EXAM RETAKE WS 19/20:

1.a).

we want to read: 15000TB in 16 minutes
one local disk a 2TB 200 MB/s => 7500 local disk

how many GB per disk, so I finish in 16 min?
x / 200 MB/s < 16 * 60
192GB per disk => 78125 hard drives 977 racks => 33 cluster with 4.8 PB each

Alternative solution:
We need to scan 15.625 tera per second
therefore we need the scanning speed of 15625/0.2= 78125 harddisks
assuming 80 HDD per rack -> 977 racks, assuming 30 racks per cluster -> 33 clusters

b).

we have 10^5 key checks → latency per check is neglectable

6 levels with 256 pointers a 8 Byte:
uniform distribution: we need to check on average
1/256 * sum{from i = 1 to 256}(i) = 257/2 = 128.5
pointers before finding the correct one

it fits into ram → 20GB/s

6 x 128.5 x 8Bytes / 20GB/s = 0.3 microseconds

2.a).

+ This way we need less storage, since strings need commonly more than 128 bit storage
+ It can be physically faster to be looked up, since string comparisons are slower than integer comparisons.
- On the other side, we need a conversion of the hash back into “birthday” in the application layer which is not applicable since a hash can only be generated in one way, but nevertheless we need to know what 3457230467245 is, so we need at least a additional look up table, and therefore more complexity in the application, another downside is that with that it is no longer human readable.

- Collisions can still occur. And if they occur, one would prefer to distinguish keys that hash to the same value, e.g. using linear probing. Therefore, discarding keys fully is not desirable.

b.

From an old script:
1 - exp(-m^2/2n)
with m=10^12 and n=2^128 because of 128 Bit
1 - exp(-10^12 / 2 * 2^128) = 1.47 x 10^-15

With birthday-paradox:
1-((10^12)! x (2^128 choose 10^12)/ (2^(128 x 10^12)) = 1.47 x 10^-15

3.a)

```
with tmp as (
select id, sum(output) as total
from output
group by id)
, tmp2 as (
select *, rank() over (order by total desc) as top from tmp
)
select * from tmp2 where top < 2
```

```
with so(id, sumout) as (
  select distinct id, sum(output) over (partition by id) as sumout
  from output)

select *
from so
where sumout = (select max(sumout) from so)
```

b).

```
with tmp as (
select id, sum(output) as total
from output
group by id),
tmp2 as (
select id, sum(invested_power) as invested
from costs
group by id)
```

```
select t.id, t.total - c.invested as diff
from tmp t
join tmp2 c on t.id = c.id
order by  diff desc
limit 1
```

```
with osum(id, outsum) as (
  select distinct id, sum(output) over (partition by id)
  from output),
isum(id, insum) as(
  select distinct id, sum(input) over (partition by id)
  from costs)

select id, o1.outsum-i1.insum
from osum o1, isum i1
where o1.id = i1.id
and o1.outsum-i1.insum = (
  select max(o2.outsum-i2.insum)
  from osum o2, isum i2
  where o2.id = i2.id)
```

c).

```
select id, min(timestamp), inoutsum
from (
    select id, timestamp, sum(output) over (partition by id order by timestamp as) as
          inoutsum)
    from (
      select id, timestamp, -input as output
      from costs
      union all
      select id, timestamp, output
      from output)
and inoutsum >= 0
group by id
```

d)

For the starting timestamp of the last five low-yield-periods:

```
select id,timestamp as low_begin, lead(timestamp) over (partition by id) as low_end,
lowyield_nr
from (
  select id, timestamp, case when  lowyield = 1 and lagyield = 0 then 1 when lowyield = 1
        and leadyield = 0 then 0 end as begin_end, sum(lowyield) over (partition by id order
        by timestamp desc) as lowyield_nr
  from (
    select id, timestamp, lowyield, lag(lowyield,1,0) over (partition by id order by timestamp
        asc) as lagyield, lead(lowyield,1,0) over (partition by id order by timestamp asc) as
        leadyield
    from (
      select id, timestamp, case when output <= 250.0 then 1 else 0 end as lowyield
      from output))
  where lowyield = 1 and lagyield = 0 or lowyield = 1 and leadyield = 0)
where lowyield_nr <= 5
```

For starting and ending timestamp of last five low-yield-periods:

```
select *
from (
select id, timestamp as time_begin, lowyield_nr
from (
  select id, timestamp, sum(lowyield) over (partition by id order by timestamp desc) as
lowyield_nr
  from (
        select id, timestamp, lowyield, lag(lowyield,1,0) over (partition by id, order by
                timestamp asc) as lagyield
        from (
                select id, timestamp, case when output <= 250.0 then 1 else 0 end as
                        lowyield
                from output))
  where lowyield = 1 and lagyield = 0)
where lowyield_nr <= 5) t1,
(select id, timestamp as time_end, lowyield_nr
from (
  select id, timestamp, sum(lowyield) over (partition by id order by timestamp desc) as
        lowyield_nr
  from (
        select id, timestamp, lowyield, lead(lowyield,1,0) over (partition by id, order by
                timestamp asc) as leadyield
```

```
        from (
                select id, timestamp, case when output <= 250.0 then 1 else 0 end as
                        lowyield
                from output))
  where lowyield = 1 and leadyield = 0)
where lowyield_nr <= 5) t2
where t1.id = t2.id and t1.lowyield_nr = t2.lowyield_nr
```

Example for HyperDB:

```
with output(id, timestamp_, output) as (
  values (1,100000,500),
  (1,100001,240),
  (1,100002,300),
  (1,100003,200),
  (1,100004,150),
  (1,100005,300),
  (1,100006,100),
  (1,100007,320),
  (1,100008,130),
  (1,100009,260),
  (1,100010,220),
  (1,100011,310),
  (1,100012,111),
  (2,100000,500),
  (2,100003,240),
  (2,100005,300),
  (2,100007,200),
  (2,100009,150),
  (2,100011,300),
  (2,100013,100),
  (2,100015,320),
  (2,100017,130),
  (2,100019,260),
  (2,100021,220),
  (2,100023,310),
  (2,100025,111))

select t1.id, time_begin, time_end, t1.lowyield_nr
from (
select id, timestamp_ as time_begin, lowyield_nr
from (
  select id, timestamp_, sum(lowyield) over (partition by id order by timestamp_ desc) as
lowyield_nr
  from (
        select id, timestamp_, lowyield, lag(lowyield,1,0) over (partition by id order by
timestamp_ asc) as lagyield
        from (
```

```
        select id, timestamp_, case when output <= 250.0 then 1 else 0 end as lowyield
        from output))
  where lowyield = 1 and lagyield = 0)
where lowyield_nr <= 5) t1,
(select id, timestamp_ as time_end, lowyield_nr
from (
  select id, timestamp_, sum(lowyield) over (partition by id order by timestamp_ desc) as
lowyield_nr
  from (
        select id, timestamp_, lowyield, lead(lowyield,1,0) over (partition by id order by
timestamp_ asc) as leadyield
        from (
        select id, timestamp_, case when output <= 250.0 then 1 else 0 end as lowyield
        from output))
  where lowyield = 1 and leadyield = 0)
where lowyield_nr <= 5) t2
where t1.id = t2.id and t1.lowyield_nr = t2.lowyield_nr4.a.
```

With row-number-trick, assuming the timestamps are consecutive:

```
select id, min(timestamp_), max(timestamp_), max(timestamp_) - min (timestamp_) +1 as
length, period, rank() over (partition by id order by max(timestamp_) desc) as RANK
from (select id, timestamp_, timestamp_ - row_number() over (partition by id order by
timestamp_) as period
        from output
        where output < 250)
group by id, period
having max(timestamp_) - min(timestamp_) +1 >= 1
and rank() over (partition by id order by max(timestamp_) desc) < 6
order by id, RANK
```

With row-number-trick, assuming the timestamps are not consecutive:

```
WITH
tmp(id, time, y) as (
SELECT id, timestamp,  case when output<250 then 1 else  0 end
   FROM output),

tmp2(id, time, y, s) as(
SELECT id, time, y, sum(y) over(partition by id order by time range between unbounded
AND unbounded following)
FROM tmp),

tmp3(id, time ,y , s, d) as (
```

```
SELECT * , s-row_number()over (partition by id order by time)
FROM tmp2),

tmp4(id, time, y,s,d) as (
SELECT id, time, y, s, d,
FROM tmp3
WHERE y=1),

tmp5(d,r) as(
SELECT d, rank() over (order by desc)
FROM (SELECT distinct d from tmp4))

SELECT id, time, r
FROM tmp5 t5 join tmp4 t4 on t4.d=t5.d
WHERE r<=5
```

4.a).

```
select distinct To
from exchange
where To not in (select From from exchange)
```

b).

```
with recursive curr as (
select count(currency) as nr_curr
from (
        select distinct to as currency
        from exchange
        union
        select distinct from as currency
        from exchange),

 hull(From, To, dist, Rate) as (
 select From, To, 1, Rate
 from exchange
 union
 select h.From, e.To,  dist +1, h.Rate*e.Rate,
 from hull h, exchange e, curr
 where h.To = e.From and dist < curr.nr_curr
```

select From (1) -- 1,5 → (2) -- 1,5 → (3) -- 1,5 → (1) = (1, 1, 3.3)

```
from hull                                   (1) -- 0.5 → (4) -- 0.5 → (1) = (1, 1, 0.25)
where From = To
and Rate > 1
```

Example for HyperDB with different approach for the rate (diameter of the graph = 4):

```
with exchange (from_cur, to_cur, rate) as (
  values
  ('EUR', 'GBP', 0.91),
    ('GBP', 'USD', 1.24),
    ('USD', 'EUR', 0.89),
      ('USD', 'GBP', 1.89),
        ('USD', 'TINGELTANGEL', 1.89),
          ('EUR', 'BLUBB', 2.89)
  )




with recursive exchange (from_cur, to_cur, rate) as (
  values
  ('EUR', 'GBP', 0.91),
           ('GBP', 'USD', 1.24),
           ('USD', 'EUR', 0.89),
           ('USD', 'GBP', 1.89),
           ('USD', 'TINGELTANGEL', 1.89),
           ('EUR', 'BLUBB', 2.89)
  ),
 tmp (from_cur, to_cur, dist, diff) as (
select from_cur, to_cur, 1, (100 * rate) as diff from exchange
union
select t.from_cur, e.to_cur,dist +1, (diff * rate::float)
from tmp t, exchange e where e.from_cur = t.to_cur and dist < 5
)

select distinct from_cur, to_cur , diff from tmp where diff > 100 and from_cur = to_cur
```

5.a).

```
mapCont(ContainerID,Port,RecordType,timestamp)
  emit(ContainerID,timestamp)

reduceCont(ContainerID, [timestamp])
  bool used = false
  for tstamp in [timestamp]:
          if(tstamp > timestamp_now-(<one year>)):
          used = true
          break
  emit(ContainerID,used)

main(S)
  c = mapAll(mapCont(S))
  usedlist = reduceAll(reduceCont(c))
  return usedlist.filter((ID,used) -> !used)
```

Situation:
[(ContainerID, Port, RecordType, Timestamp)]

```
map((ContainerID, Port, RecordType, Timestamp)){
    emit(extract year from Timestamp, ContainerID)
}
reduce(year, [ContainerID]){
    if year is lastYear{
        for c in ContainerIDs{
            emit("result", c)
        }
    }else{
    emit("all", c)
}

}

main(){
    results = reduce(map(allContainers))
    used = results["result"].distinct()
    all = results["all"].distinct()
    return all - used
}
```

b).

```
map_ports((ContainerID, Port, RecordType, Timestamp)){
    emit(1, Port)
}
reduce_ports(key, list = [Ports]){
    list = list.distinct()
    emit("result", length(list))
}
map((ContainerID, Port, RecordType, Timestamp)){
    emit(ContainerID, Port)
}
reduce(ContainerID, list = [Port]){
    emit((ContainerID, list.distinct()))
}
main(){
    result_ports = reduce_ports(map_ports([(ContainerID, Port, RecordType, Timestamp)]))
    count_of_ports = result_ports["result"]

    result_container = reduce(map([(ContainerID, Port, RecordType, Timestamp)]))
    for result in result_container{
        if length(result.second) == count_of_ports{
            print(result.first)
        }
    }
}
```

```
mapCount(S)
  emit('count',Port)

getCount(k,ports : list)
  emit(k,unique(ports).count()

mapPorts(S)
  emit(ContainerID,Port)

reducePorts(ContainerID,ports : list)
  emit(ContainerID, unique(ports).count()

main(S)
  p = mapAll(mapCount(S))
  numPorts = reduceAll(getCount(p))['count']
  c = mapAll(mapPorts(S))
  ContPorts = reduceAll(reducePorts(c))
  return ContPorts.filter((ID,num) -> num = numPorts)
```

c).

```
mapRec(S)
  emit(Port,RecordType)

reduceRec(Port,records : list)
  emit(Port,records.filter(record -> record = Departure).count() -
          records.filter(record -> record = Arrival).count())

main(S)
  r = mapAll(mapRec(S))
  numrec = reduceAll(reduceRec(r))
  return numrec.filter((Port,num) -> num > 0)
```

```
mapRec(R):
          if RecordType==Departure:
                    emit(Port,1)
          else:
                    emit(Port,-1)

reduceRec(Port, list):
          if sum(list)>0:
                    emit(“result”,Port)

main(R):
          map = mapAll(mapRec,R)
          return reduceAll(reduceRec, map)[“result”]
```

```
map_send(containers){
   for each container: {
      if container.RecordType == "Departure"{
         emit("Departure", (port, containerID))
      }else{
         emit("Arrival", (port, containerID))
      }
   }
}

reduce_arrival("Arrival", [(port, containerID)]){
   for each c in  [(port, containerID)] {
      emit(port, containerID)
   }
```

```
}
reduce_arrival2(port, list = [containerID]){
emit(port, length(list))
}

reduce_departure("Departure", [(port, containerID)]){
   for each c in  [(port, containerID)] {
      emit(port, container)
   }
}

reduce_departure2(port, list = [containerID]){
   emit(port, length(list))
}

main(){
   port_arr, amount_a = reduce_arrival2(reduce_arrival(map_send(containers)))
   port_dep, amount_d = reduce_departure2(reduce_departure(map_send(containers)))

   for each join port_arr, port_dep as port{
   if(amount_d - amount_a > 0){
      print port
   }
}
```

6. a)

There is still no consensus on the answer of this question, the stuff below is just ideas:

We use the radix of the userid and users friend id to exploit the locality of the data so that we store user and users' friends, the status, friend status on the same machine (also the status) so we minimize the traffic exploiting the locality.

users:
(user1, name1) …,Status1)
(user2, name2)

friends:
(user1, user2)
(user2, user1)

status:
(user1, status_friend1, status_friend2, …)
(user2, status_friend1, status_friend2, …)

Machine1:
(user1, name1)
(user1, user2)
(user1, status_friend1, status_friend2, …)

Machine2:
(user2, name2)
(user2, user1)
(user2, status_friend1, status_friend2, …)

b).

OK, so to reduce random access we can use a clustered B+ tree which stores the friends user data next to each other. To reduce updates which are never read we can calculate a probability conditioned of the past activity of a user if he/she will read the status update, if the probability is low, we will not precompute the status update for him/her -> we do not waste effort. If he/she read the status update a service requests triggers an update and the data will be generated. We can also edit this to individual scopes, so we can calculate a probability if user a wants to see status of user b or we apply groups to users and assign them a probability of interest.

6 (a)
Cluster friends together in machines as good as possible.
When a user writes a status, for each of his friends, it is saved in a list of statuses (unique to that friend) on the same machine as the originating user.
All other users on the same machine with the same friends also save their statuses in these lists.
Because the friends are clustered, there will be many common friends and the number of user-status-lists per machine won't be too big.
For some of the "friend-heavy" users, the number of lists will be big, but most lists won't exist for too long on the machine, since there are probably only relatively few common friends with other users on the same machine and a status is only valid for 2h (there should be a cleanup-protocol, that regularly deletes outdated statuses). The amount of data is also the same as if the list of friends' statuses for each user would be on their own machine, the lists are just split on different machines this way.
When a user wants to see their friends statuses, only these lists have to be gathered, instead of every single friend, which makes the lookup time only depend on the number of machines and the data per status and not on the number of friends.
When a user updates their status, the status just has to be written to status-lists on the same machine, instead of the collected lists on the friends' machines. This way communication is minimized.

(b)
Use the data structure as explained in (a).
This way, the statuses are only communicated between machines, when they are actually needed and otherwise only ever reside on the machine of the originating user, where they will be deleted after roughly 2h. Also, as explained in (a), many friends already reside on the

same machine, which means that some statuses might never have to leave their original machine.
The lookup time for the friends' status list is a bit longer, than if all statuses would be immediately (after creation) communicated to the machine of the user who's trying to look up the list. But, the necessary computational resources are still much smaller than x, since not every friend has to be accessed, but instead every machine (that has a friend on it, which should be known to the up-looking user) is only accessed once.

7.a).

dbo:mountainbike p:have dbo:gear_shifters .
dbo:mountainbike p:have dbo:brakes .
dbo:mountainbike p:have dbo:handle_bars .
dbo:mountainbike p:is dbo:bikes .
dbo:bikes p:is dbo:vehicle .
dbo:boat p:is dbo:vehicle .
dbo:skateboard p:is dbo:vehicle .

S:	P:	O:
Mountainbike	has	gear shifters
Mountainbike	has	breaks
Mountainbike	has	handle bars
Mountainbike	is	bike
bike	is	vehicle
boat	is	vehicle
skateboard	is	vehicle

b).

```
PREFIX dbo : <ourdata.de/dbo>
select ?o
where {
?o p:is+ dbo:vehicle .
}
```

EXAM WS 19/20

4.a).

```
with recursive hull(from_acc, to_acc, timestamp) as
select from_acc, to_acc, timestamp
from transfer
union
select h.from_acc, t.to_acc
from hull h, transfer t
where h.timestamp < t.timestamp
and h.to_acc = t.from_acc

select * from hull
where from_acc = 7553
```

terminates, because infinite cycles are not possible due to the timestamp constraint.

b).

```
with alltransfer as
select from_acc, to_acc, amount, timestamp
from transfer
union
select to_acc as from_acc, from_acc as to_acc, -amount as amount, timestamp
from transfer

select from_acc, timestamp, sum(amount) over (partition by from_acc order by timestamp
asc)
```

from alltransfer

5.a).

```
mapRide(taxi_ride)
   if(start_time > now-<three days>):
         emit(start_district, 1)
   if(end_time > now-<three days>):
         emit(end_district, 1)

reduceRide(district, c)
   emit(district, c.count())

main(taxi_ride)
   d = mapAll(mapRide(taxi_ride))
   num = reduceAll(reduceRide(d))
   return num
```

b).

```
mapRide(taxi_ride)
   day = timestamp -> date
   emit((start_district,day), -1)
   emit((end_district,day), 1)

reduceRide((district,day), c)
   emit(district, (day,c.sum()))

main(taxi_ride)
   d = mapAll(mapRide(taxi_ride))
   bal = reduceAll(reduceRide(d))
   return bal
```