Sample Solution to Assignment #7 of CS 5200 Fall 2010.

Part A

```
create table Building (
  id int primary key,
  address varchar(750) not null unique,
  photo mediumblob
) engine=InnoDB;

create table Apartment (
  id int primary key,
  number varchar(31) not null,
  building int not null references Building(id)
    on update cascade on delete cascade,
  photo mediumblob,
  rent double,
  unique(number, building)
) engine=InnoDB;

create table Room (
  id int primary key,
  area double not null,
  type varchar(50),
  apartment int not null references Apartment(id)
    on update cascade on delete cascade,
  photo mediumblob
) engine=InnoDB;

create table Person (
  id int primary key,
  name varchar(5000) not null,
  email varchar(100) not null unique
) engine=InnoDB;

create table Owner (
  person int references Person(id)
    on update cascade on delete cascade,
  apartment int references Apartment(id)
    on update cascade on delete cascade,
  primary key(person, apartment)
) engine=InnoDB;
```

- A record requires an average of 6 bytes administrative overhead, including the space required for specifying the rid.

- An int uses 4 bytes.
- A double uses 8 bytes.
- A building address uses an average of 150 bytes.
- An apartment number uses an average of 10 bytes.
- A person name or email address uses an average of 25 bytes.
- A room type uses an average of 10 bytes.
- A photo uses an average of 32K bytes stored separately from the record. Within the record a photo uses an average of 10 bytes.
- There are 20K buildings.
- There are 2M apartments and 2M persons.
- There are 20M rooms.
- There are 20 room types.
- On average each apartment is owned by 2 persons.
- The disk block size is 8K = 8192 bytes.
- A random disk access requires 10ms.
- A page can be transferred (read or write) in 0.05 ms = 50 microseconds.

The following solution is an example of how to solve this part of the assignment. It gives data flow sizes in blocks. To obtain the sizes in bytes, multiply by 8K. Sizes can also be given in the number of records. It is only necessary to have an approximate value for the sizes.

Not all the projections are shown in the solution. It is acceptable to omit showing these in the solution as long as the data flow sizes take them into account.

Record sizes and number of blocks in each table:

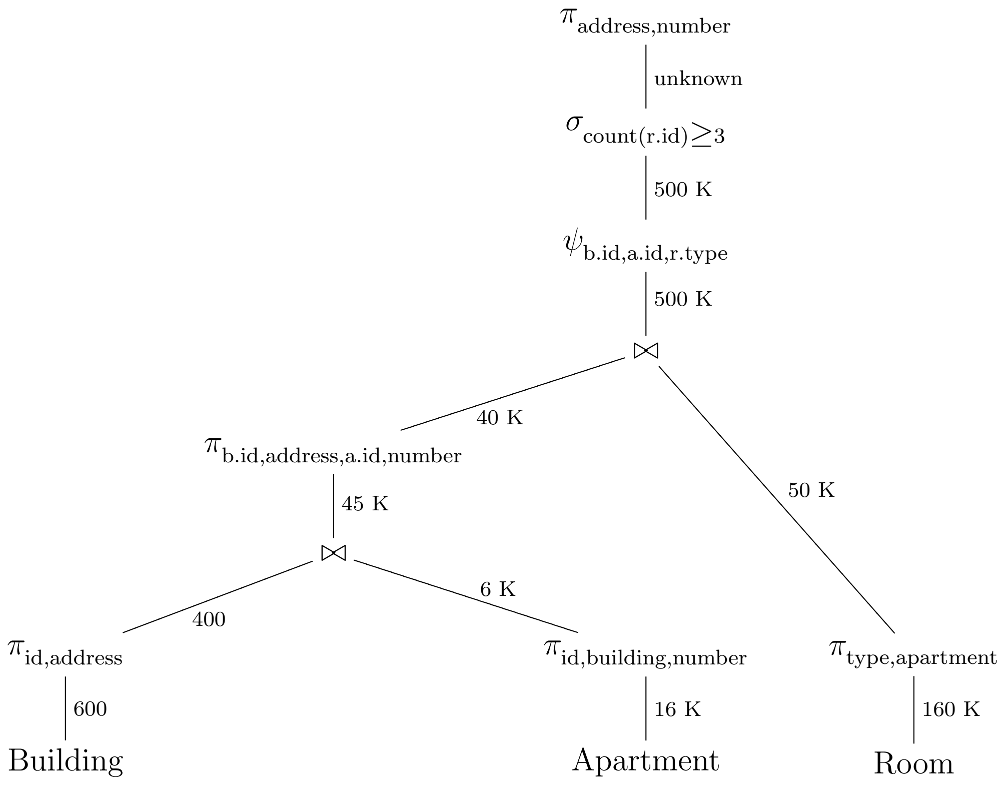
- Building size is 170 bytes. So about 400 full blocks or about 600 2/3 blocks.
- Apartment size is 42 bytes. So 10K full blocks or 15K 2/3 blocks.
- Room size is 42 bytes. So 100K full blocks or 150K 2/3 blocks.
- Person size is 60 bytes. So 15K full blocks or 23K 2/3 blocks.
- Owner size is 14 bytes. So 7K full blocks or 10K 2/3 blocks.

Note that 100K blocks is 800MB so it is not possible to keep much more than this in the main memory buffer if the entire memory is 1GB and the main memory is needed for many other purposes (such as the operating system, database system and other software).

1. Find all apartments, showing the address and apartment number that have 3 or more rooms with the same type.

   ```
   select b.address, a.number
     from Building b, Apartment a, Room r
    where b.id = a.building
      and r.apartment = a.id
    group by b.id, a.id, r.type
   having count(r.id) >= 3
   ```

   The tables must be joined in either the order Building, Apartment, Room or the order Room, Apartment, Building. The two choices are quite close, with the former being somewhat better. Projections are added to each edge to reduce the size of the data flows along each edge. The joins are unselective, so both joins use the merge-join algorithm. The second one is too large for a 1GB memory, so it requires writing to temporary files and then scanning them to perform the sorting and merging. In the following diagram, all data flow sizes are in blocks. Note that blocks in memory are full, unlike the blocks in the B-tree.

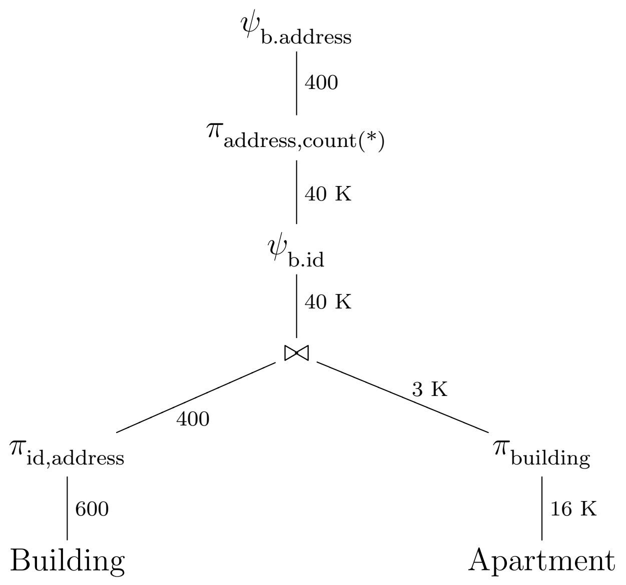


$\pi_{\text{address,number}}$
unknown
$\sigma_{\text{count(r.id)}\geq 3}$
500 K
$\psi_{\text{b.id,a.id,r.type}}$
500 K
$\bowtie$
40 K
$\pi_{\text{b.id,address,a.id,number}}$
50 K
45 K
$\bowtie$
6 K
400
$\pi_{\text{id,address}}$
$\pi_{\text{id,building,number}}$
$\pi_{\text{type,apartment}}$
600
16 K
160 K
Building
Apartment
Room

2. List the buildings in order by address together with the number of apartments in each building.

```
select b.address, count(*)
  from Building b, Apartment a
 where b.id = a.building
 group by b.id
 order by b.address
```

The join order is irrelevant since a merge-join will be performed. In this case, main memory is large enough so that no temporary files will be needed.

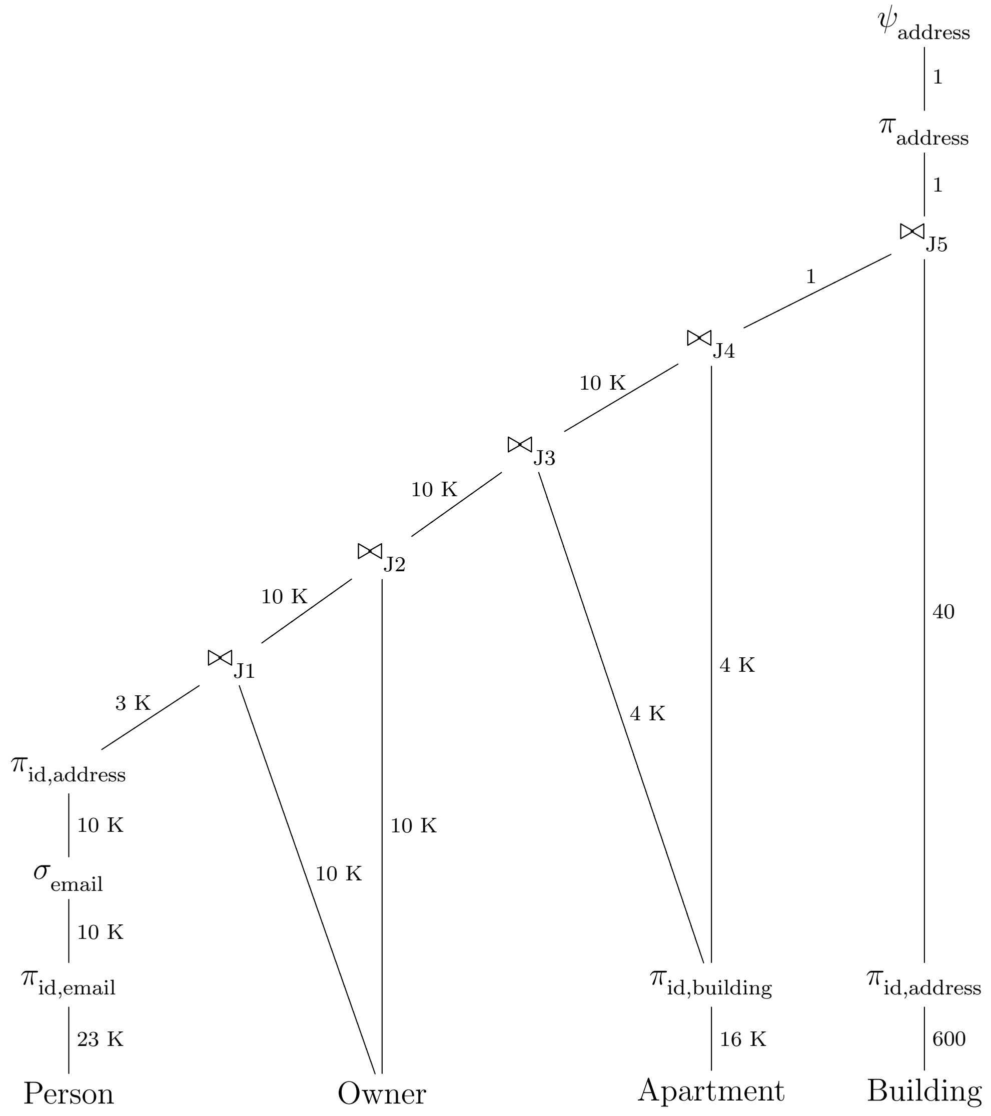


3. Find the distinct addresses of all buildings that have two different apartments owned by the same person who does not have email address fred@gmail.com.

```
select distinct b.address
  from Building b, Apartment a, Apartment c,
       Owner o, Owner q, Person p
 where b.id = a.building
   and b.id = c.building
   and a.id = o.apartment
   and c.id = q.apartment
   and o.apartment <> q.apartment
   and o.person = q.person
   and o.person = p.id
   and p.email <> 'fred@gmail.com'
```

One can either start with Person or Building. Starting with Building results in a very large data flow after the two Apartment tables are joined. This would have to be sorted in two ways. Since it is too large for the main memory, a temporary table would be required. No temporary file would be required if the first table is the Person table. So this is the optimal join order.

Since there are 20K buildings, the probability that an apartment will be in a particular building is $1/(20K)$. On average each person owns 2 apartments. The first apartment will be in one building so the chance that the second apartment is in the same building is $1/(20K)$. There are 2M persons, so the average number of persons who own two apartments in the same building is $2M/20K$ or about 100. The time required to use an index 100 times is 100 * 20ms or 2000ms. It would be faster to sequentially read the target table into memory. So all the joins should be performed using merge-join.



The join conditions are the following:

J1	p.id = o.person
J2	o.person = q.person and o.apartment <> q.apartment
J3	o.apartment = a.id
J4	q.apartment = c.id and a.building = c.building
J5	a.building = b.id

Note that two of the join conditions involve more than a foreign key constraint.

Part B

1. $R_1(X), R_2(Y), W_2(Y), W_2(X), C_2, W_3(Y), C_3, R_1(X), C_1$

   There are 4 conflicting pairs of operations:

   (a) $R_1(X)W_2(X)$
   (b) $W_2(X)R_1(X)$
   (c) $R_2(Y)W_3(Y)$
   (d) $W_2(Y)W_3(Y)$

   Therefore, the conflict graph has the edges $1 \rightarrow 2$, $2 \rightarrow 1$ and $2 \rightarrow 3$. This has a cycle, so the schedule is not conflict-serializable.

   Now consider view-serializability. The X item takes two values: the initial value and the value written by transaction 2: $W_2(X)$. Transaction 1 reads both of these values. This is incompatible with any serial schedule. So the schedule is not view-serializable.

   Finally consider strictness. Only 2 of the conflicting operations are relevant:

   (a) $W_2(X)R_1(X)$
   (b) $W_2(Y)W_3(Y)$

   To be strict we must have that C_2 precede the conflicting operations of the other two transactions. Since C_2 precedes both of the conflicting operations, the schedule is strict.

2. $R_1(X), R_2(Y), W_3(Z), W_2(Y), W_2(X), R_1(Z), W_3(Y), C_3, W_2(X), C_2, C_1$

   There are 5 conflicting pairs of operations:

   (a) $R_1(X)W_2(X)$
   (b) $R_1(X)W_2(X)$
   (c) $R_2(Y)W_3(Y)$
   (d) $W_2(Y)W_3(Y)$

(e) $W_3(Z)R_1(Z)$

Therefore, the conflict graph has the edges $1 \rightarrow 2$, $2 \rightarrow 3$ and $3 \rightarrow 1$. This is a cycle, so the schedule is not conflict-serializable.

Concerning view-serializability, the only transaction that reads X is transaction 1, and it reads the initial value of X. Since transaction 2 writes X, transaction 1 must precede transaction 2 in the equivalent serial schedule. Similarly, the only transaction that reads Y is transaction 2, and it reads the initial value of Y. Since transaction 3 writes Y, transaction 2 must precede transaction 3. Finally, the only transaction that reads Z is transaction 1, and it reads the value written by transaction 3. Therefore, transaction 3 must precede transaction 1. No serial order satisfies all of these requirements, so the schedule is not view-serializable.

Concerning strictness, there are 2 relevant pairs of conflicting operations:

(a) $W_2(Y)W_3(Y)$

(b) $W_3(Z)R_1(Z)$

These require that C_2 precede $W_3(Y)$ and C_3 precede $R_1(Z)$. Neither of these hold, so the schedule is not strict.

3. $R_1(X), R_2(Y), R_3(Z), W_2(Y), W_2(X), W_1(Z), W_3(Y), C_3, W_2(X), C_2, C_1$

There are 5 conflicting pairs of operations:

(a) $R_1(X)W_2(X)$

(b) $R_1(X)W_2(X)$

(c) $R_2(Y)W_3(Y)$

(d) $W_2(Y)W_3(Y)$

(e) $R_3(Z)W_1(Z)$

This gives the same conflict graph as in the previous schedule, so the schedule is not conflict-serializable.

Concerning view-serializability, the only transaction that reads X is transaction 1, and it reads the initial value of X. Since transaction 2 writes X, transaction 1 must precede transaction 2. Similarly, the only transaction that reads Y is transaction 2, and it reads the initial value of Y. Since transaction 3 writes Y, transaction 2 must precede transaction 3. Finally, the only transaction that reads Z is transaction 3, and it reads the initial value of Z. Since transaction 1 writes Z, transaction 3 must precede transaction 1. There is no order that satisfies all of these, so the schedule is not view-serializable.

Concerning strictness, there is only one relevant conflicting pair of operations: $W_2(Y)W_3(Y)$. This requires that C_2 precede $W_3(Y)$. This does not hold, so the schedule is not strict.