TECHNISCHE UNIVERSITEIT DELFT
Faculteit Elektrotechniek, Wiskunde en Informatica

Uitwerkingen van het Tentamen

TI2500 Informatie en Datamodellering

Vrijdag, 28 juni 2013
14u00-17u00

Dit tentamen bestaat uit 10 delen, elk met een aantal open vragen.
Van alle vragen dienen de antwoorden op het gewone antwoordblad(en) ingevuld te worden. Kladpapier wordt niet nagekeken

Deel	Punten
1	10
2	10
3	10
4	5
5	10
6	20
7	5
8	10
9	10
10	10
Totaal	**100**

Het gebruik van het boek "*Database Systems: Global Edition, 6/E*" door Ramez Elmasri en Shamkant Navathe is toegestaan, evenals slides, oude tentamens met uitwerkingen en eigen aantekeningen.

Vul je naam, studienummer en studierichting in op ieder antwoordblad.

Veel succes

Deel 1 – Relationele Algebra (10 punten)

Vraag 1. Geef een voorbeeld van een query uitgedrukt als een algebra expressie met een theta join die niet uitgedrukt kan worden in de relationele algebra zonder de theta join.

Antwoord: Het bijzondere van de theta-join ten opzichte van de equi-join en natural join is dat het in de match-conditie een *on*gelijkheid kan uitdrukken zoals bijvoorbeeld $R \bowtie_{a \leq b} S$. Echter, dit kan ook uitgedrukt worden met selectie en Cartesisch product als $\sigma_{a \leq b}(R \times S)$. Strikt genomen bestaat er dus niet zo'n voorbeeld.

Vraag 2. Beschouw het volgende relationele schema zoals ook gebruikt in het boek:

EMPLOYEE(Fname, Minit, Lname, Ssn, Bdate, Address, Sex, Salary, Super_ssn, Dno)
DEPARTMENT(Dname, Dnumber, Mgr_ssn, dept-sdate, Mgr_start_date)
DEPT_LOCATIONS(Dnumber, Dlocation)
PROJECT(Pname, Pnumber, Plocation, Dnum)
WORKS_ON(Essn, Pno, Hours)
DEPENDENT(Essn, Dependent_name, Sex, Bdate, Relationship)

Geef voor de volgende queries de algebra-expressie die deze queries uitdrukt, met de operatoren zoals besproken in Hfdst. 6 van het boek. Dit mag net zoals in het boek in de vorm van een enkele expressie of voor ingewikkelde queries in de vorm
<tabel> ← <algebra expressie>;
<tabel> ← <algebra expressie>;
... ;
RESULT ← <algebra expressie>.

a) Geef de namen van de employees waarvan de leidinggevende (aangegeven door Super_ssn) in een ander departement zit dan de employee zelf.

Antwoord:
$EMP1 := \pi_{Fname,Minit,Lname,Super\_ssn,Dno}(EMPLOYEE)$
$EMP2 := \rho_{Ssn2,Dno2}(\pi_{Ssn,\ Dno}(EMPLOYEE))$
$RESULT := \pi_{Fname,Minit,Lname}(\sigma_{Dno <> Dno2}(EMP1 \bowtie_{Super\_ssn = Ssn2} EMP2))$

Toelichting: In EMP1 selecteren we de relevante informatie voor werknemers. In EMP2 de informatie benodigd voor de leidinggevende en hernoemen de kolommen zodat we kunnen joinen met EMP1. Voor RESULT joinen we EMP1 en EMP2 op basis van het Super_ssn en selecteren vervolgens alleen de combinaties met verschillende departementen.

b) Geef de locaties waar alle projecten van hetzelfde departement zijn.

Antwoord: We geven twee oplossingen:

$RESULT := \pi_{Plocation}(\sigma_{COUNT\_Pname=1}({}_{Plocation}\mathcal{F}_{COUNT\ Pname}(PROJECT)))$

PR1 := $\pi_{Plocation,Dnum}$(PROJECT)
PR2 := $\rho_{Plocation2,Dnum2}(\pi_{Plocation,Dnum}$(PROJECT))
TWODEPTLOC := $\pi_{Plocation}(\sigma_{Dnum<>Dnum2}$(PR1 $\bowtie_{Plocation=Plocation2}$ PR2))
RESULT := $\pi_{Plocation}$(PROJECT) – TWODEPTLOC

Toelichting: In PR1 en PR2 selecteren we de relevante information van PROJECT. Daarna joinen we ze als ze dezelfde locatie betreffen, maar selecteren daarna alleen de combinaties die verschillende departementen hebben. Dat geeft ons dus de locaties die minstens twee verschillende projecten van verschillende departementen hebben. Tenslotte trekken we dat af van alle andere locaties, resulterende in alle locaties die projecten hebben uit ten hoogste 1 departement.

Deel 2 – Normaalvormen (10 punten)

Vraag 3. Beschouw een relatie met schema R(A, B, C, D, E) met sleutels AB en BC, en bovendien de set van functionele dependencies {ABD→E, C→B}. Wat is de hoogste normaalvorm waarin dit schema zit? Beargumenteer je antwoord.

Antwoord: Deze vraag is niet correct gesteld want AB en BC kunnen geen sleutels zijn als de gegeven dependencies gelden. Als AB en BC sleutels zijn, dan gelden de FDs AB→CDE en BC→ ADE. Als we controleren of AB en BC ook werkelijk *candidate keys* zijn, dan zien we dat dit niet klopt omdat C ook sleutel is (immers C→ABDE) en dus kan BC geen *candidate key* zijn, want het is niet minimaal. De echte set van *candidate keys* is: AB, C. Er geldt dus voor beide gegeven FDs dat de linkerkant een superkey is, want ABD ⊆ AB en C ⊆ C, en dus is dit schema in BCNF.

Als we er toch van uit gaan dat AB en BC de candidate keys zijn, dan is de relatie in 3NF. Immers, ABD→E voldoet want ABD is superkey (want superset van AB), en C →B ook omdat B een sleutel-attribuut is. Echter, om in BCNF te zijn zou dan C ook superkey meoten zijn, en dat is niet het geval.

Vraag 4. Stel we hebben een relatie R(A, B, C) met *join dependency* JD(AB, AC, BC). Is het dan mogelijk dat de inhoud van de relatie gelijk is aan: $\{(a_1, b_1, c_1), (a_2, b_1, c_2)\}$? Motiveer je antwoord.

Antwoord: Er moet gelden voor elke instantie *r* van R dat $r = \pi_{A,B}(r) * \pi_{B,C}(r) * \pi_{A,C}(r)$. We controleren of dit het geval is:

- $\pi_{A,B}(r) = \{(a_1, b_1), (a_2, b_1)\}$
- $\pi_{B,C}(r) = \{(b_1, c_1), (b_1, c_2)\}$
- $\pi_{A,C}(r) = \{(a_1, c_1), (a_2, c_2)\}$
- $\pi_{A,B}(r) * \pi_{B,C}(r) = \{(a_1, b_1, c_1), (a_1, b_1, c_2), (a_2, b_1, c_1), (a_2, b_1, c_2)\}$
- $\pi_{A,B}(r) * \pi_{B,C}(r) * \pi_{A,C}(r) = \{(a_1, b_1, c_1), (a_2, b_1, c_2)\}$

We krijgen nu dus als resultaat inderdaad weer $r = \{(a_1, b_1, c_1), (a_2, b_1, c_2)\}$. Dit is dus inderdaad een mogelijke inhoud.

Deel 3 – Normalizatie (10 punten)

Vraag 5. Beschouw een relatie met schema R(A, B, C, D, E) en de set van functionele dependencies S = {B→C, AC→D, D→E}. Bewijs met de regels van Armstrong (dus alleen IR1, IR2 en IR3) dat de functionele dependency AB→E in S^+ zit.

Antwoord:

1. B → C (gegeven)
2. AB → AC (IR2 toepassen op 1, met toevoeging van A aan beide kanten)
3. AC → D (gegeven)
4. AC → D (IR3 toepassen op 2 en 3)
5. D → E (gegeven)
6. AC → E (IR3 toepassen op 4 en 5)

Vraag 6. Beschouw een relatie met schema R(A, B, C, D, E) met de set van functionele dependencies S = {C→A, B→D, D→E}. Bepaal of de decompositie {AC, BC, BDE} de *lossless join property* heeft. Laat zien hoe je dit bepaald hebt.

Antwoord: Omdat het hier geen binaire decompositie betreft kunnen we de *NJB* property niet gebruiken om dit te beslissen (zie blz. 541 in boek). Daarom moeten we het algemene algoritme gebruiken (alg. 15.3 op blz. 538/9). Dan is het antwoord als volgt:

1. Initiele invulling van de tabel:

	A	B	C	D	E
R1	a_1	$b_{1,2}$	a_3	$b_{1,4}$	$b_{1,5}$
R2	$b_{2,1}$	a_2	a_3	$b_{2,4}$	$b_{2,5}$
R3	$b_{3,1}$	a_2	$b_{3,3}$	a_4	a_5

2. Toepassing van C→A

	A	B	C	D	E
R1	a_1	$b_{1,2}$	a_3	$b_{1,4}$	$b_{1,5}$
R2	**a_1**	a_2	a_3	$b_{2,4}$	$b_{2,5}$
R3	$b_{3,1}$	a_2	$b_{3,3}$	a_4	a_5

3. Toepassing van B→D:

	A	B	C	D	E
R1	a_1	$b_{1,2}$	a_3	$b_{1,4}$	$b_{1,5}$

R2	a_1	a_2	a_3	**a_4**	$b_{2,5}$
R3	$b_{3,1}$	a_2	$b_{3,3}$	a_4	a_5

4. Toepassing van D→E:

	A	B	C	D	E
R1	a_1	$b_{1,2}$	a_3	$b_{1,4}$	$b_{1,5}$
R2	a_1	a_2	a_3	a_4	**a_5**
R3	$b_{3,1}$	a_2	$b_{3,3}$	a_4	a_5

5. We hebben nu alleen a'tjes in R2, en dus mogen we concluderen dat de decompositie de *lossless/nonadditive join property* heeft.

Vraag 7. Beschouw een relatie met schema R(A, B, C, D, E, F, G) en de set van functionele dependencies S = {AB→CDE, BCD→F, F→B}.

a) Bepaal een decomposite naar een BCNF schema met de *nonadditive join property* volgens algoritme 15.5 in het boek op blz. 543. Geef per component ook de sleutels.

Antwoord: We bepalen eerst de sleutels van de start-tabel. Dat zijn in dit geval: {ABG, AFG}. Alle dependencies overtreden de BCNF regel, dus we kunnen kiezen met welke we willen beginnen om af te splitsen. Elke keuze is goed, en je hoeft er maar 1 uit te werken, maar voor de volledigheid geven we ze allemaal.

We kunnen bijvoorbeeld beginnen met AB→CDE af te splitsen en per component de sleutels te bepalen:

- R1(A, B, C, D, E) met sleutels { AB } en lokale dep's { AB→CDE }
- R2(A, D, F G) met sleutels { ADFG } en lokale dep's { }

Binnen deze relaties gelden er verder geen FDs die BCNF overtreden, en dus zijn we in BCNF.

We kunnen ook beginnen met BCD→F af te splitsen. Dan krijgen we:

- R1(B, C, D, F) met sleutels { BCD, CDF } en lokale dep's { BCD→F, F→B }
- R2(A, B, C, D, E, G) met sleutels { ABG } en lokale dep's { AB→CDE

Tenslotte kunnen we ook beginnen met F→B af te splitsen. Dan krijgen we:

- R1(B, F) met sleutels { F } en lokale dep's { F→B }
- R2(A, C, D, E, F, G) met sleutels { ACDEFG } en lokale dep's { }

Dit is uiteindelijk in BCNF.

b) Is deze decompositie *dependency preserving*? Beargumenteer je antwoord.

Antwoord: Dit hoeft alleen bepaald te worden voor de uitkomst gegeven bij a), maar voor de volledigheid beschouwen we hier alle mogelijke uitkomsten:

De eerste oplossing was:

- R1(A, B, C, D, E) met sleutels { AB } en lokale dep's { AB→CDE }
- R2(A, D, F G) met sleutels { ADFG } en lokale dep's { }

Hier is maar 1 van de 3 dependencies overgebleven en de andere zijn niet afleidbaar uit de overgeblevene. Dus hier we zijn **niet** *dependency-preserving.*

De tweede oplossing was:

- R3(B, F) met sleutels { F } en lokale dep's { F→B }
- R4(C, D, F) met sleutels { CDF } en lokale dep's { }
- R5(A, B, C, D, E) met sleutels { AB } en lokale dep's { AB→CDE }
- R6(A, B, G) met sleutels { ABG } en lokale dep's {

Hier zijn maar 2 van de 3 dependencies overgebleven, en de verdwenen dependency is niet afleidbaar uit de overgeblevenen. Dus hier we zijn **niet** *dependency-preserving*

De derde oplossign was:

- R1(B, F) met sleutels { F } en lokale dep's { F→B }
- R2(A, C, D, E, F, G) met sleutels { ACDEFG } en lokale dep's { }

Hier is maar 1 van de 3 dependencies overgebleven en de verdwenen dependencies zijn niet afleidbaar uit de overgeblevene. Dus hier we zijn **niet** *dependency-preserving.*

Deel 4 – File organizatie (5 punten)

Vraag 8. Een ARTICLES bestand met Art# als de hash-key bevat records met de volgende Art# waardes: 4819, 3290, 5182, 6101, 1129, 1211, 3334, 4515, 1440, 9878, 3123, 910, 3325, 5631, en 9748. Het bestand gebruikt tien *buckets* genummerd 0 t/m 9. Elke bucket is een *disk block* en bevat twee records. Laad deze records in het bestand in de aangegeven volgorde, gebruikmakende van de hash-functie $h(K) = K$ **mod** 10. Ga er daarbij vanuit dat bij overflow van een bucket er een overflow bucket wordt aangemaakt waarnaar een pointer verwijst.

Bereken het gemiddelde aantal *block accesses* dat nodig is voor het ophalen van een record op basis van een gegeven Art#. Ga er daarbij van uit dat de hash-tabel zelf zich in het geheugen bevindt en de toegang daartoe dus geen block-access veroorzaakt.

Antwoord:
De opgebouwde disk-structuur is als volgt.

block 0 : [3290, 1440] → [910]
block 1 : [6101, 1211] → [5631]
block 2 : [5182]
block 3 : [3123]
block 4 : [3334]
block 5 : [4515, 3325]
block 6 : []
block 7 : []
block 8 : [9878, 9748]
block 9 : [4819, 1129]

Dus voor 2 nummers zijn 2 *block-accessen* nodig, voor alle andere 13 is er 1 nodig. Dus gemiddeld 17/15 = 1 2/15 = 1.133.. accessen.

Deel 5 – Indexering (10 punten)

Vraag 9. Een PARTS bestand met Part# als sleutelveld bevat records met de volgende Part# waardes: 12, 8, 3, 9, 1, 10, 11, 7, 6, 15, 13, 14. Stel dat deze zoekwaardes in deze volgorde ingevoegd worden in een lege B^+-boom van orde p=4 en p_{leaf}=3. Laat zien hoe de uiteindelijke B^+-boom er uitziet.

Antwoord: Voor de volledigheid laten we de opbouw van de boom stap voor stap zien.

[12]

========

[8, 12]

========

[3, 8, 12]

========

. [9, 12]
8
. [3, 8]

========

```
 . [9, 12]
8
 . [1, 3, 8]

========
```

```
 . [9, 10, 12]
8
 . [1, 3, 8]

========

 . [11, 12]
10
 . [9, 10]
8
 . [1, 3, 8]

========

 . [11, 12]
10
 . [9, 10]
8
 . [7, 8]
3
 . [1, 3]

========

 . [11, 12]
10
 . [9, 10]
8
 . [6, 7, 8]
3
 . [1, 3]

========

 . [11, 12, 15]
10
 . [9, 10]
8
 . [6, 7, 8]
3
 . [1, 3]

========
```

```
.  . [13, 15]
. 12
.  . [11, 12]
. 10
.  . [9, 10]
8
.  . [6, 7, 8]
. 3
.  . [1, 3]

========

.  . [13, 14, 15]
. 12
.  . [11, 12]
. 10
.  . [9, 10]
8
.  . [6, 7, 8]
. 3
.  . [1, 3]

========
```

Deel 6 – Queryoptimalisatie (20 punten)

Vraag 10. Kan een *nondense index* gebruikt worden in de implementatie van een aggregatie-operator? Leg uit waarom wel of niet.

Antwoord: Nee. In een *dense* index komen alle geïndexeerde waardes voor, en dus kan deze gebruikt worden om bijvoorbeeld een maximum, som, etc., over de geïndexeerde kolom te berekenen. Maar in een *nondense* index kunnen waarden ontbreken, en dus volstaat de index niet om de aggregatie berekenen.

Vraag 11. Een bestand van 16.000 blocks moet gesorteerd worden met een beschikbare bufferruimte van 10 blocks. Hoeveel *passes* zijn er nodig in de *merge*-fase van het *external sort-merge* algoritme?

Antwoord: Na de eerste fase hebben we $\lceil$ 16.000/10 $\rceil$ = 1.600 *initial runs*. Vervolgens kunnen we in elke pass 9 *runs mergen*. Dus zijn er $\lceil \log_9(1.600) \rceil = 4$ passes nodig.

Vraag 12. Beschouw de volgende SQL-query voor het schema van vraag 2:

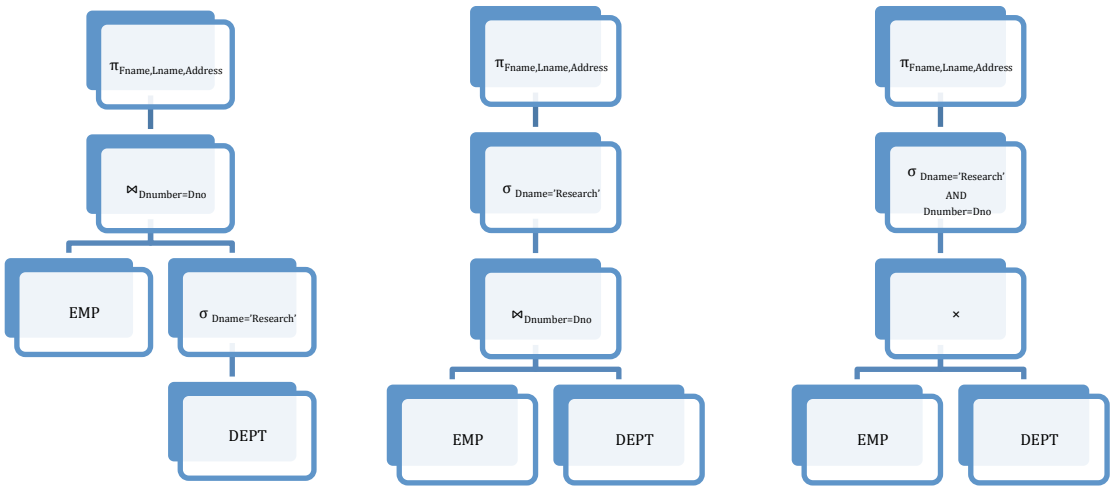
SELECT Fname, Lname, Address
FROM EMPLOYEE, DEPARTMENT
WHERE Dname='Research' **AND** Dnumber=Dno;

Geef twee verschillende query trees voor deze query.

Antwoord:
Beschouw de volgende drie mogelijk algebra-expressies die de query berekenen:

1. $\pi_{\text{Fname, Lname, Address}}(\text{EMP} \bowtie_{\text{Dnumber=Dno}} (\sigma_{\text{Dname='Research'}} (\text{DEPT})))$
2. $\pi_{\text{Fname, Lname, Address}}(\sigma_{\text{Dname='Research'}} (\text{EMP} \bowtie_{\text{Dnumber=Dno}} \text{DEPT}))$
3. $\pi_{\text{Fname, Lname, Address}}(\sigma_{\text{Dname='Research' AND Dnumber=Dno}} (\text{EMP} \times \text{DEPT}))$

Hiermee corresponderen de volgende query trees:



Vraag 13. Een voorbeeld van een transformatie in de relationele algebra die gebruikt kan worden voor optimalisatie is $\sigma_c(R \bowtie S) \equiv \sigma_c(R) \bowtie S$ mits de conditie c alleen naar kolommen in R verwijst. Leg kort uit hoe en waarom deze regel gebruikt kan worden voor query-optimalisatie.

Antwoord: De regel staat toe om iets in de linkervorm $\sigma_c(R \bowtie S)$ te herschrijven tot de rechtervorm $\sigma_c(R) \bowtie S$ die waarschijnlijk efficiënter is. De reden hiervoor is dat in die vorm we zo vroeg mogelijk records uit R weg-selecteren die niet nodig zijn in de rest van de berekening, zodat we deze niet onnodig koppelen met records in S.

Deel 7 – Databasetuning (5 punten)

Vraag 14. Stel we hebben besloten dat een tabel met kolommen A, B en C een index nodig heeft op zowel kolom A als op kolom B, en we willen nu beslissen welke van deze twee indexen een *clustered index* wordt. Geef een voorbeeld van een argument om de juiste index hiervoor te kiezen.

Antwoord: Als op een van de twee kolommen vaker een range-query wordt uitgevoerd, dan verdient het de voorkeur om hierop een *clustered-index* aan te leggen. Deze zorgt in dat geval ervoor dat er minder page-access nodig is als de bladeren van de index-boom achtereenvolgens worden afgelopen.

Deel 8 – Transacties (10 punten)

Vraag 15. Beschouw de drie transacties T_1, T_2, and T_3, en de schedules S_1 en S_2 van deze transacties zoals beneden gegeven. Teken de *precedence graph* voor S_1 en S_2, en bepaal welk schedule serialiseerbaar is. Als een schedule serialiseerbaar is, geef dan een equivalent *serial schedule*.

T_1: $r_1(X)$; $r_1(Z)$; $w_1(Z)$;
T_2: $r_2(Y)$; $w_2(Y)$; $r_2(X)$; $w_2(X)$;
T_3: $r_3(Z)$; $w_3(Z)$; $r_3(Y)$; $r_3(X)$; $w_3(Y)$;

S_1: $r_2(Y)$; $r_3(Z)$; $w_2(Y)$; $r_2(X)$; $w_3(Z)$; $r_3(Y)$; $w_2(X)$; $r_3(X)$; $r_1(X)$; $r_1(Z)$; $w_3(Y)$; $w_1(Z)$;
S_2: $r_2(Y)$; $r_3(Z)$; $w_3(Z)$; $r_3(Y)$; $w_2(Y)$; $r_2(X)$; $r_1(X)$; $w_2(X)$; $r_3(X)$; $r_1(Z)$; $w_3(Y)$; $w_1(Z)$;

Antwoord:

Voor S_1 zien we de volgende dependencies/pijlen:

- omdat $w_2(X)$ voor $r_3(X)$ en $r_1(X)$: $T_2 \rightarrow T_3$, $T_2 \rightarrow T_1$
- omdat $w_2(Y)$ voor $r_3(Y)$ en $w_3(Y)$: $T_2 \rightarrow T_3$
- omdat $w_3(Z)$ voor $r_1(Z)$ en $w_1(Z)$: $T_3 \rightarrow T_1$

Er is niet sprake van een cykel, dus het schedule is serialiseerbaar. Een mogelijk serial schedule is: $T_2 \rightarrow T_3 \rightarrow T_1$.

Voor S_2 zien we de volgende dependencies:

- omdat $r_1(X)$ voor $w_2(X)$ en deze weer voor $r_3(X)$: $T_1 \rightarrow T_2 \rightarrow T_3$
- omdat $r_2(Y)$ voor $w_3(Y)$: $T_2 \rightarrow T_3$
- omdat $r_3(Y)$ voor $w_2(Y)$: $T_3 \rightarrow T_2$
- omdat $w_2(Y)$ voor $w_3(Y)$: $T_2 \rightarrow T_3$
- omdat $r_3(Z)$ voor $w_1(Z)$: $T_3 \rightarrow T_1$
- omdat $w_3(Z)$ voor $r_1(Z)$ and $w_1(Z)$: $T_3 \rightarrow T_1$

Er is sprake van diverse cykels, zoals bvb. $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1$ en $T_2 \rightarrow T_3 \rightarrow T_2$. Het gegeven schedule is dus niet serialiseerbaar.

Deel 9 – Concurrency protocollen (10 punten)

Vraag 16. Beschouw de volgende abstracte beschrijving van de lees- en schrijfstappen in een transactie:

[1] r(X); [2] r(Y); [3] r(Z); [4] r(U); [5] w(Z); [6] w(Y); [7] w(Y); [8] w(U); [9]

Geef aan op welke plekken (aangegeven met vierkante haken) we welke shared/exclusive lock-operaties in welke volgorde moeten plaatsen voor het aanvragen en vrijgeven van locks willen we tegelijk (1) ons aan het *shared/exclusive locking scheme* houden, (2) ons aan het *two-phase locking protocol* houden en (3) telkens zo min mogelijk resources bezet houden.

Antwoord: De standaard oplossing waarbij locks worden aangevraagd als ze nodig zijn en weer zo vroeg mogelijk losgelaten (dwz. nadat de laatste lock-aanvraag is gebeurd, om aan eis (2) te voldoen) is de volgende:

- [1] : read_lock(X)
- [2] : read_lock(Y)
- [3] : read_lock(Z)
- [4] : read_lock(U)
- [5] : write_lock(Z)
- [6] : write_lock(Y)
- [7] : -
- [8] : write_lock(U); unlock(X); unlock(Y); unlock(Z)
- [9] : unlock(U)

In dit geval is het aantal stappen dat de resources worden bezet: U(5), X(7), Y(6), Z(5). Maar het kan beter als we locks soms al eerder aanvragen, zodat het moment van de laatste aanvraag eerder is, zodat we dan eerder weer kunnen beginnen met vrijgeven:

- [1] : read_lock(X)
- [2] : read_lock(Y)
- [3] : read_lock(Z)
- [4] : read_lock(U)
- [5] : write_lock(Z); write_lock(U); write_lock(Y); unlock(X)
- [6] : unlock(Z)
- [7] : -
- [8] : unlock(Y)
- [9] : unlock(U)

In dit geval is het aantal stappen dat de resources worden bezet: U(5), X(4), Y(6), Z(3). Dat is dus waarschijnlijk beter dan het voorgaande schema.

Vraag 17. Beschouw opnieuw de twee schedules S1 en S2 uit vraag 15. Bepaal welke van deze zijn toegestaan zijn volgens het *basic timestamp ordering* (TO) algoritme.

Antwoord: Aangezien S2 niet serializeerbaar is, zal deze ook niet toegestaan zijn door het TO algoritme. Maar we werken toch beide gevallen uit ter illustratie.
We volgen de executie van S1. De timestamps van de transacties worden aangenomen te zijn bepaald door hun eerste operatie, dus $TS(T_1) = 9$, $TS(T_2) = 1$ en $TS(T_3) = 2$. Na elke stap registreren we de read timestamp (Rd_ts(*D*)) en write timestamp (Wr_ts(*D*)) van elk data item (*D*=*X,Y,Z*).

T	Operat.	Rd_ts(*X*)	Wr_ts(*X*)	Rd_ts(*Y*)	Wr_ts(*Y*)	Rd_ts(*Z*)	Wr_ts(*Z*)
0		-	-	-	-	-	-
1	$r_2(Y)$	-	-	1	-	-	-
2	$r_3(Z)$	-	-	1	-	2	-
3	$w_2(Y)$	-	-	1	1	2	-
4	$r_2(X)$	1	-	1	1	2	-
5	$w_3(Z)$	1	-	1	1	2	2
6	$r_3(Y)$	1	-	2	1	2	2
7	$w_2(X)$	1	1	2	1	2	2
8	$r_3(X)$	1	1	2	1	2	2
9	$r_1(X)$	9	1	2	1	2	1
10	$r_1(Z)$	9	1	2	1	9	1
11	$w_3(Y)$	9	1	2	2	9	1
12	$w_1(Z)$	9	1	2	2	9	9

Er was geen conflict, dus dit schedule is toegestaan.

We beschouwen op dezelfde manier de uitvoering van schedule S2. De timestamps van de transacties worden opnieuw aangenomen te zijn bepaald door hun eerste operatie, dus $TS(T_1) = 7$, $TS(T_2) = 1$ en $TS(T_3) = 2$.

T	Operat.	Rd_ts(*X*)	Wr_ts(*X*)	Rd_ts(*Y*)	Wr_ts(*Y*)	Rd_ts(*Z*)	Wr_ts(*Z*)
0		-	-	-	-	-	-
1	$r_2(Y)$	-	-	1	-	-	-
2	$r_3(Z)$	-	-	1	-	2	-
3	$w_3(Z)$	-	-	1	-	2	2
4	$r_3(Y)$	-	-	**2**	-	2	2
5	$w_2(Y)$				**1**		
6	$r_2(X)$						
7	$r_1(X)$						
8	$w_2(X)$						
9	$r_3(X)$						
10	$r_1(Z)$						
11	$w_3(Y)$						
12	$w_1(Z)$						

Op T=5 is er een conflict want Rd_ts(*Y*) = 2 > $TS(T_2)$ = 1. Dus dit schedule is niet toegestaan.

Deel 10 – Recovery protocollen (10 punten)

Vraag 18. Beschouw de volgende geabstraheerde *system log* na een crash:

[**start**, T1] [**write**, T1, *D*, 15, 20] [**start**, T4] [**write**, T4, *B*, 25, 15] [**checkpoint**] [**write**, T4, *A*, 10, 20] [**commit**, T1] [**commit**, T4] [**start**, T2] [**write**, T2, *B*, 15, 12] [**start**, T3] [**write**, T3, *A*, 20, 30] [**commit**, T3] [**write**, T2, *D*, 20, 25]

De log-records hebben de volgende betekenis:
[**start**, *Ti*] = start van transactie *Ti*
[**write**, *Ti*, *X*, *old*, *new*] = update van data item *X* van waarde *old* naar *new*
[**commit**, *Ti*] = commit van transactie *Ti*
[**checkpoint**] = checkpoint

Veronderstel dat we voor de *recovery* de procedure RIU_M (UNDO/REDO with checkpoints) volgen (zie blz. 797). Welke operaties worden dan in welke volgorde uitgevoerd? Gebruik dezelfde notatie als voor de getoonde *system log*.

Antwoord: We volgen de beschreven procedure van drie stappen:

1. We bepalen eerst de twee lijsten:
   a. Committed transactions since last checkpoint: T1, T4, T3
   b. Active transactions: T2
2. Vervolgens doen we een undo voor de write operaties van de actieve transacties, in de omgekeerde volgorde van waarin ze in de log staan. Dit zijn in de log de volgende operaties voor T2:

   [**write**, T2, *B*, 15, 12] [**write**, T2, *D*, 20, 25]

   Dus dat keren we nu om, en draaien daarbij ook oude en nieuwe waarde om:

   [**write**, T2, *D*, 25, 20] [**write**, T2, *B*, 12, 15]

3. Vervolgens doen we een redo voor write operaties van de committed transactions, (T1, T3 en T4) in de volgorde waarin ze in de log voorkomen na de laatste checkpoint. Dat zijn de volgende operaties:

   [**write**, T4, *A*, 10, 20] [**write**, T3, *A*, 20, 30]

In totaal levert dat dus het volgende UNDO/REDO schedule:

[**write**, T2, *D*, 25, 20] [**write**, T2, *B*, 12, 15] [**write**, T4, *A*, 10, 20] [**write**, T3, *A*, 20, 30]

Einde van het tentamen