Fachpraktikum “Erweiterbare Datenbanksysteme” im WS 2015/16

Aufgabe 4 der Phase 2

Anfrageoptimierung für verteilte Datenbanken auf Basis der Distributed Algebra

Ralf Hartmut Güting, Thomas Behr, Fabio Valdés, Holger Helmut Hennings

19.11.2015

Lehr gebiet Datenbanksysteme für neue Anwendungen
Fakultät für Mathematik und Informatik, Fernuniversität in Hagen

1 Einleitung

Es ist möglich, mehrere Secondo-Systeme auf einem Rechnercluster zu verwenden, um verteilte Datenbanken aufzubauen und Anfragen darauf auszuwerten. Dabei spielt ein Secondo-System die Rolle eines *Masters*, der eine Reihe von Secondo-Servern als *Worker* verwendet. Der Master verteilt Daten an die Worker oder sammelt sie von ihnen ein; er beauftragt die Worker, Berechnungen auszuführen und läßt sich von ihnen die Ergebnisse liefern.

Grundlage dafür ist die *Distributed Algebra* (genauer: ihre 2. Version, die *Distributed2Algebra*), die für den Master Datentypen bereitstellt, um verteilte Datenobjekte zu beschreiben, sowie Operationen, um verteilte Daten zu erzeugen und zu bearbeiten. Mit Hilfe der Distributed Algebra kann man also verteilte Anfrageauswertung auf der ausführbaren Sprachebene von Secondo durchführen.

Das Ziel dieser Aufgabe ist es, verteilte Anfrageauswertung in SQL zu ermöglichen. Damit kann man es dem Benutzer ersparen, Operationen der Distributed Algebra zu erlernen und komplexe Ausführungspläne zu schreiben. Für bestimmte Klassen von Queries in Secondo-SQL, die bisher für Datenbanken auf einem einzelnen Server unterstützt wurden, soll nun Auswertung auf einer verteilten Datenbank ermöglicht werden. Dazu ist der Secondo-Optimierer zu erweitern.

2 Beispieldatenbank

Wir verwenden für Beispiele die Datenbank NRW, die von OpenStreetMap bzw. GeoFabrik[1] in Form von Shapefiles heruntergeladen und in Secondo importiert werden kann. Mit Auswertungen auf dieser Datenbank ist auch die Lösung der Aufgabe zu demonstrieren. Die Datenbank enthält acht Klassen von Objekten mit Raumbezug in Nordrhein-Westfalen, nämlich

- Roads, ca. 1,5 Millionen.
- Waterways
- Railways

1. http://download.geofabrik.de/europe/germany/nordrhein-westfalen-latest.shp.zip

- Points
- Places
- Natural
- Buildings, ca. 6 Millionen
- Landuse

Diese werden jeweils durch Sachattribute mit Standarddatentypen und geometrischen Attributen der Typen *point*, *line*, oder *region* beschrieben. Die Relation Roads hat z.B. das Schema:

```
Roads(Osm_id: ... , Name: ..., ..., GeoData: line)
```

Die Datenbank kann in Secondo aufgebaut werden, indem man die Datei `nordrhein-westfalen-latest.shp.zip` entpackt und sie ins Verzeichnis `secondo/bin` bringt. Anschließend kann man das Skript `nrwImportShape` aus dem verzeichnis `secondo/bin/Scripts` laufen lassen, um die Datenbank z.B. auf dem Master aufzubauen.

3 Verteilte Anfrageauswertung mit der Distributed Algebra

Die Distributed Algebra[1] benutzt als Worker eine Reihe von Secondo-Instanzen (Secondo-Servern), die auf dem gleichen oder anderen Rechnern laufen. Bevor die Algebra verwendet werden kann, müssen auf den beteiligten Rechnern Secondo-Monitore gestartet werden. Unter welchen IP-Adressen und welchen Ports Secondo-Server gestartet werden können, wird für die Algebra in einer Relation beschrieben, die bestimmten Operationen als Parameter mitgegeben wird.

Die Distributed Algebra besitzt Operationen auf zwei Ebenen:

- Die *untere Ebene* bietet primitive Operationen an, mit denen man z.B. einen oder mehrere Server starten kann, die anschließend jeweils unter einer Servernummer ansprechbar sind. Man kann Servern Secondo-Befehle oder Anfragen schicken, die dann parallel ausgeführt werden. Die folgende obere Ebene ist mit Hilfe der unteren Ebene implementiert.
- Die *obere Ebene* bietet die Abstraktion eines *verteilten Arrays* als Datentyp an. Die Felder des Arrays sind über die verfügbaren Server verteilt und können jeweils Objekte beliebiger Secondo-Datentypen enthalten. Ein Feld kann also z.B. eine Relation, eine Indexstruktur oder ein atomares Objekt, z.B. eine Zahl oder ein Rechteck enthalten.
- Auf der oberen Ebene gibt es Operationen, die
  - einen verteilten Array erzeugen, indem Daten vom Master verteilt werden;
  - auf jedem Feld eines verteilten Arrays von dem zuständigen Server eine Secondo-Query auswerten und das Ergebnis in einem neuen verteilten Array speichern;
  - zwei verteilte Arrays *R*, *S* als Argumente nehmen und auf jedem Paar $R[i]$, $S[i]$ eine Secondo-Query auswerten; damit lassen sich Joins auswerten, falls *R* und *S* passend partitioniert sind.
  - verteilte Arrays neu partitionieren (z.B. mittels Hash-Funktion auf einem Attribut), wichtig für Joins
  - Inhalte von verteilten Arrays auf den Master transportieren und dort aggregieren.

1. In diesem Text ist damit immer die Distributed2Algebra gemeint.

- Es gibt eine Variante eines verteilten Arrays, den *distributed file array.* In seinen Feldern können nur Relationen gespeichert werden, nicht beliebige Secondo-Typen. Diese Relationen werden verteilt in Dateien gespeichert. Solche Arrays können insbesondere für Zwischenergebnisse in der Anfrageauswertung verwendet werden, da Dateien effizienter ohne Transaktionskontrolle geschrieben werden können und sie zum Datentransport zwischen Rechnern bzw. Secondo-Servern verwendet werden.
- Die Worker speichern die erhaltenen Daten in einer Datenbank gleichen Namens wie auf dem Master und verwenden Objektnamen mit Suffixen, um zu kennzeichnen, zu welchem Feld des verteilten Arrays dieses Objekt gehört. So kann z.B. `Roads_11` das Objekt *Roads*[11] des Masters bezeichnen.

Genauere Informationen zur Distributed2Algebra erhält man wie üblich mittels

```
list algebra Distributed2Algebra
```

Darüber hinaus wird die Benutzung in [1] erklärt.

4 Verteilte Datenbanken

Eine Relation kann in einer verteilten Datenbank grundsätzlich auf zwei Arten verteilt gespeichert werden, nämlich (i) repliziert oder (ii) partitioniert.

Replizierte Speicherung bedeutet, dass jeder Worker in seiner Datenbank eine vollständige Kopie der Relation hat. Falls m Worker auf n Datenbanken arbeiten ($m \geq n$) wird die Datenmenge also mit n multipliziert. Replizierte Speicherung kann so erfolgen, dass einfach jede Worker-Datenbank eine Kopie der Relation enthält. Alternativ könnte man auch einen verteilten Array verwenden, in dem alle Felder den gleichen Inhalt haben.

Partitionierte Speicherung bedeutet, dass die Relation in disjunkte Teilmengen von Tupeln zerlegt wird. Dabei gibt es mehrere Möglichkeiten, wie die Zerlegung vorgenommen wird.

(1) *Zufällig*. Es ist unabhängig von den Attributen eines Tupels, in welches Feld es gelangt. Tupel können z.B. round-robin verteilt werden oder sequentiell, indem jeweils ein Feld aufgefüllt wird, bis es eine gegebene Tupelzahl erreicht hat.

Ein Join kann auf einer so partitionierten Relation nur gegen eine replizierte Relation durchgeführt werden. Jede Partition (jedes Feld) wird also mit der vollständigen anderen Relation verglichen. Für allgemeine, beliebig komplexe Join-Bedingungen, die nur durch Vergleich aller Paare von Tupeln auszuwerten sind, ist das in Ordnung und muss auch so gemacht werden. Jede andere Partitionierung eignet sich dafür natürlich genauso.

(2) *Mittels Hash-Funktion über (Standard-) Attributen*. So kann z.B. mit dem Ausdruck

```
hashvalue(.Name, 999997) mod 50
```

eines der Felder 0, ..., 49 ausgewählt werden, dem das Tupel dann zugeordnet wird. Hier ist $p = 50$ die Anzahl der Partitionen, also der Felder des verteilten Arrays.

Solche Partitionierungen für zwei Relationen R und S mit Partitionierungsattributen $R.A$ und $S.B$ erlauben anschließend einen verteilten Equijoin mit der Bedingung $R.A = S.B$, indem für alle Par-

titionen $i = 0, ..., 49$ der Equijoin auf den Feldern $R[i]$ und $S[i]$ ausgeführt wird. Dies ist korrekt, da gleiche Attributwerte in gleiche Felder gelangen.

(3) *Räumliche Verteilung über geometrischen Attributen*. Hier wird ein regelmäßiges Gitter *G*, vergleichbar einem Schachbrett, in zwei oder drei Dimensionen zugrunde gelegt, dessen Zellen fortlaufend nummeriert sind. Für einen Wert *v* eines geometrischen Datentyps (z.B. Punkt, Linienzug oder Gebiet) wird zunächst die Bounding Box *BB*(*v*), das kleinste achsenparallele umschließende Rechteck, gebildet. Dieses fällt gewöhnlich in eine Zelle des Gitters *G*; im Allgemeinen kann es mehrere Zellen überlappen. Dies ist in Abbildung 1 illustriert.

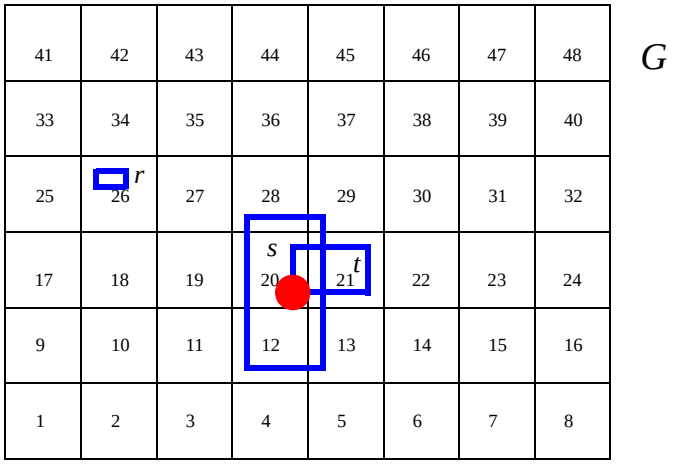

Abbildung 1: Die Rechtecke *r*, *s*, *t* werden mit Hilfe des Gitters *G* auf Zellnummern abgebildet.

Seien $c_1, ..., c_k$ die Zellnummern von *BB*(*v*) überlappter Zellen. Dann wird das Tupel, das *v* enthält, auf die Partitionen

```
c_1 mod p, ..., c_k mod p
```

abgebildet, wobei *p* wieder die Anzahl der Partitionen ist. Das heißt, es werden *k* Kopien des Tupels hergestellt und jede Kopie wird in eines der *k* Felder geschrieben.

In Secondo gibt es Datentypen für Gitter und Operationen, die eine solche Abbildung unterstützen. Im folgenden Secondo-Befehl werden Straßen mit Hilfe eines Gitters verteilt.

```
let RoadsB1 = Roads feed
  extendstream[Cell: cellnumber(bbox(.geoData), grid)]
  ddistribute[Cell, 50, Worker]
```

Hier verwendet der Operator *cellnumber* ein Rechteck und ein Gitterobjekt *grid* und liefert einen Strom von Zellnummern im Gitter, die vom Rechteck überdeckt werden. Der *extendstream*-Operator erzeugt eine Kopie des Eingabetupels für jede Zellnummer und speichert diese in der Kopie im neuen Attribut *Cell*. Der *ddistribute*-Operator erzeugt den verteilten Array, indem er für jedes empfangene Tupel den Wert von Cell modulo 50 nimmt und das Tupel in dieses Feld einordnet. Worker ist die Relation, die die zu verwendenden Secondo-Server definiert.

Räumliche Partitionierung unterstützt den raumbezogenen Verbund (Spatial Join). Ein Spatial Join über zwei Relationen *R* und *S* mit geometrischen Attributen *A* und *B* bildet alle Paare von Tupeln $r \in R$ und $s \in S$, für die gilt, dass die Bounding Boxen von *r.A* und *s.B* sich überlappen. Man kann den verteilten Spatial Join wiederum auf Paaren von Feldern $R[i]$ und $S[i]$ zweier ver-

teilter Arrays *R* und *S* ausführen. Dies ist korrekt, weil zwei Tupel, deren Bounding Boxen überlappen, auf dieselbe Gitterzelle und damit auf dieselbe Partition abgebildet werden.

Es ist allerdings möglich, dass zwei zu verbindende Tupel mehrfach gefunden werden. In Abbildung 1 treffen sich die Bounding Boxen *s* und *t* sowohl in der Zelle 20 wie auch in Zelle 21. Um Duplikate im Ergebnis des Spatial join zu vermeiden, gibt es einen Operator *gridintersects*, der zwei Rechtecke anhand einer Gitterdefinition und einer Zellnummer vergleicht. Der Operator prüft, ob die beiden Rechtecke sich schneiden. Falls sie sich schneiden, gibt es einen kleinsten gemeinsamen Punkt. Dieser kann nur in einer Zelle liegen (Zellränder gehören jeweils nur zu einer Zelle). Nur für die Zelle, in der der kleinste gemeinsame Punkt liegt, liefert *gridintersects* TRUE. In Abbildung 1 ist der kleinste gemeinsame Punkt von *s* und *t* in Zelle 20 gezeigt; nur für die Partition, die Zelle 20 enthält, wird das Tupelpaar ins Ergebnis gelangen.

5 SQL-Anfragen

5.1 Grundsätzlich unterstützte Arten von Anfragen

In dieser Aufgabe soll Optimierung für die Auswertung von Selektionen und Joins auf verteilten Relationen realisiert werden. Das bedeutet:

1. Der Fokus liegt auf der Behandlung von Selektionen und Joins (sog. *conjunctive query optimization*). Im Wesentlichen sollen folgende Arten von Anfragen unterstützt werden:

   ```
   select count(*) from [R1, ..., Rn] where [P1, ..., Pm]
   ```

   ```
   select * from [R1, ..., Rn] where [P1, ..., Pm]
   ```

   Dabei sind die R_i Relationen und die P_i jeweils Selektions- oder Joinbedingungen. Die Ergebniszahl ist auf dem Master anzuzeigen; die Ergebnismenge von Tupeln soll auf dem Master erscheinen.

   Als Erweiterungsmöglichkeit kann man Projektionen in der select-Klausel und weitere Konstrukte wie *groupby, orderby* anbieten, sofern man die Optimierung dafür problemlos von der Standardoptimierung übernehmen kann. Das heißt, die Umsetzung dieser Sprachelemente braucht erst nach dem Zurückliefern der Tupel an den Master zu erfolgen und muss nicht in die verteilte Optimierung eingehen.

2. In der Aufgabe werden nur Anfragen behandelt, in denen alle Relationen verteilt (partitioniert oder repliziert) vorliegen. Falls zusammen mit verteilten Relationen solche erwähnt werden, die nur lokal auf dem Master liegen, ist eine Fehlermeldung zu erzeugen. Falls in der Anfrage nur lokale (nicht verteilte) Relationen erwähnt werden, soll die Standardoptimierung zum Zuge kommen.

   Allerdings soll für jede verteilte Relation die gleiche Relation auch auf dem Master vorhanden sein, möglicherweise reduziert auf Sample-Größe. Die Idee dabei ist, dass das Überprüfen der Anfrage in Bezug auf das Relationenschema und die Bestimmung von Selektivitäten von der Standardoptimierung übernommen werden kann.

Um die lokale Relation und ihre verteilte Version unterscheiden zu können, soll letztere mit dem Suffix "_d" (für distributed) versehen werden. Auf dem Master gäbe es also z.B. die Relation *Roads*; in der SQL-Anfrage würde man sich mit *Roads* auf diese und mit *Roads_d* auf die verteilte Version beziehen.

5.2 Konkrete Anfragetypen

Konkret sollen folgende Anfragetypen unterstützt werden:[1]

1. Selektion mit einem Standardattribut
   - ohne Index
   - mit Index

   Beispiel:

   ```
   select count(*) from Roads_d where Name = "Universitätsstraße"
   ```

   Für die Implementierung mit Index müsste auf jeder Partition ein B-Baum-Index vorhanden sein.

2. Selektion mit geometrischem Attribut
   - ohne Index
   - mit Index

   Beispiel:

   ```
   select * from Buildings_d where GeoData intersects eichlinghofen
   ```

   Hier ist *eichlinghofen* ein Objekt vom Typ *region*, das das Gebiet des Stadtteils Eichlinghofen von Dortmund grob beschreibt. Achtung: Hier muss die Optimierung das Objekt *eichlinghofen* ggf. an die Server-Datenbanken verteilen. Für die Optimierung mit Index müsste ein R-Baum auf jeder Partition existieren.

3. Equijoin über Standardattributen (ohne Index)

   Beispiel: Finde Paare verschiedener Objekte der Klasse Natural, die gleich benannt sind.

   ```
   select * from [Natural_d as n1, Natural_d as n2]
   where [n1:Name = n2:Name, n1:Osm_id < n2:Osm_id]
   ```

4. Spatial Join (ohne Index)

   Beispiel:

   ```
   select count(*) from [Roads_d as r, Waterways_d as w]
   where r:GeoData intersects w:GeoData
   ```

Als Erweiterungsmöglichkeiten könnte Folgendes realisiert werden:

5. Indexbasierter Equijoin
6. Indexbasierter Spatial Join

---

1. Die hier gezeigten Anfragebeispiele könnten so an der Javagui eingegeben werden, die für den Optimierer große Anfangsbuchstaben in Kleinbuchstaben umwandelt. In der Prolog-Umgebung, etwa mit SecondoPL, müssen Relationen- und Attributnamen klein geschrieben werden.

7. Allgemeiner Join mit beliebigen Bedingungen zwischen einer partitionierten und einer replizierten Relation

   Beispiel:

   ```
   select * from [Roads_d as r1, Roads_d as r2]
   where r1:Name contains r2:Name
   ```

8. Attribute und Ausdrücke in der select-Klausel, die bereits verteilt ausgewertet werden.

   Beispiel: Berechne die Positionen von Schnittpunkten zwischen Straßen und Wasserwegen, also vermutlich von Brücken.

   ```
   select [r:Osm_id, r:Name, w:Osm_id, w:Name,
     intersection(r:GeoData, w:GeoData) as BridgePosition]
   from [Roads_d as r, Waterways_d as w]
   where r:GeoData intersects w:GeoData
   ```

9. Verteiltes Auswerten von Gruppieren und Aggregatfunktionen

   Beispiel:

   ```
   select [Type, count(*) as Cnt]
   from Roads_d
   groupby Type
   ```

10. Abstandsbasierter Spatial Join mit verteilter Auswertung

    Beispiel: Finde alle Paare von Straßen und Gebäuden, deren Abstand kleiner ist als 500 m.

    ```
    select *
    from [Roads_d as r, Buildings_d as b]
    where distance(gk(r:GeoData), gk(b:GeoData)) < 500
    ```

    Hier wandelt der *gk*-Operator geometrische Koordinaten in Gauss-Krüger-Format um, wodurch Abstände in Metern entstehen. Abstandsbedingungen können über den Schnitt vergrößerter Bounding Boxen überprüft werden. Im verteilten Fall muss die Verteilung einer der beteiligten Objektklassen bereits vergrößerte Bounding Boxen benutzen, damit die Abstandsbedingung korrekt auf Partitionen ausgewertet werden kann.

6 Anforderungen an die Lösung

Der Optimierer soll seine Kenntnis über verteilte Relationen aus im Kern anzulegenden Systemrelationen beziehen (normale Relationen, die aber speziell benannt sind, etwa SEC2DISTRIBUTED und SEC2WORKERS). Ein “Systemadministrator” (also ein Benutzer in dieser Rolle) muss z.B. beim Anlegen einer verteilten Relation ein entsprechendes Tupel in einer solchen Relation hinzufügen.

Eine Erweiterungsmöglichkeit besteht darin, im Optimierer Syntax für das Anlegen verteilter Relationen zu implementieren. Dann könnte der Benutzer über den Optimierer eine verteilte Relation erzeugen, wobei der Optimierer automatisch die Systemrelation erweitert.

Der Optimierer muss überprüfen, ob die erwähnten Relationen existieren und dafür sorgen, dass die in einer Anfrage erwähnten atomaren Objekte der Master-Datenbank auch verteilt vorliegen oder jetzt verteilt werden.

Erweiterung: Der Optimierer überprüft vor Ausführung der Anfrage, ob die zu verwendenden Server laufen.

Kardinalitäten und Selektivitäten können auf dem Master ausgewertet werden.

Der Optimierer wird jede Selektion und jeden Join zunächst für sich übersetzen. Dabei können Ketten von verteilten Operationen entstehen, die man verschmelzen kann. Diese sollten in einer Nachbehandlung des erzeugten Plans tatsächlich verschmolzen werden.

Beispiel:

```
select * from Roads where [Type = "residential", Maxspeed > 30]
```

Die einzelnen Bedingungen könnten vom Standardoptimierer in Filteroperationen auf Tupelströmen übersetzt werden, als

```
filter[.Type = "residential"] filter[.Maxspeed > 30]
```

Im verteilten Fall würden solche Bedingungen in *dloop* oder *dmap*-Operationen übersetzt werden, also z.B.

```
dmap["", . filter[.Type = "residential"]] dmap["", . filter[.Maxspeed >
30]]
```

Dies kann verschmolzen werden zu

```
dmap["", . filter[.Type = "residential"] filter[.Maxspeed > 30]]
```

Allgemein kann eine Kette von Selektionen in eine einzige Selektion transformiert werden. Bei gemischten Plänen, in denen Selektionen und Joins vorkommen, können Selektionen mit nachfolgenden oder vorangehenden Joins verschmolzen werden.

Für Experimente, insbesondere zur Kostenermittlung, kann der Rechnercluster des Lehrgebiets mit 6 Rechnern, 12 Platten und 36 Kernen benutzt werden.

Referenzen

[1] R.H. Güting und T. Behr, Distributed Query Processing in Secondo. Using the Distributed Algebra 2. Secondo-Dokumentation, zu finden in `secondo/Algebras/Distributed2/Docu`, November 2015.