Datenbanksysteme

Wintersemester 2016/17

Prof. Dr.-Ing. Sebastian Michel
TU Kaiserslautern

smichel@cs.uni-kl.de

Join-Berechnung: Problemstellung

Wir betrachten einen Join zwischen Tabellen S und T.

- Wie kann garantiert werden, dass ein S Tupel in einem Block (Split) der S Datei mit allen T Tupeln verknüpft wird, die irgendwo in der T Datei sein könnten?

- Einfachster Fall. Natürlicher- oder Equi-Join anhand Attribut A.

- **Idee 1:** Sende alle Tupel mit dem gleichen A Wert zum gleichen Reduce-Task, welcher dann die Verknüpfung ausführt.
- **Idee 2:** Stelle T auf allen an der Berechnung beteiligten Maschinen zur Verfügung. Map-Tasks lesen S Tupel und greifen auf die lokale Kopie von T zu.

Idee 1: Reduce-Side Join

- Offensichtlich muss geschaut werden welche Tupel aus T und S verbunden (gejoint) werden können.
- Im Fall des Natürlichen-Joins ist dies besonders einfach: Ergebnisse müssen, hier im Beispiel, im Attribut A übereinstimmen, also $T.A = S.A$
- Analog für Equi-Joins.

- Wie kann man erreichen, dass Tupel mit gleichem Attributwert für A auf der gleichen Maschine landen?

Reduce-Side Join (Equi-Join)

Map

- Sende Tupel t zu Reducer anhand Schlüssel $t.A$
- Zusätzlich zum Tupel wird auch noch mitgeschickt ob t aus T oder aus S ist. Wieso?

Reduce

- Joine Tupel t_1, t_2 falls $t_1.A = t_2.A$ und t_1 aus T ist und t_2 aus S.

Reduce-Side Join (Equi-Join)

```
map(...., tuple x) {
    if (x is from S)
        emit (x.A, (x, 'S'))
    else // x is from T
        emit (x.A, (x, 'T'))
}
```

```
reduce(A-value, [[(x1, flag1), (x2, flag2), ...]) {
    initialize S_list and T_list

    for all (x, flag) in input do
        if (flag=='S')
            S_list.add(x)
        else
            T_list.add(x)

    for each s in S_list
        for each t in T_list
            emit(NULL, (s,t))
}
```

Diskussion

- **Der Reduce-Side-Join Algorithmus vermeidet es, Paare von nicht verknüpfbaren Tupel zu testen.**
- Dies ist sehr ähnlich zum Hash-Join Algorithmus.
- Er gruppiert die Eingabe nach dem Joinattribut, dann werden Tupel in der gleichen Gruppe verknüpft.

- Leider gibt es mehrere Nachteile...

Diskussion

- **Schlechte Lastverteilung bei schiefer Verteilung der Joinattributwerte**: Alle Tupel mit dem gleichen Joinattributwert müssen im gleichen Reduce-Aufruf bearbeitet werden.
- **Skaliert nicht für Joinattribute mit kleiner Anzahl verschiedener Werte**: Bei k verschiedenen Werten gibt es höchtens k Partitionen.
- Die Idee des Joinattributes als Schlüssel **funktioniert nicht bei anderen Joinbedingungen**, z.B. Ungleichheit ($S.A < T.A$) oder Band-Joins ($|S.A - T.A| < \varepsilon$).

Idee 2: Map-Only-Join ("Replicated Join")

- **Datei T wird mithilfe eines verteilten Dateipuffers (DistributedCache) an alle Knoten verteilt.**
- Eine "setup" Funktion des Map-Tasks lädt T aus dem Puffer und erzeugt eine **lokale Datenstruktur**, z.B. einen Hash-Index

- Die Map-Funktion bearbeitet nur S als Eingabe: Für jedes gelesene S Tupel wird im Index nachgeschaut, um alle passenden T Tupel zu finden.

Idee 2: Map-Only-Join ("Replicated Join")

```
Class Mapper {
    //Index H maps a join attribut value to all T tuples with that value
    hashIndex H

    setup() {
        H = new hashMap
        for each tuple t in DistributedCache
            H.insert(t.A, t)
    }

    map(..., S-tuple s) {
        for each tuple t in H.lookup(s.A) do
            emit(NULL, (s,t))
    }

    cleanup() { clean up H}
}
```

Diskussion

Der Replicated-Join scheint alle Probleme des Reduce-Side-Joins zu beheben:

- Schlechte Lastverteilung bei schiefer Verteilung
- Skaliert nicht für Joinattribute mit kleiner Anzahl versch. Werte
- Beschränkung auf Equi-Joins
- Doppelter Transfer von S und T

D.h. der Replicated-Join ist perfekt?

Diskussion

- Der Replicated-Join kann insgesamt mehr Daten über das Netzwerk übertragen als der Reduce-Side-join
  - Replicated-Join:
    - Liest $|T|$, sendet $n \times |T|$ zu den n Mapper-Maschinen
    - Liest $|S|$
  - Reduce-Side-Join
    - Liest $|S| + |T|$
    - Sendet $|S| + |T|$ von den Mappern zu den Reducern.
- Datensatz T sollte in den Speicher passen
  - Ansonsten sind teure Festplattenzugriffe nötig

Diskussion

- Insgesamt bietet sich der Replicated-Join nur dann an, wenn T viel kleiner als S ist und idealerweise komplett in den Arbeitsspeicher passt.
- Allerdings hat der Replicated-Join einen großen Vorteil gegenüber dem Reduce-Side-Join, da er **jeden Theta-Join** berechnen kann!
- Aber, können wir noch bessere Algorithmen für Theta-Joins finden?

Größe von S	**Größe von T**	**Equi-Join**	**Theta-Join**
Klein	Klein	Einfach	Einfach
Klein	Groß	Replicated-Join (Reduce-Side-Join)	Replicated-Join
Groß	Klein	Replicated-Join (Reduce-Side-Join)	Replicated-Join
Groß	Groß	Reduce-Side-Join (Probleme bei schiefen Verteilungen, wenigen versch. Werten)	?????

Problem Definition

- **Theta-Join** generalisiert den Equi-Join: Für $S = \{s_1, s_2, \ldots\}$ und $T = \{t_1, t_s, \ldots\}$, finde alle Paare (s_i, t_j), die ein gegebenes **Prädikat** $\theta(s_i, t_j)$ erfüllen.

- Typische Beispiele: Ungleicheitsbedingungen oder Distanzen unter Schwellwert

- **Ziel:** Berechnung des Theta-Joins so auf Knoten zu verteilen, dass Antwortzeit (d.h. von Job Start bis Ende) minimiert wird.