

    def get(self):
        self._mutex.acquire()
        if self._isempty():
            logging.debug("WAITING Thread blocked by empty buffer of input
                ↪ stream")
            self._mutex.wait()  # wait for tuples enqueued
            logging.debug("WORKING Buffer of input stream not empty anymore
                ↪ . Buffer is now = %s" % str(self))
        t = self._dequeue()
        self._mutex.notify()
        self._mutex.release()
        return t
```

A.7.3 Implementierung der Klasse `StreamFMJSource`

```
class StreamFMJSource(Stream):
    def getFMJCandidate(self, dominantTime):
        candidateFound = False
        self._mutex.acquire()
        while not candidateFound:
            if(self._isempty()):
                logging.debug("WAITING Thread blocked by empty buffer of
                    ↪ input stream")
                self._mutex.wait()  # wait for tuples dequeued
                logging.debug("WORKING Buffer of input stream not empty
                    ↪ anymore. Buffer is now = %s" % str(self))
            t = self._dequeue()
            self._mutex.notify() # Notify threads which waited for a
                ↪ dequeue
            if t[0] < dominantTime: # t[0] represents always the time by
                ↪ assertion
                logging.debug("FMJ: %s is not a join partner for any
                    ↪ dominant tuple, because its timestamp is not >= than
                    ↪  %s, dropping this tuple", t, dominantTime)
            else:
                candidateFound = True
        self._mutex.release()
        return t
```

A.7.4 Implementierung der Klasse `StreamFMJBSource`

```
class StreamFMJBSource(Stream):
    def getFMJBcandidate(self, dominantTime):
        self._mutex.acquire()
        if (self._isempty()):
            logging.debug("WAITING Thread blocked by empty buffer of input
                ↪ stream")
            self._mutex.wait()  # wait for tuples dequeued
            logging.debug("WORKING Buffer of input stream not empty anymore
                ↪ . Buffer is now = %s" % str(self))
        t = self._readWithoutDequeue() # do not delete it from buffer yet
        if t[0] > dominantTime:
            return None
        else :
            self._dequeue() # dequeue it now, it is a matching partner
        return t
```

A.7.5 Implementierung der Klasse `StreamMDJSource`

```
class StreamMDJSource(Stream):
    def __init__(self, name, bufferSize, sid=None):
        parent = Stream.__init__(self, name, bufferSize, sid)
        self._windowStart = 0
        self._windowEnd = 0
        self.lastDominantElement = [-1,-1]
        self.statisticsOverwrittenElementCount = 0
        self.statisticsOverwrittenElements = []
        self.highestJoinedCandidate = [-1,-1]
        self.highestSelfJoinedElem = [-1, -1]

    def _enqueue(self, t):
        if not self._count == self._bufferSize: # don't exceed the count
            ↪ when overwriting tuples
            self._count += 1
        else: # full buffer, overwriting
            self.statisticsOverwrittenElementCount += 1
            self.statisticsOverwrittenElements.append([self.
                ↪ lastDominantElement[0], self.
                ↪ statisticsOverwrittenElementCount])
        self._buffer[self._writePos] = t
        self._windowEnd = (self._windowEnd + 1) % self._bufferSize
        if self._sid is not None and self._bufferSize <= 8:
            sensorSource.set_pixel(self._writePos, self._sid, COLOR_BLUE)
        self._writePos = (self._writePos + 1) % self._bufferSize

    def put(self, t):
```

```
        self._mutex.acquire()
        if self._isfull():
            logging.warning("Buffer is full! Overwriting oldest tuple %s
                ↪ with new tuple %s! (wpos= %s, rpos= %s, joinpos= %s)",
                ↪ self._readWithoutDequeue(), t, self._writePos, self.
                ↪ _readPos, self._nextCandidateToJoinPointer)
            if self._readPos == self._writePos:
                self._readPos = (self._readPos + 1) % self._bufferSize #
                    ↪ adapt readPos to next oldest value in ring buffer
                logging.warning("  Also incrementing readPos")
            if self._nextCandidateToJoinPointer == self._writePos:
                self._nextCandidateToJoinPointer = (self.
                    ↪ _nextCandidateToJoinPointer + 1) % self._bufferSize
                logging.warning("  Also incrementing joinPointer")
            if self._windowStart == self._writePos:
                self._windowStart = (self._windowStart + 1) % self.
                    ↪ _bufferSize
                logging.warning("  Also incrementing Sliding Window start
                    ↪ position")
            if self._windowEnd == self._writePos:
                self._windowEnd = (self._windowEnd - 1) % self._bufferSize
                logging.warning("  Also decrementing Sliding Window end
                    ↪ position")

        self._enqueue(t)
        self._mutex.notify()
        self._mutex.release()

    def statistics_get_buffer_items(self):
        self._mutex.acquire()
        fullItems = 0
        for x in self._buffer:
            if x is not None:
                fullItems += 1
        self._mutex.release()
        return fullItems

    def getCandidateIndexesList(self, bufferLabel, printSW = False):
        result = []
        currIdx = self._windowStart
        while True:
            result.append(currIdx)
           # if currIdx == self._windowEnd:
            if currIdx == self._windowEnd:
                break
            currIdx = (currIdx + 1) % self.getBufferSize()
        if printSW == True:
```

```
            logging.info("Sliding Window of %s stream has a range of %s (
                ↪ windowStart=%s, windowEnd=%s)", bufferLabel, result,
                ↪ self._windowStart, self._windowEnd)
        return result

    def incrementNextCandidateToJoinPointer(self):
        self._nextCandidateToJoinPointer = (self.
            ↪ _nextCandidateToJoinPointer + 1) % self._bufferSize

    def readAtNextCandidateToJoinPointer(self):
        self._mutex.acquire()
        if self._isempty():
            logging.debug("WAITING Thread blocked by empty buffer of input
                ↪ stream")
            self._mutex.wait()  # wait for tuples dequeued
            logging.debug("WORKING Buffer of input stream not empty anymore
                ↪ . Buffer is now = %s" % str(self))
        if self._buffer[self._nextCandidateToJoinPointer] is None:
            logging.info("WAITING FOR NEW ITEM = %s" % str(self))
            self._mutex.wait()
        # candidate = self._readWithoutDequeue()
        candidate = self._buffer[self._nextCandidateToJoinPointer]
        # self._mutex.notify()
        self._mutex.release()
        return candidate

    def getMDJMinimalDelta(self, dominantTime):
        logging.info("Looking for min-delta according to tuple with
            ↪ timestamp %f", float(dominantTime))
        minimalDelta = float("inf")
        minimalDeltaCandidates = []
        possibleMDCandidates = self.getCandidateIndexesList("REC", True)
        while len(minimalDeltaCandidates) == 0: # while no candidates found
            self._mutex.acquire()
            #logging.debug("BUFFER of REC S is %s", self._buffer)
            if (self._isempty()):
                logging.debug("WAITING Thread blocked by empty buffer of
                    ↪ input stream")
                self._mutex.wait()  # wait for tuples dequeued
                logging.debug("WORKING Buffer of input stream not empty
                    ↪ anymore. Buffer is now = %s" % str(self))

            # Wait until there is at least one element in buffer that is
                ↪ bigger or equal than the current element
            oneBiggerElementFound = False
            while oneBiggerElementFound == False: # and self._isfull() ==
                ↪ False:
```

```
                for item in [self._buffer[i] for i in self.
                    ↪ getCandidateIndexesList("REC", False)]: # search for
                    ↪  bigger candidates inside the window
                    if item is not None:
                        if item[0] >= dominantTime:
                            logging.debug("Found at least one element with
                                ↪ bigger Timestamp: %s", item)
                            try:
                                self._windowEnd = self._buffer.index(item)
                            except ValueError:
                                logging.error("This should never happen.
                                    ↪ The found item is not in the list
                                    ↪ anymore")
                            oneBiggerElementFound = True
                        else:
                            self._windowEnd = (self._windowEnd +1) % self.
                                ↪ _bufferSize # otherwise we will never
                                ↪ find sth in the window

            # Search minimal delta in buffer
            for item in [self._buffer[i] for i in possibleMDCandidates]: #
                ↪ only determine min-delta inside the window
                if item is not None:
                    minimalDelta = abs(dominantTime - item[0]) if abs(
                        ↪ dominantTime - item[0]) < minimalDelta else
                        ↪ minimalDelta

            self._mutex.notify()
            self._mutex.release()
            logging.info("Minimal delta found: %s", minimalDelta)
            return minimalDelta

    def getMDJJoinCandidates(self, dominantTime, minimalDelta):
        logging.debug("Looking for MDJ candidates with minimal delta of %s"
            ↪ , minimalDelta)
        minimalDeltaCandidates = []
        while len(minimalDeltaCandidates) == 0:
            self._mutex.acquire()
            #logging.debug("BUFFER of REC S is %s", self._buffer)
            if (self._isempty()):
                logging.info("WAITING Thread blocked by empty buffer of
                    ↪ input stream")
                self._mutex.wait()  # wait for tuples dequeued
                logging.debug("WORKING Buffer of input stream not empty
                    ↪ anymore. Buffer is now = %s" % str(self))

            # Add items with minimal delta to list
```

```
            elementsInWindow = self.getCandidateIndexesList("REC")
            for item in [self._buffer[i] for i in elementsInWindow]: # Only
                ↪ look inside window
                if item is not None:
                    if abs(dominantTime - item[0]) == minimalDelta:
                        minimalDeltaCandidates.append(item)

                        if item[0] >= dominantTime:
                            # we have a min delta with a bigger value
                            # track this to avoid a double join after a
                                ↪ dominance swap
                            # currently we are in the recessive stream (
                                ↪ important for later!)
                            self.highestJoinedCandidate = item

                    #if item[0] - minimalDelta > dominantTime: # emulate
                        ↪ end of sliding window, do not check candidates
                        ↪ out of window
                    #    break;
                self._mutex.notify()
                self._mutex.release()
        logging.debug("Minimal Delta Candidate Set found: %s, candidates
            ↪ were tuples with indexes %s", minimalDeltaCandidates,
            ↪ elementsInWindow)
        return minimalDeltaCandidates

    def removeOldMDJItems(self, threshold, bufferLabel):
        logging.info("Removing all tuples with a smaller timestamp than %f
            ↪ from buffer of %s stream", float(threshold), bufferLabel)
        # pop items until threshold is reached. Works because items are
            ↪ assumed to be in order
        self._mutex.acquire()
        while True:
            t = self._readWithoutDequeue() # Possible to do this since
                ↪ elements are implicit ordered by time
            logging.debug("Reading %s", t)
            if t is not None:
                if t[0] < threshold:
                    self.get()
                    logging.info("  Removed %s from %s buffer (threshold
                        ↪ was %f)", t, bufferLabel, float(threshold))
                    # lowering end of sliding window
                    self._windowStart = (self._windowStart + 1) % self.
                        ↪ _bufferSize
                    logging.info("    Adapted lower end of sliding window
                        ↪ of %s stream (windowEnd=%s)", bufferLabel, self.
                        ↪ _windowStart)
```

```
                else:
                    # logging.info("  No tuples removed")
                     break
        logging.info("  BUFFER REC: %s", str(self))
        self._mutex.notify()
        self._mutex.release()
```

A.7.6 Auslesen der Sensordaten

```
def fetchHumidityData(oStream):
    while True:
        humidity = sensorSource.get_humidity()
        humidity = [time.time(), float(humidity)]
        logging.info("[==>] Humidity[%f, %f]", float(humidity[0]), float(
            ↪ humidity[1]))
        oStream.put(humidity)
        time.sleep(DECAY_HUMIDITY)

def fetchHumidityDataLC(oStream):
    clock = float(0.0)
    while True:
        humidity = sensorSource.get_humidity()
        humidity = [clock, float(humidity)]
        clock = clock + float(DECAY_HUMIDITY)
        logging.info("[==>] Humidity[%f, %f]", float(humidity[0]), float(
            ↪ humidity[1]))
        oStream.put(humidity)
        time.sleep(DECAY_HUMIDITY * 0.995)

def fetchPressureData(oStream):
    while True:
        pressure = sensorSource.get_pressure()
        pressure = [time.time(), float(pressure)]
        logging.info("[==>] Pressure[%f, %f]", float(pressure[0]), float(
            ↪ pressure[1]))
        oStream.put(pressure)
        time.sleep(DECAY_PRESSURE)

def fetchPressureDataLC(oStream):
    clock = float(0.0)
    while True:
        pressure = sensorSource.get_pressure()
        pressure = [clock, float(pressure)]
        clock = clock + float(DECAY_PRESSURE)
        logging.info("[==>] Pressure[%f, %f]", float(pressure[0]), float(
            ↪ pressure[1]))
        oStream.put(pressure)
```

```
        time.sleep(DECAY_PRESSURE)

def fetchCompass(oStream):
    while True:
        compass = sensorSource.get_compass();
        compass = [time.time(), float(compass)]
        logging.debug("Compass is %f at time %s", compass[1], compass[0])
        oStream.put(compass)
        time.sleep(DECAY_COMPASS)
```

A.7.7 Implementierung der Selektionsoperation

```
def selectionOperation(iStream, oStream, selectionPredicateList):
    while True:
        selectionPredicateSatisfied = True
        ops = {'>': operator.gt, '<': operator.lt, '>=': operator.ge, '<=':
            ↪  operator.le, '=': operator.eq,
               '!=': operator.ne}
        iStreamItem = iStream.get()

        for selectionCondition in selectionPredicateList:
            if ops[selectionCondition[1]](iStreamItem[selectionCondition
                ↪ [0]], selectionCondition[2]):
                pass
            else:
                selectionPredicateSatisfied = False
                logging.info("Ignored %s (condition %s is False)",
                    ↪ iStreamItem, str(selectionCondition))
                break

        if selectionPredicateSatisfied:
            logging.info("Added %s to OutputStream (conditions %s is/are
                ↪ true)", iStreamItem, str(selectionPredicateSatisfied))
            oStream.put(iStreamItem)
```

A.7.8 Implementierung der Projektion

```
def projectionOperation(iStream, oStream, projectionList):
    if 0 not in projectionList:
        logging.warning("Timestamp not in projection list! Automatically
            ↪ added index 0 to projection list!")
        projectionList = (0,) + projectionList
    while True:
        iStreamItem = iStream.get()
        oStreamItem = list(iStreamItem[i] for i in projectionList)
        logging.debug("%s is now %s (Projection on indexes %s)",
            ↪ iStreamItem, oStreamItem, projectionList)
```

```
        oStream.put(oStreamItem)
```

A.7.9 Implementierung der datenstrombasierten Extremwertbestimmung

```
def streamExtremaOperation(iStream, oStream, extremaPosition,
    ↪ extremaOperation):
    ops = {'>': operator.gt, '<': operator.lt}
    currentExtrema = float("-inf") if extremaOperation == ">" else float("
        ↪ inf") # Init with Anti-Extremum
    extremaOperationLabel = "maxima" if extremaOperation == ">" else "
        ↪ minima"
    while True:
        iStreamItem = iStream.get()
        if ops[extremaOperation](iStreamItem[extremaPosition],
            ↪ currentExtrema):
            currentExtrema = iStreamItem[extremaPosition]
            oStream.put(iStreamItem)
            logging.debug("Found new %s: %s in tuple %s",
                ↪ extremaOperationLabel, iStreamItem[extremaPosition],
                ↪ iStreamItem)
```

A.7.10 Implementierung der fensterbasierten Extremwertstimmung

```
def windowExtremaOperation(iStream, oStream, extremaPosition,
    ↪ extremaOperation):
    ops = {'>': operator.gt, '<': operator.lt}
    extremaOperationLabel = "maxima" if extremaOperation == ">" else "
        ↪ minima"
    while True:
        currentExtrema = float("-inf") if extremaOperation == ">" else
            ↪ float("inf")  # Init with Anti-Extrema
        buffCache = list(iStream._buffer) # list to avoid call by reference
        extremaItem = None
        logging.critical(str(buffCache))
        for bufferItem in buffCache:
            if bufferItem is not None:
                if ops[extremaOperation](bufferItem[extremaPosition],
                    ↪ currentExtrema):
                    currentExtrema = bufferItem[extremaPosition] # new
                        ↪ extrema founds
                    extremaItem = bufferItem
        if not (currentExtrema == float("-inf") or currentExtrema == float(
            ↪ "inf")): # Buffer consists of None
            oStream.put(extremaItem)
            logging.critical("Found %s: %s", extremaOperationLabel,
                ↪ currentExtrema)
```

```
        while buffCache == iStream._buffer: # Wait until buffer has new
            ↪ item
            pass
        if iStream._isfull():
            iStream.get() # start deleting old items
```

A.7.11 Implementierung des Fuzzy-Merge-Joins

```
def fuzzyMergeJoin(iStreamDominant, iStreamRecessive, oStream):
    while True:
        iStreamDominantItem = iStreamDominant.get()
        iStreamRecessiveItem = iStreamRecessive.getFMJCandidate(
            ↪ iStreamDominantItem[0])
        oStreamItem = iStreamDominantItem + iStreamRecessiveItem[1:] #
            ↪ ignore the time of iStreamRecessiveItem
        logging.debug("Joined tuple found: %s, consists of %s (dom) and %s
            ↪ (rec)", oStreamItem, iStreamDominantItem,
            ↪ iStreamRecessiveItem)
        oStream.put(oStreamItem)
```

A.7.12 Implementierung des bufferlosen Fuzzy-Merge-Joins

```
def fuzzyMergeJoinBufferless(iStreamDominant, iStreamRecessive, oStream):
    while True:
        iStreamDominantItem = iStreamDominant.get()
        iStreamRecessiveItem = iStreamRecessive.getFMJBcandidate(
            ↪ iStreamDominantItem[0])
        if iStreamRecessiveItem is None: # no matching partner
            logging.debug("No join partner for dominant tuple %s found,
                ↪ dropping this tuple", iStreamDominantItem)
        else: # matching partner found
            oStreamItem = iStreamDominantItem + iStreamRecessiveItem[1:] #
                ↪ ignore the time of iStreamRecessiveItem
            logging.debug("Joined tuple found: %s, consists of %s (dom) and
                ↪  %s (rec)", oStreamItem, iStreamDominantItem,
                ↪ iStreamRecessiveItem)
            oStream.put(oStreamItem)
```

A.7.13 Implementierung des Minimal-Delta-Joins

```
def minimalDeltaJoin(iStreamOne, iStreamTwo, oStream):
    mdjStart = datetime.now()
    listOfTimes = []
    listOfDominantBufferItems = []
    listOfRecessiveBufferItems = []
    numpyCnt = 0
```

```
fig, ax = plt.subplots()
axes = [ax, ax.twinx(), ax.twinx(), ax.twinx()]
fig.subplots_adjust(right=0.75)
axes[1].spines['right'].set_position(('axes', 1.15))
axes[2].spines['right'].set_position(('axes', 1.25))
axes[1].set_frame_on(True)
axes[1].patch.set_visible(False)

# Arbitrary initialization
dominantStream = iStreamOne
recessiveStream = iStreamTwo

while True:
    logging.info("")
    logging.info("+++++ Beging New Minimal Delta Join Phase +++")
    #time.sleep(0.1)


    while (dominantStream.readAtNextCandidateToJoinPointer())[0] <=
        ↪ dominantStream.highestSelfJoinedElem[0]:
        # MDJ ist faster than tuple generating
        # Full round in ring buffer reached
        # Slow it down a bit to avoid re-join of ring buffer items
        pass

    currentDominantCandidate = dominantStream.
        ↪ readAtNextCandidateToJoinPointer()
    currentRecessiveCandidate = recessiveStream.
        ↪ readAtNextCandidateToJoinPointer()
    logging.info("BUFFER DOM: %s", str(dominantStream))
    logging.info("BUFFER REC: %s", str(recessiveStream))

    # Compare candidates to check next dominant item
    if(currentRecessiveCandidate[0] < currentDominantCandidate[0]): #
        ↪ swap dominance if current recessive stream has smaller
        ↪ timestamp value at pointer
        logging.info("[<->] MDJ Dominance swap! %f is now smaller than
            ↪ %f", float(currentRecessiveCandidate[0]), float(
            ↪ currentDominantCandidate[0]))
        dominantStream._mutex.acquire()
        recessiveStream._mutex.acquire()
        tmp = dominantStream
        dominantStream = recessiveStream
        recessiveStream = tmp
        dominantStream._mutex.release()
        recessiveStream._mutex.release()
        del tmp
```

```
        # now we have to have a look in the dominant stream
        if dominantStream.readAtNextCandidateToJoinPointer() ==
            ↪ dominantStream.highestJoinedCandidate:
            logging.critical("Already joined candidate over here!. %s
                ↪ of the DOM stream was already a join partner when it
                ↪  was in the REC stream. Adapting logic and joining
                ↪ next element", str(dominantStream.
                ↪ highestJoinedCandidate))
            dominantStream.incrementNextCandidateToJoinPointer()
            recessiveStream.removeOldMDJItems(dominantStream.
                ↪ highestJoinedCandidate[0], "REC")
            continue # restart the MDJ process

        logging.info("  BUFFER DOM: %s", str(dominantStream))
        logging.info("  BUFFER REC: %s", str(recessiveStream))

    dominantCandidate = dominantStream.readAtNextCandidateToJoinPointer
        ↪ ()
    dominantStream.lastDominantElement = dominantCandidate

    # Check for: MDJ ist faster than tuple generating
    # Slow it down a bit to avoid re-join of ring buffer items
    if dominantStream.highestSelfJoinedElem[0] < dominantCandidate[0]:
        dominantStream.highestSelfJoinedElem = dominantCandidate

    logging.info("Looking for a partner for %s", dominantCandidate)
    minimalDelta = recessiveStream.getMDJMinimalDelta(dominantCandidate
        ↪ [0])
    recessiveCandidates = recessiveStream.getMDJJoinCandidates(
        ↪ dominantCandidate[0], minimalDelta)
    for candidate in recessiveCandidates:
        oStreamItem = dominantCandidate + candidate[1:]
        logging.fatal("New MDJ tuple found: %s, derived from tuples %s
            ↪ and %s (min-delta was %s)", oStreamItem,
            ↪ dominantCandidate, candidate, abs(oStreamItem[0] -
            ↪ candidate[0]))
        oStream.put(oStreamItem)

        # Track and plot statistics stuff
        listOfTimes.append([dominantCandidate[0], candidate[0]])
        listOfDominantBufferItems.append([dominantCandidate[0],
            ↪ dominantStream.statistics_get_buffer_items()])
        listOfRecessiveBufferItems.append([dominantCandidate[0],
            ↪ recessiveStream.statistics_get_buffer_items()])
        numpyCnt = numpyCnt + 1
        if STAT_PLOT_STATISTICS and numpyCnt == STAT_AMOUNT_OF_JOINS:
```

```
logging.info("[/\/] Drawing statistics...")
mdjEnd = datetime.now()

skippedElements = dominantStream.
    ↪ statisticsOverwrittenElements
logging.info("List of skipped elements: %s", str(
    ↪ skippedElements))

for ax in axes:
    ax.set_xlim(-0.01, listOfTimes[numpyCnt-1][0] + 0.1)
    ax.set_ylim(-0.01, listOfTimes[numpyCnt-1][1] + 0.1)
axes[0].set_xlabel("Dominanter Zeitstempel", color='Blue')
axes[0].set_ylabel('Nichtdominanter Zeitstempel', color='
    ↪ Blue')
axes[1].set_ylabel('Pufferbelegung Dominant [Anz. Tupel]',
    ↪ color='Red')
axes[2].set_ylabel('Pufferbelegung Nicht-Dominant [Anz.
    ↪ Tupel]', color='Green')
axes[3].set_ylabel('$\sum$ Ueberschriebene Pufferelemente [
    ↪ Anz. Tupel]', color='Orange')
#axes[1].set_autoscale_on(False)
#axes[2].set_autoscale_on(False)
axes[1].set_ylim([0, dominantStream._bufferSize])
axes[2].set_ylim([0, recessiveStream._bufferSize])
axes[0].scatter(*zip(*listOfTimes), marker='x')
axes[1].scatter(*zip(*listOfDominantBufferItems), color='
    ↪ Red', marker='^')
axes[2].scatter(*zip(*listOfRecessiveBufferItems), color='
    ↪ Green', marker='v')
if len(skippedElements) > 0:
    axes[3].set_ylim([0, len(skippedElements)])
    axes[3].scatter(*zip(*skippedElements), color='Orange',
        ↪  marker='.')
else:
    axes[3].set_ylim([0, 1])
# axes[3].plot(*zip(*skippedElements), color='Orange')

axes[0].plot([0, STAT_UPPER_BORDER], [0, STAT_UPPER_BORDER
    ↪ ])

runtime = mdjEnd - mdjStart
plt.title('Messfrequenzen ' + str(DECAY_HUMIDITY) + ' sec
    ↪ und ' + str(DECAY_PRESSURE) + ' sec\n' + str(
    ↪ numpyCnt) + ' Joins, Laufzeit der Joins: ' + str(
    ↪ runtime.seconds) + ',' + str(runtime.microseconds) +
    ↪  ' sec', fontsize=14)
plt.show()
```

```
                sys.exit()

        timeOfRecessiveCandidate = recessiveCandidates[0]
        recessiveStream.removeOldMDJItems(timeOfRecessiveCandidate[0], "
            ↪ recessive")
        # dominantStream.get() # wrong!!! cannot be removed !!!
        dominantStream.incrementNextCandidateToJoinPointer()

        logging.info("+++++ End Of Minimal Delta Join Phase +++")
        logging.info("")
```

A.7.14 Implementierung einer Datensenke

```
def sink(iStream):
    allMatches = []
    while True:
        num = iStream.get()
        allMatches.append(num)
        logging.info("Consumed %s", num)
```

A.7.15 Programm-Setup

```
sensorSource = SenseHat()
sensorSource.clear()
logging.basicConfig(level=logging.INFO, format='[%(levelname)-7s] (%(
    ↪ threadName)-10s) %(message)s')

if vm:
    logging.info("Using emulator package sense_emu")
else:
    logging.info("Using live data from sense_hat")

if DECAY_HUMIDITY > 0:
    logging.debug("Set humidity decay parameter to %s", DECAY_HUMIDITY)
else:
    logging.debug("No humidity decay, live data")
```

A.8 DVD

Dieser Arbeit liegt eine DVD bei, auf der folgende Inhalte zu finden sind:

- Die Masterarbeit als PDF-Dokument.
- Die Literaturliste der verwendeten Literatur im BibTeX(`.bib`) Format.
- Verwendete Artikel, die im Namen den Zitierschlüssel, den Autor und den Titel des Dokumentes tragen.
- Verwendete Webquellen, die im Namen die Zugriffszeit, den Zitierschlüssel und den Titel der Webseite tragen.
- Der Python-Quellcode des Demonstrators einschließlich eines Installationsscriptes für die Installation von Dependencies für Linux-Distributionen, die `apt-get` unterstützen.

Selbstständigkeitserklärung zur Masterarbeit

Hiermit erkläre ich, Mark Lukas Möller, dass ich die vorgelegte Masterarbeit zum Thema „*Stromdatenverarbeitung für Data-Science-Anwendungen basierend auf Sensordaten*“ selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe. Verwendete Hilfsmittel und Referenzen in Form von direkten oder sinngemäßen Zitaten, Bild- oder anderen Quellen wurden als solche kenntlich gemacht. Ferner versichere ich, dass diese Arbeit bisher nicht in gleicher oder ähnlicher Form als Prüfungsleistung einer Prüfungskommission vorgelegt wurde.

.....	.....
Ort, Datum	Unterschrift