Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Supporting Consumptions in Parallel Complex Event Processing Operators on Multicore Architectures

Manuel Gräber

Studiengang: Informatik

Prüfer/in: Prof. Dr. Kurt Rothermel

Betreuer/in: Dipl.-Inf. Ruben Mayer

Beginn am: 17. Oktober 2016

Beendet am: 18. April 2017

CR-Nummer: C.2.4

Abstract

Complex Event Processing has grown in importance over the last few years trough the Internet of Things and the increasing sensor data generated. Through interpreting and combining those Events, Complex Events are generated. In order to increase the processing speed, this task is usually performed multithreaded. Events are distributed to different Threads and duplicated. There are already systems capable of processing Events multithreaded. Regularly event consumption is required, so an event can only be used to generate a single Complex Event. Since sometimes threads share an event, this results in dependencies. Current systems are not able to handle this dependencies while processing in parallel. So they have to process sequentially. This thesis introduces and evaluates the SPECRTE Framework, which is capable of parallel processing by using a speculative Model based on Markov Chains.

Kurzfassung

Complex Event Processing hat in letzter Zeit durch das Internet of Things und die steigende Anzahl an Sensordaten immer mehr an Bedeutung gewonnen. Diese zu verwendeten Events müssen in Zusammenhang gestellt werden. Dadurch entstehen aus mehreren Events ein Complex Event. Um das Erstellen der Complex Events zu beschleunigen wird normalerweise auf parallele Verarbeitung zurückgegriffen. So werden die einzelnen Events an mehrere Worker verteilt und dupliziert. Es existieren bereits Systeme, die das parallele Verarbeiten solcher Events unterstützen. Häufig wird allerdings das Konsumieren der Events gefordert. Das heißt, es steht den anderen Workern nicht mehr zur Verfügung. Dadurch entstehen Abhängigkeiten zwischen den Workern. Aktuelle Systeme sind aufgrund dieser Abhängigkeiten nicht in der Lage, parallel zu arbeiten. In dieser Arbeit wird das SPECTRE Framework vorgestellt und evaluiert, das durch ein spekulatives Model auf Basis von Markov Ketten, die parallele Verarbeitung dieser Events ermöglicht.

Inhaltsverzeichnis

1 Einleitung **15**

2 Hintergrund **19**
- 2.1 Definitionen 19
- 2.2 Allgemeine Architektur 20
- 2.3 Complex Event Spezifikation 21
- 2.4 Zusammenfassung 24

3 Basis System **27**
- 3.1 Definitionen 27
- 3.2 Architektur 29
- 3.3 Parallele Verarbeitung 31
- 3.4 Zusammenfassung 36

4 Problembeschreibung **37**
- 4.1 Veranschaulichung von Abhängigkeiten 37
- 4.2 Ziel der Arbeit 39
- 4.3 Herausforderungen 39

5 Lösungsansatz **41**
- 5.1 Architektur 41
- 5.2 Wahrscheinlichkeit 42
- 5.3 Scheduling 49
- 5.4 Weitere Optimierung 50
- 5.5 Zusammenfassung 51

6 Evaluation **53**
- 6.1 Rahmenbedingungen 53
- 6.2 Evaluation 55
- 6.3 Zusammenfassung 66

7 Verwandte Arbeiten **69**
- 7.1 T-Rex 69
- 7.2 Speculative Out-Of-Order Event Processing 70
- 7.3 Adaptive Speculative Processing of Out-of-Order Event Streams 71
- 7.4 GraphCEP 72

7.5 Zusammenfassung 72

8 Zusammenfassung und Ausblick **75**

Literaturverzeichnis **77**

Abbildungsverzeichnis

2.1 Architektur eines CEP Systems 21

3.1 Die Abbildung zeigt den Zusammenhang der Komponenten von SPECTRE. . . 29
3.2 Darstellung des Abhängigkeitsbaumes. 33

4.1 Darstellung des Eventstreams $A_1A_2B_1C_1C_2A_3B_2C_3$. 38
4.2 Abhängigkeitsbaum zum Eventstream $A_1A_2B_1C_1C_2A_3B_2C_3$. 39

5.1 Zusammenhang der Komponenten 41
5.2 Wahrscheinlichkeiten, dass ein Partial Match noch Vervollständigt wird. . . . 42
5.3 Ein Zustandsgraph mit Übergangswahrscheinlichkeiten, der als Beispiel für die Generierung einer Markov Matix genutzt wird. 44
5.4 Beispielhafter Eventstream zur Markov Matrix Erstellung. 45

6.1 Events pro Sekunde für 1-16 Workerthreads. 55
6.2 Dauer der Erkennung des Complex Events seit das Erste Event das System betreten hat. 56
6.3 Frequenz der Schedulings des Path Manager 57
6.4 Events/Sekunde für Patterngrößen von 10-160. 58
6.5 Events/Sekunde für Fenstergrößen von 60-480s. 59
6.6 Events/Sekunde feste Complex Event Wahrscheinlichkeiten, Patterngröße 80 . 60
6.7 Events/Sekunde feste Complex Event Wahrscheinlichkeiten, Patterngröße 300 60
6.8 Events/Sekunde feste Complex Event Wahrscheinlichkeiten, Patterngröße 500 61
6.9 Events/Sekunde für Matrinminimierungsfaktor 1-8 62
6.10 Events/Sekunde für unterschiedliche Checkponteinstellungen. 63
6.11 Events/Sekunde verschiedene Anzahl an Workerthreads und Abhängigkeitsbaum Tiefen. 64
6.12 Events/Sekunde für 1-16 Workerthreads ohne Konsumierung. 65
6.13 Events/Sekunde für T-Rex bei Fenstergrößen 60-480s. 66

7.1 Die Architektur von T-Rex 70
7.2 Darstellung für Speculative Out-Of-Order Event Processing 70
7.3 Darstellung des Adaptive Speculative Processing of Out-of-Order Event Streams Systems. 71
7.4 Architektur des GraphCEP Systems 72

Verzeichnis der Listings

2.1 Beispiel für eine Eventspezifikation mit Snoop. 23
2.2 Beispiel für eine Eventspezifikation mit TESLA. 24

3.1 Pseudocode für die ps-Funktion 30
3.2 Pseudocode für die pc-Funktion 30

Verzeichnis der Algorithmen

3.1 Pseudocode: vereinfachter mainloop des Pathmanagers. 32
3.2 Pseudocode für das Hinzufügen eines neuen Fensters. In einem Consumption Group Knoten werden ebenfalls die vorangegangenen Consumption Groups gespeichert. 34
3.3 Pseudocode für das Hinzufügen einer neuen Consumption Group. 34
3.4 Pseudocode für die Vervollständigung einer Consumption Group. 34
3.5 Pseudocode für das Verwerfen einer Consumption Group. 35
3.6 Pseudocode für das Scheduling. 35

5.1 Pseudocode: berechneWahrscheinlichkeit 47
5.2 Pseudocode: empfangeCGroup 48
5.3 Pseudocode: Top K Algorithmus. 50

1 Einleitung

In den letzten Jahren hat sich die Menge an Daten, die durch verschiedenste Art und Weise generiert werden, vervielfältigt. Diese sollen möglichst schnell und zuverlässig ausgewertet werden, sowohl um zeitnah darauf reagieren zu können als auch um den kontinuierlichen Strom an Daten überhaupt verarbeiten zu können, ohne dass Vieles zwischengespeichert oder verworfen werden muss. Beispiele hierfür sind:

- In der Verkehrsüberwachung senden Fahrzeuge regelmäßig ihren Standort. Somit können Staus erkannt und anderen Verkehrsteilnehmern eine alternative Route vorgeschlagen werden[KUM15].
- Im automatisiertem Handel müssen Computersysteme so zeitnah wie möglich auf Kursänderungen reagieren können [ME09].
- Beim Energiemanagement Änderungen frühzeitig zu erkennen und auf diese reagieren zu können [Oef].
- Die Überwachung von Geschäftsprozessen und Unternemenskritischen Ressourcen im Rahmen des Business Activity Monitoring[ME09].

Für all diese Fälle wird eine sehr große Anzahl an Ereignissen, im weiteren verlauf der Arbeit als Event bezeichnet, generiert. Viele dieser Events stehen oft im keinem Zusammenhang zueinander, könnten es aber unter gewissen Umständen. CEP Systeme werden dazu eingesetzt, um zeitnah Zusammenhänge zu erkennen und an darauf spezialisierte Anwendung weiterzugeben.

Ein zu langsames Erkennen kann dazu führen, dass die gewonnenen Erkenntnisse mittlerweile falsch oder nicht mehr von Bedeutung sind. Ein verspätetes Erkennen eines Staus führt zu einer deutlich höheren Reisezeit für den Verkehrsteilnehmen, während im automatischen Handel eine zu spät gewonnene Information schon falsch ist und viel Geld verloren geht.

Durch den Anstieg der Menge an Daten steigt auch die Notwendigkeit an Software, die diese Verarbeiten kann. Es existiert bereits eine Palette an verschiedenen CEP Systemen, die von verschiedenen Unternehmen entwickelt und genutzt werden [See10][May+16][CM12a].

Um die Events schnell verarbeiten zu können, bietet sich die parallele Verarbeitung dieser an. Hierzu werden die Events vorsortiert und an mehrere Threads verteilt. Solange keine Abhängigkeit zwischen den Threads besteht, kann dadurch der Eventdurchsatz auf ein vielfaches

erhöht werden. Durch CUDA ist mittlerweise auch möglich, nicht nur die CPU sondern auch die GPU für diese Verarbeitung einzusetzen[CM12b].

Eine weitere Anforderung an CEP Systeme ist die Skalierbarkeit. Häufig werden ein Großteil der Events, die ein CEP System zu bearbeiten hat, in einem kurzen Zeitabschnitt generiert. Offizielle Angaben des California Department of Transportation zeigen, dass während einer Stunde - der „Rush Hour“ - bis zu 25% des Verkehrs eines Tages aufkommt. CEP Systeme müssen auf diese Schwankungen schnell reagieren können [MKR15].

Häufig ist es aber der Fall, dass bestimmte Events nur für die Generierung eines Complex Events verwendet werden dürfen, also das Event bei der Generierung des Complex Events konsumiert wird. In diesem Fall, ist parallele Verarbeitung nicht mehr ohne weiteres möglich, da es zu Abhängigkeiten kommt. Somit müssen diese Events sequenziell verarbeitet werden, was auf Kosten des Eventdurchsatz geht.

Um das Problem anzugehen wird in dieser Arbeit das SPECTRE Framework vorgestellt. SPECTRE ist in der Lage, Abhängigkeiten zu erkennen und durch Spekulation vorherzusehen, ob Events von einem anderen Thread konsumiert werden. Das ermöglicht parallele Verarbeitung und erhöht so den Eventdurchsatz des Systems.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Hintergrund: Hier werden die Grundlagen zu Complex Event Processing beschrieben und die Grundbegriffe definiert. Des Weiteren wird auf die Event Specification Languages Snoop und TESLA eingegangen.

Kapitel 3 – Basis System: In diesem Kapitel wird ein Einblick in das System gegeben, das die Basis des SPECTRE Frameworks darstellt.

Kapitel 4 – Problembeschreibung: Dieses Kapitel stellt das Problem des bisherigen Systems dar und warum dieses nicht einfach zu lösen ist.

Kapitel 5 – Lösungsansatz: In diesem werden die Lösungsansätze beschrieben, um die Probleme des vorangegangenen Kapitels zu beheben. Als Grundlage wird das in Kapitel 3 beschriebene System verwendet.

Kapitel 6 – Evaluation: Das resultierende System wird anschließend in diesem Kapitel evaluiert, indem verschiedene Parameter variiert werden. Als Vergleichssystem kommt T-Rex zum Einsatz.

Kapitel 7 – Verwandte Arbeiten: Hier wird auf verwandte Arbeiten eingegangen und diese verglichen.

Kapitel 8 – Zusammenfassung und Ausblick Die Arbeit wird mit einer kurzen Zusammenfassung und einem Ausblick auf zukünftige Arbeiten abgeschlossen.

2 Hintergrund

Complex Event Processing (CEP) bezeichnet Methoden, die eine Menge von Ereignissen (Events) analysieren, um Zusammenhänge zu erkennen und diese für weitere Bearbeitung bereit zu machen. Diese Events enthalten meist verschiedenste Messdaten, wie zum Beispiel die eines Luftdruckmessgerätes. Diese Messungen werden kontinuierlich von vielen Messgeräten durchgeführt. Ein CEP System kann aus diesen einfachen Events Zusammenhänge erkennen und diese an spezialisierte Anwendungen weitergeben. Daraus könnten unter anderem Wettervorhersagen generiert werden[CM12b]. In der Arbeit [ZU99] wurde auf viele der Begriffe, die im Folgenden relevant sind, eingegangen.

2.1 Definitionen

Im Folgenden werden allgemeine Begriffe definiert, die im Weiteren Verwendung finden.

2.1.1 Event

Ein Event ist eine Informationsmenge. Diese Informationen werden in der Regel durch einen Sensor generiert. Im Beispiel oben wären das der Standort, die Uhrzeit, Luftdruck und der Event-Typ, in diesem Fall Luftdruckmessung. Diese Daten werden in der Regel als Schlüssel-Wert-Paare angegeben.

2.1.2 Event Stream

Der Eventstream ist die Menge von Events, die das CEP System betreten haben. Um diese in einer absoluten Reihenfolge zu halten, werden den einzelnen Events beim Eintritt in das System Seriennummern zugeteilt. Der Stream wird von Operator Instanzen dazu verwendet, um Complex Events zu generieren. Es kann durchaus vorkommen, dass einzelne Events nicht benötigt werden.

2.1.3 Complex Event

Ein Complex Event entsteht, wenn der Operator einen vordefinierten Zusammenhang zwischen den Events erkennt. Der Operator könnte zum Beispiel aus verschiedenen Luftdruckmessungen und Temperaturmessungen Daten für eine Wettervorhersage generieren [ZU99].

2.1.4 Operator

Der Operator ist eine programmierbare Instanz, die, anhand von Patterns, aus den Informationen der Events, Complex Events generieren kann.

2.1.5 Pattern

Ein Pattern beschreibt, wie ein Complex Event erstellt wird. Also welche Events dafür benötigt werden, die Reihenfolge in der sie auftreten, die Anzahl, ihre Werte und/oder andere Kriterien. Ein Pattern wird in der Regel durch eine Ereignisspezifikationssprache (Event Specification Language) beschrieben.

2.2 Allgemeine Architektur

In Abbildung 2.1 ist die allgemeine Architektur eines CEP Systems dargestellt. Eine Menge an Quellen detektiert einfache Events. Diese werden an die CEP Einheit geschickt. Mithilfe der Event Definitionen werden hier Complex Events generiert und an die Event Consumer weitergegeben.

Der Aufbau der CEP Einheit ist je nach CEP System unterschiedlich. Das im Kapitel 3 beschriebene System ist nur auf einem Computer lauffähig, während andere Systeme auf mehrere Computer verteilt werden können. Dadurch werden aber auch die Latenzen der Kommunikation der einzelnen Komponenten größer. Es bietet aber auch den Vorteil, dass die so verfügbare Rechenleistung deutlich größer ist und das System so besser skaleren kann [MTR16].

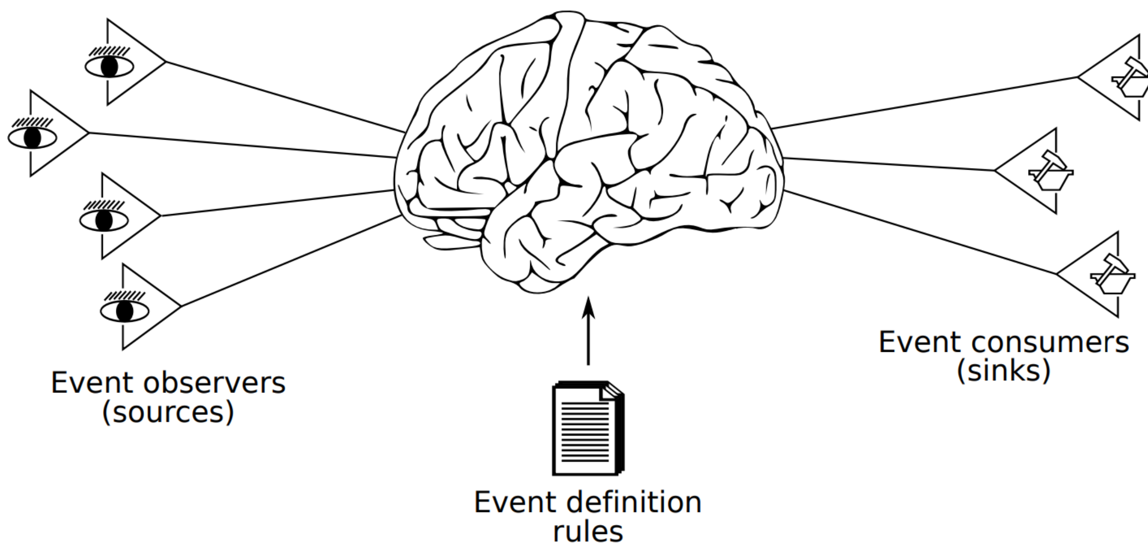

Abbildung 2.1: Architektur eines CEP Systems[CM12b].

2.3 Complex Event Spezifikation

Für die Operator Instanz muss formal spezifiziert werden, welche Events mit welchen Eigenschaften, auf welche Weise auftreten müssen. Um das zu spezifizieren werden Event Specification Languages wie Snoop und TESLA verwendet. Hierfür können einfache Disjunktionen, Konjunktionen, Sequenzen und Negationen verwendet und kombiniert werden, aber auch zeitliche Abhängigkeiten sind möglich.

2.3.1 Selections Policy

Eine Event Spezifikation sollte eindeutig sein. Wenn beispielsweise eine Sequenz ABC erkannt werden soll und der Eventstream $A_1A_2B_1C_1$ enthält, gibt es verschiedene Möglichkeiten daraus Complex Events zu generieren: $A_1B_1C_1$ oder $A_2B_1C_1$. Welche Complex Events generiert, werden ist abhängig von der Selection Policy [CM12c].

- Earliest Selection: Es wird immer das erste Vorkommen eines Events verwendet, in diesem Fall also $A_1B_1C_1$.
- Latest Selection: Es wird immer das letzte Vorkommen eines Events verwendet, in diesem Fall also $A_2B_1C_1$.

- Each Selection: Es werden alle möglichen Kombinationen der Events verwendet um mehrere Complex Events zu erzeugen. Es werden also aus $A_1B_1C_1$ und $A_2B_1C_1$ Complex Events generiert.

2.3.2 Comsumptions Policy

Ein wichtiger Faktor ist auch, ob ein Event für die Erstellung mehrerer Complex Events verwendet werden darf oder nicht. Hier wird zwischen Selected Consumption und Zero Consumption unterschieden.

Bei Selected Consumption werden alle Events, die zum Erstellen des Complex Event benötigt werden, konsumiert und stehen somit nicht für weitere Complex Events zur Verfügung [CM12c].

Das Gegenstück stellt Zero Consumption dar. Hier werden keine der Events konsumiert.

In speziellen Fällen kann bei verschiedenen Event Specification Languages auch definiert werden, dass nur ein Teil der Events konsumiert wird.

2.3.3 Event Specification Languages

Im Folgenden wird auf zwei populäre Event Specification Languages eingegangen, Snoop und TESLA.

Snoop

Ursprünglich wurde Snoop für Datenbankmanagement Systeme entwickelt. Da Snoop allerdings vom Sprachdesign keine Datenbank voraussetzt, kann es auch für die Spezifikation von Complex Events verwendet werden.

Der Aufbau einer Eventspezifikation besteht aus vier Teilen. On, $Condition$, $Action$ und $ParameterContext$.

Bei On werden die verwendeten Eventtypen spezifiziert, welche verwendet werden, in welcher Reihenfolge sie auftreten und Weiteres. Unterstützt werden:

- Sequenz ;: Das Auftreten von zwei Events in dieser Reihenfolge.
- Disjunktion $\vee$: Das Auftreten mindestens eines der Events.
- Konjunktion Any, All: $Any(I, Events)$ spezifiziert das Auftreten von I Events aus der Menge von $Events$ in beliebiger Reihenfolge. $Any(2, E_1, E_2)$ würde also die Sequenzen E_1E_2 und E_2E_1 einschließen. $All(Events)$ ist gleichbedeutend mit $Any(\#Events, Events)$.

- Aperiodisch $A(E_1, E_2, E_3)$: Das Auftreten des Events E_2 im Intervall E_1 bis E_3. Wird für jedes Auftreten von E_2 getriggert.
- Periodisch $P(E_1, t, E_3)$: Triggert ab dem ersten Auftreten von E_1 alle t Zeiteinheiten bis E_3 eintritt.

$Condition$ gibt die Bedingungen der Events an, z.B. $E_1.Wert > E_2.Wert$. Sollte keine solche Bedingung nötig sein, muss $true$ angegeben werden.

$Action$ gibt die Aktion an, die jedes mal durchgeführt werden soll, wenn On und $Condition$ erfüllt werden. Für CEP wäre das in der Regel: Generiere Complex Event CE_1.

Snoop bietet vier verschiedene $ParameterContexte$ an. Diese spezifizieren die Selection Policy und die Consumption Policy.

- $Recent$: Entspricht Latest Selection, Zero Consumption
- $Chroncle$: Entspricht Earliest Selection, Selected Consumption
- $Continuous$: Ein Sliding Window innerhalb dessen immer das erste und letzte Event in Betracht gezogen wird. Es ist vergleichbar mit Each Selection, Selected Consumption. Wobei die Consumption für dieses Complex Event gilt, für andere Complex Events stehen die Events noch zur Verfügung.
- $Cumulative$: Es werden alle Vorkommen der Events, die für den On Parameter in Betracht kommen verwendet und konsumiert. Für $Condtion$ werden alle Werte der Events verwendet.

Snoop bietet viele Stellschrauben um Complex Events zu definieren [CM94]. Listing 2.1 zeigt ein Beispiel für die Erkennung des Patterns $E_1; E_2; E_3$ zur Generierung eines Complex Events. Die Bedingung ist, dass der E_1 einen kleineren der Wert als E_2 hat, und dieser Wert wiederum kleiner als der von E_3 ist. Der Parameter Context ist $Recent$, es werden also keine Events konsumiert.

`On`	$E_1; E_2; E_3$
`Condition`	$(E_1.Wert > E_2.Wert)\&\&(E_2.Wert > E_3.Wert)$
`Action`	`Generiere CE`
`Parameter Context`	$Recent$

Listing 2.1: Beispiel für eine Eventspezifikation mit Snoop.

TESLA

TESLA (Trio-based Event Specification LAnguage) ist eine weitere Event Specification Language. Im Gegensatz zu Snoop wurde sie speziell für CEP Systeme entwickelt. TESLA unterstützt alle drei Selection Policys und kann keine oder beliebige Events konsumieren, bietet also mehr Möglichkeiten als Selected Consumption und Zero Consumption.

Trio wurde als Basis für TESLA gewählt, damit möglich ist, zeitliche Ausdrücke formal darzustellen.

Vom Aufbau ist TESLA Snoop ähnlich. Es beginnt mit $define$ mit dem das Complex Event benannt wird. Es folgt $from$ das angibt, welche Events verwendet werden und $where$ das die Events auf Bedingungen prüft. Am Schluss kommt das optionale $consuming$ mit dem spezifiziert wird, welche Events konsumiert werden [CM10].

Listing 2.2 zeigt eine beispielhafte Eventspezifikation. Es soll eine Veränderung des Aktienmarktes erkannt werden. Das generierte Complex Event enthällt die das die Höhe des Anstieges. Hierfür werden die Eventtypen StockMarketOpenEvent und StockMarketCloseEvent berücksichtigt. Die verwendeten Events werden konsumiert.

```
define          MarketRiseEvent(Val)
from            StockMarketOpenEvent(StartVal) and StockMarketCloseEvent(EndVal)
where           Val = StockMarketCloseEvent.Endval - StockMarketOpenEvent.StartVal
consuming       StockMarketOpenEvent, StockMarketCloseEvent
```

Listing 2.2: Beispiel für eine Eventspezifikation mit TESLA.

Eine Implementierung von TESLA wäre beispielsweise das CEP System T-Rex [CM12a]. Im weiteren Verlauf der Arbeit wird etwas weiter auf T-Rex eingegangen, da es als Vergleichssystem für die Evaluation dient und Event Consumption unterstützt.

Andere Event Specification Languages

Es existieren noch weitere Sprachen die zur Event Spezifikation verwendet werden können. Es ist nicht möglich alle mit ihren Unterschieden hier aufzulisten. Dennoch finden die Event Specification Languages Amit und SASE+ aufgrund ihrer Besonderheiten eine kurze Erwähnung.

Amit spezifiziert eine Lebensdauer für die Eventspezifikationen, in der diese gelten. Selection und Consumption Policy können beliebig spezifiziert werden [AE04].

SASE+ ist eine weitere Event Specification Language. Sie bietet zum Beispiel für Sliding Windows eine einfache Spezifizierung, allerdings ist es in der Selection und Consumption Policy nicht so flexibel wie Amit oder TESLA [Gyl+08].

2.4 Zusammenfassung

Das Kapitel hat einige grundlegende Begriffe des Complex Event Processing definiert. Zudem hat es hat die allgemeine Architektur von CEP Systemen erläutert. Des weiteren ging es näher auf die Complex Event Spezifikation ein, indem es Consumption Policy und Selection Policy erläutert hat.

Zuletzt wurden noch die zwei Event Specifikation Languages Snoop und TESLA anhand ihrer Syntax und Beispielen beschrieben.

3 Basis System

In diesem Kapitel wird die Funktionsweise von SPECTRE (SPECulaTive Runtime Environment) vorgestellt. SPECTRE ist ein System, das die Abhängigkeiten der verschiedenen Fenster erkennt und diese Fenster an die verfügbaren Operatorinstanzen vergibt. Das System ist in C++ implementiert und auf einem Linux Betriebssystem lauffähig.

3.1 Definitionen

Hier werden die Begriffe, die im weiteren Verlauf der Arbeit Anwendung finden, definiert.

3.1.1 Definition: Partial Match

Bevor ein Complex Event generiert werden kann, müssen die einzelnen Events vom Operator erkannt werden. Wurbe bisher nur ein Teil der benötigen Events erkannt, werden diese als Partial Match bezeichnet. Sollten die fehlenden Events noch erkannt werden, wird daraus ein Complex Event.

3.1.2 Definition: Consumption Group

Eine Consumption Group ist eine Menge von Events die alle zusammen konsumiert werden, falls das Complex Event generiert wird.

Falls als Consumption Policy Selected Consumption gewählt wird, ist das Partial Match gleich der Consumption Group.

3.1.3 Definition: Fenster

Als Fenster bezeichnet man eine Menge von Events. Diese Menge beinhaltet Events die zwischen dem Anfangsevent und dem Endevent des Fensters liegen. Je nach Konfiguration des Splitters sind das alle oder nur ein Teil der Events.

3.1.4 Definition: Fensterversion

Jedes Fenster hat mindestens eine Fensterversion. Eine Fensterversion beinhaltet eine Teilmenge der Events des Fensters. Die Fensterversionen des Fensters F_i ergeben sich aus den Consumption Groups der Fenster von denen F_i direkt oder indirekt abhängt. Die Fensterversionen werden dadurch realisiert, dass sie die Consumption Groups der vorangegangenen Fenster enthalten, die nicht Teil des Fensters sein sollen. Diese werden regelmäßig aktualisiert. Jede der Fensterversionen, kann von einem Worker ausgeführt werden.

3.1.5 Abhängigkeiten

Sei O_i die Anzahl der offenen Fenster nach dem Event E_i, ID(E) die Nummer des Events, Öffnen(F) das Erste Event des Fensters und Schließen(F) das letzte Event des Fensters. Im Folgenden werden verschiedene Formen der Abhängigkeit definiert.

Definition: Unabhängigkeit

Ein Fenster ist unabhängig genau dann, wenn es nicht mit einem zeitlich älteren Fenster überlappt. Also das erste Event von F_i nach dem letzten Event von F_{i-1} kommt.

Definition: Direkte Abhängigkeit

Ein Fenster F_j ist direkt zu einem vorangegangenen Fenster F_i abhängig genau dann, wenn es mit dem vorangegangenen Fenster überlappt - also das erste Event von F_j vor dem letzten Event von F_i kommt - und zwischen dem ersten Event F_i und dem ersten Event von F_j kein anderes Fenster begonnen hat und noch nicht wieder geschlossen wurde.

Definition: Indirekte Abhängigkeit

Ein Fenster F_k ist indirekt zu einem vorangegangenen Fenster F_i abhängig genau dann, wenn vor dem öffnenden Event $O_{ID(F_k)-1} > 1$ oder das erste Event von F_k nach dem letzten Event von F_i kommt - formal $ID(Öffnen(F_k)) > Schließen(ID(F_i))$ - und $\prod_{l=ID(Schließen(F_i))}^{ID(Öffnen(F_k)-1} O_i > 0$, also immer ein offenes Fenster existiert. Einfacher ausgedrückt: F_k ist indirekt von F_i abhängig, wenn eine Kette aus direkten Abhängigkeiten mit mehr als zwei Elementen besteht.

3.2 Architektur

Zunächst wird die Architektur von SPECTRE beschrieben. Abbildung 3.1 zeigt den Zusammenhang der einzelnen Komponenten und den Weg, den die einzelnen Events durch das System nehmen. Abbildung 3.1 veranschaulicht den Zusammenhang der einzelnen Komponenten von SPECTRE. Diese werden im Weiteren näher bschrieben.

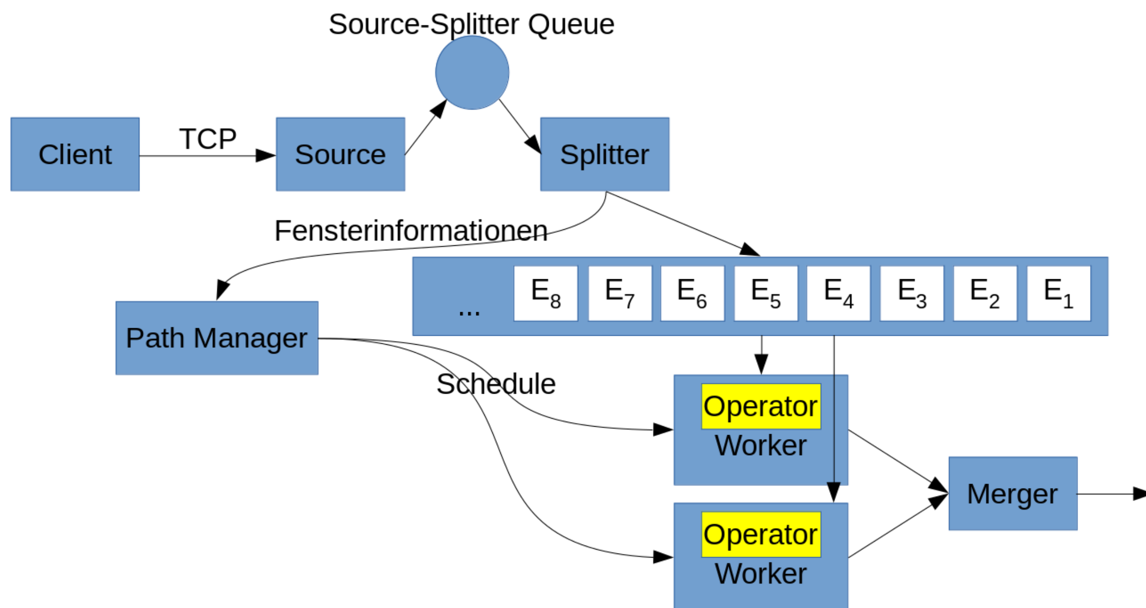

Abbildung 3.1: Die Abbildung zeigt den Zusammenhang der Komponenten von SPECTRE.

3.2.1 Client

Der Client ist eine eigenständige Anwendung. Diese kommuniziert über TCP mit der Source des Hauptprogramms. Die Events werden als ASCII-String verschickt.

3.2.2 Source

Die Source empfängt die ASCII-Strings, wandelt diese in systeminterne Events um, allokiert den Speicher und gibt einen Pointer an die Source-Splitter Warteschlange weiter. Jedem Event wird eine steigende ID zugewiesen. Sobald die Source ein Terminierungszeichen empfängt, wird ein spezielles Event erzeugt und die Source wird beendet. Sobald der Splitter dieses Event empfängt, werden alle offenen Fenster geschlossen.

3.2.3 Splitter

Um Parallelisierung zu ermöglichen, muss der Eventstream aufgeteilt werden. Der Splitter teilt den Eventstream in verschiedene sogenannte Fenster auf, damit an jedem Fenster ein Workerthread arbeiten kann. Um das zu ermöglichen, muss der Splitter eine Vorprüfung der einzelnen Events durchführen, damit nicht potenziell zusammengehörende Events in verschiedenen Fenstern enden. Es ist durchaus möglich, dass ein Event in mehreren Fenstern vorkommt.

Der Splitter überprüft anhand einer Funktion ps für jedes ankommende Event, ob ein neues Fenster geöffnet werden muss. Anhand einer Funkton pc wird überprüft, ob ein offenes Fenster geschlossen wird. Diese Prüfung muss für jedes offene Fenster durchgeführt werden. Die Listings 3.1 und 3.2 stellen Beispiele für diese Funktionen dar. Wenn ps $true$ zurückgibt, wird ein neues Fenster geöffnet. Liefert pc $true$ zurück, wird das Fenster f geschlossen.

Die Events werden in einem globalen Event Stream platziert. Dieser ist als verkettete Liste implementiert. Ein Event ist Teil eines Fensters, wenn es zwischen dem öffnenden und dem schließenden Event liegt.

Die Funktionen müssen anhand der Pattern Definition implementiert sein. Möglichkeiten hierfür wären Zeit- oder Eventanzahlbeschränkte Fenster, die anhand eines bestimmten Events geöffnet werden.

Durch das Verwenden von Fenstern wird der Eventstream geteilt und kann somit parallel von verschiedenen Operatorinstanzen bearbeitet werden, sofern diese von einander unabhängig sind.

```
ps (Event e)
  if istStartEvent(e) then
     return true
  else
     return false
  end if
end function
```

Listing 3.1: Pseudocode für die ps-Funktion

```
pc (Event e, Fenster f)
  if eventSchliesstFenster(e, f) then
     return true
  else
     return false
  end if
end function
```

Listing 3.2: Pseudocode für die pc-Funktion

3.2.4 Worker

Der Worker bearbeitet das Fenster, das er vom Path Manager zugeteilt bekommen hat. Er beginnt mit dem ersten Event eines Fensters, prüft ob es konsumiert wurde, und falls dies nicht der Fall ist, gibt er es an den Operator. Dieser wird für jedes Fenster neu initialisiert.

3.2.5 Operator

Anhand des Patterns prüft der Operator, ob die Events, die er vom Worker bekommt, Teil eines Complex Events sind oder sein könnten. Ist das der Fall, wird es einem oder mehreren Partial Matches hinzugefügt. Wird ein Partial Match vervollständigt, werden je nach Consumption Policy alle entsprechenden Events als konsumiert markiert und das Complex Event erzeugt.

3.2.6 Merger

Der Merger bekommt alle erzeugten Complex Events, sortiert diese und gibt sie dann nach außen weiter. Des Weiteren werden nicht mehr benötigte Fenster gelöscht und somit auch der Speicher für die fensterlosen Events freigegenden.

3.3 Parallele Verarbeitung

Solange keine Abhängigkeit zwischen den Fenstern besteht, kann der Path Manager einfach die verfügbaren Fenster an die entsprechenden Workerthreads vergeben. Diese bearbeiten die Events und geben alle generierten Complex Events an den Merger weiter. Das funktioniert problemlos, solange keine Abhängigkeit zwischen den Fenstern besteht.

Allerdings treten Abhängigkeiten zwischen den Fenstern auf, wenn gefordert wird, dass Events konsumiert werden. Für diesen Fall pflegt der Path Manager einen Abhängigkeitsbaum.

Jedes Partial Match kann vervollständigt und somit konsumiert werden. Alle Events die zusammen konsumiert werden sind Teil der selben Consumption Group. Erst beim Vervollständigen des Complex Events - oder am Ende des Fensters - stellt sich heraus, ob die entsprechenden Events konsumiert werden. Es gibt also für jede Consumption Group zwei Möglichkeiten. Daraus folgt, dass es für jedes davon direkt abhängige Fenster zwei unterschiedliche Fensterversionen gibt, eine in der alle Events der Consumption Group zur Verfügung stehen und eins in dem das nicht der Fall ist.

3.3.1 Path Manager

Der Path Manager verwaltet den Abhängigkeitsbaum und weist die verschiedenen Fensterversionen den Workern zu. Der Path Manager bekommt vom Splitter und den Workern neue Informationen zum Zustand des Systems. Diese werden in einer Queue zwischengespeichert und nacheinander abgearbeitet. Anschließend werden anhand von neuen Informationen die Fensterversionen den Workerthreads zugewiesen. Algorithmus 3.1 stellt vereinfacht dar, wie der Path Manager arbeitet.

Algorithmus 3.1 Pseudocode: vereinfachter mainloop des Pathmanagers.

```
PathmanagerMain() begin
   while true loop
      ManageAbhaengigkeitsbaum()
      scheduleWorker()
   end while
end function
```

3.3.2 Abhängigkeitsbaum

Um die unterschiedlichen Fensterversionen zu verwalten, werden diese als Baumstruktur realisiert. Der Wurzelknoten ist ein Fensterknoten und stellt immer die einzige Version des ältesten Fensters dar. Ein Fensterknoten hat maximal einen Kindknoten, ein Consumption Group Knoten 0 falls es kein jüngeres Fenster und keine weiteren Consumption Groups gibt oder 2. Der linke Knoten stellt den Zustand dar, dass die Consumption Group nicht konsumiert wird.

Die verschiedenen Fensterknoten, die dieselbe Anzahl an vorangegangenen Fensternknoten haben, sind verschiedene Versionen des selben Fensters. In der Abbildung 3.2 würde das bedeuten, dass der Wurzelknoten die einzige Version des Fensters F_i ist, $FV_2..FV_5$ die verschiedenen Versionen des Fensters F_{i+1} und $F_6..F_{12}$ die Versionen des Fensters F_{i+2}.

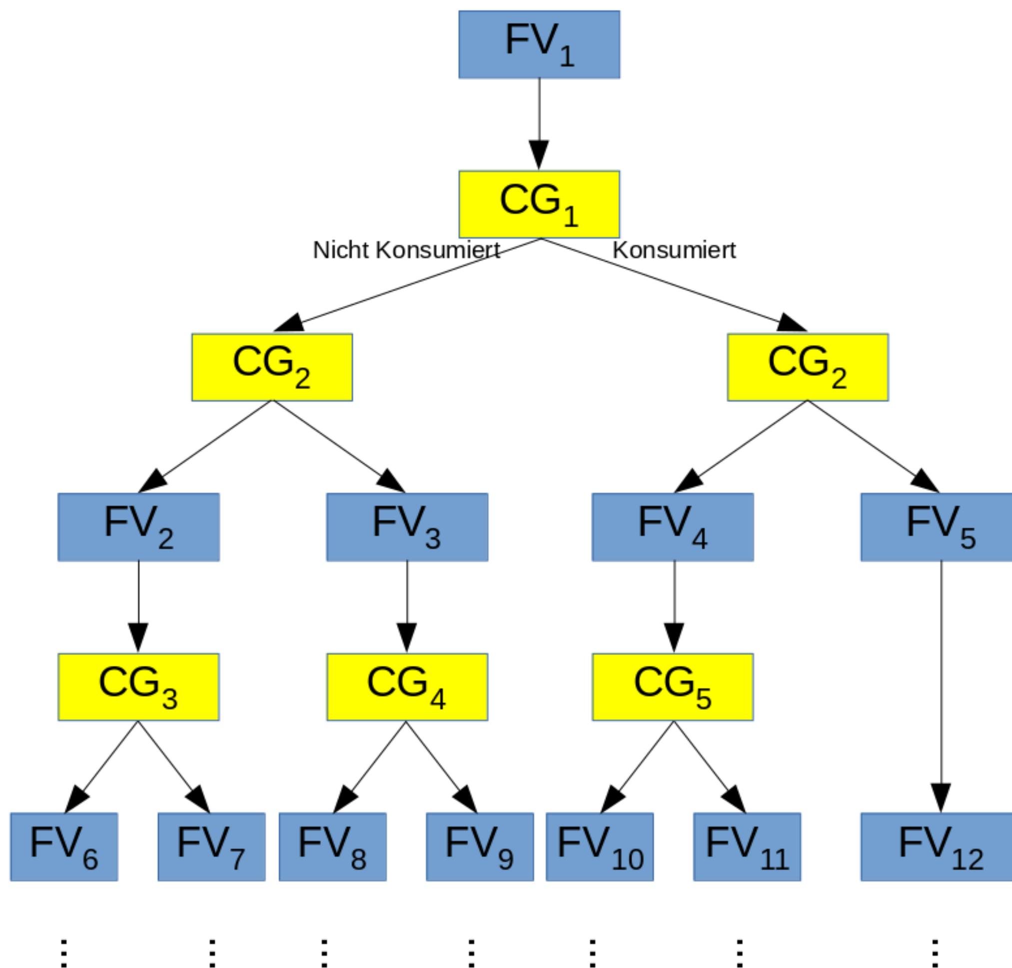

Abbildung 3.2: Darstellung des Abhängigkeitsbaumes.

3.3.3 Neues Fenster

Öffnet der Splitter ein neues Fenster, wird an jedem Blattknoten ein neuer Fensterknoten erzeugt. Dieser hat keine, bzw alle Consumption Groups des Fensterknotens, von dem er direkt abhängt. Algorithmus 3.2 stellt in Pseudocode dar, wie das im Programm umgesetzt wird.

3.3.4 Neue Consumption Group

Sobald ein Worker eine neue Consumption Group erstellt, wird ein entsprechender Consumption Group Knoten erstellt und in den Baum eingefügt. Der weiter unten liegende Teilbaum wird

Algorithmus 3.2 Pseudocode für das Hinzufügen eines neuen Fensters. In einem Consumption Group Knoten werden ebenfalls die vorangegangenen Consumption Groups gespeichert.

```
neuesFenster(Fenster F1) begin
   for each Blattknoten im Abhaengigkeitsbaum
      if Blattknoten is Fensterknoten then
         Blattknoten.kind = new Fensterknoten(F1, Blattknoten.ConsumptionGroups)
      else
         Blattknoten.linkesKind = new Fensterknoten(F1, Blattknoten.ConsumptionGroups)
         Blattknoten.rechtesKind = new Fensterknoten(F1, Blattknoten.ConsumptionGroups +
             Blattknoten)
      end if
   end for
end function
```

zum linken Kindknoten und ignoriert somit diese Consumption Group. Das rechte Kind muss neu erstellt werden. Hierzu wird der letzte Checkpoint verwendet, der vor dieser Consumption Group kam. Im Algorithmus 3.3 wird die Umsetzung im Programm dargestellt.

Algorithmus 3.3 Pseudocode für das Hinzufügen einer neuen Consumption Group.

```
neueConsumptionGroup(ConsumptionGroup CG, Knoten K1) begin
   CG.linkesKind = K1.kind
   CG.rechtesKind = new Fensterknoten(K1.kind.getFenster(), F1.kind.CGs + CG)
   K1.kind = CG
end function
```

3.3.5 Consumption Group vervollständigt oder verworfen

Durch Vervollständigung einer Consumption Group wird der nicht mehr benötigte Teil des Baumes abgeschnitten. Dieser Teil ist das linke Kind der Consumption Group. Das gleiche passiert mit dem rechten Kind sobald das Complex Event verworfen wird. In beiden Fällen wird der Consumption Group Knoten nicht mehr benötigt. Die Algorithmen 3.4 und 3.5 stellen den Pseudocode dar.

Algorithmus 3.4 Pseudocode für die Vervollständigung einer Consumption Group.

```
ConsumptionGroupVollstaendig(ConsumptionGroup CG) begin
   CG.linkesKind = null
   CG.elternKnoten.kind = CG.rechtesKind
end function
```

3.3.6 Abhängigkeitsbaum Größenlimitierung

Die Größe des Baumes wächst exponentiell. Dadurch wird das Hinzufügen von neuen Consumption Groups eine sehr Speicher und Rechenintensive Aufgabe. Da die wahrscheinlichsten

Algorithmus 3.5 Pseudocode für das Verwerfen einer Consumption Group.

```
ConsumptionGroupVerwerfen(ConsumptionGroup CG) begin
   CG.rechtesKind = null
   CG.elternKnoten.kind = CG.linkesKind
end function
```

Fensterversionen immer eine kleine Distanz zur Wurzel haben, werden die weit entfernten Fensterversionen fast nie ausgeführt. Aus diesem Grund ist es möglich, die Tiefe - für die Tiefe werden hier nur die Fensterknoten gezählt - des Abhängigkeitsbaumes zu begrenzen und das Hinzufügen neuer Fenster zu verzögern. Somit wird die Größe des Abhängigkeitsbaumes limitiert und das Klonen von Teilbäumen wird nicht exponentiell teuer.

Scheduling

Wenn sich der Abhängigkeitsbaum oder der Zustand der Fensterversionen ändert, müssen diese unter Umständen neu zugewiesen werden. Worker die bereits eine zu schdulende Fensterversion ausführen, werden vom reschedulen ausgenommen. Damit wird verhindert, dass diese Worker ihre Arbeit unterbrechen, um an einer anderen Fensterversion zu arbeiten. Durch Locks in den Fensterversionen wird verhindert, dass mehrere Worker die selbe Fensterversion ausführen. Algorithmus 3.6 stellt das Scheduling in Pseudocode dar.

Algorithmus 3.6 Pseudocode für das Scheduling.

```
schedule() begin
   zuSchedulen = {} // Liste
   freieWorker = workerThreads // Liste
   topk = TopkFensterKnoten(B, k)
   for each Fenster in topk
      if not Fenster.wirdAusgefuert then
         zuSchedulen.add(Fenster)
      else
         freieWorker.remove(Fenster.getWorker())
      end if
   end for
   for each Fenster in zuSchedulen do
      WorkerThread WT = freieWorker.pop()
      WT.assign(Fenster)
   end for
end function
```

3.3.7 Checkpoints

Mit der Zeit werden neue Events zu Consumption Groups hinzugefügt. Dadurch ist es möglich, dass aus einer Fensterversion Events entfernt werden können, während diese ausgeführt wird.

In der Regel passiert das, wenn eine Fensterversion schneller ausgeführt wird als dessen Elter. Für die Events, die bisher noch nicht betrachtet wurden, stehen keine Informationen zur Verfügung. Somit wird angenommen, dass diese nicht von einem früheren Fenster konsumiert werden. Falls diese Annahme falsch ist, muss der Vorgang rückgängig gemacht werden. Der Operator bietet allerdings keine Funktionalität, um einzelne ihm übergebene Events rückgängig zu machen. Es ist aber möglich den Zustand des Operators zu speichern. Somit werden in regelmäßigen Abständen Checkpoints erstellt, zu denen der Operator zurückgesetzt werden kann, sollte ein Event fälschlicherweise dem Operator übergeben worden sein. Der erste Checkpoint wird bei der Generierung der Fensterversion erstellt, bevor das erste Event dem Operator übergeben wurde.

3.4 Zusammenfassung

Die Basis des SPECTRE Systems bietet bereits ein Abhängigkeitsmanagement. Dieses wird als Baum dargestellt und bietet alle notwendigen Funktionen an. Der Path Manager verwaltet diesen Baum und vergibt die Arbeit an die Workerthreads.

4 Problembeschreibung

Das SPECTRE Framework ist bereits in der Lage Abhängigkeiten zwischen Fenstern zu erkennen. Dem System fehlt allerdings noch ein Weg, wie das Scheduling die besten Fensterversionen an die Worker vergibt. Eine suboptimale Vergabe der Fensterversionen an die Workerthreads führt zu einer entsprechend schlechten Performance. Naiv könnte man im Abhängigkeitsbaum immer die obersten Knoten vergeben, das würde aber die Wahrscheinlichkeiten der Consumption Groups außer acht lassen. Die Wahrscheinlichkeit auf Vervollständigung eines Partial Match und die damit verbundene Konsumierung des Compex Events verändert sich mit der Zeit. Je weiter das Fenster sich dem Ende nähert, desto unwahrscheinlicher wird das Konsumieren wenn das Pattern keine Fortschritte macht. Umgekehrt nimmt die Wahrscheinlichkeit zu, wenn schon am Anfang des Fensters große Teile des Pattern erfüllt sind.

4.1 Veranschaulichung von Abhängigkeiten

Als einfaches Beispiel soll hier das Pattern ABC erkannt werden. Die Fenster bestehen je aus 10 Events, als selection Policy ist earliest Selection gewählt, und jedes Event, das Teil eines Complex Events ist, wird konsumiert. Des weiteren wird nur das erste Event eines Fensters als Teil des Pattern verwendet. Der Splitter öffnet mit jedem A ein neues Fenster. Das Beispiel wird in Abbildung 4.1 verdeutlicht.

Die erzeugten Complex Events sollten hier $A_1B_1C_1$ und $A_2B_2C_3$ sein. Allerdings kann das zweite Complex Event erst erzeugt werden, sobald bekannt ist, welche Events im ersten Fenster F_1 konsumiert werden. Zu beachten ist auch, dass keines der Events, die in F_3 vorkommen, in F_1 konsumiert werden. Die Events $A_3B_2C_3$ stehen dennoch nicht mehr zur Verfügung, da sie in F_2 konsumiert werden.

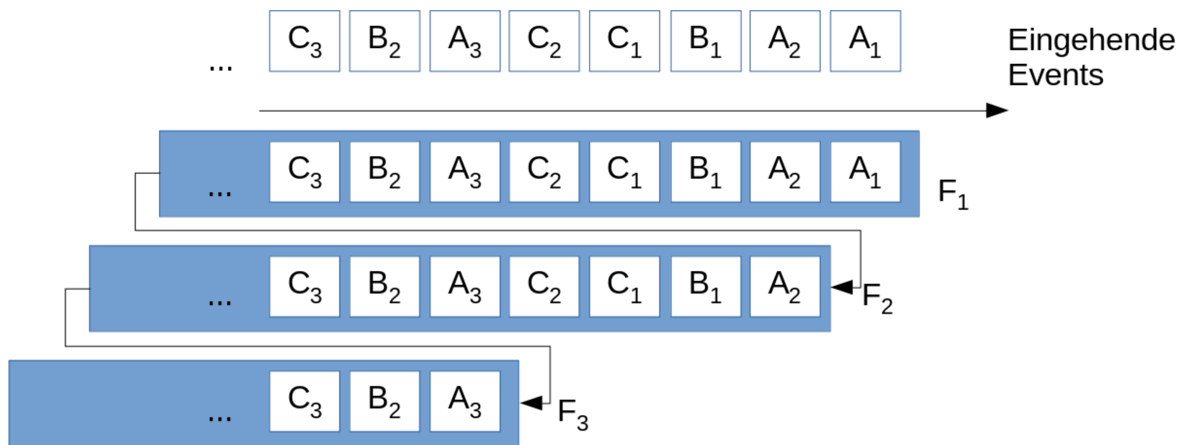

Abbildung 4.1: Darstellung des Eventstreams $A_1A_2B_1C_1C_2A_3B_2C_3$.

Es ist nicht ohne weiteres möglich, drei Worker auf den Fenstern $F_{1..3}$ arbeiten zu lassen, da das Erkennen eines Events in der Praxis mit Aufwand verbunden ist. Außerdem ist es nicht möglich, ein Event aus einem Partial Match wieder zu entfernen, da sich der interne Zustand des Operators ändert. Somit müsste mit jedem Event, das dem Operator fälschlicherweise übergeben wurde, das Fenster neu begonnen werden. Des Weiteren muss immer erst abgewartet werden, bis das Complex Event tatsächlich erzeugt wurde, da immer die Wahrscheinlichkeit besteht, dass es nicht zu Stande kommt und somit die Events doch für ein späteres Fenster zur Verfügung stehen.

Um alle Möglichkeiten abzudecken müssten für dieses einfache Beispiel 7 Worker parallel arbeiten wie an Abbildung 4.2 zu sehen ist.

Da das Pattern im ersten Fenster schon sehr früh fast fertiggestellt ist, kann man spekulieren, dass es vervollständigt werden wird. Daher macht es wenig Sinn, Rechenleistung in FV_2 zu investieren.

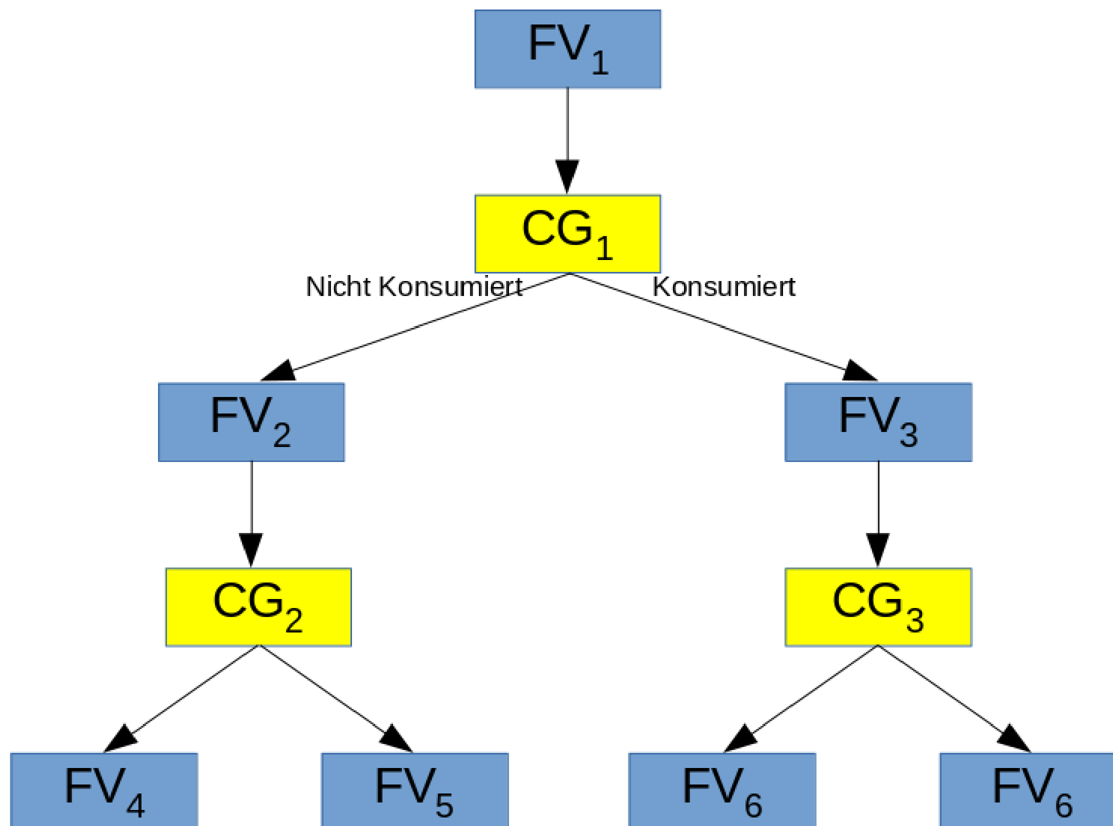

Abbildung 4.2: Abhängigkeitsbaum zum Eventstream $A_1A_2B_1C_1C_2A_3B_2C_3$.

4.2 Ziel der Arbeit

Das Ziel dieser Arbeit ist es, ein spekulatives Modell für die Bestimmung der Vervollständigungswahrscheinlichkeiten zu entwerfen und damit ein möglichst optimales Scheduling für SPECTRE zu ermöglichen.

4.3 Herausforderungen

Dadurch, dass sich Consumption Groups schnell ändern können, ändert sich auch deren Wahrscheinlichkeit. Das Model muss also schnell auf Änderungen reagieren können, um die neue Wahrscheinlichkeit zu berücksichtigen.

Zudem ist die Lernfähigkeit ein wichtiger Aspekt. Die Häufigkeit der Events im Eventstream ist nicht konstant, sondern ändert sich stetig.

Des Weiteren ist es nicht möglich den Zustandsgraphen der Operatoren einzusehen und zu verwenden. Es muss somit eine Vereinfachung dieses Graphen generiert werden.

5 Lösungsansatz

In diesem Kapitel werden die Erweiterungen für die spekulative Ausführung von Fensterversionen vorgestellt.

5.1 Architektur

Für die spekulative Ausführung wird das System um das Markov Model erweitert. Abbildung 5.1 stellt das Zusammenspiel dieser Komponenten dar. Die erwiderte Funktionalität wird in diesem Kapitel erläutert.

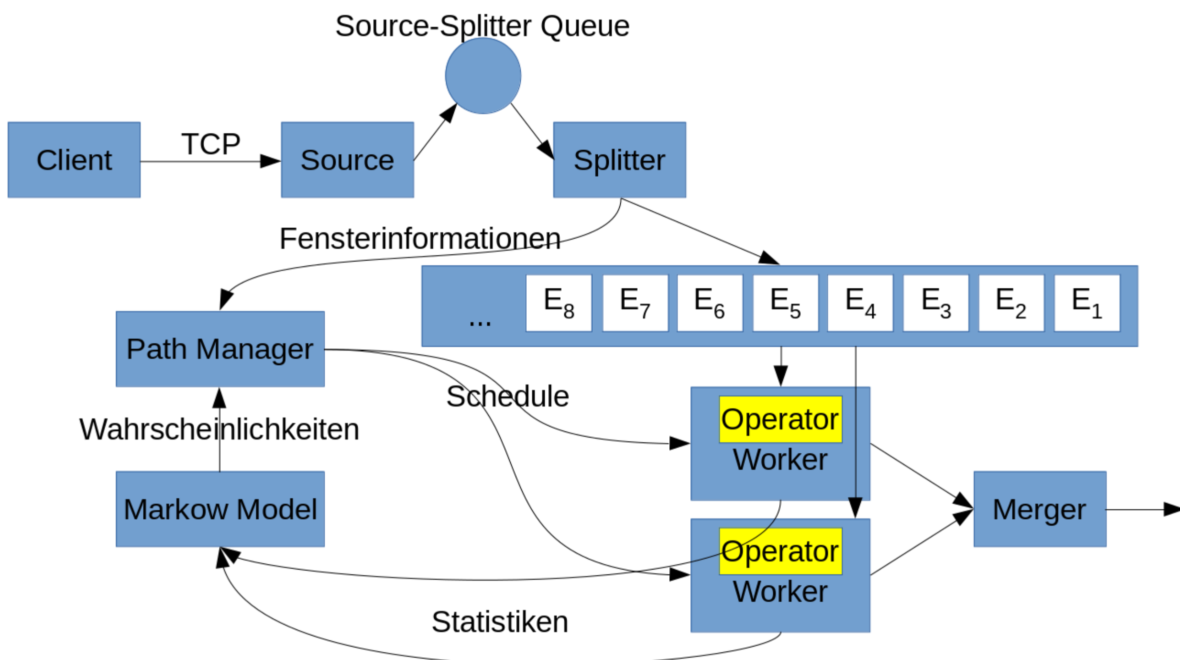

Abbildung 5.1: Zusammenhang der Komponenten

5.2 Wahrscheinlichkeit

Um zu gewährleisten, dass die wahrscheinlichsten Fensterversionen den Workerthreads zugewiesen werden, wird ein Model benötigt, um die Wahrscheinlichkeit möglichst genau und schnell zu berechnen.

5.2.1 Naiver Ansatz

Anhand der Häufigkeit verschiedener Events ist es möglich, falls das Pattern bekannt ist, die Wahrscheinlichkeit zu berechnen, mit der das Pattern in diesem Fenster noch vervollständigt wird. Je mehr vom Fenster noch übrig ist, desto wahrscheinlicher ist das Vervollständigen.

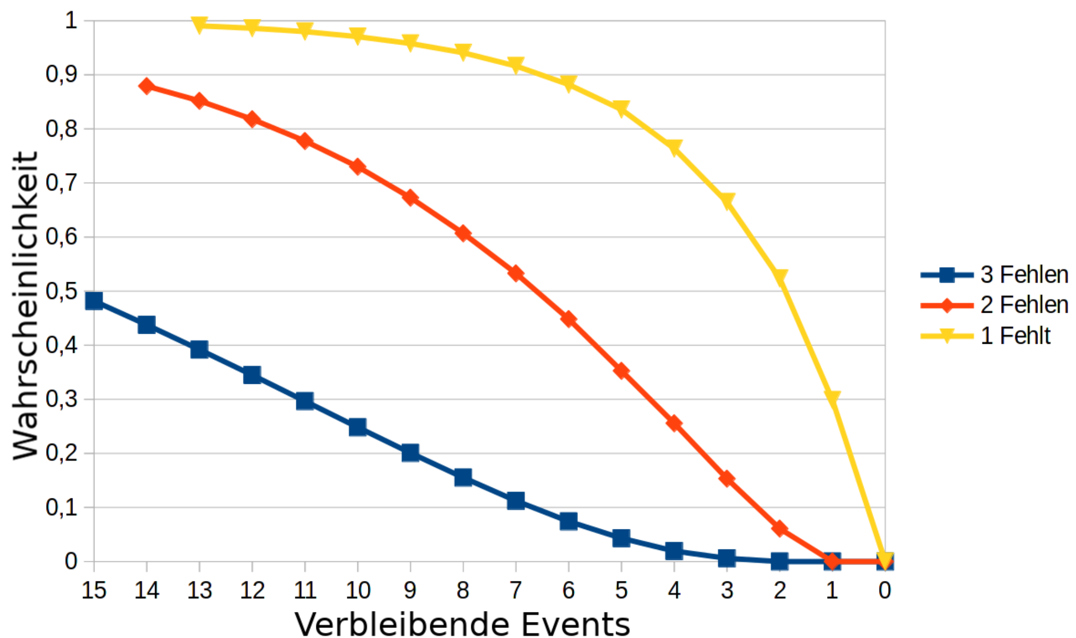

Abbildung 5.2: Wahrscheinlichkeiten, dass ein Partial Match noch Vervollständigt wird.

Das Diagramm in Abbildung 5.2 zeigt die Wahrscheinlichkeiten, dass das Pattern ABC bei entsprechender Restfensterlänge noch vervollständigt wird. Die Wahrscheinlichkeiten für A, B und C sind konstant und liegen bei 10% für A, 20% für B und 30% für C. Die Fensterlänge beträgt 15. Die Wahrscheinlichkeit des Complex Events liegt zu Beginn bei etwa 48%. Sollte das Event A nicht eintreten, sinkt die Wahrscheinlichkeit mit jedem Event bis es bei 2 verbleibenden Events 0 erreicht. Dies wird anhand der blaue Kurve veranschaulicht. Sobald ein A eingetreten ist gilt die rote Kurve und für AB die Gelbe.

Naiv lässt sich die Wahrscheinlichkeit der Vervollständigung eines Complex Events für ein fehlendes Event wie folgt berechnen:

$1 : P(Vollständig) = \sum_{x=0}^{l-1}(1 - P(E_n))^x \cdot P(E_n)$

Wobei $P(E_n)$ die Wahrscheinlichkeit darstellt, dass ein Event vom Typ E_n ist. Das Complexe Event besteht aus n Events. E_k ist das Event das an Stelle k benötigt wird. Somit ist E_n das letzte benötigte Event. Die Restlänge des Fensters wird als l bezeichnet. $P(Vollständig)$ ist die Summe aller Wahrscheinlichkeiten, dass E_n auftritt. Diese Formel lässt sich vereinfachen, wenn die Gegenwahrscheinlichkeit betrachtet wird:

$1 : P(Vollständig) = \sum_{x=0}^{l-1}(1 - P(E_n))^x \cdot P(E_n) = 1 - (1 - P(E_n))^l$

Soll allerdings berechnet werden, dass von dem Pattern noch 2 oder mehr Events fehlen, muss die Formel angepasst werden. Hierzu wird einfach die Wahrscheinlichkeit, dass das erste Event unter den ersten x Events liegt, mit der Wahrscheinlichkeit, dass der Rest des Patterns im Rest des Fensters erfüllt wird, multipliziert und alle diese Wege aufsummiert.

$2 : P = \sum_{x=0}^{l-2}(1 - P(E_{n-1}))^x \cdot P(E_{n-1}) \cdot (1 - (1 - P(E_n))^{l-x-1})$

$3 : P = \sum_{w=0}^{l-3}(1 - P(E_{n-2}))^w \cdot P(E_{n-2}) \cdot \sum_{x=0}^{l-w-3}(1 - P(E_{n-1}))^x \cdot P(E_{n-1}) \cdot (1 - (1 - P(E_n))^{l-x-w-2})$

$4 : P = \sum_{v=0}^{l-4}(1-P(E_{n-3}))^v \cdot P(E_{n-3}) \cdot \sum_{w=0}^{l-v-4}(1-P(E_{n-2}))^w \cdot P(E_{n-2}) \cdot \sum_{x=0}^{l-w-v-4}(1-P(E_{n-1}))^x \cdot P(E_{n-1}) \cdot (1 - (1 - P(E_n))^{l-x-w-v-3})$

...

Jedes fehlende Event fügt der Funktion eine neue Summe hinzu. Da jede Summe als Schleife berechnet werden muss, hat dieser Ansatz eine Laufzeit von $\mathcal{O}(l^{\text{Patterngröße}-1})$.

5.2.2 Markov Ketten

Eine Markov Kette ist ein stochastischer Prozess, mit dem es möglich ist die Wahrscheinlichkeit des Eintretens zukünftiger Ereignisse zu ermitteln. Der Zustandsraum muss hierfür endlich oder abzählbar unendlich sein. Des weiteren wird zwischen gedächtnislosen Markov Ketten und Markov Ketten n-ter Ordnung unterschieden. Für gedächtnislose Markov Ketten ist für den nächsten Zustandsübergang nur der aktuelle Zustand relevant, während Markov Ketten n-ter Ordnung die n letzten Zustände berücksichtigt [Pan03]. Das zu lösende Problem hat einen endlichen Zustandsraum, weswegen auf unendlich Markov ketten nicht weiter eingegangen wird. Ebenso sollten Events, die nicht zum Complex Event beitragen, nicht den Zustand beeinflussen, sonder nur die Übergangswahrscheinlichkeiten. Somit wird sich hier auf gedächtnislose endliche Markov Ketten beschränkt.

Beispiel für eine gedächtnislose endliche Markov Kette

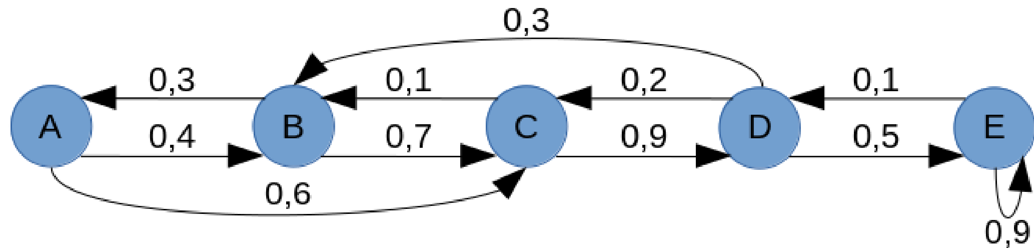

Abbildung 5.3: Ein Zustandsgraph mit Übergangswahrscheinlichkeiten, der als Beispiel für die Generierung einer Markov Matix genutzt wird.

Für den Startzustand A in Abbildung 5.3 lassen sich die Übergangswahrscheinlichkeiten wie folgt beschreiben:

$$P(X_{t+1} = i | X_t = A) = \begin{cases} 0 & \text{falls } i = A \\ 0,4 & \text{falls } i = B \\ 0,6 & \text{falls } i = C \\ 0 & \text{falls } i = D \\ 0 & \text{falls } i = E \end{cases}$$

Diese Zustandsübergänge müssen für jeden Zustand angegeben werden.

Diese Art von Markov Ketten lassen sich auch mit einer Matrix darstellen. Für den Zustandsgraph in Abbildung 5.3 würde die Markov Matrix folgendermaßen aussehen:

$$\begin{pmatrix} 0 & 0,4 & 0,6 & 0 & 0 \\ 0,3 & 0 & 0,7 & 0 & 0 \\ 0 & 0,1 & 0 & 0,9 & 0 \\ 0 & 0,3 & 0,2 & 0 & 0,5 \\ 0 & 0 & 0 & 0,1 & 0,9 \end{pmatrix}$$

Die erste Zeile der Matrix steht hier für die möglichen Zustandsübergänge des Zustandes A. Die Wahrscheinlichkeit von A in den Zustand A zu wechseln beträgt 0, zu B 0,4, zu C 0,6 und 0 zu D und E.

Angenommen der Startzustand ist A. Um nun die Zustandsverteilung nach k Schritten zu berechnen muss nun die Start Zustandsverteilung mit der k-ten Potenz der Matrix multipliziert werden:

$$\begin{pmatrix}1 & 0 & 0 & 0 & 0\end{pmatrix} \cdot \begin{pmatrix} 0 & 0,4 & 0,6 & 0 & 0 \\ 0,3 & 0 & 0,7 & 0 & 0 \\ 0 & 0,1 & 0 & 0,9 & 0 \\ 0 & 0,3 & 0,2 & 0 & 0,5 \\ 0 & 0 & 0 & 0,1 & 0,9 \end{pmatrix}^k$$

5.2.3 Umsetzung des Markov Models

Um Markov Ketten nutzen zu können, muss ein Zustandsgraph erstellt werden. Da der Operator eine Blackbox ist, kann sein interner Zustandsgraph leider nicht verwendet werden. Aus diesem Grund muss der für die Markov Ketten verwendete Zustandsgraph vereinfacht werden.

Vom Operator kann das Model nur die Anzahl der für das Complex Event mindestens benötigten Events bekommen. Zudem hat es Zugriff auf das Partial Match. Als Zustandsgraph wird die Anzahl der noch für ein Complex Event benötigten Events genommen. Für diese Arbeit wurde davon ausgegangen, dass der Operator nur eine Art von Complex Event erkennt.

Erstellung der Zustandsmatrix

Die Zustandsmatrix ist eine quadratische Matrix mit einer Spalten und Zeilenanzahl gleich der minimalen Anzahl an Events die es benötigt, um ein Complex Event zu vervollständigen. Zustand 0 bedeutet hier, dass das Complex Event vervollständigt wurde. Um die Wahrscheinlichkeiten der Zustandsübergänge zu ermitteln, wird für eine beim Programmstart festgelegte Anzahl von Events untersucht, wie viele Events es benötigt, um von einem Zustand in den nächsten zu gelangen. Hierbei sei angemerkt, dass die Anzahl der Events durch die von der Source vergebene ID errechnet wird. Somit werden auch Events mitgezählt, die nicht Teil der Fensterversion sind.

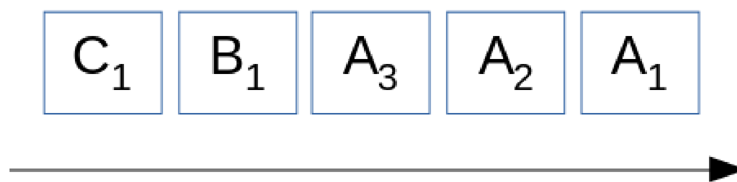

Abbildung 5.4: Beispielhafter Eventstream zur Markov Matrix Erstellung.

Für den in Abbildung 5.4 gezeigten Eventstream und dem Pattern ABC würden die Messdaten wie folgt aussehen:

$$\begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 2 \end{pmatrix}$$

A_1 hat den Partial Match erzeugt und zurückgegeben, dass es noch mindestens 2 weitere Events benötigt, um das Complex Event zu vervollständigen. Daraus resultiert eine 3x3 Matrix, da es noch drei Zustände gibt, die das Partial Match annahmen kann. Der letzte Zustand ist das vollständige Complex Event. Es hat 3 weitere Events benötigt um in den nächsten Zustand zu gelangen, 2 haben also den Zustand nicht verändert. Daraus ergibt sich die letzte Zeile der Matrix: 2 Zustandsübergänge vom Zustand 2 in den Zustand 2 und ein Übergang in den Zustand 1. Das letzte fehlende Event C kam danach sofort. Also ein Übergang in den Zustand 0 in der mittleren Zeile. Die Matrix wird solange gefüllt, bis eine gewisse Anzahl an Events gemessen wurde. Die erste Zeile wird nie Messwerte erhalten, da fertige Complex Events nicht weiter beobachtet werden. Sobald das der Fall ist, wird der linke obere Wert auf "1"gesetzt, es ist nicht möglich den Zustand des Vollständigen Complex Events wieder zu verlassen. Um daraus eine Zustandsübergangsmatrix zu machen, wird nun jede Zeile durch die Summe ihrer Einträge geteilt:

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0,33 & 0,67 \end{pmatrix}$$

Für die Erstellung der Matrix werden ausschließlich Daten von korrekten Fensterversionen verwendet. Somit fallen Spekulationen nicht ins Gewicht, allerdings müssen die Worker diese Informationen zwischenspeichern, falls sie nicht am Wurzelknoten arbeiten. Sobald ein Knoten zur Wurzel wird, werden die Informationen an das Markov Model übergeben.

Falls ein Complex Event nicht beendet wird, bevor das Ende des Fensters erreicht ist, wird es so gewertet, dass der Operator für die Zeit der letzten Events in seinem aktuellen Zustand geblieben ist.

Verwenden des Models

Um die Wahrscheinlichkeit eines Complex Events zu erhalten, wird die voraussichtliche Restlänge des Fensters benötigt. Dazu kommt die Anzahl der Events, die für die Vervollständigung des Partial Match mindestens noch nötig sind. Die fehlende Anzahl an Events stellt den Startzustand dar und die Restlänge des Fensters die Anzahl der Zustandsübergänge. Somit ist es möglich, wie im Beispiel für die Markov Kette, die Wahrscheinlichkeit zu berechnen. Die Wahrscheinlichkeit des Zustandes 0 gibt hierbei die Vervollständigungswahrscheinlichkeit an.

Das Problem ist allerdings, dass Matrixmultiplikation eine rechenintensive Aufgabe darstellt. Der naive Ansatz hat eine Laufzeit von $\mathcal{O}(n^3)$, diese Laufzeit lässt sich zwar durch verschiedene Algorithmen, wie z.B. den Strassen-Algorithmus reduzieren, allerdings müssen hier in der Regel viele dieser Matrixmultiplikationen durchgeführt werden [Wei]. Der Path Manager muss für jedes Partial Match schnellstmöglich die Wahrscheinlichkeit bekommen.

Um dieses Problem zu beheben, werden nach der Erstellung der Zustandsübergangsmatrix verschiedene Potenzen der Matrix vorberechnet. Die vorberechneten Matrizen sind $M, M^{j+1}, M^{2j+1}, ..., M^{kj+1}$. Die Werte für j und k werden beim Programmstart übergeben. Die Matrix M^l gibt die Zustandsübergänge so an, als würden l Schritte nacheinander gemacht. Die Zahlen in der ersten Spalte geben also an, wie groß die Wahrscheinlichkeit ist vom jeweiligen Zustand innerhalb von l Schritten das Partial Match zu vervollständigen.

Da die erwartete Restfenstergröße nicht unbedingt vorberechnet wurde, wird in diesem Fall zwischen den zwei am nächsten gelegenen vorberechneten Matrizen linear interpoliert. Sollte eine größere Fenstergröße gefordert werden als Matrizen berechnet wurden, wird stattdessen die Wahrscheinlichkeit der größten berechneten Potenz verwendet. Die Laufzeit des Abfragens der Wahrscheinlichkeit ist somit immer $\mathcal{O}(1)$. Algorithmus 5.1 zeigt den Pseudocode für das Erhalten der Wahrscheinlichkeit. Algorithmus 5.2 beschreibt, wie das Empfangen einer Consumption Group verarbeitet wird.

Algorithmus 5.1 Pseudocode: berechneWahrscheinlichkeit

```
berechneWahrscheinlichkeit(int Restlaenge, int FehlendeEvents) begin
   if Restlaenge == 0 then
      return 0.0
   end if
   if FehlendeEvents == 0 then
      return 1.0
   end if
   if Restlaenge > k∗j+1
      return Matrix[k](FehlendeEvents, 0)
   end if
   untererIndex = (Restlaenge - 1) / j
   obererIndex = untererIndex + 1
   Rest = (windowLeftLocal - 1) modulo j
   unteresErgebnis = Matrix[untererIndex](FehlendeEvents, 0)
   oberersErgebnis = Matrix[obererIndex](FehlendeEvents, 0)
   return (oberesErgebis - unteresErgebnis) / j ∗ Rest + unteresErgebnis
end function
```

Aktualisieren der Matrix

Die Wahrscheinlichkeiten der Zustandsübergänge können sich mit der Zeit ändern. Somit ist es notwendig, dass die Matrix regelmäßig aktualisiert wird. Wie bereits erwähnt, besteht eine Messung aus einer festgelegten Anzahl an gemessenen Events. Sobald diese erreicht

Algorithmus 5.2 Pseudocode: empfangeCGroup

```
empfangeCGroup(ConsumptionGroup CG) begin
   eventsSeitLetztenUpdate = CG.letztesEvent.id - this.letztesUpdate
   if eventsVerbleibendMap ist not empty then
      bearbeiteteEvents = bearbeiteteEvents + eventsSeitLetztenUpdate
   end if

   // Behandele alle offenen Consumption Groups als waeren sie im aktuellen zustand
       geblieben
   for each eventsVerbleibend in eventsVerbleibendMap
      ResultatMatrix[eventsVerbleibend, eventsVerbleibend] += eventsSeitLetztenUpdate
   end for

   iterator = eventsVerbleibendMap[CG.id]
   if exists iterator then
      ResultatMatrix[iterator.eventsVerbleibend, iterator.eventsVerbleibend] -= 1
      ResultatMatrix[iterator.eventsVerbleibend, CG.eventsVerbleibend] += 1
   else
      eventsVerbleibendMap.add(CG)
   end if

   this.letztesUpdate = CG.letztesEvent.id

   if bearbeiteteEvents >= EventsProMatrix then
      berechneNaechsteMatrix()
      bearbeiteteEvents = 0
      clear(ResultatMatrix)
   end if
end function
```

ist, wird damit begonnen die Matrix zu berechnen. Solange bis die nächste Matrix und alle vorberechneten Potenzen fertiggestellt sind, wird weiterhin die alte Matrix verwendet.

Damit eine Messung, die extreme Ergebnisse erzielt hat, nicht die sonst durchschnittliche Matrix ersetzt, werden die Matrizen miteinander verrechnet. Hierfür wird exponentielle Glättung verwendet:

$$M_n = M_{n-1} \cdot (\alpha - 1) + A * \alpha$$

wobei α die Gewichtung der gerade ermittelten Matrix ist.

Matrix Minimierung

Das Vorberechnen kann bei großen Matrizen immer noch eine lange Zeit in Anspruch nehmen. In diesem Fall gibt es die Möglichkeit die Schrittgröße j zu erhöhen und die Anzahl der vorauszuberechnenden Matrizen zu reduzieren. Allerdings ist das nicht immer ausreichend.

Eine weitere Möglichkeit bietet das Zusammenfassen von mehreren Zuständen. Der Matrixminimierungsfaktor gibt an, wie viele Zustände als einer gewertet werden sollen. Dadurch kann die Größe der Matrix um ein vielfaches verkleinert werden.

5.3 Scheduling

Dem System stehen eine bestimmte Anzahl von Workerthreads zur Verfügung. Da der Abhängigkeitsbaum exponentiell mit der Anzahl an Consumption Groups wächst, können in der Regel nur ein Bruchteil der Fensterknoten ausgeführt werden.

5.3.1 Top k

Mithilfe des Top-k Algorithmus lassen sich die k-wahrscheinlichsten Fensterknoten aus dem Abhängigkeitsbaum finden. Jede Consumption Group hat eine gewisse Wahrscheinlichkeit, vervollständigt zu werden. Wie diese Wahrscheinlichkeit berechnet wird, findet sich in diesem Kapitel unter Markov Modell. Um die Wahrscheinlichkeit eines beliebigen Knotens zu erhalten, muss das Produkt der Vervollständigungswahrscheinlichkeit aller Consumption Groups, von dem dieser Knoten abhängt, berechnet werden. Wird für den Knoten davon ausgegangen, dass Consumption Groups nicht vervollständigt werden, wird die Gegenwahrscheinlichkeit verwendet. Für einen Fensterknoten gibt dieser Wert an, wie wahrscheinlich es ist, dass diese Fensterversion richtig spekuliert. Ein Consumption Group Knoten hat zwei verschiedene Wahrscheinlichkeiten: Wie wahrscheinlich der aktuelle Weg ist, und die eigene Vervollständigungswahrscheinlichkeit, sollte der Weg eintreten.

Die Wahrscheinlichkeit des Wurzelknotens ist 100%. Die Wahrscheinlichkeitsverteilung des Abhängigkeitsbaums ist ein Max-Heap, die Wahrscheinlichkeit der Elternknoten ist also immer mindestens so groß wie die der Kindknoten. Dadurch ist es zur Ermittelung der k-wahrscheinlichsten Fensterknoten nicht nötig den gesamten Baum zu betrachten. In Algorithmus 5.3 ist der Pseudocode zu sehen.

Solange die Anzahl der Complex Events pro Fenster beschränkt ist, ist die Laufzeitkomplexität des Algorithmus $\mathcal{O}(k \cdot log(k))$. Da die Prioritätswarteschlange als Baum implementiert wurde, hat das Einfügen und das Entfernen je eine Laufzeit von $\mathcal{O}(k)$. Diese Operationen werden ausgeführt bis die k wahrscheinlichsten Knoten gefunden sind.

Die Consumption Group Wahrscheinlichkeit wird für jede besuchte Consumption Group berechnet, während der Algorithmus ausgeführt wird. Somit werden nur die Wahrscheinlichkeiten berechnet, die für das Scheduling relevant sind.

Algorithmus 5.3 Pseudocode: Top K Algorithmus.

```
TopkFensterknoten(Abhaengigkeitsbaum B, integer k) begin
   result = {}      //Menge
   kandidaten = B.wurzel //Prioritaetswarteschlange sortiert nach Wahrscheinlichkeit
   while |resumt| < k and kandidaten.isNotEmpty do
      tmp = candidates.pop()
      if tmp is Fensterknoten then
         result.add(tmp)
      for each kind von tmp
         if kind ist linkes Kind eines Consumption Group Knoten
            kind.wahrscheinlichkeit = tmp.wahrscheinlichkeit * (1 -
                berechneWahrscheinlichkeit(tmp.CGWahrscheinlichkeit))
         else if kind ist rechtes Kind eines Consumption Group Knoten
            kind.wahrscheinlichkeit = tmp.wahrscheinlichkeit *
                berechneWahrscheinlichkeit(tmp.CGWahrscheinlichkeit)
         else // Kind eines Fensterknotens
            kind.wahrscheinlichkeit = tmp.wahrscheinlichkeit
         kandidaten.add(kind)
      end for
   end for
   return result
end function
```

5.4 Weitere Optimierung

Um die Leistung des Systems zu verbessern, wurden an anderen Stellen weitere Optimierungen durchgeführt.

5.4.1 Garbage Collection Thread

Damit der Path Manager so schnell wie möglich auf Veränderung des Abhängigkeitsbaums reagieren kann, muss er für den Mainloop so wenig Zeit wie möglich benötigen. Das Vervollständigen oder Verwerfen einer Consumption Group schneidet einen großen Teil des Baumes ab, der rekursiv gelöscht werden muss. Da es nicht wichtig ist, dass der Speicher sofort wieder freigegeben wird, werden Speicherfreigaben nicht vom Path Manager ausgeführt, sondern an einen Garbage Collection Thread übergeben. Damit das Übergeben an den Garbage Collection Thread möglichst schnell geschieht, wird hierfür ein Lockfreier Ringbuffer verwendet. Sollte der Grabage Collection Thread zu langsam sein - der Ringbuffer also voll ist - während der Path Manager weiteren Speicher freigeben möchte, muss er das solang selbst durchführen, bis wieder Platz im Ringbuffer ist.

5.4.2 tcmalloc

Während der Testläufe hat sich herausgestellt, dass die von der Gnu Compiler Collection (GCC) standardmäßig verwendete Speichervergabe für diese Anwendung zu langsam ist. Aus diesem Grund wird stattdessen auf tcmalloc [SG] zurückgegriffen, da es für die Verwendung in Multithread-Programmen optimiert wurde.

5.5 Zusammenfassung

Durch die Verwendung von Markov Ketten wurde ein Model entwickelt, mit dem es möglich ist, die Wahrscheinlichkeiten der einzelnen Consumption Groups zu bestimmen. Das Model sammelt während der Ausführung des Systems andauernd Statistiken der Operatoren und verwendet diese um ein Zustandsmodel und dessen Zustandsübergangswahrscheinlichkeiten zu erstellen. Zu diesem Zustandsmodel wird eine Markov Ketten Matrix erstellt und damit die Vervollständigungswahrscheinlichkeiten für vorher bestimmte Restlängen des Fensters zu berechnen.

Anhand von linearer Interpolation werden die Wahrscheinlichkeiten der dazwischen liegenden Restlängen berechnet. Dadurch, dass die Wahrscheinlichkeiten vieler Restlängen schon im voraus berechnet werden, ist das Abfagen der Wahrscheinlichkeitswerte in $\mathcal{O}(1)$ möglich.

Durch exponentielle Glättung wird das Model aktuell gehalten.

Für den Path Manager wurde ein Top K Algorithmus beschrieben, mit dem es möglich ist, die k wahrscheinlichsten Fensterversionen zu finden, ohne den gesamten Baum betrachten zu müssen.

6 Evaluation

In diesem Kapitel wird die Leistungsfähigkeit von SPECTRE evaluiert. Speziell wird hier darauf eingegangen, wie verschiedene Parameter das System beeinflussen. Gemessen wird hauptsächlich der Event-Throughput, aber auch die Latenz der Complex Events und die Geschwindigkeit des Path Manager. Simuliert wird ein Szenario im automatischen Handel.

6.1 Rahmenbedingungen

Im folgenden werden die Rahmenbedingungen der Evaluation beschrieben.

6.1.1 Platform

Als Testplattform kommt ein Supermicro X8OBN Server mit 8 x 10 Kerne Xeon E7 8870@2.2GHz CPUs und 1 TB RAM. Auf dieser läuft ein Openstack Icehouse. Die Experimente wurden in einer virtuellen Maschine mit 24 virtuellen CPUs und 32GB RAM ausgeführt. Als Betriebssystem kam Ubuntu 14.04 LTS zum Einsatz.

6.1.2 Testsystem

SPECTRE ist in C++14 implementiert. Die Datenstrukturen sind lockfree, mit Ausnahme der Ausführung der Fensterversionen, die nur von einem Worker zu einem Zeitpunkt bearbeitet werden können. Als zusätzliche Bibliotheken kommen Boost [Boo] und für die Matrixberechnungen Eigen [GJ+10] zum Einsatz.

Bisher ist noch kein TESLA oder SNOOP Parser implementiert, weswegen der Splitter und der Operator im Testsystem hardcoded sind. Der Client ist in C implementiert. Als Compiler kommt GCC 4.9 (G++ 4.9) mit der Optimierungsstufe -O2 zum Einsatz. Des Weiteren wird zur Speicherverwaltung gegen libtcmalloc[SG] gelinkt. Der Client kommuniziert über TCP mit der Source und wird in der selben virtuellen Instanz ausgeführt. Der Client schickt die Events so schnell wie möglich, dadurch kommt es nicht vor, dass SPECTRE zeitweise keine Events mehr zu bearbeiten hat. Die Geschwindigkeit des Client wird durch den TCP-Buffer limitiert.

6.1.3 Daten und Query

Bei den verwendeten Daten handelt es sich um Intraday Handel von ca. 3000 Stock Symbols des New York Stock Exchange über einen Zeitraum von zwei Monaten. Bezogen wurden diese Daten von Google Finance. Zwischen zwei Daten des selben Stock Symbols liegt eine Minute. Insgesamt sind es zusammen etwa 24 Millionen Daten.

Die Daten haben die Form:

```
StockSymbol,UnixTimeStamp,Anfangswert,HoechsterWert,NiedrigsterWert,Abschlusswert

AAAP,1477060200.0,38.56,38.56,38.56,38.56
FLWS,1477060200.0,9.29,9.29,9.29,9.29
FCCY,1477060200.0,13.55,13.55,13.55,13.55
```

Ziel der Query ist es herauszufinden, inwiefern das Steigen und Fallen des Kurses der großen Unternehmen wie z.B. Apple, IBM, ... einen Einfluss auf andere Unternehmen hat. Der Splitter öffnet jedes mal ein neues Fenster, sobald ein Event eines solchen Unternehmens erkannt wird. Diese Fenster haben eine bei den Programmparametern festgelegte Länge. Handelt es sich bei dem öffnenden Event um ein Steigen des Kurses (also $Abschlusswert > Anfangswert$) versucht der Operator eine gewisse Anzahl an Events zu erkennen und konsumiert diese. Entsprechendes gilt für fallende Anfangsevents ($Abschlusswert \leq Anfangswert$). Wird ein Complex Event erkannt, werden alle Events, die daran beteiligt sind, konsumiert. Des Weiteren ist anzumerken, dass die Stock Symbols, die als Öffnendes Event verwendet werden, niemals als Teil eines anderen Complex Events Verwendung finden. Die Anzahl der Complex Events pro Fenster ist auf 1 begrenzt.

6.1.4 Parameter

Um den Einfluss verschiedener Parameter auf SPECTRE zu erfassen, werden Standard Parameter verwendet. Bei den verschiedenen Versuchen werden immer nur die Parameter verändert, die hier speziell zum Einsatz kommen.

Die Parameter sind Folgende, in Klammern gesetzten Werte geben den Standardwert an: Anzahl der Workerthreads (8), Patterngröße (80), Fenstergröße in Sekunden (120), Abstand zwischen zwei Checkpoints (deaktiviert), maximale Abhängigkeitsbaumgröße (16), Alpha (0.7), Events für jede Matrixmessung (10000), Abstand der Potenzen (50), Anzahl berechneter Matrixpotenzen (120), Matrixminierungsfaktor (1).

Bei den verschiedenen Testläufen werden immer der minimale und maximale Wert angegeben. Sofern nicht anders spezifiziert, ist der Wert des nächsten Schrittes immer das Doppelte des Vorangegangenen. Wird ein Wertebereich von 1-16 abgegeben, bedeutet das, dass die Werte 1, 2, 4, 8 und 16 verwendet werden.

6.2 Evaluation

Es folgen die Ergebnisse der Testläufe. Jedes Testsetup wurde 10 mal unter gleichen Bedingungen durchgeführt, um die Auswirkung von extremen Werten zu minimieren.

6.2.1 Anzahl der Workerthreads

Zunächst werden die Auswirkungen der Anzahl der parallel arbeitenden Workerthreads gemessen. Das System benötigt ohne Workerthreads bereits 7 Threads, für Client, Source, Splitter, Path Manager, Garbage Collection, Markov Modell und Merger, von den 24 vCPUs stehen also noch maximal 17 zur Verfügung. Der Wertebereich der Anzahl von Workerthreads ist auf 1-16 gesetzt.

Ergebnisse

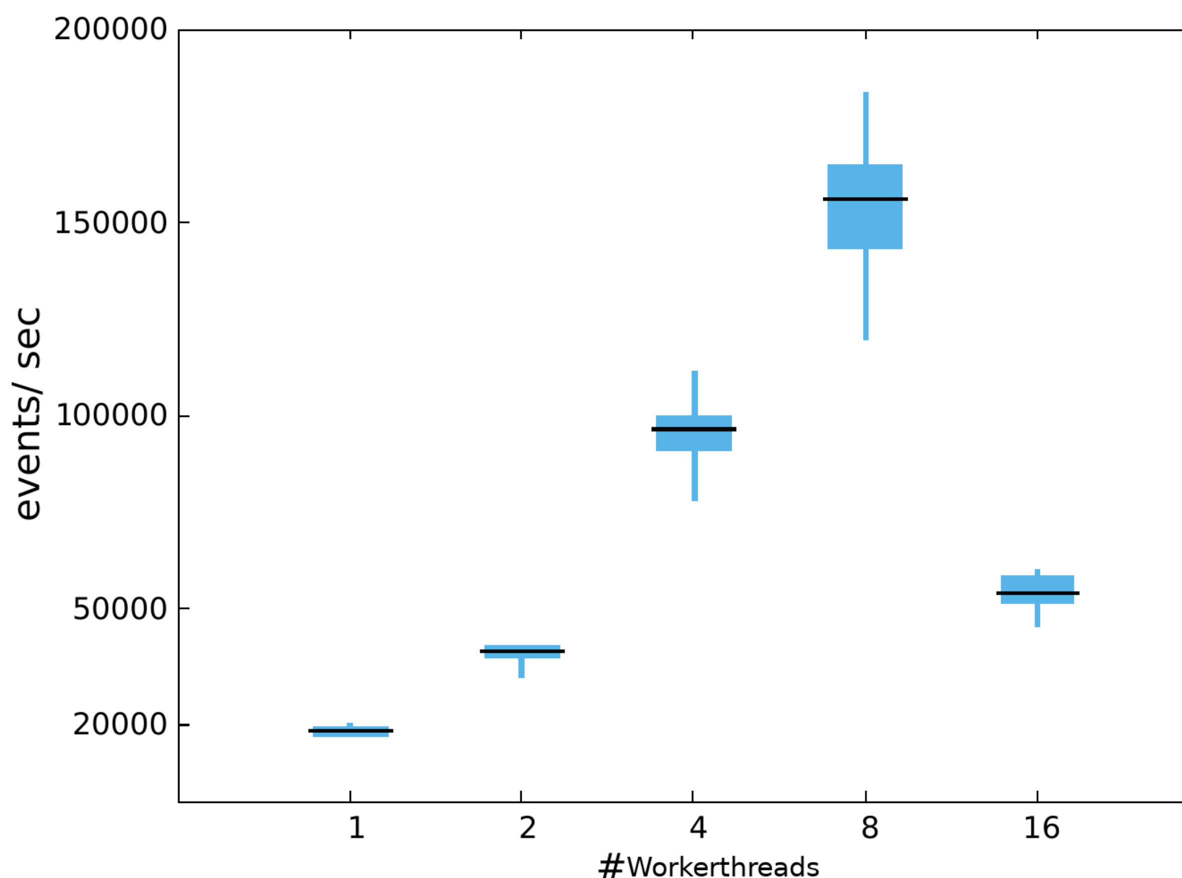

Abbildung 6.1: Events pro Sekunde für 1-16 Workerthreads.

Es ist anhand von Abbildung 6.1 leicht erkennbar, dass das Erhöhen der Workerthread-Anzahl, den Durchsatz deutlich erhöht. Ein einzelner Worker kann bei diesem Test etwa 2000 Events pro Sekunde bearbeiten. Bis 8 Worker nimmt die Performance fast linear zu. Für 16 Worker bricht allerdings die Performance ein.

Ähnliches sieht man in Abbildung 6.2. Die Dauer bis ein Complex Event erkannt wird, nimmt von 1 bis 4 Threads fast linear ab von ca 100ms für einen Worker bis 30ms für 4. Bei 8 Workerthreads ist noch eine kleine Verbesserung erkennbar, bei 16 Worker wird die Leistung wieder signifikant schlechter.

Abbildung 6.3 zeig einen möglichen Grund hierfür. Wie zu erkennen ist, nimmt die Schdulefrequenz des Path Manager deutlich ab. Je mehr Fensterversionen gleichzeitig ausgeführt werden, desto mehr Arbeit muss für das Verwalten dieser aufgebracht werden. Eine weitere Erklärung für das schlechte Abschneiden der 16 Workerthreads ist die begrenzte Bandbreite des Speichers. Dadurch können nur eine limitierte Anzahl an Speicherzugriffen stattfinden und die wenig wahrscheinlichen Fensterversionen belegen einen Teil dieser Bandbreite.

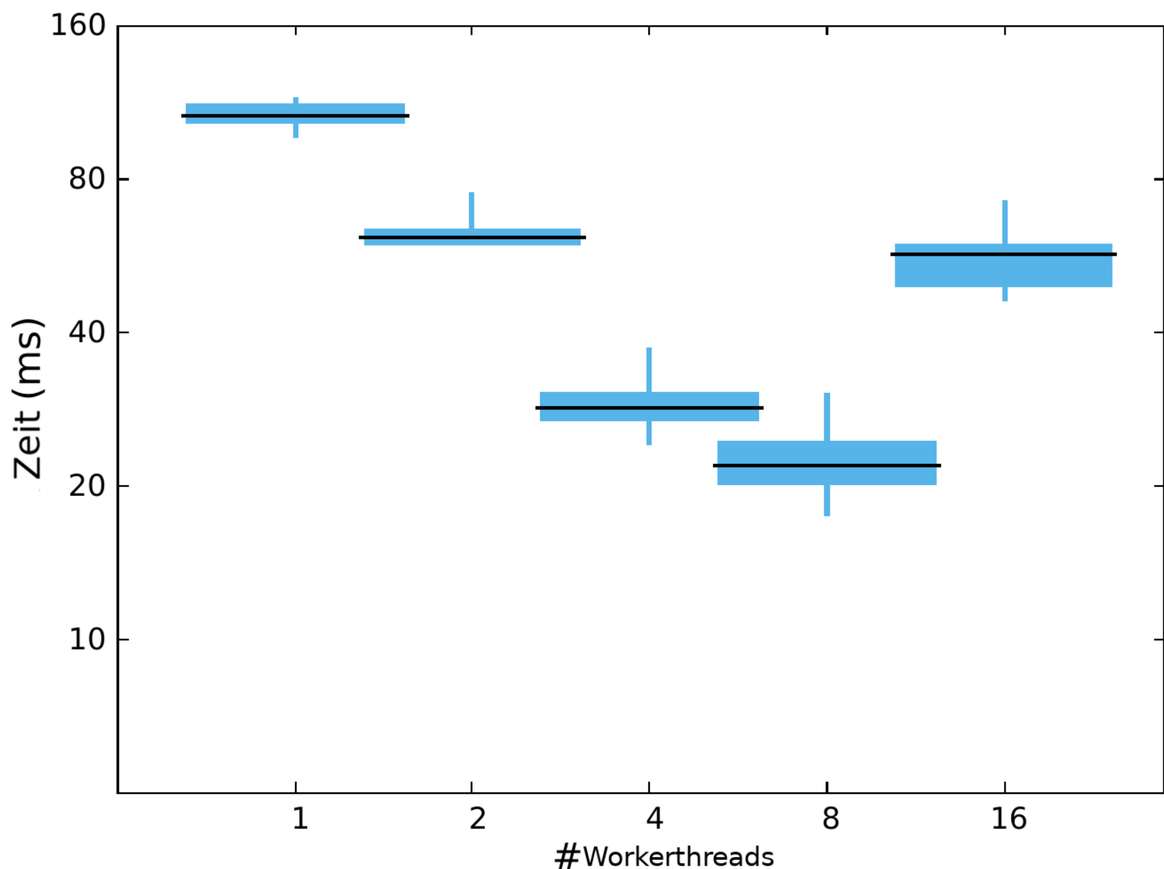

Abbildung 6.2: Dauer der Erkennung des Complex Events seit das Erste Event das System betreten hat.

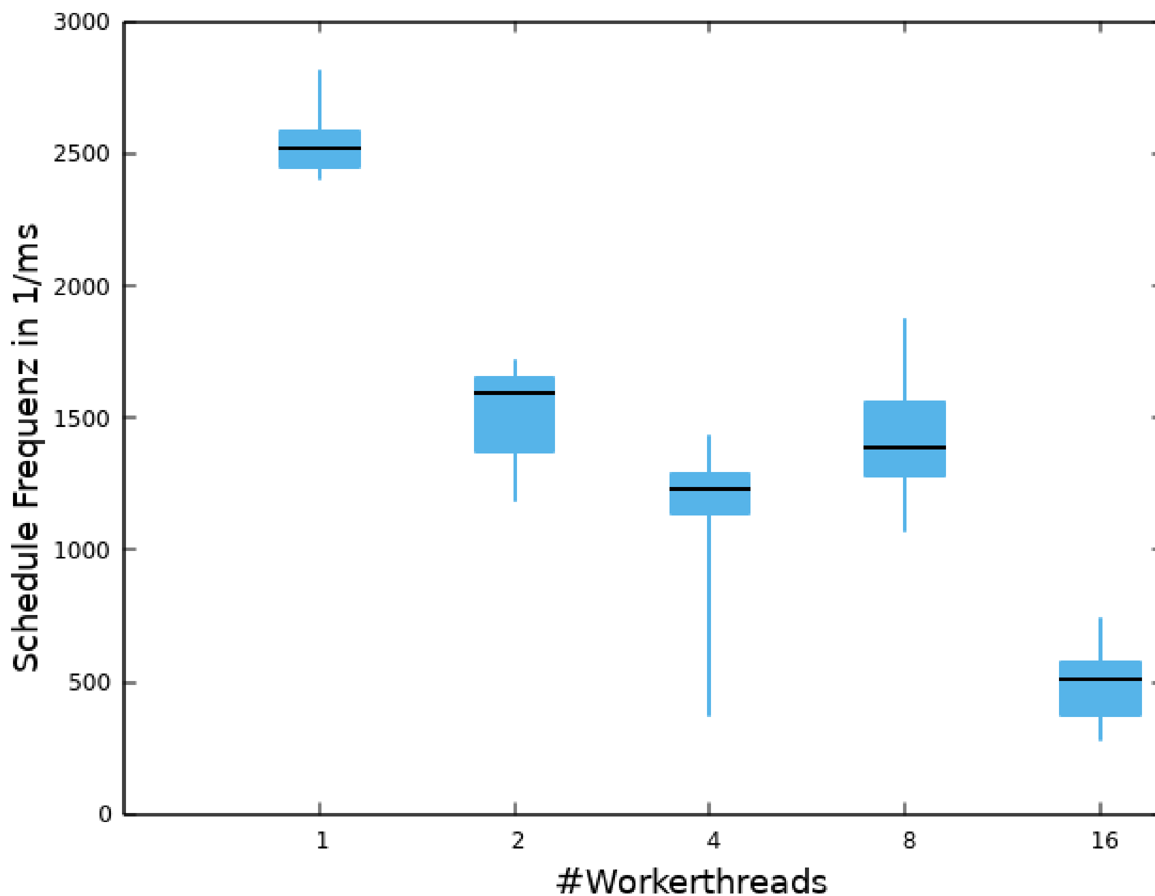

Abbildung 6.3: Frequenz der Schedulings des Path Manager

6.2.2 Patterngröße

Die Patterngröße wurde auf einen Wertebereich von 10-160 gesetzt. Durch das variieren der Patterngröße wird einerseits die Wahrscheinlichkeit, dass ein Complex Event erzeugt wird, verändert. Zum anderen ändert sich auch die Anzahl der Events, die ein früheres Fenster konsumiert. Da das Berechnen von Matrizen der Größe 160x160 sehr Zeitaufwändig ist, wurde hier der Matrixminimierungsfaktor auf 2 gesetzt.

Ergebnisse

Es verwundert nicht, dass durch eine kleine Patterngröße, ein deutlich erhöhter Durchsatz erzielt wird, wie in Abbildung 6.4 zu sehen ist. Die Anzahl der Events, die ein früheres Fenster konsumiert, ist deutlich geringer. Dadurch kommt es seltener zu Fehlspekulationen. Des Weiteren muss der Operator nicht mehr arbeiten sobald das Complex Event erzeugt wurde, wodurch das Fenster schneller abgeschlossen werden kann.

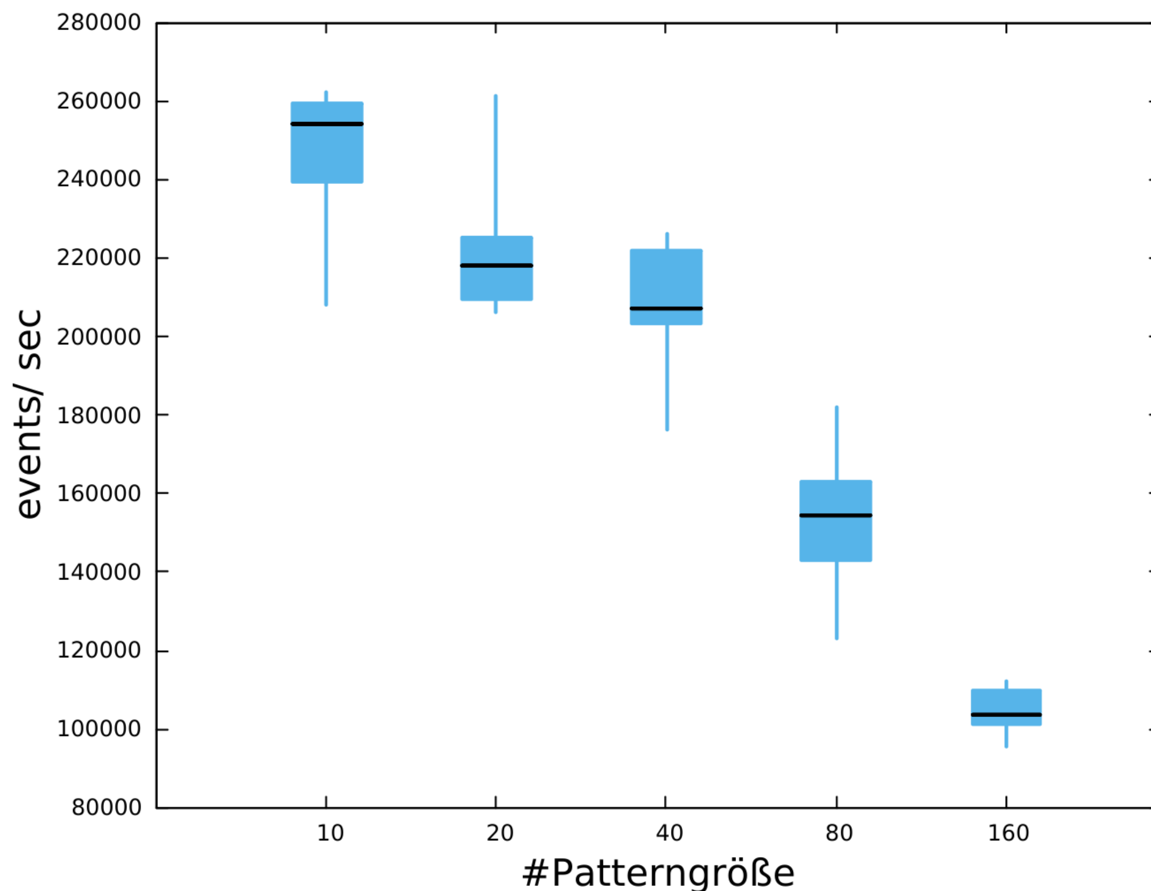

Abbildung 6.4: Events/Sekunde für Patterngrößen von 10-160.

6.2.3 Fenstergröße

Durch das Variieren der Fenstergröße verändert sich die Anzahl der Überlappungen der Fenster. Bei niedriger Fenstergröße sind somit weniger Fenster von einander abhängig. Die Werte der Fenstergröße wurden auf 60-480 Sekunden gesetzt.

Ergebnisse

Ähnlich wie bei der Patterngröße gibt es in Abbildung 6.5 keine Überraschungen. Kleinere Fenster verursachen weniger Abhängigkeiten und kürzere Bearbeitungszeit.

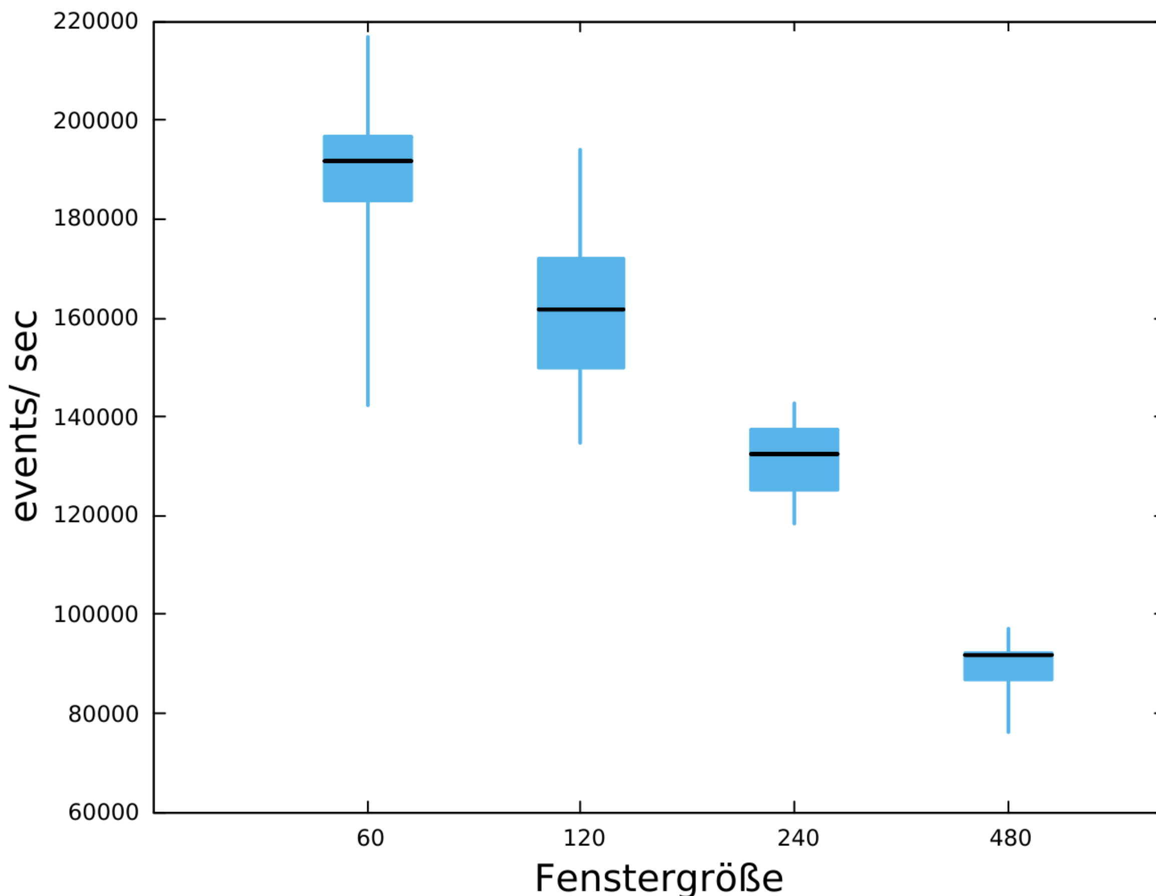

Abbildung 6.5: Events/Sekunde für Fenstergrößen von 60-480s.

6.2.4 Markov Model

Hier zeigt sich der eigentliche Nutzen des Markov Models. Für die Tests wurde das Markov Model deaktiviert und feste Wahrscheinlichkeiten für das Vervollständigen eines Complex Events gesetzt. Die Tests wurden für die Patterngrößen 80, 300 und 500 durchgeführt. Die Vervollständigungswahrscheinlichkeit eines Patters variiert hierdurch deutlich. Zum Vergleich wurde das Experiment auch mit aktiviertem Markov Model durchgeführt. Der Matrixminimierungsfaktor wurde auf 8 gesetzt, um Probleme bei einer Patterngröße von 500 zu vermeiden.

Ergebnisse

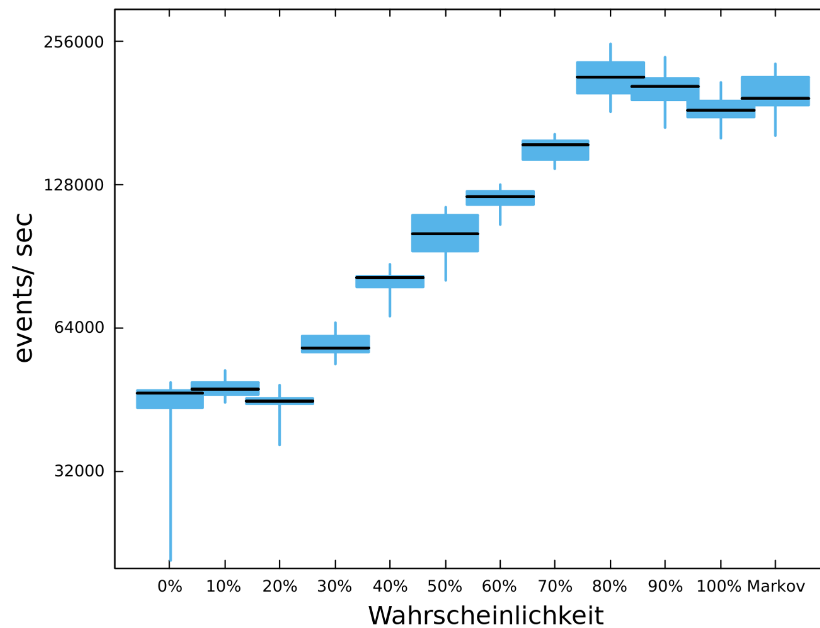

Abbildung 6.6: Events/Sekunde feste Complex Event Wahrscheinlichkeiten, Patterngröße 80

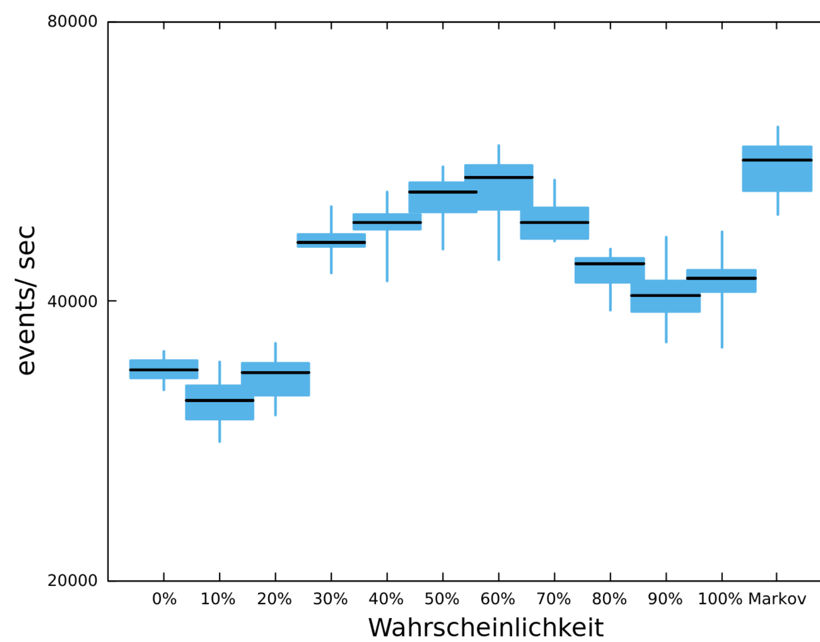

Abbildung 6.7: Events/Sekunde feste Complex Event Wahrscheinlichkeiten, Patterngröße 300

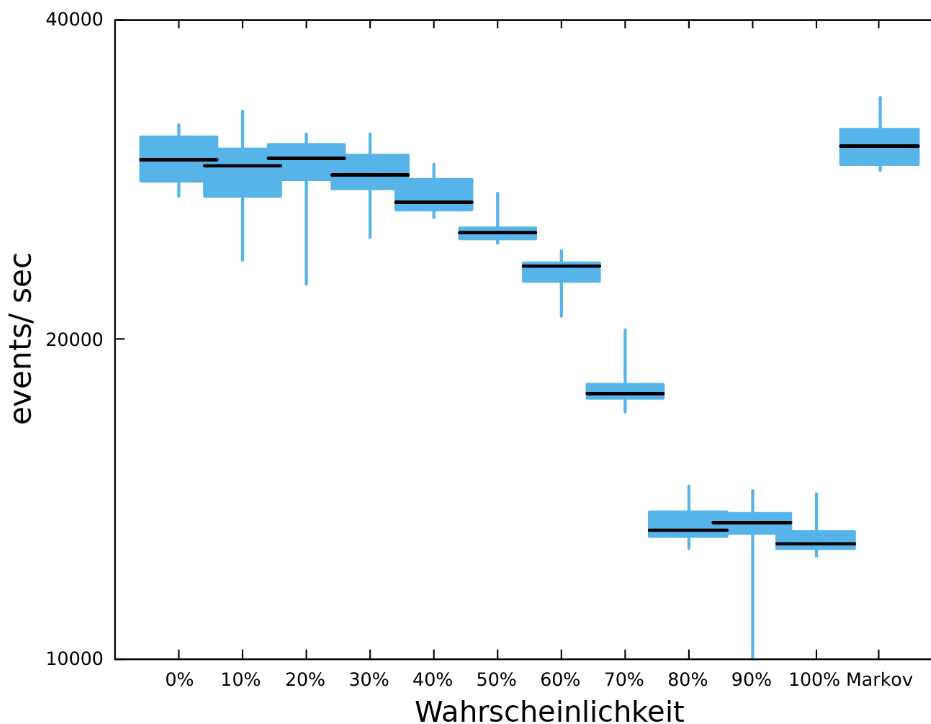

Abbildung 6.8: Events/Sekunde feste Complex Event Wahrscheinlichkeiten, Patterngröße 500

Wie man an den Ergebnissen in den Abbildungen 6.6 - 6.8 gut sehen kann, sind die Resultate der fest vergebenen Wahrscheinlichkeiten für das Vervollständigen eines Complex Events fast immer von Nachteil. Schlechter war das Markov Model nur bei der Patterngröße von 80 wo es knapp von den festen Wahrscheinlichkeiten 80% und 90% geschlagen wurde. Dadurch, dass das Markov Model anpassungsfähig ist, und für jede der gemessenen Patterngrößen sehr gute Ergebnisse erzielt hat, sollte es beim regulären Einsatz durchweg gute Ergebnisse erzielen.

6.2.5 Matrix Minimierungsfaktor

Durch den Matrixminimierungsfaktor wird die Matrix verkleinert. Das sorgt dafür, dass diese einerseits schneller erstellt werden kann, andererseits geht dies auf Kosten der Genauigkeit der Wahrscheinlichkeitswerte. Die Werte des Matrixminimierungsfaktors gehen von 1-8.

Ergebnisse

Der Matrixminimierungsfaktor hat bei diesem Experiment keinen großen Einfluss auf den Eventdurchsatz, wie man an Abbildung 6.9 sieht. Die Ergebnisse für 2 und 4 sind minimal besser als bei 1 wobei 8 am schlechtesten abschneidet. Eine mögliche Erklärung ist, dass das schnellere

Generieren der Matrix für dieses Experiment eine höhere Wichtigkeit hat als genauere Werte, solange die Vereinfachung die Wahrscheinlichkeiten nicht zu sehr verfälscht.

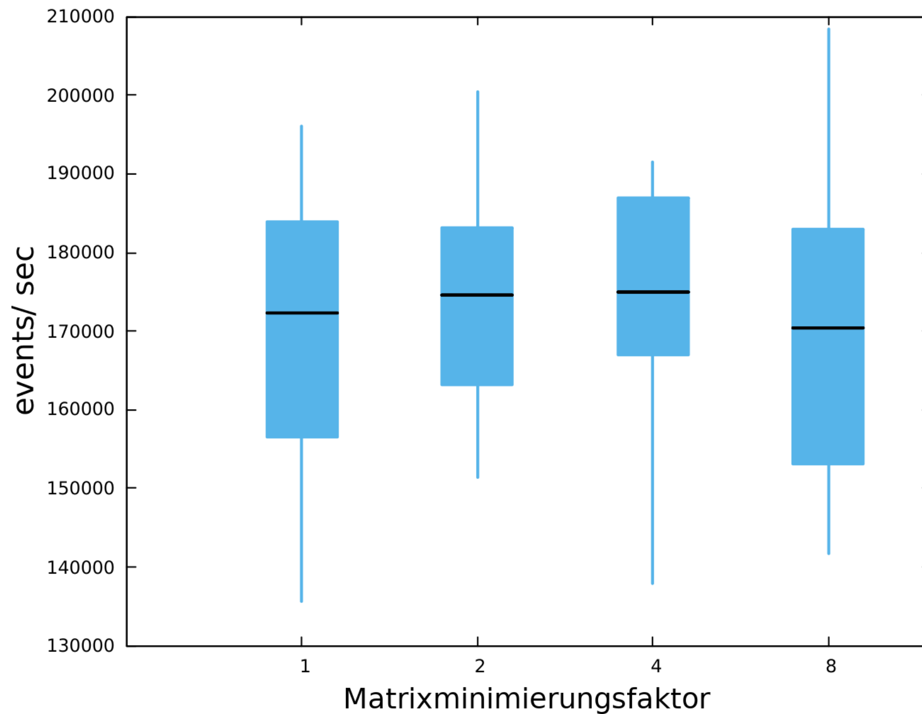

Abbildung 6.9: Events/Sekunde für Matrinminimierungsfaktor 1-8

6.2.6 Checkpointing

Durch Checkpoints wird verhindert, dass eine Fensterversion bei einer Fehlspekulation nicht alle bisher verrichtete Arbeit verwirft. Sie stattdessen an zu einem Zustand zurückkehren, bei dem die Fehlspekulation noch nicht passiert ist. Wie sich aber herausstellte sind die Kosten Checkpoints zu erstellen oftmals Größer als ihr Nutzen. Gerade bei sehr kleinen Events ist es oft schneller zum Ausgangszustand zurückzukehren und das Fenster erneut zu bearbeiten. Bei großen Events hingegen kann es durchaus zu einer Verbesserung der Performance führen.

Für dieses Experiment wurde der Operator so implementiert, dass er für das bearbeiten eines Events, das Teil eines Partial Matches ist, eine Sekunde zusätzlich rechnet. Insgesamt werden nur 2000 Events generiert, die Fenstergröße ist 100, das zu erkennende Pattern ist ABCDEF. Die generierten Events sind A, B, C, D, E, F, G und H. Es wird nur ein Complex Event pro Fenster Generiert und der Splitter öffnet bei jedem A ein neues Fenster. Die Parameter für das Markov Model sind auf Abstand der Potenzen = 2, Anzahl Berechneter Matrixpotenzen = 50, Matrixminimierungsfaktor = 1 gesetzt.

Ergebnisse

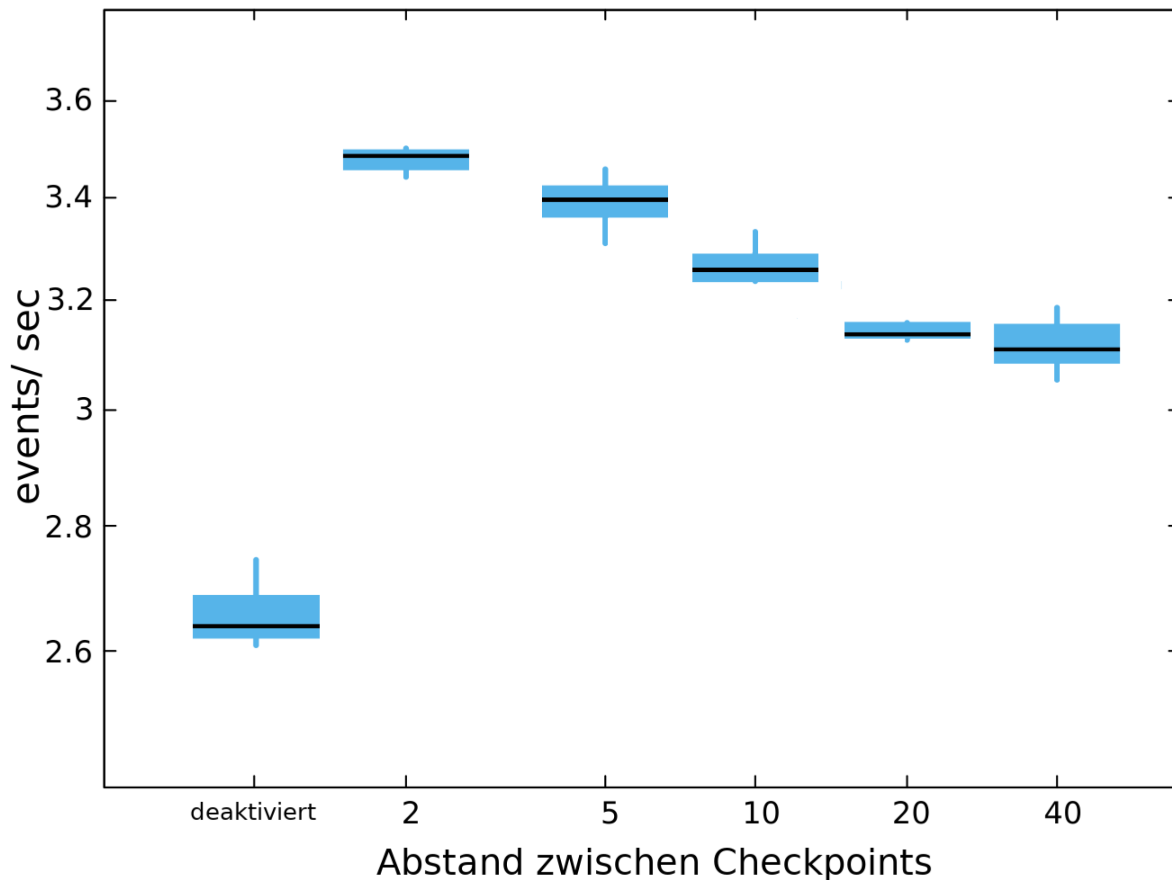

Abbildung 6.10: Events/Sekunde für unterschiedliche Checkponteinstellungen.

Für dieses Szenario hat das Nutzen von Checkpoints eine um ca. 30% verbesserte Leistung erzielen, was Abbildung 6.10 entnehmbar ist.

6.2.7 Abhängigkeitsbaum Tiefe

Die Limitierung des Abhängigkeitsbaums kann einen Einfluss auf die Performance des Systems haben. Für dieses Experiment ist die Anzahl der Worker auf 1-16 gesetzt ebenso wie die Abhängigkeitsbaum Tiefe.

Ergebnisse

Abbildung 6.11 zeigt die Ergebnisse. Die Limitierung des Abhängigkeitsbaumes führt wie erwartet dazu, dass die Performance nachlässt, wenn die Anzahl der arbeitenden Worker größer

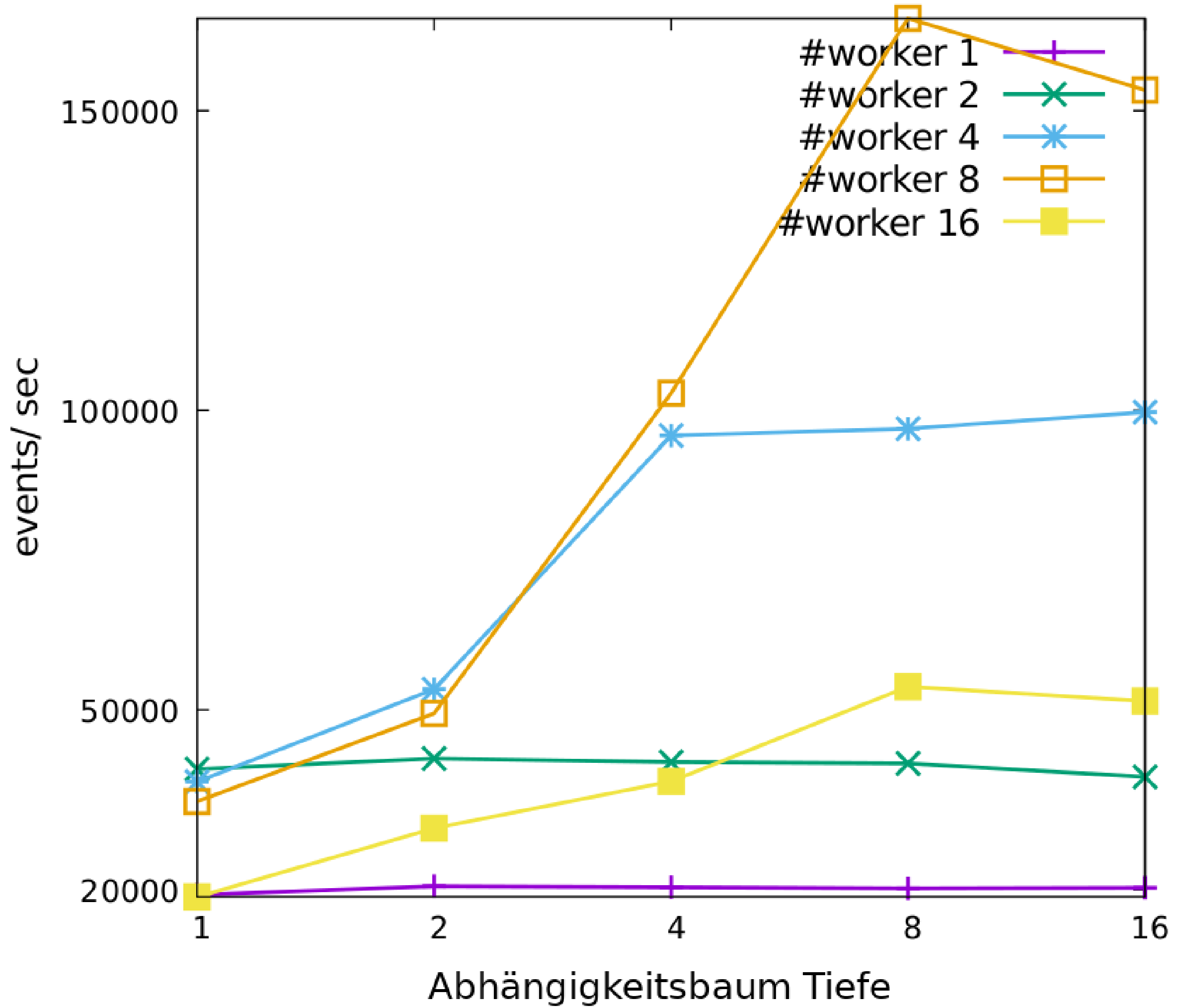

Abbildung 6.11: Events/Sekunde verschiedene Anzahl an Workerthreads und Abhängigkeitsbaum Tiefen.

ist, als die Anzahl der verschiedenen Fenster. Zudem kann ein zu großer Baum durchaus die Leistung des Systems reduzieren, wie man anhand des Performanceabfalls bei 8 Workerthreads sehen kann.

6.2.8 Kein Konsumieren

Nicht immer wird gefordert, dass Events konsumiert werden. Der folgende Test zeigt die Leistungsfähigkeit des Systems mit 1-16 Workerthreads wenn auf das Konsumieren verzichtet wird.

Ergebnisse

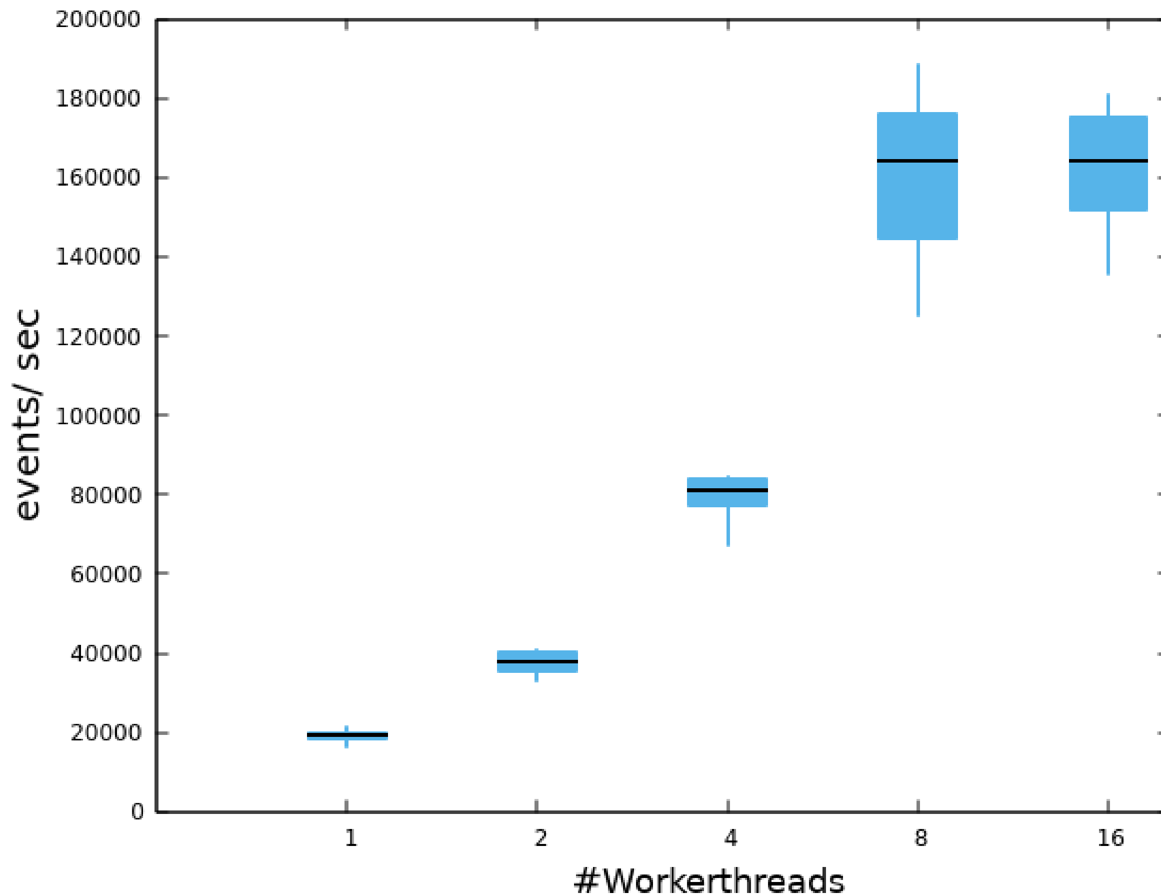

Abbildung 6.12: Events/Sekunde für 1-16 Workerthreads ohne Konsumierung.

Wenn auf das Konsumieren von Events verzichtet wird, wird kein spekulatives Model verwendet und alle Fenster können ohne Berücksichtigung der anderen Fenster bearbeitet werden. Erkennbar ist anhand von Abbildung 6.12 allerdings auch, dass das Verwenden von 16 Workerthreads hier keinen signifikanten Einfluss auf die Leistung des Systems, im Vergleich zu 8 Workerthreads, erwirkt.

6.2.9 T-Rex

Um die Resultate zu vergleichen, folgen die Testergebnisse von T-Rex. Das Problem ist allerdings, dass T-Rex bei einer Patterngröße von mehr als 5 wegen eines Buffer Overflow abstürzt. Aus diesem Grund kann hier nur das Ergebnis mit einer Patterngröße von 5 gezeigt werden. Als variabler Parameter wird die Fenstergröße von 60-480 Sekunden gewählt.

Ergebnisse

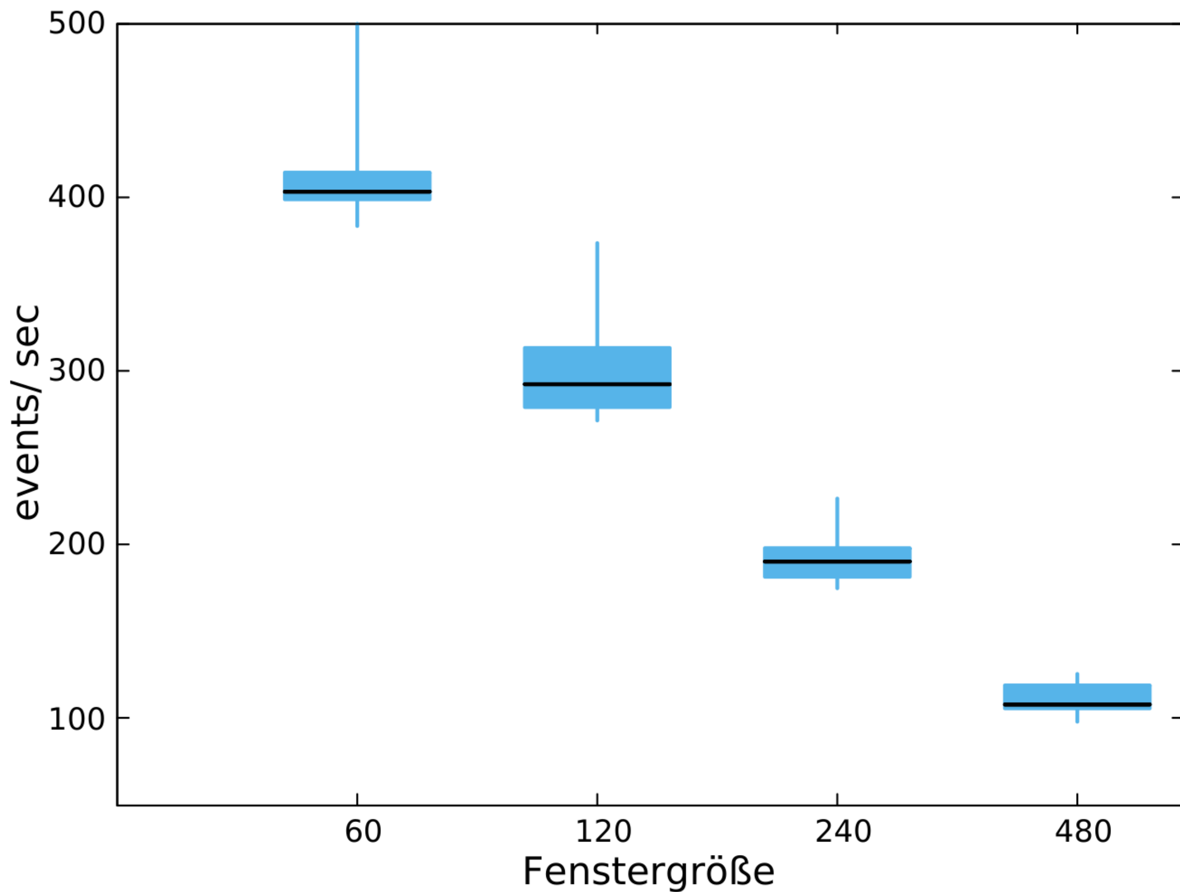

Abbildung 6.13: Events/Sekunde für T-Rex bei Fenstergrößen 60-480s.

Abbildung 6.13 zeigt die Resultate für verschiedene Fenstergrößen bei der Verwendung von T-Rex. Auch hier haben größere Fenster einen negativen Einfluss auf die Performance. Auffällig ist allerdings die deutlich niedrigere Leistung im Gegensatz zu SPECTRE das bei einer Fenstergröße von z.B. 120Sec ca. 250000 Events/Sec - bei einer Patterngröße von 10 - erzielt. Das ist etwa das 860-fache des T-Rex-Systems.

6.3 Zusammenfassung

Mit SPECTRE ist es möglich, bis zu 8 CPUs gleichzeitig arbeiten zu lassen, um Complex Events zu erkennen. Das Verwenden von mehr CPUs ist bisher nicht sinnvoll. Durch das Markov Model sind Spekulationen möglich und haben eine gute Qualität.

Das System erzielt mit den Standardparametern einen Durchsatz von ca. 160 000 Events/Sec. Mit T-Rex hingegen ist es nicht möglich eine Patterngröße von über 5 zu verwenden. Und selbst

bei dieser Patterngröße bringt SPECTRE bei der selben Fenstergröße und längeren Pattern einen etwa 860-fachen Leistungsgewinn.

7 Verwandte Arbeiten

Hier wird auf verwandte Arbeiten eingegangen. Als erstes wird auf T-Rex eingegangen. Es folgt das Verarbeiten von einzelnen Transaktionen durch Speculative Out-Of-Order Event Processing. Anschließend wird in Adaptive Speculative Processing of Out-of-Order Event Streams auf das spekulative Vorsortieren von Events eingegangen. Darauf folgt ein anderer Ansatz zur Verarbeitung von Eventstreams mit GraphCEP. Das Kapitel wird durch eine kurze Zusammenfassung der verschiedenen Ansätze abgeschlossen.

7.1 T-Rex

T-Rex ist ein CEP System das als Spezifikationssprache TESLA verwendet und Eventkonsumierung unterstützt. Allerdings verursacht das Konsumieren von Events eine deutlich geringere Leistung.

7.1.1 Architektur

Die Architektur von T-Rex wird in Abbildung 7.1 veranschaulicht. T-Rex muss die TESLA Event Spezifikationen parsen. Der Rule Manager erkennt statische und dynamische Bedingungen. Eine statische Bedingung ist unabhängig von anderen Events wie z.B. Temperatur > 45. Diese statischen Regeln werden an den Static Index weitergegeben und können schnell ausgewertet werden. Die Regeln werden als Automaton Model gespeichert.

Sobald ein Event das System betritt entscheidet der Static Index für welche Automaton Models es relevant sein kann und gibt es an die entsprechenden weiter. Falls es zu keinem passt, wird es verworfen.

Um den Speicherverbrauch zu minimieren werden Events in der Stored Events Datenstruktur gespeichert und nur mit Referenzen gearbeitet. So wird für ein Event nicht mehrfach Speicher verbraucht, wenn es für mehrere Regeln benötigt wird.

In Sequences wird der aktuelle Zustand gespeichert. Sobald alle Anforderungen für ein Complex Event erfüllt sind, werden die nötigen Informationen an den Generator weitergegeben. Dieser baut daraus ein Complex Event und gibt dieses an die entsprechenden Clients weiter [CM12a].

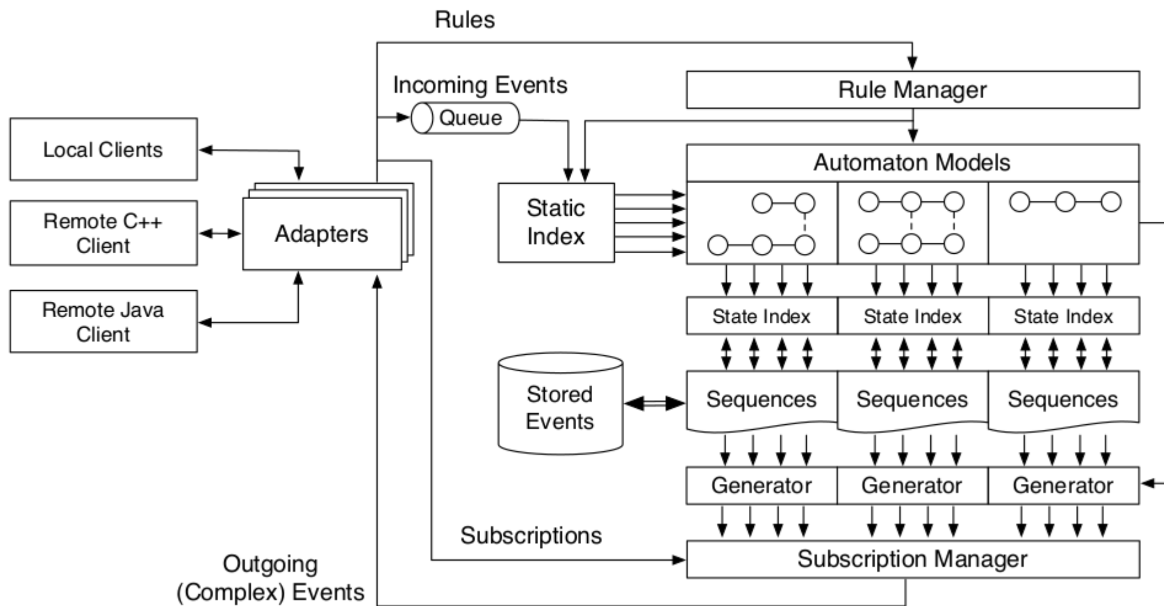

Abbildung 7.1: Die Architektur von T-Rex [CM12a]

7.2 Speculative Out-Of-Order Event Processing

Dadurch, dass Events generiert und an das System geschickt werden, kommt es vor, dass sie nicht in der richtigen Reigenfolge im System ankommen. In dieser Arbeit wird davon ausgegangen, dass ein Event eine Transaktion verursacht. Abbildung 7.3 zeigt den Programmaufbau. Durch die falsche Reihenfolge kann es bei der Verarbeitung zu Problemen kommen, allerdings nur, falls dadurch Konflikte entstehen. Das System bearbeitet die Events in der Reihenfolge, in der sie ankommen und prüft, bevor diese das System verlassen, ob es zu einem Konflikt gekommen ist. Sollte das der Fall sein, muss diese Operation mit den richtigen Werten neu durchgeführt werden [Bri+08].

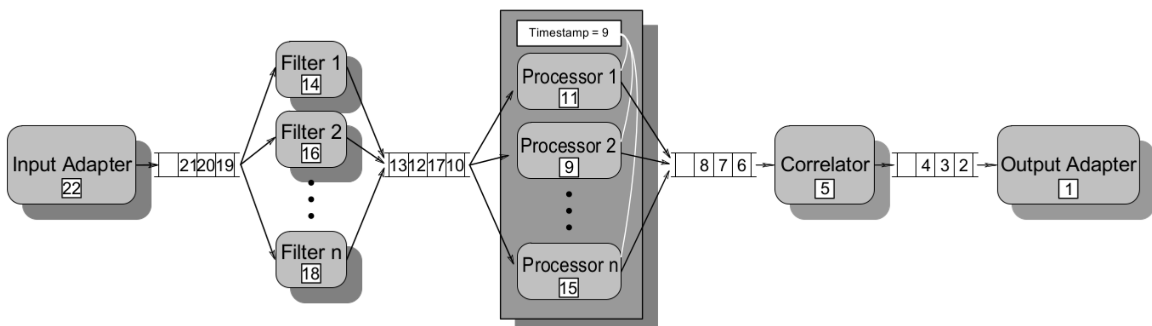

Abbildung 7.2: Darstellung für Speculative Out-Of-Order Event Processing [Bri+08]

7.3 Adaptive Speculative Processing of Out-of-Order Event Streams

Ähnlich wie bei Speculative Out-Of-Order Event Processng wird auch bei dieser Arbeit davon ausgegangen, dass die Event in falscher Reihenfollge im System ankommen können. Der Systemaufbau wird in Abbildung 7.3 dargestellt. Adaptive Speculative Processing of Out-of-Order Event Streams geht das Problem durch Event Buffering an. Die Events werden eine Zeit gebuffert und anschließend sortiert an das System gegeben. Um die Verzögerung minimal zu halten, passt das System die Buffergröße spekulativ zur Laufzeit an. Sollte eine Fehlspekulation eintreten, also der Buffer zu klein gewählt worden sein, wird mithilfe von Checkpoints das System in einen korrekten Zustand zurück gesetzt. [MP14]

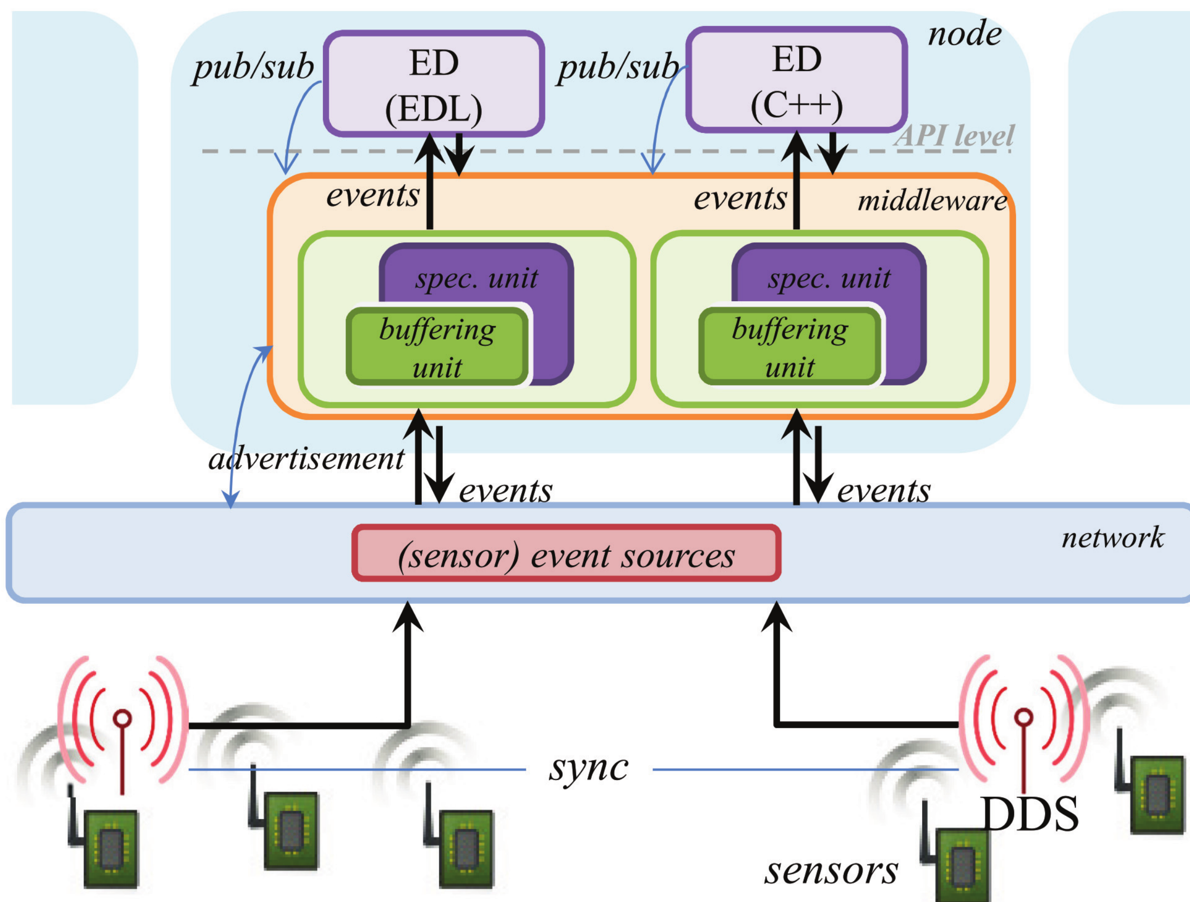

Abbildung 7.3: Darstellung des Adaptive Speculative Processing of Out-of-Order Event Streams Systems [MP14].

7.4 GraphCEP

Mit GraphCEP wird versucht die Stärken der Graphenverarbeitung mit parallelem CEP zu verbinden. Der Vorteil einer Graphstruktur ist, dass sie gut parallel verarbeitet werden kann. Die Architektur wird in Abbildung 7.4 dargestellt. Die Events werden vom Splitter an die verschiedenen Operatoren weitergegeben. Diese werden von den Operatoren als Graph organisiert. Dadurch, dass eine Operatorinstanz parallel arbeiten kann, wird ein sehr hoher Grad an Parallelität erreicht, was die Ausführgeschwindigkeit deutlich anhebt. Zudem ist das System skalierbar, da nicht nur die Anzahl der Operatorinstanzen sondern auch deren Threadanzahl wachsen kann [May+16].

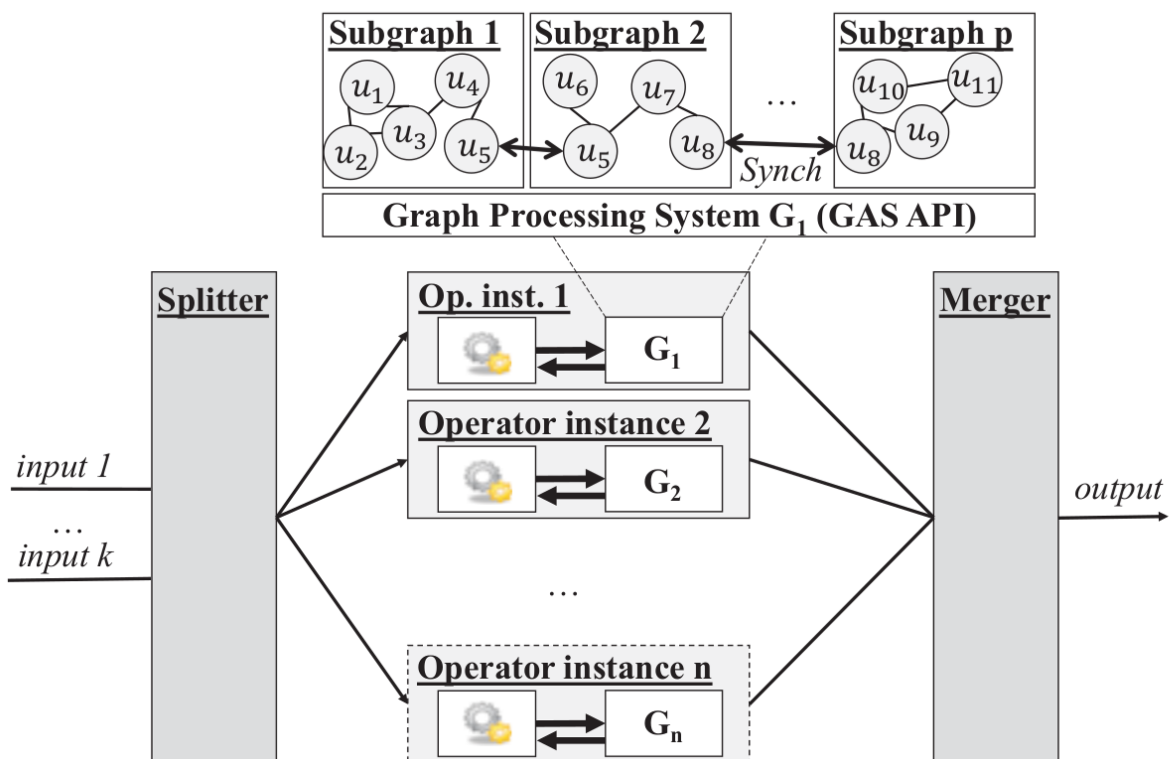

Abbildung 7.4: Architektur des GraphCEP Systems [May+16]

7.5 Zusammenfassung

Die hier beschriebenen Arbeiten verwenden verschiedene Methoden, um Events zu schnell wie möglich verarbeiten zu können. T-Rex hat für die verschiedenen Complex Events je einen Operator, der nichts anderes bearbeitet.

Speculative Out-Of-Order Event Processing geht von einem Eventstream aus, der nicht zwangsläufig geordnet sein muss, geht aber erst einmal davon aus, dass es bei einer Verarbeitung in

falscher Reihenfolge keine Konflikte gibt. Bei einem Konflikt wird das entsprechende Event neu bearbeitet.

In Adaptive Speculative Processing of Out-of-Order Event Streams wird ebenfalls von einem ungeordneten Eventstream ausgegangen. Hier wird ein Buffer verwendet, um die Events zu sortieren. Die Größe dieses Buffers wird bei der Laufzeit Spekulativ angepasst.

Bei GraphCEP werden die Events als Graph organisiert, um eine sehr effiziente parallele Verarbeitung zu ermöglichen.

8 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Einblick in Complex Event Processing gegeben. Das Problem, dass aktuelle CEP-Systeme keine Parallelisierung der Operatorinstanzen erlauben, solange Event Consumption benötigt wird, wurde erläutert.

Es wurde auf die Event Specification Languages Snoop und TESLA eingegangen und deren Eigenheiten beschrieben.

Des Weiteren wurde das SPECTRE Framework ausführlich dargestellt und die Funktionsweise beschrieben. Es war schon in der Lage, Abhängigkeiten zu erkennen und zu berücksichtigen, allerdings fehlte ein Model, das die Wahrscheinlichkeit der einzelnen Complex Events bestimmen kann. Ebenso gab es noch keinen Algorithmus, um die wahrscheinlichsten Fensterversionen zu ermitteln.

Durch das spekulative Model auf Basis von Markov Ketten, war es nun möglich, die Vervollständigungswahrscheinlichkeiten der einzelnen Consumption Groups zu berechnen. Das Model hat anhand von Statistiken einen Zustandsgraphen mit Übergangswahrscheinlichkeiten generiert. Anhand von diesem wurden die Wahrscheinlichkeiten für festgelegte Restfensterlängen vorberechnet. Anhand von Interpolation wurde der passende Wert ermittelt.

Die Evaluation zeigte gute Ergebnisse für bis zu 8 Workerthreads. Das Verwenden von 16 Workerthreads hat allerdings zu einem Einbruch der Performance geführt. Für die Standardkonfiguration hat SPECTRE etwa 160 000 Events/sec Verarbeiten können. Eine größere Fenster- oder Patterngröße führen zu mehr Abhängigkeiten und reduzieren somit die Performance.

Die Evaluation hat ebenso ergeben, dass das Verwenden des Markov Models in der Regel deutlich bessere Leistungen erzielt, als fest vorgegebene Wahrscheinlichkeiten.

T-Rex konnte leider nur Patterns bis zu einer Größe von 5 Events verarbeiten und hatte selbst dort eine deutlich geringere Leistung.

Abschließend lässt sich sagen, dass durch das Markov Model gute Spekulationen möglich sind und das beschriebene Problem dadurch gelöst wurde.

Ausblick

Die Testergebnisse von SPECTRE sind größtenteils positiv. Allerdings wird in Zukunft weiter daran gearbeitet werden die Skalierbarkeit weiter zu erhöhen.

Auch das Markov Model kann an einigen Stellen noch verbessert werden. Es wird überlegt, ein Müll-Zustand hinzuzufügen, der verwendet werden kann, falls es nicht mehr möglich ist, ein Complexes Event zu vervollständigen. Des Weiteren wird bisher nur eine Zustandsübergangsmatrix und deren Potenzen verwendet. Wenn das System verschiedene Complexe Events erkennen soll, die eine deutlich andere Wahrscheinlichkeit besitzen, kann die Qualität der zurückgegebenen Wahrscheinlichkeit leiden. Es ist denkbar, dass für diesen Fall zukünftig mehrere solcher Modelle gleichzeitig im System laufen und somit jede Art von Complex Event ihr eigenes Markov Model hat.

Bisher benötigt SPECTRE eine Leistungsstarke Plattform mit vielen CPUs. Zukünftig wird versucht das System verteilt zu organisieren, damit der Workload auf mehrere leistungsschwächere Plattformen verteilt werden kann.

Literaturverzeichnis

[AE04] A. Adi, O. Etzion. „Amit - the Situation Manager“. In: *The VLDB Journal* 13.2 (Mai 2004), S. 177–203. ISSN: 1066-8888. DOI: 10.1007/s00778-003-0108-y. URL: http://dx.doi.org/10.1007/s00778-003-0108-y (zitiert auf S. 24).

[Boo] *Boost C++ Libraries*. URL: http://boost.org (zitiert auf S. 53).

[Bri+08] A. Brito, C. Fetzer, H. Sturzrehm, P. Felber. „Speculative Out-of-order Event Processing with Software Transaction Memory“. In: *Proceedings of the Second International Conference on Distributed Event-based Systems*. DEBS ’08. Rome, Italy: ACM, 2008, S. 265–275. ISBN: 978-1-60558-090-6. DOI: 10.1145/1385989.1386023. URL: http://doi.acm.org/10.1145/1385989.1386023 (zitiert auf S. 70).

[CM10] G. Cugola, A. Margara. „TESLA: A Formally Defined Event Specification Language“. In: *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*. DEBS ’10. Cambridge, United Kingdom: ACM, 2010, S. 50–61. ISBN: 978-1-60558-927-5. DOI: 10.1145/1827418.1827427. URL: http://doi.acm.org/10.1145/1827418.1827427 (zitiert auf S. 24).

[CM12a] G. Cugola, A. Margara. „Complex Event Processing with T-REX“. In: *J. Syst. Softw.* 85.8 (Aug. 2012), S. 1709–1728. ISSN: 0164-1212. DOI: 10.1016/j.jss.2012.03.056. URL: http://dx.doi.org/10.1016/j.jss.2012.03.056 (zitiert auf S. 15, 24, 69, 70).

[CM12b] G. Cugola, A. Margara. „Low Latency Complex Event Processing on Parallel Hardware“. In: *J. Parallel Distrib. Comput.* 72.2 (Feb. 2012), S. 205–218. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2011.11.002. URL: http://dx.doi.org/10.1016/j.jpdc.2011.11.002 (zitiert auf S. 16, 19, 21).

[CM12c] G. Cugola, A. Margara. „Processing Flows of Information: From Data Stream to Complex Event Processing“. In: *ACM Comput. Surv.* 44.3 (Juni 2012), 15:1–15:62. ISSN: 0360-0300. DOI: 10.1145/2187671.2187677. URL: http://doi.acm.org/10.1145/2187671.2187677 (zitiert auf S. 21, 22).

[CM94] S. Chakravarthy, D. Mishra. „Snoop: An Expressive Event Specification Language for Active Databases“. In: *Data Knowl. Eng.* 14.1 (Nov. 1994), S. 1–26. ISSN: 0169-023X. DOI: 10.1016/0169-023X(94)90006-X. URL: http://dx.doi.org/10.1016/0169-023X(94)90006-X (zitiert auf S. 23).

[GJ+10] G. Guennebaud, B. Jacob et al. *Eigen v3*. 2010. URL: http://eigen.tuxfamily.org (zitiert auf S. 53).

[Gyl+08] D. Gyllstrom, J. Agrawal, Y. Diao, N. Immerman. „On Supporting Kleene Closure over Event Streams“. In: *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*. ICDE ’08. Washington, DC, USA: IEEE Computer Society, 2008, S. 1391–1393. ISBN: 978-1-4244-1836-7. DOI: 10.1109/ICDE.2008.4497566. URL: http://dx.doi.org/10.1109/ICDE.2008.4497566 (zitiert auf S. 24).

[KUM15] F. Köster, M. W. Ulmer, D. C. Mattfeld. „Cooperative Traffic Control Management for City Logistic Routing“. In: *Transportation Research Procedia* 10 (2015), S. 673 –682. ISSN: 2352-1465. DOI: http://dx.doi.org/10.1016/j.trpro.2015.09.021. URL: http://www.sciencedirect.com/science/article/pii/S2352146515002082 (zitiert auf S. 15).

[May+16] R. Mayer, C. Mayer, M. A. Tariq, K. Rothermel. „GraphCEP: Real-time Data Analytics Using Parallel Complex Event and Graph Processing“. In: *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. DEBS ’16. Irvine, California: ACM, 2016, S. 309–316. ISBN: 978-1-4503-4021-2. DOI: 10.1145/2933267.2933509. URL: http://doi.acm.org/10.1145/2933267.2933509 (zitiert auf S. 15, 72).

[ME09] F. B. Michael Eckert. „Complex Event Processing (CEP)“. In: (2009). URL: http://www.en.pms.ifi.lmu.de/publications/PMS-FB/PMS-FB-2009-5/PMS-FB-2009-5-paper.pdf (zitiert auf S. 15).

[MKR15] R. Mayer, B. Koldehofe, K. Rothermel. „Predictable Low-Latency Event Detection With Parallel Complex Event Processing“. In: *IEEE Internet of Things Journal* 2.4 (2015), S. 274–286. ISSN: 2327-4662. DOI: 10.1109/JIOT.2015.2397316 (zitiert auf S. 16).

[MP14] C. Mutschler, M. Philippsen. „Adaptive Speculative Processing of Out-of-Order Event Streams“. In: *ACM Trans. Internet Technol.* 14.1 (Aug. 2014), 4:1–4:24. ISSN: 1533-5399. DOI: 10.1145/2633686. URL: http://doi.acm.org/10.1145/2633686 (zitiert auf S. 71).

[MTR16] R. Mayer, M. A. Tariq, K. Rothermel. „Real-Time Batch Scheduling in Data-Parallel Complex Event Processing“. In: (2016). ISSN: 2327-4662. DOI: 10.1145/1235. URL: https://scholar.google.de/citations?view_op=view_citation&hl=de&user=ex-eO0UAAAAJ&citation_for_view=ex-eO0UAAAAJ:a0OBvERweLwC (zitiert auf S. 20).

[Oef] *COMPLEX EVENT PROCESSING TECHNOLOGIEN*. URL: https://www.oeffentliche-it.de/trendsonar (zitiert auf S. 15).

[Pan03] U. Pantle. *Markov-Ketten*. Sep. 2003. URL: http://www.mathematik.uni-ulm.de/stochastik/lehre/ss03/markov/skript/node3.html (zitiert auf S. 43).

[See10] B. Seeger. *Kontinuierliche Kontrolle*. 2010. URL: https://www.heise.de/ix/artikel/Kontinuierliche-Kontrolle-905334.html (zitiert auf S. 15).

[SG] P. M. Sanjay Ghemawat. *TCMalloc : Thread-Caching Malloc*. URL: http://goog-perftools.sourceforge.net/doc/tcmalloc.html (zitiert auf S. 51, 53).

[Wei] E. W. Weisstein. *Strassen Formulas.* URL: http://mathworld.wolfram.com/StrassenFormulas.html (zitiert auf S. 47).

[ZU99] D. Zimmer, R. Unland. „On the semantics of complex events in active database management systems“. In: *Proceedings 15th International Conference on Data Engineering (Cat. No.99CB36337).* 1999, S. 392–399. DOI: 10.1109/ICDE.1999.754955 (zitiert auf S. 19, 20).

Alle URLs wurden zuletzt am 6. 04. 2017 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift