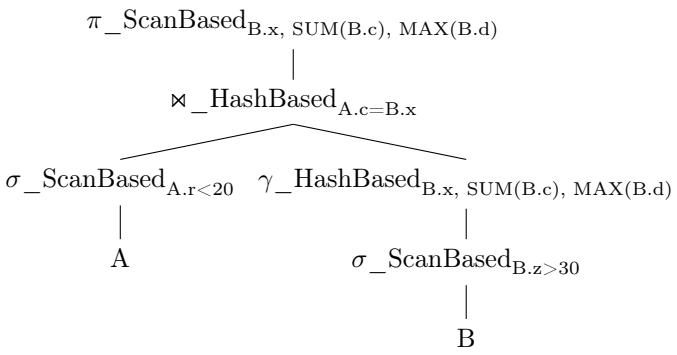
1 Codekompilierung (5 Punkte)

Betrachten Sie den folgenden physischen Plan.



Setzen Sie diesen Plan in Pseudocode um. Orientieren Sie sich dabei an Folie 39 aus der Vorlesung Anfrageoptimierung (Teil 2). Produzieren Sie dabei so wenig Zwischenergebnisse wie möglich.

Lösung: Vorgesehen sind 5 Punkte (1 Punkt für das Aufbauen einer Hashtabelle für B, 2 Punkte für korrekte Einträge in der Hashtabelle, 1 Punkt für die Wiederverwendung der Hashtabelle und 1 Punkt für den richtigen Output)

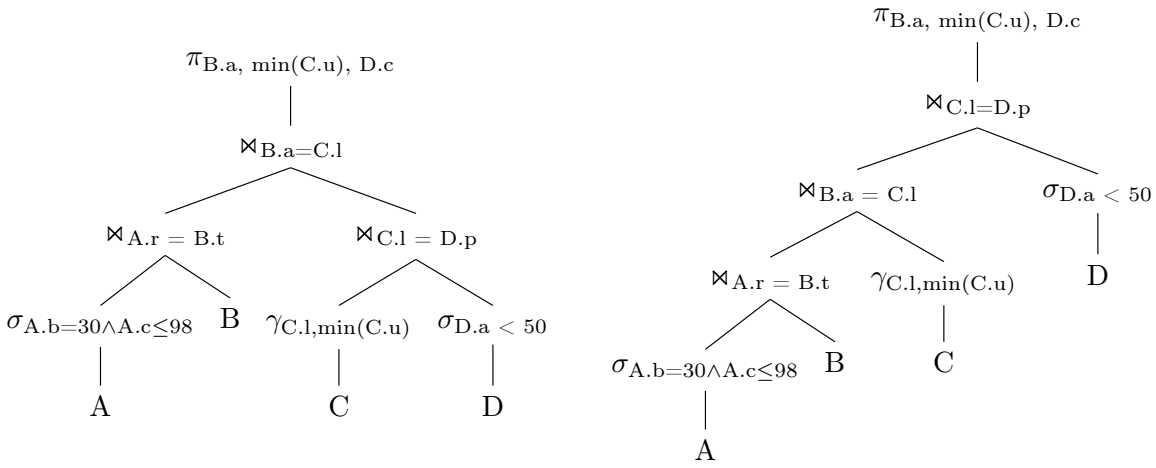
```
HashMap hm // maps B.x to (sum B.c, max B.d)

/* fill hash table, prepare aggregate computation */
for each b in B:
    if b.z > 30:
        if b.x in hm:
            /*add b.c to sum, change maximum if b.d is higher
                than the current max */
            if b.d > hm[b.x][1]:
                hm.update(b.x, (hm[b.x][0] + b.c, b.d))
            else:
                hm.update(b.x, (hm[b.x][0] + b.c, hm[b.x][1]))
        else:
            /* add new entry */
            hm.insert(b.x, (b.c, b.d))

/* reuse hash table to probe A */
for each a in A:
    if a.r < 20:
        e = hm.probe(a.c)
        /* produce join result if entry found */
        if e exists:
            /* group key A.c (= B.x), SUM(B.c), MAX(B.d) */
            yield(a.c, e[0], e[1])
```

2 Pipelines (3 Punkte)

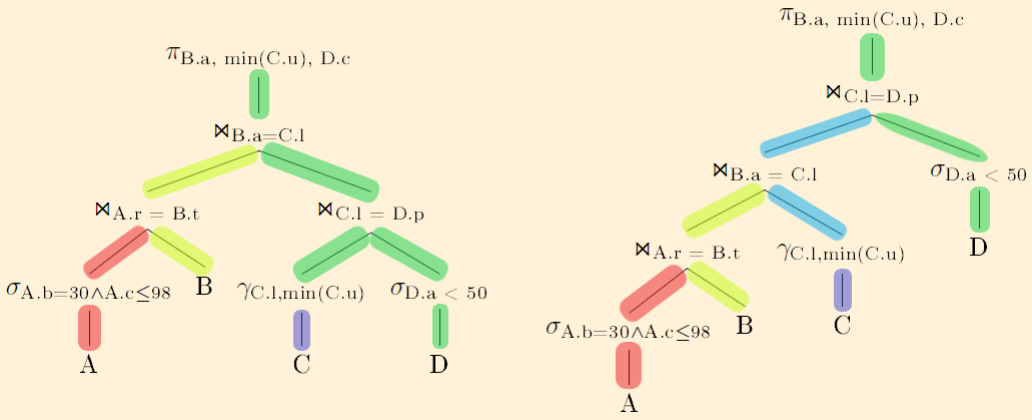
Betrachten Sie die beiden unten stehenden logischen Pläne.



Arbeiten Sie die Planabschnitte der beiden Pläne heraus, indem Sie die Pipeline-Breaker identifizieren. Beachten Sie dabei, dass unter Umständen Materialisierungspunkte zusammengelegt werden können. Nehmen Sie an, dass bei einem Join immer die linke Eingabe materialisiert wird. Zeichnen Sie anschließend die verschiedenen Abschnitte in die beiden Pläne ein. Orientieren Sie sich dabei an Folie 34 und 35 aus der Vorlesung Anfrageoptimierung (Teil 2).

Argumentieren Sie, welcher der beiden Pläne effizienter bezüglich Pipelining ist.

Lösung: 1 Punkt pro Plan: 0,5 Punkte für korrekte Pipeline-Breaker + 0,5 Punkte für korrekte Planabschnitte; 1 Punkt für die Nennung und Begründung der effizienteren Pipeline.



Im linken Plan haben wir insgesamt 3 Pipeline-Breaker. Der erste ist hierbei die Erstellung der Hashtabelle für A $\bowtie$ B, der zweite die Erstellung der Hashtabelle für B $\bowtie$ C und der letzte die Erstellung der Hashtabelle für die Gruppierung in C, welche anschließend für den Join mit D weiter benutzt werden kann. C $\bowtie$ D erzeugt also keinen weiteren Pipeline-Breaker.
Im rechten Plan hingegen haben wir insgesamt 4 Pipeline-Breaker. Hier haben wir die gleichen drei Breaker wie bereits im linken Plan, hier stellt allerdings der Join C $\bowtie$ D ebenfalls einen Pipeline-Breaker dar, da hier nicht die Gruppierungstabelle benutzt werden kann.

Die Planabschnitte A1 (rot gekennzeichnet), A2 (gelb) und A3 (lila) sind in beiden Plänen völlig identisch, somit wird hier die gleiche Menge an Daten materialisiert. Für den linken Plan muss nichts weiter materialisiert werden. Im rechten Plan hingegen muss zusätzlich noch A4 (blau) materialisiert werden. Somit wird im rechten Plan immer mehr materialisiert als im linken Plan, wodurch der linke Plan der effizientere Plan bezüglich Pipelining ist (unabhängig von den Relationsgrößen).

3 ACID-Eigenschaften (8 Punkte)

Im folgenden Beispiel finden Sie drei verschränkt ausgeführte Transaktionen. Die erste Transaktion überweist 24 Euro von Account A auf Account C, während die zweite Transaktion 1000 Euro von Account B auf Account A und die dritte Transaktion 95 Euro von Account C auf Account B überweist.

Gehen Sie davon aus, dass alle Schreiboperationen direkt durchgeführt werden und sofort für alle Transaktionen sichtbar werden, wobei sichtbar hier nicht persistent bedeutet.

	T_1	T_2	T_3
1	bal_{a1} = r(balA)		
2	$bal_{a1} = bal_{a1} - 24$		
3		bal_{a2} = r(balA)	
4			bal_{b3} = r(balB)
5		$bal_{a2} = bal_{a2} + 1000$	
6			$bal_{b3} = bal_{b3} + 95$
7		w(balA=bal_{a2})	
8		bal_{b2} = r(balB)	
9		$bal_{b2} = bal_{b2} - 1000$	
10	w(balA=bal_{a1})		
11	bal_{c1} = r(balC)		
12			w(balB=bal_{b3})
13		w(balB=bal_{b2})	
14		commit	
	Der aktuelle Zustand aller Transaktionen wird persistent auf die Festplatte geschrieben.		
15	$bal_{c1} = bal_{c1} + 24$		
16	w(balC=bal_{c1})		
17	commit		
18			bal_{c3} = r(balC)
19			$bal_{c3} = bal_{c3} - 95$
20			w(balC=bal_{c3})
21			commit

Nehmen Sie außerdem an, dass unsere Konsistenzbedingung darin besteht, dass die Summe aller Kontostände von Transaktionen nicht verändert werden darf, da sonst Geld verloren gehen oder generiert werden kann.

Geben Sie für jede der vier ACID-Eigenschaften an, wie und wo sie im obigen Beispiel potenziell verletzt werden. Geben Sie neben Ihrer Begründung auch die genaue Zeile an, in der die jeweilige Eigenschaft potenziell verletzt wird.

Lösung: Vorgesehen sind 2 Punkte pro Eigenschaft, davon ein Punkt für die Begründung und ein Punkt für die genaue Zeile bzw. das korrekte Beispiel.

Atomarität (Atomicity, A):
Atomarität ist in dieser Aufgabe an zwei verschiedenen Stellen verletzt.

1. Atomarität fordert, dass eine Transaktion eine untrennbare Einheit ist. Das bedeutet, dass eine Transaktion entweder vollständig oder gar nicht ausgeführt werden muss. Zwischen Zeile 14 und 15 werden jedoch alle bisherigen Aktionen persistent auf die Festplatte geschrieben, obwohl lediglich Transaktion 2 committet hat. Würde nun die Datenbank crashen, wären möglicherweise nur Teile der restlichen Transaktionen festgeschrieben, was im Konflikt zur Atomarität steht.

2. In unserem Beispiel werden alle Schreiboperationen direkt durchgeführt und sofort für alle Transaktionen sichtbar. Daher wird Atomarität eigentlich schon bei allen Schreiboperationen (Zeile 7, 10, 12, 13, 16, 20) verletzt.

Da sich Atomarität eigentlich nur auf eine einzelne Transaktion bezieht, verletzt der erste Punkt Atomarität im eigentlichen Sinne. Der zweite Punkt stellt lediglich eine Verletzung der Atomarität aufgrund der Annahme dar, dass alle Schreiboperationen direkt sichtbar sind.

Konsistenz (Consistency, C):
Konsistenz fordert, dass eine Transaktion T die Datenbasis beim committen in einem konsistenten Zustand hinterlässt. Wichtig ist jedoch, dass während der Abarbeitung einer Transaktion die Datenbasis inkonsistent sein darf, zumindest aus der Sicht dieser Transaktion. Hier ist es jedoch so, dass die Datenbank in einem inkonsistenten Zustand hinterlassen wird (Commit in Zeile 14), da der Kontostand von Account B um 1000 Euro verringert wurde, ohne dass im Gegenzug der Kontostand von Account A um 1000 Euro erhöht wurde. Dies liegt daran, dass T_1 beim Schreiben von balA in Zeile 10 nicht weiß, dass T_2 den vorher ausgelesenen Wert in Zeile 7 bereits überschrieben hat. Dadurch geht die Information verloren, dass T_2 1000 Euro zu balA hinzugefügt hat. Das gleiche Problem tritt für Account B in Zeile 13 auf. Hier wird der von T_2 vorher ausgelesene Wert um 1000 verringert in balB eingetragen, wobei nicht beachtet wird, dass T_3 durch eine Überweisung von 95 Euro diesen Wert in Zeile 12 ebenfalls überschrieben hat. Daher hat Account B nach obiger Ausführung 95 Euro weniger auf dem Konto, als es der Fall wäre, wenn alle Transaktionen sequentiell ausgeführt worden wären. Somit ist die Konsistenzbedingung bei allen Commits von T_1, T_2 und T_3 verletzt.

Isolation (Isolation, I):
Isolation fordert, dass nebenläufig ausgeführte Transaktionen sich nicht gegenseitig beeinflussen. T_1 und T_2 sind jedoch in einer Weise verzahnt, dass das Erhöhen von balA um 1000 Euro in Zeile 7 durch das nachfolgende Überschreiben durch T_1 in Zeile 10 verloren geht. Dies hat zur Folge, dass die Isolation von T_1 und T_2 verletzt ist. Außerdem geht das Erhöhen von balB in Zeile 12 seitens T_3 durch das Überschreiben durch T_2 in Zeile 13 verloren, wodurch auch die Isolation von T_2 und T_3 verletzt ist.

Dauerhaftigkeit (Durability, D):
Dauerhaftigkeit fordert, dass alle Schreiboperationen einer comitteten Transaktion dauerhaft in der Datenbasis erhalten bleiben. Da jedoch lediglich T_2 der hier angegebenen Transaktionen auf der Festplatte persistiert wird nachdem sie committet hat, ist die Dauerhaftigkeit (in den Zeilen 17 und 21) verletzt.

Anmerkung:
Obwohl am Ende von z.B. T_3 nach dem Commit nicht auf die Festplatte geschrieben wird, ist die Konsistenz hierdurch nicht zwingend verletzt. Das liegt daran, dass Änderungen wie beispielsweise

$$bal_{c3} = \text{r(balC))}$$
$$bal_{c3} = bal_{c3} - 95$$
$$\text{w(balC=}bal_{c3}\text{)}$$

Konsistenz nicht verletzen, da diese möglicherweise noch im "working set" (z.B Arbeitsspeicher bei Hauptspeicherdatenbanken) der Datenbank befinden und somit auch von weiteren Transaktionen "gesehen" werden.

4 Serialisierbarkeitstheorie (4 Punkte)

Betrachten Sie folgende Transaktionen.

T_1	T_2	T_3
read(B)	read(C)	read(A)
B := B + 110	C := C + 10	A := A ∗ 10
write(B)	write(C)	write(A)
	read(B)	read(C)
	B := B − 40	C := C − 130
	write(B)	write(C)

(a) Identifizieren Sie alle Paare von Konfliktoperationen.

(b) Es seien die folgenden drei Ausführungspläne gegeben:

	Ausführungspläne
AP_1	$r_2(C) \rightarrow r_1(B) \rightarrow w_1(B) \rightarrow r_3(A) \rightarrow w_2(C) \rightarrow r_2(B) \rightarrow w_3(A) \rightarrow w_2(B) \rightarrow r_3(C) \rightarrow w_3(C)$
AP_2	$r_2(C) \rightarrow r_3(A) \rightarrow r_1(B) \rightarrow w_3(A) \rightarrow r_3(C) \rightarrow w_3(C) \rightarrow w_2(C) \rightarrow r_2(B) \rightarrow w_2(B) \rightarrow w_1(B)$
AP_3	$r_3(A) \rightarrow r_1(B) \rightarrow w_3(A) \rightarrow r_3(C) \rightarrow w_3(C) \rightarrow r_2(C) \rightarrow w_2(C) \rightarrow w_1(B) \rightarrow r_2(B) \rightarrow w_2(B)$

Argumentieren Sie für jeden Ausführungsplan, ob dieser konfliktserialisierbar ist. Sofern der Ausführungsplan konfliktserialisierbar ist, sortieren Sie die Nicht-Konfliktoperationen so um, dass Sie einen seriellen, konfliktäquivalenten Ausführungsplan erhalten und geben Sie diesen an.

Lösung: 4 Punkte

(a) Vorgesehen ist 1 Punkt. (je 0,5 Punkte Abzug für fehlende/falsche Konfliktoperationen)

- $r_1(B)$, $w_2(B)$
- $w_1(B)$, $r_2(B)$
- $w_1(B)$, $w_2(B)$
- $r_2(C)$, $w_3(C)$
- $w_2(C)$, $r_3(C)$
- $w_2(C)$, $w_3(C)$

(b) Vorgesehen sind 1 Punkt pro Plan (0,5 Punkte für korrekte Argumentation bezüglich Konfliktserialisierbarkeit; 0,5 Punkte für seriellen, konfliktäquivalenten Ausführungsplan)

- AP_1:
  Ausführungsplan ist konfliktserialisierbar, da er konfliktäquivalent zu folgendem seriellen Ausführungsplan ist:
  $T_1 \rightarrow T_2 \rightarrow T_3$ bzw.
  $r_1(B) \rightarrow w_1(B) \rightarrow r_2(C) \rightarrow w_2(C) \rightarrow r_2(B) \rightarrow w_2(B) \rightarrow r_3(A) \rightarrow w_3(A) \rightarrow r_3(C) \rightarrow w_3(C)$
- AP_2:
  Ausführungsplan ist nicht konfliktserialisierbar, da wir die Nicht-Konfliktoperationen nicht so vertauschen können, sodass wir einen seriellen Ausführungsplan erhalten. Das liegt zum Beispiel an den Konfliktoperationen $r_1(B) \rightarrow w_2(B)$, $w_2(B) \rightarrow w_1(B)$, die es nicht möglich machen, einen seriellen Ausführungsplan zu erzeugen.
- AP_3:
  Ausführungsplan ist konfliktserialisierbar, da er konfliktäquivalent zu folgendem seriellen Ausführungsplan ist:
  $T_1 \rightarrow T_3 \rightarrow T_2$ oder $T_3 \rightarrow T_1 \rightarrow T_2$ bzw.
  $r_1(B) \rightarrow w_1(B) \rightarrow r_3(A) \rightarrow w_3(A) \rightarrow r_3(C) \rightarrow w_3(C) \rightarrow r_2(C) \rightarrow w_2(C) \rightarrow r_2(B) \rightarrow w_2(B)$
  oder
  $r_3(A) \rightarrow w_3(A) \rightarrow r_3(C) \rightarrow w_3(C) \rightarrow r_1(B) \rightarrow w_1(B) \rightarrow r_2(C) \rightarrow w_2(C) \rightarrow r_2(B) \rightarrow w_2(B)$

Abgabe

Lösungen sind in Teams von 2 bis 3 Studierenden bis zum 23. Juni 2022, 10:15 Uhr über Ihre persönlichen Statusseite im CMS einzureichen. Nutzen Sie hierfür die Team Groupings Funktionalität im CMS.

Reichen Sie die Abgabe als PDF ein, die Ihre Lösungen zu den Aufgaben 1, 2, 3 und 4 enthält.

Abgaben, die nicht den oben angegeben Vorgaben entsprechen, führen zu Punktabzug. Einzelabgaben werden nicht mehr korrigiert und mit 0 Punkten bewertet.