Faculty of science and technology

UiT

THE ARCTIC
UNIVERSITY
OF NORWAY

ASSESSMENT GUIDELINE

For exam in: **INF-2700 Database Systems**
Date: **Thursday 01.03.2018**

The assessment guideline contains 11 pages, including this cover page

Contact person: Weihai Yu.
Phone: 41429077

Question 1 (40%)

Below are some database tables with example data for a messaging archive application.

- `Users`

Uid	Name	Country
u1	Eva	Sweden
u2	Ole	Norway
u3	Ida	Norway
u4	Eva	Norway
u5	Ed	USA

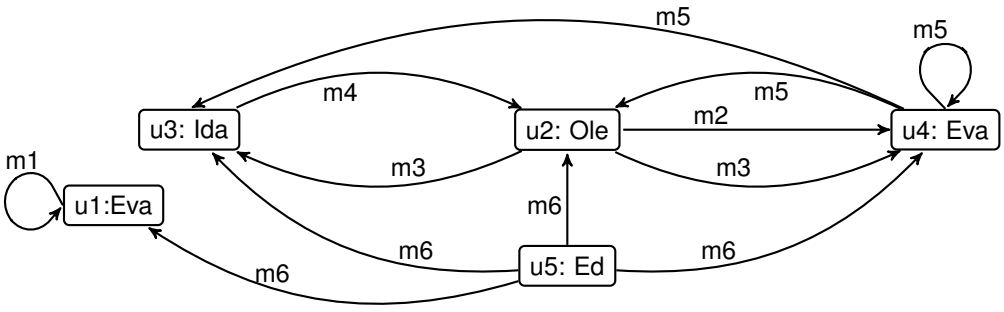
- `Messages`

Mid	Author	Content	Date
m1	u1	test new account	2018-02-28
m2	u2	Are you ready?	2018-02-28
m3	u2	Good luck!	2018-03-01
m4	u3	thanks! you, too.	2018-03-01
m5	u4	Thank you! I need it.	2018-03-01
m6	u5	You win!	2018-03-01

- `Recipients`

Mid	**Uid**	Read
m1	u1	true
m2	u4	false
m3	u3	true
m3	u4	true
m4	u2	false
m5	u2	false
m5	u3	false
m5	u4	false
m6	u1	true
m6	u2	true
m6	u3	false
m6	u4	false

The figure illustrates the association between messages and users.



In the tables, the *primary keys* of the tables are in **bold** text.

Foreign key in `Messages`:

- `Author`: references `Uid` of `Users`.

Foreign keys in `Recipients`:

- `Mid`: references `Mid` of `Messages`.
- `Uid`: references `Uid` of `Users`.

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. All countries of messaging users.
   The result for the example database is:

Country
Sweden
Norway
USA

$\Pi_{Country}(Users)$

```
SELECT DISTINCT country
FROM   users;
```

2. Content of all messages on 2018-03-01
   The result for the example database is:

Content
Good luck!
thanks! to you, too.
Thank you! I need it.
You win!

$\Pi_{Content}(\sigma_{Date='2018-03-01'}Messages)$

```
SELECT DISTINCT content
FROM   messages
WHERE  date = '2018-03-01';
```

3. Dates and content of messages with at least one recipient who is not the author himself/herself.
   The result for the example database is:

Date	Content
2018-02-28	Are you ready?
2018-03-01	Good luck!
2018-03-01	thanks! to you too.
2018-03-01	Thank you! I need it.
2018-03-01	You win!

$\Pi_{Date,Content}(\sigma_{Author \neq Uid}(Messages \bowtie Recipients))$

```
SELECT DISTINCT date, content
FROM   messages NATURAL JOIN recipients
WHERE  author != uid;
```

4. Dates and content of messages that are *only* sent to the author himself/herself.

   The result for the example database is:

Date	Content
2018-02-28	test new account

$m \leftarrow Messages \bowtie Recipients$
$m' \leftarrow \sigma_{Author \neq Uid}(Messages \bowtie Recipients)$
$\Pi_{Content,Date}(m - m')$

```
SELECT DISTINCT date, content
FROM   messages NATURAL JOIN recipients
WHERE  mid NOT IN
       (SELECT mid
        FROM messages NATURAL JOIN recipients
        WHERE author != uid);
```

5. Names and Countries of people who did not send any message on 2018-03-01.

   The result for the example database is:

Name	Country
Eva	Sweden

$\Pi_{Name,Country}(Users \bowtie (\Pi_{Uid} Users - \Pi_{Author}(\sigma_{Date='2018-03-01}Messages)))$

```
SELECT DISTINCT name, country
FROM   users
WHERE  uid NOT IN
       (SELECT author
        FROM messages
        WHERE date = '2018-03-01');
```

6. Number of different countries.

   The result for the example database is:

NumberOfCountries
3

```
SELECT COUNT(DISTINCT country) AS number_of_countries
FROM   users;
```

7. Content, author names and dates of unread messages to “Eva” from “Norway”. List in the ascending order of the dates.

   The result for the example database is:

Content	Name	Date
Are you ready?	Ole	2018-02-28
Thank you! I need it.	Eva	2018-03-01
You win!	Ed	2018-03-01

```
SELECT content, a.name, date
FROM   messages m NATURAL JOIN recipients r, users e, users a
WHERE  e.name = 'Eva' AND e.country = "Norway" AND  e.uid = r.uid
       AND r.read = false AND m.author = a.uid
ORDER BY date;
```

8. List of `Mid`s of messages and the numbers of recipients of the messages, in descending order of the numbers.

   The result for the example database is:

Mid	NumberOfRecipients
m6	4
m5	3
m3	2
m1	1
m2	1
m4	1

```
SELECT mid, COUNT(*) AS number_of_recipients
FROM   recipients
GROUP BY mid
ORDER BY number_of_recipients DESC;
```

9. Messages on 2018-03-01 with at least 3 recipients. List the `Mid`s and the numbers of recipients of the messages.

   The result for the example database is:

Mid	NumberOfRecipients
m5	3
m6	4

```
SELECT mid, COUNT(*) AS number_of_recipients
FROM   messages NATURAL JOIN recipients
WHERE  date = '2018-03-01'
GROUP BY mid
HAVING number_of_recipients >= 3;
```

10. Names and Countries of people who sent a message to everybody except himself/herself.

    The result for the example database is:

Name	Country
Ed	USA

```
SELECT  a.name, a.country
FROM    users a JOIN messages m ON author = uid
WHERE  NOT EXISTS
(SELECT uid         -- everybody
 FROM   users
 WHERE  uid != a.uid
 EXCEPT
 SELECT uid         -- recipients of m
 FROM   recipients
 WHERE  mid = m.mid);
```

Question 2 (20%)

Now consider the physical data organization for the message archiving database.

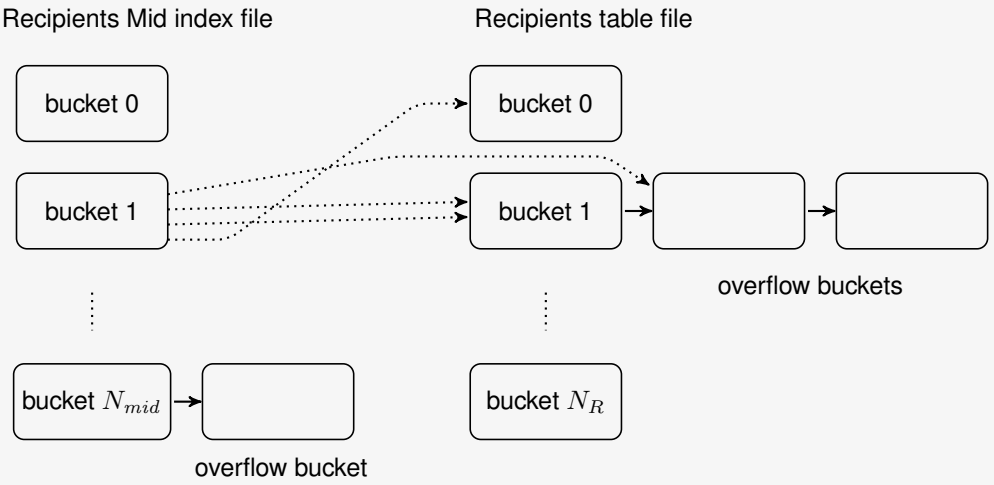
When the application becomes popular, there are a huge amount of messages archived.

We decide to organize the table Recipients with hash on `Uid`. In addition, there is a hash index on `Mid`.

Answer the following questions.

When discussing query performance, you should make reasonable assumptions of data sizes.

1. Sketch how Recipients data are organized with a figure and some brief description.



Recipients table file:

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

Recipients Mid index file:

- The organization of the index file is similar to the hash-organized file, except that
- the file contains *index entries*, and
- there is a Mid index entry for every record in the Recipients table.

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

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

3. Given the `Uid` value of a user, how to find the `Mid`s of the messages sent to the user?
What is the performance overhead of the search?

We first calculate the hash value and find the bucket of the Recipients table. Then we walk through all blocks of the bucket and all overflow buckets.
The overhead is to walk through all blocks of the buckets.
Assume a bucket has B blocks and there are N overflow buckets. The search takes $N + 1$ seeks and $(N + 1) \cdot B$ disk reads.
A user may have received many massages. It is reasonable to assume that the records with the same `Uid` value take several disk blocks. If the records of a `Uid` value takes 20% of a bucket's space, 80% of disk reads are "wasted".

4. Given the `Mid` value of a message, how to find the `Uid`s of recipients of the message?
   What is the performance overhead of the search?

   We first calculate the hash value and find the bucket of the index file. Then we walk through all blocks of the bucket and all overflow buckets to get the index entries. For each index entry, we read the block in the Recipients table file to get the record.
   The overhead is to walk through all blocks of the buckets to get the index entries and to get records using the index entries.
   Assume a bucket has b blocks and there are n overflow buckets. It takes $n+1$ seeks and $(n+1) \cdot b$ disk reads to get the index entries. Assume there are r recipients of a message, it takes r random disk reads to get the records.
   Totally, we need $(n+1)+r$ seeks and $(n+1) \cdot b + r$ disk reads.
   The overhead depends very much on r. For typical messages that have few recipients, the secondary index is quite effective. For messages like spams that have a large r, the overhead is large and there is no performance benefit to use the index.

Question 3 (20%)

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

1. What is *functional dependency* $X \to Y$ of a relation instance r?

   Y's value is determined by X's value in r. More formally, for any pair of tuples in r, if they have the same value in X, they also have the same value in Y.

   For the example instance of table `Users` in Question 1, check if the following functional dependencies are satisfied.

   a) $Name \to Country$

   No.

   b) $Country \to Name$

   No.

   c) $\{Name, Country\} \to Uid$

   Yes.

   d) $Uid \to \{Name, Country\}$

   Yes.

2. What is a *superkey* of a relation schema?

   A set of attributes K that uniquely identifies tuples in the relation. In any instance of relation schema R, no two distinct tuples have the same value on all attributes in K.

   Can you define a superkey using functional dependencies?

   $K \to R$
   or $K^+ = R$

3. Given a relation schema $R = ABCD$ with functional dependencies $\{AB \to CD, D \to B\}$.
   Is AB a super key of R? Is AC a super key of R? Is AD a super key of R?
   Why and why not?

   AB is a superkey, because $AB^+ = ABCD$.
   AC is *not* a superkey, because $AC^+ = AC \neq ABCD$.
   AD is a superkey, because $AD^+ = ABCD$.

4. What is a *candidate key* of a relation schema?

   A candidate key is a "minimal" superkey, i.e. a superkey with no proper subset as a superkey.

Can you find *all* candidate keys of the above schema R?

AB and AD, because and none of A, B or C is a superkey.

5. Why is the concept of functional dependency useful in the design of a database schema?
   Explain with the schema R.

   Functional dependencies could help us find some undesirable redundancies.
   With the above schema, due to $D \rightarrow B$ and D is not a superkey, the same B value must repeat for every occurrence of the same D value. This may lead to anomalies:

   - insertion: every insertion of a tuple with the same D value, the same B value must be inserted.
   - update: if an update of an D value does not occur at all places, the data may become inconsistent.
   - deletion: when the last tuple with a certain D value is deleted, a corresponding B value might disappear.

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 Database is kept consistent (static) and individual transactions are consistent (dynamic).

   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. What is a *log* for transaction processing?

   A log is a sequence of log records. It is used for transaction rollback and database recovery.

   How is a log organized?

   It typically has two parts: a stable part as a file on disk and a buffer part (tail) in main memory that will be flushed to disk.

   What are the operations on a log?

   A log is only updated by appending.

   append (unforced) appends a log record at the end of the log buffer.

   buffer flush the whole buffer part is flushed to disk (appended to the end of the log file).

   A forced append consists of an unforced append and a flush.
   A log is read upon rollback of transactions and recovery at DBMS restart after a system crash. It is typically read from the end and sequentially backward.

   What are the performance costs of these operations?

   An unforced append is a memory operation and the overhead can be ignored.
   A flush needs a seek and a number of disk block transfers.
   A read/scan needs a seek and a number of disk block transfers.
   If a hard drive is dedicated to the log, there is normally no need of seeks.

3. What is *write-ahead logging* (WAL)?

   Before we write a database update to disk, we first write (flush) the corresponding update log record to disk.

4. Describe an implementation of write-ahead logging.

   Each log record has a *log sequence number* (LSN). Each database page keeps the LSN of the last update.
   To write a database page to disk, we first check if the log record with the page's LSN is still in log buffer. If it is, we flush the log buffer first. Then we write the database page to disk.

5. Describe briefly how to *commit* a transaction with a single disk write.
   Explain why it works.

   An update log record (called a *redo* record) contains both the old and new values of the update.
   To commit: force-append a commit record.
   A commit log record on disk means that the transaction has committed.
   Since the new value is now in the log on disk, if the system crashes before the update is written to the table file on disk, we can redo the transaction to restore the new value.

—END—