TU München, Fakultät für Informatik
Lehrstuhl III: Datenbanksysteme
Prof. Alfons Kemper, Ph.D.

Übung zur Vorlesung *Einsatz und Realisierung von Datenbanken* im SoSe23

Alice Rey, Maximilian Bandle, Michael Jungmair (i3erdb@in.tum.de)
http://db.in.tum.de/teaching/ss23/impldb/

Blatt Nr. 09

Hinweise Für die aktive Teilnahme an der dieswöchigen Übung benötigen Sie Spark auf Ihrem Rechner installiert. Eine Anleitung finden Sie unter `https://db.in.tum.de/teaching/ss23/impldb/Spark_preparation.pdf`.

Hausaufgabe 1

HyPer schafft 120.000 Transaktionen pro Sekunde. Pro Transaktion werden 120 Byte in die Log geschrieben. Berechnen Sie den benötigten Durchsatz zum Schreiben der Log.

Die Datenbank läuft für einen Monat und stürzt dann ab. Es wurde kein Snapshot erstellt. Berechnen Sie die Recoveryzeit. Gehen Sie davon aus, dass die Recovery durch die Festplatte limitiert ist (100 MiB / s). Wieviel Log Einträge werden pro Sekunde recovert?

Durchsatz = 120.000 * 120 = 14400000 = 13,7 MiB/s.

LogEinträge =120.000*60*60*24*30=311040000000

LogGröße = LogEinträge*120 = 33,95 TiB

RecoveryZeit = 4,12 Tage

RecoveryDurchsatz = 873813 Tx / s.

Hausaufgabe 2

Gegeben eine Tabelle *Produkte* mit folgendem Schema und 10000 Einträgen:

Id (8 Byte) | Name (32 Byte) | Preis (8 Byte) | Anzahl (8 Byte)

Wieviele Daten werden für folgende Queries in die CPU-Caches geladen? Unterscheiden sie jeweils zwischen Row und Column Store.

1. *select * from Produkte*
2. *select Anzahl from Produkte*

Daten können maximal mit Granularität (64 Byte) in den Cache geladen werden. Das heißt, selbst wenn nur auf einen 64 Bit Integer Wert zugegriffen wird, muss ein kompletter 64-Byte Block geladen werden. Mit diesem Hintegrund ergeben sich folgende Ergebnisse:

1. *select * from Produkte*
   a) Row: $10000 * 56 = 560000$ Byte
   b) Column: $10000 * 8 + 10000 * 32 + 10000 * 8 + 10000 * 8 = 560000$ Byte
2. *select Anzahl from Produkte*
   a) Row: $10000 * 56 = 560000$ Byte
   b) Column: $10000 * 8 = 80000$ Byte

Hausaufgabe 3

Sie sollen für die Alexander-Maximilians-Universität (AMU) ein Hauptspeicherdatenbanksystem optimieren. In dem System sind die Daten aller Studenten gespeichert. Schätzen Sie für jede der untenstehenden Anfragen einzeln, ob ein Row- oder Column-Store besser geeignet ist.

Relationen

Studenten: MatrNr (8 Byte), Name (48 Byte), Studiengang (4 Byte), Semester (4 Byte)

MatrNr ist der Primärschlüssel der indiziert ist.

Anfragen:

1. select * from Studenten;
2. select Semester, count(*) from Studenten group by Semester;
3. select Name, Studiengang, Semester from Studenten where MatrNr = 42;
4. select Studiengang from Studenten where MatrNr = 42;
5. select * from Studenten where Semester < 5;
6. select * from Studenten where Semester = 25;
7. insert into studenten values(4242, Max Meyer, Info, 7);

Lösung:

1. Beides optimal: Gesamte Tabelle wird gelesen
2. Column-Store: Nur Spalte Semester wird gelesen. Bei Row Store muss die gesamte Tabelle gelesen werden.
3. Row-Store: Zeile passt in eine Cache Line (CL). Bei Column Store müssen mindestens 3 CLs gelesen werden.
4. Theoretisch Column-Store: Nur Studiengang muss gelesen werden, da ein Index existiert. Row-Store aber in der Realität gleich schnell, da Daten immer CL granular gelesen werden.
5. Beides optimal: Es muss die gesamte Tabelle gelesen werden, da das Prädikat nicht sehr selektiv ist.
6. Column-Store: Prädikat ist hoch selektiv. Es wird wahrscheinlich kein Student gefunden. Demnach müssen die anderen Felder nicht nachgeladen werden.
7. Row-Store: Zeile passt in eine CL. Damit muss beim Einfügen nur eine CL in den Speicher geschrieben werden. Bei Column Store müsste jedes Element einzeln geschrieben werden.

Hausaufgabe 4

Rekonstruieren Sie die ursprüngliche SQL-Anfrage aus dem folgenden (Pseudo-)Code eines codegenerierenden Datenbanksystems. Welche Art von physikalischem Join wurde benutzt? Handelt es sich um Column- oder Row-Store?

```
struct Student { int matrnr; std::string name; int semester; };
struct Hoeren { int matrnr; int vorlnr; };
```

```
struct Result { int vorlnr; int a; };

std::vector<Result> compute(std::vector<Student>&ses,std::vector<Hoeren>&hs){
       std::unordered_multimap<int, Hoeren*> h_map;
       std::unordered_map<int, Student*> s_map;
       for (auto &h : hs)
              h_map.insert(std::make_pair(h.matrnr, &h));
       for (auto &s : ses)
              s_map.insert(std::make_pair(s.matrnr, &s));

   // Group by h.vorlnr; avg(s.semester) = sum(s.semester)/count(*)

       std::unordered_map<int, int> count_map;
       std::unordered_map<int, int> sum_map;
       for (auto &h : hs) {
              count_map.insert(std::make_pair(h.vorlnr, 0));
              sum_map.insert(std::make_pair(h.vorlnr, 0));
       }
       for (auto &h : h_map) {
              sum_map[h.second->vorlnr] += s_map[h.first]->semester;
              count_map[h.second->vorlnr]++;
       }
       std::vector<Result> res;
       for (auto &r : sum_map)
              res.push_back({ r.first, r.second / count_map[r.first] });
       return res;
}
```

Lösung: Anfrage gibt Durchschnittssemester pro Vorlesung aus, Hash-Join (Vebesserung wäre Singleton-Join, da `matrnr` unique), Row-Store.

```
select vorlnr, avg(s.semester) from studenten s, hoeren h
where s.matrnr=h.matrnr group by h.vorlnr
```

Hausaufgabe 5

Führen Sie die folgenden Abfragen in der Spark-Shell aus. Als Grundlage für die Abfragen dient das TPC-H Schema. Laden Sie dazu die TPC-H Daten wie in der Vorlesung gezeigt in die Spark-Shell.

(a) Ermitteln Sie pro Marktsegment die Anzahl der Bestellungen in 1997.

(b) Ermitteln Sie die Zahl der Kunden und Lieferanten pro Land.

(c) Ermitteln Sie die Stückzahlen der verschiedenen Bauteile in Deutschland.

(d) Ermitteln Sie, welche Kunden kein *goldenrod lavender spring chocolate lace* bestellt haben.

Lösung:

(a) Ermitteln Sie pro Marktsegment die Anzahl der Bestellungen in 1997.

```
val ordersOf1997 = orders.where(year($"o_orderdate") === 1997)
val customerOrders = ordersOf1997.join(customer, $"c_custkey" === $"o_custkey")
val mktSegmentOrders = customerOrders
   .groupBy($"c_mktsegment")
   .agg(count($"o_orderkey").as("orderCount"))

mktSegmentOrders.show
```

(b) Ermitteln Sie die Zahl der Kunden und Lieferanten pro Land.

```
val countryCustomer = customer.groupBy($"c_nationkey")
   .agg(count($"c_custkey").as("customerCount"))
val countrySupplier = supplier.groupBy($"s_nationkey")
   .agg(count($"s_suppkey").as("supplierCount"))
val countryCustSupp = nation
   .join(countryCustomer, $"c_nationkey" === $"n_nationkey")
   .join(countrySupplier, $"s_nationkey" === $"n_nationkey")

countryCustSupp
   .select($"n_nationkey", $"n_name", $"customerCount", $"supplierCount")
   .show
```

(c) Ermitteln Sie die Stückzahlen der verschiedenen Bauteile in Deutschland.

```
val germany = nation.filter($"n_name" === "GERMANY")
val germanSupplier = supplier
   .join(germany, $"n_nationkey" === $"s_nationkey", "leftsemi")
val germanParts = partsupp
   .join(germanSupplier, $"ps_suppkey" === $"s_suppkey","leftsemi")
   .groupBy($"ps_partkey")
   .agg(sum($"ps_availqty").as("stueckzahl"))

germanParts.show
```

(d) Ermitteln Sie, welche Kunden kein *goldenrod lavender spring chocolate lace* bestellt haben.

```
val goldenRodLavenderSpringChoclateLaceOrders = part
   .filter($"p_name" === "goldenrod␣lavender␣spring␣chocolate␣lace")
   .join(partsupp, $"ps_partkey" === $"p_partkey")
   .join(lineitem, $"l_partkey" === $"ps_partkey")
   .join(orders, $"o_orderkey" === $"l_orderkey")
   .select($"o_custkey")
   .distinct

val noChocolateCustomer = customer.join(
   goldenRodLavenderSpringChoclateLaceOrders,
   $"o_custkey" === $"c_custkey",
   "leftanti")

noChocolateCustomer.show
```

Hausaufgabe 6

Führen Sie die folgenden Abfragen in der Spark-Shell aus. Als Grundlage für die Abfragen dient das TPC-H Schema. Laden Sie dazu die TPC-H Daten wie in der Vorlesung gezeigt in die Spark-Shell.

(a) Laden Sie die `region.tbl` Datei als DataFrame Objekt in die Spark-Shell.

Um die Datei als DataFrame zu laden, braucht man das Format der Datei, in diesem Fall CSV, das Schema der Daten, das Zeichen, mit dem die Spalten in der CSV-Datei getrennt werden, sowie den Pfad. Für das Schema gibt man für jede Spalte den Namen und den Typ an, sowie ob das Feld auch einen null-Wert enthalten darf:

```
val region = spark.read.format("csv").schema(StructType(
   List(
     StructField("r_regionkey", IntegerType, false),
     StructField("r_name", StringType, false),
     StructField("r_comment", StringType, false)
   )
 )).option("delimiter", "|").load("region.tbl")
```

(b) Ermitteln Sie die Namen aller Regionen.

Mithilfe der `select` Funktion können bestimmte Spalten ausgewählt werden:

```
region.select($"r_name").show
```

(c) Ermitteln Sie die Zahl der Länder die nicht in Europa liegen.

Zuerst wird aus dem DataFrame `region` Europa entfernt. Danach wird das gefilterte DataFrame mit `nation` gejoined um alle nicht-europäischen Länder zu finden. Statt der Aktion `show` wird die Aktion `count` verwendet, die keinen Text ausgibt, sondern einen Integer zurückgibt:

```
val notEurope = region.filter($"r_name" =!= "EUROPE")

val nonEuropeanCountries = nation
   .join(notEurope, $"r_regionkey" === $"n_regionkey", "leftsemi")

nonEuropeanCountries.count
```

(d) Ermitteln Sie die größte Bestellung aus dem Jahr 1996.

Zuerst werden alle Bestellungen herausgesucht, die im Jahr 1996 getätigt wurden. Danach sucht man alle Einträge aus dem `lineitem` DataFrame heraus, die zu einer der Bestellungen aus dem Jahr 1996 gehören. Um den Gesamtumfang der Bestellung zu ermitteln summiert man die Menge der `lineitems` pro Bestellung auf. Der größte Umfang kann dann durch eine Sortierung ermittelt werden:

```
val ordersOf1996 = orders
    .filter(year($"o_orderdate") === 1996)

val biggestOrder = lineitem
    .join(ordersOf1996, $"o_orderkey" === $"l_orderkey")
    .groupBy($"o_orderkey")
    .agg(sum($"l_quantity").as("size"))
    .orderBy($"size".desc)
    .limit(1)

biggestOrder.show
```

(e) Ermitteln Sie welcher europäische Kunde im Jahr 1996 am meisten Geld ausgegeben hat.
Zuerst werden alle Kunden herausgesucht aus europäischen Ländern mithilfe der `nation` und `region` DataFrames. Die Bestellungen werden dann wie bereits in Aufgabe 4 nach dem Jahr 1996 gefiltert. Danach werden die Gesamtpreise der Bestellungen pro Kunde aufsummiert. Der Kunde mit den höchsten Ausgaben kann dann wieder mit einer Sortierung ermittelt werden:

```
val nationsOfEurope = region
    .filter($"r_name" === "EUROPE")
    .join(nation, $"r_regionkey" === $"n_regionkey")
    .select($"n_nationkey")

val customerOfEurope = customer
.join(nationsOfEurope, $"n_nationkey" === $"c_nationkey", "leftsemi")
.select($"c_custkey")

val ordersOf1996 = orders.filter(year($"o_orderdate") === 1996)

val customerWithHighestCosts = ordersOf1996
.join(customerOfEurope, $"o_custkey" === $"c_custkey")
.join(lineitem, $"o_orderkey" === $"l_orderkey")
.groupBy($"c_custkey")
.agg(sum($"l_quantity" * $"l_extendedprice").as("costs"))
.orderBy($"costs".desc)
.limit(1)

customerWithHighestCosts.show
```

(f) Ermitteln Sie welche Unternehmen keine Kunden in Europa haben.

Um herauszufinden welche Unternehmen keine europäische Kunden haben, müssen alle Einträge in `lineitem` ermittelt werden, die von europäischen Kunden stammen. Dafür werden die DataFrames `region`, `nation`, `customer`, `orders` und `lineitem` miteinander verbunden. Mit einem anti-Join von `supplier` mit der Ergebnis-Relation bleiben nur die Unternehmen übrig, die keine Kunden in Europa haben:

```
val lineitemsOrderedByEuropeans = region
    .filter($"r_name" === "EUROPE")
    .join(nation, $"r_regionkey" === $"n_regionkey")
    .join(customer, $"n_nationkey" === $"c_nationkey")
    .join(orders, $"o_custkey" === $"c_custkey")
    .join(lineitem, $"o_orderkey" === $"l_orderkey")

val companiesWithoutEuropeanCustomers = supplier
    .join(lineitemsOrderedByEuropeans, $"l_suppkey" === $"s_suppkey", "leftanti")

companiesWithoutEuropeanCustomers.show
```