UNIVERSITÄT DES SAARLANDES
Fachbereich 14 – Informatik
Lehrstuhl Prof. Weikum

Datenbanksysteme WS 1999/2000

Übung 8 — Beispiellösung zu Aufgaben 1,4,5,6

Aufgabe 1: Algebraische Optimierung

Gegeben seien die beiden Relationen

Produkte (<u>PNr</u>, Bez, Preis, Gewicht, Vorrat) und
Bestellungen (<u>BestNr</u>, Monat, Tag, KNr, PNr, Menge, Status)

sowie die View

BestInfo (BestNr, PNr, Bez, Gewicht, Menge)
:= π[BestNr,PNr, Bez, Gewicht, Menge] (Produkte $\bowtie$ Bestellungen).

Es gelte die Integritätsbedingung:

π[PNr, Bez, Preis, Gewicht, Vorrat] (Produkte) =
π[PNr, Bez, Preis, Gewicht, Vorrat] (Produkte $\bowtie$ Bestellungen),

d.h. jedes Produkt wird mindestens einmal bestellt und es gibt nur Bestellungen für existierende Produkte. Betrachten Sie nun die folgende Query:

π[PNr, Bez] (σ[Gewicht > 100] (BestInfo)).

Vereinfachen Sie diese Query mit Hilfe der Äquivalenzregeln der Relationenalgebra und der oben angegebenen Integritätsbedingung, so daß die Query ohne Join-Berechnung ausgeführt werden kann. <u>Lösung:</u>

$$
\begin{aligned}
&\pi[PNr, Bez](\sigma[Gewicht > 100]\ \text{BestInfo})\\
\overset{\text{Def View}}{=}\ &\pi[PNr, Bez](\sigma[Gewicht > 100]\\
&(\pi[BestNr, PNr, Bez, Gewicht, Menge](\text{Produkte} \bowtie \text{Bestellungen})))\\
\overset{\text{Regel 4}}{=}\ &\pi[PNr, Bez](\pi[PNr, Bez, Gewicht](\sigma[Gewicht > 100]\\
&(\pi[BestNr, PNr, Bez, Gewicht, Menge](\text{Produkte} \bowtie \text{Bestellungen}))))\\
\overset{\text{Regel 1}}{=}\ &\pi[PNr, Bez](\sigma[Gewicht > 100](\pi[PNr, Bez, Gewicht]\\
&(\pi[BestNr, PNr, Bez, Gewicht, Menge](\text{Produkte} \bowtie \text{Bestellungen}))))\\
\overset{\text{Regel 4}}{=}\ &\pi[PNr, Bez](\sigma[Gewicht > 100](\pi[PNr, Bez, Gewicht](\text{Produkte} \bowtie \text{Bestellungen})))\\
\overset{\text{Regel 4}}{=}\ &\pi[PNr, Bez](\sigma[Gewicht > 100](\pi[PNr, Bez, Gewicht]\\
&(\pi[PNr, Bez, Gewicht, Preis, Vorrat](\text{Produkte} \bowtie \text{Bestellungen}))))\\
\overset{\text{Integrität}}{=}\ &\pi[PNr, Bez](\sigma[Gewicht > 100](\pi[PNr, Bez, Gewicht]\\
&(\pi[PNr, Bez, Gewicht, Preis, Vorrat]\ \text{Produkte})))\\
\overset{\text{Regel 4}}{=}\ &\pi[PNr, Bez](\sigma[Gewicht > 100](\pi[PNr, Bez, Gewicht]\ \text{Produkte}))\\
\overset{\text{Regel 1}}{=}\ &\pi[PNr, Bez](\pi[PNr, Bez, Gewicht](\sigma[Gewicht > 100]\ \text{Produkte}))\\
\overset{\text{Regel 4}}{=}\ &\pi[PNr, Bez](\sigma[Gewicht > 100]\ \text{Produkte})
\end{aligned}
$$

Regel 1:	$\pi[R1](\sigma[P](R)) = \sigma[P](\pi[R1](R))$	falls P nur R1–Attribute enthält
Regel 4:	$\pi[R1](\pi[R2](R)) = \pi[R1](R)$	falls $R1 \subseteq R2$

Aufgabe 2: Algebraische Query-Transformation

Lösung siehe anderes Blatt.

Aufgabe 3: Optimierung von Joins

Lösung siehe anderes Blatt.

Aufgabe 4: Kostenschätzung von Joins

Gegeben seien die beiden Relationen R und S mit dem gemeinsamen Joinattribut J vom Typ $NUMBER$. J ist Primärschlüssel der Relation S und Fremdschlüssel von R, wobei angenommen wird, daß die Werte gleichverteilt sind. R enthalte 10000 Tupel der Länge 360 Byte und S 200 Tupel der Länge 180 Byte. Nehmen Sie an, daß beide Relationen und alle Indizes jeweils einen eigenen Tablespace belegen, wobei die Seitengröße 4 KBytes sei und der Seiten-Header die Länge 96 Bytes habe. Datenseiten seien zu 90 Prozent und Indexseiten zu 70 Prozent gefüllt. Der Datentyp $NUMBER$ belege 10 Bytes und ein TID oder Zeiger im B*-Baum habe die Länge 6 Bytes.

Bestimmen Sie die Anzahl der benötigten Blockzugriffe für die Joinberechnung nach Nested-Loop, Nested-Block, Nested-Loop mit Indizes über R bzw. S und Merge-Sort. Nehmen Sie dazu an, daß die Größe des zur Verfügung stehenden Datenpuffers einmal 2 Seiten und einmal 201 Seiten betrage. Benutzen Sie für das Sortieren durch Mischen den Datenpuffer als internen Speicher für den Sortierschritt. Für die Indexseiten stehe ein separater Puffer der Größe 2 Seiten zur Verfügung.

Lösung:

Wir unterscheiden bei unseren Berechnungen nicht zwischen RIO- und SIO-Zugriffen.

Verfügbarer Platz auf einer Datenseite:

$$b_d = 0,9 * (4096 - 96) = 3600 \quad \text{Byte}$$

Verfügbarer Platz auf einer Indexseite:

$$b_i = 0,7 * (4096 - 96) = 2800 \quad \text{Byte}$$

Aus diesen Angaben können wir zunächst die Größe von R, S, I_R, I_S berechnen.

	Wert	Formel	R	S
a.	#Tupel		10000	200
b.	Tupellänge (Byte)		360	180
c.	#Seiten	$(a.) * (b.)/b_d$	1000	10

	Wert	Formel	I_R	I_S
d.	TID-Listenlänge (Byte)	$10 + 6 * (a.)/\#\text{Werte}(J)$	310	16
e.	#TID-Liste/Seite	$\lfloor b_i/(d.) \rfloor$	9	175
f.	#Blätter	$\lceil \#\text{Werte}(J)/(e.) \rceil$	23	2
g.	#Sohnzeiger/Seite	$\lfloor ((b_i - 6)/(10 + 6)) + 1 \rfloor$	175	175
h.	Indexhöhe	$\lceil \log_{(g.)}(f.) \rceil + 1$	2	2

Aus diesen Angaben läßt sich nun Anzahl der benötigten Blockzugriffe für die Joinberechnung durch die verschiedenen Ausführungsprimitive bestimmen. Unserer Rechnung legen wir einmal einen Datenpuffer von 2 Seiten und einmal einen Puffer von 11 Seiten (201 war ein Druckfehler) zu Grunde. Einen Datenpuffer von zwei Seiten benötigt man mindestens, da zur Joinberechnung für jeder Relation mindestens eine Seite gleichzeitig im Speicher gehalten werden müssen.

a) *Nested-Loop-Join (NESTED LOOPS):* Einfache Berechnung eines Joins mittels einer Doppelschleife über alle Tupel der Relationen. Wenn möglich, wird mindestens die kleinere, innere der beiden Relationen (also S) im Puffer gehalten. Daraus lassen sich die beiden folgenden Formeln herleiten:

Puffergröße $p < S(INNER) + 1$ *Seiten:*

IO-Kosten: $S(\text{OUTER}) + T(\text{OUTER}) * S(\text{INNER})$

Puffergröße $p \geq S(INNER) + 1$ *Seiten:*

IO-Kosten: $S(\text{OUTER}) + S(\text{INNER})$

	OUTER	INNER	Puffer	Berechnung	#IO
1.	R	S	2	$1000 + 10000 * 10$	101000
2.	S	R	2, 11	$10 + 200 * 1000$	200010
3.	R	S	11	$1000 + 10$	1010

b) *Nested-Block-Join:* Verbesserte Variante des Nested-Loop-Join, ohne daß ein Index benötigt wird. Bei dem Nested-Block-Join wird jede Seite der Relation INNER für jede Seite der Relation OUTER nur einmal eingelesen. Daraus ergeben sich folgende Formeln:

Puffergröße $p < S(INNER) + 1$ *Seiten:*

IO-Kosten: $S(\text{OUTER}) + S(\text{OUTER}) * S(\text{INNER})$

Puffergröße $p \geq S(INNER) + 1$ *Seiten:*

IO-Kosten: $S(\text{OUTER}) + S(\text{INNER})$

Bei ausreichend großem Puffer sind Nested-Block- und Nested-Loop-Join also gleich schnell. Da Nested–Block keinen höheren Programmier– oder Speicheraufwand verursacht, jedoch nie langsamer als Nested-Loop ist, empfiehlt es sich immer den Nested-Block Algorithmus zu benutzen.

	OUTER	INNER	Puffer	Berechnung	#IO
1.	R	S	2	$1000 + 1000 * 10$	11000
2.	S	R	2, 11	$10 + 10 * 1000$	10010
3.	R	S	11	$1000 + 10$	1010

c) *Nested-LOOP-Join mit Indexzugriff auf innere Relation:* Variante des Nested-Loop-Join, die allerdings einen Index über Join-Attribut J der Relation INNER benötigt. Der Algorithmus bestimmt die zum jeweiligen OUTER-Tupel passenden Ausprägungen der INNER-Relation, indem er über den Index die TID-Liste und über die TID-Liste die passenden Tupel der Relation INNER traversiert. Da die Relation INNER nicht nach J sortiert ist, handelt es sich bei den Operationen um RIO-Zugriffe. Im Indexpuffer hält man am sinnvollsten die Wurzelseite des Index und den momentan benötigten inneren Knoten (bzw. ein Blatt). Es lassen sich folgende Formeln ableiten:

Puffergröße $< S(INNER)$*:*

$$\begin{aligned}\text{IO-Kosten} &= S(\text{OUTER}) \\ &+ T(\text{OUTER}) * (H(I_{\text{INNER}})\text{-}1\text{+}(\#\text{TID in Liste}))\text{+}1\end{aligned}$$

Puffergröße $\geq S(INNER)$*:*

$$\begin{aligned}\text{IO-Kosten} &= S(\text{OUTER}) + T(\text{OUTER}) * (H(I_{\text{INNER}}) - 1) \\ &+ S(INNER) + 1\end{aligned}$$

	OUT.	IN.	P	Berechnung	#RIO
1.	R	S	2	$1000 + 10000 * 2 + 1$	21001
2.	S	R	2,11	$10 + 200 * 51 + 1$	10211
3.	R	S	11	$1000 + 10000 + 11$	11011

d) *Merge-Join (MERGE-JOIN):* Berechnung eines Equi-Joins zwischen zwei nach dem Join-Attribut sortierten Tupellisten mittels Mischen der Listen. In jedem Sortierscan werden alle Tupel einer Relation in Blöcken gelesen, sortiert und in eine temporäre Datei zwischengespeichert. Da sich die Länge der Sortierintervalle pro Sortierschritt um den Faktor der Puffergröße vervielfacht, ist die Anzahl der Durchläufe logarithmisch zur Anzahl der zu sortierenden Blöcke. Der eigentliche Join kann durch jeweils einen Scan über die sortierten Tupelmengen berechnet werden. Die Anzahl der Puffer gibt die Startgröße der sortierten Teilfolgen vor und bestimmt damit die Anzahl der Durchläufe. Es ergibt sich folgende Formel:

IO-Kosten $= \lceil \log_{\#Puffer} S(R_1) \rceil * 2 * S(R_1) + \lceil \log_{\#Puffer} S(R_2) \rceil * 2 * S(R_2) + S(R_1) + S(R_2)$

	P	Berechnung	#IO
1.	2	$10 * 2 * 1000$ $+4 * 2 * 10$ $+1000 + 10$	21090
2.	11	$3 * 2 * 1000$ $+1 * 2 * 10$ $+1000 + 10$	7030

Aufgabe 5: Kostenschätzung und Tuning

In dieser Aufgabe sollen verschiedene Alternativen für den physischen Entwurf einer Oracle-Datenbank betrachtet und bewertet werden. Hierzu sollen Sie mittels der Analyse von Ausführungsplänen die zu erwartenden Anfragekosten schätzen.

Daten

Wir betrachten eine Relation R, die zwei numerische Attribute A und B enthält. Die beiden Attribute haben jeweils die Länge 4 Bytes. In dieser Relation sind 50 000 Tupel gespeichert (das entspricht 345 Datenbankseiten à 2KBytes). Die Attributwerte von A sind aus dem Intervall $[0 \ldots 999]$ mit Gleichverteilung der Tupel über die Attributwerte. Die Attributwerte von B sind aus dem Intervall $[0 \ldots 999]$, wobei 40 000 Tupel über das Intervall $[0 \ldots 99]$ gleichverteilt und 10000 Tupel über das Intervall $[100 \ldots 999]$ gleichverteilt sind.

Problemstellung

Voraussetzung für die Bewertung eines Entwurfs ist die genaue Kenntnis der Anwendungslast. Wir haben dieses in der Praxis nichttriviale Problem stark vereinfacht und wollen die verschiedenen Entwurfsalternativen nur bezüglich der drei folgenden Anfragen beurteilen:

```
Q1:  SELECT COUNT(*)
     FROM R
     WHERE A = 321 AND B = 50;
Q2:  SELECT COUNT(*)
     FROM R
     WHERE A = 321 AND B <> 50;
Q3:  SELECT COUNT(*)
     FROM R
     WHERE A <> 321 AND B = 50;
```

Entwurfsalternativen

Es sollen insgesamt sechs verschiedene Alternativen des physischen Entwurfs miteinander verglichen werden.

1. Es ist lediglich die Relation R gespeichert (d.h. es gibt keinerlei Indizes)
2. Es existiert ein Index über dem Attribut A
3. Es existiert ein Index über dem Attribut B
4. Es existiert sowohl über A als auch über B ein Index
5. Es existiert ein „clustered“ Index über A, d.h. die Relation ist zusätzlich bezüglich A physisch sortiert
6. Es existiert ein kombinierter Index über A und B (die Attributwerte werden konkateniert und als ein Wert im Index verwaltet)

Alle Indizes seien mittelbar und — mit Ausnahme der Alternative 5 — jeweils „unclustered“.

Versuchen Sie nun, die verschiedenen physischen Entwürfe miteinander zu vergleichen, indem Sie für die drei Anfragetypen die aufgrund der physischen Entwürfe möglichen Ausführungsstrategien bezüglich ihrer I/O-Kosten analysieren.

Füllen Sie dann untenstehende Tabelle aus, indem Sie für jede Query und jede Entwurfsalternative die geschätzten I/O-Kosten der jeweils besten Ausführungsstrategie eintragen.

Entwurfsalternative	Queries		
	Q1	Q2	Q3
kein Index			
Index über A			
Index über B			
Indizes über A und B			
Clusterd Index			
Kombinierter Index			

Es ist nicht nötig, für alle Queries und alle Entwürfe eine vollständige Analyse durchzuführen. Durch Überlegung lassen sich einige Resultate auf andere Entwürfe übertragen.

Lösung:
Daten: Gegeben ist eine Relation R mit zwei numerischen Attributen A und B, die jeweils eine Länge von 4 Byte haben. In der Relation sind 50000 Tupel gespeichert, das entspricht 345 Datenbankseiten (in einem Oracle-TABLE). Die Attributwerte von A sind gleichverteilt über dem Intervall [0...999]. Die Attributwerte von B sind auch aus dem Intervall [0...999], wobei jedoch 40000 Tupel über das Intervall [0...99] und 10000 Tupel über das Intervall [100...999] gleichverteilt sind.

Beurteilt werden sollen verschiedene Entwurfsalternativen bezüglich drei verschiedener Anfragen. Dazu benötigen wir die Höhe des Indexbaums. Bei einem Füllgrad von 100% und einer Sohnzeigerlänge von 6 Byte errechnet sich die Höhe der Indizes über R und S zu 2 Seiten und für den kombinierten Index zu 3 Seiten. Zur Berechnung der I/O-Kosten verwenden wir die in der Vorlesung eingeführten Funktionen $S(R)$ =#SEITEN(R) und $T(R)$ =#TUPEL(R).

a) *Es ist lediglich die Relation R gespeichert (ohne Indizes):*

   i) *Query 1:*
   ```
   SELECT COUNT(*)
   FROM R
   WHERE A = 321 AND B = 50
   ```

   ii) *Query 2:*
   ```
   SELECT COUNT(*)
   FROM R
   WHERE A = 321 AND B <> 50
   ```

   iii) *Query 3:*
   ```
   SELECT COUNT(*)
   FROM R
   WHERE A <> 321 AND B = 50
   ```

   Da nur die Relation selbst zur Verfügung steht, kann man sinnvoll nur mit einem Relationen-Scan (TABLE-ACCESS FULL) die Abfragen ausführen. Man braucht dazu jeweils 345 I/O Zugriffe.

b) *Es existiert ein Index über dem Attribut A:*

   i) *Query 1:*
   Wir können über einen INDEX-RANGE-SCAN schneller Exact-Match-Queries auf A ausführen und über einen TID-Zugriff (TABLE ACCESS BY ROWID) die Ergebnistupel auf die Bedingung

für das Attribut B überprüfen. Die Länge der TID-Liste beträgt durchschnittlich $\frac{\#\text{Tupel}}{\#\text{Werte}(A)} = \frac{50000}{1000} = 50$, d.h. wir brauchen $2 + 50 = 52$ I/O Zugriffe.

ii) *Query 2:*
(Analog zu Query 1)

iii) *Query 3:*
Es ist ungünstig einen Indexscan für einen <>-Operator zu benutzen. Daher muß man einen normalen Relationen-Scan durchführen. (345 I/O-Zugriffe)

c) *Es existiert ein Index über dem Attribut B:*

i) *Query 1:* Argumentation wie bei Index über A, jedoch ist die TID-Liste zum Schlüsselwert $B = 50$ sehr lang: $\#TID = \frac{40000}{100} = 400$. Das bedeutet 400 I/O-Zugriffe. Daher ist ein Relationen-Scan günstiger.

ii) *Query 2:* Index-Scan ist nicht anwendbar, daher Relation-Scan.

iii) *Query 3:* (Analog zu Query 1)

d) *Es existiert sowohl über A als auch über B einen Index:*

i) *Query 1:* Da sowohl für A als auch für B Gleichheitsprädikate spezifiziert sind, kann in beiden Fällen über den Index zugegriffen werden. Weil nur die Anzahl der Tupel gesucht ist, braucht nicht auf die eigentlichen Daten zugegriffen zu werden. Es wird einfach der Durchschnitt der beiden TID-Listen gebildet. D.h. wir kommen mit $2 + 2 = 4$ I/O-Zugriffen aus.

ii) *Query 2:* Der Index über B kann wegen des <>-Operators nicht benutzt werden. Daher brauchen wir 52 I/O-Zugriffe. (Siehe Entwurfsalternative 2)

iii) *Query 3:* (Analog zu Query 2 und Entwurfsalternative 3) Indexzugriff ist zu teuer, daher wird ein Relation-Scan benutzt: 345 I/O-Zugriffe

e) *Es existiert ein clustered Index über A:*

i) *Query 1:* Da die Tupel nach J sortiert sind, liegen die in der TID-Liste referenzierten Tupel alle hintereinander auf einer Seite. Da für jeden Attributwert eine neue Seite angefangen wird, und da für das Prädikat $a = 321$ AND $b = 50$ im Mittel weniger als 1 Tupel existiert reichen: $2 + 1 = 3$ I/O-Zugriffe.

ii) *Query 2:* (Analog zu Query 1)

iii) *Query 3:* (Analog zu Variante ohne Cluster:) Tablescan ist billiger: 345 I/O-Zugriffe.

f) *Es existiert ein kombinierter Index über A und B:* Die Indexhöhe für den zusammengesetzten Schlüssel ist unter obigen Annahmen 3. Gesucht ist noch die Länge der jeweiligen TID-Liste.

i) *Query 1:* Wir berechnen zunächst die Wahrscheinlichkeit $P(A = 321; B = 50)$:

$$\begin{aligned} P(A = 321; B = 50) &= P(A = 321) * P_{[0\ldots999]}(B = 50) \\ &= P(A = 321) * P(B < 100) * P_{[0..99]}(B = 50) \\ &= \frac{1}{1000} * \frac{40000}{50000} * \frac{1}{100} = \frac{1}{125000} \end{aligned}$$

Also gibt es $\lceil \frac{1}{125000} * 50000 \rceil = 1$ TIDs und damit insgesamt $3 + 1 = 4$ I/O-Zugriffe.

ii) *Query 2:* Wir führen über den zusammengesetzten Index eine Präfix-Match Abfrage `AB="321%"` und anschließend einen TID-Zugriff über die Ergebnistupel mit Filterprädikat `B<>50` durch. Die Länge der TID-Liste beträgt (analog zu Entwurfsalternative 2) 50 TIDS, d.h. insgesamt braucht man $3 + 50 = 53$ I/O-Zugriffe.

iii) *Query 3:* Der Index ist für dieses Prädikat nicht geeignet, d.h. Relationen-Scan: 345 I/O-Zugriffe

Aufgabe 6: Kostenschätzung mit Histogramm

Für die im Hinblick auf Queries wichtigsten Attribute einer Datenbank sollen Histogramme der folgenden Form im Datenbankkatalog abgelegt sein:

```
hist(R.A): /* Histogramm fuer
              Attribut A der
              Relation R    */

           array[1..maxbin] of
              record
                 low  : dom(A);
                 high : dom(A);
                 count: integer;
                 freq : integer;
              end;
```

Dabei sind $hist(R.A)[i].count$ die Anzahl der in der Relation vorkommenden verschiedenen Attributwerte und $hist(R.A)[i].freq$ die Anzahl der vorkommenden Tupel, die in das Intervall von $hist(R.A)[i].low$ bis $hist(R.A).high$ fallen (einschließlich der Grenzen).

Es gilt (bei $dom(A) = integer$, sonst analog):
$hist(R.A)[i].high = hist(R.A)[i+1].low - 1$ für $1 \leq i < maxbin$.

Geben Sie (als Formel oder Pseudocode) an, wie man aufgrund dieser Histogramme die Resultat-Kardinalitäten der folgenden Anfragetypen schätzen kann:

a) $\sigma[R.A = x](R)$ Exact-Match-Query mit $x \in dom(A)$

Lösung:

- Bei allen folgenden Lösungen bezeichnen wir einen Eintrag $hist(R.A)[i]$ als *Bin* i des Histogramms $hist(R.A)$.
- Zur Schätzung der Kardinalität des Resultates muß zunächst das Bin i bestimmt werden, in dem die Tupel $t \in R$ registriert sind, für die gilt: $t.A = x$. Für das gesuchte i muß also gelten: $hist(R.A)[i].low \leq x \leq hist(R.A)[i].high$.

- *Gleichverteilungsannahme*: Wir nehmen an, daß die $hist(R.A)[i].freq$ vielen Einträge in $hist(R.A)[i]$ gleichmäßig über die $hist(R.A)[i].count$ verschiedenen Attributwerte verteilt sind. Weitergehende Information über die tatsächliche Verteilung der Werte innerhalb eines Bins ist nicht vorhanden.
- Wir unterscheiden zwei Fälle. Im ersten Fall gehört der Wert x zu den $hist(R.A)[i].count$ verschiedenen Werten in $hist(R.A)[i]$. Die Wahrscheinlichkeit dafür ist $p_{hit} = count/(high - low + 1)$. In diesem Fall liefert die Query im Mittel $freq/count$ viele Tupel. Im zweiten Fall gehört x nicht zu den $hist(R.A)[i].count$ verschiedenen Werten und die Query liefert kein Tupel zurück.

Der Erwartungswert für die Kardinalität der Ergebnismenge ist dann:

$$|\sigma[R.A = x]| = p_{hit} \times freq/count + (1 - p_{hit}) \times 0 = freq/(high - low + 1)$$

b) $\sigma[a \leq R.A \leq b](R)$ Range-Query mit $a, b \in dom(A)$

Lösung:

- Wir gehen auch hier von derselben Gleichverteilungsannahme wie in a) aus.
- Folgender Pseudocode der Funktion $getCardOfRangeSelection(a, b)$ liefert das gewünschte Ergebnis.

```
FUNCTION getCardOfRangeSelection( INTEGER a, INTEGER b ):FLOAT
BEGIN
  cardinality:= 0;  // enthaelt schliesslich das Ergebnis
  FOR i:= 1 to maxbin DO
    IF hist(R.A)[i].low <= b AND hist(R.A)[i].high >= a THEN
      // Ueberlappung von [a,b] und [low,high].
      // Bestimme Anzahl der ueberdeckten Attributwerte im Bin
      attrValueCount := MIN(b, hist(R.A)[i].high)-MAX(a, hist(R.A)[i].low)+1;
```

```
      // Bestimme von [a,b] ueberdeckten Anteil
      coverRatio := attrValueCount/(hist(R.A)[i].high-hist(R.A)[i].low+1);
      // Bestimme Anzahl der in hist(R.A)[i] selektierten Tupel.
      cardinality := cardinality + coverRatio*hist(R.A)[i].freq;
    FI;
  OD;
  RETURN cardinality;
END;
```

c) $\pi[R.A](R)$ Projektion mit Duplikateliminierung

Lösung:

- Hier muss einfach die Gesamtanzahl der verschiedenen Attributwerte $R.A$ in R ermittelt werden.
- Folgender Pseudocode der Funktion $getCardOfProjection$ liefert das gewünschte Ergebnis.

```
FUNCTION getCardOfProjection:INTEGER
BEGIN
  cardinality:= 0;  // enthaelt schliesslich das Ergebnis
  FOR i:= 1 to maxbin DO
    cardinality := cardinality + hist(R.A)[i].count;
  OD;
  RETURN cardinality;
END;
```

d) $R \bowtie [R.A = S.B]\ S$ Equijoin

Lösung:

- Wir gehen auch hier von derselben Gleichverteilungsannahme wie in a) aus.
- Wir nehmen an, daß jeweils ein Histogramm für $R.A$ und $S.B$ existiert.
- Wir nehmen an, daß die Attribute $R.A$ und $S.B$ dieselben Werte in den Relationen annehmen, d.h. daß $\pi[R.A](R) = \pi[S.B](S)$ gilt. Eine solche Annahme ist z.B. oft bei Joins über Fremdschlüssel gerechtfertigt.
- Folgender Pseudocode der Funktion $getCardOfJoin$ liefert das gewünschte Ergebnis.

```
FUNCTION getCardOfJoin:INTEGER
BEGIN
  cardinality:= 0;  // enthaelt schliesslich das Ergebnis
  FOR i:= 1 to maxbin DO
    l := hist(S.B)[i].low;
    h := hist(S.B)[i].high;
    // Bestimme die Anzahl der Tupel r in R,
    // deren R.A im Intervall [h,l] liegt.
    coveredRTuples := getCardOfRangeSelection(l,h);
    // Bestimme durchschn. Anzahl der R-Tupel fuer
    // einen Wert aus hist(S.B)[i]
    avgRtuplesPerValue := coveredRTuples/hist(S.B)[i].count;
    // Bestimme durchschn. Anzahl der R-Tupel fuer einen Wert S.B
    cardinality := cardinality + hist(S.B)[i].freq * avgRtuplesPerValue;
  OD;
  RETURN cardinality;
END;
```

- Eine effizientere Implementierung würde vermeiden, daß bei jedem Aufruf der Funktion $getCardOfRangeSelection$ im Mittel die halbe Anzahl der Bins in $hist(R.A)$ durchlaufen wird. Durch die Sortierung der Bins reicht es aus, wenn bei jedem Aufruf bei dem Bin gestartet wird, das zuletzt eine Überdeckung mit [a,b] aufgezeigt hat. Dadurch brauchen die Bins in $hist(R.A)$ bei Aufruf von $getCardOfJoin$ nur einmal inspiziert werden.
- Nehmen wir zusätzlich an, daß für alle $i = 1..maxbin$ gilt: $hist(R.A)[i].low = hist(S.B)[i].low \wedge hist(R.A)[i].high = hist(S.B)[i].high$, so kann das Ergebnis durch folgende vereinfachte Funktion berechnet werden:

```
FUNCTION getCardOfJoin:INTEGER
BEGIN
  cardinality:= 0;  // enthaelt schliesslich das Ergebnis
  FOR i:= 1 to maxbin DO
    cardinality := cardinality + hist(S.B)[i].freq * hist(R.A)[i].freq/hist(R.A)[i].count;
  OD;
  RETURN cardinality;
END;
```