Anfragesprachen

Markus Wörz

8.7.2005

Motivation für Anfragesprachen

Warum Anfragesprachen?

- Datenströme liefern enorme Datenmengen
- Problem: unverarbeitet für Mensch nicht brauchbar
- Information entsteht erst durch komplexe Analysen
- Anfragesprache macht Analysen leicht formulierbar

Motivation für Anfragesprachen

Warum Anfragesprachen?

- Datenströme liefern enorme Datenmengen
- Problem: unverarbeitet für Mensch nicht brauchbar
- Information entsteht erst durch komplexe Analysen
- Anfragesprache macht Analysen leicht formulierbar

Hier:

- Fokus auf kontinuierlichen Anfragen
- Lediglich tupelbasierte Ströme

Klassifikation von Anfragesprachen

Sprachen lassen sich klassifizieren in

deklarative

```
SELECT ...
FROM InputStream ...
WHERE ...
```

Klassifikation von Anfragesprachen

Sprachen lassen sich klassifizieren in

deklarative

```
SELECT ...
FROM InputStream ...
WHERE ...
```

imperative

```
Iterate ( over InputStream
          filteredby ... ) {
  ...
};
```

Klassifikation von Anfragesprachen

Sprachen lassen sich klassifizieren in

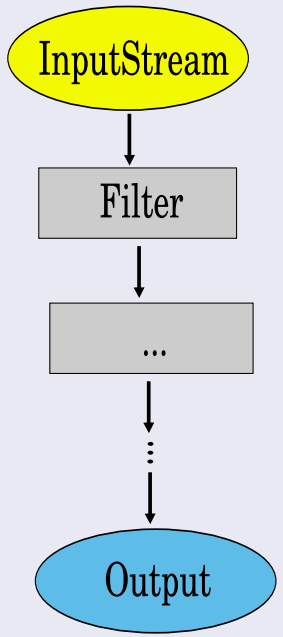
deklarative

```
SELECT ...
FROM InputStream ...
WHERE ...
```

imperative

```
Iterate ( over InputStream
          filteredby ... ) {
  ...
};
```

graphorientierte



Beispielstrom zur Demonstration

Gegeben: Strom von Telefongesprächen mit der Signatur:

```
CallStream
(
  origin: integer
  called: integer
  duration: integer
)
```

Beispielstrom zur Demonstration

Gegeben: Strom von Telefongesprächen mit der Signatur:

```
CallStream
(
  origin: integer
  called: integer
  duration: integer
)
```

origin	SUM(duration)
0176/875611	1347
0160/4297855	65
0631/765432	23460
⋮	⋮

Realisierung in jeder vorgestellten Sprache:

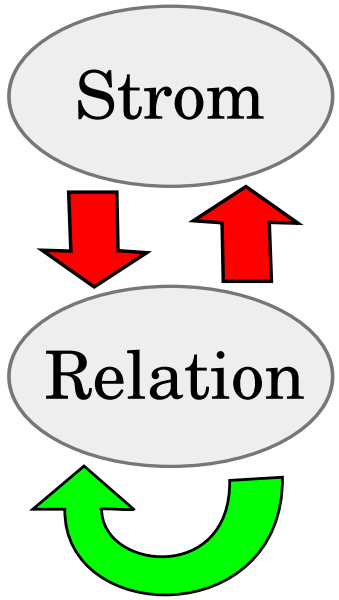
Liste die Gesamtdauer der langen Gespräche nach Anrufer gruppiert auf.

Merkmale der Continuous Query Language

CQL

- deklarative Sprache, basiert auf SQL
- steht für Continuous Query Language
- implementiert im DSVS *STREAM*
- unterstützt Relationen und Ströme.

Ziele: Verständlichkeit, exakte Semantik



Konzepte der Continuous Query Language

Zeit

- diskreter, geordneter Zeitbereich $\mathfrak{T}$
- oft Zeitstempel verwendet

Konzepte der Continuous Query Language

Zeit

- diskreter, geordneter Zeitbereich $\mathfrak{T}$
- oft Zeitstempel verwendet

Strom

- Folge von Tupeln aus Attributschema versehen mit Zeitstempeln $t \in \mathfrak{T}$
- Neuere Tupel haben nie kleinere Zeitstempel als ältere.

Konzepte der Continuous Query Language

Zeit

- diskreter, geordneter Zeitbereich $\mathfrak{T}$
- oft Zeitstempel verwendet

Strom

- Folge von Tupeln aus Attributschema versehen mit Zeitstempeln $t \in \mathfrak{T}$
- Neuere Tupel haben nie kleinere Zeitstempel als ältere.

Relation

- Funktion von $\mathfrak{T}$ in Multimenge von Tupeln
- Relationen sind nur zu den Zeitpunkten in $\mathfrak{T}$ definiert.

Eine einfache CQL-Anfrage

Aufgabe: Filtere die langen (mehr als 60 Sek) Gespräche aus dem Telefonstrom.

```
SELECT RStream(*)
FROM CallStream [Now]
WHERE duration > 60
```

Eine einfache CQL-Anfrage

Aufgabe: Filtere die langen (mehr als 60 Sek) Gespräche aus dem Telefonstrom.

```
SELECT RStream(*)
FROM CallStream [Now]
WHERE duration > 60
```

Beobachtung

Selbst für einfachen Filter drei Operatoren.

`SELECT RStream(*) FROM CallStream [Now] WHERE dura...`

$t \in \mathfrak{T}$	`CallStream`	`[Now]`	`RStream(*)`
1	$\langle 1, (o_1, d_1, 50)\rangle$	$(o_1, d_1, 50)$	-
2	$\langle 1, (o_1, d_1, 50)\rangle$ $\langle 2, (o_2, d_2, 85)\rangle$ $\langle 2, (o_3, d_3, 12)\rangle$	$(o_2, d_2, 85)$ $(o_3, d_3, 12)$	$\langle 2, (o_2, d_2, 85)\rangle$
3	$\langle 1, (o_1, d_1, 50)\rangle$ $\langle 2, (o_2, d_2, 85)\rangle$ $\langle 2, (o_3, d_3, 12)\rangle$ $\langle 3, (o_4, d_4, 62)\rangle$	$(o_4, d_4, 62)$	$\langle 2, (o_2, d_2, 85)\rangle$ $\langle 3, (o_4, d_4, 62)\rangle$
⋮	⋮	⋮	⋮

Das Telefonbeispiel mit CQL realisiert

```
SELECT origin, SUM(duration)
FROM CallStream [Range Unbounded]
WHERE duration > 60
GROUP BY origin
```

Das Telefonbeispiel mit CQL realisiert

```
SELECT origin, SUM(duration)
FROM CallStream [Range Unbounded]
WHERE duration > 60
GROUP BY origin
```

Mögliche Erweiterung durch Relation

```
PhoneUser = (phoneNumber, name)
```

```
SELECT name, origin, SUM(duration)
FROM CallStream [Range Unbounded], PhoneUser
WHERE duration > 60
  AND origin = phoneNumber
GROUP BY origin
```

Bewertung

Vorteile:

- einfache Syntax für Kenner von SQL
- Optimierung durch das System
- Unterstützung von (statischen) Relationen

Bewertung

Vorteile:

- einfache Syntax für Kenner von SQL
- Optimierung durch das System
- Unterstützung von (statischen) Relationen

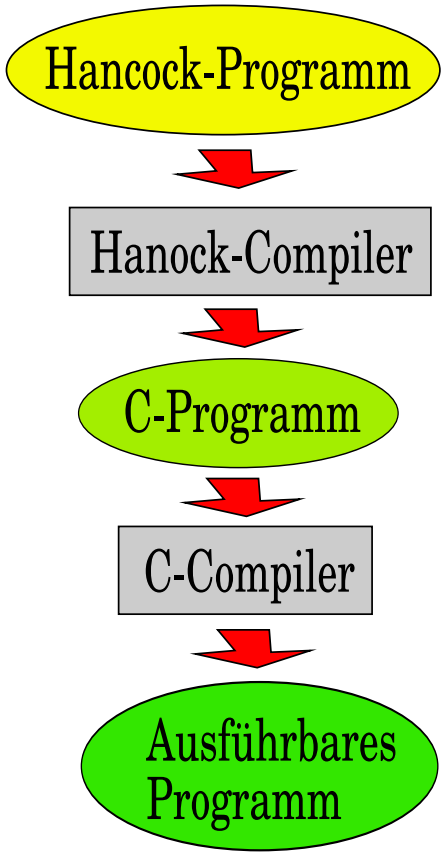
Nachteile:

- geringe Transparenz der Implementierung
- unendlich große Fenster?

Merkmale der Sprache Hancock

Hancock

- imperative Sprache
- entwickelt und verwendet bei AT&T (Telekommunikation)
- erweitert C-Programme um Konstrukte zur Stromverabeitung.
- bietet bessere Wartbarkeit und Lesbarkeit als C.



Einsatz von Hancock

- Mit Hancock formuliert man sogenannte *Signaturprogramme*.
- Signaturprogramme konsumieren Stromausschnitte und aktualisieren Signaturen
- Signaturen speichern Informationen über Ströme. (Anfrageergebnisse)

Einsatz von Hancock

- Mit Hancock formuliert man sogenannte *Signaturprogramme*.
- Signaturprogramme konsumieren Stromausschnitte und aktualisieren Signaturen
- Signaturen speichern Informationen über Ströme. (Anfrageergebnisse)
- Abwechselnd:
  1. Stromausschnitt ansammeln
  2. Signaturprogramme auf dem Ausschnitt ausführen

Einsatz von Hancock

- Mit Hancock formuliert man sogenannte *Signaturprogramme*.
- Signaturprogramme konsumieren Stromausschnitte und aktualisieren Signaturen
- Signaturen speichern Informationen über Ströme. (Anfrageergebnisse)
- Abwechselnd:
  1. Stromausschnitt ansammeln
  2. Signaturprogramme auf dem Ausschnitt ausführen

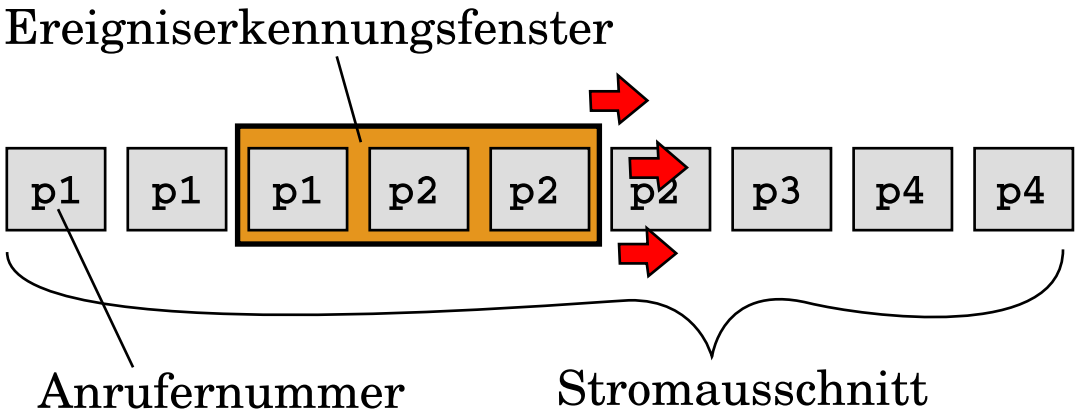
Zunächst:

Wie arbeiten Signaturprogramme konzeptionell?

Arbeitsweise eines Signaturprogramms

Vorgehensweise

1. Stromausschnitt sortieren
2. mit Ereigniserkennungsfunktion abtasten
3. dabei auftretende Ereignisse behandeln



Das Multi-Union-Konstrukt

Ein Multi-Union-Objekt

- repräsentiert eine Teilmenge einer Ereignismenge.

```
munion PhoneEvents
{:
  CallStreamElem call,
  int newOriginNumber,
  int endOriginNumber
:};
```

Das Map-Konstrukt

Eine Hanock-Map

- dient der Speicherung von Signaturen unter je einem Schlüssel.
- liegt meistens nicht komplett im Hauptspeicher.

```
map DurationSumMap {
  key (MINVALPN .. MAXVALPN);
  value int;
  default 0;
  compress frequencySqueeze;
  decompress frequencyUnsqueeze;
};
```

```
void sig_main(CallStream stream, DurationSumMap map)
{

  iterate ( over stream
            sortedby origin
            filteredby(c) (c->duration > 60)
            withevents originDetect )
  {









  };
}
```

```
void sig_main(CallStream stream, DurationSumMap map)
{
  int durationSum;
  iterate ( over stream
            sortedby origin
            filteredby(c) (c->duration > 60)
            withevents originDetect )
  {









  };
}
```

```
void sig_main(CallStream stream, DurationSumMap map)
{
  int durationSum;
  iterate ( over stream
            sortedby origin
            filteredby(c) (c->duration > 60)
            withevents originDetect )
  {
    event newOriginNumber(int number) {
      durationSum = 0; }

    event call(CallStreamElem e) {
      durationSum = durationSum + e->duration; }

    event endOriginNumber(int number) {
      map<:number:> = durationSum + map<:number:>; }
  };
}
```

Bewertung

Vorteile:

- volle Transparenz der Verarbeitung
- Optimierung durch Benutzer möglich
- intuitiv für Kenner von C/C++

Bewertung

Vorteile:

- volle Transparenz der Verarbeitung
- Optimierung durch Benutzer möglich
- intuitiv für Kenner von C/C++

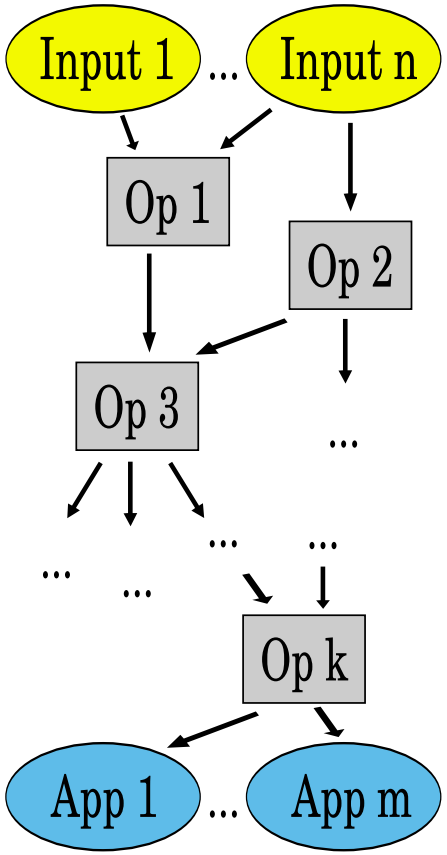
Nachteile:

- Programmierkenntnisse erforderlich
- Anfragen schwer verständlich für Laien

Merkmale von Aurora

Aurora

- eigentlich keine Sprache, sondern DSVS
- Laufzeitsystem + GUI zur Anfragespezifikation
- reine Stromverarbeitung, keine Relationen



Konzepte in Aurora

- Operatorknoten und Datenflusspfeile bilden gerichteten azyklischen Graphen
- an Ausgang eines Operators können beliebig viele Pfeile angeschlossen werden
- Anschluss von Ad-Hoc-Anfragen an sog. *Connection Points*
- Dienstgütespezifikation am Ende eines Pfades möglich

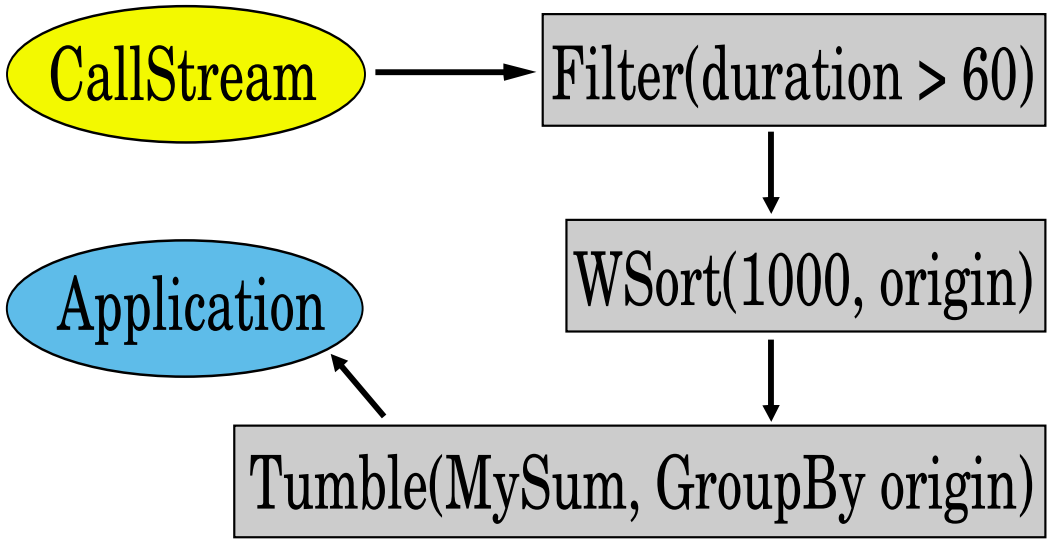
Operator Tumble in Aurora

Tumble(<func>, GroupBy <attribute list>)

Tumble

- wird durch Funktion und Attributmenge **A** parametrisiert
- baut mit ankommenden Tupeln Fenster auf
  - bis neues Tupel sich in **A** unterscheidet
  - dann Anwendung der Funktion auf das Fenster
- Funktion selbst definierbar

Telefonanfrage in Aurora realisiert

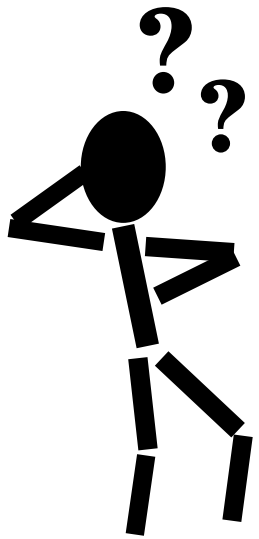


Problem

keine direkte Unterstützung von Relationen

Zusammenfassung

- zahlreiche Sprachansätze
- noch keine Universallösung
- CQL (deklarativ)
  - gewohnte Syntax, exakte Semantik
  - Realisierung wenig transparent
- Hancock (imperativ)
  - volle Kontrolle, C-Bibliotheken
  - kompliziert, keine automatische Optimierung
- Aurora (graphorientiert)
  - komplett neuer Ansatz, Dienstgüte spezifizierbar
  - keine Relationen, automatische Anfragen?



Danke für die Aufmerksamkeit. Fragen?