## Problem 1 Unix Command Line Tools (9 credits)

The following questions cover *Unix command line tools*. You may use gnu coreutils and other programs that are commonly contained in Linux distributions.

0 1 2 3

a)*
It is part of the Unix philosophy to use programs which do only one thing, but do it well. What are the advantages of this approach, as opposed to using one monolithic program? Explain in *at most three sentences*.

- Modular, thus flexible.
- Easy to extend.
- Easy to adapt to new use cases.

1p per sensible argument.

0 1 2

b)*
You are given a text file `access.log`. Write a Bash command to count all lines that contain the pattern `'abbbbbccc'` at least once.

```
grep 'abbbbbccc' access.log | wc -l
```

- 1p: `grep`
- 1p: `wc -l`

0 1 2 3 4

c)*
You are given a file `results.txt` of the alpine skiing results which contains: race id, race completion time, athlete id, athlete name in this order. These are some entries:

```
1 85  37 Corinne Suter
1 86  240 Mikaela Shiffrin
2 90  39 Petra Vlhova
5 99  45 Ester Ledecka
5 101 240 Mikaela Shiffrin
```

Write a Bash command to determine the overall skiing time of Mikaela Shiffrin *using her athlete id (not name)*.

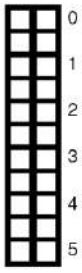
```
awk 'BEGIN{sum=0}{if($3==240) sum=sum+$2} END{print sum}' results.txt
```

- 1p: correct initialization of sum (BEGIN or leaving it out for auto initialization)
- 1p: print of sum in END statement
- 1p: if or filtering with grep
- 1p: adding up the correct race times

## Problem 2 Performance Spectrum (10 credits)

The following questions test your understanding of the performance spectrum of modern computers.

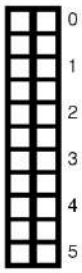
a)*
Program P generates thumbnails of 158 images. The images are 0.3 MB in size and all in different positions on one rotating disk (HDD). Each thumbnail is 52 KB. Estimate how long it takes to generate the thumbnails and calculate the total time in seconds. Show your calculation steps.




158 * (0.01s seektime latency + 0.3/200MB/s + 52/1000/100MB/s) = 1.8992 seconds

- 1p: realizing that this task is disk bound (not CPU bound)
- 2p: using seektime (needed due to random accesses on disk)
- 1p: for reading the images
- 1p: for writing the thumbnails

b)*
Program Q computes a join of two tables.
Program P (from the first task) and Q are now **both altered** to work well on a cluster of machines. Which one benefits more from the usage of a large amount of machines? What are the performance limitations of both tasks? Explain in at most five sentences.




- P is embarrassingly parallel, therefore easy to distribute data and computation. (2p)
- Q has many data dependencies. Communication must occur during computation. (2p)
- Communication slows down implementation. Depending on amount of communication. (1p)
- A correct answer must discuss the amount of communication between nodes and how it affects scaling.

# WS20/21 Endterm Exam

Explain in detail why the decorrelated variant can be executed more efficiently than the correlated query (assuming that the database system does not automatically decorrelate the query).

- Correlated query executed for every tuple in `transactions`.
- In the decorrelated query, only one group by operation is necessary to retrieve comparison values.
- Therefore, the correlated variant executes one scan of transfers for every tuple in transfers.
- The decorrelated version performs only one scan.
- Decorrelation thus reduces runtime complexity from $O(n^2)$ to $o(n^2)$.

- 2p: correlated subquery is performed for each tuple again
- 2p: decorrelated query is precomputed/only computed once (1p) and then used for each tuple (1p)
- 1p: correlated query variant thus has quadratic runtime/subquery scan for each tuple

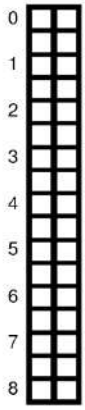
d)*
Next, the MF and FDE-Bank want to find suspicious accounts analyzing the account balances. Therefore, they want to find all accounts whose account balance was (at least once) negative. Write a SQL query which produces all account numbers of accounts which ever had a negative balance. Assume that initially all account balances are 0. You may use window functions for this.

```
with transactionsByAccount (account, amount, time) as (
      select account_to as account, amount, timestamp as time
      from transactions
  union all
      select account_from as account, -1 * amount as amount,
             timestamp as time
      from transactions
)
balance (account, running_sum) as(
  select account,
                sum(amount) over(partition by account
                                 order by time asc
                                 -- this is the default range
                                 -- thus not strictly necessary in the solution
                                 range between unbounded preceding
                                           and current row)
  from transactionsByAccount
)
select distinct account
from balance
where running_sum < 0;
```

- 2p: selecting account_from & account_to to cover both cases
- 2p: correct use of union all to unify positive and negative balances
- 1p: negative amount
- 1p: partition by account
- 1p: order by time asc
- 1p: distinct account
- 1p: running sum < 0

Revision of table `transactions` to reduce scrolling:

```
CREATE TABLE transactions(
    account_from    bigint,
    account_to      bigint,
    amount          decimal(20,2),
    timestamp       bigint
    );
```



e)*
The MF suspects that there is illegal money on account `7363` and that it was moved to other accounts. They now want to find all accounts to which (part of) the illegal money may have been moved.

1. Write a SQL query which finds these accounts, that is all accounts which received money from `7363` or received money from those accounts which received from `7363` and so on.
2. Also, explain why your query terminates in the presence of cycles (e.g., A sends money to B, B sends money to C, C sends money to A).

```
with recursive tainted (account) as (
  select 7363
union
  select account_to
  from tainted, transactions
  where tainted.account = transactions.account_from
)
select *
from tainted;
```

Duplicates are removed from working table, at most all accounts can go into the working table and the number of accounts is finite, thus the query eventually terminates.

- 1p: non-recursive part
- 2p: union
- 1p: from tainted-transactions
- 1p: join condition
- 1p: select * from tainted

Second part:

- 1p: Duplicates are removed from working table
- 1p: At most all accounts can go into the working table and the number of accounts is finite. Thus, the query eventually terminates after all accounts were seen once.

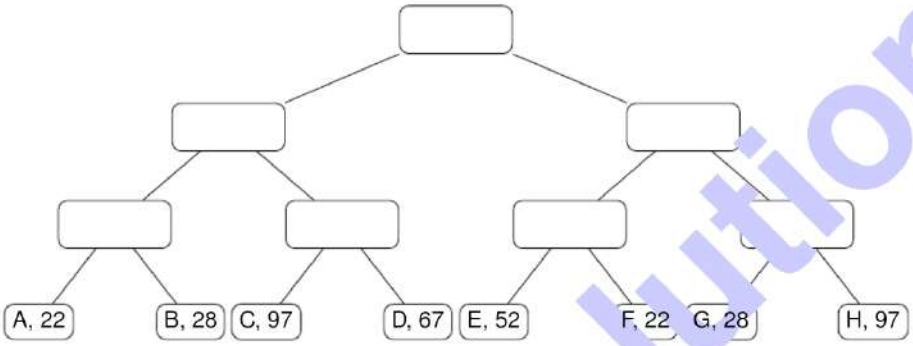
# WS20/21 Endterm Exam

## Problem 4 Segment Trees (13 credits)

The following questions test your knowledge of segment trees.

The FDE-Bank wants to optimize their window function queries and thus creates segment trees for the entries in `transactions`.

a)*
1) First, complete the segment tree for `min` aggregation. Write directly into the empty nodes, the PDF contains editable text boxes.
2) Then, calculate the `min` for the window [A; G] using the segment tree and explain your approach in at most 2 sentences.





Calculating the min works by choosing the topmost possible node, in this case: level 2 node 1, level 3 node 3, level 4 node 7.

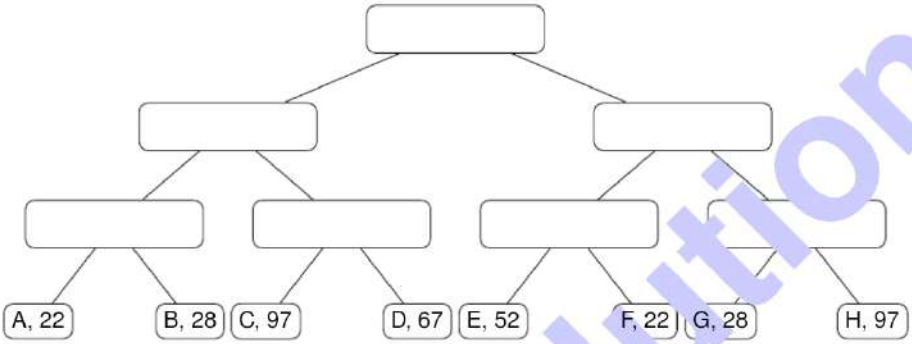
- 3.5p: for the correct tree: (0.5p per node)
- 0.5p: for the correct minimum
- 2p: for correct explanation of the window evaluation.

# WS20/21 Endterm Exam

0
1
2
3
4
5
6
7

b)*
The FDE-Bank also wants to compute running averages, e.g., to analyze customers' transaction volumes over time. Unfortunately, storing the `avg` of the child nodes, like for `sum` or `min`, is not possible due to numeric errors for floating numbers.

1) Thus, your task is to build a segment tree for `avg` without storing the averages itself in the nodes. You are allowed to store more than one value per node. Write directly into the empty nodes, the PDF contains editable text boxes.
2) Explain in at most 5 sentences how your solution works, e.g., with an example, and why it produces the correct average values.




Instead of `avg`, we store (`sum`, `count`) in each node and then calculate the average with it.
This produces the correct results, because we perform integer operations to add up all sums and counts and then divide the total sum by the total count. Thus, we have no numeric errors.

- 3.5p: for the correct tree sums: (0.5p per node)
- 1p: for all counts being correct.
- 2.5p: for correct explanation.

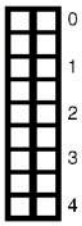
## Problem 5 Distributed Systems (12 credits)

The following question tests your knowledge of distributed systems.

The FunnyCatz company wants to enter the video streaming market, therefore, it needs a system that supports the following actions:
Users can upload a funny cat video and get a short generated URL to access and share it. Once the video was uploaded for one URL, it cannot be updated by the user.

Design a system like FunnyCatz with your knowledge from the lecture. Make sure that your system design can handle the following situations. In your answer, describe very shortly how your system works. Then describe for each situation how your design handles it:

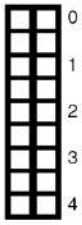
a)* Initially, your service is not very popular. At this point you do not employ many computers to run it. Eventually, though, it becomes very popular and the number of users grows rapidly. How do you make sure that your system can service a growing user base?




Store data in a distributed system/hash table, e.g., CHORD, to build a system that scales out. That will be able to add a number of computers to handle larger workloads and add new nodes without redistributing all data. The data must be distributed since single machines won't be able to store all videos.
Other approaches are also accepted if they are feasible and effective in practice. No credits for fuzzy explanations.

- 1p: systems scales out
- 1p: distribution of data over all nodes (not replicas on all nodes)
- 1p: adding new machines possible (no redistribution of all data)
- 1p: explanation how the system works

b)* Some cat videos are very popular (orders of magnitude more than many others). How do you make sure that your system can reply to all these requests quickly?

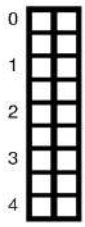



Have some group of servers which handle the CHORD DHT. Have another group of web servers which handle user requests. The number of web servers can be scaled with the number of users (replicas) and apply geographically distributed server infrastructure for locality (physical latency). User requests are equally distributed to the web servers (load balancing). To handle popular URLs, each server maintains a cache of least recently requested URLs (caching).
Other approaches are also accepted if they are feasible and effective in practice. No credits for fuzzy explanations.

- 1p: replications
- 1p: load balancing via nodes
- 1p: caching
- 1p: geographical distribution

Explanation how the system works required.

# WS20/21 Endterm Exam




c)* Some cat videos are not watched anymore but not deleted by the user. How do you make sure these require only a very small amount of cheap resources?

On the CHORD machines: Store the hashtable data in a data structure which is backed by hard drives (/cheap machines). The simplest solution is to just mmap your data structure to a file. That way pages with seldom used pages will be evicted to disk. More advanced data structures like finger trees etc., give extra points here. Reduce the replicas to a minimum value and also apply compression.
Further solutions should:

- Reduce the replicas not watched videos
- Move them to a cheaper memory hierarchy/node
- Apply compression to reduce the memory footprint

Other approaches are also accepted if they are feasible and effective in practice. No credits for fuzzy explanations.

- 1p: evict to disk/cheap machine
- 1p: reduce replicas to minimum
- 1p: compression
- 1p: analyzation strategy for less used values/explanation why slower access time is ok

Explanation how the system works required.

## Problem 6 MapReduce (11 credits)

The following questions test your command of the Map-Reduce concept.

The Allgau wind-park has hundreds of wind turbines. For each turbine they log how long the turbine produces power and store this data in the data set T. Each row in the data set T contains the following columns:

- *TurbineID* (unique ID of each turbine: integer)
- *TimeStart* (timestamp where the turbine starts producing energy: integer)
- *TimeEnd* (timestamp where the turbine stops producing energy: integer)
- *District* (the post code of the turbine's district: integer)
- *Day* (day of the recorded log: date)

The timestamps in *TimeStart* and *TimeEnd* are stored as unix timestamps. Thus, they are integer values and contain the elapsed seconds since January 1st, 1970. To determine a duration in seconds you can just subtract them.

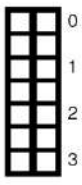
Write Map-Reduce pseudo code to answer the the following questions. Use a main function to call all map and reduce functions.
You may use the following functions:

- `now()`: returns the current timestamp.
- `count()`: counts the number of input values.
- `max()`: returns the maximum of its input values.
- `min()`: returns the minimum of its input values.
- `sum()`: produces the sum of its input values.
- `asSet()`: returns the set of its input values.

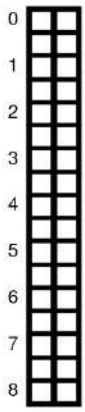
a)*
Determine the total power production time of each turbine in seconds.

```
def map(t):
        emit(t.TurbineID, t.TimeEnd - t.TimeStart)

def reduce(t_id, t_duration):
        emit(t_id, sum(t_duration))

def main():
        m = mapAll(T, map)
        return reduceAll(m, reduce)
```

- 0.5p: for map emit + 0.5p for time difference
- 0.5p: for reduce emit + 0.5p for sum of t_duration
- 1p: for main



# WS20/21 Endterm Exam



b)*
Now, Allgau wind-park wants to find the most productive district for each day. Thus, determine for each day which district has the highest power production time. Ignore all log entries of wind gust, i.e., where the turbine runs for *less than* 4 minutes.
For each day, return the day, the district with the highest power production time, the district's power production time and the number of turbines that count into the power production time. In case of a tie, return all districts with the maximum amount of power production time for the particular day.

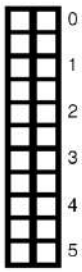
```
def mapFilterGusts(t):
        duration = (t.TimeEnd - t.TimeStart)
        if (duration/60 >= 4))
                emit((t.Day, t.District), (t.TurbineID, duration))

def reduceAgg((day, district), (tID, duration)):
        t_count = count(asSet(tID))
        d_sum = sum(duration)
        emit((day, district), (t_count, d_sum))

def mapDay((day, district), (t_count, d_sum)):
        emit(day, (district, t_count, d_sum))

def reduceDay(day, T): % T: (district, t_count, d_sum)
        sumList = ()
        for t in T:
                sumList. append(t.d_sum)
        maxTime = max(sumList)
        for t in T:
                if (t.d_sum = maxTime):
                        emit(day, (district, t_count, d_sum))

def main():
        mFiltered = mapAll(T, mapFilterGusts)
        rFiltered = reduceAll(mFiltered, reduceAgg)
        mMax = mapAll(rFiltered, mapDay)
        return reduceAll(mMax, reduceDay)
```

- mapFilterGusts: 0.5p duration, 0.5p conversion to minutes, 0.5p if (for filtering), 0.5p emit new key (day, district)
- reduceAgg: 0.5p t_count 0.5 as set, 0.5p d_sum, 0.5p emit (key (day, district))
- mapDay: 0.5p correct emit (change key to days)
- reduceDay: 0.5p loop 1 or addressing the correct attribute of the value list, 0.5p max, 0.5p for loop 2, 0.5p if, 0.5p emit (key (day), all other attributes as values)
- 1p for main

# WS20/21 Endterm Exam

## Problem 7 RDF Databases (5 credits)

The following question tests your knowledge of RDF databases.




We use an RDF database to store the results of the alpine skiing seasons. The data is provided as arbitrary JSON objects. Each object has an attribute "ID" which contains a unique identifier for the object.
To store objects, we use the predicates:

- attribute,
- name
- value

Thus for example the JSON-object

```
{
    "ID":38,
    "athlete":"Mikaela Shiffrin",
    "race time": 228
}
```

is represented using these triples:

| S | P | O |
|---|---|---|
| obj1 | attribute | attr1 |
| attr1 | name | "athlete" |
| attr1 | value | "Mikaela Shiffrin" |
| obj1 | attribute | attr2 |
| attr2 | name | "race time" |
| attr2 | value | 228 |
| obj1 | attribute | attr3 |
| attr3 | name | "ID" |
| attr3 | value | 38 |

State a SPARQL query that:

- finds the names of all athletes with at least one "race time" faster than 228 and
- determine the number of races in which they were faster than 228.

Your output for "Mikaela Shiffrin" might look like:

```
"Mikaela Shiffrin" 5
```

```
select ?name (count(?val) as ?count) --0.5 + 0.5p
  where {
    ?obj attribute ?attr1.          --0.5p
    ?attr1 name "athlete".          --0.5p
    ?attr1 value ?name.             --0.5p
    ?obj attribute ?attr2.          --0.5p
    ?attr2 name "race time".        --0.5p
    ?attr2 value ?val.                  --0.5p
    filter(?val < 228). --0.5p
 }
group by ?name                      --0.5p
```

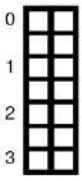
## Problem 3 SQL (30 credits)

The following questions test your knowledge of SQL.

The FDE-Bank stores all transactions of its clients in the table `transactions`:

```
CREATE TABLE transactions(
    account_from    bigint,
    account_to      bigint,
    amount          decimal(20,2),
    timestamp       bigint
    );
```

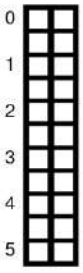
The Ministry of Finance (MF) informs the FDE-Bank that some clients are involved in illegal trading. Thus, the MF and FDE-Bank analyzes transactions in the following tasks to investigate the illegal trading.



a)*
At first, the bank wants to find all accounts that transfer high amounts of money. Write a SQL-query that outputs all accounts with at least 15 outgoing transactions higher than 6580$.

```
SELECT account_from
FROM transactions
WHERE amount > 6580
Group by account_from
Having count(*) >= 15;
```

- 1p: for correct select, from & where statements (not just the schema!)
- 1p: group by, 1p: having.



b)*
Now, the bank searches transactions that are above each client's average amount of transferred money. Unfortunately, the SQL-query is quite slow. Decorrelate the following SQL-query. You may assume that no `NULL` values occur in table `transactions` and the table is not empty.

```
SELECT *
FROM   transactions t1
WHERE  amount > (
         SELECT avg(amount)
         FROM transactions t2
         WHERE t1.account_from = t2.account_from
         )
```

```
SELECT *
FROM transactions t1,
     ( select avg(amount) a, t2.account_from acc
       from transactions t2
       group by t2.account_from) preagg
WHERE
    preagg.acc = t1.account_from and
    t1.amount > preagg.a
```

- 1p: avg(amount)
- 1p: t2.account_from
- 1p: group by t2.account_from
- 1p: preagg.acc = t1.account_from and
- 1p: t1.amount > preagg.a