DMA1 Exam

Exercise 1

1) Consider the following database of a given company

EMPLOYEE(**TaxCode**, Code, Name, Surname, BirthDate, AppointmentDate, Role, Division)
DIVISION(**CODE**, Function, HeadDivision)
PROJECT(**CODE**, Title)
PARTICIPATION(**EmpCode,ProjCode,** StartDate, EndDate)

NOTE:

EMPLOYEE.Division refers to a value in DIVISION.Code

EMPLOYEE.Role can take values "Technician", “Admin”, “Head of Division”

DIVISION.HeadDivision refers to a value in EMPLOYEE.Code

DIVISION.Function can take values “Customer Care”, “Development”, “Accounting”

PARTICIPATION.EmpCode refers to a value in EMPLOYEE.Code

PARTICIPATION.ProjCode Refers to a value in PROJECT.CODE

PARTICIPATION.StartDate and PARTICIPATION.EndDate are the starting and ending dates, respectively, in which an EMPLOYEE participated in a project.

All dates are in the format YYYY-MM-DD

a) Find the tax codes, names and surnames of the head of division of “Development” in which there is some EMPLOYEE involved in the project “SUPERCAR”

b) Find the tax codes, names and surnames of the employees that joined the project “CASAMIA” as last.

In order to solve this, we need to know how the operators used to make queries:

- Selection(σ): Takes a subset of tuples.
- Projection(π): Takes a subset of columns.
- Join($\bowtie$): There are apparently two kinds:
  - Natural: We join on attributes with the same name (we don't use this).
  - Theta: We choose the fields to use for the join (even if they have the same name)
- Cartesian product(X): We join the two tables and we end up with every tuple combined with every other tuple.
- Rename(p): Renames attributes, can be useful for joins.

> **Note**
> The grade of a relation is the number of columns.

Prof's solution:

1. We get the code of the Project supercar by selecting code=Supercar, we name the table as supercar_code.
2. We want to find the employees that have that particular supercar project code, so we join participation and supercar_code.
3. We are left with the codes of the employees that are involved in the supercar project, now we just join employee with the new table.
4. But we actually want only the employees that have an head of division with function=Development

Takeaways

- We can use the join to filter.
- After we create new tables with the join we can name them
- We can subtract tables.
- In order to get the maximum or minimum of a table, we can do the cartesian product of the table with itself, and then select with a comparative condition and eliminate all the tuples that don't satisfy it.

Exercise 2a

2a) Consider the relational schema R=ABCDE, the set of functional dependencies F={AC→E, AE→CD, CE→B, DC→EB} and the decomposition ρ={ABCE, CD} of R. Say whether ρ preserves F. Motivate and show the procedure adopted for your answer.

Checking for equivalence

Two sets of functional dependencies F and G are equivalent if

$$F^+ = G^+$$

Obviously, F and G are not the same, but the two supersets of all axiom-derivable dependencies must be.

So how do we check if the two closures are equivalent?

$$F^+ \subseteq G^+ \quad \text{and} \quad F^+ \supseteq G^+$$

How to make this shit faster

Obviously computing this would take too much time so we use this lemma:

$$\text{If} \quad F \subseteq G^+ \quad \text{then} \quad F^+ \subseteq G^+$$

This is because if G^+ contains F, then axioms can be used to obtain all of F^+.

In order to determine whether a decomposition with dependencies G preserves F, we need to check if $F \subseteq G^+$.

For every dependency we must know if the right side of the dependencies are contained within X_G^+.

These are the steps to compute X_G^+:

1. For each table in the decomposition, we start with two sets:
   - Z: contains the attributes functionally determined by X in the decomposition.
   - S: to be used in a loop, progressively collects the attributes that are determined by Z in each sub-schema.
2. We use the following formula in a loop(i++) to collect the determined attributes from each sub-schema:

$$S \mathrel{+}= \underbrace{\underbrace{\left(\underbrace{Z \cap R_i}_{\text{Part of X also in R}_i} \right)_F^+}_{\text{Attributes determined}} \cap R_i}_{\text{Determined attributes that are also in R}_i}$$

3. We add S to Z.
4. Repeat step 2 and 3.

Example

$R = ABCDE$,
$F = \{AC \rightarrow E, AE \rightarrow CD, CE \rightarrow B\}$
$\rho = \{ABCE, CD\}$ (decomposition)

We want to verify that $AC \rightarrow E$ is satisfied.

$Z = AC$

$$S \mathrel{+}= (AC \cap ABCE)_F^+ \cap ABCE, \quad S \mathrel{+}= (AC \cap CD)_F^+ \cap CD$$

$$S \mathrel{+}= (AC)_F^+ \cap ABCE, \quad S \mathrel{+}= (C)_F^+ \cap CD$$

$$S \mathrel{+}= ACE \cap ABCE, \quad S \mathrel{+}= \emptyset \cap CD$$

$$S \mathrel{+}= ACE, \quad S \mathrel{+}= \emptyset$$

$Z = ACE$

We stop the loop here, since E is inside Z.

If this wasn't the case, we would have needed to do another iteration of the loop.

Exercise 2b

2b) Consider the relational schema R=ABCDEG, the set of functional dependences F={ A→GB, GC→ED, E→B, BE→A} and decomposition ρ={AGB, ADE, CDG} of R. Say whether ρ has a lossless join and show the procedure adopted for your answer.

What is a Lossless Join?
We have a lossless join when upon applying natural join to a decomposition of R, we get exactly R.

Steps for checking whether a join is lossless or not:

1. We construct a table where the rows represent the sub-schemas, and the columns represent the individual attributes of R. In each cell, we put a_j if the attribute is contained in the subschema, we put b_{ij} otherwise (i=row, j=column).

for every $X \rightarrow Y \in F$ — Index j = attribute = column

if there are two tuples t_1 and t_2 in r such that $t_1[X]=t_2[X]$ and $t_1[Y]\neq t_2[Y]$
then for every attribute A_j in Y: **if** $t_1[A_j]=a_j$ **then** $t_2[A_j] = t_1[A_j]$
else $t_1[A_j] = t_2[A_j]$

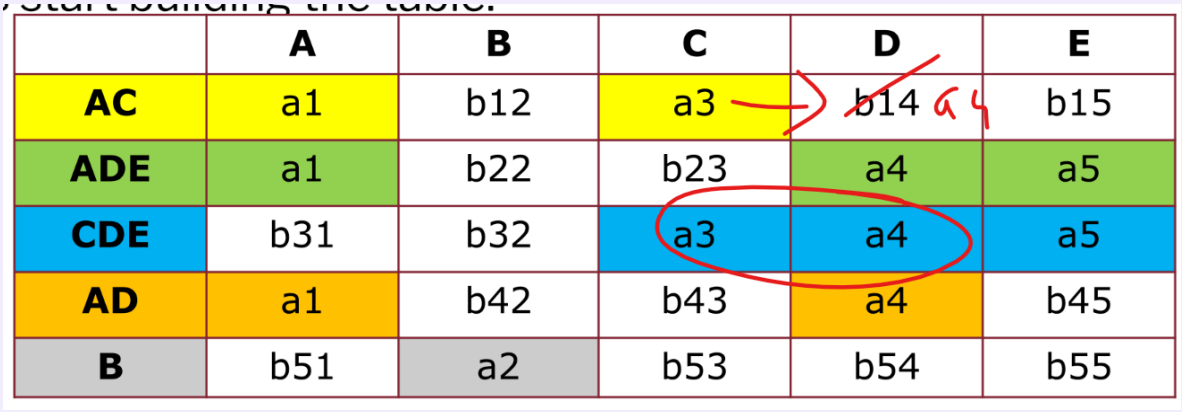
2.

3. You loop until no more changes can be made or if one row is full of a's. If we have a row full of a's the join is lossless, otherwise it is lossy.

Example
$R = (A, B, C, D, E)$
$F = \{C \rightarrow D, AB \rightarrow E, D \rightarrow B\}$
Say whether decomposition
$\rho = \{AC, ADE, CDE, AD, B\}$
has a lossless join.

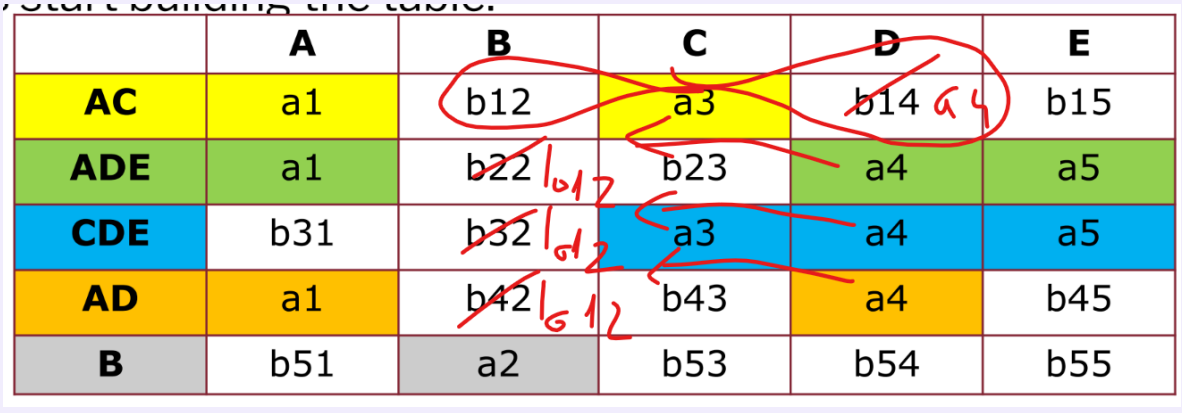
	A	B	C	D	E
AC	a1	b12	a3	b14	b15
ADE	a1	b22	b23	a4	a5
CDE	b31	b32	a3	a4	a5
AD	a1	b42	b43	a4	b45
B	b51	a2	b53	b54	b55

We examine the dependency $C \rightarrow D$, and we can see that C has 2 rows(a's) in its columns, so we need to make the elements in the columns of Y equal in correspondence of those a's.



We examine the dependency $AB \rightarrow E$, there are no rows in which A and B are equal, so we have no way of verifying the dependency right now(already verified).

We examine the dependency $D \rightarrow B$, we can see that D has 3 equal rows, so we make the rows of B equal.



We have completed the first loop and we don't have a row full of a's, but changes can still be made, so we keep going.

We examine the dependency $C \rightarrow D$, which is already satisfied.

	A	B	C	D	E
AC	a1	b12	a3	~~b14~~ a4	b15
ADE	a1	~~b22~~ b12	b23	a4	a5
CDE	b31	~~b32~~ b12	a3	a4	a5
AD	a1	~~b42~~ b12	b43	a4	b45
B	b51	a2	b53	b54	b55

We examine the dependency $AB \to E$, and we can make some changes:

	A	B	C	D	E
AC	a1	b12	a3	~~b14~~ a4	~~b15~~ a5
ADE	a1	~~b22~~ b12	b23	a4	a5
CDE	b31	~~b32~~ b12	a3	a4	a5
AD	a1	~~b42~~ b12	b43	a4	~~b45~~ a5
B	b51	a2	b53	b54	b55

We examine the dependency $D \to B$, which is already satisfied.

We keep the loop going.

	A	B	C	D	E
AC	a1	b12	a3	a4	a5
ADE	a1	b12	b23	a4	a5
CDE	b31	b12	a3	a4	a5
AD	a1	b12	b43	a4	a5
B	b51	a2	b53	b54	b55

No more changes can be made at all.
Since we don't have a row full of a's, our join is lossy.

Exercise 2c

2) Consider the relation schema R=ABCDEFG and the set of functional dependencies F={AB→CD, AC→BD, BDE→F, DE→G, G→F}

a) Find the two keys of the schema and show the procedure adopted;

~~b) Decide whether the schema is in 3NF and justify your answer;~~

~~c) Find a decomposition of R with all subschema in 3NF, preserves F and has a lossless join.~~

Definition of Key

A subset K of a relation is a key if:

- $K \to R \in F^+$: K determines all other attributes.
- There are no subsets K' such that $K \to R \in F^+$: There are no subsets of K that determines all other attributes.

Closure of a set of attributes

Steps for checking the closure of attributes X^+:

1. We have two sets:
   - Z: Contains the attributes in the closure at the current iteration, at $i = 0$ $Z = X$.
   - S: Temporarily contains the attributes that are determined by Z at each iteration.
2. We loop:
   1. At each iteration we add to S the single attributes that make up the right-hand parts of dependencies in F whose left part is contained in Z.
   2. Add S to Z.

Example

$F = \{AB \to CD, AC \to BD, BDE \to F, DE \to G, G \to F\}$
$R = ABCDEFG$

We want to calculate the closure of AB.

1. $Z = AB, \quad S \mathrel{+}= \{ \underbrace{C, D}_{AB \to CD} \}$
2. $Z = ABCD, \quad S \mathrel{+}= \emptyset$

$(X)^{+=} ABCD$

We leverage the algorithm for the closure of attributes to find the keys of a schema.

1. We bruteforce test some subsets of the schema and we check if they are a key(by using X^+ algorithm).
2. Once we find some keys, we test the subsets of the keys.

Hint

1. Start from those with higher cardinality. If their closure does not contain R, it is not necessary to check their subsets.
2. The attributes that never appear on the right of the functional dependencies of F, are not functionally determined by any other attribute, so they must be in all the keys of the schema.
3. A brute force approach is not wrong but can be inefficient.

Example

$R = ABCDEH$
$F = \{AB \to CD, C \to E, AB \to E, ABC \to D\}$

We begin by testing ABH:

1. $Z = ABH, \quad S \mathrel{+}= \{C, D, E\}$
2. $Z = ABCDEH, \quad S \mathrel{+}= \{\emptyset\}$

We conclude that ABH is a at least a super key.

We gotta check for subsets. Since H doesn't appear in any dependency, it must be in any key. So we only need to check AH and BH.

We test AH:

1. $Z = AH, \quad S \mathrel{+}= \{\emptyset\}$

We test BH:

1. $Z = BH, \quad S \mathrel{+}= \{\emptyset\}$

Neither of those subsets is a key, so the only key is ABH.

Exercise 2d

2) Consider the relation schema R=ABCDEFG and the set of functional dependencies F={AB→CD, AC→BD, BDE→F, DE→G, G→F}

~~a) Find the two keys of the schema and show the procedure adopted;~~

b) Decide whether the schema is in 3NF and justify your answer;

~~c) Find a decomposition of R with all subschema in 3NF, preserves F and has a lossless join.~~

Some definitions

- **Super key**: Set of attributes that functionally determines all the other attributes. Can be a superset of a key.
- **Candidate key**: Minimal superkey, doesn't contain unneeded attributes.
- **Prime set of attributes**: Every attribute of the set belongs to a candidate key.

A relation is in 3NF if for every functional dependency $X \to Y$, at least one of the following holds:

- X is a superkey.
- Y is prime.

Example

$R = ABCDEFG$
$F = \{AB \to CD, AC \to BD, BDE \to F, DE \to G, G \to F\}$
$\text{Keys} = \{ABE, ACE\}$

- $AB \to CD$: AB is not superkey, CD is not prime.

3NF already doesn't hold.

Exercise 2e

2) Consider the relation schema R=ABCDEFG and the set of functional dependencies F={AB→CD, AC→BD, BDE→F, DE→G, G→F}

~~a) Find the two keys of the schema and show the procedure adopted;~~

~~b) Decide whether the schema is in 3NF and justify your answer;~~

c) Find a decomposition of R with all subschema in 3NF, preserves F and has a lossless join.

Definition of Minimal Cover

A minimal cover is a set of functional dependencies $X \to Y$ equivalent to F such that:

- Each Y is non-redundant.
- Each X is non-reduntant.
- Each functional dependency is non-redundant.

1. We split the dependencies by their dependent attributes.

$$X \to AB \quad \Rightarrow \quad \{X \to A,\ X \to B\}$$

2. We try to reduce the left side of the dependencies. We want to remove the attributes that can be removed without changing the right side of the dependencies. If one subset of attributes determines the right side, remove the rest of the set.

Example

Assume we have dependencies
$F = \{AB \to C, AB \to D, AC \to B, AC \to D, BDE \to F, DE \to G, G \to F\}$.

- $AB \to C$? $(A)^+ = \{A\},\ (B)^+ = \{B\}$ Nothing wrong here.
- $BDE \to F$? $(BD)^+ = \{BD\},\ (DE)^+ = \{DEGF\}$

Here DE alone contains the right side of the dependency, so there is no reason to keep BD in anymore(we remove B as D is needed).

3. We remove the reduntant dependencies, those that if removed from F, F doesn't change.

$$F \equiv F - \{X \to A\}$$

Example

$F = \{AB \to C, AB \to D, AC \to B, AC \to D, DE \to F, DE \to G, G \to F\}$

$AB \to C$? $(AB)^+_{F-\{AB \to C\}} = \{ABD\}$ ABD doesn't contain C, so this is non-redundant.
$AB \to D$? $(AB)^+_{F-\{AB \to D\}} = \{ABCD\}$ $ABCD$ contains D, so this is redundant and must be removed.

TO BE CONTINUED.

Exercise 3 - HASH

3) Assume to have a file with 19.000.000 records. Each record is of size 355 bytes. Each block is of size 2048 bytes and a block pointer is of 5 bytes. We are using a HASH organization with 400 buckets. Compute:

- number of blocks for the bucket directory
- number of block for each bucket
- average number of access to find a record using its key
- total number of records that can be inserted in the file without increasing the average number of access for the search.

> ⚠ **Warning**
> When we compute "Records per block", "Pointers per block" or "Records per bucket", we always floor round for some fucking reason.

> **Principles of Physical Storage**
> Usually we have an index file, to store the pointers to blocks or buckets, and we have the main file, which actually contains the data divided in blocks or buckets.

> **Heap database definition**

Main file

The main file is divided in blocks.

Block

Each block can host n records. Every time a record is added, it is added as last record, so all the blocks up until that point are filled, while the last ones are empty.

Search - Loop

When we need to find a record, we must iterate through every block until we find the record. If the record is in block k, we must do k read accesses.

> **Hash database definition**

Main file

The main file is divided in buckets, enumerated from 0 to $\text{number of buckets} - 1$.

Bucket

A bucket is made up of multiple blocks and is organized like a heap. Each block in the bucket has a pointer to another block in the bucket.

Bucket directory

File that contains the pointers to each bucket.

Search - Hash function

Given a key, an hash function returns the corresponding bucket. A good hash function distributes the keys evenly among the buckets.

Number of blocks in bucket directory

We can compute the number of blocks required to store the entire bucket directory by computing how many bytes there are in total and checking how many blocks are needed to store that many bytes.

$$\frac{5 \cdot 400}{2048} = \lceil 0.98 \rceil = 1$$

> **Hint**
> Obviously we use ceiling rounding with this one, otherwise we wouldn't have space for the data lol.

Number of blocks in each bucket

We compute the number of blocks in each bucket by:

- dividing the total number of records by the number of buckets -> number of records in each bucket(we floor round this).

> **Hint**
> We floor round this because in the next steps we are gonna use a ceiling on the number of blocks for each bucket, so we try not to put too many records in advance.

- Then we multiply the number of records -> number of bytes that our buckets must contain.
- Then we dividie the number of bytes in each bucket for the block size -> number of blocks in each bucket(we ceil round this).

Average number of memory accesses to find a record

The buckets are orgnized like a heap, so we need number of blocks / 2.

Exercise 3 - BTreez

3) Assume we have a file of 16.450.000 records. Each record is of size 250 bytes, 15 of which for the field key..Blocks are of size 2048 bytes. A block pointer is of size 5 bytes. We use a B-tree organization where both **blocks of the main and index file are filled to the minimum.**
Compute**:**

- total number of blocks for the main file
- total number of block for the index file
- max number of block accesses to find a record in the main file

B-Tree Definition

Index file

The index file is still divided in blocks, but now it has a hierarchical index. The high order indexes will point to low order indexes and those will point to the actual main file blocks.

Additional info

The blocks must be at least half filled.

Search cost

WIP

Number of blocks in main file

We know that we need to keep the blocks at least half filled, so we find out how many records we can fit in half a block.

$$(2048/2)/250 = \lceil 4.096 \rceil = 5$$

Hint

We ceil round because we want more than half.

Number of blocks in index file

We still need to fill half the blocks.
A record in the index file is a key + pointer, so in this case 15 + 5 = 20 bytes.

Now, a block in the index file also has an additional pointer, so we need to remove it from the calculation.

$$(1024 - 5)/20 = \lceil 50.95 \rceil = 51$$

So in each block of the index file there are 51 keys and 52 pointers.

> **Note**
> Each layer of the index files has blocks of the same dimension. They just have more blocks.

Search cost

Every serch requires $h + 1$ accesses, where h is the height of the tree.

To find out the height, you just keep dividing the number of records by the number of pointers, until you get 1.