Aufgabenblatt 3
Anfrageausführung

- Abgabetermin: **Montag, 09.12.2018 (23:59 Uhr)**
- Zur Prüfungszulassung muss ein Aufgabenblatt mit mind. 25% der Punkte bewertet werden und alle weiteren Aufgabenblätter mit mindestens 50% der Punkte.
- Die Aufgaben sollen in Zweiergruppen bearbeitet werden.
- Abgabesytem unter
  `http://www.dcl.hpi.uni-potsdam.de/submit`
  - für alle Nicht-Programmieraufgaben: ausschließlich pdf-Dateien (nur eine Datei pro Aufgabe). Bitte mit Namen beschriftet.
  - bei der Programmieraufgabe: alle relevanten .cpp und .h Dateien (Das Framework selbst müsst ihr nicht mit abgeben)

Aufgabe 1: Ausführungsstrategien

Betrachte die Relation $R(a, b, c, d)$, die 1 Million Tupel enthält. Nimm an, dass das Attribut a Schlüsselkandidat für R ist und die Werte von a zwischen 0 und 999.999 liegen. Jeder Block der Relation R fasst 10 Tupel. Außerdem sind die Tupel in R nicht sortiert.

Benenne für jede der folgenden Anfragen diejenige Ausführungsstrategie, die die wahrscheinlich wenigsten I/Os für die Bearbeitung der Anfrage benötigt. Begründe deine Antwort durch Angabe der I/Os für *alle* Ausführungsstrategien. Die beiden aufgeführten Indexe existieren bereits und müssen nicht für eine Anfrage neu angelegt werden. Sie liegen allerdings auf der Festplatte, so dass für genutzte Index-Strukturen ebenfalls I/O-Kosten anfallen. Falls nötig, triff geeignete Annahmen bzgl. benötigter Parameter und deren Werte!

Die zu betrachtenden Ausführungsstrategien sind:

- Scannen der kompletten Relation R.
- Nutzen eines B+-Baum-Indexes für $R.a$.
- Nutzen eines Hash-Indexes für $R.a$.

Die Anfragen sind:

a) Finde alle Tupel aus R. **3 P**

b) Finde alle Tupel aus R mit $a < 50$. **3 P**

c) Finde alle Tupel aus R mit $a = 50$. **3 P**

d) Finde alle Tupel aus R mit $a > 50$ und $a < 100$. **3 P**

Aufgabe 2: Sort-Merge Join-Algorithmus

Gegeben sind zwei Relationen $R(\underline{A}, B)$ und $S(\underline{A}, C)$. Die Tupel in R umfassen 2^{13} = 8.192 Blöcke und die Tupel in S umfassen 2^{10} = 1.024 Blöcke. Für die Berechnung von $R \bowtie S$ können insgesamt 2^7 = 128 Blöcke des Hauptspeichers genutzt werden.

a) Wie viele sortierte Teillisten entstehen in der ersten Phase des Two-Phase Multiway Merge-Sort (TPMMS) bei der Sortierung von R? Wie viele für S? **2 P**

b) Wie viele Blöcke müsste eine der beiden Eingaberelationen *mindestens* umfassen, damit die Sortierung mittels des TPMMS *nicht* mehr möglich ist? **2 P**

c) Notiere kurz die Ausführungsschritte des Sort-Merge Join-Algorithmus für die Berechnung von $R \bowtie S$. Notiere zusätzlich für jeden Teilschritt die Anzahl an I/O-Operationen.
Hinweis : Beachte, dass das Attribut A in beiden Relationen Primärschlüssel ist. **4 P**

d) Angenommen die Blockanzahl von S vergrößert sich auf $2^{13} + 1 = 8.193$ Blöcke. Welches Problem ergibt sich dann bei der Berechnung von $R \bowtie S$ unter Verwendung des Sort-Merge Join-Algorithmus? Begründe deine Antwort. **2 P**

Aufgabe 3: Wahl der Join-Implementierung

Gegeben ist folgende SQL-Anfrage, die nach allen Schauspielerinnen des Films 'King Kong' sucht:

```
SELECT *
FROM SpieltMit M, Schauspieler S
WHERE M.name = S.name
AND titel='King Kong' AND geschlecht='w';
```

Die angefragten Relationen sind:

```
SpieltMit(titel, jahr, name)
Schauspieler(name, adresse, geschlecht, geburtsdatum)
```

In Relationaler Algebra lässt sich die Anfrage folgendermaßen darstellen:

$$\sigma_{\text{titel}='KingKong'}(\text{SpieltMit}) \bowtie \sigma_{\text{geschlecht}='w'}(\text{Schauspieler})$$

Nimm an, dass beide Relationen jeweils etwa 10.000 Tupel umfassen und es einen Primärindex auf Schauspieler.name gibt.

Vergleiche die I/O-Kosten eines Index-basierten, eines Sort-basierten und eines Hash-basierten Joins für die gegebene Anfrage. Falls nötig, triff geeignete Annahmen bzgl. der Parameter der einzelnen Join-Implementierungen (z.B. Blockgrößen). Wähle für jede Join-Variante außerdem die jeweils optimale Ausführungsreihenfolge von Selektionen und Join. Welche Algorithmus-Klasse sollte die Datenbank hier wählen?

4 P

Aufgabe 4: Implementierung: Block-basierter Nested-Loop Join

In den Materialien zur Vorlesung findet ihr ein kleines c++ Framework für die Implementierung von Algorithmen zur Anfrageausführung[1]. Eure Aufgabe ist es in dem Framework den Block-basierten Nested-Loop Join zu implementieren. Dafür ist bereits die Klasse *NestedLoopEquiJoinAlgorithm.cpp* angelegt. Diese enthält die Methode *join*, welche ihr implementieren sollt. Um uns auf den Algorithmus konzentrieren zu können machen wir folgende **Vereinfachungen**:

- Es gibt nur den *string* Datentyp.
- Die einzelnen Character werden als Ascii-Zeichen interpretiert. Dadurch werden Unicode-Zeichen, die in den ursprünglichen Daten vorhanden waren, unleserlich (im Ergebnis), aber dies ist nicht weiter dramatisch.
- Wir implementieren nur den Equi-join mit einer einzigen Bedingung, also die Äquivalenz zweier Spalten.

Die Schnittstelle für den Join, sowie der erwartete Output sind in der dazugehörigen Header-Datei (*NestedLoopEquiJoinAlgorithm.h*) beschrieben. Beim Bearbeiten der Aufgabe gelten folgende *Regeln*:

- Der *MemoryManager* ist eure Schnittstelle um Blöcke einzulesen bzw. neue Blöcke oder Tupel zu erzeugen bzw. deren Speicher wieder freizugeben. Der verfügbare Speicher ist begrenzt. Um diese Beschränkungen einzusehen gibt es entsprechende *getter*-Methoden.

[1] `FG Informationssysteme/VL DBS II/Programmierübung/Framework`

- Eure einzige Schnittstelle zu den Daten (sowohl Blöcke als auch Tupel) ist der MemoryManager. Gebt dementsprechend auch nicht den Speicher von Objekten selbst frei, sondern macht dies nur über die Schnittstellen des MemoryManagers! Sämtlicher weiterer Speicher, den ihr für euren Algorithmus braucht (z.B. für Datenstrukturen um Pointer auf die bereits reservierten Blöcke zu speichern) sollte so gering wie möglich gehalten werden.

Hinweise:

- Nutzt die *BlockReader*-Klasse um Blöcke der Input-Relationen einzulesen.
- Die Schnittstellen aller benötigten Methoden der vorgegebenen Klassen sind in den Header-Dateien (.h) beschrieben.
- Ihr könnt die Klasse *Main.cpp* verwenden um eure Implementierung zu testen. Größere Testdateien findet ihr ebenfalls in den Materialien[2]. Die beiden Relationen haben den Primärschlüssel als erstes Attribut. Die Größe des Join-Ergebnisses für den Join von Movies and Plots sind 1447 Tupel.
- Sollte ihr Probleme beim build, bzw. allgemeine Fragen haben, wendet euch gerne an die Mailingliste.

10 P

[2] `FG Informationssysteme/VL DBS II/Programmierübung/Daten`