UiT
THE ARCTIC UNIVERSITY OF NORWAY

Faculty of science and technology

ASSESSMENT GUIDELINE

For exam in: **INF-2700 Database Systems**
Date: **Wednesday 22.02.2017**

The assessment guideline contains 10 pages, including this cover page

Contact person: Weihai Yu.
Phone: 41429077

Question 1 (40%)

Below are some database tables with example data for an online show application.

- People

pid	name
p01	anna
p02	jan
p03	hanna
p04	ole
p05	tom

- Shows

sid	title
s01	dance
s02	talk
s03	concert

- Performers

sid	**pid**
s01	p01
s01	p02
s02	p03
s03	p01
s03	p04

- Watchers

sid	**pid**
s01	p01
s01	p03
s01	p05
s03	p02
s03	p05

The *primary keys* of the tables are in **bold** text.

Foreign key in Performers:

- sid: references sid of Shows
- pid: references pid of People

Foreign key in Watchers:

- sid: references sid of Shows
- pid: references pid of People

Write queries to find the required information.

Queries 1–5 must be formulated in *both relational algebra and SQL*.

Queries 6–10 need only be formulated in *SQL*.

Note: In the result tables of your SQL queries, there should be *no* identical (duplicate) rows.

Relational algebra *and* SQL (1–5):

1. The titles of all shows.

   The result for the example database is:

title
dance
talk
concert

$\Pi_{title} Shows$

```
SELECT DISTINCT title
FROM   shows;
```

2. Names of all performers.

   The result for the example database is:

name
anna
jan
hanna
ole

$\Pi_{name}(shows \bowtie performers)$

```
SELECT DISTINCT name
FROM   people NATURAL JOIN performers;
```

3. Names of the watchers of the 'dance' show.

   The result for the example database is:

name
anna
hanna
tom

$\Pi_{name}(watchers \bowtie \sigma_{title='dance'} show \bowtie performers)$

```
SELECT name
FROM   watchers NATURAL JOIN shows NATURAL JOIN people
WHERE  title = 'dance';
```

4. Titles of shows that nobody watches

   The result for the example database is:

title
talk

$\Pi_{title}(\sigma_{sid \notin \Pi_{sid} watchers}(shows))$

```
SELECT title
FROM   shows
WHERE  sid NOT IN (SELECT sid FROM watchers);
```

5. Titles of shows that are watched by some of their own performers.
   Display also the the performers who watch the show.
   The result for the example database is:

title	name
dance	anna

$\Pi_{title,name}(shows \bowtie performers \bowtie watchers \bowtie people)$

```
SELECT title, name
FROM   shows NATURAL JOIN performers
             NATURAL JOIN watchers
             NATURAL JOIN people;
```

SQL *only* (6–10):

6. Number of showss
   The result for the example database is:

numberOfShows
3

```
SELECT COUNT(*) AS number_of_shows
FROM   shows;
```

7. Titles of shows and the number of their watchers, in descending order of the numbers.
   You do not have to display the shows that are not watched.
   The result for the example database is:

title	numberOfWatchers
dance	3
concert	2

```
SELECT   title, COUNT(pid) AS number_of_watchers
FROM     shows NATURAL JOIN watchers
GROUP BY sid
ORDER BY number_of_watchers DESC;
```

8. Titles and performers of solos.

   A *solo* is a show performed by a single person.

   The result for the example database is:

title	name
talk	hanna

```
SELECT title, name
FROM   shows NATURAL JOIN performers NATURAL JOIN people
GROUP BY sid
HAVING count(pid) = 1;
```

or

```
SELECT title, name
FROM   shows NATURAL JOIN performers NATURAL JOIN people
WHERE  sid NOT IN
       (SELECT p1.sid
        FROM   performers p1, performers p2
        WHERE  p1.sid = p2.sid AND p1.pid != p2.pid);
```

9. Titles of shows that have more watchers than performers.

   Display also the numbers of performers and wathers.

   The result for the example database is:

title	numberOfPerformers	numberOfWathers
dance	2	3

```
SELECT title, number_of_performers, number_of_watchers
FROM   (SELECT title, sid, COUNT(pid) AS number_of_performers
        FROM   shows NATURAL JOIN performers
        GROUP BY sid)
        NATURAL JOIN
        (SELECT sid, COUNT(pid) AS number_of_watchers
        FROM   shows NATURAL JOIN watchers
        GROUP BY sid)
WHERE   number_of_watchers > number_of_performers;
```

10. Names of people who watched all shows performed by 'anna'.

    The result for the example database is:

name
tom

```
SELECT  name
FROM    people p
WHERE   name != 'anna' AND
        NOT EXISTS
        (SELECT sid
         FROM   shows NATURAL JOIN performers NATURAL JOIN people
         WHERE  name='anna'
         EXCEPT
         SELECT sid
         FROM   watchers
         WHERE  pid = p.pid);
```

Question 2 (20%)

Now consider the physical data organization of the database in Question 1.

Suppose that our online show application gets very popular. There are millions of shows. Some shows are watched by millions of people, while some are only watched by very few people.

We consider the file organization of table `Watchers`. We focus now on the operation to find the watchers of a given show.

Should we organize the file based on hash or B$^+$-tree?

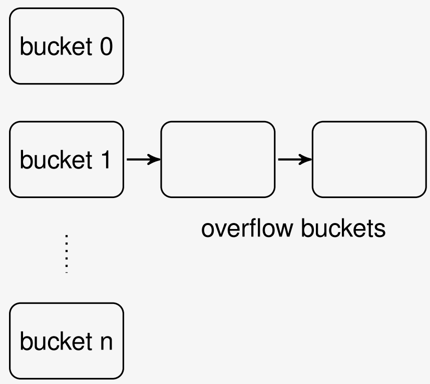
Answer the following questions.

1. What is the primary performance overhead of database systems in general?

   Disk IOs: for hard disk, number of seeks and number of block transfers.

2. Describe how table `Watchers` is organized with hash on `sid`.

   `Watchers` table file

   `Watchers` table file:

   - A *hash-organized file* consists of buckets.
   - A *bucket* is one or more disk blocks containing records.
   - *hash function* $b = h(sid)$ calculates the address b of bucket from the search key value sid.
   - All records in the same bucket have the same hash value on search key.
   - When a bucket is full, *overflow buckets* are created.

3. Describe how table `Watchers` is organized as a B$^+$-tree with `sid` as the search key.

B$^+$-tree file organization:

- balanced tree
- one block for a node
- search-key (`sid` of `Watchers`) values are ordered
- leaf nodes are linked
- leaf nodes contain `Watchers` records (this is the difference between B$^+$-tree file organization and B$^+$-tree index whose leaf nodes contain pointers to records)
- node: $\boxed{P_1 \mid K_1 \mid P_2 \mid \ldots \mid P_{m-1} \mid K_{m-1} \mid P_m}$
- m: node *fanout*
- a B$^+$-tree has a fixed n
  - non-leaf nodes: $\lceil \frac{n}{2} \rceil \leq m \leq n$
  - leaf nodes: also between half full and full (not measured with m or n, because records are larger than pointers)
  - root: 2 (if also leaf, 0) $\leq$ m $\leq n$

4. Discuss which file organization you would choose for table `Watchers`.
Focus on the performance of the operation to find the watchers of a given `sid`.
Take into account the fact that the numbers of watchers of shows can vary from very few to millions.

- For hash-organized file, the overhead is to walk through all blocks of the buckets (including overflow buckets) with the same `sid` hash value. Assume a bucket has only one block and the length of the overflow chain is H, finding a watcher takes on average $\frac{H}{2}$ seeks and blocks reads.
- For B$^+$-tree organized file, the overhead is

  (1) navigating from the root of the tree to the leave node containing the first record with the `sid` value (B seeks and blocks), and

  (2) walk through the blocks containing the records with the `sid` value ($\frac{R}{2}$ seeks and blocks).

  If there are 10,000,000 shows and a non-leave node has 100 pointers, $B = log_{100}1000000000 = 3.5$ (the height of the tree is 4).

  If a show has 10,000,000 watchers and a block has 100 $Watchers$ records, $R = \frac{10000000}{100} = 100000$. Finding a watcher takes on average $B + \frac{H}{2}$ seeks and block reads.
- For a popular show, the two mechanisms have similar performance, since $H \cong R$.
- For a show with very few watchers, the performance with hash-organized file depends on H, which may vary. The performance with B$^+$-tree organization is only dependent on the height of the tree.

So I prefer B$^+$-tree organization, but a hash-organized file is also acceptable.

Question 3 (20%)

Answer the following questions. Please explain the relevant concepts while answering the questions.

1. What is a relation schema in *Boyce-Codd Normal Form* (BCNF)?

   For schema R with a set of functional dependencies F. R is in BCNF if for any $\alpha \to \beta \in F$, one of the following is true:

   - $\beta \in \alpha$ ($\alpha \to \beta$ is trivial),
   - α is a superkey for R ($\alpha \to R$),

2. Given the relation schema $R(A, B, C), F = \{A \to B, B \to C\}$.
   Is the schema in BCNF?

   No.
   In R, A is the only candidate key.
   R is not in BCNF, because in $B \to C$, B is not a superkey.

3. Given the relation schema $R'(A, B, C), F' = \{A \to B, B \to A, B \to C\}$.
   Is the schema in BCNF?

   Yes.
   In R', A and B are two candidate keys.
   In all the given functional dependencies, the attribute on the left-hand side is a superkey.

4. What is a *lossless decomposition* of a relation schema?

   A decomposition of R into $R_1, R_2, \ldots, R_n$ is *lossless* if for every instance r,
   $r = \Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) \bowtie \ldots \bowtie \Pi_{R_n}(r)$

5. Use one of the schemas R or R' above to discuss why a lossless BCNF decomposition is useful.

   The problem with a relation schema not in BCNF is redundancy in data.
   For schema R, because B is not a superkey, there might be multiple tuples that have the same B value. For all these tuples, the same C value repeats, due to $B \to C$. This will lead to (insertion, update and deletion) anomalies.
   A lossless decomposition of R into $R_1(A, B)$, $F_1 = \{A \to B\}$ and $R_2(B, C)$, $F_2 = \{B \to C\}$ eliminates the redundancies, because R_1 and R_2 are both in BCNF. It is lossless because $R_1 \cap R_2 = B \to C$, i.e., $R_1 \cup R_2$ is a superkey of R_2.

Question 4 (20%)

1. What is an *ACID transaction*?

   A transaction is a group of operations on shared (database) data.

   Atomicity The final effect on the data is all or nothing.

   Consistency A transaction, if executed in isolation, meets its specification (dynamic consistency) and keeps the database consistent (static consistency).

   Isolation Interleaved executions of concurrent transactions have the same effect as isolated (serial) executions.

   Durability If a transaction commits, the result is not affected by possible subsequent undesirable events.

2. Describe a concurrency control mechanism based on *timestamp ordering*.

- each transaction is given a unique timestamp $TS(T_i)$ when initiated
- each database item x is associated with two timestamps and a flag

  $wt(x)$ the timestamp of the transaction that has last written x
  $rt(x)$ the timestamp of transaction that has last read x
  $c(x)$ whether the last write has committed

- Timestamp ordering algorithm
  - T_i reads x
    - $\boxed{TS(T_i) < wt(x)}$ T_i is too old, rollback T_i
    - $\boxed{TS(T_i) > wt(x)}$ wait till $c(x)$, read x, $rt(x) \leftarrow max(rt(x), TS(T_i))$
  - T_i writes x
    - $\boxed{TS(T_i) < rt(x)}$ T_i is too old, rollback T_i
    - $\boxed{TS(T_i) < wt(x)}$ skip (Thomas' write rule)
    - wait till $c(x)$, write x, $wt(x) \leftarrow TS(T_i)$, $c(x) \leftarrow false$

3. Discuss what this concurrency control mechanism achieves.

Guarantees transaction serializability (I of ACID).
Transaction executions are equivalent to sequential executions in *transaction initiation order*.

–END–