**CS 5200 Database Management Final Exam Fall 2010**

The exam is a two-hour exam and there are 60 points in all. The exam is an open-book/open-notes exam. No laptops, cell phones, PDAs or similar devices may be used during the exam. Only your own book and notes may be used during the exam: no sharing of books or notes is allowed. You must take the exam in your assigned seat. Your exam will have your name on it. You must turn in this sheet when you turn in your exam. Points will be deducted if this sheet is missing whether by accident or intentionally.

Some of the problems use the following Talent Agency Database:

create table Person(

id int,

name varchar(200) not null,

email varchar(200) not null,

primary key using hash(id),

unique using btree(name),

unique using btree(email)

);

create table Talent(

id int,

description varchar(2000) not null,

primary key using hash(id)

);

create table Performer(

id int,

stageName varchar(200),

talent int not null references Talent(id),

primary key using hash(id),

index using hash(talent),

foreign key(id) references Person(id)

);

create table Client(

id int,

company varchar(200),

primary key using hash(id),

foreign key(id) references Person(id)

);

create table Interest(

client int references Client(id),

interest int references Talent(id),

primary key tree(client, interest)

);

In this database the talent descriptions are not unique, but they are almost unique.

* A record requires an average of 8 bytes administrative overhead, including the space required for specifying the rid.
* An int uses 4 bytes.
* A double uses 8 bytes.
* A Date uses 8 bytes.
* A name (including stage name and company) or email address uses an average of 20 bytes.
* A description uses an average of 200 bytes.
* There are 5K performers and 5K clients, so there are 10K persons.
* There are 1K talents.
* Each client has an average of 10 interests.
* The disk block size is 8K = 8192 bytes.
* A random disk access requires 4ms.
* A page can be transferred (read or write) in 0.1 ms = 100 microseconds.

1. **JPA**. (10 points)

The following method finds all clients that are interested in the talent of a specific performer. Each table has a corresponding class, except that the Interest table is specified by an interests field in the Client class and by a clients field in the Talent class, both of which are of type Set. Here is the full method:

public Set<Client> getInterestedClients(EntityManagerFactory factory, String performerName) {

EntityManager em = factory.createEntityManager();

em.getTransaction().begin();

Query q = em.createQuery("select c from Client c join c.interests i, Performer p where i=p.talent and p.name=:name");

q.setParameter("name", performerName);

Set<Client> clients = new HashSet<Client>();

for (Client client : (List<Client>) q.getResultList()) {

clients.add(client);

}

em.getTransaction().commit();

em.close();

return clients;

}

Some common errors were to select c.id rather than c, i.e., the identifier of the clients rather than the client objects themselves. This is, perhaps, the most serious misunderstanding one could have about JPA compared with JDBC. JPA is an object-relational mapping, not just a relational API. JPA deals directly with objects, while SQL and JDBC are unable to deal with objets. In this case, the objects must be retrieved in order to be able to return a set of Client objects.

Another common error is to print the results. The method clearly specifies that it returns an object clients of type Set<Client>. No printing is allowed. No points were deducted for printing results even though it is wrong.

1. **Query performance**. (10 points)

How long does it take to compute the stage name of a performer given the email address of the performer using the following file structures?

* 1. Clustered heap for all tables.

A Person record requires 52 bytes and there are 10K Person records or 520K bytes in all. This requires 65 blocks. On average, half of these will have to be scanned to find an email address, so the time required is 4 + 32\*0.1 = 7ms. A Performer record requires 36 bytes and there are 5K Performer records or 180K bytes in all. This requires about 20 blocks. On average, half of these will have to be scanned to find an id, so the time required is 4 + 10\*0.1 = 5ms. The total time is 12ms.

The most common mistake was to scan all of the blocks rather than just half. In both cases, the record being retrieved is unique, so one must only scan half of the blocks.

Another mistake was to use the number of records per block rather than the number of blocks when computing how many blocks to scan.

* 1. Using the file structures specified in the schema:
     + Performer table uses a clustered hash table on its primary key.
     + Person table uses a clustered hash table on its primary key.
     + Person table has an unclustered B-tree index on the email address.

Two random accesses will obtain the Person record with the email address or 8ms. Since the Performer table is using a clustered hash index, only one random access is required to find the Performer record or 4ms. The total time is 12ms.

One common mistake was not to do this part at all. Another was to do some kind of sequential scan. Since both queries have unique results, no sequential scanning is involved.

1. **Query Optimization**. (9 points)

The following query lists the stage names and email addresses of all performers who have a specified talent.

select f.stageName, p.email

from Performer f, Person p, Talent t

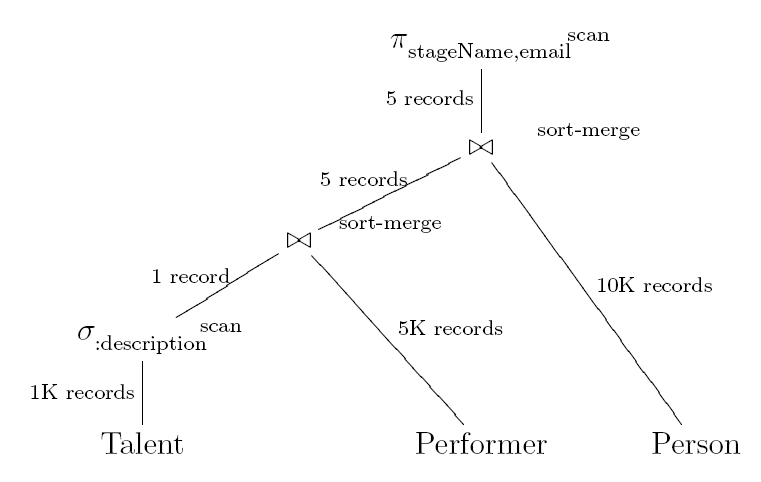
where f.id = p.id

and f.talent = t.id

and t.description = ?

Show the optimized query plan for this query. Estimate the size of the data flow (either in number of records or number of blocks) along each edge in the query evaluation plan.

Because the description attribute of Talent is not indexed, it will be necessary to scan the Talent table no matter what join order is used. If the Talent table is the first table in the join order, then the number of records in the data flow will be approximately 1, so the best join order is Talent, Performer, Person. There are 1K talents and 5K performers, so the result of the query will have about 5 performers. Although the joins for Performer and Person are selective, the tables are relatively small, so sort-merge is faster.



The most common mistake was not to show the technique for the selection and join operations. As a query evaluation plan is required, these are also required, and most students supplied them, but many omitted them entirely.

1. **Normalization**. (10 points)

The relation ABCDEF has the following functional dependencies:

* 1. F→A
  2. BF→C
  3. CF→B
  4. BE→F

Find at least one candidate key. Normalize the relation to 3NF. If possible, normalize the relation to BCNF.

* 1. Neither D
  2. Right A
  3. Left E
  4. Both BCF

The core is DE  
Closure of core is DE  
Exterior attributes are BCF  
The first choice gives a key: BDE  
It was not required, but if you continue the algorithm the other key is CDEF.

Applying 3NF gives the following:

* 1. AF unique(F)
  2. BCF unique(BF), unique(CF)
  3. BEF unique(BE)
  4. BDE unique(BDE) [or the other key]

The decomposition is BCNF.

The most common mistake was to list BCF twice because BF→C and CF→B use the same three attributes. Another common mistake was to regard one of the FDs as being a unique constraint and the other as not being a unique constraint in BCF. In fact, both FDs determine all of BCF, so both are unique constraints.

1. **Physical Database Design**. (12 points)

Suppose that your database consists of the following User table:

create table User(

id int,

name varchar(200) not null,

email varchar(200) not null,

primary key(id),

);

Assume that there are 10K User records. The most common operations and their relative frequencies are:

* 1. Add a new user. (100)
  2. Find a user (or users) by the beginning of their email address. (100)
  3. Find a user (or users) by their name. (500)
  4. Find a user given their id. (200)

What indexes should be on this table, and what kind of indexes should be used? In the case of the primary key, determine what kind of clustered index it should have. You may assume that all secondary indexes are non-clustered. Explain your choices quantitatively using a cost-benefit analysis.

The candidate indexes are:

* 1. Primary key hash table since only exact matches are ever done.
  2. Secondary btree index on email since the matches are range queries.
  3. Secondary hash index on name since the matches are exact.

The USER table requires about 100 blocks so scanning it requires 4 + 10 = 14ms to perform the second task without an index. If there is an unclustered index it would have to be a B-tree, so it would require 8ms for the same query (assuming a single result). This is an improvement of 6ms. If the result consists of multiple matches, then there is no significant improvement. This is the benefit. The cost of the index is that it must be updated each time a new user is added. This requires 4ms to obtain the leaf block, and another 4ms to update it. This is a total of 8ms. The cost is more than the benefit even if all matches have a single result, since both operations have the same frequency.

Concerning the third task, scanning again requires 14ms. A secondary hash index requires 8ms. So there is benefit of 6ms. The cost of the index is the same as in the second task, but now it is required only 1/5 as many times as the update. So the benefit is more than the cost.

The fourth task is concerned with the primary key which necessarily must have an index. Only exact match is required, so it should be a hash index.

The design should be the following:

create table User(

id int,

name varchar(200) not null,

email varchar(200) not null,

primary key using hash(id),

index using hash(name)

);

The most common mistake was to ignore the requirement that in the case of the primary key, one must determine what kind of clustered index it should have. It is also important that the analysis be quantitative, not qualitative.

Another very common misconception is that exact match queries are related to whether a column is unique. I saw statements such as "Since name is nearly unique one can use a hash index." Such a statement shows a complete misconception of what "exact match" means. One could have a hash index on a column that is far from being unique, and it can be useful to have a B-tree index on a column that is unique. An exact match is one that is specified by a where condition such as "attribute = value". A range query, by contrast, is one that is specified by a where condition such as "attribute like 'abc%'" or "attribute >= 5".

1. **Security**. (9 points)

Allow the user named 'Fred' to access information about clients who are interested in a talent whose description begins with either "dance" or "tap".

The following is the view:

create view ClientDanceTap as

select p.id, p.name, p.email, c.company

from Person p, Client c, Interest i, Talent t

where p.id = c.id

and i.client = c.id

and i.interest = t.id

and (t.description like 'dance%' or t.description like 'tap%');

Grant access as follows:

grant select on ClientDanceTap to 'Fred';