Oppgave 1: Lagringsstrukturer og queryutføring
Anta at vi har følgende tabell som lagrer informasjon om oppmeldinger til eksamener:

Eksamensregistrering(Studentnr, Eksamennr, Dato, Status)

Anta videre at vi har 4 lagringsalternativer som er mulige:

1. Heapfil.
2. Clustered B+-tre med Eksamennr som søkenøkkel. Treet har 4 nivåer.
3. Clustered B+-tre med (Eksamennr, Studentnr) som søkenøkkel. Treet har 4 nivåer.
4. Clustered, statisk hash-struktur med Eksamennr som søkenøkkel. Gjennomsnittlig aksesseres 1.25 blokker per søk på Eksamennr (ettersom overflyt her håndteres ved at posten lagres i første etterfølgende blokk med ledig plass).

For hver av SQL-setningene nedenfor, argumenter for hvilket lagringsalternativ over du ville brukt. Anta for hver SQL-setning at denne typen setning er mest brukt.

Antar vi skal velge den mest effektive lagrinsstrukturen med tanke på den aktuelle spørringen?

a) SELECT * FROM Eksamensregistrering;

Heapfil. Siden vi bare skal ha ut all informasjon, vil det aksesseres færrest mulig blokker med heap. Usortert.

b) SELECT * FROM Eksamensregistrering WHERE Eksamennr = 963432;

B+ Tre med Eksamensnummer. For å hente ut eksamensnummer i, behøver bare gå innom 4 blokker, 1 i hvert nivå.

c) SELECT * FROM Eksamensregistrering WHERE Eksamennr = 963432 ORDER BY Studentnr DESC;

Clustered B+ tre med (eksamensnr,studnr) som nøkkel. Studentnr er sortert i treet, og dermed vil det være raskere å hente ut dataentryen.

d) INSERT INTO Eksamensregistrering (14556589, 963439, '26.11.2018', "Øvingsopplegg bestått");

Heap er usortert, så ved insert av en ny indeks, vil den bare settes inn første ledige blokk. Heap raskest.

Oppgave 2: Nested-loop-join
Gitt at vi har en tabell **Student**(Studentnr, Navn, Fødselsår, InstituttID) som skal joines med **Eksamensregistrering**(Studentnr, Eksamennr, Dato, Status) fra forrige oppgave. Student er lagret med 47 000 poster i 800 blokker og Eksamensregistrering er lagret med 500 000 poster i 12 800 blokker. Vi ønsker å bruke nested-loop-join, det vil si at vi for hver blokk i den ene tabellen scanner hele den andre tabellen og ser etter når Studentnr er like.

Hvis vi har 34 blokker tilgjengelig i buffer, hvor mange blokker leses totalt i løpet av joinen?

800 blokker i Student. 12800 blokker i Eksamensregistrering, 34 blokker i buffer.

Buffer 34 blir 32 blokker til tabell 1, 1 til tabell 2 og 1 til resultat. Student har 800 blokker / 32 = 25 innlesinger.

Totalt antall blokker lest blir da

800(student) + 25*12800(eksamensregistrering) = 320800 blokker lest.

(Hadde man brukt omvendt, med student som indre input, ville totalt leste blitt 332800)

Oppgave 3: Transaksjoner

a) Nevn to årsaker til hvorfor vi ønsker å ha transaksjoner i utgangspunktet.

Muliggjør flerbrukerfunksjonalitet ved å hindre direkte tilgang til data for enkeltbrukere.
Sikrer samtidighet i data, at dataene ikke endres under acces.

b) Forklar kort ACID-egenskapene til transaksjoner.

Atomic: en enkelt "udelelig" transaksjon som ikke kan forstyrres. Om noe skjer under transaksjonen, skal hele avbrytes. Ingen "halvveis-operasjoner".

Consistency: Alle endringer må være gyldige(i databasetermer).

Isolation: Parallelle transaksjoner skal ikke være synlige for andre tansaksjoner mens de pågår, og slutttilstanden av databasen etter paralelle isolerte skal være den samme som fra flere serielle transaksjoner..

Durability: Lagring av database til ikke-volatilt minne. Dette sikrer data-intigritet ved strømbrudd, systemkræsj o.l.

c) Avgjør recovery-egenskapene (strict, ACA, recoverable, unrecoverable) til følgende historier:

H1 : w1(X); r2(X); w1(X); w3(Y); w2(Y); c2; c1; c3

Unrecoverable- T2 leser x etter w1(x) og committer før T1.Ved en abort etter c2, vil man ikke kunne gjenskape x

H2 : w1(X); w3(Y); w2(Y); c1; r2(X); r3(X); c2; c3

ACA - ingen galopperende aborts, r2(x) kommer etter T1 committer.

H3 : r3(Y); w3(Y); r1(Y); w2(X); c2; w1(X); c3; c1

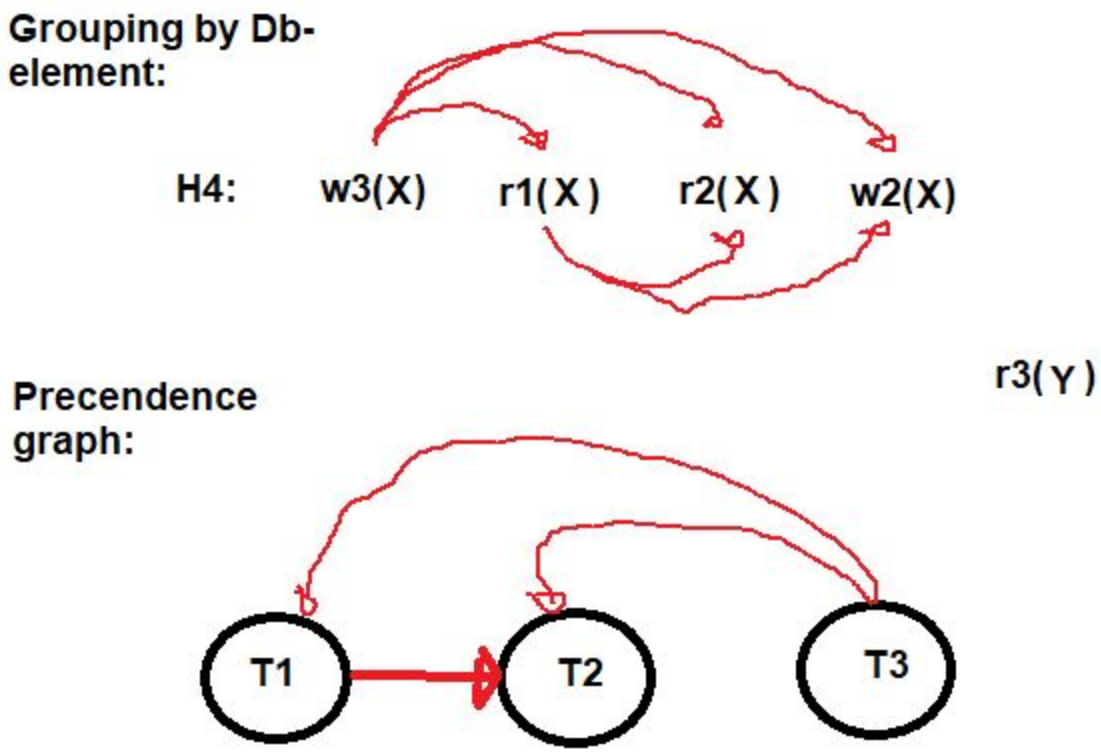
Recoverable - r1(Y) etter w3(Y), men C3 kommer før C1.

d) Når er to operasjoner i en historie i konflikt?

Hvis de:

- Er fra forskjellige transaksjoner
- Gjelder samme dataelement
- Minst en av operasjonene er er write-operasjon
- Hvis rekkefølgen i historien er avhengig/dependent for utfallet.

e) Undersøk om følgende historie, H4 , er konfliktserialiserbar ved å tegne presedensgrafen til historien. H4 : w3(X); r1(X); r2(x) w2(X); r3(Y); c1; c2; c3



Historien er konfliktseriealiserbar siden det ikke er sykler i presedensgrafen:

f) Hva vil det si at vi har en vranglås (dead-lock) mellom to eller flere transaksjoner?

Transaksjoner venter på informasjon eller tillatelse fra annen transaksjon, som også er låst i vente på å gå videre. Systemet er låst.

g) Vi ønsker å skrive om H4 slik at den gjør bruk av låser. Vis hvordan historien ser ut med rigorous tofaselåsing. Innfør operasjonene rl1(A), wl1(A) og ul1(A) - det vil si read_lock1(A), write_lock1(A) og unlock1(A). Husk at leselåser kan deles av flere, mens skrivelåser er ekslusive.

H4 med rigorous 2PL:

T1	T2	T3
		writelock(x)
readlock(X)		write(x)
wait	readlock(x)	
wait	wait	readlock(Y)
wait	wait	read(Y)
wait	wait	commit/unlock(x)(Y)
read(x)	wait	
commit/unlock(x)	wait	
	read(X)	
	readlock->writelock(x)	
	write(x)	
	commit/unlock(x)	

Oppgave 4: Recovery etter krasj med Aries

a) Tenk deg at strømmen har gått midt under noen viktige databaseoperasjoner på dataelementene A og B. Vi er i analysefasen i Aries og skal finne vinnertransaksjonene som skal være permanente og senere få REDO, og tapertransaksjoner som senere skal aborteres (UNDO). Vi restarter systemet og får opp følgende logg:

LSN	Prev_LSN	Transaction	Operation	Page_ID	...
147			End_ckpt		
148	0	T_1	Update	A	
149	0	T_2	Update	B	
150	149	T_2	Commit		
151	148	T_1	Update	A	
152	0	T_3	Update	B	
Krasj!					

Hvilke transaksjoner her er vinnere og tapere? Begrunn svaret.

T1 er taper, jobbet med en side A når systemet kræsjet. Ingen commit.
T3 er taper, oppdaterte B når kræsj, ingen commit.

T2 gjorde oppdatering og commitet før kræsjet.

b) Før vi kan gjennomføre UNDO og REDO må vi lage en transaksjonstabell og en Dirty Page Table (DPT). Anta at både transaksjonstabellen og DPT er helt tomme i sjekkpunktet funnet i loggposten med LSN 147. Hvordan ser transaksjonstabellen og DPT ut etter at analysefasen er ferdig? (Tips: Transaksjonstabellen skal inneholde transaksjonene med tilstand og Last_LSN, dvs. nummeret til den nyeste loggposten for transaksjonen. DPT skal inneholde Page_ID og Recovery_LSN, dvs. nummeret til den eldste loggposten som først gjorde pagen skitten.)

Transaksjonstabell:

TransactionID	**lastLSN**	**Status**
T2	150	Committed
T1	148	In progress
T3	0	In prgress

Dirty Page Table:

PageID	**recoveryLSN**
A	148
B	152