Data Stream Processing

Johannes Götz

Database and Information Systems
University of Kaiserslautern

j_goetz11@cs.uni-kl.de

Abstract. Die folgende wissenschaftliche Arbeit behandelt die Grundlagen des Data Stream Processing (DSP) sowie dessen Einsatz.

Der erste Teil der Arbeit beschäftigt sich mit den Grundlagen des DSP. Dabei werden zunächst Data-Stream-Management-Systeme (DSMS) von den bereits bekannten Datenbankmangementsystemen abgegrenzt. Im weiteren Verlauf wird kurz eine mögliche Anfragesprache für ein DSMS, in unserem Falle Continuous Query Language (CQL) am Beispiel des Stanford Stream Data Manager (STREAM) der Stanford-Universität [11] unter Zuhilfenahme einiger Beispiele präsentiert.

Im zweiten Teil wird die Funktionsweise eines DSMS im Inneren näher untersucht. So wird zunächst auf die Behandlung eines Query Plans sowie seine Besonderheiten in einem DSMS eingegangen. Daraufhin wird das Themengebiet der Approximation bearbeitet.

Danach wird als anderer Ansatz für ein DSMS Aurora vorgestellt, auf dessen Besonderheiten eingegangen und die Unterschiede zwischen Aurora und STREAM aufgezeigt.

Anschließend wird ein Fazit gezogen und noch kurz auf die unterschiedlichen Einsatzgebiete von Aurora und Stream eingegangen.

1 Einleitung

Data Stream Processing gewinnt aufgrund immer größer werdender Datenmengen und Datenraten zunehmend an Bedeutung. Es ist mittlerweile nicht mehr möglich, alle Daten zu speichern und dann zu bearbeiten, allein schon wegen ihrer gewaltigen Menge. Außerdem würde man mit diesem Ansatz theoretisch ewig auf ein Ergebnis warten, wenn beständig neue Daten ankommen. Um diesem Umstand gerecht zu werden, gibt es neben den traditionellen Datenbank-Management-Systemen (DBMS) extra für die Herangehensweise des Data Stream Processing ausgelegte Management-Systeme, die sogenannten Data Stream Management Systems (DSMS). Auf die Unterschiede zwischen einem DBMS und einem DSMS wird später noch einmal im Detail eingegangen.

Als Motivation, sich mit Data Stream Processing zu beschäftigen, dienen neben der bereits erwähnten stetig wachsenden Bedeutung auch noch folgende Anwendungsfälle, in denen Streams erzeugt werden:

- Auslesen von Sensordaten
- Beobachtung des Straßenverkehrs
  - zur Stauwarnung bzw. Verhinderung
  - zur Berechnung der genauen Fahrtdauer
  - zur Mautüberprüfung
  - zur Geschwindigkeitsüberwachung
- Umweltbeobachtung
  - zur Sturmwarnung
  - zur Klimaüberwachung
  - zur Luftqualitätsüberwachung
  - zur Optimierung von landwirtschaftlichen Prozessen
- Aufzeichnungen von Telefongesprächen
- Logaufzeichnungen von Webservern
- Analyse von Finanzströmen

Auch eine Studie über die immer weiter steigende Datenmenge im Internet, die durch die folgende Abbildung (Abbildung 1) illustriert wird, bestätigt uns die Notwendigkeit von Data Stream Processing. Die Abbildung zeigt die Entwicklung des verfügbaren Speicherplatzes und die Menge der entstehenden Informationen weltweit. Wie man sieht, übersteigt die Menge der Informationen die des Speicherplatzes gewaltig. Es wird also eine Technik benötigt, um aus der Informationsmenge nur die wichtigen Informationen herauszufiltern, da für mehr nicht genügend Speicherplatz vorhanden ist. Diese Technik ist das Data Stream Processing.

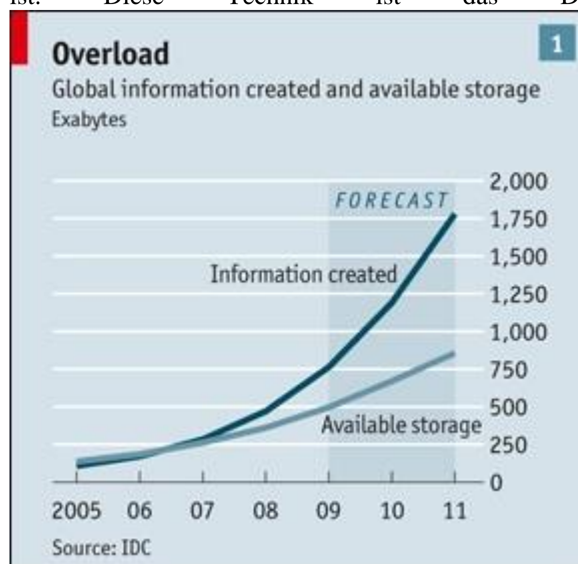


Abbildung (1), „Study Projects Nearly 45-Fold Annual Data Growth by 2020” EMC Press Release

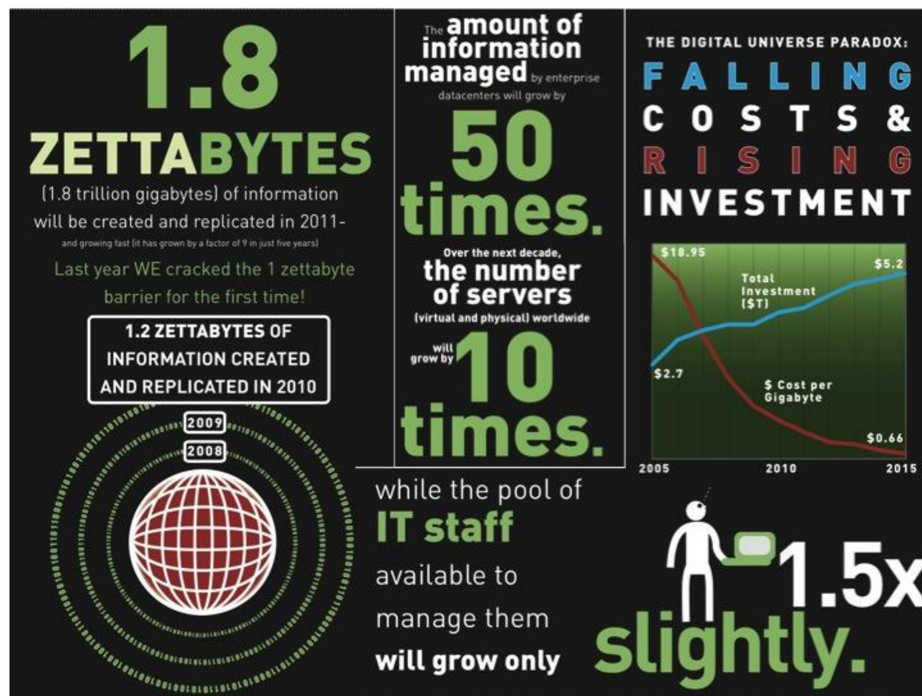


Abbildung (2), IDC 2011 Digital Universe Study.

Im Verlaufe der Arbeit wird als Beispiel für ein DSMS „The Stanford Data Stream Management System“ (http://infolab.stanford.edu/stream/), kurz STREAM, verwendet. STREAM wurde als Teil des gleichnamigen Forschungsprojekts an der Stanford University entwickelt und ist für Interessierte im Quelltext erhältlich (http://infolab.stanford.edu/stream/code/). Außerdem ist ein Testsystem zum Ausprobieren von STREAM unter folgendem Link erreichbar (http://skate.stanford.edu:8080/).

Das Kapitel 2 zeigt im Detail die Unterschiede zwischen einem DSMS und einem DBMS auf und fasst diese am Ende noch einmal tabellarisch zusammen.

Kapitel 3 behandelt die Continuous Query Language. Ich gehe kurz darauf ein, aus welchem Antrieb die CQL entwickelt wurde, und erläutere dann ihre Grundlagen anhand der Beispiele in Abschnitt 3.1.

In Kapitel 4 wird auf den Query Plan eingegangen, erst auf verschiedene Ansätze im Allgemeinen, danach spezifisch auf die Herangehensweise in STREAM. An-

schließend werden die verschiedenen dafür verfügbaren Operatoren aufgelistet und ein Beispiel für einen Query Plan gegeben. Abschnitt 4.2 gibt einen Einblick in die Optimierungsmöglichkeiten eines Query Plan.

In Kapitel 5 untersuche ich die Gründe für die Nutzung von Annäherungsberechnungen in STREAM. Dabei behandelt Abschnitt 5.1.1 die CPU-limitierte Annäherung und 5.1.2 die Speicher-limitierte Annäherung. Abschnitt 5.2 geht auf die Methoden zur statischen Annäherung ein, in 5.2.1 auf die Fensterverkleinerung und in 5.2.2 auf die Sampling-Verkleinerung, während Abschnitt 5.3 sich mit der dynamischen Annäherung befasst: mit der Synopsis Compression in 5.3.1 und Sampling and Load Shedding in 5.3.2.

Kapitel 6 thematisiert das DSMS Aurora[10]. Abschnitt 6.1 zählt die in Aurora verwendeten Operatoren auf, erläutert diese und vergleicht abschließend die Query Language mit der von STREAM. Abschnitt 6.2 geht auf den in Aurora verwendeten Query Plan und dessen Unterschiede zu dem in STREAM verwendeten ein. 6.3 befasst sich mit der Anfrageoptimierungen in Aurora, 6.3.1 mit den dynamischen und 6.3.2 mit den Ad-hoc-Anfrageoptimierungen. In Abschnitt 6.4 werden die Dienstgüte-Datenstrukturen erläutert, in 6.5 die Datenspeicherung mithilfe des Aurora-Speichermanagers – unter anderem das Queue-Management in 6.5.1 und das Verhalten des ASM an Verbindungspunkten in 6.5.2. Abschnitt 6.6 ist der Ablaufplanung gewidmet: 6.6.1 der Zug-Ablaufplanung, 6.6.2 der Prioritätszuweisung und 6.6.3 auf die Ablaufplanung speziell in Aurora.

2 DBMS vs. DSMS

Im folgenden Abschnitt werden zunächst die Unterschiede zwischen einem Datenbank-Management-System (DBMS) und einem Data-Stream-Management-System (DSMS) aufgezeigt.

Während ein DBMS auf Grundlage von persistent gespeicherten Relationen und ihrer Tupel arbeitet, stellen bei einem DSMS, hinzukommend zu den gespeicherten Relationen, auch noch transiente Streams eine Datenquelle dar.

Einen weiteren Unterschied findet man im Datenzugriff. Während im DBMS ein sogenannter wahlfreier Zugriff stattfindet, also beliebig auf alle Elemente zugegriffen werden kann, kann man in einem DSMS auf die Elemente eines Streams nur sequentiell zugreifen.

Auch bezüglich der Anfragen und ihrer Häufigkeit lassen sich Unterschiede erkennen. In einem DBMS wird eine Anfrage einmalig ausgeführt, in einem DSMS wird eine Anfrage kontinuierlich auf die zugrundeliegenden Streams ausgeführt.

Die Geschwindigkeit, mit der neue Daten in das System gelangen, ist bei beiden Systemen ebenfalls unterschiedlich. Während bei einem DBMS ein Update „relativ“ selten ist, die Update-Raten also eher niedrig sind, kommt es bei einem DSMS zu sehr hohen Update-Raten, da die zugrundeliegenden Streams prinzipiell permanent geupdatet werden und somit auch das darauf arbeitende DSMS.

Die Art der zeitlichen Anforderungen variiert dabei folgendermaßen: Bei einem DBMS liegen normalerweise keine strengen Zeitvorgaben vor. Bei einem DSMS wird unter anderem aufgrund der hohen Update-Frequenz jedoch eine Echtzeitanforderung gestellt, damit alle Daten verarbeitet werden können.

Den unterschiedlichen Update-Frequenzen sowie Zeitanforderungen geschuldet gibt es bezüglich der Speicherverwendung und somit bezüglich des möglichen Speicherplatzes grundlegende Unterschiede. Während bei einem DBMS theoretisch eine unbeschränkte Größe des Sekundärspeichers z.B. in Form von Magnetfestplatten u.o.ä. zur Verfügung steht, da diese bei gelegentlichen Updates noch schnell genug sind, ist bei einem DSMS aufgrund der bereits erwähnten hohen Update-Frequenz die Größe des Hauptspeichers sehr wichtig. Das liegt daran, dass die Daten im Hauptspeicher direkt verarbeitet werden müssen, um die Echtzeitanforderung einzuhalten. Der Hauptspeicher lässt sich jedoch nur in einem gewissen Rahmen erweitern.

Bei der Verarbeitung der Daten ist auch der Umstand zu beachten, dass bei einem DBMS grundsätzlich nur der aktuelle Zustand der Daten Relevanz hat. Bei einem DSMS hingegen müssen neben den eigentlichen Daten auch noch einige andere Parameter beachtet werden. Hierzu zählt unter anderem die Eingangsreihenfolge. Die Eingangsreihenfolge ist z.B. wichtig, um die Aktualität der jeweiligen Daten zu bestimmen. Ist ein Tupel schon seit sehr langer Zeit im System, ist es eventuell nicht mehr aktuell und somit für die Bearbeitung irrelevant.

Ein weiterer Unterschied bezüglich der Daten besteht darin, dass in einem DBMS die vorhandenen Daten stets als exakt angenommen werden und auch als solche behandelt werden. In einem DSMS hingegen kann es vorkommen, dass die Daten veraltet sind oder ungenau, weshalb sie häufig gemittelt werden.

Ein unterschiedliches Herangehen gibt es auch beim Erstellen des Zugriffplans sowie dessen Optimierung. In einem DBMS hängen der Zugriffsplan und seine Optimierung hauptsächlich vom Query Operator und dem Design der Datenbank ab. Eine Optimierung ist in diesem Falle effizient möglich. In einem DSMS sind jedoch deutlich weniger Faktoren konstant und vorhersehbar. So ist es nicht möglich im Voraus zuverlässige Aussagen über die Art der Daten oder ihre Ankunft zu treffen. Dementsprechend ist die Optimierung des Zugriffsplans auch deutlich schwieriger und greift z.T. auch auf Annäherungen zurück.

Des Weiteren unterscheiden sich die Daten in ihrer Entstehung. In einem DBMS werden die Anfragen und Transaktionen von Menschen initialisiert, es liegt also ein sogenanntes „Human active, DBMS passive“-Model (HADP) vor. In einem DSMS hingegen liegt ein „DBMS active, Human passive“-System (DAHP) vor, weil die Daten von externen Quellen eintreffen und überwacht werden, ohne Zutun des menschlichen Nutzers. Der Nutzer wird nur dann tätig, wenn das DSMS ihn alarmiert, z.B. beim Auftreten abnormaler Werte für Sensoren.

	DBMS	**DSMS**

Datengrundlage	Persistent gespeicherte Relationen	Persistent gespeicherte Relationen + transiente Streams
Datenzugriff	Wahlfreier Zugriff	Sequentieller Zugriff
Update-Raten	Geringe Update-Rate	Hohe Update-Rate
Zeitliche Anforderungen	Keine besonderen Zeitanforderungen	Echtzeitanforderungen
Speicherplatz	Theoretisch unbeschränkter Sekundärspeicher	Durch vorhandenen Hauptspeicher beschränkt
Relevante Daten	Nur die Einträge in der Datenbank relevant	Zusätzlich zu den eigentlichen Daten noch weitere Parameter
Exaktheit der Daten	Alle im DBMS vorhandenen Daten als exakt angenommen	Datenbestand kann ältere Daten beinhalten
Zugriffsplanoptimierung	Hängt vom Query-Operator und Datenbank-Design ab	Hängt stark von Annäherungen ab, variable Datenankunft und Merkmale
Anfragen	Einmalige Anfragen	Kontinuierliche Anfragen

3 Continuous Query Language

Für eine einfache Abfrage an Streams würde es theoretisch zwar ausreichen, eine relationale Anfrage-Sprache zu nutzen, allerdings ist das bei wachsender Komplexität das Streams nicht mehr möglich. Um dieses Problem anzugehen, ist als Pendant zur SQL für relationale Datenbanken die sogenannte Continuous Query Language, kurz CQL, entstanden.

Man könnte die CQL auch als SQL mit einigen Erweiterungen und Modifikationen betrachten. Das ist unter anderem in den Beispielen in Abschnitt 3.1 ersichtlich.

Eine solche Erweiterung ist die Einführung eines neuen Datentyps für Streams. Die Behandlung der einzelnen Anfragen an sich verändert sich auch: Während Anfragen

auf einem DBMS als „one time event“ angesehen werden, so sind in einem DSMS alle Anfragen „continuous“. In einem DSMS geht man also grundsätzlich davon aus, dass eine Anfrage, wenn sie erst einmal registriert wurde, unendlich lange läuft. Damit die Anfragen auch auf Streams unendlich lange laufen können, zieht man Fenster zu Hilfe. Mithilfe eines Fensters kann man einen Teil eines Stream auswählen und die Anfrage nur noch auf diesem Teil des Streams ausführen. Es wird also ein Snapshot des dynamischen Streams in eine statische Relation umgewandelt. Hierbei grundlegend ist das Konzept der gleitenden Fenster. Ein gleitendes Fenster fokussiert die Aufmerksamkeit immer auf die neuesten Elemente innerhalb des Streams. Die Größe eines Fensters wird entweder als Zeitraum angeben, also z.B. „Range 1 Day“, oder aber über die Anzahl der zu betrachtenden letzten Tupel, also „Rows 50“. Des Weiteren greift man auf die Sample-Methode zurück, um Ergebnisse annähern zu können. Das genaue Verhalten sowie die Nutzung werden anhand einiger Beispiele verdeutlicht. Um Relationen und Streams zu vereinen, wurden neben den oben genannten drei Stream-to-Relation-Operatoren bei der CQL drei Relation-to-Stream-Operatoren eingeführt, welche in der folgenden Abbildung (Abbildung 3) veranschaulicht werden:

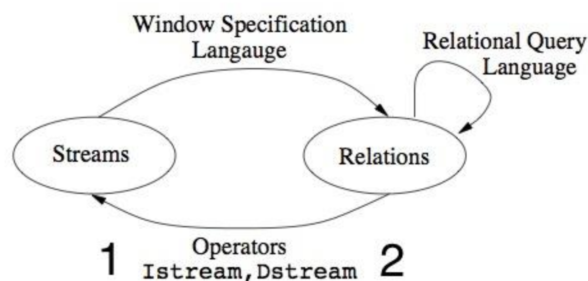


Abbildung (3), [9]

1. IStream („insert stream“)
   IStream wird immer dann ausgeführt, wenn ein Tupel in die zugehörige Relation eingefügt wird. Es wird hierbei aus einer Relation ein Datenstrom erstellt, in dem das Tupel, sofern es keine Kopie ist, an den Datenstrom angehängt wird.
2. DStream („delete stream“)
   DStream wird immer dann ausgeführt, wenn ein Tupel aus der zugehörigen Relation gelöscht wird. Hierbei wird das Tupel immer an den Datenstrom gesandt.
3. RStream („relation stream“)
   RStream wird immer dann ausgeführt, wenn ein Stream einen Teil der Tupel der zugehörigen Relation enthält. Die gesamte Relation wird in einen Datenstrom umgewandelt, der alle derzeit vorhandenen Tupel enthält.

Für weitere Informationen bezüglich der genauen Verwendung möchte ich an dieser Stelle auf die Beispiele verweisen.

3.1 CQL-Beispiele

Im folgenden Kapitel soll anhand einiger Beispiele die Verwendung der CQL demonstriert werden.

Unser Beispielstream ist ein Stream F bestehend aus Fotos. Hierbei setzt sich jedes Tupel aus dem eigentlichen Foto, welches der Einfachheit halber im Folgenden als BLOB-Variable betrachtet wird, einem Namen, der Größe des Fotos in bytes sowie einem Zeitstempel zusammen. In unserem ersten Beispiel wollen wir die Fotos herausfiltern, deren Namen das Wort „foo“ beinhalten und die im Laufe der letzten 24 Stunden den Stream passiert haben. Als Anwendung könnte man diese Anfrage nutzen, um die Bilder nach bestimmten Stichwörtern zu sortieren.

```
Select *
From  Fotos Fo [Range 1 Day Precending]
Where Fo.name like ‘foo’
```

An diesem Beispiel kann man sehr schön die bereits erwähnte Ähnlichkeit der Syntax der CQL mit der Syntax der SQL erkennen. Der einzige Unterschied zur klassischen SQL stellt „[Range 1 Day Preceinding]“ dar. Ein Fenster filtert die Daten heraus, die innerhalb eines Tages das System durchquert haben. Ein solches Fenster benötigt zwingend eine angegebene Größe und kann noch eine optionale Partitionsklausel oder Filterung beinhalten. Die Partitionsklausel unterteilt die Daten in kleinere Untergruppen. Für jede Untergruppe wird ein eigenes Fenster erzeugt. Zuletzt werden diese Fenster dann wiederum in ein einziges Fenster eingefügt.

Unser nächstes Beispiel soll nun die Verwendung einer Partitionsklausel demonstrieren:

```
Select *
From Fotos F
        [Partition BY F.name
        Rows 10 Precending
        Where timestamp <01.01.2000 00:00:00]
Where F.größe < 40000
```

Es ist auch möglich eine Anfrage auf mehrere Streams gleichzeitig anzuwenden. Zusätzlich zu unserem bereits bekannten Stream „Fotos“ gibt es nun noch einen Stream „Filme“. Die Tupel von „Filme“ besitzen zur Vereinfachung dieselben Eigenschaften wie die von „Fotos“.

```
Select *
From Fotos Fo, Filme Fi [Range 1 Day]
Where  Fo.name = Fi.name
```

Die oben bereits erwähnte Sample()-Methode dient dazu, einen Teil der Tupel auszuwählen. Sample(30) bedeutet hierbei, dass 30% des Streams per Zufall extrahiert werden und die restlichen 70% verworfen werden, z.B:

```
Select *
From Fotos Fo Sample(30), Filme Fi [Range 1 Day]
Where  Fo.name = Fi.name
```

Wie bereits vorgestellt, dienen die drei Operatoren IStream, Rstream und DStream dazu, aus einer Relation einen Stream zu kreieren, sie sind sozusagen die Gegenstücke zu den Fenstern, mit denen man aus einem Stream eine Relation erstellen kann.

```
Select IStream (*)
From Fotos f [Range Unbounded] where F.timestamp  <01.01.2000 00:00:00
```

Die Relation wird also in einen Stream transformiert. Allerdings ist es mit der CQL auch möglich, diese Anfrage deutlich einfacher und intuitiver aufzuschreiben:

```
Select *
From Fotos  F where F.timestamp  <01.01.2000 00:00:00
```

Man kann denselben Sachverhalt auch mit Hilfe des RStream-Operators und einem „Now"-Fenster ausdrücken, was für dieselbe Anfrage wie folgt aussieht:

```
Select RStream (*)
From Fotos f [NOW] where F.timestamp  <01.01.2000 00:00:00
```

4 Query Plan

4.1 Allgemein

Bei der Verarbeitung von Anfragen an ein Datenbanksystem gibt es mehrere unterschiedliche Ansätze. Entweder wird ein großer allgemeiner Query Plan erstellt, der sich um alle Anfragen, Berechnungen usw. sämtlicher User kümmert. Der Vorteil

dieses Vorgehens besteht unter anderem darin, dass eine Optimierung der Anfragen einfacher vonstattengehen kann. Auf die Optimierungsmöglichkeiten eines Query Plan wird später noch intensiver eingegangen. Ein Nachteil hierbei ist allerdings, dass jedes Ändern, Hinzufügen oder Löschen einzelner Anfragen den Query Plan aller anderen Anfragen mit beeinflusst, was einigen Overhead zur Folge hat. Diese Methode wird unter anderem in Aurora verwendet.

Die andere mögliche Herangehensweise ist, jeder Anfrage einen eigenen Query Plan zuzuweisen. Dabei werden die gerade beschriebenen Vorteile zu Nachteilen und andersherum. Da diese zweite Vorgehensweise in STREAM präferiert wird, nutze ich sie als Grundlage für Erklärungen und Beispiele.

Die Anfragen werden innerhalb des Systems registriert und daraufhin immer wieder ausgeführt, sobald neue Daten eintreffen. Im Folgenden gehe ich davon aus, dass für jede einzelne Anfrage ein eigener Query Plan erstellt und genutzt wird. Um den Ressourcenverbrauch gering zu halten, ist es sinnvoller, wenn Anfragen ihre Query Plans untereinander teilen. Wir gehen weiter vereinfachend davon aus, dass die Anfragen wie oben erwähnt bereits im System registriert sind, bevor ihre Streams die ersten Daten senden.

Ein Query Plan für STREAM unterstützt die Operatoren, welche in der folgenden Abbildung (Abbildung 4) veranschaulicht werden:

Name	Operator Type	Description
`select`	relation-to-relation	Filters elements based on predicate(s)
`project`	relation-to-relation	Duplicate-preserving projection
`binary-join`	relation-to-relation	Joins two input relations
`mjoin`	relation-to-relation	Multiway join from [22]
`union`	relation-to-relation	Bag union
`except`	relation-to-relation	Bag difference
`intersect`	relation-to-relation	Bag intersection
`antisemijoin`	relation-to-relation	Antisemijoin of two input relations
`aggregate`	relation-to-relation	Performs grouping and aggregation
`duplicate-eliminate`	relation-to-relation	Performs duplicate elimination
`seq-window`	stream-to-relation	Implements time-based, tuple-based, and partitioned windows
`i-stream`	relation-to-stream	Implements *Istream* semantics
`d-stream`	relation-to-stream	Implements *Dstream* semantics
`r-stream`	relation-to-stream	Implements *Rstream* semantics

Abbildung (4), [9]

Wie man sieht, lassen sich in der CQL alle Operatoren in die drei folgenden Operatoren-Klassen einteilen:

- Ein Relation-to-Relation-Operator nimmt eine oder mehrere Relationen als Input und gibt eine Relation als Output zurück.
- Ein Stream-to-Relation-Operator nimmt einen Stream als Input und gibt eine Relation als Output zurück.
- Ein Relation-to-Stream-Operator nimmt eine Relation als Input und gibt einen Stream als Output zurück.

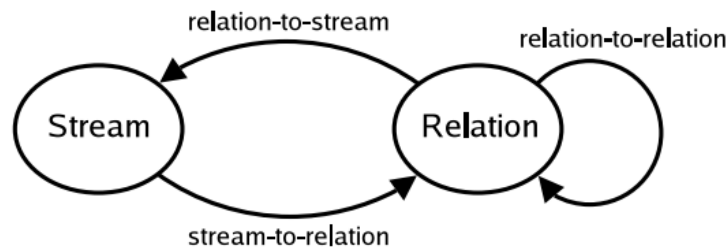


Abbildung (5),[9]

- Query-Operatoren: Diese Operatoren sind bereits bekannt aus den DBMS.
- Inter-operator-Queues: Die Funktion von Queues besteht darin, die einzelnen Operatoren miteinander zu verbinden und so einen Ablaufplan zu definieren.
- Synopsen: Synopsen fassen die Tupel der Ergebnismenge eines Operators in der Form zusammen, wie sie der darauffolgende Operator benötigt. Soll z.B. ein Operator zwei Ergebnismengen vereinigen, so wurden diese im vorherigen Schritt mithilfe einer Synopse zusammengefasst. Da die Synopsen mit wachsenden Ergebnismengen immer größer werden, müssen Techniken eingesetzt werden um das Wachstum zu beschränken oder zu verlangsamen. Beliebte Techniken hierfür sind *fixed-size hash tables, sliding windows, reservor samples, quantile estimates* und *histograms*.

An einem Beispiel möchte ich nun die neu eingeführten Begriffe demonstrieren. Die in Abbildung 6 dargestellte Anfrage ist:

```
Select *
From S1 [Range 15 Minutes], S2 [Rows 1000]
Where S1.A = S2.A
And S1.A <20
```

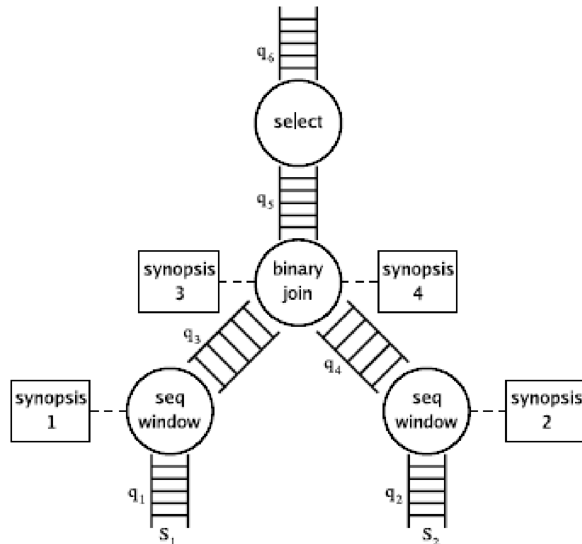


Abbildung (6), [9]

Es gibt also zwei Inputstreams S1 und S2, auf welche zunächst Fenster gelegt werden. Diese Fenster werden daraufhin miteinander gejoint und auf diesem *join* wird eine *select*-Anfrage ausgeführt. Die verschiedenen Operatoren werden durch die Queues q1-q6 miteinander verbunden. Die zu den jeweiligen Operatoren gehörenden Synopsen befinden sich neben den Operatoren.

In q1 und q2 befinden sich am Anfang nur die beiden Input-Streams, auf welche die Fenster mit ihren dazugehörigen Synopsen (synopsis1 und synopsis2) angewandt werden. Die Elemente dieser Fenster werden nun von q3 und q4 repräsentiert. In q3 befinden sich nun die Elemente des Fensters S1 [Range 15 Minutes] und in q4 die Elemente des Fensters S2 [Rows 1000]. Im nächsten Schritt findet die *join*-Operation, also der erste Teil der *where*-Klausel statt („where" S1.A = S2.A). Hierbei wird wie bereits erwähnt für jedes zu joinende Element eine eigene Synopse angelegt. Das Resultat der *join*-Operation ist dann wiederum q5. Auf q5 wird nun nur noch die *select*-Anfrage „S1.A < 20" ausgeführt und das Ergebnis als q6 übergeben.

4.2 Optimierungen am Query Plan

Bisher haben wir bei der Betrachtung der Query Plans einen wichtigen Faktor außen vor gelassen, Optimierungsmöglichkeiten bezüglich des Ressourcenverbrauchs. Am Beispiel des Query Plan in Abbildung 6 fallen zwei Verbesserungsmöglichkeiten auf: Besteht eine Anfrage aus mehreren zusammenhängenden Unteranfragen, ist es häufig möglich, dass diese Anfragen Ressourcen austauschen, um so den Overhead zu verringern. In STREAM ist genau das möglich und auch vorgesehen, wie Abbildung

7 demonstriert. In dieser Abbildung sieht man, wie die Operatoren untereinander ihre Synopsen über die Speicher Store 1 und Store 2 miteinander teilen und austauschen. Das ist besonders dann nützlich, wenn mehrere Synopsen beinahe identische Datenmengen abspeichern, wie man es beispielsweise in Abbildung 6 an Synopsis 1 und 3 sieht. Aus technischer Sicher ist noch hinzuzufügen, dass die Stubs dieselben Interfaces implementieren wie normale Synopsen, wodurch das Synopsis Sharing beliebig an- und abgeschaltet werden kann. Intern werden die Stubs wie folgt verwaltet: Da ein Operator nicht immer alle Daten innerhalb des Store berücksichtigen darf, wird festgelegt, welches Tupel innerhalb des Stores zu welchem Operator gehört. Dementsprechend werden neue Tupel aufgenommen, sobald sie von einem Operator benötigt werden, und erst dann entfernt, wenn kein Operator sie mehr braucht. Für Interessierte, die ins Detail gehen möchten, verweise ich auf [9, Kapitel 4.1].

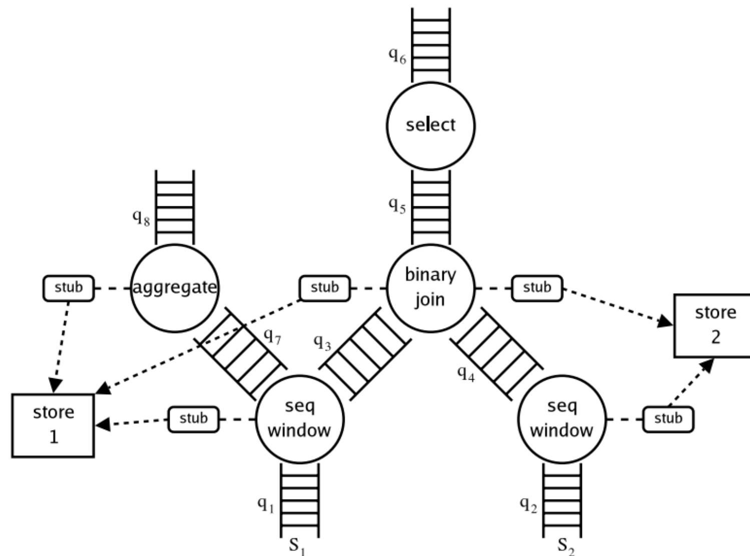


Abbildung (7), [9]

5 Approximation

5.1 Allgemein

Wegen der Unvorhersehbarkeit des Verhaltens von STREAM kann es in einem DSMS vorkommen, dass das System, trotz der bereits bekannten Maßnahmen zur Reduzierung des Speicher-Overheads, überlastet ist und somit nicht mehr in der Lage, die Anfragen in Echtzeit und korrekt zu beantworten. Die Ressourcen eines Systems werden im Allgemeinen durch zwei Hauptfaktoren limitiert: Auf der einen Seite ist die Rechenleistung limitiert, auf der anderen Seite ist die Leistung des Systems auch durch den verfügbaren Speicher nach oben hin begrenzt.

5.1.1 CPU-limitierte Annäherung

Dies kann auftreten, wenn die Geschwindigkeit, in der die neuen Daten eintreffen, zu hoch ist, sodass die CPU nicht mehr in der Lage ist, die einzelnen Anfragen abzuarbeiten. Das Problem kann durch das sogenannte „load shedding" umgangen werden. Unter „load shedding" versteht man die Streichung bestimmter Anfrageelemente, um Rechenzeit einzusparen. Für kürzere Rechenzeit verzichtet man also auf Präzision. Hierbei ist jedoch die genaue Implementierung entscheidend. So muss darauf geachtet werden, dass trotz des Weglassens von Elementen immer noch eine gewisse Präzision der Anfrage gewährleistet werden kann. Für einfache Beispiele, wenn z.B. von vornherein die statistische Verteilung der Werte bekannt ist, ist es möglich, die Präzision der Anfrage mathematisch zu berechnen. Was kompliziertere Anfragen betrifft, unter anderem solche, die *joins* beinhalten, verweise ich auf [9, Kapitel 6.1].

5.1.2 Speicher-limitierte Annäherung

Ein Engpass beim Speicherplatz entsteht klassischerweise, wenn zu viele Anfragen innerhalb des System registriert werden. Da nicht mehr alle Ergebnisse u.o.ä. bearbeitet und gespeichert werden können, „verschwinden" Teile mancher Ergebnisse.

Die Methodik zum Vermeiden von Speicherengpässen ist vom Grundgedanken her der Methodik zur Umgehung von Rechenzeitengpässen sehr ähnlich. Allerdings versucht man hierbei, die Größe der Synopsen zu verringern, statt wie zuvor die der Fenster. Dadurch bedingt ähneln sich auch die Vorzüge und Probleme beim Vorgehen. So führt die Reduzierung einer Synopse häufig dazu, dass Synopsen, die höher im Query Plan angesiedelt sind, ebenfalls kleiner werden und somit Speicherplatz eingespart wird. In Ausnahmefällen kann es jedoch vorkommen, dass eine Verkleinerung der Synopsengröße statt zu einer Verringerung des Speicherbedarfs zu mehr Speicherverbrauch führt – unter anderem dann, wenn die Synopse eingesetzt wurde, um Duplikate zu eliminieren.

Für weitere Methoden bezüglich der Verkleinerung der Synopsen sei auf [9, Kapitel 6.2] verwiesen.

5.2 Statische Annäherung

Nachdem wir nun die Annäherung zunächst in CPU-limitierte Annäherung und Speicher-limitierte Annäherung unterteilt haben, möchte ich sie noch in statische Annäherung und dynamische Annäherung unterteilen.

Zunächst zur statischen Annäherung: Der Grundgedanke der statischen Approximation ist die „Optimierung“ der Anfragen, wenn sie, wie in Abschnitt 4.1 erläutert, im System registriert werden, sodass sie bei Ausführung weniger Ressourcen benötigen. Bei der statischen Approximation gibt es zwei verschiedene Ansätze: die Window Reduction und die Sampling Rate Reduction.

5.2.1 Window Reduction

Wie bereits in einem der CQL-Beispiele (Abschnitt 3.1) gezeigt, können Anfragen ein Fenster beinhalten. Die Window Reduction setzt genau hier an und optimiert die Anfrage, indem sie die Größe dieses Fensters reduziert. Dadurch lassen sich sowohl Rechenzeit als auch Speicherplatz einsparen. Des Weiteren kann durch die Verkleinerung eines Fensters auch bei weiteren der Anfrage anhänglichen Anfragen Rechenzeit und Speicherplatz eingespart werden. Hierzu ein Beispiel:

Gehen wir von einer Anfrage aus, die eine *join*-Operation beinhaltet. Wenn nun die Größe eines der zu joinenden Fenster reduziert wird, wird auch die Größe aller *join*-Ergebnisse, die mit diesem Fenster verbunden werden, reduziert. Es gibt jedoch zwei Sonderfälle, in denen das nicht der Fall ist. Entweder wenn es die Aufgabe der Anfrage war, Duplikate zu eliminieren, oder aber wenn es sich um eine Negation handelte. Negationen sind z.B. „NOT EXIST“. In diesen beiden Sonderfällen würde eine Verkleinerung des Fensters ein gegensätzliches Ergebnis hervorrufen und zur Vergrößerung der Ergebnismenge am Ende führen. Somit müssen die beiden Sonderfälle vor der Optimierung abgefangen werden.

5.2.2 Sampling Rate Reduction

Vom Grundgedanken her unterscheidet sich die Sampling Rate Reduction von der Window Reduction nur dadurch, dass hier anstelle des Fensters der Input-Stream verkleinert wird. Das ganze kann wie in Kapitel 3 demonstriert über den Sample-Operator getan werden, indem man z.B. nur noch 70% des originalen Inputs betrachtet und 30% „wegwirft“. Die Eingangsfrequenz kann jedoch auch noch nachträglich umgeändert werden, auch wenn das in der originalen Anfrage nicht vorgesehen war. Einen großen Unterschied zur Window Reduction gibt es jedoch. Während bei der Window Reduction der Input verändert wird und somit bereits für die aktuelle Anfrage eine Ersparnis an Rechenzeit und Speicherverbrauch möglich ist, wird bei der Sampling Rate Redcution der Output verringert. Somit tritt eine Ersparnis frühestens für folgende Anfragen ein.

5.3 Dynamic Approximation

Nach den statischen Annäherungsmethoden kommen wir nun zu den dynamischen. Die dynamischen Annäherungsmethoden umgehen den großen Nachteil der statischen Optimierung, dass hier die Optimierung lediglich zum Zeitpunkt der Registrierung ins System geschehen kann. Die Anfragen können also nicht zur Laufzeit auch noch nachträglich an die jeweilige Auslastung im System angepasst werden. Bei der dynamischen Annäherung findet die Optimierung jedoch zum Zeitpunkt der Ausführung statt und kann so jedes Mal an die momentane Systemauslastung angepasst werden. Ist das System nicht ausgelastet, so werden die Anfragen präzise beantwortet, die Annäherung wird nur dann ausgeführt, wenn es gerade notwendig ist. Gerät das System an seine Leistungsgrenzen, kann die Auslastung durch später vorgestellte Methoden der dynamischen Annäherung gesenkt werden.

5.3.1 Synopsis Compression

Das Ziel der Synopsis Compression ist die dynamische Anpassung des Speicherbedarfs an den aktuellen Systemzustand. Das wird durch eine Verkleinerung der Synopsengrößen innerhalb des Query Plan erreicht. Das genaue Vorgehen bei der Synopsen-Verkleinerung wurde bereits im Abschnitt 5.1.2 beschrieben, da es mit dem Vorgehen beim Speicher-limitierten Ansatz identisch ist.

5.3.2 Sampling und Load Shedding

Das Ziel dieser Methode, wie auch das der Synopsis Compression, ist es, eine Optimierung durch Reduzierung des Speicherverbrauchs zu erreichen. Dabei werden, wie der Abschnittstitel schon andeutet, zwei Verfahren verwendet: das Sampling und das Load Shedding. Da Load Shedding auch in den CPU-limitierten Ansätzen seine Anwendung findet, ist bereits in Abschnitt 5.1.1 näher darauf eingegangen worden. Sampling wurde ebenfalls bereits erläutert, in Abschnit 5.2.2.

Unter Zuhilfenahme dieser beiden Techniken können wir also ebenfalls den Speicherverbrauch dynamisch an die aktuelle Situation anpassen, wobei Sampling in diesem Fall ein unverzerrteres Ergebnis liefert, während Load Shedding einfacher zu implementieren und flexibler zu verwenden ist.

6 Aurora

Neben dem in den vorigen Kapiteln vorgestellten DSMS STREAM gibt es noch weitere DSMS. Ein solches stellt Aurora[10] dar, welches unter Zusammenarbeit des M.I.T., der Brandeis-Universität sowie der Brown-Universität entstanden ist. Wie STREAM ist auch Aurora darauf ausgelegt, Daten aus mehreren unterschiedlichen

Quellen als Input entgegenzunehmen, diese nach im Vorfeld durch den Benutzer festgelegten Regeln zu bearbeiten und letztendlich dann das Ergebnis zu liefern. Zur Veranschaulichung des Arbeitsflusses wird in Aurora das „Boxes and Arrows“-Paradigma verwendet, wie in Abbildung 8 ersichtlich wird.

An der linken Seite der Abbildung sind die Input-Streams. Wie man sieht, besitzt der Graph keine Zyklen, wobei die Boxen die verschiedenen an den Daten durchgeführten Operationen darstellen. Auf der rechten Seite der Abbildung sind nun die Ausgangs-Streams zu sehen, welche an die Anwendungen weitergereicht werden. Des Weiteren ist Aurora in der Lage, die Speicherhistorie zu verwenden, um Ad-hoc-Anfragen unterstützen zu können.

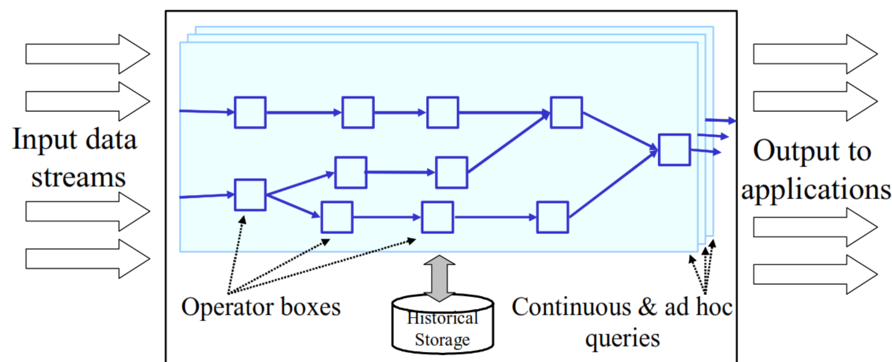


Abbildung (8), [10]

6.1 Query Language

Die Grundlage der Stream-Bearbeitung mit Aurora bilden acht Basisoperatoren. Zu diesen gehören unter anderem die „windowed operators“. Nachdem der Benutzer einen Operatoren ausgewählt und auf das Fenster angewandt hat, wird das Fenster „verschoben“, also neue Tupel aufgenommen, und der Prozess beginnt wieder von vorne.

Der *Slide*-Operator lässt ein Fenster auf dem Input-Stream um einige Tupel „hinabgleiten“. Das ist unter anderem dann nützlich, wenn laufende Berechnungen durchgeführt werden müssen, also z.B. Durchschnittswerte berechnet werden.

Der *Tumble*-Operator funktioniert ähnlich. Der Unterschied zwischen *Tumble* und *Slide* besteht darin, dass aufeinanderfolgende Fenster beim *Tumble*-Operator keine gemeinsamen Tupel besitzen. *Tumble* partitioniert einen Stream also vielmehr in aufeinanderfolgende disjunkte Fenster. Das ist zum Beispiel sinnvoll, wenn man die Indexe von täglichen Lagerbeständen errechnet, wo jede Kurseingabe in genau einer Indexberechnung verwendet wird.

Latch ähnelt *Tumble*, erstellt jedoch keine disjunkten Fenster, sondern ist in der Lage, die innere Verknüpfung von Fenster-Berechnungen aufrechtzuerhalten. Dies wird für Berechnungen benötigt, denen ein sogenanntes „endloses Fenster“ zugrunde

liegt – z.B. die Berechnung des Durchschnittwertes jedes einzelnen Bestandteils eines Lagerbestandes, und das über die gesamte Lebenszeit.

Einen grundsätzlich anderen Operator stellt *Resample* dar. *Resample* dient dazu, einen teilweise synthetischen Stream herzustellen, indem in den Ursprungsstream zusätzliche Tupel eingefügt werden. Das kann nützlich sein, wenn zusätzliche Informationen dem Stream für weitere Anfragen hinzugefügt werden sollen.

Außer den windowed operators gibt es noch Operatoren, die anstatt auf einem Fenster nur auf einem einzigen Tupel arbeiten. Das sind der *filter-, drop-, map-, groupby-* und der *join*-Operator.

Der Filter Operator filtert aus dem Stream bestimmte Tupel heraus, die vorher definierte Attribute besitzen.

Einen Spezialfall des *filter*-Operatoren stellt hierbei der *Drop* dar. *Drop* sortiert per Zufall Tupel aus, deren Häufigkeit und zuvor vom Benutzer definiert wird.

Map hingegen arbeitet statt auf dem Output auf dem Input. Bei *Map* wird eine definierte Inputfunktion auf sämtlichen Input-Tupel ausgeführt.

Mithilfe von *Groupby* lassen sich Tupel aus mehreren Streams in neue Streams zusammenfassen, deren Tupel entweder bestimmte Eigenschaften miteinander teilen oder in bestimmten Kategorien denselben Wert besitzen.

Zu guter Letzt gibt es noch den *join*-Operator. *Join* verbindet Tupel aus mehreren Input-Streams miteinander. Hierbei darf jedoch ein vorher definierter „Abstand" zwischen den Tupel nicht überschritten werden, als Beispiel kann hierfür der zeitliche Abstand zwischen zwei Zeitstempeln dienen. Z.B. kann man bei einem maximalen Abstand von 30 Minuten alle Wertpapiere herausfiltern, die innerhalb einer halben Stunde den gleichen Preis hatten.

Alle anderen Operatoren in Aurora entstehen durch die Kombination der primitiven Basisoperatoren, beispielsweise das Case-Statement, das durch eine Verbindung von *Map* und *Groupby* zustande kommt.

Was die Anfragesprache und ihre Operatoren betrifft, lässt sich bei Aurora also ein anderes Konzept erkennen als bei STREAM. Im Gegensatz zu STREAM kommt hier keine CQL zum Einsatz, sondern eine eigene Anfragesprache, die wie bereits erwähnt nur auf den acht Basisoperatoren basiert.

Aurora setzt also auf ein imperatives Modell der Anfragesprache, während STREAM CQL verwendet.

6.2 Query Plan/Model

Der Query Plan in Aurora basiert auf den drei Ausführungsmodi: kontinuierliche Anfragen, Sichten und Ad-hoc-Queries. Allen drei Modi liegt dasselbe Konzept zugrunde.

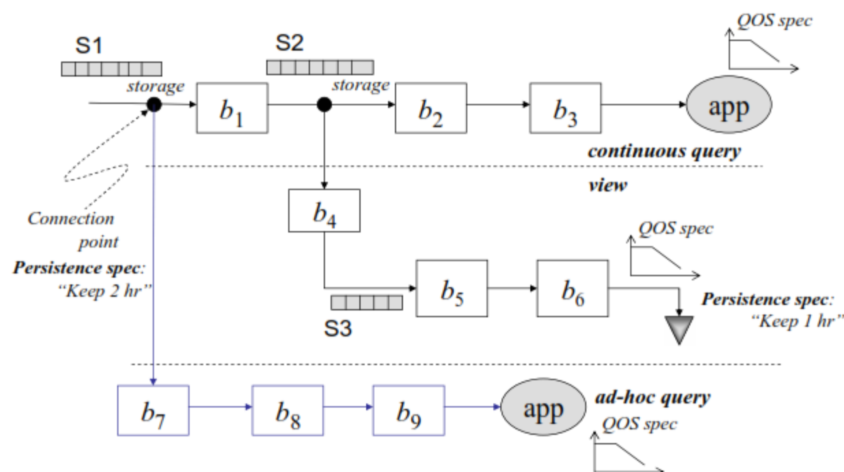


Abbildung (9) [10]

Die Abbildung zeigt den beispielhaften Aufbau eines Aurora-Netzwerkes an, der die grundlegenden Begriffe enthalten soll.

Der oberste Pfad ist der Pfad einer kontinuierlichen Anfrage, deren Input die beiden Streams S1 und S2 sind und auf welche die Operatoren b1, b2 und b3 angewandt werden. Die schwarzen Punkte auf dem Pfad stellen die Verbindungspunkte dar. Also die Punkte, an denen neue Boxen, bzw. Operatoren hinzugefügt oder gelöscht und neue Anfragen auf Teilergebnisse der bestehenden Anfragen gestellt werden können. Ein weiterer Vorteil von Verbindungspunkten ist, dass diese auch als persistenter Speicher benutzt werden können, da sämtliche Daten, die einen Verbindungspunkt passieren, für eine vorher bestimmte Zeitdauer zwischengespeichert werden und das Netzwerk nicht über die Anwendungen verlassen. In unserer Abbildung wird am Verbindungspunkt eine Speicherdauer von zwei Stunden angegeben.

Der mittlere Pfad hat als Besonderheit keine Anwendung, an die die Daten am Ende weitergegeben werden. Daran erkennt man, dass der mittlere Pfad für eine Sicht und nicht für eine Anfrage steht. Es ist Anwendungen jedoch jederzeit möglich, die Daten der Sicht zu bekommen. In unserem Beispiel wäre das nützlich, wenn eine Anwendung die Daten des Input-Streams S2 benötigen würde, bevor diese verarbeitet werden, oder aber um diese Daten an spätere Anfragen weiterzureichen. Die Teilergebnisse können aber auch an jedem beliebigen Punkt des Sicht-Pfades gespeichert werden. Ähnlich ist es bei materialized oder partially materialized views (ausgeführten oder teilweise ausgeführten Sichten). Die Umsetzung der Sicht verläuft unter der Kontrolle des Ablaufplaners.

Vergleichen wir nun die Query Plans von Aurora und STREAM, so lassen sich einige Unterschiede, aber auch einige Gemeinsamkeiten entdecken:

Während es bei STREAM für jede Anfrage einen eigenen Query Plan gibt, existiert in Aurora nur ein gesamter Query Plan für alle Anfragen zusammen, woraus bei der Ausführung und der Optimierung einige Unterschiede resultieren.

Auch die grafische Darstellung des jeweiligen Query Plan weist Unterschiede auf. In Aurora wird hierzu das „Boxes and Arrows"-Paradigma verwendet, in STREAM hingegen nicht.

Die einzelnen Elemente unterscheiden sich ebenfalls. Es gibt in Aurora keine äquivalenten Pendants für die Synopsen und Queues in STREAM. Dafür ist es in STREAM nicht ohne weiteres möglich, Ad-hoc-Anfragen zu stellen oder Sichten in den Query Plan zu integrieren.

Ein Vorteil des Query-Plan-Modells in STREAM besteht darin, dass im Gegensatz zu Aurora direct entry of plans möglich sind.

Eine weitere Eigenheit von Aurora sind die Dienstgüte-Angaben an jedem Output-Stream.

6.3 Optimierung

In Aurora lassen sich die Optimierungsmöglichkeiten in zwei große Hautgruppen einteilen: die dynamische kontinuierliche Anfragenoptimierung und die Ad-hoc-Anfragenoptimierung.

6.3.1 Dynamische kontinuierliche Anfragenoptimierung

Um das Vorgehen der dynamischen kontinuierlichen Anfragenoptimierung zu demonstrieren, starten wir zunächst mit einem komplett unoptimierten Netzwerk, also z.B. dem vom User entworfenen Aurora-Netzwerk in seiner Urform. Unser Ziel ist es nun, das Netzwerk zur Laufzeit zu optimieren.

Es ist möglich, die Anzahl der Tupel innerhalb des Netzwerks mithilfe des *map*-Operators zu minimieren, indem man den *map*-Operator an der frühestmöglichen Stelle, die durch eine Prüfung des gesamten Netzwerks ermittelt wird, einsetzt, um alle nicht benötigten Tupel zu entfernen. Dadurch wird die Tupelmenge für alle folgenden Operatoren verkleinert und die Leistungsstärke des Netwerkes verbessert. Hierfür ist es allerdings notwendig, dass das System mit Operatorbeschreibungen arbeitet, anhand welcher die benötigten Attribute erkennbar werden.

Eine weitere Möglichkeit zum Einsparen von Ressourcen ist die Kombination verschiedener Boxen. So können unter anderem die Operatoren *Map* und *Filter* mit fast allen anderen Operatoren kombiniert werden. Zwei *filter*-Operatoren können sehr einfach zu einem einzigen zusammengefasst und dadurch effizienter eingesetzt werden. Dadurch sparen wir nicht nur den Overhead der nicht länger benötigten Box, sondern wir könnten auch gleichzeitig bereits bekannte Optimierungsmöglichkeiten aus relationalen Datenbanken auf die neue große Box anwenden. Ein weiterer Vorteil: Durch die Kombination zweier Boxen wird die Gesamtanzahl der Boxen reduziert und das Aurora-Netzwerk dadurch insgesamt entlastet.

Eine dritte Optimierungsmöglichkeit stellt die Neuanordnung einzelner Boxen dar. Eine Neuordnung ist häufig dann nützlich, wenn von vornherein bereits Tupel ausgeschlossen werden können und nicht weiter verarbeitet werden. Wenn z.B. ein *filter*-Operator vor eine Anfrage gesetzt wird anstatt dahinter, können bereits Tupel ausgefiltert werden, und die Anfrage muss weniger Tupel bearbeiten. Die Art und Weise, auf die Aurora entscheidet, wie und ob die einzelnen Boxen in ihrer Reihenfolge verändert werden, zeigt das folgende Modell. Wir gehen davon aus, dass jede Box b in unserem Netzwerk die Kosten c besitzt, wobei die Kosten als die Ausführungsdauer betrachtet werden, die b braucht, um ein Input-Tupel zu verarbeiten. Jede Box hat noch eine Selektivität s, die die Anzahl der erwarteten Output-Tupel pro Input-Tupel angibt. Nehmen wir nun also an, wir haben zwei Boxen b1 und b2, wobei b2 ein Nachfolger von b1 ist. Für jedes Input-Tupel von b1 sind die Gesamtkosten für das Durchlaufen unseres Netzes folglich:

$$c(b1) + c(b2) * s(b1).$$

Drehen wir die Operatoren um, erhalten wir ein ähnliches Ergebnis. Somit können wir die Bedingung errechnen, auf Grundlage derer wir entscheiden können, ob die Boxen wie folgt vertauscht werden müssen:

$$1 - s(b2)/b2) > 1 - s(b1)/c(b1)$$

Eine optimale Lösung ist dann erreicht, wenn alle Boxen nach der Menge ihrer Gemeinsamkeiten sortiert sind, also die Boxen, die die wenigsten Elemente mit anderen teilen, zu Beginn des Netzwerkes. In Aurora geschieht dies durch einen heuristischen Algorithmus, der die Boxen so lange neu anordnet, bis keine weiteren möglichen Reihenfolgen mehr erstellt werden können. Sind so alle sinnvollen Reihenfolgen gefunden worden, wird ein neues Unternetzwerk im bereits bestehenden Netzwerk erzeugt, und dem Ablaufplaner mitgeteilt, dass keine weiteren Nachrichten mehr am Input-Verbindungspunkt zwischengespeichert werden müssen. Während es bei den Outputs durch das Neueinfügen der Unternetzwerke zu einer kleinen Verzögerung kommt, werden die restlichen Prozesse, z.B. die Input-Streams, nicht gestört.

Ein Aurora-Netz besteht aus vielen kleinen Unternetzwerken, die jeweils eigenständig optimiert werden. So verbietet Aurora sogar eine Optimierung über Verbindungspunkte hinweg.

Die Optimierung wird vom Netzwerk periodisch für alle Unternetzwerke angestoßen und läuft dabei als Hintergrundtask. So werden keine laufenden Prozesse im Netzwerk behindert.

6.3.2 Ad-hoc-Anfragenoptimierung

Die Ad-hoc-Anfragenoptimierung wird in Aurora getrennt von der in Abschnitt 6.3.1 erwähnten Optimierung behandelt. Eine Ad-hoc-Anfrage muss die gesamte Informationshistorie an den Verbindungspunkten, an denen die Anfrage angedockt ist, berücksichtigen.

In Aurora werden Ad-hoc-Anfragen bearbeitet, indem sie in zwei unterschiedliche Unternetzwerke aufgeteilt werden. Die Optimierung kann also für jedes Netzwerk einzeln vorgenommen werden.

In Aurora wurde festgelegt, dass immer zuerst das Unternetzwerk mit der Datenhistorie durchlaufen wird.

Da die Informationshistorie in Aurora als ein B-Baum organisiert ist, startet der Optimierungsprozess an jedem Verbindungspunkt und untersucht die darauf folgenden Boxen. Handelt es sich bei einer der auf den Verbindungspunkt folgenden Boxen um einen Filter, wird die Filterkondition auf Kompatibilität mit dem Speicherschlüssel des Verbindungspunkts hin untersucht. Liegt Kompatibilität vor, wird die Implementierung des Filters dergestalt geändert, dass ein katalogisiertes Nachschlagen auf dem B-Baum ausgeführt wird.

Handelt es sich bei der folgenden Box um eine *join*-Operation, wählt der Optimierer aus einem Merge-Sort oder einem katalogisierten Nachschlagen die jeweils billigere Variante aus und passt die Implementierung der *join*-Operation dann seiner Wahl an.

Join und *Filter* sind die einzigen Operatoren, die eine indexierte Struktur nutzen können, und werden deshalb als einzige berücksichtigt. Sobald jedoch die erste Box mit ihrer Arbeit an der Tupel-Historie angefangen hat, geht der Index verloren und alle darauffolgenden Boxen arbeiten nach dem bereits bekannten Schema.

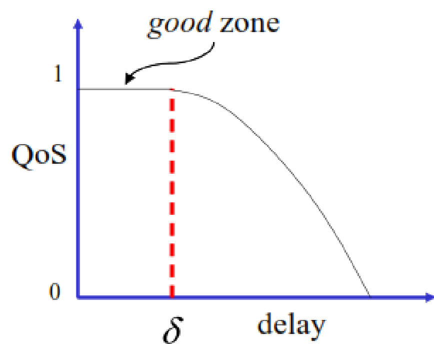
6.4 Dienstgüte-Datenstrukturen

Aurora ist bestrebt, die objektive Dienstgüte des Outputs zu verbessern. Die Dienstgüte wird als mehrdimensionale Funktion mit den folgenden Eigenschaften dargestellt:

- Antwortzeiten: Der Output sollte rechtzeitig erstellt werden, da die Wartedauer sich sonst verlängert und infolgedessen die Dienstgüte abnimmt.
- Tupel Drops: Wenn Tupel gedropt werden, z.B. um die Systemauslastung zu verringern, verschlechtert sie die Dienstgüte des betroffenen Outputs.
- Produzierter Wert: Die Dienstgüte hängt außerdem noch davon ab, ob wichtige Werte produziert wurden oder nicht.

Es ist kaum sinnvoll, den Anwendungsadministrator eine mehrdimensionale Funktion entwerfen zu lassen. Stattdessen greift Aurora auf eine simplere Taktik zurück, die für

Menschen einfacher anzuwenden ist: Für jeden Output-Stream liefert der Anwendungsadministrator Aurora ein zweidimensionales Dienstgüte-Diagramm, das die Wartezeiten beim Verarbeiten der erstellten Tupel aufzeigt (Abbildung 10). Hier hat die Dienstgüte des Outputs ihr Maximum erreicht, wenn die Wartedauer vor der Grenze δ liegt. Im Bereich hinter δ nimmt die Dienstgüte umso mehr ab, je länger die Wartedauer ist.



(a) Delay-based

Abbildung (10),[10]

Es besteht die Möglichkeit, dass der Anwendungsadministrator Aurora noch zwei zusätzliche Diagramme liefert, für alle Outputs eines Aurora-Systems. Das erste, Abbildung 11(b), veranschaulicht den Prozentsatz der gelieferten Tupel. In diesem Fall gibt der Anwendungsadministrator an, dass eine hohe Dienstgüte erreicht wird, wenn so viele Tupel wie möglich ihr Ziel erreichen, und dass die Dienstgüte abnimmt, wenn Tupel fallen gelassen werden. Der zweite optionale Graph für Outputs ist als Abbildung Abbildung 11(c) zu sehen. Die möglichen Ergebnisse (values), die als Outputs erstellt werden, befinden sich auf der horizontalen Achse und der Dienstgüte-Graph gibt ihre Wichtigkeit an. Dieser auf Ergebnisse ausgelegte Dienstgüte-Graph zeigt uns, dass manche Outputs wichtiger sind als andere. Zum Beispiel sind Outputs in einer Anwendung zur Betriebsüberwachung in der Nähe eines kritischen Bereichs wesentlich wichtiger als solche, die weiter davon entfernt sind. Auch kann Aurora auf Grundlage der Ergebnis-orientierten Informationen über die Dienstgüte die Systemauslastung umsichtiger reduzieren, als es ohne besagte Informationen des Anwendungsadministrators möglich wäre.

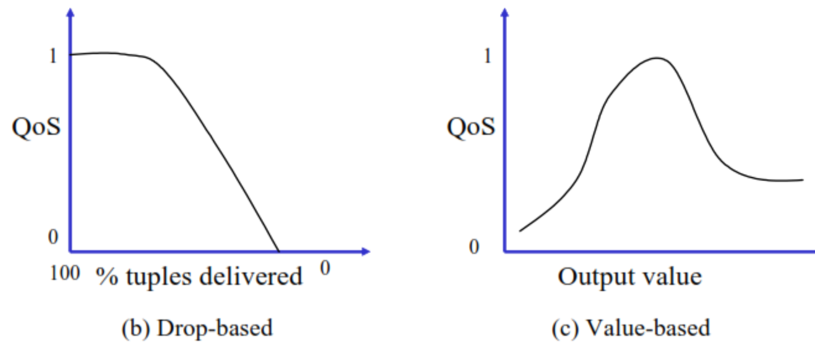


Abbildung (11),[10]

Aurora stellt einige Anforderungen an die Dienstgüte-Diagramme. Zum einen setzt Aurora voraus, dass alle Dienstgüte-Diagramme genormt sind, damit die Dienstgüte verschiedener Outputs quantitativ verglichen werden kann. Zum anderen setzt Aurora voraus, dass der Wert von δ ausgeführt werden kann, das bedeutet, dass ein entsprechend großes Aurora-Netzwerk alle Outputs aus dem guten Bereich, links von δ, dauerhaft verarbeiten kann. Das erfordert die Wartezeit, die von den kompletten Verarbeitungskosten entlang des längsten Pfades von einer Datenquelle bis zu diesem Output verursacht wird, und die δ nicht überschreiten darf. Wenn der Anwendungsadministrator Aurora nicht mit ausführbaren Diagrammen versorgt, können die Algorithmen in den unteren Abschnitten keine guten Resultate erzielen. Zu guter Letzt setzt Aurora voraus, dass alle Dienstgüte-Diagramme konvex sind (der Ergebnis-orientierte Graph in Abbildung 4c bildet eine Ausnahme). Diese Anforderung ist nicht nur sinnvoll, sondern auch notwendig, damit die von Aurora verwendeten gradient-walking-Methoden zur Ablaufplanung und Systementlastung angewandt werden können.

Es ist zu beachten, dass Auroras Verständnis von Dienstgüte über die hier aufgeführten Diagramme hinausgeht. Aurora kann auch mit anderen eigenständigen Attributen arbeiten, zum Beispiel mit *troughput*, oder zusammengesetzten Attributen, zum Beispiel *weighted*, einer linearen Kombination von *troughput* und *latency*.

6.5 Speicherung

In Aurora wird die Speicherung aller benötigten Daten vom Auror-Speichermanager (ASM) übernommen. Dieser soll einerseits sicherstellen, dass die Tupel, die das Aurora-Netzwerk gerade passieren, gespeichert werden, und andererseits den z.B. an Verbindungspunkten benötigten Speicherplatz garantieren.

6.5.1 Queue Management

Jede Windows Operation benötigt mindestens eine Speicherung der Tupel, die innerhalb des Fensters ankommen. Es kann jedoch vorkommen, dass z.B. die Ankunftsrate neuer Daten größer ist als die Bearbeitungsgeschwindigkeit des Systems. In diesem Fall müssen zusätzlich Daten zwischengespeichert werden, damit sie nicht einfach verloren gehen. Der ASM hat dabei die Aufgabe, die verschiedenen über das System verteilten Queues mit ihren Tupel zu managen. Eine solche Queue befindet sich am Ende jedes Operators und wird von allen nachfolgenden Operatoren ebenfalls benutzt. Ein nachfolgender Operator besitzt zwei auf die Queue des vorigen Operators gerichteten Zeiger. Ein Zeiger weist auf den Kopf und somit auf das älteste Tupel, welches noch nicht vom nachfolgenden Operator verarbeitet worden ist. Der andere Zeiger weist auf den Schwanz der Liste, der gleichzeitig das älteste benötigte Tupel ist. Diese beiden Zeiger markieren somit das Fenster, das gerade vom Operator bearbeitet wird, und wandern während der Bearbeitung über den Input-Stream. Für die Verwaltung der Zeiger im System sowie für das Verwerfen von Tupel, die älter sind als das Tupel auf dem Schwanz der Queue, ist der ASM zuständig.

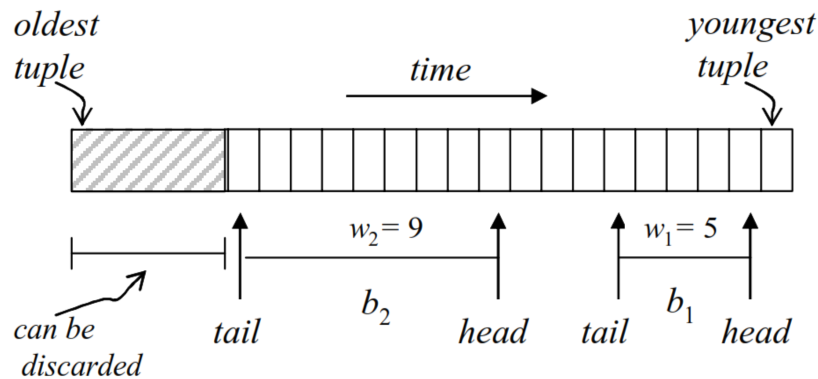


Abbildung (12), [10]

Um eine beliebige Skalierbarkeit des ASM sicherzustellen, wurde in Aurora darauf verzichtet, sämtliche Queues einfach in den Hauptspeicher abzulegen. Stattdessen wird der Festplattenspeicher in Blöcke eingeteilt, deren Größe im Voraus in der Variable „block_size“ angegeben wird. Als für Festplatten typische Blockgröße werden hierbei 128kb oder größer angesehen. Jeder Queue wird nun ein solcher Block als Speicherort zugewiesen. Dies funktioniert, solange die Queue kleiner bleibt als die vorher festgelegte Speichergröße der einzelnen Blöcke. Kommt es nun innerhalb der Queue zu einer Überstrapazierung des Speichers, versucht der ASM einen weiteren leeren Block zu finden und fügt diesen als der Queue zugehörigen Speicherplatz hinzu, die Queuegröße wird hierbei verdoppelt.

Beim Start legt der ASM zunächst einen Speicherbereich zum Zwischenpuffern der Queues fest und partitioniert diesen in Blöcke der gegebenen Größe. Der Wechsel

zwischen Haupt- und Festplattenspeicher findet in Aurora mithilfe einer „Novel Replacement Policy“ statt. In Aurora sieht die Replacement Policy wie folgt aus:

Ablaufplaner und ASM teilen sich innerhalb von Aurora eine Tabelle, in der jeder Box eine Reihe zugewiesen wird. In dieser Reihe werden die jeweilige Priorität der Box, der Prozentsatz ihrer Queue, der sich im Hauptspeicher befindet, und ein Flag, das aussagt, ob die Box gerade aktiv ist oder nicht, gespeichert. Logischerweise wird die Priorität hierbei vom Ablaufplaner beeinflusst und verändert, während der ASM für den Prozentsatz der Queue zuständig ist.

Wird nun Speicherplatz im Hauptspeicher benötigt, wird der Block mit der geringsten Priorität aus dem Hauptspeicher verdrängt. Des Weiteren werden Blöcke, die nicht zu einer laufenden Queue gehören, aus dem Hauptspeicher verdrängt und durch einen Speicherblock ersetzt, der zu einer Queue mit einer höheren Priorität gehört. Auf diese Weise versucht ASM immer die benötigten Daten im Hauptspeicher zu halten und somit lange Verzögerungen bei Zugriffen auf den Festplattenspeicher zu verhindern.

6.5.2 Verbindungspunkte

Die Verbindungspunkte im Aurora-Netzwerk sind Punkte, an denen andere Boxen andocken können. Dieses Design dient dem Zweck, dass das Netzwerk in der Lage ist, Ad-hoc-Anfragen zu bearbeiten. Jeder Verbindungspunkt hat eine HISTORY REQUIRMENTS und einen optimalen Speicherschlüssel. Während die HISTORICAL REQUIREMENTS angeben, wie viele historische Informationen gespeichert bleiben müssen, gibt der Speicherschlüssel an, nach welchem Primärschlüssel die Informationen gespeichert werden sollen. Im Normalfall ist die Menge der historischen Informationen, die noch aufgenommen werden müssen, größer als die Fenstergröße der folgenden Boxen, und in diesem Fall muss weiterer Speicher zugewiesen werden. Der ASM organisiert die historischen Tupel in einem B-Baum, wobei als Speicherschlüssel entweder der angegebene oder, wenn kein Schlüssel angegeben ist, der Zeitstempel verwendet wird. Hierbei werden die Tupel, die sich am Ende der Queue befinden, also am Verbindungspunkt, zusammengefasst und im B-Baum eingefügt. Periodisch wird dieser B-Baum durchlaufen und dabei werden dann wiederum alle Tupel gelöscht, die älter sind, als es die historische Bedingung vorschreibt.

6.6 Ablaufplanung

Die Ablaufplanung muss in Aurora viele verschiedene Bedingungen berücksichtigen, unter anderem die Abhängigkeiten zwischen den einzelnen Boxen, die Echtzeitanforderung für die Leistungsstärke des Systems usw. Sie ist dementsprechend komplex.

Grundlegend lässt sich sagen, dass die Entscheidungen bei der Ablaufplanung nicht auf einem einzigen Kriterium basieren sollten. Würden zur Ablaufplanung z.B. lediglich die Dienstgüte-Anforderungen herangezogen und andere Entscheidungskriterien wie z.B. die Zeit, die ein Tupel benötigt, um das System zu durchqueren, vernachlässigt, so könnte dies zu großen Leistungsverlusten führen. Es ist folglich nicht

Auroras einziges Ziel, die Dienstgüte zu maximieren, sondern auch die Ausführungskosten für einzelne Tupel zu reduzieren. Beide Ziele werden in Aurora durch den folgenden Ansatz erfüllt:

6.6.1 Zug-Ablaufplanung

Die Minimierung der Prozesskosten basiert in Aurora auf zwei „non-linearities“:

InterBox non linearity: Die Kosten für die Bearbeitung eines Ende-zu-Ende-Tupels steigen sehr stark an, wenn der Pufferspeicher nicht genügend groß ist und die Tupel immer wieder zwischen Haupt- und Festplattenspeicher hin und her verschoben werden müssen. Somit ist es unter dem Gesichtspunkt der Leistungsstärke sehr wichtig, genau diesen Fall zu verhindern. Eine weitere Form der Inter-Box non linearity liegt vor, wenn wir Tupel haben, die von der Queue einer Box in eine andere übergehen, wenn also der Output der ersten Box b1 der Input einer auf b1 folgenden Box b2 ist. Wenn nun der Ablaufplaner in der Lage ist, b2 direkt nach b1 auszuführen, kann der „Umweg“ über den Speichermanager eingespart werden und damit verbunden auch einiges an Overhead.

Intra-Box non-linearity: Die Kosten für die Tupel-Verarbeitung sinken in einigen Fällen, wenn die Anzahl der Tupel, die bereits für die Verarbeitung an einer bestimmten Box verfügbar sind, steigt. Dieser Effekt kann aufgrund von zwei Ursachen eintreten:

Erstens, wenn die Gesamtanzahl der Box-Aufrufe, die für die Verarbeitung einer bestimmten Anzahl Tupel nötig sind, abnimmt, wodurch auch die „low level“-Overheads sinken. Zweitens, wenn eine Box ihre Ausführung umso besser optimieren kann, je mehr Tupel sie besitzt. Zum Beispiel kann eine Box Zwischenergebnisse erstellen und diese bei Fenster-Operationen wiederverwenden.

Zug-Ablaufplanung ist eine Menge von heuristischen Methoden, mithilfe derer Aurora die Vorteile der beiden genannten non-linearities bei der Ablaufplanung nutzen möchte. Die heuristischen Methoden müssen hierbei drei Bedingungen erfüllen:

- Es wird versucht möglichst große Queues anzulegen, ohne dass es zu Auswertungen der Tupel kommt. Es werden also lange Tupel-Züge angelegt.
- Diese langen Tupel-Züge sollen wenn möglich auf einmal abgearbeitet werden. Es wird die Intra-Box non-linearity genutzt.
- Die langen Tupel-Züge sollen in kleine Untergruppen eingeteilt werden, um weniger zugriffe auf den Festplattenspeicher zu haben. Es wird also die Inter-Box non-linearity genutzt.

Aus diesen drei genannten Punkten lassen sich die Hauptziele der Zug-Ablaufplanung leicht ablesen: die Minimierung der I/O-Operationen pro Tupel und die Minimierung der Box-Aufrufe pro Tupel.

6.6.2 Prioritätszuweisung

Ein weiteres Problem, welchem sich die Ablaufplanung stellen muss, ist die Minimierung der Latenz der einzelnen Tupel. Die Latenz eines Tupel setzt sich aus der Summe seiner Auswertungsverzögerungen sowie seiner Warteverzögerung zusammen. Während die Ablaufplanung auf die Auswertungsverzögerungen keinen Einfluss nehmen kann, wird die Warteverzögerung direkt durch die Ablaufplanung beeinflusst. Das Ziel ist es also, die Outputs mit Prioritäten zu versehen, damit die daraus resultierenden Warteverzögerungen eine Maximierung der Dienstgüte zur Folge haben. Die Priorität, die hierbei einem Output zugewiesen wird, soll seine Dringlichkeit anzeigen. In Aurora gibt es zwei Methoden, die jeweiligen Prioritäten zu bestimmen: Status-basiert und Feedback-basiert.

Bei der Status-basierten Methode wird die Dringlichkeit der Outputs anhand seines erwarteten Nutzens für den momentanen Systemzustand eingeteilt. Es wird stets der Output mit der höchsten Nützlichkeit als nächstes ausgeführt. Die Nützlichkeit wird bestimmt, indem berechnet wird, wie viel Dienstgüte geopfert werden müsste, wenn der jeweilige Output verzögert würde.

Der Feedback-basierte Ansatz hingegen überwacht die Gesamtleistung innerhalb des Systems und passt die Dringlichkeit dynamisch an. Hierbei wird die Priorität der Outputs erhöht, die Verzögerungen verursachen, und die Prioritäten der Anwendungen herabgesetzt, die gut laufen.

6.6.3 Aurora-Ablaufplanung

Aufgrund des gewaltigen Umfangs des hochdynamischen System sowie der Granularität (Berechnungen pro Kommunikation) der Ablaufplanung ist die Suche nach der besten Lösung sinnlos. Die für die Suche benötigte Rechenzeit würde in vielen Fällen zu einer Nichterfüllung des Echtzeitkriteriums führen. Deswegen werden in Aurora heuristische Methoden eingesetzt, die sowohl Echtzeitanforderungen als auch eine Kostenreduzierung berücksichtigen.

Eine der in Aurora zum Einsatz kommenden heuristischen Methoden funktioniert wie folgt:

Als erstes wird ein Output zur Ausführung ausgewählt, dann wird die erste Downstream-Box ermittelt, deren Queue gespeichert ist. (Hierbei ist zu beachten, dass eine Box nur dann strukturiert werden kann, wenn ihre Queue mindestens so viele Tupels enthält wie das dazugehörige Fenster.) Danach werden in Upstream-Richtung so lange weitere Boxen überprüft, bis Aurora eine Box ohne dazugehörige gespeicherte Queue findet, oder aber bis alle Boxen überprüft wurden sind. Die somit überprüften Boxen werden nun nacheinander aufgereiht, es entsteht also eine „Superbox". Diese Superbox wiederum kann nun Box für Box strukturiert und umsortiert werden.

Bei der Ausführung einzelner Boxen kontaktiert Aurora den Speichermanager, damit dieser die Queue in den Zwischenspeicher ablegt. Daraufhin werden die Positionen der Input-Queue im Verarbeitungscode der Box übermittelt, sowie die Menge der zu verarbeiteten Tupel. All das übergibt Aurora nun einem im System verfügbaren Thread.

7 Fazit

Es wurde festgestellt, dass dem Data Stream Processing eine immer größere Bedeutung zuteil wird. Um diese anspruchsvollen Anwendungen effizient einsetzen zu können, müssen wir nicht nur zahlreiche bestehende Aspekte des Datenbank-Designs und ihrer Ausführung überdenken, sondern auch neuartige initiative Datenspeicher- und Verarbeitungskonzepte entwickeln. Als mögliche Lösungsansätze für auftretende Probleme wurden zwei verschiedenen DSMS vorgestellt und miteinander verglichen.

8 Reference

1. Kaushik Chakrabarti Minos Garofalakis Rajeev Rastogi Kyuseok Shim, **Approximate Query Processing Using Wavelets**
2. Minos Garofalakis Johannes Gehrke Rajeev Rastogi, **Querying and Mining Data Streams: You Only Get One Look**
3. Konstantinos Giannakopoulos, **Static Optimisation vs. Dynamic Evaluation for Data Stream Processing**, University of Edinburgh, , 2011
4. Sirish Chandrasekaran Michael J. Franklin, **Streaming Queries over Streaming Data**, University of California at Berkeley
5. Mengmeng Liu Svilen Mihaylov, **Data Stream Processing**
6. Arvind Arasu,Brian Babcock Shivnath Babu Jon McAlister Jennifer Widom, Characterizing Memory Requirements for Queries over Continuous Data Streams, Stanford University
7. Brian Babcock Shivnath Babu Mayur Datar Rajeev Motwani Jennifer Widom ,**Models and Issues in Data Stream Systems**, epartment of Computer Science Stanford University Stanford, CA 94305
8. Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, Rohit Varma, **Query Processing Resource Management and Approximation in a Data Stream Management System**, Stanford University
9. Arvind Arasu Brian Babcock Shivnath Babu John Cieslewicz Mayur Datar Keith Ito Rajeev Motwani Utkarsh Srivastava Jennifer Widom, **STREAM: The Stanford Data Stream Management System**, Department of Computer Science, Stanford University
10. Don Carney Uğur Çetintemel Mitch Cherniack Christian Convey Sangdon Lee Greg Seidman Michael Stonebraker Nesime Tatbul Stan Zdonik, **Monitoring Streams – A New Class of Data Management Applications,** Brown University M.I.T.