TECHNISCHE UNIVERSITEIT DELFT
Faculteit Elektrotechniek, Wiskunde en Informatica

Uitwerkingen van het Tentamen

TI2500 Informatie en Datamodellering

Donderdag, 18 april 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. Wat is het verschil tussen een *equijoin* en een *natural join*?

Antwoord: Beide zijn *joins* waarbij records gekoppeld worden op basis van gelijkheid van waarden in bepaalde kolommen, maar het verschil is dat (1) bij de *natural join* altijd precies die kolommen worden gelijkgesteld die dezelfde naam hebben en (2) van deze twee kolommen er vervolgens maar eentje in het resultaat terechtkomt. Bij de equijion is het mogelijk om willekeurige kolommen gelijk te stellen, en worden deze bovendien altijd beide in het resultaat opgenomen.

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 afdelingen waarvan de manager werkt voor een van de projecten van deze afdeling.

Antwoord:
$DPW := DEPARTMENT \bowtie_{Dnumber=Dnum} PROJECT \bowtie_{Pname=Pno} WORKS\_ON;$
$MDPW := \sigma_{Mgr\_ssn=Essn}(DPW);$
$RESULT := \pi_{Dname}(MDPW)$

Uitleg: DPW bevat combinaties van departementen, projecten en werknemers zodanig dat het project van dat departement is en de werknemer voor dat project werkt. In MDPW selecteren we vervolgens alleen de records waar de werknemer dezelfde is als de manager van het departement. Tenslotte projecteren we op de namen van de departementen die overgebleven zijn.

Korter equivalent:
$RESULT := \pi_{Dname}((DEP'T \bowtie_{Dnumber=Dnum} PROJECT) \bowtie_{Pname=Pno,\ Mgr\_ssn=Essn} WORKS\_ON);$

b) Geef de namen van de werknemers die een man als chef hebben en geen vrouwelijk familielid hebben.

Antwoord:
MAN := $\pi_{Ssn}(\sigma_{Sex='M'}(EMPLOYEE))$;
HEEFT_MAN_CHEF := $\pi_{Ssn}(EMPLOYEE * \rho_{Mgr\_ssn}(MAN))$;
HEEFT_VR_FAM := $\pi_{Essn}(\sigma_{Sex='F'}(DEPENDENT)$;
MC_GEEN_VF := HEEFT_MAN_CHEF - $\rho_{Ssn}(HEEFT\_VR\_FAM)$;
RESULT := $\pi_{Fname,Minit,Lname}(EMPLOYEE * MC\_GEEN\_VF)$;

Uitleg: In MAN(Ssn) staan de mannelijke employees. In HEEFT_MAN_CHEF(Ssn) staan de employees die een man als chef hebben. In HEEFT_VR_FAM(Essn) staan de employees met een vrouwelijk familielid. In MC_GEEN_VF(Ssn) staan de ssn's van de employees die een mannelijke chef hebben en geen vrouwelijk familielid. Voor RESULT bepalen we de deelverzameling van EMPLOYEE waarvan de Ssn in MC_GEEN_VF voorkomt en nemen daarvan de kolommen die de naam bevatten.

Deel 2 – Normaalvormen (10 punten)

Vraag 3. Beschouw een relatie met schema R(A, B, C, D, E) met sleutels AE, BE en DE, en de set van functionele dependencies { DE→ABC, A→B, BE→A }. In welke normaalvorm is dit schema? Beargumenteer je antwoord.

Antwoord: Het is in 3NF. Daarvoor hoeven we immers alleen de non-triviale functionele dependencies te beschouwen die in non-prime attributen (dwz. attributen die in geen enkele sleutel zitten) aankomen, en dat is DE→C. Daarvoor geldt dat de linkerkant inderdaad een superkey is, want het is een sleutel. Het is niet in BCNF want dan zou dit voor alle non-triviale functionele dependencies moeten gelden, en dat is niet het geval voor A→B.

Vraag 4. Stel we hebben een relatie R(A, B, C) met *join dependency* JD(AB, AC). Is het dan mogelijk dat de inhoud van de relatie gelijk is aan:{ (a1, b1, c1), (a1, b1, c2) }? Motiveer je antwoord.

Antwoord: Ja, dat is mogelijk. De join dependency geldt precies voor de inhoud van een relatie als R = $\pi_{AB}(R) * \pi_{AC}(R)$. We zien dat $\pi_{AB}(R)$ = { (a1, b1) } en $\pi_{AB}(R)$ = { (a1, c1), (a1, c2) }. Als we hier een natural join op loslaten krijgen we terug { (a1, b1, c1), (a1, b1, c2) }. De JD geldt dus voor deze 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 = { AB→C, AC→D, B→E }. Bewijs met de regels van Armstrong (dus alleen IR1, IR2 en IR3) dat de functionele dependency AB→D in S^+ zit.

Antwoord:
1. AB → C (gegeven)

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

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

Antwoord: Omdat het hier een binaire decompositie betreft kunnen we de *NJB* property gebruiken om dit te beslissen (zie blz. 541 in boek). Die zegt dat de *lossless join property* getest kan worden door te kijken of AC→DB of AC→DE in S^+ zit. Aangezien $\{A,C\}^+ = \{A, C, B_{[A\rightarrow B]}, D_{[C\rightarrow D]}, E_{[AD\rightarrow E]}\}$ geldt dat $\{D, B\} \subseteq \{A, C\}^+$ en $\{D, E\} \subseteq \{A, C\}^+$ en dus zowel AC→DB als AC→DE in S^+ zitten.

Als je het algemene algoritme (Alg. 15.3 op blz. 538/9) hebt toegepast is het antwoord als volgt:

1. Initiele invulling van de tabel:

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

2. Toepassing van A→B

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

3. Toepassing van C→D:

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

4. Toepassing van AD→E:

	A	B	C	D	E
R1	a_1	a_2	a_3	a_4	**a_5**
R2	a_1	a_2	a_3	a_4	a_5

5. Verdere toepassing van de dependencies levert geen verdere veranderingen op.
6. Er is inderdaad een rij met alleen a'tjes, in zowel R1 als R2, dus concluderen we dat de decompositie de *lossless/nonadditive join property* heeft.

Merk op dat we eigenlijk al hadden kunnen stoppen na stap 2, want toen was er al een volledige rij a'tjes.

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

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: { AG }. 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 A→BC af te splitsen en per component de sleutels te bepalen:

- R1(A, B, C) met sleutels { A } en lokale dependencies { A→BC }
- R2(A, D, E, F G) met sleutels { DEFG } en lokale dependencies { AD→E }

Binnen R2 hebben we nu nog de dependencies { AD→E } en deze overtreedt BCNF, dus splitsen we R2, resulterend in:

- R3(A, D, E) met sleutels { AD } en lokale dependencies { AD→E }
- R4(A, D, F, G) met sleutels { ADFG } en lokale dependencies { }

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

We kunnen ook beginnen met B→DF af te splitsen. Dan krijgen we:

- R1(B, D, F) met sleutels { B } en lokale dependencies { B→DF }
- R2(A, B, C, E, G) met sleutels { AEG } en lokale dependencies { A→BC }

Binnen R2 hebben we nu nog de dependencies { A→BC } en deze overtreedt BCNF, dus splitsen we R2, resulterend in:

- R3(A, B, C) met sleutels { A } en lokale dependencies { A→BC }
- R4(A, E, G) met sleutels { AEG } en lokale dependencies { }

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

Tenslotte kunnen we ook beginnen met AD→E af te splitsen. Dan krijgen we:

- R1(A, D, E) met sleutels { AD} en lokale dependencies { AD→E }
- R2(A, B, C, D, F, G) met sleutels { AG } en lokale dependencies { A→BC, B→DF }

Binnen R2 hebben we nu nog twee dependencies die beide BCNF overtreden. Stel we splitsen R2 op basis van A→BC, dan krijgen wel:

- R3(A, B, C) met sleutels { A } en lokale dependencies { A→BC }
- R4(A, D, F, G) met sleutels { ADFG } en lokale dependencies { }

In dat geval zijn we nu in BCNF.
We kunnen eventueel ook R2 splitsen op basis van B→DF, en dan krijgen we:

- R3(B, D, F) met sleutels { B } en lokale dependencies { B→DF }
- R4(A, B, C, G) met sleutels { AG } en lokale dependencies { A→BC }

Merk op dat binnen R4 nu A→BC BCNF overtreedt, dus moeten we ook nog R4 splitsen:

- R5(A, B, C) met sleutels { A } en lokale dependencies { A→BC }
- R6(A, G) met sleutels { AG } en lokale dependencies { }

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:
- R1(A, B, C) met sleutels { A } en lokale dependencies { A→BC }
- R3(A, D, E) met sleutels { AD } en lokale dependencies { AD→E }
- R4(A, D, F, G) met sleutels { ADFG } en lokale dependencies { }

In dit geval mist B→DF en kan ook niet afgeleid worden uit de overgebleven FDs. We zijn dus **niet** *dependency preserving*.

De tweede:
- R1(B, D, F) met sleutels { B } en lokale dependencies { B→DF }
- R3(A, B, C) met sleutels { A } en lokale dependencies { A→BC }
- R4(A, E, G) met sleutels { AEG } en lokale dependencies { }

In dit geval mist AD→E en kan ook niet uit de overgebeleven FDs worden afgeleid, We zijn dus **niet** *dependency preserving*.

De derde:
- R1(A, D, E) met sleutels { AD} en lokale dependencies { AD→E }
- R3(A, B, C) met sleutels { A } en lokale dependencies { A→BC }
- R4(A, D, F, G) met sleutels { ADFG } en lokale dependencies { }

In dit geval mist B→DF en kan ook niet uit de overgebeleven FDs worden afgeleid, We zijn dus **niet** *dependency preserving*.

De vierde:
- R1(A, D, E) met sleutels { AD} en lokale dependencies { AD→E }
- R3(B, D, F) met sleutels { B } en lokale dependencies { B→DF }
- R5(A, B, C) met sleutels { A } en lokale dependencies { A→BC }
- R6(A, G) met sleutels { AG } en lokale dependencies { }

In dit geval zijn alle oorspronkelijke dependencies aanwezig als lokale dependencies, en dus zijn we *dependency preserving*.

Deel 4 – File organizatie (5 punten)

Vraag 8. Een bestand heeft r = 20,000 STUDENT records van vaste lengte. Elke record heeft de volgende velden: **Name** (30 bytes), **Ssn** (9 bytes), **Address** (40 bytes), **Phone** (10 bytes), **Birth_date** (8 bytes), **Sex** (1 byte), **Major_dept_code** (4 bytes),

Minor_dept_code (4 bytes), **Class_code** (4 bytes, integer), and **Degree_program** (3 bytes). Een extra byte is gebruikt als *deletion marker*. Het bestand is opgeslagen op een schijfeenheid waarvan de parameters zijn: *block size* B = 512 bytes; *interblock gap* grootte G =128 bytes; aantal blocks per track = 20; aantal tracks per vlak = 400. Een disk pack bestaat uit 15 dubbelzijdige disks. De schijven draaien 2400 rpm, en de *average seek time* is 30 msec. Beantwoord de volgende vragen:

a) Bereken de *record size* R in bytes.

Antwoord: R = 30 + 9 + 40 + 10 + 8 + 1 + 4 + 4 + 4 + 3 + 1 = 114 bytes

b) Bereken de *blocking factor* bfr en het aantal *file blocks* b, aangenomen dat er geen sprake is van *record spanning* in de bestandsorganizatie.

Antwoord: In dat geval geldt bfr = $\lfloor$ B/R $\rfloor$, dus bfr = $\lfloor$ 512 / 114 $\rfloor$ = 4. Dus voor 20.000 records zijn dan $\lceil$ 20.000 / 4 $\rceil$ = 5.000 blocks nodig.

c) Bereken de gemiddelde zoektijd om een record te vinden met *linear search* als de file blocks aansluitend (*contiguously*) zijn opgeslagen, en *double buffering* wordt gebruikt.

Antwoord: De tijd wordt bepaald door (1) de tijd om bij de eerste track te komen en (2) de tijd per track om die track in te lezen. Door aansluitende opslag en double buffering hoeven we niet tussen de tracks te wachten. De tijd voor (1) is gegeven als 30 msec (de *average seek time*). Het aantal blocks wat we gemiddeld moeten lezen is 5.000 / 2 = 2.500. Daarvoor moeten we 2.500 / 20 = 125 tracks lezen. Het lezen van een track duurt (1 / 2400 rpm) × 60 sec = 0.025 sec. = 25 msec. Dus in totaal kost het: 30 msec + (125 × 25 msec) = 3155 msec. Als we ook nog rekening mee houden dat het bestand misschien niet aan het begin van een track start, dan moeten we gemiddeld nog een halve track extra lezen, en krijgen we dus: 30 msec + (125.5 × 25 msec) = 3167.5 msec.

d) Neem aan dat het bestand gesorteerd is op **Ssn**; bereken hoe lang het duurt bij een *binary search* om een record te vinden als het **Ssn** is gegeven.

Antwoord: We moeten gemiddeld $\lceil \log_2(2.500) \rceil$ = 12 blocks lezen voor we het record vinden. Voor elk blok is er de seek time om de track te vinden en dan gemiddeld de helft van de track aflopen: 30 msec + 12.5 msec = 42.5 msec. Dus in totaal: 12 × 42.5 msec = 510 msec.

Deel 5 – Indexering (10 punten)

Vraag 9. Een PARTS bestand met Part# als sleutelveld bevat records met de volgende Part# waardes: 47, 72, 55, 59, 61, 45, 93, 17, 20, 22, 66, 36. Stel dat de 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 boom er uitziet.

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

```
[47]
========
[47, 72]
========
[47, 55, 72]
========
 . [59, 72]
55
 . [47, 55]
========
 . [59, 61, 72]
55
 . [47, 55]
========
 . [59, 61, 72]
55
 . [45, 47, 55]
========
 . [72, 93]
61
 . [59, 61]
55
 . [45, 47, 55]
========
 . [72, 93]
61
 . [59, 61]
55
 . [47, 55]
45
 . [17, 45]
========
 . [72, 93]
61
 . [59, 61]
55
 . [47, 55]
45
 . [17, 20, 45]
========
```

```
 .   . [72, 93]
 . 61
 .   . [59, 61]
 . 55
 .   . [47, 55]
45
 .   . [22, 45]
 . 20
 .   . [17, 20]
========
 .   . [66, 72, 93]
 . 61
 .   . [59, 61]
 . 55
 .   . [47, 55]
45
 .   . [22, 45]
 . 20
 .   . [17, 20]
========
 .   . [66, 72, 93]
 . 61
 .   . [59, 61]
 . 55
 .   . [47, 55]
45
 .   . [22, 36, 45]
 . 20
 .   . [17, 20]
========
```

Deel 6 – Queryoptimalisatie (20 punten)

Vraag 10. Omschrijf kort het verschil tussen *heuristic* en *cost-based* optimization.

Antwoord: In *heuristic optimization* worden er een aantal vuistregels op het query evaluation plan toegepast om het te transformeren in een optimaler plan. Elke regel is op basis van algebraïsche identiteiten voor de relationele algebra en zodanig gekozen dat deze zeer waarschijnlijk tot een optimaler plan leidt, zoals bijvoorbeeld het vervroegen van projecties en selecties om tussenresultaten te verkleinen. Bij *cost-based optimization* wordt er een reeks equivalente *query evaluation plans* gegenereerd, waarvan vervolgens de kosten worden geschat. Het plan met de laagste schatting wordt dan gekozen en uitgevoerd.

Vraag 11. Een bestand van 4096 blocks moet gesorteerd worden met een beschikbare bufferruimte van 16 blocks. Hoeveel *passes* zijn er nodig in de *merge*-fase van het *external sort-merge* algoritme?

Antwoord: Na de eerste fase hebben we $\lfloor$ 4096/16 $\rfloor$ = 256 *initial runs*. Vervolgens kunnen we in elke pass 15 *runs mergen*. Dus zijn er $\lceil \log_{15}(256) \rceil = 3$ passes nodig.

Vraag 12. Ontwikkel de kostenfunctie voor het INTERSECTION algoritme zoals besproken in sectie 18.4 op basis van sort-merge. Zie ook figuur 18.3 op blz. 671 voor het algoritme, en sectie 18.8.3 voor voorbeelden van kost-functies. Net zoals in 18.8.3 dienen de kostfuncties een schatting te geven van de hoeveelheid block-transfers.

Antwoord: Het algoritme zal eerste de twee input relaties sorteren en daarna met een sort-merge de twee mergen in een enkele pass over beide relaties. De kosten voor sorteren wordt geschat door (zie blz. 664) de formule: sort(b, n_B) = $2b + 2b.\log_{dM}(n_R)$. Hierbij veronderstellen we: b = aantal blocks van de te sorteren file, dM = graad van merging en n_R = aantal initiele runs. Deze worden op hun beurt als volgt berekend: dM = **min**(n_B-1, n_R) en $n_R = \lceil (b/n_B) \rceil$ waar n_B de beschikbare bufferruimte in blocks is. We vatten deze formule samen als sort(b, n_B). Voor files van grootte b_1 en b_2 krijgen we dan dus de volgende kostenfunctie voor de INTERSECTION:

$$\text{sort}(b_1, n_B) + \text{sort}(b_2, n_B) + b_1 + b_2 + \textbf{min}(b_1,b_2)$$

De eerste twee componenten sort(b_1, n_B) + sort(b_2, n_B) komen voort uit het feit dat we de twee bestanden apart sorten, en daarvoor waarschijnlijk dezelfde bufferruimte beschikbaar hebben. De volgende component $b_1 + b_2$ + **min**(b_1,b_2) komt doordat we alle blocks weer moet lezen (dus $b_1 + b_2$ blocks lezen) en ook weer terug moeten schrijven maar alleen de records die in beide relaties voorkomen, dus ten hoogste **min**(b_1,b_2) blocks schrijven.

Vraag 13. Omschrijf kort het verschil tussen *pipelining* en *materialization*.

Antwoord: Bij *materialization* wordt bij het uitvoeren van een *query evaluation plan* de operaties 1 voor 1 uitgevoerd, en aan het eind van de executie van een operatie wordt het resultaat volledig in het geheugen opgebouwd en opgeslagen. Bij pipelining worden de operaties zoveel mogelijk tegelijkertijd uitgevoerd waarbij een operatie die invoer verwacht deze stapsgewijs, record voor record, opvraagt van de voorgaande operatie en dan verwerkt. Dat betekent dat alleen de records zoals deze uitgewisseld worden tijdelijk in het geheugen worden gematerialiseerd en niet het complete tussenresultaat.

Deel 7 – Databasetuning (5 punten)

Vraag 14. Geef een voorbeeld van *vertical partitioning* en leg uit waarom dit de database onder sommige omstandigheden efficiënter maakt.

Antwoord: Bijvoorbeeld de tabel EMPLOYEE(Ssn, Name, Phone, Grade, Salary) kan gesplitst worden in EMP1(Ssn, Name, Phone) en EMP2(Ssn, Grade, Salary). Dit maakt

queries die alleen de kolommen uit één van EMP1 of EMP2 nodig hebben sneller, want deze queries hebben dan met kleinere tabellen te maken die dus sneller te scannen en te doorzoeken zijn. Dit is dus gunstig als de verdeling in kolommen samenvalt met de benodigde informatie voor bepaalde taken. Als echter kolommen uit beide tabellen nodig zijn, zal er een join nodig zijn en de queries dus juist trager worden.

Deel 8 – Transacties (10 punten)

Vraag 15. Beschouw de drie transacties T1, T2, and T3, en de schedules S1 en S2 van deze transacties zoals beneden gegeven. Teken de *precedence graph* voor S1 en S2, en bepaal welk schedule serialiseerbaar is. Als een schedule serialiseerbaar is, geef dan een equivalent *serial schedule*.

T1: r1(X); r1(Z); w1(Z);
T2: r2(Y); w2(Y); r2(X); w2(X);
T3: r3(Z); r3(X); w3(Z); r3(Y); w3(Y);

S1: r2(Y); r3(Z); w2(Y); r1(X); r2(X); r3(X); w3(Z); r3(Y); w2(X); r1(Z); w3(Y); w1(Z);
S2: r2(Y); r3(Z); r3(X); w3(Z); w2(Y); r2(X); r3(Y); w2(X); r1(X); r1(Z); w3(Y); w1(Z);

Antwoord:

De *precedence* graaf voor S1 bevat de volgende pijlen:

T2 → T3 want w2(Y) voor r3(Y)/w3(Y)
T3 → T1 want r3(Z) voor w1(Z), w3(Z) voor r1(Z)
T1 → T2 want r1(X) voor w2(X)
T3 → T2 want r3(X) voor w2(X)

Er zijn diverse cykels, bvb. T2 → T3 → T2, en T1 → T2 → T3 → T1. Dus S1 is niet serialiseerbaar.

De *precedence* graaf voor S2 bevat de volgende pijlen:

T2 → T3 want r2(Y) voor w3(Y), w2(Y) voor r3(Y)/w3(Y)
T3 → T1 want r3(Z) voor w1(Z), w3(Z) voor r1(Z)
T3 → T2 want r3(X) voor w2(X)
T2 → T1 want w2(X) voor r1(X)

Er is een cykel T2 → T3 → T2. Dus S2 is 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] w(Y); [3] r(Z); [4] w(Y); [5] w(X); [6] w(U); [7] w(Z); [8]

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: Merk op dat gevraagd is om een locking schema waarbij aan alle drie de eisen voldaan is. 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] : write_lock(Y)
- [3] : read_lock(Z)
- [4] : -
- [5] : write_lock(X)
- [6] : write_lock(U)
- [7] : write_lock(Z); unlock(U); unlock(X); unlock(Y)
- [8] : unlock(Z)

In dit geval is het aantal stappen dat de resources worden bezet: U(1), X(6), Y(5), 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] : write_lock(Y)
- [3] : read_lock(Z)
- [4] : -
- [5] : write_lock(X); write_lock(U); write_lock(Z); unlock(Y)
- [6] : unlock(X)
- [7] : unlock(U)
- [8] : unlock(Z)

In dit geval is het aantal stappen dat de resources worden bezet: U(2), X(4), Y(3), Z(5). 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:
We volgen de executie van S1. De timestamps van de transacties worden aangenomen te zijn bepaald door hun eerste operatie, dus TS(T1) = 4, TS(T2) = 1 en TS(T3) = 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	r2(Y)	-	-	1	-	-	-
2	r3(Z)	-	-	1	-	2	-
3	w2(Y)	-	-	1	1	2	-
4	r1(X)	4	-	1	1	2	-
5	r2(X)	4	-	1	1	2	-
6	r3(X)	4	-	1	1	2	-
7	w3(Z)	4	-	1	1	2	2
8	r3(Y)	**4**	-	2	1	2	2
9	w2(X)		**1**				
10	r1(Z)						
11	w3(Y)						
12	w1(Z)						

Op *T*=9 is er een conflict want Rd_ts(X) = 4 > TS(T2) = 1. Dus dit schedule is niet 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(T1) = 10, TS(T2) = 1 en TS(T3) = 2.

T	Operat.	Rd_ts(*X*)	Wr_ts(*X*)	Rd_ts(*Y*)	Wr_ts(*Y*)	Rd_ts(*Z*)	Wr_ts(*Z*)
0		-	-	-	-	-	-
1	r2(Y)	-	-	1	-	-	-
2	r3(Z)	-	-	1	-	2	-
3	r3(X)	2	-	1	-	2	-
4	w3(Z)	2	-	1	-	2	2
5	w2(Y)	2	-	1	1	2	2
6	r2(X)	2	-	1	1	2	2
7	r3(Y)	**2**	-	2	1	2	2
8	w2(X)		1				
9	r1(X)						
10	r1(Z)						
11	w3(Y)						
12	w1(Z)						

Op T=8 is er een conflict want Rd_ts(X) = 2 > TS(T2) = 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 RDU_M (NO-UNDO/REDO with checkpoints) volgen (zie blz. 794). Welke operaties worden dan in welke volgorde uitgevoerd? Gebruik dezelfde notatie als voor de getoonde *system log*.

Antwoord: We volgen de beschreven procedure van twee stappen:

1. We bepalen eerst de twee lijsten:
    a. Gecommitteerde transacties sinds laatste checkpoint: T1, T4, T3
    b. Actieve transacties: T2
2. Vervolgens doen we een redo voor de write operaties van de gecommitteerde transacties in de volgorde waarin ze in de log staan. Dit zijn in de log de volgende operaties voor T1, T4 en T3:

    [**write**, T1, *D*, 15, 20] [**write**, T4, *B*, 25, 15] [**write**, T4, *A*, 10, 20]
    [**write**, T3, *A*, 20, 30]

    Merk op dat we de ook de operaties voor de checkpoint opnieuw moeten doen, want onder *deferred update* zijn deze updates nog niet op schijf gezet.

Einde van het tentamen