HW5

1.

```
Select count(s.id) as SolutionCount
From Issue i, Alternative a, Solution s
Where s.id=a.solution and a.id=i.id
Group By i.id
Union
Select count(c2.id) as CriteriaCount
From Issue i2, Alternative a2, Solution s2, Criterion c2, Posting p2
Where s2.id=p2.id and s2.id=a2.solution and a2.id=i2.id and c2.id=p2.id
Group By i2.id
Union
Select count(arg3.id) as ArgumentCount
From Issue i3, Solution s3, Argument arg3, Alternative a3, Posting p3
Where s3.id=p3.id and s3.id=a3.solution and a3.id=i3.id and arg3.concerns=s3.id and
arg3.id=p3.id
Group By i3.id
```

2.

```
Update support
Set value=-2
Where level=”strongly opposed to”

Update support
Set value=-1
Where level=”opposed to”;

Update support
SET value=0
Where level=”neutral”;

Update support
SET value=1
Where level=”supportive”;


Update support
SET value=2
Where level=”very supportive”;

Select i.id, a.solution, sum(sup.level)
From Issue i, Alternative a, Solution s, Support sup
Where i.id=a.id and a.solution=s.id and s.id=sup.solution
Group by i.id, a.solution
```

3. For each person (author name) show how many arguments posted by that person were favorable to an alternative and how many were unfavorable.

```
Select count(arg.id) as FavorCount
From Posting p, Author a, Argument arg, Solution s, Alternative alt,
Where a.posting=p.id
    and s.id=p.id
    and alt.solution=s.id
    and arg.id=p.id
    and arg.concerns=s.id
    and arg.isFavorableToward=true
Group By a.name
Union
Select count(arg.id) as NotFavorCount
From Posting p2, Author a2, Argument arg2, Solution s2, Alternative alt2,
Where a2.posting=p2.id
    and s2.id=p2.id
    and alt2.solution=s2.id
    and arg2.id=p2.id
    and arg2.concerns=s2.id
    and arg2.isFavorableToward=false
Group By a2.name
```

HW6

| Table | Record Size | Table Size | Records per full block | Per 2/3 full block | No. full blocks | No. 2/3 blocks |
|---|---|---|---|---|---|---|
| Paint | 60 | 1M | 273.1 | 182.0 | 3661.7 | 5494.5 |
| Portion | 118 | 10M | 138.8 | 92.6 | 72046.1 | 107991.4 |

(1) Using portion table, find all paints that use oil
Heap:
The portion table has 10M records and it takes 72046.1 blocks. The scan is sequential so the scan will take 10ms to read the first block, and (72046.1-1)*0.1ms to read others. The total time 10+7204.5=7214.5ms

Primary B-Tree:
There are 10M=$10^7$ portions. Oil is a material, and there are 1k=$10^3$ Material. That means for one kind of material, we have $10^7/10^3=10^4$ records on average in this Portion table.
It takes 2 random disk accesses to retrieve the first relevant leaf block which requires 20 ms.

In this block, we only retrieve 92.6 records (because this block is 2/3 full), so we still need to find (10000-92.6) =9907.4 records. These records require 1.5*(9907.4/138.8) = 107.1 blocks.
The total time 20+107.1*10=1091ms

Secondary B-Tree:
It needs to retrieve the $10^4$ records which require their own random I/O.
The total time $10^4$*10=$10^5$ms

(2) Find all red paints
Heap:
The paint table has 1M records and it takes 3661.7 blocks. The scan is sequential so the scan will take 10ms to read the first block, and (3661.7-1)*0.1ms to read others.
The total time 10+366.2=376.2ms

Primary B-Tree:
There are 1M=$10^6$ paints, and 10K=$10^4$ colors. That means for one kind of color, we have $10^6/10^4$=100 records on average.
It takes 2 random disk accesses to retrieve the first relevant leaf block which requires 20 ms.
In this block, we only retrieve 182 records (because this block is 2/3 full). So it is likely all the records we look for are in the same block.
The total time 20ms

Secondary B-Tree:
It is very likely that all the records (100) will be in the same leaf block of the secondary index which can be found in 10ms. However, each of the 100 records requires its own random I/O. The total time is 10+10*100ms=1010ms

(3) Find all red paints with at least 50% latex
Heap:
We first search the paint table, and time on this is 376.2ms (as in (2))
Then we search the portion table, and time on this is 7214.5ms(as in (1))
The total time is 376.2+7214.5=7590.7ms

Primary B-Tree:
The first query will find the red paints, and there are 100 records. This will take about 20ms(as in (2))
Then we'll execute a query in the portion table. We know the average number of portion per material is $10^7/10^3=10^4$. However, we don't know the distribution for latex in this table, so we assume the average number of portion with latex>50% are 50% of this number. So we need to retrieve 5000 records. It takes 2 random disk accesses to retrieve the first relevant leaf block which requires 20 ms. In this block, we only retrieve 92.6 records (because this block is 2/3 full). We still need to retrieve (5000-92.6)=4907.4

records, and they take 1.5*4907.4/138.8=53.0 blocks. So time for this query will take 20+53*10=550ms
The total time is 20+550=570

how to make the assumption is on http://www.ccs.neu.edu/home/kenb/cs3200/resources/a5ans.html)

Secondary B-Tree
The first query will retrieve the red records, and this takes 1010ms (as in (2)).
The second query will retrieve 5000 records, and this takes 10+10*5000=50010ms
The total time=51020ms

(4) Insert a new color
There is no color table, so all time is 0.

(5) Modify the color of a paint
Heap:
There are 1M paints, and they take 3661.7 blocks, so we need on average 10+(3661.7 -1)/2*0.1=193.0ms to find the paint.
Then we modify it and write it back, which takes another 193.0ms
The total time=386ms

Primary B-Tree:
It takes 3 random I/Os.
The total time=30ms

Secondary B-Tree:
Same as Primary B-Tree
The total time=30ms

You can find this on http://www.ccs.neu.edu/home/kenb/cs3200/resources/a5ans.html

HW8

I use the following statement from the book:
Every conflict serializable schedule is serializable, if we assume that the set of items in the database does not grow or shrink. Every conflict serializable schedule is view serializable, although the converse is not true.

A schedule is said to be strict if a value written by a transaction T is not read or overwritten by other transactions until T either aborts or commits.

1. R1X, R3Z, W2Y, R3Z, R2Z, W1Y, W2Z, R1X, W3Z, C3, C1, C2
   R3(z)->W2(z),

W2(y)->W1(y)

**R3(z)->W2(z)**

**R2(z)->W3(z)**

Not Conflict serializable

6 Possible serial orders:

R1(x), W1(y), R1(x), C1, W2(y), R2(z), W2(z), C2, R3(z), R3(z), W3(z), C3

The first R3(z) in the original schedule reads the value of (z)

The first R3(z) in the serial order reads the value of W2(z)

R1(x), W1(y), R1(x), C1, R3(z), R3(z), W3(z), C3, W2(y), R2(z), W2(z), C2

R2(z) in the original schedule reads the value of (z)

R2(z) in the serial order reads the value of W3(z)

W2(y), R2(z), W2(z), C2, R3(z), R3(z), W3(z), C3, R1(x), W1(y), R1(x), C1

The first R3(z) in the original schedule reads the value of (z)

The first R3(z) in the serial order reads the value of W2(z)

W2(y), R2(z), W2(z), C2, R1(x), W1(y), R1(x), C1, R3(z), R3(z), W3(z), C3

The first R3(z) in the original schedule reads the value of (z)

The first R3(z) in the serial order reads the value of W2(z)

R3(z), R3(z), W3(z), C3, R1(x), W1(y), R1(x), C1, W2(y), R2(z), W2(z), C2

R2(z) in the original schedule reads the value of (z)

R2(z) in the serial order reads the value of W3(z)

R3(z), R3(z), W3(z), C3, W2(y), R2(z), W2(z), C2, R1(x), W1(y), R1(x), C1

R2(z) in the original schedule reads the value of (z)

R2(z) in the serial order reads the value of W3(z)

Thus not view serializable

Not seriablizable

Not strict: because W2(y) happens before W1(y), but C2 ends after C1.

2. W1(X), R1(Z), R3(X), R1(X), R2(Z), R3(Y), C1, W2(Z), W2(Y), R2(Y), W3(X), C3, C2

   W1(x)->R3(x)

   W1(x)->W3(x)

   R1(z)->W2(z)

   R1(x)->W3(x)

   R3(y)->W2(y)

   Conflict-serializable,

   Each conflict serializable schedule is view serializable, thus it’s view serializable

   Every conflict serializable schedule is serializable, thus it’s serializable

   Not Strict: because W1(x) happens before R3(x), but T1 commits after R3(x)

3. R2(Z), W2(Z), W1(X), W2(Y), R3(X), R1(Z), R3(Y), W3(X), R2(Y), W1(Z), C1, C2, C3

R2(z)->W1(z)
W2(z)->R1(z)
W2(z)->W1(z)
W1(x)->R3(x)
W1(x)->W3(x)
W2(y)->R3(y)
Conflict-serializable,
Each conflict serializable schedule is view serializable, thus it's view serializable
Every conflict serializable schedule is serializable, thus it's serializable
Not Strict: because W2(z) happens before R1(z) but C2 ends after C1

4. R3(X), R2(Z), W1(X), R1(Z), W2(Z), R1(X), W2(Y), R3(Y), W3(X), C3, W1(Z), C1, R2(Y), C2
R3(x)->W1(x)
R2(z)->W1(z)
W1(x)->W3(x)
**R1(z)->W2(z)**
**W2(z)->W1(z)**
R1(x)->W3(x)
W2(y)->R3(y)
Not Conflict serializable
"R3(y) reads W2(y)" means C2 comes before C3. Thus only 3 possible serial orders.
W1(x), R1(z), R1(x), W1(z), C1, R2(z), W2(z), W2(y),R2(y),C2,R3(x),R3(y),W3(x),C3
R3(x) in the original schedule reads the value of (x)
R3(x) in the serial order reads the value of W1(x)
R2(z), W2(z), W2(y),R2(y),C2,R3(x),R3(y),W3(x),C3, W1(x), R1(z), R1(x), W1(z), C1
R1(z) in the original schedule reads the value of (z)
R1(z) in the serial order reads the value of W2(z)
R2(z), W2(z), W2(y),R2(y),C2, R1(z), R1(x), W1(z), C1, R3(x),R3(y),W3(x),C3
R1(z) in the original schedule reads the value of (z)
R1(z) in the serial order reads the value of W2(z)
Not view serializable
Not serializable
Not Strict: because W1(x) happens before W3(x), but C3 ends before C1.

5. R1(A), R1(B), W1(A), R2(A), W1(B), R3(C), C1, W2(C), W2(B), R2(C), W3(A), C3, C2
R1(a)->W3(a)
R1(b)->W2(b)
W1(a)->R2(a)
W1(a)->W3(a)
**R2(a)->W3(a)**
W1(b)->W2(b)
**R3(c)->W2(c)**
Not Conflict serializable

"R2(a) reads W1(a)" means C1 comes before C2. Thus only 3 possible serial orders.

R1(a),R1(b),W1(a),W1(b),C1,R2(a),W2(c),W2(b),R2(c),C2,R3(c),W3(a),C3

R3(c) in the original schedule reads the value of (c)

R3(c) in the serial order reads the value of W2(c)

R1(a),R1(b),W1(a),W1(b), C1,R3(c),W3(a),C3, R2(a),W2(c),W2(b),R2(c),C2

R2(a) in the original schedule reads the value of W1(a)

R2(a) in the serial order reads the value of W3(a)

R3(c),W3(a),C3, R2(a),W2(c),W2(b),R2(c),C2, R1(a),R1(b),W1(a),W1(b), C1

R2(a) in the original schedule reads the value of W1(a)

R2(a) in the serial order reads the value of W3(a)

Not view serializable

Not serializable

Not Strict: because W1(a) happens before R2(a) but T1 commits after R2(a).

6. R2(A), W2(C), R1(A), W2(B), R1(B), R3(C), W3(A), R2(C), W1(B), C1, C2, C3

R2(a)->W3(a)

W2(c)->R3(c)

R1(a)->W3(a)

W2(b)->W1(b)

Conflict-serializable,

Each conflict serializable schedule is view serializable, thus it's view serializable

Every conflict serializable schedule is serializable, thus it's serializable

Not Strict: "W2(b)->W1(b)" means C2 should commit before C1, but in the schedule we have C1 commits before C2.

7. R3(B), R2(A), R3(C), W3(A), R1(A), R1(B), W2(C), C3, W1(A), W2(B), W1(B), C1, R2(C), C2

R3(b)->W2(b)

R3(b)->W1(b)

**R2(a)->W3(a)**

R2(a)->W1(a)

**R3(c)->W2(c)**

W3(a)->R1(a)

W3(a)->W1(a)

R1(b)->W2(b)

W2(b)->W1(b)

Not Conflict serializable

"W3(a)->R1(a)" means C3 comes before C1. Thus only 3 possible serial orders.

R3(b),R3(c),W3(a),C3,R1(a),R1(b),W1(a),W1(b),C1,R2(a),W2(c),W2(b),R2(c),C2

R2(a) in the original schedule reads the value of (a)

R2(a) in the serial order reads the value of W1(a)

R3(b),R3(c),W3(a),C3, R2(a),W2(c),W2(b),R2(c),C2, R1(a),R1(b),W1(a),W1(b),C1

R2(a) in the original schedule reads the value of (a)

R2(a) in the serial order reads the value of W3(a)

R2(a),W2(c),W2(b),R2(c),C2, R3(b),R3(c),W3(a),C3,R1(a),R1(b),W1(a),W1(b),C1

R3(b) in the original schedule reads the value of (b)

R3(b) in the serial order reads the value of W2(b)

Not view serializable

Not serializable

Not Strict: "W2(b)->W1(b)" means C2 should commit before C1, but in the schedule C1 commits before C2.

8. R1(A), R1(B), W1(A), R2(A), W1(B), R3(C), W3(A), C3, C1, W2(C), W2(B), R2(C), C2

   R1(a)->W3(a)

   R1(b)->W2(b)

   W1(a)->R2(a)

   W1(a)->W3(a)

   R2(a)->W3(a)

   W1(b)->W2(b)

   R3(c)->W2(c)

   Not Conflict-serializable,

"W1(a)->R2(a)" means C1 should come before C2, Thus only 3 possible serial orders.

R1(a),R1(b),W1(b),C1,R2(a),W2(c),W2(b),R2(c),C2,R3(c),W3(a),C3

R3(c) in the original schedule reads the value of (c)

R3(c) in the serial order reads the value of W2(c)

R1(a),R1(b),W1(b),C1, R3(c),W3(a),C3,R2(a),W2(c),W2(b),R2(c),C2

R2(a) in the original schedule reads the value of W1(a)

R2(a) in the serial order reads the value of W3(a)

R3(c),W3(a),C3, R1(a),R1(b),W1(b),C1,R2(a),W2(c),W2(b),R2(c),C2

R1(a) in the original schedule reads the value of (a)

R1(a) in the serial order reads the value of W3(a)

Not view serializable

Not serializable

Not Strict: "W1(a)->W3(a)" means C1 should commit before C3, but in the schedule C3 commits before C1.

9. R1(A), R2(A), W2(C), W2(B), R1(B), R3(B), W1(A), R2(C), C2, R3(C), W1(B), W3(A), C3, C1

   **<u>R1(a)->W3(a)</u>**

   R2(a)->W1(a)

   R2(a)->W3(a)

   W2(c)->R3(c)

   W2(b)->R1(b)

   W2(b)->R3(b)

   W2(b)->W1(b)

   **<u>R3(b)->W1(b)</u>**

   W1(a)->W3(a)

   Not Conflict serializable

“W2(c)->R3(c)” and “W2(b)->R1(b)” means C2 should commit before the start of C3 and C1. Thus only 2 possible serial orders.

R2(a),W2(c).W2(b),R2(c),C2,R1(a),R1(b),W1(a),W1(b),C1,R3(b),R3(c),W3(a),C3

R3(b) in the original schedule reads the value of W2(b)

R3(b) in the serial order reads the value of W1(b)

R2(a),W2(c).W2(b),R2(c),C2, R3(b),R3(c),W3(a),C3,R1(a),R1(b),W1(a),W1(b),C1

R1(b) in the original schedule reads the value of W2(b)

R1(b) in the serial order reads the value of R3(b)

Not view serializable

Not serializable

Not Strict: “W1(a)->W3(a)” means C1 should commit before C3, but in the schedule C3 commits before C1.

10. R2(A), W2(C), R1(A), R1(B), R3(B), W2(B), R2(C), C2, R3(C), W1(A), W1(B), W3(A), C1, C3

**<u>R2(a)->W1(a)</u>**

R2(a)->W3(a)

W2(c)->R3(c)

R1(a)->W3(a)

**<u>R1(b)->W2(b)</u>**

R3(b)->W2(b)

R3(b)->W1(b)

W2(b)->W1(b)

W1(a)->W3(a)

Not Conflict serializable.

“W2(c)->R3(c)” means C2 comes before C3. Thus only 3 possible serial orders.

R2(a),W2(c),W2(b),R2(c),C2,R1(a),R1(b),W1(a),W1(b),C1,R3(b),R3(c),W3(a),C3

R3(b) in the original schedule reads the value of (b)

R3(b) in the serial order reads the value of W1(b)

R2(a),W2(c),W2(b),R2(c),C2, R3(b),R3(c),W3(a),C3,R1(a),R1(b),W1(a),W1(b),C1

R1(a) in the original schedule reads the value of (a)

R1(a) in the serial order reads the value of W3(a)

R2(a),W2(c),W2(b),R2(c),C2, R1(a),R1(b),W1(a),W1(b),C1,R3(b),R3(c),W3(a),C3

R3(b) in the original schedule reads the value of (b)

R3(b) in the serial order reads the value of W1(b)

Not view serializable

Not serializable

Not Strict: W1(a) happens before W3(a) but T1 commits after W3(a)

HW9

Q1:the keys are CDA and CDE

A->B is redundant, thus we can decompose it to a BCNF

ABCDE

/\

```
DEB ACDE
      /\
    AE    ACD
```

Q2:the key is BE
  after decomposing to 3NF, it's BDA,AC,BD,BE
Q3:the keys are DCA and DCBE
  after decomposing to 3NF, it's ACE,AEB,BCEA,ABEF,DCA
Q4:the keys are CFD and CFE
  after decomposing to 3NF, it's FB,BDEA,ED,BCDE,CFD
Q5:the key is AEGB
  after decomposing to 3NF, it's AEFD,AC,CGF,DEF,AEGB
Q6:the key is BDGE
  after decomposing to 3NF, it's BF,DA,GC,CGA,BDGE
Q7:the key is CEFGH
  after decomposing to 3NF, it's BD,DA,ED,EGHB,ECFGH
Q8:the key is DEACH
  after decomposing to 3NF, it's AGHB,CEB,BG,ABCGF,DEACH

HW10
An intermediate table is used which results in redundancy. There are better solutions for this assignment.

create table center(id int primary key, a double, b double, c double, d double, e double, f double, g double) which stores the 10 centers.
create table point(a double, b double, c double, d double, e double, f double, g double) which stores all the points.
create table inter(a double, b double, c double, d double, e double, f double, g double, mindist double) which stores the intermediate value – the minimum distance between a point to the ten centers.
create table cluster(id int references center(id),a double, b double, c double, d double, e double, f double, g double) which stores the center number for each point

Note I didn’t include constraint when creating tables. Then we can use the load data infile command to load data in. After that we can normalize the data

```
update point
    set
        a=(a-avg(a))/stddev(a),
        b=(b-avg(b))/stddev(b),
        c=(c-avg(c))/stddev(c),
        d=(d-avg(d))/stddev(d),
        e=(e-avg(e))/stddev(e),
        f=(f-avg(f))/stddev(f),
        g=(g-avg(g))/stddev(g),
```

Then “insert into center” with 10 random points.

After this we start running the following queries until the algorithm converge.

```
replace into inter
    select p.a a, p.b b, p.c c, p.d d, p.e e, p.f f, p.g g,
                        min(pow(c.a-p.a,2) + pow(c.b-p.b,2) + pow(c.c-p.c,2) + pow(c.d-p.d,2) +
pow(c.e-p.e,2) + pow(c.f-p.f,2) + pow(c.g -p.g,2)) dist
        from center c, point p
group by p.a, p.b, p.c, p.d, p.e, p.f, p.g;

replace into cluster
    select c.id, md.a a, md.b b, md.c c, md.d d, md.e e, md.f f, md.g g
        from center c, inter
    where inter.mindist = (pow(c.a- inter.a,2) + pow(c.b- inter.b,2) + pow(c.c- inter.c,2) + pow(c.d-
inter.d,2) + pow(c.e- inter.e,2) + pow(c.f- inter.f,2) + pow(c.g - inter.g,2))
group by inter.a, inter.b, inter.c, inter.d, inter.e, inter.f, inter.g;

replace into center
  select id, avg(a), avg(b), avg(c), avg(d), avg(e), avg(f), avg(g)
      from cluster
group by id;
```