Datenströme und Complex Event Processing

Timo Michelsen

Abstract: Datenströme sind potenziell unendliche Sequenzen von Datenelementen, welche von aktiven Datenquellen wie Sensoren o. Ä. gesendet werden. Konventionelle Datenbankmanagementsysteme (DBMS) sind nicht in der Lage, diese zu verarbeiten. Deswegen existieren Datenstrommanagementsysteme (DSMS), die speziell auf die Eigenschaften von Datenströmen zugeschnitten sind. Daher unterscheiden sie sich in vielen Punkten signifikant zu DBMS. Operatoren und Operatorgraphen ermöglichen es, Datenströme in Echtzeit zu verarbeiten. Gleichzeitig erlaubt das Fenster-Prinzip die Einschränkung der unendlichen Datenmenge auf ein endlichen Bereich. Somit werden Datenströme handhabbar. Sie können auch als Ereignisströme interpretiert werden. Mit dem Complex Event Processing ist eine Technologie vorhanden, um aus primitiven Elementen komplexe, aussagekräftige Ereignisse zu bilden und zu verarbeiten.

1 Einleitung

Datenströme sind potenziell unendliche i. d. R. zeitlich geordnete Sequenzen von Daten. Normale Datenbankmanagementsysteme (DBMS) sind nicht in der Lage, diese geeignet zu verarbeiten und zeitnah Ergebnisse zu liefern. Datenstrommanagementsysteme (DSMS) sind auf die Verarbeitung von Datenströmen spezialisiert. Sie sind zuverlässig, hoch skalierbar und adaptiv ausgelegt. Es existieren viele Anwendungsgebiete, in denen solche Systeme eingesetzt werden. Um dabei ein DSMS nicht jedes mal neu zu implementieren, existieren Frameworks wie Odysseus oder Aurora, die die Realisierung unterstützen. Da Datenelemente in Datenströmen in vielen Anwendungsgebieten primitive Ereignisse darstellen, ist es hilfreich, durch Definition von Ereignismustern komplexere Ereignisse erkennen zu können. Dies wird mittels Complex Event Processing umgesetzt.

Ziel dieser Arbeit ist es, einen groben Überblick über Datenströme und deren Eigenschaften und Verarbeitung zu geben. Darauf aufbauend wird ein kleiner Einstieg in Complex Event Processing gegeben.

Im folgenden Abschnitt 2 werden zunächst Datenströme erklärt. Dabei wird darauf eingegangen, was bei der Verarbeitung zu beachten ist. Darauf aufbauend wird in Abschnitt 3 genauer auf DSMS eingegangen, welche Unterschiede es zu konventionellen DBMS gibt, wie die Verarbeitung mittels kontinuierlichen Anfragen und Operatoren realisiert wird und wie insbesondere mit der potenziellen Unendlichkeit der Datenströme umgegangen wird. Anschließend werden einige weitere Aspekte von DSMS kurz angerissen und mit der prinzipiellen Architektur von DSMS abgeschlossen. In Abschnitt 4 wird dann auf das Konzept des Complex Event Processing eingegangen und einige Beispiele von CEP-Realisierungen gezeigt. Abschnitt 5 fasst diese Arbeit zusammen.

2 Datenströme

Im Gegensatz zu passiven, persistent gespeicherten Datenquellen wie Datenbanken liefern *aktive Datenquellen* selbständig kontinuierlich zeitbehaftete *Datenelemente* in Echtzeit. Dabei wird jedes Datenelement genau einmal gesendet [See09, Gra10]. Nur die Datenquelle hat die Kontrolle über Reihenfolge und Ankunftsrate der Daten. Das Resultat ist eine potenziell unendliche Datenfolge, ein *Datenstrom* [Krä07]. Sie ist *append-only*, d.h. neue Datenelemente werden an das Ende des Datenstroms angehängt und können nur sequenziell zugegriffen werden. Einzelne Datenelemente können i. d. R. nachträglich nicht mehr durch Update- oder Delete-Operationen verändert werden [See09].

Ein Beispiel für aktive Datenquellen sind Sensoren (bzw. Sensornetzwerke) von Windkraftanlagen, welche über ein weites Gebiet verteilt positioniert sind. Sie liefern kontinuierlich in bestimmten Zeitintervallen Informationen über die aktuell gelieferte Leistung, Windstärke und den eigenen (internen) Zustand [Gra10]. Weitere aktive Datenquellen finden sich in der Analyse von Netzwerken in der Informatik: durch geeignete Messsoftware kann die aktuelle Netzbelastung ermittelt werden um damit kritische Situationen wie Überlastungen zu erkennen [Gs03]. Ein drittes und letztes Beispiel sei das Verkehrsmanagement zu erwähnen: durch das gezielte Plazieren von statitionären Sensoren an Autobahnbrücken sowie mobilen Sensoren in Fahrzeigen könnten Positionsdaten erfasst werden. Diese könnten dann ausgewertet werden, um anschließend den Verkehr intelligent umzuleiten und Staus zu vermeiden [See09].

Aufgrund der potenziell unendlichen Größe der Datenströme ist die offensichtliche Lösung, alle gelieferten Datenelemente persistent zu speichern, nicht umsetzbar. Daher sind konventionelle Datenbankmanagementsysteme (kurz: DBMS) nicht für Datenströme geeignet [Gs03]. Systeme, die Datenströme verarbeiten, müssen sich an der Aktivität der Datenquellen orientieren: die Daten müssen direkt bei der Ankunft verarbeitet werden (*data-driven*). Dabei hat das System aufgrund der Eigenschaften von Datenströmen keinen Einfluss auf die Reihenfolge und Ankunftsrate der Datenelemente. Es muss für jedes Datenelement entscheiden, ob es nach der Verarbeitung sofort verworfen oder explizit für weitere Verarbeitungsschritte gespeichert werden soll (*One-Pass-Paradigma*). Die eigentliche Verarbeitung der Daten geschieht über sogenannte *kontinuierliche Anfragen*. Sie werden einmalig eingepflegt und liefern – auf Datenströme angewendet – kontinuierlich Ergebnisse [See09].

Es existieren unterschiedliche Techniken zur Realisierung solcher Systeme: die Verarbeitung kann fest verdrahtet in Hardware realisiert werden, über festen Programmcode oder aber dynamisch über Datenstrommanagementsysteme [Gra10]. Auf Letzteres wird im nächsten Abschnitt genauer eingegangen.

3 Datenstrommanagementsysteme (DSMS)

Datenstrommanagementsysteme (kurz: DSMS) sind Systeme, welche sich explizit mit der Verarbeitung von Datenströmen beschäftigen. Sie besitzen die nötigen Eigenschaften und

Funktionen, um Datenströme zeitnah zu verarbeiten und Ergebnisse nahezu in Echtzeit auszugeben. Dabei können sie durch ihre hohe Skalierbarkeit eine Vielzahl von kontinuierlichen Anfragen parallel bearbeiten, die auf multiplen, heterogenen Datenströmen basieren. Gleichzeitig sind sie in Bezug auf fluktuierende Datenraten und Laständerungen höchst adaptiv ausgelegt. Beispiele für DSMS sind Odysseus [BGJ+09], Aurora, Stream oder Pipes [See09]. Am Ende dieses Abschnitts wird etwas genauer auf diese eingegangen.

3.1 Unterschiede zu Datenbankmanagementsystemen (DBMS)

Durch die speziellen Eigenschaften von Datenströmen unterliegen die DSMS anderen Anforderungen als die der konventionellen DBMS. Eine Gegenüberstellung ist in Abbildung 1 zu sehen.

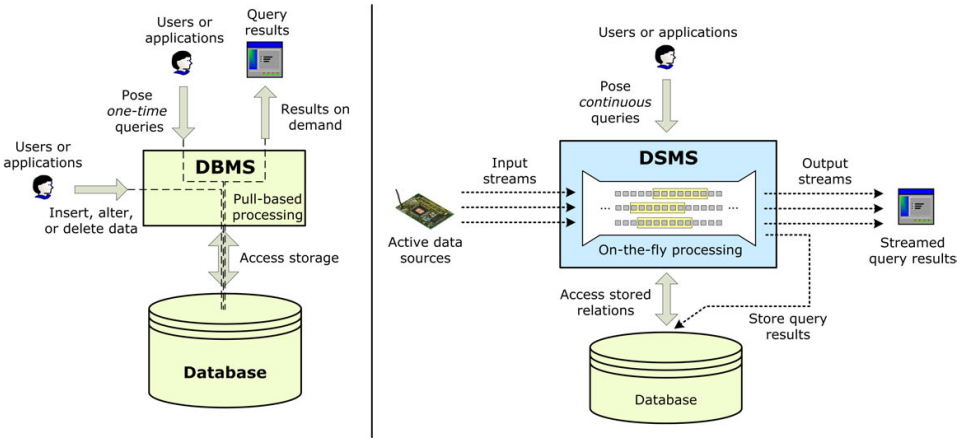

Abbildung 1: Vergleich von DBMS und DSMS [Krä07]

Dort ist rechts die Verarbeitung in einem konventionellen DBMS zu erkennen, links die eines DSMS. Die folgenden wesentlichen Unterschiede lassen sich erkennen:

Datenquellen Konventionelle DBMS verarbeiten persistente, *passive Datenquellen*. Die Datenelemente liegen zumeist im Externspeicher als Datenbank vor. Wie in Abschnitt 2 erwähnt, verarbeiten Datenstrommanagementsysteme aktive Datenquellen [See09]. Die Datenelemente werden zwecks schneller Verarbeitung hauptsächlich im Hauptspeicher gehalten [Krä07].

Datenzugriff DBMS besitzen wahlfreien Zugriff auf ihre persistenten Daten. Sie können zur jeder Zeit auf jedes beliebige Datenelement zugreifen, falls gefordert. In DSMS hingegen ist ausschließlich ein sequenzieller Zugriff möglich: Datenelemente können nur nacheinander eingelesen und verarbeitet werden. Muss auf ein Datenelement mehrfach zugegriffen werden, muss es explizit zwischengespeichert werden [Krä07].

Anfragetyp In DBMS werden historische Anfragen gestellt. Diese werden einmal abgearbeitet und liefern Ergebnisse zu einem bestimmten Zeitpunkt [See09]. Anschließend ist die Anfrage beendet [Krä07]. Im Gegensatz dazu werden in DSMS kontinuierliche Anfragen formuliert [See09]. Diese produzieren während der Verarbeitung der Datenströme kontinuierlich Ergebnisse [Krä07], welche entweder gespeichert und aktualisiert werden (z.B. bei Aggregationen) oder selbst wieder einen Datenstrom als Ausgabe liefern (z.B. Join oder Selektion) [BBD+02].

Verarbeitungsprinzip Bei konventionellen DBMS geschieht der Zugriff auf Daten erst bei Bedarf (*demand-driven*) [See09]: Erst durch das Stellen einer neuen Anfrage wird ein Zugriff auf den Datenbestand angestoßen [Krä07]. Da Datenströme in DSMS sofort verarbeitet werden müssen, wird der Zugriff auf die Daten direkt bei Ankunft ausgelöst [CHKS04].

Verarbeitungsstruktur Während DBMS mit Operatorbäumen arbeiten, wird die Abfrageverarbeitung in DSMS über Operatorgraphen verwaltet [See09]. Genaueres über Operatorgraphen wird in Abschnitt 3.2 erläutert.

Anfrageergebnisse In DBMS sind die Ergebnisse einer Anfrage stets exakt, da die Verarbeitung über den gesamten passiven Datenbestand erfolgen kann. In DSMS kann aufgrund der potenziellen unendlichen Sequenz der Datenelemente kein Zugriff auf alle Elemente erfolgen, sodass nur approximative Antworten möglich sind [Krä07]. Einer der Gründe für diese Ungenauigkeit ist die Tatsache, dass einige Operatoren wie Aggregation den kompletten Datenbestand benötigen, um exakte Ergebnisse liefern zu können. Die Problematik wird im folgenden Abschnitt im Zusammenhang mit Operatoren genauer erläutert.

3.2 Anfragen und Operatoren

DSMS nutzen kontinuierliche Anfragen, um Datenströme relativ zeitnah auszuwerten. Die meisten Implementierungen bedienen sich hierbei der Hilfe von *Operatoren*. Sie bilden ein zentrales Konzept: jeder Operator steht für einen einzelnen Verarbeitungsschritt (z.B. Projektion, Selektion). Sie nehmen einen oder mehrere Datenströme als Eingabeströme entgegen, verarbeiten die Datenelemente je nach Typ und liefern einen Ausgabestrom, welcher direkt ausgegeben oder für weitere Operatoren verwendet werden kann [See09].

Durch Verbinden von Operatoren entsteht ein *Operatorgraph*. Dabei handelt es sich um einen gerichteten, azyklischen Graphen. Die Knoten sind Operatoren, die Kanten Datenströme, welche zwischen ihnen fließen. Ein Operatorgraph beinhaltet eine oder mehrere kontinuierliche Anfragen, welche aktuell im DSMS verarbeitet werden. Wird zur Laufzeit eine neue Anfrage hinzugefügt, prüft das System, ob die Anfrage bereits im Operatorgraphen vorhanden ist. Ist dies nicht der Fall, so kann er entsprechend erweitert werden [See09].

Ein Beispiel für Operatorgraphen ist in Abbildung 2 zu sehen. Links ist ein prinzipieller Aufbau dargestellt, während rechts der Operatorgraph mittels einem konkreten Beispiels

verdeutlicht wird. Dabei sind drei verschiedene Arten von Operatoren zu erkennen: *Quellen* besitzen keinen Eingabestrom, sondern liefern nur einen Ausgabestrom. Sie stellen die Verbindung zu den aktiven Datenquellen dar. Im Gegensatz dazu besitzen *Senken* nur Eingabeströme. Sie können zum Beispiel Terminals, Bildschirme für die Ausgabe der Ergebnisse o. ä. sein. Alle anderen Operatoren haben Eingabe- und Ausgabeströme, können also gleichzeitig als Quelle und Senke bezeichnet werden [See09]. Dabei spielen Pufferoperatoren eine gesonderte Rolle: sie empfangen auch Datenströme, senden sie aber nicht sofort als Ausgabestrom weiter, sondern speichern sie ab und geben sie erst bei Bedarf an die nachfolgenden Operatoren weiter. Wann dies genau geschieht, wird von einem *Scheduler* bestimmt. In der Abbildung 2 ist die Positionierung der Pufferoperatoren willkürlich. Je nach Scheduling-Strategie können unterschiedliche Pufferplazierungsstrategien angewendet werden. Für genauere Informationen sei hier auf [BGJ+09] verwiesen.

Der Weg eines Datenelements ist nun wie folgt: zunächst wird es von einer aktiven Datenquelle (z.B. Sensor) erstellt und an das DSMS gesendet. Eine Quelle im Operatorgraphen empfängt es und wandelt es in ein internes Datenformat für die weitere Verarbeitung um. Anschließend wird es dem Operator geliefert. Nach der Verarbeitung wird es solange an die nächsten Operatoren weitergereicht und ggf. in Puffern vorgehalten, bis es an einer Senke ankommt, wo das Datenelement z.B. ausgegeben wird.

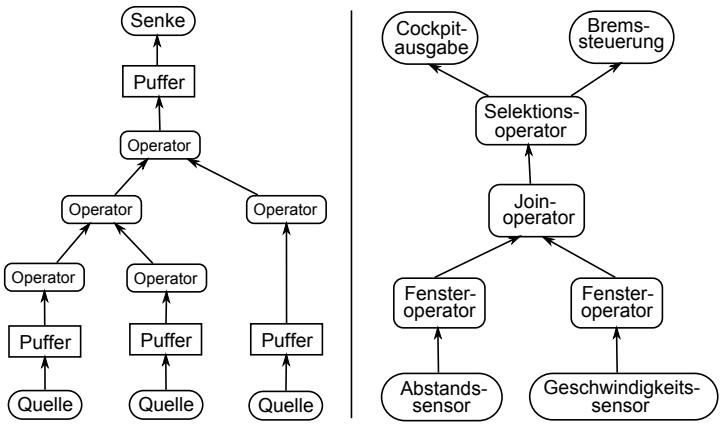

Abbildung 2: Zwei Beispiele von Operatorgraphen

Das konkrete Beispiel auf der rechten Seite soll eine mögliche Anwendung von Operatoren, Quellen und Senken im Autoverkehr präsentieren. Es wird angenommen, dass zwei Sensoren existieren: ein Abstandssensor vorne am Auto angebracht und ein Geschwindigkeitssensor, welches die aktuelle Geschwindigkeit des Autos liefert. Beide sind als Quellen im Operatorgraph platziert. Die beiden dahinter geschalteten Fensteroperatoren sorgen dafür, dass nur die relativ neuesten Werte betrachtet werden (vgl. dazu Abschnitt 3.3). Der Join-Operator verbindet beide Datenströme miteinander. Zuletzt prüft der Selektionsoperator, ob eine kritische Situation vorliegt, d.h. ob der Anstand zum vorderen Wagen bei der aktuellen Geschwindigkeit viel zu gering ist. Es könnte die Gefahr eines Auffahrunfalls bestehen. Ist dies der Fall, wo wird ein Warnsignal an das Cockpit gesendet und evtl.

gleichzeitig die Bremssteuerung für eine Notbremsung aktiviert. Es wurden keine Pufferoperatoren plaziert, damit eine schnelle Reaktionszeit erreicht werden kann.

3.3 Fenster

Nicht jeder Operatortyp kann direkt aus einem konventionellen DBMS entnommen und in einem DSMS eingesetzt werden. Da die Datenströme in Echtzeit verarbeitet werden, müssen die eingesetzten Operatoren ihre Ergebnisse so schnell wie möglich liefern. Jedoch können einige Operatoren nur genau dann Ausgaben produzieren, wenn sie die gesamte Menge an Eingabedaten gelesen haben. Ein Beispiel ist der Aggregationsoperator MAX, welcher aus einer gegebenen Menge an Zahlen den Maximalwert liefert. Um ein exaktes Ergebnis zu liefern, müsste dieser Operator alle Datenelemente eines Eingabestroms einlesen. Die potenziell unendliche Größe von Datenströmen macht dies unmöglich: MAX würde nie einen Maximalwert liefern, da ständig neue Datenelemente eingelesen werden müssen. Es handelt sich also um einen *blockierenden Operator*.

Viele Operatoren, welche statusbehaftet sind, sind blockierend. D.h. sie benötigen zur Ergebniserzeugung zuvor konsumierte Datenelemente, besitzen also einen internen Zustand zur Speicherung. Diese Operatoren müssen für das DSMS so angepasst werden, dass sie nicht mehr blockieren können. Beispiele sind Aggregationsoperatoren (MAX, MIN, SUM uw..) sowie Joins. Alle nicht statusbehafteten Operatoren können wie gewohnt verwendet werden [BGJ+09, CHKS04].

Zur Auflösung von Blockierungen existieren unterschiedliche Techniken. Eine davon ist das *Fenster-Prinzip*. Sie basiert darauf, in einer potenziell unendlichen Sequenz von Datenelementen eine endliche Menge als gültig zu markieren. Dabei gilt die Annahme, dass in der Regel nur aktuelle Daten relevant sind und weniger aktuelle Daten nicht weiter für die Berechnung betrachtet werden müssen (vgl. Abbildung 3) [See09]. Die zuvor blockierenden Operatoren erhalten eine Möglichkeit, ihre Operationen auf eine klar abgegrenzte Menge anzuwenden und Ergebnisse zu liefern. Nachteilig ist, dass die Ergebnisse nur noch approximativ sind.

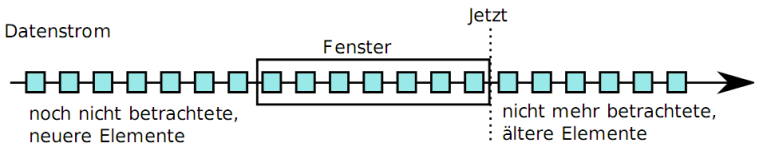

Abbildung 3: Funktionsweise eines Fensters

Beispiel Aggregationsoperator MAX: Ein Fenster bestimmt, dass nur die zehn aktuellsten Datenelemente für MAX relevant sein sollen. Dadurch liefert MAX immer das Maximum der letzten zehn Datenelemente. Im Vergleich zur konventionellen Implementierung (s.o.) liefert diese Version nur ein lokales Maximum. Je größer die Menge der gleichzeitig zu betrachtenden Datenelemente gewählt wird, desto genauer werden die Ergebnisse, benötigen

aber auch mehr Speicher und Prozessorleistung. Hier muss je nach Anwendungskontext entschieden werden. Es gibt unterschiedliche Konzepte zur Realisierung von Fenstern. Zur Klassifzierung werden daher folgende Kriterien verwendet [Gs03]:

Physisch, logisch oder prädikatenbasiert Bei *physischen* Fenstern wird die Größe des Fensters zeitenbasiert festgelegt. Ein Datenelement ist genau dann im Fenster, wenn es sich innerhalb der Zeitgrenzen des Fensters befindet. Die genaue Menge der Datenelemente ist beliebig. Bei *logischen* Fenstern definiert die Anzahl der Datenelemente die Fenstergröße. Es ist irrelevant, zu welchen Zeitpunkten die Datenelemente erschienen sind. *Prädikatenbasierte* Fenster definieren ihren Gültigkeitsbereich anhand der Daten selbst.

Beweglichkeit der Fensterendpunkte Bei einem *Fixed Window* sind beide Endpunkte fest. Sind beide beweglich, so handelt es sich um einen *Sliding Window*. Ist ein Endpunkt fest, während der andere beweglich ist, so wird das Fenster als *Landmark Window* bezeichnet.

Update-Intervall Bei welchen Anzahl n von Takten werden die Fensterendpunkte aktualisiert? Ist n größer als Eins, dann springt das Fenster. Ist n gleich der Größe des Fensters, so wird von einem *Tumbling Window* gesprochen. Dabei ist ein Takt entweder ein neues Datenelement (logisch) oder eine neue Zeiteinheit (physisch).

Im Allgemeinen werden Fenster als spezielle Operatoren realisiert, welche sich im Operatorgraphen direkt an die Quellen befinden. Neue Datenelemente erhalten sofort einen Gültigkeitsbereich zugewiesen. Alle nachfolgenden Operatoren sind selbst dafür verantwortlich, die Gültigkeitsbereiche zu prüfen und entsprechend Datenelemente zu betrachten oder zu ignorieren. Exemplarisch werden hier zwei Umsetzungen der Gültigkeiten erläutert [Gra10]:

Positiv-Negativ-Ansatz Jedes neu eintreffende Datenelement wird vom Fensteroperator mit einem positiven Marker versehen und weitergeleitet. Damit wird allen Operatoren mitgeteilt, dass das Datenelement ab sofort gültig ist. Der Fensteroperator merkt sich das Datenelement. Läuft die Gültigkeit einige Zeit oder Datenelemente später ab, so wird das gleiche Datenelement erneut gesendet. Diesmal mit einem negativen Marker und kennzeichnet das Datenelement als nun ungültig. Operatoren müssen bei allen Datenelementen die Marker überprüfen und dementsprechend reagieren (verarbeiten oder ignorieren).

Intervall-Ansatz Jedem neuen Datenelement wird ein Gültigkeitsintervall zugeordnet, welches sich aus Start- und Endzeitstempel zusammensetzt. Trifft nun bei einem Operator ein Datenelement mit einem Startzeitstempel ein, welcher größer ist als der Endzeitstempel eines vorangegangenen Elements, so ist das vorangegangene Element ungültig.

3.4 Weitere Aspekte: Scheduling und Metadaten

Neben Datenströmen, kontinuierlichen Anfragen, Operatoren und Operatorgraphen beinhalten Datenstrommanagementsysteme einige weitere Merkmale, die bei der Realisierung betrachtet werden müssen.

Wie oben in Abschnitt 3 beschrieben, sind DSMS im höchsten Maße adaptiv und skalierbar ausgelegt: sie passen sich zur Laufzeit den Ankunftsraten der Datenströme an und können mehrere kontinuierliche Anfragen parallel verarbeiten. Es ist daher wichtig, die verfügbaren Ressourcen wie Speicher und Prozessorleistung optimal einzusetzen. Wenn beispielsweise eine Datenquelle plötzlich eine sehr hohe Datenrate aufweist, so muss das DSMS den entsprechenden Operatoren im Operatorgraphen, welche genau diese Daten verarbeiten, mehr Speicher und Prozessorleistung zuordnen, um der Datenflut gerecht zu werden. Dies wird über geeignetes *Scheduling* erreicht. Der Scheduler ist eine Komponente im DSMS, welches für die Ablaufsteuerung, d.h. der Verteilung von Speicher und Prozessorleistung an die Operatoren, zuständig ist. Die Pufferoperatoren geben eine Möglichkeit, den Ablauf der Verarbeitung zu kontrollieren. Genaueres ist in [BGJ+09] an dem DSMS-Framework Odysseus erläutert.

Während der Verarbeitung der Daten benötigt ein DSMS neben den Datenelementen zusätzlich *Metadaten*. Sie stellen wichtige Hintergrundinformationen über Operatoren, Scheduling und Datenelementen dar. Beispiele sind Durchsatzraten und Speicherverbrauch. Auf Basis dieser Daten kann das Scheduling durchgeführt werden kontinuierliche Anfragen im Operatorgraphen (re-)optimiert werden. Zudem unterstützt es das Monitoring der DSMS. Näheres zu Metadaten kann in [BGJ+09] und [CHKS04] nachgelesen werden.

3.5 Archtiektur von DSMS

Durch die zuvor genannten Aspekte in der Realisierung von DSMS und Verarbeitung von Datenströmen, lässt sich eine abstrakte Architektur von DSMS konstruieren, wie in Abbildung 4 dargestellt. Dabei wird auch die prinzipielle Arbeitsweise von DSMS verdeutlichen: es existieren mehrere Sensoren, welche Datenströme senden. Ein DSMS empfängt diese als Eingangsdatenströme und übermittelt diese dem Operatorgraphen, welcher von einem Scheduler bzw. Query Processor ausgeführt wird. Dabei können auch persistente, statische Datenquellen angeschlossen werden, um beispielsweise die Daten im Datenstrom mit Richtwerten vergleichen zu können. Im Operatorgraphen sind alle benutzerdefinierten Anfragen zusammengefasst. Zu Laufzeit können neue Anfragen gestellt oder bereits vorhandene entfernt werden. Sind die Datenelemente durch den Operatorgraphen verarbeitet worden, werden die Ergebnisse als Ausgabedatenstrom nach außen gegeben.

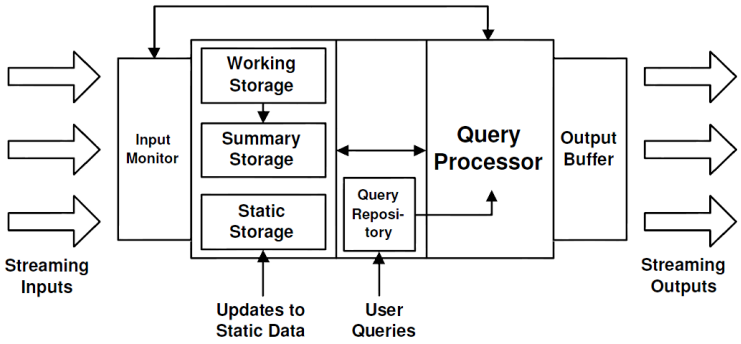

Abbildung 4: Abstrakte Architektur von DSMS [Gs03]

4 Complex Event Processing

In vielen Anwendungsgebieten werden die Datenelemente von Datenströmen als einfache Datencontainer angesehen: sie enthalten neben dem Erfassungszeitpunkt und einigen weiteren Metadaten die eigentlichen Daten der aktiven Quelle, welche in weiteren Verarbeitungsschritten innerhalb eines DSMS transformiert werden. Jedoch existieren Situationen, in denen die Datenelemente als *Ereignisse* (Events) angesehen werden.

Ein Ereignis ist ein Objekt, welche eine bereits geschehene Aktivität in einem System beschreibt. Beispielsweise kann ein Ereignis das Entnehmen eines Produktes aus einem Regal in einem Supermarkt darstellen. Ereignisse lassen sich in drei Aspekte aufteilen [Luc02]:

Form Beinhaltet die Datenkomponenten, welches das Ereignis näher beschreiben. Beispielsweise können Informationen wie Dauer des Ereignisses, sein Startzeitpunkt oder der Auslöser enthalten sein.

Signifikanz Aktivität, die durch das Ereignis beschrieben wird.

Relativität Die Relativität beschreibt die Beziehung des Ereignisses zu anderen Ereignissen. Dabei können sie zeitlich in beziehung stehen, aber auch durch Kausalität oder Aggregation gegeben sein.

Jedoch werden je nach Anwendungsszenario semantisch höherwertige Ereignisse benötigt, diese sich aber im Allgemeinen nicht direkt durch Sensoren beobachten lassen. Beispielsweise kann ein einzelner Sensor alleine keinen Ladendiebstahl feststellen.

Allerdings lassen sich komplexere Ereignisse durch Aggregationen aus mehreren elementaren Aktivitäten, welche sich durch mehrere verschiedene Sensoren beobachten lassen, bilden. Es lassen sich also höherwertige Ereignisse generieren, indem elementare Ereignisse in Beziehungen zueinander gesetzt werden. Im Allgemeinen wird es dadurch realisiert, dass in einem Datenstrom aus Ereignissen nach definierten Mustern geprüft wird.

Wird ein Muster entdeckt, werden die betreffenden primitiven Ereignisse zu einem neuen, höherwertigen Ereignis zusammengefasst und dem Datenstrom hinzugefügt [ADGI08]. Auf das Beispiel angewendet bedeutet das, dass ein Ladendiebstahl genau dann vorliegen könnte, wenn die Sensoren nacheinander melden, dass ein Produkt aus dem Regal genommen wurde und anschließend durch die Ausgangstür transportiert wird, ohne dass es bei von Kasse erkannt wurde. Die entsprechende Anfrage könnte beispielsweise in der SASE Event Language wie in Listing 1 formuliert sein [WDR06].

```
EVENT SEQ( SHELF_READING x,
           !(COUNTER_READING y),
           EXIT_READING z)
WHERE [id]
WITHIN 12 hours
```

Listing 1: Anfrage für das Erkennen von Diebstählen

`SHELF_READING`, `COUNTER_READING` und `EXIT_READING` stellen die elementaren Ereignisse dar, die in einer Sequenz auftreten sollen, um einen Ladendiebstahl zu erkennen. Dabei deutet das Ausrufezeichen das Nichtauftreten eines Ereignisses an. Der Ausdruck `[id]` nach dem Schlüsselwort `WHERE` schränkt das Muster auf Ereignissequenzen ein, deren Elementarereignisse x, y und z die gleiche ID haben. Dabei müssen die Elementarereignisse alle innerhalb von zwölf Stunden auftreten. Das wird durch die letzte Zeile im Listing ausgedrückt [WDR06].

Ein Ereignis, welches sich aus mehreren semantisch primitiven Ereignissen zusammensetzt, wird als *komplexes Event* bezeichnet. Der Prozess seiner Erstellung wird als *Complex Event Processing* (kurz: CEP) bezeichnet [Luc02]. Um CEP innerhalb eines DSMS realisieren zu können, müssen einige zentrale Anforderungen erfüllt sein:

- Das DSMS muss in der Lage sein, komplexe Ereignisse durch Muster spezifizieren zu können [ADGI08]. Es sollte eine benutzerfreundliche, aber dennoch ausdrucksstarke Abfragesprache vorhanden sein.
- CEP kollidiert in gewisser Weise mit dem Fenster-Prinzip (vgl. Abschnitt 3.3): durch die Fenster kann es passieren, dass zusammengehörige, zeitlich weit auseinander liegende, primitive Ereignisse nicht zum gleichen Zeitpunkt gültig sind. Es verhindert, dass komplexe Ereignisse durch die Mustersuche richtig erkannt werden. Auch das Vergrößern des Fensters (egal ob zeit- oder elementbasiert) garantiert nicht, dass alle komplexen Ereignisse erkannt werden. Es ist daher wichtig, die Fenster für CEP prädikatenbasiert zu definieren [Riz05].
- Das DSMS eine Komponente besitzen, welche für die effiziente Verarbeitung der Ereignisse verantwortlich ist. Diese sogenannte Event Processing Agent (EPA) muss in einem Eingabestrom aus Ereignissen die zuvor definierten Muster erkennen und daraus komplexe Eregnisse bilden [Luc02].

Es ist offensichtlich, dass ein DSMS, welches Datenströme verarbeitet, nicht automatisch CEP umsetzt. Dazu sind einige Anpassungen und Erweiterungen notwendig. Beispielswei-

se müssen spezielle Operatoren definiert werden, welche die CEP-Komponente realisieren, dabei gleichzeitig die oben genannten Anforderungen erfüllen.

Es existieren bereits einige Realisierungen von CEP-Systemen. Beispiele hierfür sind *Sase* [WDR06] und *Cayuga* [BDG+07]. Beide Systeme realisieren die Verarbeitung von Ereignisfolgen über nicht-deterministische endliche Automaten (NFA, nondeterministic finite automaton). Dabei handelt es sich um einen Automaten mit einer endlichen Menge von Zuständen. Der Automat beginnt in einem Startzustand und wechselt je nach Eingabe und aktuellen Zustand in einen neuen Zustand. Mit dieser Verarbeitungsart lassen sich viele Gegebenheiten der realen Welt abbilden. Genaueres zu NFA ist in [EP08] zu finden. Beide o. gen. CEP-Systeme verwenden es, um aus einer gegebenen Ereignisfolge Muster zu erkennen, um daraus komplexere Ereignisse zu erzeugen. Um Anfragen zu stellen, bieten beide Systeme eine eigene Abfragesprache an: während in SASE *SASE+* verwendet wird, kommt in Cayuga die *Cayuga Event Language* (CEL) zum Einsatz. Für weitere Informationen sei auf [WDR06] sowie [BDG+07] verwiesen.

5 Zusammenfassung

Datenströme stellen im Vergleich zu persistenten Datenmengen eine neue äußere Form von Daten dar. Sie sind potenziell unendliche Sequenzen von zeitlich geordneten Datenelementen. Sie werden durch selbständige, aktive Datenquellen generiert und versendet. Konventionelle Datenbankmanagementsysteme sind nicht in der Lage, diese Art von Daten zu verarbeiten. Speziell zugeschnittene Datenstrommanagementsysteme unterscheiden sich grundlegend von Datenbankmanagementsystemen und bieten Funktionen und Eigenschaften an, um die Datenströme in Echtzeit zu verarbeiten und Ergebnisse ausgeben zu können. Dabei werden als zentrales Konzept Operatoren eingesetzt. Kontinuierliche Anfragen werden über die Verknüpfung der Operatoren zu einem Operatorgraphen realisiert und gewährleisten eine zeitnahe Verarbeitung. Um mit der Unendlichkeit der Datenströme umgehen zu können, werden Fenster eingesetzt, die in einer unendlichen Menge an Daten eine endliche Menge als gültig markieren. Zuvor blockierende Operatoren können damit aufgelöst werden. Beispiele für DSMS sind Odysseus und Aurora. Beide sind Frameworks, die durch die Bereitstellung zentraler DSMS-Konzepte die Konstruktion und Wartung anwendungsspezifischer DSMS vereinfachen.

Ein Datenstrom kann auch als eine Folge von Ereignissen betrachtet werden. Dabei können komplexere Ereignisse durch Aggregation anderer Ereignisse gebildet werden. Das Erstellung von komplexen Ereignissen wird Complex Event Processing (CEP) genannt. Wenn DSMS dies realisieren wollen, müssen sie Mustererkennung, prädikatenbasierte Fenster und entsprechende Abfragesprachen unterstützen.

Literatur

[ADGI08] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom und Neil Immerman. Efficient Pattern Matching over Event Streams. *Proceedings of SIGMOD 2008*, 2008.

[BBD+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani und Jennifer Widom. Models and Issues in Data Stream Systems. *Invited paper in Proc. of PODS 2002*, 2002.

[BDG+07] Lars Brenna, Alan J. Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte und Walker M. White. Cayuga: A High-Performance Event Processing Engine. *SIGMOD Conference 2007*, 2007.

[BGJ+09] Andre Bolles, Marco Grawunder, Jonas Jakobi, und H.-Jürgen Appelrath. Odysseus: Ein Framework für maßgeschneiderte Datenstrommanagementsysteme. *Universität Oldenburg, Department für Informatik, Informationssysteme*, 2009.

[CHKS04] Michael Cammert, Christoph Heinz, Jürgen Krämer und Bernhard Seeger. Anfrageverarbeitung auf Datenströmen. *Datenbank-Spektrum 11/2004*, 2004.

[EP08] Katrin Erk und Lutz Priese. Theoretische Informatik: Eine umfassende Einführung, 3.Auflage. *Springer-Verlag*, 2008.

[Gra10] Marco Grawunder. Informationssysteme II - Datenstromverarbeitung - Präsentationsfolien. *Universität Oldenburg, Department für Informatik*, 2010.

[Gs03] Lukasz Golab und M. Tamer Öszu. Issues in Data Stream Management. *SIGMOD Record, Vol. 32, No. 2*, 2003.

[Krä07] Jürgen Krämer. Continuous Queries over Data Streams - Semantics and Implementation. *Phillips-Universität Marburg*, 2007.

[Luc02] David Luckham. The Power Of Events. *Addison-Wesley*, 2002.

[Riz05] Shariq Rizvi. Complex Event Processing Beyond Active Databases: Streams and Uncertainities. *University of California at Berkeley*, 2005.

[See09] Bernhard Seeger. Datenströme. 2009.

[WDR06] Eugene Wu, Yanlei Diao und Shariq Rizvi. High-Performance Complex Event Processing over Streams. *Proceedings of SIGMOD 2006*, 2006.