School of Engineering and Computer Science

SWEN 304 **Database System Engineering**

Assignment 2

Due: 23:59, Friday, 15 September 2017
Venue: Cotton Building, Second Floor, SWEN 304 Hand-In Box

The objective of this assignment is to test your understanding of Relational Algebra and Query Processing and Optimization. It is worth 5.0% of your final grade. The Assignment is marked out of 100.

Since PostgreSQL does not support Relational Algebra, you will use a new DBMS called Rel to execute Relational Algebra queries. You will find an instruction how to use Rel and how to map Relational Algebra expressions to Rel in Appendix 1.

Appendix 2 contains a short recapitulation of formulae needed for cost based optimization and Appendix 3 contains an abbreviated instruction for using PostgreSQL.

Question 1. Relational Algebra [20 marks]

Consider the Musicians database schema given below.

Set of relation schemas:

Musician ({*MusicianId*, *Mus_Name*, *Telephone*, *Type*}, {*MusicianId*}),
Instrument ({*InstrumentId*, *Inst_Name*, *Musical_Key*}, {*InstrumentId*})
Played_By ({*MusicianId*, *InstrumentId*}, {*MusicianId* + *InstrumentId*})

Set of referential integrity constraints:
Playd_By[*MusicianId*] $\subseteq$ *Musician* [*MusicianId*],
Playd_By[*InstrumentId*] $\subseteq$ *Instrument* [*InstrumentId*]

The file

`MusicianDatabase.data`

on the Assignments web page contains declarations of database relational variables, referential integrity constraints and insert commands for creating and populating the musician database as a Rel database. Copy the file into your private directory. Use the Load button on the Rel – DBrowser to create and populate your Rel Musicians database.

In this question, you will be given queries on the Musicians database above in two ways. Firstly, queries are given in English and you must answer them in both Relational Algebra and the Rel query language called Tutorial D. Secondly, queries are given in Relational Algebra and you must answer them in both English and the Rel query language. Submit all your answers in printed form and, in addition, submit your Tutorial D queries as separate files electronically. Note that we will run your Tutorial D queries.

a) [12 marks] Translate the following query into Relational Algebra and Tutorial D:

1) Retrieve the names of musicians who are conductors.

2) For all singers who play instruments list their names and the instruments they play.

3) Retrieve the names of musicians who play piano or violin but not guitar.π

b) [8 marks] Translate the following two queries into English and Tutorial D:

1) $\pi_{MusicianId,\ Mus\_Name}(r(Musician)) - (\pi_{MusicianId,\ Mus\_Name}(r(Musician) * r(Played\_By)))$

2) ${}_{\text{InstrumentId, Inst\_Name}}\mathcal{F}_{(\text{COUNT}, *)}(r(Instrument) * r(Played\_By))$

Question 2. Heuristic and Cost-Based Query Optimization [50 marks]

The DDL description of a part of the University database schema is given below.

```
CREATE DOMAIN StudIdDomain AS int NOT NULL CHECK (VALUE >=
30000000 AND VALUE <= 300099999);

CREATE DOMAIN CharDomain AS char(15) NOT NULL;

CREATE DOMAIN NumDomain AS smallint NOT NULL CHECK (VALUE
BETWEEN 0 AND 10000);

CREATE TABLE Student (
StudentId StudIdDomain PRIMARY KEY,
Name CharDomain,
NoOfPts NumDomain CHECK (NoOfPts < 1000),
Tutor StudIdDomain REFERENCES Student(StudentId)
);
```

```
CREATE TABLE Course (
CourseId CharDomain PRIMARY KEY,
CourName CharDomain,
ClassRep StudIdDomain REFERENCES Student(StudentId)
);
```

```
CREATE TABLE Enrolled (
StudentId StudentIdDomain REFERENCES Student,
CourseId CharDomain REFERENCES Course,
Term NumDomain CHECK(Term BETWEEN 1997 AND 2100),
Grade CharDomain CHECK (Grade IN ('A+', 'A', 'A-', 'B+',
'B', 'B-', 'C+', 'C')),
PRIMARY KEY (StudentId, CourseId, Term)
);
```

a) [20 marks] Heuristic query optimization

Suppose we are given a query in SQL

```
SELECT StudentId, Name, NoOfPts, CourName
FROM Student NATURAL JOIN Enrolled NATURAL JOIN Course
WHERE NoOfPts > 200 AND CourseId = 'SWEN304';
```

1) **[5 marks]** Transfer the above given query into relational algebra.

2) **[5 marks]** Draw a query tree correspond to your answer for question **1)**.

3) **[10 marks]** Transfer the query tree from **2)** into an optimized query tree using the query optimization heuristics.

b) [30 marks] Query cost calculation

Assume all block sizes are 500 bytes and:

- The *Student* relation contains data about $r_s = 20000$ students (enrolled during past *10* years),
- The *Course* relation contains data about $r_c = 500$ courses,
- The *Enrolled* relation contains data about $r_e = 300000$ enrollments,
- There is a B+-tree index on (`CourseId, Term, StudentId`) of the height $h = 7$,
- A B+-tree pointer to data area is *8* bytes (`bigint`),
- All B+-tree nodes (except the root) are *50%* full,
- All data distributions are uniform (i.e. each year approximately the same number of students enroll each course),
- The intermediate results of the query evaluation are materialized,
- The final result of the query is materialized,

- The size of each intermediate or final result block should not exceed *500* bytes,
- There is a buffer pool of *4000* bytes provided for query processing in the main memory,
- **Note:** If you feel some information is missing, please make a reasonable assumption and make you assumption explicit in your answer.

1) [10 marks] Suppose the Query Processor decided to apply the nested loop join algorithm. For the given query below draw a query tree and calculate the cost of executing the query.

```
SELECT * FROM Student s, Enrolled e WHERE s.StudentId
= e.StudentId;
```

2) [20 marks] For the given query below draw a query tree and calculate the lowest cost of executing query.

```
SELECT * FROM Enrolled WHERE Term = 2014 AND CourseId =
'SWEN304';
```

Hint: *you may consider two possible algorithms, linear search and index search, and would like to find out using which algorithm leads to lowest query cost. Assume an index is implemented with a B+ tree and the lower bound for the number of entries on a tree node is 8, i.e. m = 8.*

NOTE: Some of the formulae you may need when computing the estimated query costs are given at the end of this handout. Total query cost of a query tree is the sum of the costs of all the intermediate notes and the root of a query tree.

Question 3. PostgreSQL and Query Optimization [30 marks]

You are asked here to improve efficiency of two database queries. The only condition is that after making improvements your queries produce the same results as the original ones, and your databases contain the same information as before.

For the optimization purposes, you will use two databases. A database that was dumped into the file

```
GiantCustomer.data
```

And the other database that was dumped into the file

```
Library.data
```

Both files are accessible from the course Assignments web page. Copy both files into your private directory. You are to:

i. Use PostgreSQL in order to create a database and to execute the command

```
psql –d <database_name> -f ~/<file_name>
```

This command will execute the `CREATE TABLE` and `INSERT` commands stored in the file `<file_name>`, and make a database for you.

ii. Execute the following commands:

- `VACUUM ANALYZE customer;`

on the database containing `GiantCustomer.data` file, and

- `VACUUM ANALYZE customer;`
- `VACUUM ANALYZE loaned_book;`

on database containing `Library.data` file.
These commands will initialize the catalog statistics of your database `<database_name_x>`, and allow the query optimizer to calculate costs of query execution plans.

iii. Read the PostgreSQL Manual and learn about `EXPLAIN` command, since you will need it when optimizing queries. Note that a PostgreSQL answer to `EXPLAIN <query>` command looks like:

```
NOTICE:  QUERY PLAN:

Merge Join  (cost=6.79..7.10 rows=1 width=24)
  ->  Sort  (cost=1.75..1.75 rows=23 width=12)
        ->  Seq Scan on cust_order o  (cost=0.00..1.23 rows=23
width=12)
  ->  Sort  (cost=5.04..5.04 rows=2 width=12)
        ->  Seq Scan on order_detail d  (cost=0.00..5.03 rows=2
width=12)
```

Here, PosgreSQL is informing you that it decided to apply Sort Merge Join algorithm and that this join algorithm requires Sequential Scan and Sort of both relations. The shaded number `7.10` is an estimate of the query execution cost made by PostgreSQL. When making an improved query, you will compare your achievement to this figure, and compute the relative improvement using the following formula

(original_cost – new_cost) / original_cost.

You may also want to use `EXPLAIN ANALYZE <query>` command that will give you additional information about the actual query execution time. But note, the query execution time figures are not quiet reliable. They can vary from one execution to the other, since they strongly depend on the workload imposed on the database server by users. ***To get a more reliable query time measurement, you should run your query a number of times and then calculate the average***.

a) [10 marks] Improve the cost estimate of the following query:

```
select count(*) from customer where no_borrowed = 6;
```

issued against the database containing `GiantCustomer.data`. Make such changes to your database or to the query that will allow you to produce the same result as the original query, but in a more efficient way. Of course, your changes have to be fair.

Analyze the output from the PostgreSQL query optimizer and make a plan how to improve the efficiency of the query.
Show what you have done by copying appropriate messages from the PostgreSQL prompt and explain why you have done it, calculate the improvement. Each time you want to quit with that database, please drop it, since it occupies a lot of memory space.

Marking schedule:
You will receive:

- 7 marks if your query cost estimate is at least 64% better than the original one.
- between 3 and 7 marks if your query cost estimate is between 20% and 60% better than the original one and your marks will be calculated proportionally.
- up to 3 additional marks if you give reasonable explanations of what you have done.

b) [7 marks] Improve the efficiency of the following query:

```
select * from customer where customerid = 4567;
```

issued against the database containing `GiantCustomer.data`. Make such changes to your database or to the query that will allow you to produce the same result as the original query, but in a more efficient way. Analyze the output from the PostgreSQL query optimizer and make a plan how to improve the efficiency of the query.
Show what you have done by copying appropriate messages from the PostgreSQL prompt and explain why you have done it, calculate the improvement. Each time you want to quit with that database, please drop it, since it occupies a lot of memory space.

Marking schedule:
You will receive

- 5 marks if your query cost estimate is 93% (or more) better than the original one.
- between 1 and 5 marks if your query cost estimate is better between 20% and 93% than the original one and your marks will be calculated proportionally to the improvement achieved.
- up to 2 additional marks if you give reasonable explanations of what you have done.

c) [13 marks] The following query is issued against the database containing the data from `Library.data.` It retrieves information about every customer for whom there exist less than three other customers borrowing more books than she/he did:

```
select clb.f_name, clb.l_name, noofbooks
from (select f_name, l_name, count(*) as noofbooks
     from customer natural join loaned_book
     group by f_name, l_name) as clb
     where 3 > (select count(*)
               from (select f_name, l_name, count(*) as noofbooks
                     from customer natural join loaned_book
                     group by f_name, l_name) as clb1
                     where clb.noofbooks<clb1.noofbooks)
                     order by noofbooks desc;
```

Unfortunately, the efficiency of the given query is very poor. Make such changes to your database or to the query that will allow you to produce the same result as the original query, but in a more efficient way.
Show what you have done by copying appropriate messages from the PostgreSQL prompt, calculate the improvement, and briefly explain why is the query given inefficient and why is your query better.

Marking schedule:
You will receive:

- 3 marks if you explain in English how the query computes the answer,
- 7 marks if your query has a cost estimate 95% (or more) better than the original one (otherwise, your marks will be calculated proportionally to the improvement achieved),
- Additional 3 marks if you give reasonable explanations of why the query given is inefficient and why is your query better.

Submission Instruction:

- Submit your **answers to all questions** printed on paper to the Hand-In Box.
- Additionally, submit electronically Tutorial D queries for Question 1 to the submission system.

Using Rel on the Workstations

Rel is a free open source database management system (DBMS), available at
http://sourceforge.net/project/showfiles.php?group_id=70867
It implements a database language called Tutorial D proposed by C. J. Date and Hugh Darwen. Tutorial D has a syntax that is rather similar to Relational Algebra. This way it allows learning Relational Algebra in a more efficient way, since there is no other available DBMS that directly supports Relational Algebra.

The following text presents an instruction to using Rel on SMSCS workstations and an introduction to mapping Relational Algebra expressions to Tutorial D expressions

1. How to use Rel

To run Rel, you need to issue the following commands from a Unix prompt in a shell window:

```
> need reldb
> need java2-latest
> DBrowser
```

After issuing theses commands you will get a simple window with two fields. The lower one is used to enter Tutorial D commands, whereas the upper one will contain the result of their executing. You may also enter your commands from a file using “Load” button. You run your Tutorial D command(s) with F5.

You may wish to add the “`need reldb`” and “`need java2-latest`” commands to your `.cshrc` file so that it is run automatically. Add these commands after the command `need SYSfirst`, which has to be the first `need` command in your `.cshrc` file.

2. Rel Data Definition Language Commands

Let N_1 be a relation schema name, $A_1, \ldots, A_n$ relation schema N_1 attributes and let N_1 references N_2. A relational variable `VAR` N_1 (relational database table) of the relation schema type N_1 is defined in the following way:

```
VAR N REAL RELATION
{
<A₁ data_type>,
<A₂ data_type>,
…,
<Aₙ data_type>
} KEY {Aᵢ [,…, Aₖ]};

CONSTRAINT <constraint_name> (N₁ {FK} <= N₂ {K});
```

where `REAL RELATION` means that `VAR` N_1 is not a view but a base relation, a `data_type` may be a data type from the set `{integer, char, …}`, $\{A_i, A_j, A_k\}$

is a key composed of `3` attributes, `FK` is a foreign key in N_1 and `K` is the primary key of N_2.

3. Rel Data Manipulation Commands

The following Rel command inserts a set of `m` new tuples into the relational variable `VAR N`

```
INSERT N RELATION
{
TUPLE {A₁ V₁¹, …, Aₙ Vₙ¹},
…
TUPLE {A₁ V₁¹, …, Aₙ Vₙᵐ}
};
```

where V_i^j is the value of the value of the attribute A_i in the tuple `j`.

The following command deletes a tuple from the relational variable `VAR N`

```
DELETE N WHERE (K = V);
```

where `V` is a value of the key `K`.

The following command updates an attribute `A` to the value V_a in the tuple with key value `K = V` of the variable `N`

```
UPDATE (N WHERE (K = V)) (A := Vₐ)
```

4. Mapping relational Algebra to Tutorial D of Rel

The complete syntax of Tutorial D is given in the book “Databases Types, and The Relational Data Model - The Third Manifesto” by C. J. Date and Hugh Darwen, which is available from the Vic library (QA76.9D3D232D). The following text presents an introduction to mapping Relational Algebra expressions to Tutorial D.

Let `N` be a relation schema name, $A_1, \ldots, A_n$ relation schema `N` attributes, and `r(N)` an instance of `N`. Recall that `N` also represents the name of a relational variable that contains an instance of `N` in each moment of time.

Rel returns query results in the form:

```
RELATION {Aᵢ data_typeᵢ,…, Aⱼ data_typeⱼ} {
    TUPLE {Aᵢ Vᵢ¹,…, Aⱼ Vⱼ¹},
    …
    TUPLE {Aᵢ Vᵢᵐ,…, Aⱼ Vⱼᵐ}
}
```

where $V_i^1, V_i^m \in data\_type_i$ and $V_j^1, V_j^m \in data\_type_j$.

Appendix 1

4.1 Mapping a Relational Algebra select operation

A Relational Algebra select operation

$$\sigma_{C}(r(N))$$

is mapped into the following Rel command

```
N WHERE (C)
```

where the official word `WHERE` replaces σ and `C` is a select condition.

4.2 Mapping a Relational Algebra project operation

A Relational Algebra project operation

$$\pi_{AL}(r(N))$$

is mapped into the following Rel command

```
N {AL}
```

where the expression `{AL}` replaces π and `AL`.

4.3 Mapping a Relational Algebra natural join operation

A Relational Algebra natural join operation

$$r(N_1) \star r(N_2)$$

is mapped into the following Rel command

$$N_1 \texttt{ JOIN } N_2$$

where the official word `JOIN` replaces $*$.

4.4 Mapping Relational Algebra set theoretic operations

Rel supports the following set theoretic operations: `UNION`, `INTERSECT`, and `MINUS` (`MINUS` is the set difference operator). A Relational Algebra set theoretic operation, say union of two union compatible relations N_1 and N_2

$$r(N_1) \cup r(N_2)$$

is mapped into the following Rel command

$$N_1 \texttt{ UNION } N_2$$

4.5 Mapping Relational Algebra aggregate function operations

Rel supports the following functions: `COUNT`, `SUM`, `AVG`, `MAX`, and `MIN`. A Relational Algebra aggregate function operation, say count

$$\mathcal{F}_{(COUNT,\ *)}\, r(N)$$

is mapped into the following Rel command

Appendix 1

```
SUMMARIZE (N) ADD (COUNT() AS B)
```

where `SUMMARIZE` replaces $\mathcal{F}$(script F), and official word `ADD` is used to denote

adding a new attribute `B` that is going to contain the result of the aggregate function.

Rel also supports grouping. Relational Algebra aggregate function operation with grouping, say count

$$_{(Ai,\ Aj)}\ \mathcal{F}_{(COUNT,\ *)}\,r(N)$$

where A_i and A_j are the attributes of the relation schema `N`, is mapped into the following Rel command

```
SUMMARIZE (N) PER (N {Ai, Aj}) ADD (COUNT() AS B)
```

The list of grouping attributes is given in expression `(N{Ai, Aj})`. Note, `(N {Ai, Aj})` is a projection of `N`.

Note: You need to have in mind that Rel applies relational operations like select, project, and join on relations. This may require using parentheses. For example, this is a wrong Rel expression:

```
N WHERE (condition) {AL}
```

while, the following is a correct Rel expression:

```
(N WHERE (condition)) {AL}
```

Formulae for Computing a Query Cost Estimate

Blocking factor: $f = \lfloor B / L \rfloor$

Number of blocks: $b = \lceil r / f \rceil$

Selection cardinality of the attribute A : $s(A) = r / d(A)$, where $d(A)$ is the number of different A values

Number of buffers $n = \lfloor K / B \rfloor$, where K is the size of the buffer pool

Project

If the *<attribute_list>* of a project operation contains a relation schema key, or DISTINCT is not used in the SQL SELECT command,

$$C(\text{project}) = b_1 + b_2$$

If an index is implemented on *<attribute_list>* as a B+-tree

$$C(\text{project\_distinct}) = \lceil d(Y) / m \rceil + \lceil d(Y) / f \rceil$$

with $d(Y)$ $(< r)$ as the number of different Y = *<attr_list>* values, and m the number of node entries

Select

Linear search: $C(\text{select\_linear}) = b + \lceil s(Y) / f \rceil$,

where b is the number of blocks of input relation and $s(Y)$ is the selection cardinality of the search argument Y. The complexity is **O(*r*)**

Search on primary key values with a condition $K = k$ and B+ index is used:

$$C(\text{select\_index\_B+}) = h + 2$$

Search with a B+ index being used:

$$C(\text{select\_index\_B+}) = h + \lceil s / m \rceil + s + \lceil s / f \rceil$$

where h is the height of the B+ tree, s is the selection cardinality, and m is the number of entries in a tree node.

Join algorithm

Cost of a nest loop join is

$$C = b_N + b_M * \lceil b_N / (n - 2) \rceil + \lceil r_M / f \rceil$$

Where N stands for the outer loop relation, and M stands for the inner loop relation, which contains the foreign key, and n is the number of buffers available for storing. The complexity is **O(r^2)**

Using PostgreSQL on the workstations

We have a command line interface to PostgreSQL server, so you need to run it from a Unix prompt in a shell window. To enable the various applications required, first type either

```
> need comp302tools
```

or

```
> need postgresql
```

You may wish to add either "`need comp302tools`", or the "`need postgresql`" command to your `.cshrc` file so that it is run automatically. Add this command after the command `need SYSfirst`, which has to be the first `need` command in your `.cshrc` file.

There are several commands you can type at the unix prompt:

```
> createdb ⟨database_name⟩
```

Creates an empty database. The database is stored in the same PostgreSQL server used by all the students in the class. Your database may have an arbitrary name, but we recommend to name it either `userid` or `userid_x`, where `userid` is your ECS user name and `x` is a number from 0 to 9. To ensure security, you must issue the following command as soon as you log-in into your database for the first time:

```
REVOKE CONNECT ON DATABASE <database_name> FROM PUBLIC;
```

You only need to do this once (unless you get rid of your database to start again). **Note**, your markers may check whether you have issued this command and if they find you didn't, you may be **penalized**.

```
> psql [ –d ⟨db name⟩ ]
```

Starts an interactive SQL session with PostgreSQL to create, update, and query tables in the database. The db name is optional (unless you have multiple databases)

```
> dropdb ⟨databas_name⟩
```

Gets rid of a database. (In order to start again, you will need to create a database again)

```
> pg_dump -i ⟨databas_name⟩ > ⟨file_name⟩
```

Dumps your database into a file in a form consisting of a set of SQL commands that would reconstruct the database if you loaded that file.

```
> psql –d <database_name> -f <file_name>
```

Copies the file `<file_name>` into your database `<database_name>`.

Inside and interactive SQL session, you can type SQL commands. You can type the command on multiple lines (note how the prompt changes on a continuation line). End commands with a ';'

There are also many single line PostgreSQL commands starting with '\' . No ';' is required. The most useful are

\? to list the commands,

\i 〈file_name〉
loads the commands from a file (eg, a file of your table definitions or the file of data we provide).

\dt to list your tables.

\d 〈table_name〉 to describe a table.

\q to quit the interpreter

copy <table_name> **to** <file_name>
Copy your table_name data into the file file_name.

copy <table_name> **from** <file_name>
Copy data from the file file_name into your table table_name.

Note also that the PostgreSQL interpreter has some line editing facilities, including up and down arrow to repeat previous commands.
For longer commands, it is safer (and faster) to type your commands in an editor, then paste them into the interpreter!