Relational Model: Formative Assessment 1

Feedback: Overall, I received well-descriptive relational schemata with well-defined PKs and FKs that materialize the relationships between relations! Many of you captured the underlying semantics for the one-to-many and many-to-many relationships like the academic position relation and the co-authorships. Moreover, many solutions dealt with associating relations using only the correct foreign and primary keys, which is the core component in the relational database design (Guideline No. 1 in our recent lecture). There were also some submissions including instances/tuples to provide more clarification on their models!

Hereinafter, I provide some comments/feedback over certain issues raised from some solutions:

There were some relational schemata without FKs (or FKs pointing to non-PKs). Recall, that the FKs are **mandatory** in every relational schema design since they establish the referential integrity constraints and provide the meaning of the 'relational modelling'. Hence, we should avoid having relations without any reference (participating either as referencing or as referenced relation) in the provided schema. The PKs are **mandatory** to every relation 'by definition of' the relational modelling, i.e., we need to define a specific subset of attributes to uniquely identify tuples.

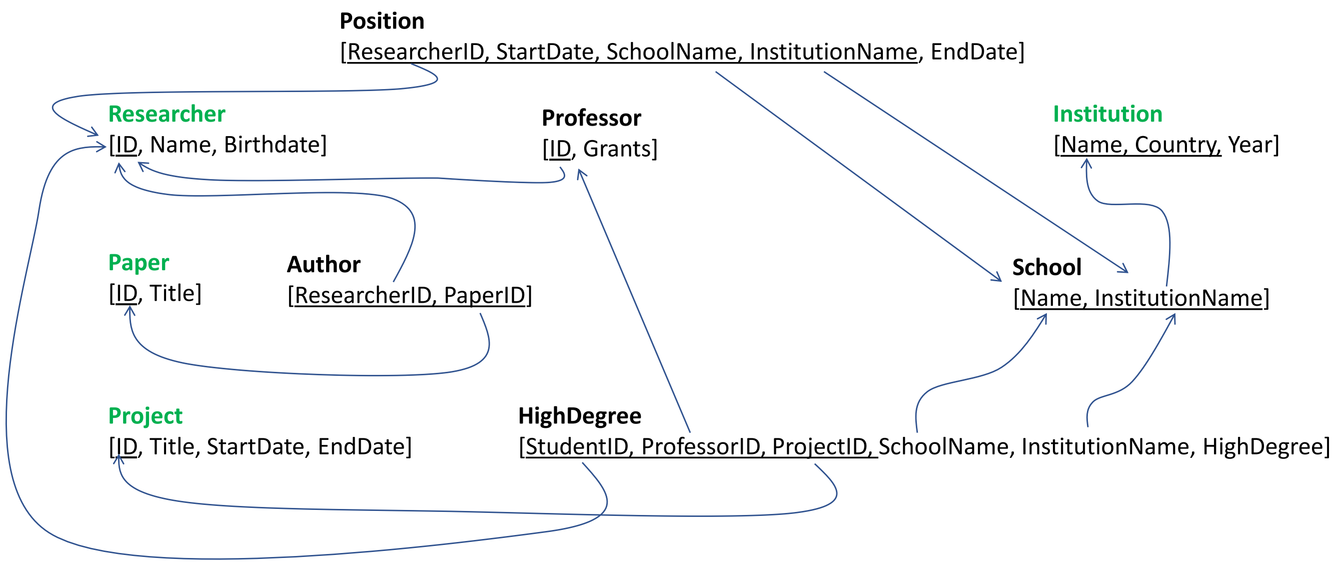
When drawing a FK, the arrow points towards the PK of the referenced relation departing from the referencing relation, e.g., in the provided schema below, ProfessorID in the HighDegree relation points to the ID in the Professor relation demonstrating that the former is the FK and the latter the PK.

For the co-authorship concept, one can associate an author (researcher) with a paper (publication) in a 'relationship' relation defining a composite primary key: research-id authors a paper-id (like in our example Company schema: employee-id works on a project-id in the WORKS_ON relation). Then, based on this association relation, one can retrieve the co-authors if we search for those authors/researchers having authored the same paper (paper-id); or in our example Company schema, we can find our colleagues/co-workers in a project by selecting those employees having the same project-id in the WORKS_ON relation. This can be achieved by a 'recursive' selection query, which we will be discussing in our SQL lecture this week. Hence, we do not need to 'constraint' our schema by putting a fixed number of co-authors to denote this relationship.

Finally, in the relational model, there is no notation on the cardinality ratio (e.g., 1:N, N:M, 1:1) between relations. When we need to model e.g., an **one-to-many** relationship between relations A and B, then, we associate the PK of the relation A with the FK of the relation B denoting that **one** tuple from A is associated with **many** tuples from relation B and vice versa. Consider in our example Company schema that an employee (relation A is EMPLOYEE) has many dependents (relation B is DEPENDENT). For a **many-to-many** relationship, consider the WORKS_ON relation associating an employee with many projects, and on the other side where a project is having more than one employee. In this case, such type of relationship is modelled via a new association relation having two FKs: one for each associated relation (look at the Guideline No. 1 in our recent lecture).

Possible Schema: I provide a possible schema that derives from the provided textual description (some of your colleagues come up with very similar model). Note that this solution is a normalized relational schema dealing with all the requirements for efficient relational modelling, which is the objective of the last lecture being in the BCNF.

Note: The provided schema is **not** unique; you can definitely derive different schemata. The relations in green correspond to basic relations, i.e., relations that do not reference to any other relations.



University of Glasgow
School of Computing Science

Formative Assessment 2: Feedback

Overall Feedback

Almost all of you found the FDs that violate the relations being in the BCNF and successfully applied the corresponding Theorem for transforming the relations to BCNF smaller relations. I am providing some details on both tasks demonstrating the current problems and why we need to further normalize the relations avoiding having fictitious tuples. Also, some of you provided a detailed description in Task 2 and, after applying the BCNF theorem, explained how to merge relations having the same determinants in their PKs (referring to the relations R2 and R4 in Task 2 solution).

Task 1

Assume the following relation **R** where it holds true that:

- Each manager works in a particular branch.
- Each project has several managers and runs on several branches.
- A project has a unique manager for each *branch*.

R

Manager	Project	Branch
Chris	Mars	Glasgow
Green	Jupiter	London
Green	Mars	London
John	Saturn	London
John	Venus	London

Q1: Which is the NF of the relation?

Since the manager attribute functionally determines the branch, then we obtain the FD: Manager → Branch. Based on this FD, the relation cannot be in the BCNF because the Manager is a non-prime attribute and not part of any PK. However, the relation is in the 3rd NF since there are no transitive functional dependencies of non-prime attributes on the PK via non-prime attributes.

Q2: Which FD violates the BCNF?

Given the explanation in Q1, the FD: Manager → Branch violates the BCNF.

Q3: Which is the splitting attribute such that **R** is transformed into a set of BCNF relations?

To decide on the splitting attribute, we need to apply the BCNF theorem over the violating FD. That is, the relation is split accordingly to:

- R1: R \ {Branch} = {Manager, Project}
- R2: {Manager} U {Branch} = {Manager, Branch}

Hence, the splitting attribute is: Manager. We need also to establish the PKs and FKs for the split relations. In R2 we obtain that: Manager $\rightarrow$ Branch. In R1, both Manager and Project uniquely identify each tuple. Also, the Manager in R2 is a FK pointing to R1.Manager or vice versa, since the attribute manager cannot be null in any case. The splitting has then as follows:

R1

<u>Manager</u>	<u>Project</u>
Chris	Mars
Green	Jupiter
Green	Mars
John	Saturn
John	Venus

R2

<u>Manager</u>	Branch
Chris	Glasgow
Green	London
John	London

If we join the relations R1 and R2 based on the common attribute Manager, we get the original information in R.

Now, as an experiment, let us split the original relation R arbitrarily by choosing let's say the attribute Branch. The corresponding relations will be: R3 (Project, Branch) and R4(Manager, Branch), i.e.:

R3

<u>Project</u>	<u>Branch</u>
Mars	Glasgow
Jupiter	London
Mars	London

Saturn	London
Venus	London

R4

Manager	Branch
Chris	Glasgow
Green	London
John	London

If we join R3 with R4 based on their common attribute Branch, we derive the following relation:

Manager	Project	Branch
Chris	Mars	Glasgow
Green	Jupiter	London
John	Jupiter	London
Green	Mars	London
John	Mars	London
Green	Saturn	London
John	Saturn	London
Green	Venus	London
John	Venus	London

Hmm.., we generate many fictitious tuples!

Q4: If you change the branch 'London' to 'York' how many tuples you need to update in the original version of the relation R and in the relation in BCNF?

In the original relation, we need to update four tuples when updating London to York, even if they correspond to two managers! In the BCNF schema, we need to update only two tuples in R2 corresponding exactly to the two managers, and this is the minimum number of updates we can do 😊

Task 2

Consider the relation: **R**(A, $\underline{B}$, C, D) and the asserted FDs:
FD1: C → D
FD2: C → A
FD3: B → C

Decompose the relation **R** into a set of BCNF relations using the BCNF Theorem.

First, we need to find which are those FDs that cause the BCNF violation. We obtain then that:

Proposition 1: C → D and C → A both cause *violations*
Because: C is not part of any candidate key (*recall*: B is the key)

We then apply the Theorem for decomposition.

Let us focus on the FD: C → D; We then *decompose* **R** to

- **R1:** **R**\{D} = {A,B,C,D} \ {D} = {A, B, C}: **R1(A, $\underline{B}$, C)**
- **R2:** {C} U {D} = {C, D}: **R2($\underline{C}$, D)**

Let us check now for violations in R1 and R2; Do you see any *further* BCNF violations?

Proposition 2: **R1**(A, $\underline{B}$, C) *still* violates BCNF because of C → A and C is not candidate key. We apply *again* the theorem and focus on the FD: C → A. In this case, we *decompose* R1 to:

- **R3: R1** \ {A} = {A,B,C}\{A} = {B, C}: **R3($\underline{B}$, C)**
- **R4:** {C} U {A} = {C, A}: **R4($\underline{C}$, A)**

The current outcome is then:

- **R2**($\underline{C}$, D) is in BCNF
- **R3**($\underline{B}$, C) is in BCNF
- **R4**($\underline{C}$, A) is in BCNF

Nonetheless, we can merge R2 and R4 since they both refer to the same PK. Hence, we obtain:

- **R3**($\underline{B}$, C) is in BCNF
- **R5**($\underline{C}$, D, A) is in BCNF by merging R2 and R4

And this is a visualization:

$$\begin{tikzcd} \textbf{R3} & \textbf{R5} \\ {[\underline{B},C]} \arrow[r, bend right] & {[\underline{C},D, A]} \end{tikzcd}$$

Formative Assessment 3: Feedback

Overall Feedback

You proposed very informative SQL statements dealing with all the problems! Regarding the Problem 1, the adoption of the NOT EXISTS operator is the key element. I therefore provide some explanation about how to 'execute' this SQL statement in an example. In Problem 2.1, the key element is the clustering of the relation with the unique attribute SSN in order to find out any duplications. In Problem 2.2, I provide a solution on how one can use an encapsulated relation within a SQL statement, where in many cases you might need to define in an ad-hoc manner attributes of this encapsulated relation which can be accessed by an outer SQL query. In Problem 3, most of the solutions you provided included not-so-standard SQL syntax. Specifically, there are many variations from the 'Standard' SQL in order to provide some 'programming tricks' to write a statement. However, when you are using some variants of the SQL you must make sure that this syntax can be executed to most of the SQL-based Systems. I therefore provide a solution based only on the 'conventional' operators in SQL.

Problem 1: Location-based Services

In Location-based services (LBS), we are given a GPS point **x** = [x_1, x_2], i.e., our location in coordinates (*longitude*, *latitude*), and we try to find the closest point **q** to our current point **x** from a set of points-of-interest. For instance, we would like to develop a LBS that at every time instance it finds the closest Gas Station or Restaurant to our current position.

In order to quantify the distance between two location points, we adopt the geospatial Euclidean distance which is the square root of the summation of the squared difference of the location coordinates of the points **x** and **q** defined as:

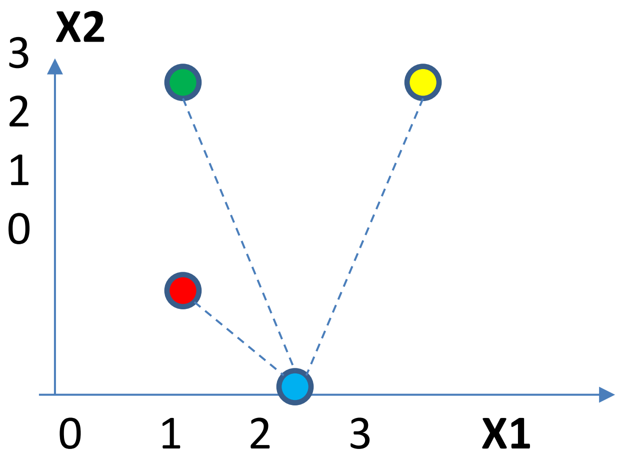
$d(\mathbf{x}, \mathbf{q}) = \text{SQRT}\{ (\mathbf{x}.x_1 - \mathbf{q}.x_1)^2 + (\mathbf{x}.x_2 - \mathbf{q}.x_2)^2 \}$ Eq(1)

where SQRT{z} is the square root of the number z. Now, back to the databases 😊 Assume that we store all the points of interest in the following relation Point, where for each point we provide the two coordinates X1 and X2, corresponding to the longitude and latitude, and a Description, e.g., M&S, Restaurant, etc. We also assign a unique ID number for each point.

Point(<u>ID</u>, X1, X2, Description)

This is an instance of the relation Point and a corresponding example over the 2-dimensional plane:

<u>ID</u>	X1	X2	Description
P1	3	3	Costa Café
P2	1	3	RBS
P3	1	1	M&S



Task: Assume that the blue point on the plane is out current position **p** = [2, 0], i.e., the longitude X1=2 and the latitude X2 = 0. Moreover, we can observe the other points corresponding to the locations of the Costa Café, RBS, an M&S. Our task is to retrieve the closest point **q**, i.e., which is the closest point-of-interest from the relation Point, to our current position **p**. Obviously, the provided SQL query should return only the point P3, which is the M&S. Write an SQL query using that the distance between points is a built-in function as provided in Eq(1).

Hint: *If the point* **q** *is the closest to a point* **p** *then there does not exist any other different point* **r** *closer to* **p** *than* **q** (**r** ≠ **q**)*. Hence, dealing with* `EXISTS` 😊

Solution. The concept is to follow the <u>hint</u> such that given a point **q**, we check if there does not exist any other point **r** (different than **q**) which is closer to our location **p** than the point **q**. Then, if we <u>cannot</u> find any other closer point, we return the point **q**. Said that, we adopt the `EXISTS` operator to check if the set of points which are closer to **p** than to **q** is empty! In this case, the point **q** is the point we are searching for…let's see now the nested correlated SQL query:

```
SELECT q.Description, q.X1, q.X2

FROM Point q

WHERE NOT EXISTS (SELECT * FROM Point r

                  WHERE r.ID <> q.ID

                  AND d(r, p) < d(p, q))
```

Let's execute this query; envisage the nested query as a two-loop process.

Target: Select point **q** where the nested query returns a **NULL set!**
Step 1: Get the point **q** from the relation Point
Step 2: Set return set RESULT = {}. It is empty, initially.
Step 3: Given the point **q**, get a point **r** from Point
Step 4: Check if the two points are different: **q**.ID <> **r**.ID
Step 5: Check if this holds true for the distances: D(**r, p**) < D(**q, p**)
Step 6: If Steps **4 & 5** hold TRUE Then
    RESULT = RESULT U {**r**}, i.e., we add point **r** to the RESULT set.
    Else the RESULT set remains unchanged.
Step 7: Repeat that for all points **r** in Point
Step 8: If RESULT == {}, i.e., RESULT set is empty, then the *closest* point is **q**
Step 9: Go to Step 1 and repeat…

Problem 2: DB Maintenance & Data Cleaning

Consider the known EMPLOYEE(SSN, FName…, DNO) relation but the designer did not declare the SSN to be the PK, thus, there might be duplications of employees…Our task is to 'clean' the EMPLOYEE relation by finding the duplicate entries.

Task 1: Provide a SQL query which returns the duplicate employee entries.

As an example, the relation is e.g.,

EMPLOYEE

SSN	FName
1	Philip
1	Philip
2	Stella
2	Stella
3	Iona
1	Philip

The SQL query then could show only the duplicate tuples:

SSN	FName
1	Philip
2	Stella

Solution. In order to get rid of the duplicates it means that each employee appears only once in the relation. Hence, if we are about to group the employees with the SSN attribute, then we expect to get some groups having more than one tuple. This is happening, since the SSN is not declared as PK. By grouping w.r.t. SSN we obtain the following:

```
SELECT   E.SSN, E.Fname, COUNT(E.SSN)
FROM     EMPLOYEES E
GROUP BY E.SSN
HAVING   COUNT(E.SSN) > 1
```

Hence, we obtain only those employees who appear more than once in the relation.

Task 2: Provide a SQL query which counts the duplicate employee entries. Considering the example in Task 1, the SQL should return the value of 3, since there are 3 extra employee entries (2 extra for Philip and one extra for Stella).

Solution. So far, we know the duplicate tuples and their corresponding appearances. For instance, in the above-mentioned query, Philip appears 3 times and Stella twice. Hence, we have 2 duplicates for Philip and one for Stella (here, we refer only to the number of the extra tuples for each employee – this is what causes the problem). It seems that we need to sum up these appearances of each duplicate tuple, thus, engaging the above-mentioned query in an outer query. That is:

```
SELECT SUM(A.Appearances) FROM
   (SELECT   E.SSN, (COUNT(E.SSN)-1) AS Appearances
    FROM     EMPLOYEES E
    GROUP BY E.SSN
    HAVING   COUNT(E.SSN) > 1) AS A
```

Problem 3: Analytics Query

Consider the known EMPLOYEE(SSN, FName, Salary, …, DNO) relation, with SSN (now) being the PK.

Task: Provide an analytics SQL query which show the top-3 highest salary values **without** using the LIMIT operator 😊.

Solution. The idea here is to find how many different (distinct) salary values are greater than a given salary of an employee. If there are, e.g., 1, or 2, or 3 then it means that the salary value of that employee should be included in the top-3 list 😊 Otherwise, we do not need to retrieve this salary… Hence, let's design this query:

```
SELECT DISTINCT Salary
FROM   EMPLOYEE E
WHERE 3 >= (SELECT COUNT(DISTINCT U.Salary)
                         FROM     EMPLOYEE U
                         WHERE    E.Salary <= U.Salary)
ORDER BY E.Salary DESC;
```

For each employee, we get their salary value, i.e., E.Salary. Then, we count how many other distinct salary values are greater than the E.Salary. If there are 1 or 2 or 3 then it implies that the considered E.Salary value is at least one of the top-3 highest value!

Formative Assessment 4: Feedback

Overall Feedback: All of your provided solutions argued on the methods used. Most of the solutions agreed that the best file type organization is 'sorting'. The reasoning was mostly based on considering two directions: the statistics of the queries and the complexity of each file type organization. Some of you analysed the data w.r.t. the distribution of values! That was the key element in this analysis. Hence, the decision on the most appropriate field and file type organization relied on this analysis, which was amazing! I also received different levels of analysis: from the most abstract ones to very detailed ones experimenting with even different hash functions (adopting modifiers to uniform the distribution of the BONUS values) and/or calculating the exact number of block accesses. Each solution departed from **the key fact:** The EMPLOYEE relation is READ ONLY, i.e., no modification operations are allowed; only selection queries.

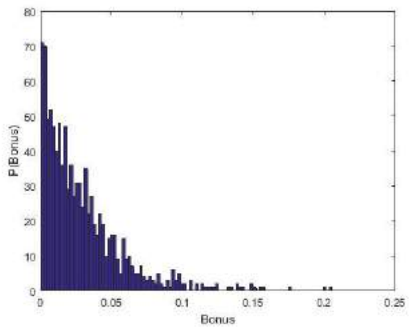
Overall Approach: I provide some insights on the considered problem taking also into consideration the solutions/ideas/approaches of your colleagues. Specifically, we need to exploit the two 'streams': *information coming* from the underlying data & *information coming* from the users' query patterns (statistics). These two streams will provide us the decision-making indicator: observing the 'live' system w.r.t. issued queries and observing the distributions of the underlying attributes. Hence, the strategy is: **Analyse Data**, **Observe the System Usage** (statistics from queries), **Reason about Candidate Solutions**.

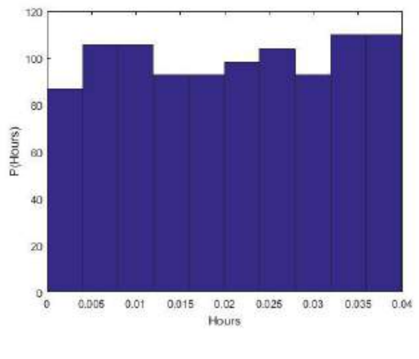
Let's put together Maths & Stats (Data Science), Database Physical Design & File Types (Systems & Engineering), and Algorithm Complexity (Computing Science).

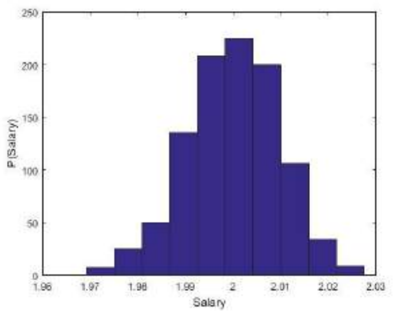
The strategy-steps are:

Step 1: Data Analysis. Let's analyse the data. The first thing of our analysis is to focus on the attributes used by the SQL **read-only queries**. Thus, the attribute ID is of no interest in analysing the data since no query involved the ID attribute. Part of the analysis is the histograms/distribution of the attribute values[1]. We desire to obtain uniform distributions, i.e., all values of an attribute appear with equal probability. However, in real-life scenarios this is not always the case ☹. Let's see what is happening in our scenario:

The following figures illustrate the distributions of the BONUS, SALARY and HOURS attributes.







[1] https://en.wikipedia.org/wiki/Histogram

As we can observe, BONUS values are following an exponential distribution[2], which is a bit **skewed**. This means that the low BONUS values are more probable to appear in the many tuple compared to high BONUS values. Hence, this might have a high impact on e.g., the Hash file organization. The HOURS values are more-or-less uniformly distributed over the tuples, thus each HOUR value is equi-probable. Finally, the Salary values are following e.g., a Gaussian/Normal like distribution[3] around a mean/average value and a fixed variance. This means that more salary values are 'gathered' around the mean salary value.

Let's start our thinking:

Step 2: Reasoning.
Thought #1. The distribution of the values might be a good decision making indicator for selecting a specific file type organization. Specifically, if we build a Hash File over the attribute HOURS then we exploit the fact of a quick search, e.g., O(1) for *any* hour value. Moreover, all the buckets will be of the same size because all HOURS values are *evenly* distributed over the buckets after observing their distribution. Hence, the expected cost of retrieval in terms of block accesses (even if we are dealing with overflown buckets) is **predictable**!

However, let's see how many times we are interested in searching w.r.t. the HOURS attribute. *Hmmm*, only 20% of the queries are involving the HOURS attribute. Hence, even it is predictable the expected cost, we exploit this 'convenience' only for the 20% of the queries. And, this is why:

The total expected cost in terms of block accesses is the summation of the expected cost per attribute times the probability of observing a query involving this attribute. In other words, if we have built a Hash File over HOURS then we would 'enjoy' this only for non-range queries, and only for queries involving equality conditions over the HOURS attributes. For the rest of the queries, we cannot avoid serial scan ☹. Let's take the best case scenario where ALL queries are involving equality conditions and not range conditions. In this case, the total expected cost is:

$$\mathbf{E[Cost\text{-}1] = 0.3*O(b) + 0.5*O(b) + 0.2*O(1) = 0.8*O(b) + 0.2*O(1)}$$

- where 0.3 is the probability of searching w.r.t. Salary time the cost for this, which is O(b): we cannot exploit the Hash File over the HOURS for searching w.r.t. Salary ☹
- where 0.5 is the probability of searching w.r.t. BONUS time the cost for this, which is O(b): we cannot exploit the Hash File over the HOURS for searching w.r.t. BONUS ☹
- where 0.2 is the probability of searching w.r.t. HOURS time the cost for this, which is O(1) in the best case, where what we are searching is in the main bucket ☺ !

Since, this is the best case, any other attempt of selecting another attribute to be hashed will result to a worse case than E[Cost-1], since the BONUS and Salary attributes are not uniformly distributed after analysing the data. Go now for a second thought.

Thought #2. Let us depart from the best-of-the-best case scenario in terms of Hashing, where we enjoy only 20% of O(1) retrieval, and focus on the Sequential file type. In this case, we 'pay' for sorting the file w.r.t. an

[2] https://en.wikipedia.org/wiki/Exponential_distribution
[3] https://en.wikipedia.org/wiki/Normal_distribution

attribute and then we expect to use binary search over the file to expedite the searching process. The sequential file type does not take into consideration the value distribution of the sorting attribute. However, we have to make sure that after sorting there are no clusters of tuples with the same sorting value. Hence, by adopting the binary search algorithm we need O(log(b)) to find a tuple and then terminate the searching process. If the sorting attribute has many values, then we might be needing to retrieve more than one blocks during the binary search, thus, more than O(log(b)) in the worst case. *Phew*, this is not happening here, since the number of the distinct values for each attribute is equal to the number of tuples in the relation. Hence, we are certain that the searching process is O(log(b)).

The second step now is: which will be the sorting attribute? The selection decision is driven by the users' queries, i.e., which attribute is relatively the most popular in the SQL selection queries. And, this is the BONUS. Hence, BONUS is a good candidate for being the sorting attribute. In this case, the expected total cost is:

$$\mathbf{E[Cost\text{-}2] = 0.5*O(\log(b)) + 0.3*O(b) + 0.2*O(b) = 0.5*O(\log(b)) + 0.5*O(b)}$$

That is because, we exploit only the 50% of the queries to speed-up the searching process by sorting the file w.r.t. BONUS attribute. The rest queries do not make use of the sorted file since they are involving the HOURS and the SALARY attributes, which are not the sorting attributes of the file ☹

An advantage of the sequential file is, of course, the good performance w.r.t. range queries. This is not the case in the Hash File, as discussed above. Now, if you assume that only equality SQL queries are issued over the relation, we could compare the E[Cost-1] (Hash File) and E[Cost-2] (Sequential File). Specifically, we obtain that E[Cost-1] < E[Cost-2] *iff* 0.8*O(b) + 0.2*O(1) < 0.5*O(log(b)) + 0.5*O(b) or *iff*:

$$\mathbf{0.3*O(b) + 0.2*O(1) < 0.5*O(\log(b))}$$

If we deal with the simple case (to remove the complexity operator O(.)), that is O(b) = b, O(1) = 1, and O(log(b)) = log(b) then we obtain that: 0.3*b + 0.2 < 0.5*log(b). We can re-write that as finding a value of b such that: log(b) > (3/5) b + 2/5 or $b > 2^{3b/5+2/5}$ which is never the case (due to exponential growth…). Thus, the selection of a Hash File instead of the selection of a Sequential File (over any attribute) is not a good candidate. And, this is happening only for equality SQL queries. For range queries, Hash File does not perform well. Of course, if we select HOURS to be sorting attribute, then, in this case the expected cost is: **0.8*O(b) + 0.2*O(log(b)).** In this case, the sequential file type organization is not so efficient since only 20% of queries are executed fast due to binary search and the rest queries are executed via a simple linear search. Hence, even if we have sorted the file, we do not exploit this sorting a lot...The selection of the attribute to be the sorting attribute matters in our decision-making process. And, this is driven by the statistics of the past issued queries.

Thought #3. Since we are dealing with only read-only queries, thus there are no insertions; the Heap File organization is inefficient for searching. It is known that the expected cost for the Heap File is:

$$\mathbf{E[Cost\text{-}3] = 0.5*O(b) + 0.3*O(b) + 0.2*O(b) = O(b)}$$

which is bigger than E[Cost-1] and E[Cost-2] and also bigger than a sequential file w.r.t. HOURS or Salary attributes. Thus, Heap File is not even a candidate!

Step 3: Lessons Learnt.

- Do analyse the data. That is, draw the histograms/distribution values to observe possible uniformity patterns.
- Moreover, if you are about to sort a file, check the number of the distinct values per attribute. This should be equal to the number of tuples (approximately speaking), thus, avoiding clusters of tuples with the same values. The target is to search at most O(log(b)) blocks, that is, when we find the tuple, we terminate the searching process (and do not need to load more blocks of tuples having the same value…).
- Furthermore, check if most of the queries involve the sorting attribute. This will provide insight on which attribute to select for sorting the file.
- Hash file is perfect for uniformly distributed attributes and if most of the queries involve only equality predicates!
- And, finally, Heap file is only used for appending tuples, e.g., log files.

Formative Assessment 5: Solution

Problem 1

Consider the following theorem for deciding whether to adopt a Primary Index or not over a sequential file, e.g., EMPLOYEE file w.r.t. an ordering key, e.g., SSN primary key.

Theorem: A Primary Index is created if and only if the file block can accommodate at least two primary index entries independently of the number of tuples in the relation.

Prove this theorem.

Solution of the Problem 1

Given that there exist b blocks, then a Binary Search cost is: **log(*b*),** since we load log(b) data blocks. After building a Primary File with $m < b$ blocks, then we require **log(*m*)** block reads to search within the index plus one data block access to retrieve the records from the file. Thus, **log(*m*) + 1** block accesses.

If r, B, R, and R’ are the number of tuples in the file, B is the block size, R is the record size and R’ is the index-entry size in bytes then, we adopt the Primary Index *iff*:

$\log(m) + 1 < \log(b) \leftrightarrow$
$\log(m) + \log(2) <= \log(b) \leftrightarrow$
$\log(2m) < \log(b) \leftrightarrow$
$\boldsymbol{m < b/2}$

And then we obtain that:

$\boldsymbol{m = b/\mathrm{bfri}}$ where bfri = B/R’ is the blocking factor of the index. Based on this, we obtain that since $\boldsymbol{m < b/2}$ then $\boldsymbol{b/\mathrm{bfri} < b/2}$ or **bfri > 2** or **B/R’ > 2** or **B > 2 R’**, i.e., the block size can store at least two index entries.

QED.

Problem 2

Consider the following SQL query:

```
SELECT DISTINCT(Salary) FROM EMPLOYEE
```

where the attribute Salary is not a key attribute in the relation EMPLOYEE.

Task 2.1: Propose an algorithm (provide a pseudo-code) for implementing the DISTINCT operator of the SQL query given that the relation EMPLOYEE can be hashed based on the attribute Salary. The hash function is assumed to be known.

Task 2.2: If the relation EMPLOYEE is stored in *b* file blocks, which is the cost (in block accesses) for your proposed implementation? Explain your answer.

Solution of the Problem 2

Since the attribute list in the SELECT clause of the SQL does not include a key of the relation Employee, then duplicated tuples might appear. To remove duplicates, we adopt the DISTINCT operator. Assume that we hash the relation Employee over the attribute Salary using given hash function *h*. The pseudo-code is as follows:

```
Algorithm
Repeat until the end of file
        Get the value of attribute Salary from the next tuple
        Hash the Salary value to get the memory-bucket h(Salary)
        Search in the memory-bucket h(Salary) and check if Salary already exists
        If Salary does not exist, then store Salary in the bucket and print out the value of Salary
End
```

Based on the proposed algorithm, the relation is scanned once, thus, ***b*** **block accesses**.

Problem 3

Consider a B+ Tree as a Secondary Index over the non-ordering key attribute: UK Post Code. An example of a UK post code is 'G12 8QQ'. The size of the UK Post Code attribute is 10 bytes and the file block size is 512 bytes. The relation contains only un-ordered UK post codes, i.e., no other attributes; it is a look-up relation. After inspecting the B+ Tree, the database designer found that:

- Each leaf node of the B+ Tree fits in **one** block.
- Each leaf node is **90% full** (on average).
- All the leaf nodes of the B+ Tree are stored in **s = 100 blocks**.
- The number of leaf-node pointers (next-tree pointers and data-pointers) is *p*=11 (leaf-node order).

The designer needs to determine the **maximum tree level *t*** to decide where to use this B+ Tree or just a linear search on the file for any random selection query over this relation. Which is this maximum tree level?

Solution of Problem 3

We should pay attention to the fact that: **Each leaf node fits in one block.** However, on overage each leaf node is 90% full, that is contains **0.9*(p-1) values**, given that p is the number of pointers (order of the leaf node). Hence, the number of values in the B+ Tree leaf level is the same with the number of values/post-codes in the relation. Said that, we have:

0.9*(p-1)*s = 0.9*10*100 = 900 values.

The file has 900 values. Each value is 10 bytes, thus, the blocking factor of the data file is:

floor(512/10) = 51 values per block.

Since we have 900 values then the file has **ceil(900/51) = 18 blocks.**

- It is strange since the B+ Tree requires ONLY 100 blocks for the leaf nodes and the FILE is only 18 blocks!

- That is more meta-data than data, and these meta data are only for searching!

The expected cost over a linear search on the file is: **9 blocks.** The expected cost over the B+ Tree is **t + 1 blocks**, given that it is of level t. To be efficient we require: t+1 < 9 or **t < 8**.

Hence, the maximum level of the B+ Tree should be **t = 7 (in the worst case)** in order to be more efficient when searching for a post code. Otherwise, it is more efficient to proceed with a linear scan on the file!