Exam INFO-H-417 Database System Architecture
11 January 2016

Name:

Student ID:
Student of ULB or VUB?
Faculty (Applied Sciences / Sciences / IT4BI):

P (60)	Q1 (20)	Q2 (10)	Q3 (15)	Q4 (40)	Q5 (55)	Tot (200)

Exam modalities

- You are allotted a maximum of **3.5 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 `JobParts(job_nbr, part_nbr)`, and `SupParts(sup_nbr, part_nbr)` of suppliers and the parts that they provide. Translate the following SQL query to the relational algebra:

```
SELECT JP1.job_nbr, SP1.sup_nbr
FROM JobParts AS JP1, SupParts AS SP1
WHERE NOT EXISTS
        (SELECT *
         FROM JobParts AS JP2
         WHERE JP2.job_nbr = JP1.job_nbr
         AND JP2.part_nbr NOT IN
             (SELECT SP2.part_nbr
              FROM SupParts AS SP2
              WHERE SP2.sup_nbr = SP1.sup_nbr)
        )
```

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

Step 1. Normalize all subqueries into a form with only EXISTS and NOT EXISTS.

```
SELECT JP1.job_nbr, SP1.sup_nbr
FROM JobParts AS JP1, SupParts AS SP1
WHERE NOT EXISTS
        (SELECT *
         FROM JobParts AS JP2
         WHERE JP2.job_nbr = JP1.job_nbr
         AND NOT EXISTS
             (SELECT SP2.part_nbr
              FROM SupParts AS SP2
              WHERE SP2.sup_nbr = SP1.sup_nbr
              AND JP2.part_nbr  =  SP2.part_nbr)
        )
```

Step 2 Normalize into CNF (for the subqueries). But this query is already in CNF.

Step 3 Translation of the innermost subquery Q_1:

```
(SELECT SP2.part_nbr
 FROM SupParts AS SP2
 WHERE SP2.sup_nbr = SP1.sup_nbr
 AND JP2.part_nbr  =  SP2.part_nbr)
```

The context relations are $SP1$, and $JP2$. The translation is hence

$$e_1 := \pi_{\texttt{SP2.part_nbr,SP1.*,JP2.*}}\sigma_{\texttt{SP2.sup_nbr = SP1.sup_nbr AND JP2.part_nbr = SP2.part_nbr}} (\rho_{\texttt{SP2}}\texttt{SupParts} \times \rho_{\texttt{SP1}}\texttt{SupParts} \times \rho_{\texttt{JP1}}\texttt{Jobparts})$$

Step 4 Translation of the subquery Q_2:

```
(SELECT *
 FROM JobParts AS JP2
 WHERE JP2.job_nbr = JP1.job_nbr
 AND NOT EXISTS
```

```
(SELECT SP2.part_nbr
 FROM SupParts AS SP2
 WHERE SP2.sup_nbr = SP1.sup_nbr
 AND JP2.part_nbr  =  SP2.part_nbr)
)
```

Here, the context relation is JP1 (for the query itself), but also SP1 since it occurs as a context relation in a NOT EXISTS subquery. The from-where part is hence

$$e_2 = \rho_{\texttt{JP2}}\texttt{JobParts} \times \rho_{\texttt{SP1}}\texttt{SupParts} \times \rho_{\texttt{JP1}}\texttt{Jobparts}$$

De-correlation is done by means of an anti-join since this is a NOT EXISTS subquery. The optimization can hence not be applied in this case. The de-correlation gives

$$e_3 = e_2 \overline{\bowtie} \pi_{\texttt{SP1.*,JP2}}(e_1)$$

It remains to translate the select and where clause, which yields:

$$e_4 = \pi_{\texttt{JP2.*,JP1.*,SP1.*}} \sigma_{\texttt{JP2.job_nbr = JP1.job_nbr}}(e_3)$$

Step 5 It now remains to translate the entire query. There are hence not context relations. Translation The from-where part of the outer query yields:

$$e_5 = \rho_{\texttt{SP1}}\texttt{SupParts} \times \rho_{\texttt{JP1}}\texttt{Jobparts}$$

De-correlation is done by means of an anti-join since this is a NOT EXISTS subquery. The optimization can hence not be applied in this case. The de-correlation gives

$$e_6 = e_5 \overline{\bowtie} \pi_{\texttt{SP1.*,JP1.*}} e_4$$

It remains to translate the select clause, which yields:

$$e_7 = \pi_{\texttt{JP1.job_nbr,SP1.sup_nbr}}(e_6)$$

In full, with merging of subsequent projections::

$$\begin{aligned}
&\pi_{\texttt{JP1.job_nbr,SP1.sup_nbr}}((\rho_{\texttt{SP1}}\texttt{SupParts} \times \rho_{\texttt{JP1}}\texttt{Jobparts}) \\
&\quad \overline{\bowtie} \pi_{\texttt{JP1.*,SP1.*}} \sigma_{\texttt{JP2.job_nbr = JP1.job_nbr}}((\rho_{\texttt{JP2}}\texttt{JobParts} \times \rho_{\texttt{SP1}}\texttt{SupParts} \times \rho_{\texttt{JP1}}\texttt{Jobparts}) \\
&\qquad \overline{\bowtie} \pi_{\texttt{SP1.*,JP2}} \sigma_{\texttt{SP2.sup_nbr = SP1.sup_nbr AND JP2.part_nbr = SP2.part_nbr}} \\
&\qquad\qquad (\rho_{\texttt{SP2}}\texttt{SupParts} \times \rho_{\texttt{SP1}}\texttt{SupParts} \times \rho_{\texttt{JP1}}\texttt{Jobparts})))
\end{aligned}$$

Question 2. (10 pts) Consider the two following queries over the relations $R(A, B)$ and $S(C, D)$. Are these queries equivalent, i.e., do they yield the same answer on every database? Explain why (not).

- $\pi_{R_1.A}\sigma_{R_2.A=S_1.C \wedge S_2.C=R_1.A \wedge S_2.D=R_2.B}\left(\rho_{R_1}(R) \bowtie_{R_1.B=R_2.A} \rho_{R_2}(R) \times \sigma_{S_1.D=\text{"ULB"}}\rho_{S_1}(S) \times \rho_{S_2}(S)\right)$
- $Q(u) \leftarrow R(u, v), R(v, u), S(u, \text{"ULB"}), S(u, u)$

These queries are not equivalent. To see why we reason as follows.

Step 1. We convert the first query into an equivalent query in the syntax of conjunctive queries.

$$Q_2(x) \leftarrow R(x, y), R(y, z), S(y, \texttt{"ULB"}), S(x, z)$$

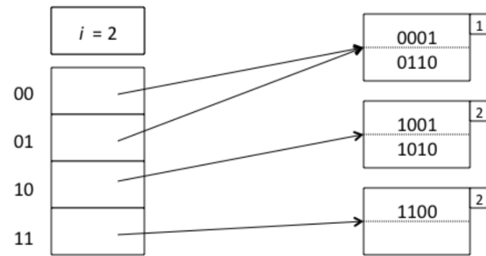
Step 2. To be equivalent, it must hold that both $Q_1 \subseteq Q_2$ and $Q_2 \subseteq Q_1$. However, $Q_2 \not\subseteq Q_1$, as shown by running Q_1 on the canonical database $D_2 = \{R(\dot{x}, \dot{y}), R(\dot{y}, \dot{z}), S(\dot{y}, \texttt{"ULB"}), S(\dot{x}, \dot{z})\}$ of Q_2. Indeed, $\dot{x} \in Q_2(D_2)$ but $\dot{x} \notin Q_1(D_2)$. To verify the latter, suppose for the purpose of contradiction that there is a matching of Q_1 in D_2 that returns $\dot{x}$. Then it has to map $u \mapsto \dot{x}$. Furthermore, in order to embed the first atom $R(u, v)$ of Q_1 into D_2, it has to map

$$u \mapsto \dot{x} \quad v \mapsto \dot{y}$$

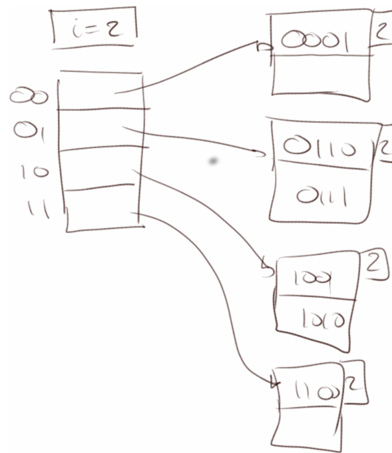
But then, the second atom $R(v, u)$ of Q_1 is mapped to $R(\dot{y}, \dot{x})$, which is not in D_2, a contradiction.

Conclusion. $\dot{x} \in Q_2(D_2)$, but $\dot{x} \notin Q_1(D_2)$. Therefore, we have established that $Q_2 \not\subseteq Q_1$, and hence Q_2 is not equivalent to Q_1.

Question 3. (15 pts) Consider the following extensible hash table with $k = 4$ (the hash function maps values to 4-bit integers) and $i = 2$.

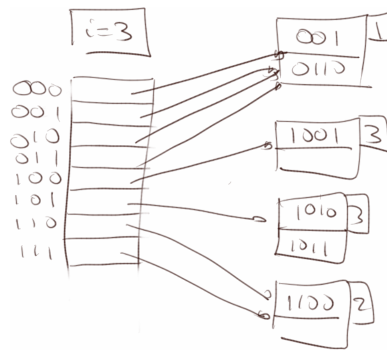


1. Insert a key with the hash value 0111 into the extensible hash table above. Document each intermediate step in the process and draw the final result.



This key is inserted into the bucket with prefix 01. The corresponding block is full, but its local depth is 1 whereas the global depth $i = 2$. We can hence just split the block, where the old block retains key 0001 (since it has prefix 00) and the new block gets keys 0110 and 0111. Both blocks have their local depth set to 2, and we re-direct the pointer of bucket 01 to the new block containing keys 0110 and 0111.

2. Insert a key with the hash value 1011 into the original extensible hash table above (not the result of (1)). Document each intermediate step in the process and draw the final result.



This key needs to be inserted into the bucket with prefix 10. The block corresponding to this bucket is full, however, and its local depth equals the global depth $i = 3$. We hence increase i to 3, causing the directory to double, with buckets for prefixes $000 - 111$. The block is split, into two blocks, one corresponding to prefix 100 (containing key 100) and the other to prefix 101 (containing keys 1010 and 1011). The directory is updated accordingly (see figure).

3. Give a sequence of two insertions that will cause the directory to double twice. If this is not possible, write NOT POSSIBLE. Start from the original extensible hash table above. The two hashed key values in your sequence should be distinct, and should be distinct from the values already shown in the hash table.

   NOT POSSIBLE with a hash function that only outputs $k = 4$ bits.

Question 4. (40 pts) State if the following statements are TRUE or FALSE. If the statement is FALSE, also briefly explain why.

1. Relational algebra expressions and conjunctive queries are two separate syntaxes for the same class of queries.

   FALSE. *Select-project-join expressions* and conjunctive queries are two separate syntaxes for the same class of queries. Relational algebra expressions are strictly more powerful.

2. A conjunctive query Q_1 is contained in a conjunctive query Q_2 if, and only if, there is a homomorphism from Q_1 to Q_2.

   FALSE. Conjunctive query Q_1 is contained in conjunctive query Q_2 if and only there is a homomorphism from Q_2 to Q_1.

3. Consider relations $R(A, B)$ and $S(B, C)$ where $T(R) = 5000$, $T(S) = 3000$, and B is a primary key on S. The expected number of tuples in $R \bowtie S$ is less than or equal to 3000.

   FALSE. The expected number is given by the formula

   $$T(R \bowtie S) = \frac{T(R) \times T(S)}{\max(V(R, B), V(S, B))}$$

   We do not know $V(R, B)$, but $V(S, B) = 3000$ (since B is a primary key for S). Therefore, the above formula yields $T(R \bowtie S) = T(R) = 5000$.

4. Consider relation $R(A)$ with $T(R) = 5000$ and A an integer attribute whose values are uniformly distributed within $[1, 1000]$. The expected number of tuples in $\sigma_{A \neq 500}(R)$ is 4995.

   TRUE.

5. For any data file, it is possible to construct two separate sparse first level indexes on different keys.

   FALSE. A sparse index only makes sense if the data file is sorted on the search key. It is hence not possible to have two separate sparse first level indexes on the same data file, for different keys.

6. For any data file, it is possible to construct two separate dense first level indexes on different keys.

   TRUE

7. For any data file, it is possible to construct a sparse first (lower) level index and a dense second (higher) level index. Both indices should be useful.

   FALSE. A dense second level index does not provide any value.

8. For any data file, it is possible to construct a dense first (lower) level index and a sparse second (higher) level index. Both indices should be useful.

   TRUE.

9. k-d tree indexes are better than hash tables based on partitioned hash functions for answering range queries on multiple key attributes.

   TRUE.

10. We have a B-tree index on attributes (A, B) of a relation R (i.e., a single B-tree index for both of these attributes together). It is possible to use this index to efficiently retrieve the records in R with $A = 10$ and $B = 50$.

    TRUE.

11. We have a B-tree index on attributes (A, B) of a relation R (i.e., a single B-tree index for both of these attributes together). It is not possible to use this index to efficiently retrieve the records in R with $A = 10$ and $B > 50$.

    FALSE. The B-tree index sorts (A, B) using the lexicographic order, i.e., $(x, y) < (x', y')$ if $x < x'$ or $x = x'$ and $y < y'$. This implies that we can first search the B-tree for the key $A = 10 \wedge B = 50$, and scan leaves to the right (in ascending order) to find those records with $A = 10 \wedge B = 50$.

12. Consider a relation $R(A, B)$ that has one clustered index on A. When retrieving records for a range query $\sigma_{A \geq 10}$, the clustered index should always be used.

    FALSE. We do not know if the index is a B-tree or a hash table. If it is a hash table, we cannot use the index to answer this query. If it is a B-tree, then yes, using the index cannot be worse than just scanning the table.

13. When it is possible, a single-pass join is always the most efficient method for joining two relations.

    FALSE. If one of the tables is very small and the other has an index on the join attribute, then an index join may be cheaper than even the one-pass join.

14. When computing the set-based union $R \cup S$ by means of a sort-merge based algorithm, we can reduce the cost by $2B(R) + 2B(S)$ I/Os, provided that $\left\lceil \frac{B(R)}{M} \right\rceil + \left\lceil \frac{B(S)}{M} \right\rceil \leq M$. Here, $B(R)$ and $B(S)$ are the size of R and S (in blocks) and M is the available memory (in blocks). $B(R)$ and $B(S)$ can be arbitrarily large.

    FALSE. We can reduce this cost only if

    $$\left\lceil \frac{B(R)}{M^{\lceil \log_M B(R) \rceil - 1}} \right\rceil + \left\lceil \frac{B(S)}{M^{\lceil \log_M B(S) \rceil - 1}} \right\rceil \leq M$$

    The formula listed hence only applies when $\log_M B(R) - 1 = 1$ and $\log_M B(S) - 1 = 1$.

15. Consider that we want to compute the set-based union $R \cup S$ of relations R and S with $B(R) = 1000$, $B(S) = 200$, and $M = 32$ buffers available. In this case, the sort-merge algorithm and the hash-based algorithm have the same cost.

    FALSE. The sort-merge join cannot be optimized and costs

    $$2B(R) \lceil \log_M B(R) \rceil + 2B(S) \lceil \log_M B(S) \rceil + B(R) + B(S) = 6000$$

    while the has join costs

    $$2B(R) \lceil \log_{M-1} B(S) - 1 \rceil + 2B(S) \lceil \log_{M-1} B(S) - 1 \rceil + B(R) + B(S) = 3600$$

16. Any lock-based schedule is conflict serializable.

    FALSE. Only if the lock-based schedule is two-phase locked.

17. Undo logging is suitable for use with archiving, although it is less efficient than redo or undo/redo logging.

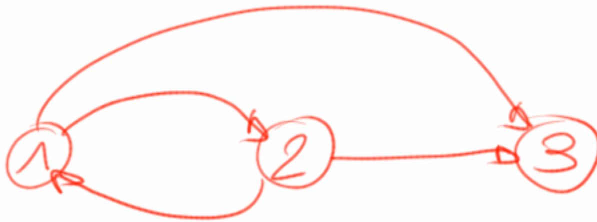
    FALSE. Undo logging is not suitable for archiving since we cannot use it to redo the part of the log that was captured after the backup was made (in order to forward the backup to our operational database).

18. Consider an undo/redo log that contains a successfully completed non-quiescent checkpoint. A crash occurs just after the checkpoint finishes and the log records were flushed on disk. To recover from this error, in no case do we need to scan further back in the log than the start of this last successfully completed non-quiescent checkpoint.

    FALSE. We may need to scan further back to undo those transactions that were active when the checkpoint start, yet did not commit yet at the time that the crash occurred.

19. The schedule $r_1(A), w_2(A), r_2(B), w_1(B), w_3(A), w_3(B)$ is conflict-serializable.

    FALSE Here is the dependency graphs, which contains a cycle.

20. Consider the sequences of events

$$R_1(A, B); R_2(B, C); V_1; R_3(C, D); V_3; W_1(A); V_2; W_2(A); W_3(B);$$

When processed by a validation-based scheduler, all validation attempts succeed (and hence, no transaction is restarted).

In the above schedule, we use $R_i(X)$ to mean "transaction T_i starts, and its read set is the list of database elements X". Also, V_i means "T_i attempts to validate," and $W_i(X)$ means that "T_i finishes, and its write set was X."

FALSE. When V_2 tries to validate, the scheduler will check that $\mathrm{RS}(T_2) \cap \mathrm{WS}(T_3) = \emptyset$, which is not the case. Hence, T_2 is aborted and restarted.

Question 5. (55 pts) The following relations store emails, and link each email with keywords that occur in the email. Primary keys are underlined.

- Email(eid integer, from char(60), to char(60), sentdate date, subject char(100), body char(1800))
- KeyWordOccurrence(kid integer, eid integer, position integer)
- KeyWord(kid integer, keyword char (100))

Integers and dates comprise 4 bytes. Each of these relations has a clustered B-tree on the primary key. In addition, there are unclustered B-tree indexes on `Email.from` and `Email.to`. Blocks are 4096 bytes large and there are 78 main memory buffers available. The Email relation has 4096 records; the KeyWordOccurrence relation has 20480 records and the KeyWord relation has 256 records. There are 20 unique senders and 40 unique receivers in the `Email` table. No keyword is repeated in the `Keyword` table.

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

$$\pi_{\texttt{kid,from}}\left(\sigma_{\texttt{to="John"}}(\texttt{Email}) \bowtie \texttt{KeywordOccurrence} \bowtie \sigma_{\texttt{keyword="databases"}}(\texttt{Keyword})\right)$$

Construct a sufficiently optimal physical query plan. Use disk I/Os as your optimization metric. Motivate your answer, and describe 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.)

Observations:

- Email tuples are 2028 bytes large. We can hence fit 2 tuples in a block. Hence
$$B(E) = \left\lceil \frac{4096}{2} \right\rceil = 2048.$$
- KeyWordOccurrence tuples are 12 bytes large. We can hence fit 341 tuples in a block. Hence
$$B(KWO) = \left\lceil \frac{20480}{341} \right\rceil = 61.$$
- Keyword tuples are 104 bytes large. We can hence fit 40 tuples per block Hence
$$B(KW) = \left\lceil \frac{256}{40} \right\rceil = 7.$$

Step 1. We start bottom-up and first determine a plan for $e_1 = \sigma_{\texttt{to="John"}}(\texttt{Email})$. A full table scan followed by a filter costs $B(E) = 2048$ blocks. Alternatively, we can use the unclustered B-tree index on `Email.to`, which costs $T(e_1)$ block I/O,s where

$$T(e_1) = \left\lceil \frac{1}{40} \times T(E) \right\rceil = \left\lceil \frac{1}{40} \times 4096 \right\rceil = 103.$$

We hence prefer to use the unclustered index, since it is cheaper.

There are $T(e_1) = 103$ tuples in the output of e_1, which fits in 52 blocks.

Step 2. To evaluate $e_2 = \sigma_{\texttt{keyword="databases"}}(\texttt{Keyword})$ we can only use a table scan (since there is no index on `keyword`. This costs $B(KW) = 7$ I/Os. The estimated number of tuples resulting from e_2 is

$$T(e_2) = \left\lceil \frac{1}{256} \times T(KW) \right\rceil = 1$$

This fits in a single block, i.e., $B(e_2) = 1$.

Step 3. Now, we have to fix a join order. We consider all pairs, and pick the one with the least cost.

Pair $e_1 \bowtie KWO$. We need 1 buffer to compute e_1. Hence we have $M' := M - 1 = 77$ main memory buffers left. Both e_1 and KWO fit in main memory and hence a one-pass join is possible, which costs $B(e_1) + B(KWO) = 52 + 61 = 113$ I/Os. An index join is not possible, because KWO has an index on the composite key (kid, eid), which we cannot use to answer point queries on eid efficiently if the index stores keys in (kid,eid) lexicographic order. Furthermore, e_1 is a subresult and hence does not have indexes. All other join methods always have a higher cost. Hence, the one-pass join is to be preferred here, which costs 113.

Pair $KWO \bowtie e_2$. We need 1 buffer to compute e_2. Hence we have $M' := M - 1 = 77$ main memory buffers left. Both KWO and e_2 fit in main memory, and hence a one-pass join is possible, which costs $B(KWO) + B(e_2) = 61 + 1 = 62$ I/Os. An index join using KWO as the outer relation is not possible, since e_2 is a subresult and does not have any indexes. An index join using e_2 as the outer relation is possible because KWO has an index on the composite key (kid, eid). If we assume that this BTree uses the lexicographic order to sort its search keys, we can use it to answer point queries on kid efficiently. Since the BTree is clustered, this costs:

$$B(e_2) + T(e_2) \times \frac{B(KWO)}{V(KWO, kid)}$$

Assuming that ever keyword in KW occurs in KWO, $V(KWO, kid) = 256$. Therefore, the index join costs

$$B(e_2) + T(e_2) \times \frac{B(KWO)}{V(KWO, kid)} = 1 + 1 \times \left\lceil \frac{61}{256} \right\rceil = 2$$

(Note: this assumes a uniform distribution of keywords in the KWO table.)

All other join methods always have a higher cost. Hence, the index join is to be preferred here, which costs 2.

Pair $e_1 \bowtie e_2$. This is a cartesian product. Both fit in main memory and a one-pass join is possible, which costs $B(e_1) + B(e_2) = 52 + 1 = 53$ I/Os. An index join is not possible since e_1 and e_2 are subresults and do not have indexes. All other join methods always have a higher cost.

Conclusion. The pair with the least expected cost is $e_3 = KWO \bowtie e_2$. The number of expected tuples in the output is

$$T(e_3) = \frac{T(KWO) \times T(e_2)}{\max(V(KWO, kid), V(e_2, kid))} = \frac{20480 * 1}{256} = 80.$$

We expect a tuple of e_3 to be of size $12 + 104 = 114$ (the sum of the sizes of KWO and KW records). We can fit $\left\lfloor \frac{4096}{114} \right\rfloor = 35$ of such tuples in a block. Hence, $B(e_3) = \left\lceil \frac{T(e_3)}{35} \right\rceil = 3$ blocks.

Step 4. It remains to join e_3 with E. We need 2 buffers to compute the index join of e_3. This hence leaves $M' := M - 2 = 76$ buffers remaining. Since e_3 fits in memory, we can then use a one-pass join to compute $e_4 = e_3 \bowtie E$. This costs $B(e_3) + B(E) = 3 + 61 = 64$ I/Os. Alternatively, we can use e_3 as an outer relation and use the index on E's primary key to compute e_4 by means of an index join. Since eid is a primary key of KWO, there can be only one matching tuple per lookup. The cost of the index join hence is

$$B(e_3) + T(e_3) \times 1 = 3 + 80 \times 1 = 81$$

which is more expensive than the one-pass join. All other join methods cost more. Hence, the one-pass join is to be preferred.

Step 5. Finally the project can be computed on the fly when we compute the final join and does not incur any cost.