Fachpraktikum “Erweiterbare Datenbanksysteme” im WS 2017/2018

Aufgabe der Phase 2

Eine Algebra für verteilte Relationen

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

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

16.11.2017

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.

Die Formulierung solcher Anfragen ist allerdings eine anspruchsvolle Aufgabe. Ein Aspekt, der es schwierig macht, besteht darin, dass die Ausführung von Joins auf verteilten Relationen davon abhängig ist, dass die beiden Argumentrelationen auf gleiche Art partitioniert und verteilt sind. Darauf muss die Benutzerin selbst achten. Außerdem muss man sich mit den ausführbaren Operationen von Secondo gut auskennen, um z.B. die richtigen Joinmethoden zu wählen.

Das Ziel dieser Praktikumsaufgabe besteht darin, auf Basis der Distributed Algebra eine leichter zu benutzende Algebra für verteilte Relationen zu implementieren. Dazu wird ein neuer Datentyp für verteilte Relationen eingeführt, der sich vor allem die jeweils vorliegende Verteilung merkt und diese in Operationen berücksichtigt. Ggf. notwendiges Umverteilen von Argumentrelationen wird damit automatisch ausgeführt und braucht vom Benutzer nicht angegeben zu werden.

2 Grundlage: die 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 Worker eine Secondo-Query auswerten lassen 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.
- 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
```

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

Eine detaillierte Einführung in die Konfiguration eines verteilten Secondo-Systems und die Anfrageauswertung mit Hilfe der Distributed Algebra findet sich in [1].

3 Ziele

Die Algebra für verteilte Relationen soll die Formulierung von Anfragen im Vergleich zur direkten Benutzung der Distributed Algebra erleichtern. Dazu gehören folgende Aspekte:

1. Verteilungsarten werden verwaltet und bei der Formulierung von Joins berücksichtigt.
2. Manche Verteilungsarten (*range*, *spatial*) benötigen eine Datenstruktur, anhand derer Objekte (Tupel) verteilt werden. Für die *range*-Partitionierung ist dies eine Menge von Intervallen über einem eindimensionalen Wertebereich (Standarddatentyp); für räumliche Partitionierung ist es eine Menge von Rechtecken in 2D oder 3D. Diese Datenstrukturen werden anhand von Samples der zu verteilenden Relation automatisch berechnet.
3. Es werden Operationen für Selektion und Join angeboten, die automatisch geeignete Methoden für Joins benutzen und darüber hinaus selbständig entscheiden, ob ein ggf. vorhandener Index benutzt wird.
4. Für Selektionen und Joins wird der jeweils beste Plan anhand von Kostenschätzungen ermittelt.

4 Verteilte Relationen

4.1 Speicherungsformen

Verteilte Relationen können gespeichert werden als

- Verteilter Array (*darray*), dessen Felder Relationen enthalten, die in Datenbanken der Worker dargestellt sind.
- Verteilter Array (*dfarray*), dessen Felder Relationen enthalten, die in Dateien auf den Rechnern der Worker dargestellt sind.
- Verteilter Array (*darray*), dessen Felder Relationen enthalten, die im Hauptspeicher der Worker dargestellt sind.

Dazu kann es verteilte Arrays mit Indexen geben, nämlich

- für normale persistente Relationen: B-Bäume und R-Bäume
- für Hauptspeicherrelationen: AVL-Bäume und Hauptspeicher-R-Bäume (MMRtree).

Dabei indiziert z.B. der R-Baum in Feld 17 eines verteilten Arrays die Relation in Feld 17 des zugrundeliegenden verteilten Relationenarrays.

4.2 Verteilungen

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 erfolgt so, dass einfach jede Worker-Datenbank eine Kopie der Relation enthält.

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

(3) *Anhand eines eindimensionalen partitionierten Wertebereichs.* Grundlage ist eine geordnete Folge von Werten eines gegebenen Wertebereichs (z.B. *int*, *string*) $x_0, ..., x_n$ mit $x_0 = -\infty$, $x_n = +\infty$ (bzw. einem minimalen und einem maximalen Wert). Tupel werden anhand eines Attributwertes x mit $x_i \leq x < x_{i+1}$ in Feld i eines verteilten Arrays eingeordnet.

Auch hier können Equijoins über gleich partitionierten Relationen ausgeführt werden.

(4) *Räumliche Verteilung über geometrischen Attributen.*

Hier wird eine räumliche Partitionierung eines zwei- oder dreidimensionalen Raumes in Rechtecke bzw. Quader zugrundegelegt (Abbildung 1).

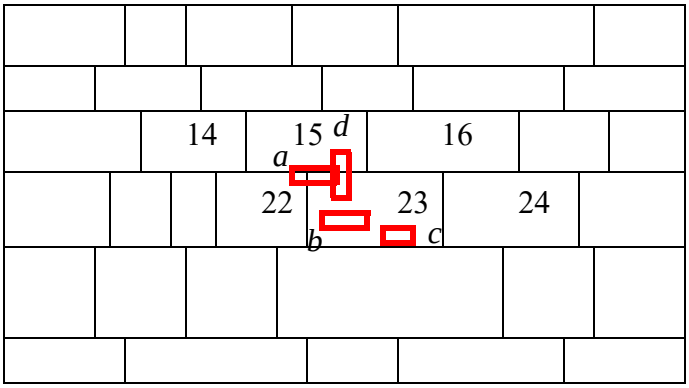

Abbildung 1: Eine räumliche Partitionierung in 2D und einige zugeordnete Rechtecke. Rechteck *a* fällt in die Felder 15, 22 und 23, *d* in Felder 15 und 23. Rechtecke *b* und *c* liegen nur in Feld 23.

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 der Partition; im Allgemeinen kann es mehrere Zellen überlappen. Dies ist in Abbildung 1 illustriert.

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

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 verteilter 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 *a* und *d* sowohl in Feld 15 wie auch in Feld 21. Eine Technik, um Duplikate im Ergebnis zu vermeiden, ist im Anhang beschrieben.

4.3 Typinformation

Der neue Datentyp für verteilte Relationen heißt *drel*. Seine Typinformation enthält

- den Tupeltyp der Relation

- einen Term, der den Verteilungstyp darstellt. Dazu werden folgende Symbole benutzt:
  - random
  - hash(<attrname>)
  - range(<attrname>, <key>)
  - spatial2d(<attrname>, <key>)
  - spatial3d(<attrname>, <key>)
  - replicated

  Für *spatial2d* muss der Attributname sich auf einen räumlichen Attributtyp wie *point*, *line*, *region*, *rect* beziehen. Für *spatial3d* muss es ein raumzeitlicher Typ wie *moving*(*point*) oder *moving*(*region*) sein. Für solche Typen werden Units für die Partitionierung zugrunde gelegt.

  Der *Key* ist eine Zufallszahl, die bei der Erzeugung einer Partitionierung (range oder spatial) mit erzeugt wird. Damit kann auf Typebene überprüft werden, ob zwei Relationen tatsächlich gleich partitioniert sind.

  Eine Relation mit Verteilungstyp *replicated* wird nicht als verteilter Array gespeichert, sondern über *share* (eine Operation der Distributed Algebra) allen Workern zur Verfügung gestellt.

- die Art der Speicherung, nämlich *rel* für eine normale Relation, *mrel* für eine Hauptspeicherrelation oder *file*, falls sie als distributed file array (dfarray) gespeichert ist.

4.4 Wertinformation

Der Wert einer *drel* auf dem Master enthält die Informationen, die auch ein verteilter Array enthält. Darüber hinaus:

- die Kardinalität der verteilten Relation
- für *range*- oder *spatial*-Partitionen den *Verteilungsindex*, eine Datenstruktur, die die Partitionierung darstellt. Also eine Folge x_0, ..., x_n für *range*-Partitionierung oder eine Menge von Rechtecken wie in Abbildung 1 für *spatial*-Partitionierung.

5 Operationen

Es gibt Operationen zum

- Verteilen von Relationen vom Master auf die Worker
- Verarbeiten verteilter Relationen auf den Workern
- Einsammeln von Relationen von den Workern auf den Master
- Anlegen und Löschen von Indexen
- Konvertieren

Operationen dieser Algebra haben im Allgemeinen den Präfix *dr* für *distributed relation*.

Syntax und Defaults

Grundsätzlich benötigen praktisch alle folgenden Operatoren zwei Parameter:

- Einen Objektnamen, der von Workern benutzt wird, um Objekte in ihrer Datenbank zu speichern. Z.B. "BuildingsR" wird angegeben; ein Worker speichert Feld 17 als "BuildingsR_17". Wird in der Distributed Algebra als Name ein Leerstring angegeben, so wird ein eindeutiger Objektname generiert. Solch Objekte gelten aber als temporär und werden bei gewissen Aufräumaktionen gelöscht. Man sollte also einen Objektnamen angeben, wenn das Objekt dauerhaft in der Datenbank gebraucht wird und ihn weglassen, wenn das Objekt nur für die aktuelle Query verwendet wird.
- Die Speicherungsform, also *rel*, *mrel* oder *file*.

Wir geben diese beiden Parameter einheitlich in einer ersten Teilliste in den eckigen Klammern eines Operators an. Beispiele:

```
let RoadsR = Roads drdistribute["RoadsR", rel; spatial2d, GeoData, 150]

query BuildingsR drselect["", rel; GeoData, hombruch; . filter[.GeoData
  inside hombruch]]
```

Default ist aber, dass der Name leer ist und dass die Speicherungsform eine persistente Relation ist. Deshalb kann man die beiden Befehle so schreiben:

```
let RoadsR = Roads drdistribute["RoadsR"; spatial2d, GeoData, 150]

query BuildingsR drselect[; GeoData, hombruch; . filter[.GeoData
  inside hombruch]]
```

Bei der Verteilungsoperation *drdistribute* wird ein Name für das Objekt benötigt; deshalb kann er hier nicht weggelassen werden.

5.1 Verteilen

drdistribute

Eine gegebene Relation auf dem Master kann verteilt werden anhand einer der Verteilungsarten aus Abschnitt 4.2.

```
drdistribute: rel(Tuple) x string [x storage] x distType [x Attrname]
  [x int] -> drel(Tuple, Distribution, Storage)

distType = {random, hash, range, spatial2d, spatial3d, replicated}
storage = {rel, mrel, file}
```

Dabei gibt das zweite Argument den Namen an, unter dem ein Feld von einem Worker gespeichert werden kann. Das dritte Element beschreibt die gewünschte Speicherungsform. Dabei ist *rel* der Default. Der Attributname wird benötigt, falls der distType einen der Werte hash, range, spatial2d or spatial3d annimmt. Das sechste Argument gibt die Anzahl der Felder des verteilten Arrays an bzw. die Anzahl der Partitionselemente; es entfällt bei *distType = replicated*.

Darüber hinaus gibt es die Möglichkeit, eine Relation ebenso zu verteilen wie eine bereits vorhandene verteilte Relation.

```
drdistribute: rel(Tuple) x string [x storage] x drel(...)
```

Beispiel: Nehmen wir an, auf dem Master gibt es eine Relation mit Straßen und eine mit Gebäuden:

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

Dann können wir sie räumlich verteilen:

```
let RoadsR = Roads drdistribute["RoadsR", rel; spatial2d, GeoData, 150]

let BuildingsR = Buildings drdistribute["BuildingsR", rel; RoadsR]
```

Dabei wird mit dem ersten Befehl eine räumliche Partitionierung mit 150 Feldern wie in Abbildung 1 anhand eines Samples der Straßen berechnet und die Straßen werden darauf verteilt. Dabei erhält jedes Tupel zwecks Duplikatvermeidung ein zusätzliches Attribut *TRClass* vom Typ *int*, wie im Anhang beschrieben. Das letzte Argument *rel* könnte weggelassen werden, da es der Default ist.

Mit dem zweiten Befehl wird die verteilte Relation *BuildingsR* mit der gleichen Verteilung wie *RoadsR* erzeugt. Ein Spatial Join zwischen den beiden wäre also ohne Umverteilen möglich.

Beim Berechnen von Partitionierungen (*range*, *spatial*) möchte man Elemente möglichst gleichmäßig verteilen. Für *range*-Partitionierung ist eine Strategie in [1] beschrieben. Für *spatial2d* kann man alle y-Koordinaten eines Samples gleichmäßig in $\sqrt{n}$ Abschnitte zerlegen. Pro Zeile kann man dann die dort vorhandenen *x*-Koordinaten gleichmäßig in ca. $\sqrt{n}$ Abschnitte zerlegen. Damit entsteht eine Partitionierung wie in Abbildung 1.

5.2 Verarbeiten

5.2.1 Abbildung

drmap

Mit *drmap* kann man auf jedes Feld einer verteilten Relation eine Funktion anwenden, die einen Tupelstrom liefert. Ergebnis ist eine verteilte Relation mit ggf. neuem Tupeltyp, aber normalerweise unverändertem Verteilungstyp.[1]

Es ist erlaubt, dass die Parameterfunktion ein Ergebnis anderen Typs liefert, z.B. *int*. In diesem Fall ist das Ergebnis ein *darray*. Dies gilt auch für weitere Parameterfunktionen z.B. in *select*- oder *join*-Operationen.

1. Es ist allerdings möglich, in der Parameterfunktion das Partitionierungsattribut herauszuprojizieren. In diesem Fall ändert sich der Verteilungstyp zu *random*. Es ist nicht erlaubt, in der Parameterfunktion den Wert des Partitionierungsattributs zu verändern, da es zu Fehlern führt. Dies gilt auch für weitere Parameterfunktionen.

5.2.2 Selektion

drselect

Bietet Selektion auf einer verteilten Relation. Parameter können sein:

- Attributname und Konstante. Weiterhin eine Funktion auf einem Tupelstrom mit Tupeln des Relationentyps.
  - Falls Attributname und Konstante zu einem Standardtyp gehören wie *int*, *real*, *string* usw., dann lautet die Bedingung „Attribut = Konstante“. Sie wird feldweise ausgewertet durch eine exactmatch-Query auf einem B-Baum, falls ein solcher Index existiert, andernfalls durch Filtern auf dem Tupelstrom der Relation.
  - Falls Attributname und Konstante zu räumlichen Typen gehören, so lautet die Bedingung „bbox(Attribut) intersects bbox(Konstante)“. Sie wird feldweise ausgewertet durch eine windowintersects-Query auf einem R-Baum, falls ein solcher Index existiert, andernfalls durch Filtern auf dem Tupelstrom der Relation. Die Parameterfunktion kann ein genaueres Prädikat wie *line_l intersects region_r* enthalten (filter+refine-Strategie).
- Attributname und zwei Konstanten eines Standardtyps. Optionale Parameterfunktion.
  - Die Bedingung lautet „Attribut between [Konstante1, Konstante2]“. Sie wird ebenfalls feldweise durch eine range-Query auf einem B-Baum ausgewertet, falls so ein Index vorhanden ist, andernfalls durch Filtern auf dem Tupelstrom der Relation.

Die Tupel, die sich qualifizieren, werden noch in die Parameterfunktion geleitet. Falls diese einen Tupelstrom liefert, so ist das Ergebnis eine verteilte Relation mit unverändertem Verteilungstyp.

Für Selektionen, die auf eine range- oder spatial-partitionierte Relation angewandt werden, kann man über den Verteilungsindex der Relation eine Menge von Feldnummern ermitteln, deren Relationen überhaupt Ergebnisse liefern können. Man kann eine spezielle (noch zu implementierende) Version von *dmap* der Distributed Algebra verwenden, um die Anfrage nur zu diesen Feldern zu senden.

Beispiel:

```
query BuildingsR drselect[; GeoData, hombruch; . filter[.GeoData inside
  hombruch]]
query CitiesR drselect[; Pop, 1000000, 2000000; . count]
```

5.2.3 Join

Es gibt drei Arten von Join:

- *equijoin*. Gleichheit von Standard-Attributen.
- *spatialjoin*. Räumliche oder raum-zeitliche Prädikate auf entsprechenden Datentypen.
- *genjoin*. Allgemeiner Join mit beliebigen Bedingungen über den Attributen der beiden Relationen

Die Implementierung von Joins enthält einfache Techniken der Anfrageoptimierung.

drequijoin

Parameter sind die beiden Attributnamen *X*, *Y* und eine Funktion, um die Ergebnistupel weiter zu verarbeiten. Die Joinbedingung ist *X* = *Y*.

```
R S drequijoin[; X, Y; .]
```

Dies wird implementiert mit *itHashJoin* oder Index-Nested-Loop-Join, falls ein B-Baum-Index existiert. Die Entscheidung zwischen diesen Möglichkeiten fällt durch Kostenschätzung.

drspatialjoin

Parameter sind die beiden Attributnamen *X*, *Y* der räumlich oder raum-zeitlich überlappenden Attribute und eine Funktion, um die Ergebnistupel weiter zu verarbeiten. Die Joinbedingung[1] ist *bbox*(*X*) *intersects bbox*(*Y*).

```
R S drspatialjoin[; X, Y; . filter[X inside Y] count]
```

Dies wird implementiert mit *itSpatialJoin* oder Index-Nested-Loop-Join, falls ein R-Baum-Index existiert. Die Entscheidung wird durch Kostenschätzung getrofen.

drgenjoin

Parameter ist ein Prädikat auf den beiden Argumenttupeln und eine Funktion, um die Ergebnistupel weiter zu verarbeiten.

```
R S drgenjoin[; .X < ..Y; . filter[.IdR < .IdS]]
```

Wird implementiert mit *symmjoin*.

Für Kostenschätzungen werden die Kardinalitäten der beiden Argumentrelationen verwendet. Eine Ermittlung der Selektivität der Joinbedingung ist nicht nötig.

In allen Fällen ist die Partitionierung der Argumentrelationen zu beachten. Falls nötig, muss mittels *partitionF* oder *dproduct* umverteilt werden. Auch hier wird die Entscheidung, falls beides in Frage kommt, mittels Kostenschätzung gefällt.

5.2.4 Sortieren

drsortby, **drsort**

Eine verteilte Relation wird global sortiert. Danach liegen die kleinsten Elemente in Feld 1, die größten im letzten Feld des verteilten Arrays. Die Vorgehensweise ist wie in [1], Abschnitt 7.4. Ergebnis ist für *drsortby* eine *range*-partitionierte verteilte Relation.

1. ergänzt um die Bedingung zur Vermeidung von Duplikaten, siehe Anhang.

5.2.5 Gruppieren und Aggregatfunktionen

drgroupby

Die Argumente sind ähnlich dem normalen *groupby*-Operator in Secondo. Kann man zunächst lokal und dann auf den Ergebnissen auf dem Master ausführen. Vgl. [1], Abschnitt 7.3. Ergebnis ist eine Relation auf dem Master.

5.3 Einsammeln

drsummarize

Analog zur entsprechenden Operation der Distributed Algebra. Ergebnis ist ein Tupelstrom auf dem Master.

```
query BuildingsR drselect[; GeoData, hombruch; .] drsummarize consume
```

5.4 Indexe

createIndex

Für eine gegebene verteilte Relation kann man einen verteilten Index anlegen. Parameter ist ein Attributname. Abhängig vom Attributtyp wird ein B-Baum- oder ein R-Baum-Index angelegt, letzterer mit bulkload (vgl. [1], Abschnitt 7.1.4). Der Indexname wird automatisch erzeugt, gemäß der Form

- <relname>_<attrname>_btree
- <relname>_<attrname>_rtree

Beispiel:

```
query BuildingsR createIndex[GeoData]
query BuildingsR createIndex[Osm_id]
```

Solche Indexe werden von *drselect* oder *drequijoin*, *drspatialjoin* automatisch verwendet bzw. zumindest in Betracht gezogen.

deleteIndex

Löscht einen verteilten Index.

```
query BuildingsR deleteIndex[GeoData]
```

5.5 Konvertieren

drgetarray

Man kann aus einer verteilten Relation, die partitioniert (nicht repliziert) gespeichert ist, den darstellenden *darray* oder *dfarray* erhalten.

drcreatedrel

Aus einem gegebenen *darray* oder *dfarray* kann man durch Hinzufügen der Verteilungsinformation eine verteilte Relation erzeugen.

Umbenennen (rename, {n})

Eine verteilte Relation kann umbenannt werden, dabei ändern sich nur die Attributnamen.

Referenzen

[1] R.H. Güting und T. Behr, Distributed Query Processing in Secondo. FernUniversität Hagen, Informatik-Report 375, December 2016.