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

Problem 1 Consider the relations `T(A,B)` and `V(A,F,G,H)`, where (*i*) both have `A` as key, (*ii*) `T` is stored in a heap with 120 pages (each page with 40 tuples) (*iii*) `V` is stored in a heap with 1.800 pages (each page with 20 tuples) with an associated hash index whose search key is `A` and (*iv*) the buffer has 62 frames available. If your goal is to compute the natural join (equi-join on `A`) between `T` and `V` as efficiently as possible in terms of number of page accesses, which algorithm would you choose among:

1.1 block-nested loop,
1.2 multi-pass based on sorting,
1.3 index-based.

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

Solution 1
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 `T`. So, the cost is $B(\mathtt{T}) + B(\mathtt{V}) \times \lceil B(\mathtt{T})/(62-2)\rceil = 3.720$.
3.2 Multi-pass based on sorting. We must determine the required number of passes, which is obviously greater than 1. Since $62 \times 61 = 3.782$ and $3.782 > 1800 + 120$, 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 $3 \times (B(\mathtt{T}) + B(\mathtt{V})) = 5.760$.
3.3 Index-based. Since the number $Tp(\mathtt{T})$ of tuples of `T` is $120 \times 40 = 4.800$ and since we assume 1 to be the cost of searching for a value of `B` in `V` using the hash index, the cost is $B(\mathtt{T}) + Tp(\mathtt{T}) \times 1 = 120 + 4.800 \times 1 = 4.920$.

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

Problem 2 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.

2.1 Prove or disprove the following two statements: (2.1.1) if $S = S''$, then S is view-serializable; (2.1.2) if S is view serializable, then $S = S''$.
2.2 Prove or disprove that S'' is ACR (Avoiding Cascading Rollback), and in case you disproved that S'' is ACR, tell how you would modify D in order to ensure that S'' is ACR.

Solution 2

2.1 Proving the statement "(2.1.1) if $S = S''$, then S is view-serializable" is easy: if $S = S''$, then D has found no cycle while processing S, and therefore S is conflict-serializable, which implies that S is view-serializable. We now disprove the statement "(2.1.2) if S is view serializable, then $S = S''$". Consider the schedule

$$S = w_1(x)\ w_2(x)\ w_2(y)\ w_1(y)\ c_1\ c_2\ w_3(x)\ w_3(y)\ c_3$$

It is easy to see that D aborts T_1 while processing S and therefore $S \neq S''$, but S is clearly view-serializable.

2.2 We disprove that S'' is ACR by simply considering the non-ACR schedule S

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

and noticing that, obviously, in this case $S = S''$, which implies that S'' is not in ACR. To ensure that S is ACR, the scheduler D could add a new rule as follows: when processing a read action $r_i(x)$ of transaction T_i, D executes such action if $r_i(x)$ is reading from an action $w_j(x)$ such that T_j has committed, otherwise it aborts and rollbacks T_i.

Problem 3 Consider the following schedule S:

$$w_0(x)\ r_1(x)\ w_2(y)\ w_3(x)\ r_4(x)\ w_1(y)\ w_2(z)$$

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

3.2 Is S a recoverable schedule? Motivate your answers in detail.

3.3 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 3

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

$$xl_0(x)\ w_0(x)\ u_0(x)\ xl_1(x)\ w_1(x)\ xl_2(y)\ w_2(y)\ xl_2(z)\ u_2(y)\ xl_1(y)\ u_1(x)\ xl_3(x)\ w_3(x)\ u_3(x)\ sl_4(x)\ r_4(x)\ u_4(x)\ w_1(y)\ u_1(y)\ w_2(z)\ u_2(z)$$

2.2 The schedule S is recoverable: it is sufficient to let transaction T_0 commit before T_1 and to let transaction T_3 commit before T_4.

2.3 Here is the behavior of the timestamp-based scheduler when processing S:

$w_0(x)$: read ok, $wts(x) = 0, cb(x) =$ `false`

$r_1(x)$: read ok, but T_1 put in a waiting queue because $cb(x) =$ `false`

$w_2(y)$: write ok, $wts(x) = 2, cb(y) =$ `false`

$w_3(x)$: write ok, but T_3 put in a waiting queue because $cb(x) =$ `false`

$r_4(x)$: read ok, but T_4 put in a waiting queue because $cb(x) =$ `false`

$w_1(y)$: write ok (to be ignored by Thomas rule), but T_1 put in a waiting queue because $cb(y) =$ `false`

$w_2(z)$: write ok, $wts(z) = 2, cb(z) =$ `false`

c_0: ok, $cb(x) =$ `true`

c_2: ok, $cb(y) = cb(z) =$ `true`

$r_1(x)$: read ok, $rts(x) = 3$

$w_1(y)$: write ok, but ignored by Thomas rule

c_1: ok, $cb(y) =$ `true`

$w_3(x)$: write ok, $wts(x) = 3, cb(x) =$ `false`

c_3: ok, $cb(x) =$ `true`

c_4: ok

Problem 4 Consider the relation `CONSTRUCTION(code,type,region,cost,year)`, 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 `region`. We know that no more than 200 constructions are allowed in the same region, 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 region, compute the code of all constructions in that region, together with the corresponding type; (2) insert a new construction. 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 region, the code of all constructions in that region, together with the corresponding type, 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 `type`. 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 `region`, 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 200 data entries (in the worst case), we know that we need to access $200/50 = 4$ 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., 200 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): $log_2 16.000 + 1 + 4 + 200 = 14 + 1 + 4 + 200 = 219$.

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 8.000 + 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 $400.000/20 = 20.000$ pages. So, for the insertion into the data file the cost is 20.000 + 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 `TaxiDriver(id,country)`, `Drives(driverid,taxi,since)`, `Taxi(tcode,type)`, `Own(ccode,tcode)`, `Company(ccode,budget)`, `Director(dcode,ccode,salary)`, where (*i*) each driver can drive many taxis (each one since a certain year) and each taxi can be driven by many drivers, (*ii*) each company can own several taxis and each taxi can be owned by several companies, (*ii*) each person can be the director of many companies (each one wirth a certain salary) and each company may have several directors.

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 = {`Driver`, `Taxi`, `Company`, `Director`} be the set of labels, P = {`id`, `tcode`, `ccode`, `dcode country`, `since`, `type`, `budget`, `salary`} the set of properties, and E = {`drives`, `owns`, `hasDirector`} the set of edge types. Each tuple $\langle t, c\rangle$ of the relation `TaxiDriver` is represented in G as a node having `Driver` as a label, and having the properties `id`$=t$ and `country`$=c$. For the property `id` of nodes representing taxi drivers, i.e., those having `Driver` 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 $\langle t, m\rangle$ of the relation `Taxi` is represented in G as a node with the label `Taxi` and the properties `tcode`$=t$ and `type`$=m$, where existence and uniqueness constraints can be applied to `tcode`. For each tuple $\langle c, b\rangle$ of the relation `Company`, there exists a node in G with label `Company` and with properties `ccode`$=c$ and `budget`$=b$. Finally, for each distinct value d of the attribute `dcode` in the relation `Director`, there exists a node in G with a property `dcode`$=d$. Each tuple $\langle d, t, s\rangle$ in the relation `Drives` is represented by means of a directed edge of type `drives` connecting a node with label `Driver` and having `id`$= d$ to a node with label `Taxi` having `tcode`$= t$. Such edge has a property `since` whose value is s. For each tuple $\langle c, t\rangle$ of relation `Own`, G contains an edge e of type `owns` going from a node representing a company having `ccode`$= c$ to a node which represents a taxi with `tcode`$= t$. Finally, each tuple $\langle d, c, s\rangle$ of relation `Director` is represented in G as an edge e of type `hasDirector` connecting a node with label `Company` and with `ccode`$= c$ to a node with label `Director` and with `dcode`$= d$, and such that e includes the property `salary` with value s. Notice that the salary is a property of the edge, since the salary of a director can vary for every company (s)he guides.

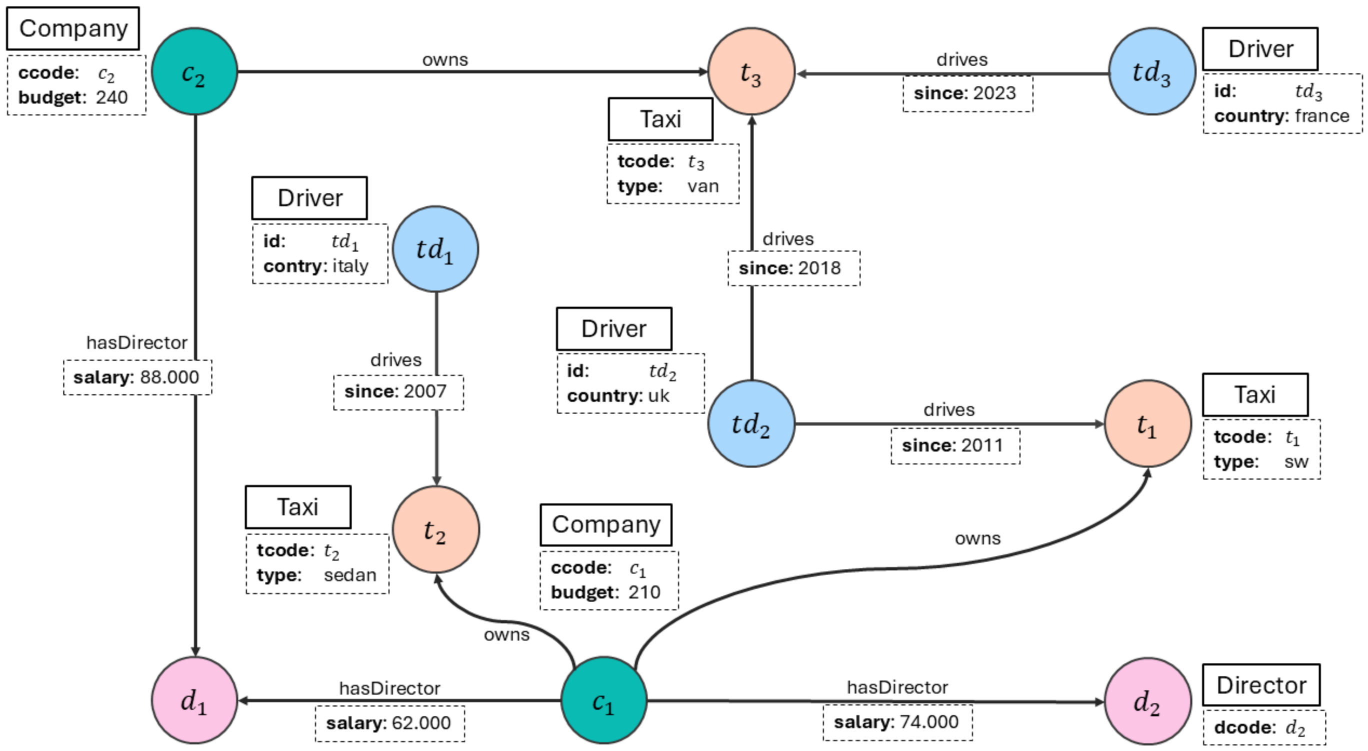

Figure 1: Property graph G corresponding to the database B.

ii) Let the database B be constituted by the following tuples: `TaxiDriver` $= \{\langle td_1, italy\rangle, \langle td_2, uk\rangle, \langle td_3, france\rangle\}$, `Drives` $= \{\langle td_1, t_2, 2007\rangle, \langle td_2, t_1, 2011\rangle, \langle td_2, t_3, 2018\rangle, \langle td_3, t_3, 2023\rangle\}$, `Taxi` $= \{\langle t_1, sw\rangle, \langle t_2, sedan\rangle, \langle t_3, van\rangle\}$, `Own` $= \{\langle c_1, t_2\rangle, \langle c_2, t_3\rangle\}, \langle c_1, t_1\rangle$, `Company` $= \{\langle c_1, 210\rangle, \langle c_2, 240\rangle\}$, `Director` $= \{\langle d_1, c_1, 62.000\rangle,$
$\langle d_1, c_2, 88.000\rangle, \langle d_2, c_1, 74.000\rangle\}$. The corresponding property graph is shown 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: `drivers`, `taxis`, and `companies`. Each document of the collection `drivers` includes the fields `id` (identifier within the collection), `country` and `vehicles`, where `vehicles` is associated to an array of documents, each consisting of the fields `tcode` which is a reference to the identifier of a taxi, and `since`. The documents of the collection `taxis` have fields `tcode` (identifier), `type`, and `owners`, where `owners` is an array containing the identifiers of the companies which own the vehicle. Finally, the collection `companies` contains documents with fields `ccode` (identifier), `budget`, and `directors`, which is an array embedding documents describing directors, each with a field `dcode` and a field `salary`. Such a solution follows a normalised approach (embedded) for relation `Own`, in that the corresponding information is simply referenced into the documents of the collection `companies`, while the information about the relations `Director` and `Drives` is managed through a denormalised approach (referenced), since documents of the collection `companies` embed the documents with the information about their directors and the corresponding salaries, while documents of the collection `driver` embed the information about the vechicles they drive. Depending on the specific requirements about the queries to be executed more often, one might decide to model the database in a different way.

ii) The document-based database corresponding to the modelling provided in *5.2 i)* is depicted in Figure 2 using the data from database B.

drivers

```
[{"id": "t1", "country": "italy", "vehicles": [{"tcode": "t2", "since": 2007}]},
 {"id": "t2", "country": "uk", "vehicles": [{"tcode": "t1", "since": 2011}, {"tcode": "t3", "since": 2018}]},
 {"id": "t3", "country": "france", "vehicles": [{"tcode": "t3", "since": 2023}]}]
```

taxis

```
[{"tcode": "t1", "type": "sw", "owners": ["c1"]},
 {"tcode": "t2", "type": "sedan", "owners": ["c1"]},
 {"tcode": "t3", "type": "van", "owners": ["c2"]}]
```

companies

```
[{"ccode": "c1", "budget": 210, "directors": [{"dcode": "d1", "salary": 62.000}, {"dcode": "d2", "salary": 74.000}]},
 {"ccode": "c2", "budget": 240}, "directors": [{"dcode": "d1", "salary": 88.000}]]
```

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