Data Management (A.A. 2024/25) – exam A of 05/06/2025
Solutions

Problem 1 Consider a scheduler D that behaves as follows when processing an input schedule S: D lets S proceed, dynamically building the precedence graph $P(S)$ by adding nodes and edges when needed and never deleting nodes or edges, and acting only whenever it processes the commit action of a transaction T_i. When processing such action, it executes the commit action if T_i is not involved in any cycle in $P(S)$, otherwise it aborts and rollbacks T_i. Let S be any complete schedule on transactions $T_1, \ldots, T_n$, where the last action of each T_i is the commit action c_i, let S' be the schedule produced in output by D when processing S, and let S'' be the schedule obtained from S' by ignoring the actions of the transactions aborted by D.

1.1 Prove or disprove that S is conflict serializable if and only if $S = S''$.

1.2 Prove or disprove that S'' is recoverable, and in case you disproved that S'' is recoverable, tell how you would modify D in order to ensure recoverability of S''.

Solution 1

1.1 Proving the statement "if S is conflict serializable, then $S = S''$" is easy. If S is conflict serializable, then $P(S)$ is acyclic and therefore when D processes the input schedule S and encounters the commit action of a transaction T_i, it will never abort and rollback T_i. It follows that the output of D will coincide with its input, i.e., $S = S''$.

We now prove that if S is not conflict serializable, then $S \neq S''$. If S is not conflict serializable, then $P(S)$ has at least one cycle γ. Let T_i be a transaction involved in the cycle γ. Since the last action of every transaction in S is the commit action, when D processes c_i, it aborts and rollbacks T_i, and therefore the actions of T_i are not part of S''. We conclude that $S \neq S''$.

1.2 We disprove that S'' is recoverable by simply considering the non-recoverable schedule S

$$w_1(x)\ r_2(x)\ c_2\ c_1$$

and noticing that, obviously, in this case $S = S''$, which implies that S'' is not recoverable. To ensure that S is recoverable, the scheduler D could be modified as follows: when processing the commit action c_i of transaction T_i, D executes such commit action if and only if T_i is not involved in any cycle in $P(S)$ and all the transactions from which T_i has read have already committed, otherwise it aborts and rollbacks T_i.

Problem 2 Consider the following schedule S:

$$r_1(x)\ w_2(x)\ w_2(y)\ r_3(x)\ w_4(x)\ w_2(z)\ w_3(y)$$

2.1 Is S a 2PL schedule with both shared and exclusive locks? Motivate your answers in detail.

2.2 Is S a strict schedule? Motivate your answers in detail.

2.3 Describe the behavior of the timestamp-based scheduler when processing S, assuming that, initially, for each element α of the database, we have rts(α)=wts(α)=wts-c(α)=0, and cb(α)=`true`, and assuming that the subscript of each action denotes the timestamp of the transaction executing such action.

Solution 2

2.1 The following lock-extended schedule shows that S is a 2PL schedule with both shared and exclusive locks:

$$sl_1(x)\ r_1(x)\ u_1(x)\ xl_2(x)\ w_2(x)\ xl_2(z)\ u_2(x)\ sl_3(x)\ r_3(x)$$

$$u_2(y)\ xl_3(y)\ u_3(x)\ xl_4(x)\ w_4(x)\ u_4(x)\ w_2(z)\ u_2(z)\ w_3(y)\ u_3(y)$$

2.2 The schedule S is not strict, because transaction T_3 reads x from T_2 before the commit of T_2.

2.3 Here is the behavior of the timestamp-based scheduler when processing S (assuming that the commit operations come after all the actions of S and follow the order of timestamp):

$r_1(x)$: read ok, $rts(x) = 1$

$w_2(x)$: write ok, $wts(x) = 2, cb(x) = \texttt{false}$

$w_2(y)$: write ok, $wts(y) = 2, cb(y) = \texttt{false}$

$r_3(x)$: read ok, but T_3 put in a waiting queue because $cb(x) = \texttt{false}$

$w_4(x)$: write ok, but T_4 put in a waiting queue because $cb(x) = \texttt{false}$

$w_2(z)$: write ok, $wts(z) = 2, cb(z) = \texttt{false}$

$w_3(y)$: not executed, since T_3 is suspended

c_1: ok

c_2: ok, $cb(x) = cb(y) = cb(z) = \texttt{true}$

$r_3(x)$: read ok, $rts(x) = 3$

$w_4(x)$: write ok, $wts(x) = 4, cb(x) = \texttt{false}$

$w_3(y)$: write ok, $wts(y) = 3, cb(y) = \texttt{false}$

c_3: ok, $cb(y) = \texttt{true}$

c_4: ok, $cb(x) = \texttt{true}$

Problem 3 Consider the relations `S(A,B)` and `R(B,C,D,E)`, where (*i*) both have `B` as key, (*ii*) `S` is stored in a heap with 60 pages (each page with 40 tuples) with an associated hash index whose search key is `B`, (*iii*) `R` is stored in a heap with 900 pages (each page with 20 tuples) and (*iv*) the buffer has 32 frames available. If your goal is to compute the natural join (equi-join on `B`) between `R` and `S` as efficiently as possible in terms of number of page accesses, which algorithm would you choose among:

3.1 block-nested loop,
3.2 multi-pass based on sorting,
3.3 index-based.

Explain your answer in detail so as to convince that you choice is the right one.

Solution 3
We simply compute the cost of each of the three algorithms, and then choose the most efficient one.

3.1 Block-nested loop. The smaller relation is `S`. So, the cost is $B(\texttt{S}) + B(\texttt{R}) \times \lceil B(\texttt{S})/(32-2) \rceil = 1.860$.
3.2 Multi-pass based on sorting. We must determine the required number of passes, which is obviously greater than 1. Since $32 \times 31 = 992$ and $992 > 900 + 60$, two passes suffice. Notice that the problem of too large fragments does not occur, because the join is on the keys of the relations. Thus, the cost is $3 \times (B(\texttt{S}) + B(\texttt{R})) = 2.880$.
3.3 Index-based. Since the number $T(\texttt{R})$ of tuples of `R` is $900 \times 20 = 18.000$ and since we assume 1 to be the cost of searching for a value of `B` in `R` using the hash index, the cost is $B(\texttt{R}) + T(\texttt{R}) \times 1 = 900 + 18.000 \times 1 = 18.900$.

We conclude that we should choose the block-nested loop algorithm.

Problem 4 Consider the relation `MEETING(code,topic,venue,city,date)`, with 800.000 tuples stored in a sorted file with search key `code` (which is also the key of the relation), and with an associated sorted index with search key `venue`. We know that no more than 300 meetings are held in the same venue, that every attribute and pointer in our system occupies 10 Bytes, and that the size of each page in our system is 1.000 Bytes. Consider the following operations (1) given a venue, compute the code of all meetings held in that venue, together with the corresponding topic; (2) insert a new meeting. For each of the two operations, tell which is the worst-case cost of its execution in terms of number of page accesses.

Solution 4

4.1 To compute, given a venue, the code of all meetings held in that venue, together with the corresponding topic, we obviously use the index to find the first appropriate data entry and all othe relevant data entries, for each of them following the pointer to the data file to retrieve the corresponding value of the attribute `topic`. In order to evaluate the cost, we need to compute the number of pages in the sorted index file. Since the data file is sorted on `code` and the search key of the sorted index is `venue`, the index is clearly unclustering and therefore dense. This means that the index has as many data entries as the number of tuples in the data file. Now, we have to compute the number of data entries that fit in one page. Since every attribute and pointer in our system occupies 10 Bytes, each data entry occupies 20 Bytes and since the size of each page is 1.000 Bytes, we conclude that we have space for $1.000/20 = 50$ data entries in each page and that we have to store $800.000/50 = 16.000$ pages in the index file. Also, since we have to retrieve 300 data entries (in the worst case), we know that we need to access $300/50 = 6$ pages of the index, besides the first one retrieved with binary search, and one page of the data file for each data entry, in the worst case (i.e., 300 pages of the data file).

The cost of the first operation is therefore ($log_2 16.000 + 1$ is the cost of locating the first appropriate data entry by means of binary search): $log_2 16.000 + 1 + 6 + 300 = 14 + 1 + 6 + 300 = 321$.

4.2 For the second operation, we evaluate the cost under the assumption that we do not use the overflow pages, rather, we keep the files sortted by compacting the pages. When we insert a new meeting, we insert both a new data entry in the index, and a new tuple in the data file, but we have to do so by to keeping both the index file and the data file sorted. The worst case for both insertion is the one where we have to move all the records and we need to allocate a new page. So, for the insertion into the index file the cost is 16.000 + 1. As for the data file, since every value of every attribute occupies 10 Bytes and we have 5 attributes in every tuple of `MEETING`, we know that every tuple occupies 50 Bytes and therefore each page (whose size is 1.000 Bytes) holds $1.000/50 = 20$ tuples. It follows that the data file is stored in $800.000/20 = 40.000$ pages. So, for the insertion into the data file the cost is 40.000 + 1.

Problem 5 (only for students who opted for **option 1**, i.e., who do **not** do the project)
Let B be a relational database with relations `Student(id,age)`, `Exam(stid,ccode,mark)`, `Course(ccode,credits)`, `Teaches(pcode,ccode)`, `Prof(pcode,age)`, `Tutoring(pcode,stid,year)`, where (i) each exam is given by a student for a given course and with a given mark, (ii) each professor can teach several courses and each course can be taught by several professors, (ii) each professor can be the tutor of several students and each student can be tutored by several professors.

5.1 Describe how you would organize a property graph database G in order to represent the relational database B. In particular, (i) specify how nodes, edges, labels, etc. of G are used in order to capture the information stored in the tables of B and (ii) choose a few tuples for the relations in B, and show the specific property graph database G obtained by applying the chosen representation method.

5.2 Describe how you would organize a document database D in order to represent the relational database B. In particular, (i) specify how collections, documents, etc. of D are used in order to capture the information stored in the tables of B and (ii) choose a few tuples for the relations in B, and show the specific D obtained by applying the chosen representation method.

Solution 5

5.1 *(i)* One possible solution for representing graph G is the following. Let $L = \{$`Student`, `Professor`, `Course`$\}$ be the set of labels, $P = \{$`age`, `mark`, `credits`, `year`$\}$ the set of properties, and $E = \{$`hasTutor`, `tookExam`, `teaches`$\}$ the set of edge types. Each tuple of the relation `Student` is represented in G as a node having `Student` as a label, and having the properties `id` and `age`. For the property `id` of nodes representing students, i.e., those having `Student` as a label, we can define both an existence and a uniqueness constraint to capture the fact that it is a key. Similarly, each tuple of the relation `Prof` is represented in G as a node with the label `Professor` and the properties `pcode` and `age`, where existence and uniqueness constraints can be applied to `pcode`. Then, for each tuple of the relation `Course`, there exists a node in G with label `Course` and with the property `credits`. Each tuple $\langle p, c\rangle$ in the relation `Teaches` is represented by means of a directed edge of type `teaches` connecting the node having `pid`$= p$ to the node having `ccode`$= c$. For each tuple $\langle p, s, y\rangle$ of relation `Tutoring`, G contains an edge e of type `hasTutor` going

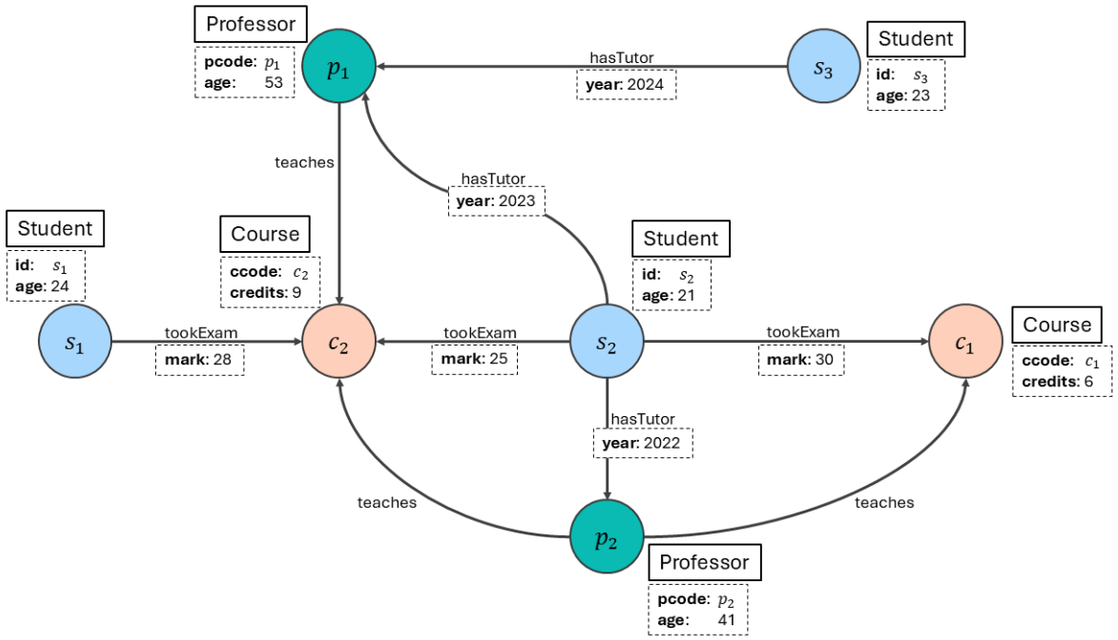

Figure 1: Property graph G corresponding to the database B.

from a node representing a student having `id`$= s$ to a node which represents a professor with `pcode`$= p$. Edge e also includes the property `year` whose value is y. Finally, each tuple $\langle s, c, m\rangle$ of relation `Exam` is represented in G as an edge e connecting a node with label `Student` and with `id`$= s$ to a node with label `Course` and with `ccode`$= c$, and such that e includes the property `mark` with value m.

(ii) Let database B be constituted by the following tuples: `Student` $= \{\langle s_1, 24\rangle, \langle s_2, 21\rangle, \langle s_3, 23\rangle\}$, `Exam` $= \{\langle s_1, c_2, 28\rangle, \langle s_2, c_2, 25\rangle, \langle s_2, c_1, 30\rangle\}$ `Course` $= \{\langle c_1, 6\rangle, \langle c_2, 9\rangle\}$ `Teaches` $= \{\langle p_1, c_2\rangle, \langle p_2, c_1\rangle\}, \langle p_2, c_2\rangle$ `Prof` $= \{\langle p_1, 53\rangle, \langle p_2, 41\rangle\}$ `Tutoring` $= \{\langle p_1, s_2, 2023\rangle, \langle p_1, s_3, 2024\rangle\langle p_2, s_2, 2022\rangle\}$. The corresponding property graph is depicted in Figure 1.

5.2 *(i)* When designing the database, we need to decide when to adopt a normalised (referenced) or denormalised (embedded) approach. Either comes with advantages and disadvantages in terms of redundancy, efficiency of query answering and complexity of the schema. The decision on whether to pick one or the other depends on the specific requirements.

We propose a solution consisting of three distinct collections: `students`, `courses`, and `professors`. Each document of the collection `students` includes the fields `id` (identifier within the collection), `age` and `exams`, where `exams` is associated to an array of documents with fields `ccode` and `mark`. The documents of the collection `professors` have fields `pcode` (identifier), `age`, `courses`, and `tutor`, where `courses` is an array containing the identifiers of the courses taught by a professor, and `tutor` is an array of documents with fields `stid` and `year`. Finally, the collection `courses` contains documents with fields `ccode` (identifier), and `credits`. Such a solution follows a denormalised approach (embedded) for both relations `Exams` and `Tutoring`, in that the corresponding information is completely encapsulated into the documents of the collections `students` and `professors`, respectively, while the information about the relation between professors and courses is managed through a normalised approach (referenced), since documents of the collection `professors` simply contain an array of references to the identifiers of the courses they teach. Depending on the specific requirements about the queries to be executed more often, one might decide to adopt different strategies, e.g., by normalising information about exams by creating a new collection `exams`, whose documents contain the fields `stid`, `ccode` and `mark`, and possibly adding references to such documents also in the collections `students` and/or `professors`.

(ii) The document-based database corresponding to the modeling provided in *(i)* is depicted in Figure 2 using the data from database B.

```
[{"id": "s1", "age": 24, "exams": [{"ccode": "c2", "mark": 28}]},
 {"id": "s2", "age": 21, "exams": [{"ccode": "c2", "mark": 25}, {"ccode": "c1", "mark": 30}]},
 {"id": "s3", "age": 21}]
```


professors


```
[{"pcode": "p1", "age": 53, "courses": ["c2"], "tutor": [{"stid": "s3", "year": 2024}, {"stid": "s2", "year": 2023}]},
 {"pcode": "p2", "age": 41, "courses": ["c1", "c2"], "tutor": [{"stid": "s2", "year": 2022}]}]
```


courses


```
[{"ccode": "c1", "credits": 6}, {"ccode": "c2", "credits": 9}]
```


Figure 2: Document-based database D, corresponding to database B.