Exam INFO-H-417 Database System Architecture
13 January 2014

Name:

ULB Student ID:

P (60)	Q1 (20)	Q2 (20)	Q3 (20)	Q4 (60)	Q5 (20)	Tot (200)

Exam modalities

- You are allotted a maximum of **4 hours** to complete this exam.
- You are allowed to use a calculator, provided that this is a bare-bones calculator, and not a "graphical" calculator capable of being programmed and/or capable of storing text.
- Draft paper is provided by the exam supervisor.
- You should answer each question in the foreseen space after the question. Should this space prove to be insufficient you are allowed to use the back of the page as well. **Draft paper will not be corrected!**
- Be sure to motivate your answer, and give subresults where appropriate. In the absense of subresults, an incorrect final answer yields a score of 0!

Question 1. (20 pts) Consider the relations $R(A, B)$, $S(B, C, D)$ and $T(D, E, F)$. Translate the following SQL query to the relational algebra:

```
SELECT R.A, S.C FROM R, S, T
WHERE  R.B = S.B AND S.D = T.D
  AND  R.A IN
  (SELECT T2.E FROM T T2
   WHERE NOT T2.E = 4
      OR (T2.F <= ALL
            (SELECT AVG(R2.B)
             FROM R R2
             WHERE R2.A = R.A AND R2.B <> S.B)
         )
  )
```

Use the algorithm studied in the course for this purpose. Give sub-results and motivate your answer.

Step 1 [4 points] First, we put the query in the form where every subquery occurs with EXISTS and NOT EXISTS.

```
SELECT R.A, S.C FROM R, S, T
WHERE  R.B = S.B AND S.D = T.D
  AND EXISTS
  (SELECT T2.E FROM T T2
   WHERE
      R.A = T2.E
      AND (
        NOT T2.E = 4
        OR NOT EXISTS
            (SELECT AVG(R2.B)
             FROM R R2
             WHERE R2.A = R.A AND R2.B <> S.B
             HAVING T2.F > AVG(R2.B)
            )
      )
  )
```

Step 2 [2 points] Next, we rewrite the query such that every WHERE clause is in disjunctive normal form, by distributing OR over AND

```
SELECT R.A, S.C FROM R, S, T
WHERE  R.B = S.B AND S.D = T.D
  AND EXISTS
  (SELECT T2.E FROM T T2
   WHERE
       (R.A = T2.E  AND NOT T2.E = 4)
    OR (R.A = T2.E
```

```
        AND NOT EXISTS
            (SELECT AVG(R2.B)
             FROM R R2
             WHERE R2.A = R.A AND R2.B <> S.B
             HAVING T2.F > AVG(R2.B)
            )
        )
  )
```

Step 3 [1 points] We then rewrite the query such that every WHERE clause is a conjunction by translating OR into UNION

```
SELECT R.A, S.C FROM R, S, T
WHERE  R.B = S.B AND S.D = T.D
  AND EXISTS
  (
     (SELECT T2.E FROM T T2
     WHERE R.A = T2.E  AND NOT T2.E = 4)
    UNION
     (SELECT T2.E FROM T T2
     WHERE R.A = T2.E
        AND NOT EXISTS
            (SELECT AVG(R2.B)
             FROM R R2
             WHERE R2.A = R.A AND R2.B <> S.B
             HAVING T2.F > AVG(R2.B)
            )
     )
  )
```

Step 4 [4 points] We translate the most inner subquery.

```
SELECT AVG(R3.B)
FROM R R2
WHERE R2.A = R.A AND R2.B <> S.B
HAVING T2.F > AVG(R2.B)
```

This subquery has R, S, and T_2 as context relations. We first translate the FROM-WHERE clause by adding these context relations to the FROM clause:

$$e_1 = \sigma_{R_2.A=R.A \wedge R_2.B<>S.B}(\rho_{R_2}(R) \times R \times S \times \rho_{T_2}(T))$$

The HAVING clause is translated by grouping on the context relations, and introducing the aggregation attribute AVG($R_2.B$)), and a subsequent selection.

$$e_2 = \sigma_{T_2.F>AVG(R_2.B)}\ \gamma_{R.*,S.*,T_2.*,AVG(R_2.B)}\ (e_1)$$

Finally, we translate the SELECT clause, which gives us the translation of the entire query.

$$e_3 = \pi_{R.*,S.*,T_2.*,AVG(R_2.B)}(e_2)$$

Step 5 [4 points] We use the results of the previous step to translate the subquery

```
SELECT T2.E FROM T T2
WHERE R.A = T2.E
  AND NOT EXISTS
        (SELECT AVG(R2.B)
         FROM R R2
         WHERE R2.A = R.A AND R2.B <> S.B
         HAVING T2.F > AVG(R2.B)
        )
```

The context relations of the subquery-free part is R. When translating this part, however, we also need to include S, since this is a context relation occurring in a NOT exist subquery. Translation of the subquery-free part hence yields

$$\sigma_{R.A=T_2.E}(\rho_{T_2}(T) \times R \times S)$$

We decorrelate this with e_3, the result of the subquery. Since it concerns a NOT exists, we cannot simplify.

$$e_4 = \sigma_{R.A=T_2.E}\left((\rho_{T_2}(T) \times R \times S) \,\overline{\bowtie}\, \pi_{R.*,S.*,T_2.*}(e_3)\right)$$

Finally, we translate the SELECT clause, adding all parameters.

$$e_5 = \pi_{R.*,S.*,T_2.E}(e_4).$$

Step 6 [1 points] We translate the subquery

```
SELECT T2.E FROM T T2
WHERE R.A = T2.E  AND NOT T2.E = 4
```

Which is straightforward. The context relation is R.

$$e_6 = \pi_{R.*,T_2.E}\sigma_{R.A=T_2.E \wedge T_2.E<>4}(\rho_{T_2}(T) \times R)$$

Step 7 [2 points] We translate the UNION subquery

```
(SELECT T2.E FROM T T2
 WHERE R.A = T2.E  AND NOT T2.E = 4)
UNION
(SELECT T2.E FROM T T2
 WHERE R.A = T2.E
   AND NOT EXISTS
       (SELECT AVG(R2.B)
        FROM R R2
        WHERE R2.A = R.A AND R2.B <> S.B
        HAVING T2.F > AVG(R2.B)
       )
)
```

This is straightforward given that we already have the translations of both subqueries from step 5 and step 7. We need to take care, however: the schema of both translations differ. We need to resolve this by adding S as a context relation to e_6. This gives the following translation:

$$e_7 = e_5 \cup (e_6 \times S)$$

Step 8 [2 points] Finally, we translate the whole query. The subquery-free part yields the translation:

$$\sigma_{R.B=S.B \wedge S.D=T.D}(R \times S \times T)$$

We decorrelate with the EXSISTS subquery

$$\sigma_{R.B=S.B \wedge S.D=T.D}\left((R \times S \times T) \bowtie \pi_{R.*,S.*}(e_7)\right)$$

Which, since it concerns an EXISTS query can be simplified to

$$\sigma_{R.B=S.B \wedge S.D=T.D}(T \bowtie e_7)$$

Finally, we translate the SELECT clause, yielding the final result.

$$e = \pi_{R.A,S.C}\, \sigma_{R.B=S.B \wedge S.D=T.D}\,(T \bowtie e_7)$$

Question 2. (20 pts) Consider the following relational algebra expression over the relations $R(A, B)$ and $S(C, D)$:

$$\pi_{S_2.D}(\rho_{R_1}(R) \underset{R_1.A=R_2.A}{\bowtie} \sigma_{R_2.B=4}(\rho_{R_2}(R)) \underset{R_2.B=R_3.A}{\bowtie} \sigma_{R_3.B=5}(\rho_{R_3}(R)) \underset{R_3.A=R_4.A}{\bowtie} \rho_{R_4}(R) \underset{S_1.C=R_1.A}{\bowtie} \rho_{S_1}(S) \underset{S_2.c=R_2.A}{\bowtie} \rho_{S_2}(S))$$

Optimize this expression by removing redundant joins. Use the algorithm studied in the course for this purpose. Give sub-results and motivate your answer.

Step 1. [4 points] We first put this SPJ expression into conjunctive query syntax.

$$Q(d_2) : -R(a_1, b_1), R(a_1, 4), R(4, 5), R(4, b_4), S(a_1, d_1), S(a_1, d_2)$$

Step 2. [12 points] We optimize the conjunctive query above by repeatedly trying to remove atoms from the body, and checking at every step whether we retain an equivalent query. Checking equivalence is done using the "canonical method":

1. create the frozen database of the modified query, and evaluate the original query on it.
2. If the frozen head of the modified query is in the result, then the modified query is contained in the original one, otherwise not.
3. There is no need to check that the original query is contained in the modified one; that is always the case since the modified query contains a subset of the atoms of the original query.

- Try to remove $R(a_1, b_1)$. The modified query Q_2 and its frozen database D_2 are as follows.

$$Q_2(d_2) : -R(a_1, 4), R(4, 5), R(4, b_4), S(a_1, d_1), S(a_1, d_2)$$
$$D_2 = \{R(a_1, 4), R(4, 5), R(4, b_4), S(a_1, d_1), S(a_1, d_2)\}$$

  The following matching ensures that $d_2 \in Q(D_2)$:

$$a_1 \mapsto a_1$$
$$b_1 \mapsto 4$$
$$b_4 \mapsto b_4$$
$$d_1 \mapsto d_1$$
$$d_2 \mapsto d_2$$

  Therefore $Q_2 \subseteq Q$. Since always $Q \subseteq Q_2$ we hence have $Q_2 \equiv Q$.

  Q_2 is hence more optimal than Q. We continue optimizing Q_2.

- We try to remove $R(a_1, 4)$ from Q_2.The modified query Q_3 and its frozen database D_3 are as follows.

$$Q_3(d_2) : -R(4, 5), R(4, b_4), S(a_1, d_1), S(a_1, d_2)$$
$$D_3 = \{R(4, 5), R(4, b_4), S(a_1, d_1), S(a_1, d_2)\}$$

Now, $d_2 \notin Q_2(D_3)$: there is no atom in D_3 in which the constant 4 occurs in the second column of R.

We hence cannot remove $R(a_1, 4)$.

- We try to remove $R(4, 5)$ from Q_2.The modified query Q_4 and its frozen database D_4 are as follows.

$$Q_4(d_2) : -R(a_1, 4), R(4, b_4), S(a_1, d_1), S(a_1, d_2)$$
$$D_4 = \{R(a_1, 4), R(4, b_4), S(a_1, d_1), S(a_1, d_2)\}$$

Now, $d_2 \notin Q_2(D_4)$: there is no atom in D_4 in which the constant 5.

We hence cannot remove $R(4, 5)$.

- We try to remove $R(4, b_4)$ from Q_2.The modified query Q_5 and its frozen database D_5 are as follows.

$$Q_5(d_2) : -R(a_1, 4), R(4, 5), S(a_1, d_1), S(a_1, d_2)$$
$$D_5 = \{R(a_1, 4), R(4, 5), S(a_1, d_1), S(a_1, d_2)\}$$

The following matching ensures that $d_2 \in Q_2(D_5)$:

$$a_1 \mapsto a_1$$
$$b_4 \mapsto 4$$
$$d_1 \mapsto d_1$$
$$d_2 \mapsto d_2$$

Therefore $Q_5 \subseteq Q_2$. Since always $Q_2 \subseteq Q_5$ we hence have $Q_5 \equiv Q_2$. Q_5 is hence more optimal than Q_2. We continue optimizing Q_5.

- We try to remove $S(a_1, d_1)$ from Q_5.The modified query Q_6 and its frozen database D_6 are as follows.

$$Q_6(d_2) : -R(a_1, 4), R(4, 5), S(a_1, d_2)$$
$$D_6 = \{R(a_1, 4), R(4, 5), S(a_1, d_2)\}$$

The following matching ensures that $d_2 \in Q_5(D_6)$:

$$a_1 \mapsto a_1$$
$$b_4 \mapsto 4$$
$$d_1 \mapsto d_2$$
$$d_2 \mapsto d_2$$

Therefore $Q_6 \subseteq Q_5$. Since always $Q_5 \subseteq Q_6$ we hence have $Q_6 \equiv Q_5$. Q_6 is hence more optimal than Q_5. We continue optimizing Q_6.

- We cannot remove $S(a_1, d_2)$ from Q_6 since this the only atom that contains the variable d_2 that occurs in the head. If we were to remove this atom, we obtain a syntactically invalid query.

The final optimized query is hence Q_6.

Step 3. [4 points] It remains to translate this query back into relational algebra syntax, which yields:

$$\pi_{S.D}(\sigma_{R_2.B=4}(\rho_{R_2}(R)) \bowtie \sigma_{R_3.A=4 \wedge R_3.B=5}\rho_{R_3}(R) \bowtie_{R_2.A=S.C} S$$

Question 3. (20 pts) Explain and illustrate, by means of an example, how R-Trees work (what such a tree looks like, how are they constructed, how one can query for points in the index, how one can do nearest-neighbour searches, how one can insert new tuples, how one can delete tuples). Be sure to explain the strengths and the weaknesses of this index.

- R-trees are generalizations of B-trees to multiple dimension. It is a tree where every internal node corresponds to a "region". Internal nodes have, in place of keys, subregions that represent the contents of its children.
- Lookup is recursive: we start at the root. In the root, we look at all stored subregions that intersect with the region that we search. We recursively search for all such intersecting subregions. Insert and delete are done similarly, keeping the tree balanced as in a search tree.
- For nearest-neighbour queries, the search again starts at the root. In the root, we look for sub regions that intersect, expanding our search region if no such region exists, and recursively search.
- The R-tree has good support for all major multi-dimensional operations. Is adaptive to updates. With good worst-case complexity.
- Students should give an example data set + the corresponding RTree (of depth at least 2).

Question 4. (60 pts) Consider the (clustered) relations $R(A, B, C)$, $S(C, D, E)$, $T(E, F)$ and $U(E, F)$. Records from R and S comprise 50, while records from T and U comprise 40 bytes. Blocks are 4000 bytes large and there are 8 main memory buffers available. The statistics show that R contains 36000 tuples; that S contains 10000 tuples; that T contains 12510 tuples; and that U contains 25000 tuples. Furthermore, $R.B$ is uniformly distributed in the range $[0, 100]$; $S.D$ is uniformly distributed in the range $[10000, 30000]$, and S has 5000 distinct values for E. Relation R has clustered B-tree index on attribute B; relation S has unclustered hashing indexes on attributes D en E (separately). Attribute A is a key for R; C is a key for S, and E is a key for T and U.

The query compiler has already obtained the following logical query plan:

$$\pi_{R.A,\, S.C,\, T.F}\left((T - U) \bowtie \sigma_{R.B \leq 60}(R) \bowtie \sigma_{S.D != 20000}(S)\right)$$

Construct a sufficiently optimal physical query plan. Use disk I/Os as your optimization metric. Motivate your answer, and describe any assumptions that you make. It suffices to make only locally-optimal decisions (in other words: you may use the greedy algorithm, and your solution need not be globally optimal.)

Subexpression $(T - U)$

$$B(T) = \frac{12510 \times 40}{4000} = 126$$
$$B(U) = \frac{25000 \times 40}{4000} = 250$$

The difference operator is evaluated like a join (on composite attributes E and F. A one-pass algorithm is not possible. There are no indexes available on the joining attributes E and F. An index difference is hence not possible. Hence, we must resort to a multi-pass sort-based difference or hash-based difference (which will always be better than the nested difference algorithm).

1. Sort-merge based algorithm. The cost is given by:

$$\begin{aligned} &2B(T)\lceil \log_M B(T)\rceil + 2B(U)\lceil \log_M B(U)\rceil + B(U) + B(T) \\ &= 2 \cdot 126 \cdot 3 + 2 \cdot 250 \cdot 3 + 126 + 250 \\ &= 2632 \text{ I/O} \end{aligned}$$

   However, the optimization is possible since

$$2 + 4 = \left\lceil \frac{B(T)}{M^{\lceil \log_M B(T)\rceil - 1}} \right\rceil + \left\lceil \frac{B(U)}{M^{\lceil \log_M B(U)\rceil - 1}} \right\rceil \leq M = 8$$

   This optimized sort-merge difference gives a cost of

$$\begin{aligned} &2B(T)\lceil \log_M B(T)\rceil + 2B(U)\lceil \log_M B(U)\rceil - B(U) - B(T) \\ &= 2 \cdot 126 \cdot 3 + 2 \cdot 250 \cdot 3 - 126 - 250 \\ &= 1880 \text{ I/O} \end{aligned}$$

2. Hash-based algorithm. The cost is given by: (we only need to partition the smallest relation)

$$\begin{aligned}&2B(T)\left\lceil \log_{M-1} B(T)-1\right\rceil + 2B(U)\left\lceil \log_{M-1} B(T)-1\right\rceil + B(T) + B(U)\\ &= 2\cdot 126\cdot 2 + 2\cdot 250\cdot 2 + 126 + 250\\ &= 1880 \text{ I/O}\end{aligned}$$

Both options hence yield the same cost. We pick the hash-based version. (But sort-based is also possible.)

The number of tuples in the output is estimated at

$$T(T) - \frac{T(U)}{2} = 12510 - 12500 = 10.$$

Which fits in 1 block.

Subexpression $\sigma_{R.B\leq 60}(R)$

$$B(R) = \frac{36000\times 50}{4000} = 450$$

Scanning R to find the matching records hence costs $B(R) = 450$ disk I/Os.

Alternatively, we can use the clustered B-tree index on attribute B to find record with $B = 60$ and then scan to the beginning from there. Since the index is clustered, the cost of this is as many blocks as there are in the result of the selection, which we calculate using the fact that $R.B$ is uniformly distributed in the range $[0, 100]$:

$$T(\sigma_{R.B\leq 60}(R)) = \left\lceil \frac{60-0+1}{100-0+1}\times T(R)\right\rceil = \left\lceil \frac{61}{101}\times T(R)\right\rceil = 21743 \text{ records.}$$

Which fits in

$$B(\sigma_{R.B\leq 60}(R)) = \left\lceil \frac{21743\cdot 50}{4000}\right\rceil = 272 \text{ blocks.}$$

The cost of the clustered index scan is $B(\sigma_{R.B\leq 60}(R)) = 272$ I/Os, which is preferred over the 450 I/Os given by the table scan.

Subexpression $\sigma_{S.D!=20000}(S)$ Hash indexes do not help in evaluating inequality predicates. Hence, a table scan is the only option, which costs

$$B(S) = \frac{10000\cdot 50}{4000} = 125 \text{ I/O}$$

Using the fact that $S.D$ is uniformly distributed in the range $[10000, 30000]$ we estimate

$$T(\sigma_{S.D!=20000}(S)) = \left\lceil \frac{30000-10000+1-1}{30000-10000+1}\times T(S)\right\rceil = 10000 \text{ records}$$

(no penalty will be given if this is estimated to 9999 records instead).

Join ordering We consider all pairs of joins between

$$e_1 := (T - U)$$
$$e_2 := \sigma_{R.B \leq 60}(R)$$
$$e_3 := \sigma_{S.D != 20000}(S)$$

and take the pair that has the least cost to evaluate.

1. Pair e_1 and e_2. Since $B(e_1)$ fits in memory a one-pass join can be used. This costs $B(e_1) + B(e_2) = 1 + 272 = 273$ I/Os. Since there are no attributes common, an index join can not be used. All other methods cost more.

2. Pair e_1 and e_3. Since $B(e_1)$ fits in memory a one-pass join can be used. This costs $B(e_1) + B(e_3) = 1 + 125 = 126$ I/Os. Subresults do not have indexes, so an index join cannot be used. All other methods cost more.

3. Pair e_2 and e_3. A one pass join is not possible. Subresults do not have indexes, so an index join cannot be used. This only leaves the possibility of a sort-based and hash-based join. Since we need 1 buffer for the index scan on e_2, we only have $M = 7$ buffers available.

   - Sort-based.

     $$\begin{aligned} &2B(e_2)\lceil \log_M B(e_2)\rceil + 2B(e_3)\lceil \log_M B(e_3)\rceil + B(e_2) + B(e_3) \\ &= 2 \cdot 272 \cdot 3 + 2 \cdot 125 \cdot 3 + 272 + 125 \\ &= 2779 \text{ I/O} \end{aligned}$$

     The optimization is possible since

     $$6 + 3 = \left\lceil \frac{B(e_2)}{M^{\lceil \log_M B(e_2)\rceil - 1}} \right\rceil + \left\lceil \frac{B(e_3)}{M^{\lceil \log_M B(e_3))\rceil - 1}} \right\rceil \not\leq M = 7$$

   - Hash-based algorithm. The cost is given by: (we only need to partition the smallest relation).

     $$\begin{aligned} &2B(e_2)\left\lceil \log_{M-1} B(e_3) - 1\right\rceil + 2B(e_3)\left\lceil \log_{M-1} B(e_3) - 1\right\rceil + B(e_2) + B(e_3) \\ &= 2 \cdot 272 \cdot 2 + 2 \cdot 125 \cdot 2 + 272 + 125 \\ &= 1985 \text{ I/O} \end{aligned}$$

     The hash-based algorithm would hence be preferred.

We hence select to join first e_1 with e_3. The number of tuples in the output is estimated to be:

$$\frac{T(e_1) \cdot T(e_3)}{\max(V(e_1, E), V(e_3, E)} = \frac{10 \cdot 10000}{5000} = 20\text{record}$$

Which fits in one block.

The one-pass join of e_1 with e_3 requires 1 block (to hold e_1). We hence have 7 buffers remaining. This is more than enough to store $e_1 \bowtie e_3$ and hence the remaining join with e_2 can hence done using the one-pass algorithm. This costs

$$B(e_1 \bowtie e_3) + B(e_2) = 1 + 272 = 273 \text{ I/O}.$$

Projection The projection can be done at the same time as the join computation and hence does not occur extra I/O.

(Note that all subresults are pipelined.)

Question 5. (20 pts) Consider the following sequence of log records:

```
<START S>;
<S, A, 30>;
<START T>;
<T, G, 40>;
<START U>;
<COMMIT S>;
<U, B, 20>;
<T, C, 60>;
<START V>;
<U, D, 70>;
<V, F, 40>;
<COMMIT U>;
<T, E, 30>;
<COMMIT T>;
<V, B, 50>;
<COMMIT V>
```

Assume that a nonquiscent checkpoint is created right after the `<U, B, 20>` log record is written.

1. When will the corresponding `<END CKPT>` record be written if we use an undo-logging strategy? When will this corresponding record be written if we use a redo-logging strategy?

2. Describe succinctly what happens when a crash occurs right after the `<T, E, 30>` record is written and we use undo-logging. Make sure to take the checkpoint into account. Alternatively, what happens if the crash occurs right after the `<V, B, 50>` record is written?

3. Describe succinctly what happens when a crash occurs right after the `<T, E, 30>` record is written and we use redo-logging. Make sure to take the checkpoint into account. Alternatively, what happens if the crash occurs right after the `<V, B, 50>` record is written?

Response .

1. **Undo logging** At the time of start of the checkpoint, the uncommitted transactions are T and U. The corresponding end chekcpoint will be written once both of these transactions have completed (committed or aborted). At the earliest, this will hence be right after the `<COMMIT T>` record is written to the log.

   Redo logging At the time of start of the checkpoint, the uncommitted transactions are T and U. The `<END CKPT>` is written once all database elements that were modified to buffers but not yet written to disk by transactions that had already committed when the `<START CKPT>` record was written have been written to disk. From the above information, we do not know when this will be.

2. **crash after** `<T, E, 30>`. At this point, the checkpoint is not finished yet. As such, we need to undo, starting from the end of the log and moving all the way to the beginning beginning, all transactions for which a `<COMMIT>` record does not appear. In particular, we need to undo T and V.

   crash after crash after . At this point, the checkpoint is already finished. As such, we need to scan back to the last successfully completed checkpoint. In particular, we only need to undo V.

3. We first need to scan the log backwards to determine the last successfully completed checkpoint. Depending on when the modified buffers have been written to disk, this may mean that we only need to scan back to the non-quiescent checkpoint or, if this has not completed yet, to the beginning of the log.

   Then we need to redo (moving forward in the log again) all actions of transactions with a commit record on disk. So, if **crash after** `<T, E, 30>`, this can be at most S and U (if the checkpoint was not finished) or U (if the checkpoint was finished). If **crash after** `<T, E, 30>`, this can be all actions of S, U, and T (if the checkpoint was not finished) or T and U (if it was finished).