Dr. Thorsten Papenbrock
Information Systems Group
Hasso Plattner Institute

February 21, 2020

Distributed Data Management – Exam

Winter Term 2019/2020 **Musterlösung:**

Musterlösung

Matriculation Number: ______________________________

0	1	2	3	4	5	6	7	8	9	Σ
1	7	8	15	6	5	3	7	6	12	70

Important Rules:

- The exam must be solved within **180 minutes** (14:00 – 17:00).
- Fill in your matriculation number (Matrikelnummer) above and **on every page**.
- Answers can be given in English or German.
- Any usage of external resources (such as scripts, prepared pages, electronic devices, or books) is not permitted.
- Make all **calculations transparent and reproducible**!
- Use the **free space under each task for your answers**. If you need more space, continue on the **back of the page**. Use the extra pages at the end of the exam only if necessary. **Provide a pointer to an extra page** if it should be considered for grading. The main purpose of the extra pages is for drafting.
- Please write clearly. **Do not use the color red or pencils**.
- If you have any questions, raise your hand.
- The exam consists of 34 pages including cover page and extra pages.
- For any multiple choice question, more or fewer than one answer might be correct.
- Good luck!

Task 0: Matriculation Number

Fill in your matriculation number (Matrikelnummer) on every page including the cover page and the extra pages (even if you don't use them).

Hint: Do it now! **1 point**

Task 1: Distributed Systems

1. Which of the following statements about distributed systems are true? Tick all true statements. **4 points**

   ☐ A distributed system is a group of independent compute nodes that communicate and collaborate to solve a common task.

   ☐ A distributed system is reliable only if it prevents faults, errors and failures.

   ☐ A distributed system can be scaled vertically by adding more nodes into the cluster.

   ☐ A distributed system that uses task-parallelism cannot guarantee ACID.

   Musterlösung:

   ☒ That is exactly the definition.
   ☐ A distributed system is reliable if it is fault-tolerant, i.e., if faults do not lead to failures. We cannot prevent faults.
   ☐ Adding more nodes is horizontal scaling.
   ☐ Task-parallelism might make ACID a bit more complicated, but there is no inherent contradiction. The same techniques (locking, snapshotting, logging etc.) also apply if certain subtasks are being executed in parallel.

   Grading: 1P for each correctly ticked or non-ticked field

2. Consider the following situation: You are given two algorithms, *mineFast* and *mineDistributed*, that both solve the frequent itemset mining problem. The algorithm *mineFast* is highly efficient, but non-distributable; the algorithm *mineDistributed*, on the contrary, is not quite as efficient, but distributable. While experimenting with different datasets on one machine, you found that *mineFast* is on average twice as fast as *mineDistributed*. You also figured out that 60% of *mineDistributed*'s runtime is distributable and the algorithm scales linearly with the number of available machines. Given that you have five equal machines, which algorithm is the faster approach? What is the expected runtime x of *mineDistributed* for some

input dataset on five machines given the runtime *y* of *mineFast* for that same input dataset on one machine? **3 points**

Musterlösung:

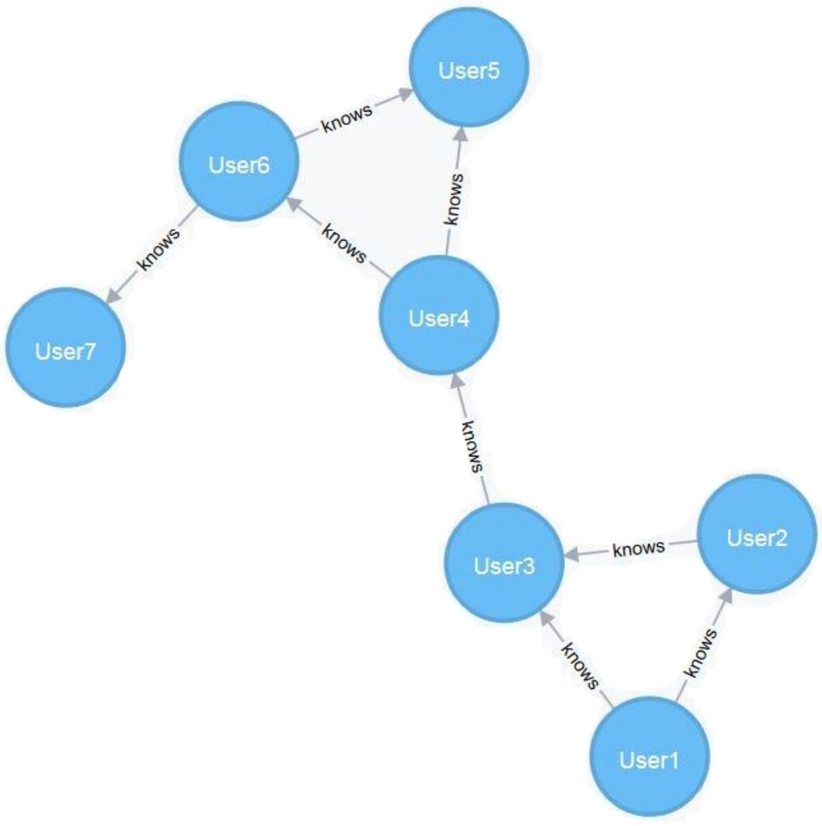
Calculate the runtime x of *mineDistributed* on 5 nodes given its runtime z on one node:
x = 40% * z + 60% * z / 5 =
z * (0.4 + 0.6 / 5) =
z * 0.52
We know that z is 2 times larger than *mineFast*'s runtime y, i.e., z = 2 * y:
x = (y * 2) * 0.52 = y * 1.04
This is *mineDistributed*'s runtime on 5 nodes relative to *mineFast*'s runtime.
Hence, the distribution does not pay off in this setting, because *mineDistributed* is 1.04 * *mineFast*'s runtime, which is larger than *mineFast*'s runtime.

Grading:

- 1P correct answer, i.e., *mineFast* is faster
- 1P correct calculation attempt or only finding that *mineDistributed*'s runtime on 5 nodes is x * 0.52
- 1P correct calculation of *mineDistributed*'s runtime is *mineFast*'s runtime * 1.04.

Task 2: Data Models and Query Languages

1. The graph below depicts a network of seven users. Some users know other users. Assume that a user can ask the users that she knows for their known users so that she then also knows these users – in that way, the `knows` edge becomes transitive. Iteratively calculating all transitive `knows` edges provides us with the transitive closure of the depicted graph that holds an explicit `knows` edge between a user and all other users that it can reach via direct or indirect `knows` edges. If we calculate the transitive closure with a *Bulk Synchronous Parallel* (BSP) transitive closure algorithm, how many steps would that algorithm need? Illustrate your answer or explain it in one or two sentences! **2 points**



Musterlösung:

Solution:

– Step 0 –
All users know nobody.

– Step 1 –
User 1: 2,3
User 2: 3
User 3: 4
User 4: 5,6
User 5: -
User 6: 7
User 7: -

– Step 2 –
User 1: 2,3,4
User 2: 3,4
User 3: 4,5,6
User 4: 5,6,7
User 5: -
User 6: 7
User 7: -

– Step 3 –
User 1: 2,3,4,5,6
User 2: 3,4,5,6
User 3: 4,5,6,7
User 4: 5,6,7
User 5: -
User 6: 7
User 7: -

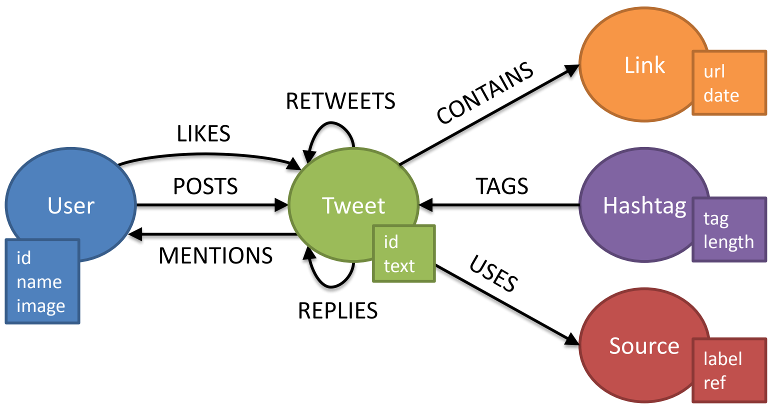
– Step 4 –
User 1: 2,3,4,5,6,7
User 2: 3,4,5,6,7
User 3: 4,5,6,7
User 4: 5,6,7
User 5: -
User 6: 7
User 7: -

Grading:

- 2P correct answer; 1P if the BSP approach seems to be understood, but some calculation is wrong; also 2P if the first step was skipped, because nodes may get their direct contacts via initialization

- Note that the graph topology does not change although we create transitive edges in processor states: messages always follow the edges in the original graph!

2. The following graph describes the schema of a database storing twitter users and their tweets. Each circle describes a possible label for node instances and each edge their possible relationships to other nodes. Assume that the graph/schema is not completely shown here and further nodes and edges might exist (open world assumption). The database was created in Neo4J and you have to use Cypher to query its content.



Write one Cypher query that adds a node with two edges to the depicted graph. The node we want to add has the new label `Website` and an attribute *content* with value “my_CV.pdf”. There should also be a new edge labeled `REFERENCES` that points from the `Link` node with *url* “www.usr42.com” to the new `Website` node and a new edge labeled `DESCRIBES` that points from the new `Website` node to the `User` node with *id* “42”. **3 points**

Musterlösung:

MATCH (user:User {id=42})
MATCH (link:Link {url=www.usr42.com})
CREATE (link)-[:REFERENCES]->(:Website {content=my_CV.pdf})
-[:DESCRIBES]->(user)

Grading:

- If bracket types are not correct or the syntax is a little bit wrong, we do not mark that as an error
- 1P Website node
- 1P DESCRIBES edge
- 1P REFERENCES edge

3. Still consider the graph of twitter users and tweets depicted above. We are now interested in tweets with the `Hashtag` *tag* “#ClimateChange” that have been re-tweeted at least 1000 times. Write a Cypher query that returns all these popular climate change `Tweet` nodes. **3 points**

Musterlösung:

MATCH (:Hashtag {tag="#ClimateChange"})-[:TAGS]->(tweet:Tweet)
<-[:RETWEETS*1000..]-(:Tweet))
RETURN tweet

Grading:

- If bracket types are not correct or the syntax is a little bit wrong, we do not mark that as an error
- 1P MATCHing the Hashtag
- 1P MATCHing to at least 1000 retweets
- 1P returning the correct tweet

Task 3: Actor Programming

1. Which of the following statements on the actor model and actor programming are *true*? Tick the true ones. **6 points**

   ☐ An *actor* is a special type of a thread that has its own private state, a mailbox and behavior.

   ☐ Actor *messages* need to be serializable, because the actor system needs to serialize and de-serialize every message that is sent from one actor to another.

   ☐ The specification of an *actor system* in Akka guarantees only at-most-once message delivery, because Akka uses fire-and-forget messaging and when firing messages with the UDP protocol, these messages might get lost.

   ☐ The *actor model* is very flexible: It allows the implementation of both task- and data-parallel applications, it supports pull- and push-based communication protocols and it lets us dynamically spawn and terminate actors at runtime.

   ☐ The *reaper pattern* defines a special actor, called the reaper, who sends out PoisonPill messages that terminate all actors in the actor system in a clean way.

   ☐ Using a *side channel* for large messages is advisable, because large messages may otherwise block important system messages, such as cluster heartbeats.

   Musterlösung:

   ☐ Actors are objects not threads. They are scheduled on threads and are therefore loosely coupled to threads – they do, in particular, not extend them.
   ☐ The actor system does not need to serialize *all* messages. Within one actor system, messages are send via reference.
   ☒
   ☒
   ☐ The reaper does not send any messages to any actor. It just waits for all actors to terminate and then terminates the actor system.
   ☒

   Grading: 1P for each correctly ticked or non-ticked field

2. The Actor model implementation in Akka ensures that every user defined actor is supervised by some other actor. If an actor encounters an error or crashes, its supervisor is notified so that the supervisor can analyze and handle the error. The supervisor then has four different options according to the actors' lifecycles to proceed the program. What are these four options? **2 points**

Musterlösung:

0.5P resume child actor
0.5P restart child actor
0.5P stop child actor
0.5P escalate

3. The following two actors should implement the *Network Time Protocol* (NTP) algorithm. Fill out the gaps in the code such that the `Client` actor frequently synchronizes its time according to the `Server` actor's reference time. **7 points**

   Hint: Use the space on this page for drafting. Then write your solution into the gaps of the already provided template code. Use the back of the template pages if you need more space.

   Musterlösung:

   Grading: Do not be strict about the syntax: missing ; or small mistakes are ok

   ERROR: The algorithm solution and its gab-version have a bug in the scheduled messaging part. The code (and expected solution) periodically sends the same TimeSyncRequest message, which is a problem, because every message needs to carry the current time, i.e., it needs to be a new message.

   ELEGANT SOLUTION ATTEMPT: Instead of passing an object to the scheduler, pass an anonymous function that creates a fresh version of the message with every call:
   `() -> new Server.TimeSyncRequest(this.getCurrentTime())`
   Unfortunately, schedulers do not accept functions and knowing how to formulate anonymous functions in Java is not required for this exam.

   WORKING SOLUTION ATTEMPT: Upon receiving the first TimeSyncResponse, the algorithm could stop the Cancellable and create a new one for the next TimeSyncRequest (or simply send the next TimeSyncRequest directly). The algorithm would work, but it had various issues. First, if the scheduler sends two TimeSyncRequest, the response to the second message would mess up the clients time. Second, the request intervals are not equi-distant, because the sync time, which as we know varies as bit, is added to the request intervals.

   ALGORITHM FIX: To fix the task, the scheduler needs to send a message to the client (and not the server). Upon receiving that static, periodic message, the client needs to create a new TimeSyncRequest message with the current time and send that message to the server. Because both the recipient of the scheduled message and the Receive object were given in this task, this solution was not programmable.

   GRADING: The algorithms needs to send a TimeSyncRequest with the current time in some way. This will grand full points for the gap. Not sending a TimeSyncRequest makes the NTP protocol incomplete so that the points cannot be given.

   Client:
   1P send TimeSyncRequest
   0.5P send thisCurrentTime() with TimeSyncRequest

0.5P time0 in TimeSyncResponse
0.5P time1 in TimeSyncResponse
0.5P time2 in TimeSyncResponse
1P calculation of newOffset as ((1-0)+(2-3))/2
0.5P update this.offset
0.5P graceful update e.g. use of this.offsetAdjustmentFactor

Server:
0.5P time0 in TimeSyncRequest
0.5P send time0 back
0.5P get time1 locally
0.5P get time2 locally (using the same time as time1 is ok)

Musterlösung:

Musterlösung:

```
package de.hpi.ddm.ntp;

import java.io.Serializable;
import java.util.concurrent.TimeUnit;
import akka.actor.AbstractLoggingActor;
import akka.actor.ActorRef;
import akka.actor.Cancellable;
import akka.actor.Props;
import scala.concurrent.duration.Duration;

public class Client extends AbstractLoggingActor {

    public static Props props(final ActorRef server) {
        return Props.create(Client.class, () -> new Client(server));
    }

    public Client(final ActorRef server) {
        this.timeSyncProcess = this.getContext().system().scheduler().schedule(
            Duration.create(0, TimeUnit.SECONDS),
            Duration.create(3, TimeUnit.SECONDS),
            server,
            -------------------------------------------------------------------------------------------
             |
             |
             ------------------------------------------------------------------------------------------
            this.getContext().dispatcher(), null);
    }

    @Override
    public void postStop() throws Exception {
        this.timeSyncProcess.cancel();
    }

    public static class TimeSyncResponse implements Serializable {
        private static final long serialVersionUID = 1208000708229308005L;
        -----------------------------------------------------------------------------------------------
         |
         |
         |
         |
         |
         |
         |
         |
         |
         |
         ----------------------------------------------------------------------------------------------
    }

    private final Cancellable timeSyncProcess;
    private long offset = 0;
    private float offsetAdjustmentFactor = 0.3f;

    private long getCurrentTime() {
        return System.currentTimeMillis() + this.offset;
    }

    @Override
    public Receive createReceive() {
        return receiveBuilder()
                .match(TimeSyncResponse.class, this::handle)
                .matchAny(object -> this.log().info("Unknown message: \"{}\"", object.toString()))
                .build();
    }

    private void handle(TimeSyncResponse message) {
        -----------------------------------------------------------------------------------------------
         |
         |
         |
         |
         |
         |
         |
         |
         |
         |
         ----------------------------------------------------------------------------------------------
    }
}
```

```java
package de.hpi.ddm.ntp;

import java.io.Serializable;
import java.util.concurrent.TimeUnit;
import akka.actor.AbstractLoggingActor;
import akka.actor.ActorRef;
import akka.actor.Cancellable;
import akka.actor.Props;
import scala.concurrent.duration.Duration;

public class Client extends AbstractLoggingActor {

    public static Props props(final ActorRef server) {
        return Props.create(Client.class, () -> new Client(server));
    }

    public Client(final ActorRef server) {
        this.timeSyncProcess = this.getContext().system().scheduler().schedule(
            Duration.create(0, TimeUnit.SECONDS),
            Duration.create(3, TimeUnit.SECONDS),
            server,
            new Server.TimeSyncRequest(this.getCurrentTime()),
            this.getContext().dispatcher(), null);
    }

    @Override
    public void postStop() throws Exception {
        this.timeSyncProcess.cancel();
    }

    public static class TimeSyncResponse implements Serializable {
        private static final long serialVersionUID = 1208000708229308005L;

        public long time0;
        public long time1;
        public long time2;

        public TimeSyncResponse(final long time0, final long time1, final long time2) {
            this.time0 = time0;
            this.time1 = time1;
            this.time2 = time2;
        }
    }

    private final Cancellable timeSyncProcess;
    private long offset = 0;
    private float offsetAdjustmentFactor = 0.3f;

    private long getCurrentTime() {
        return System.currentTimeMillis() + this.offset;
    }

    @Override
    public Receive createReceive() {
        return receiveBuilder()
                .match(TimeSyncResponse.class, this::handle)
                .matchAny(object -> this.log().info("Unknown message: \"{}\"", object.toString()))
                .build();
    }

    private void handle(TimeSyncResponse message) {
        final long time0 = message.time0;
        final long time1 = message.time1;
        final long time2 = message.time2;
        final long time3 = this.getCurrentTime();

        final long newOffset = ((time1 - time0) + (time2 - time3)) / 2;

        this.offset = this.offset + (long) Math.ceil(newOffset * this.offsetAdjustmentFactor);
    }
}
```

```
package de.hpi.ddm.ntp;

import java.io.Serializable;
import akka.actor.AbstractLoggingActor;
import akka.actor.Props;

public class Server extends AbstractLoggingActor {

    public static Props props() {
        return Props.create(Server.class);
    }

    public static class TimeSyncRequest implements Serializable {
        private static final long serialVersionUID = -3057724412864626584L;
        ------------------------------------------------------------------------------------------------
        |
        |
        |
        |
        |
        |
        |
        |
        |
        |
        ------------------------------------------------------------------------------------------------
    }

    private long getCurrentTime() {
        return System.currentTimeMillis();
    }

    @Override
    public Receive createReceive() {
        return receiveBuilder()
                .match(TimeSyncRequest.class, this::handle)
                .matchAny(object -> this.log().info("Unknown message: \"{}\"", object.toString()))
                .build();
    }

    protected void handle(TimeSyncRequest message) {
        ------------------------------------------------------------------------------------------------
        |
        |
        |
        |
        |
        |
        |
        |
        |
        |
        ------------------------------------------------------------------------------------------------
    }
}
```

```java
package de.hpi.ddm.ntp;

import java.io.Serializable;
import akka.actor.AbstractLoggingActor;
import akka.actor.Props;

public class Server extends AbstractLoggingActor {

    public static Props props() {
        return Props.create(Server.class);
    }

    public static class TimeSyncRequest implements Serializable {
        private static final long serialVersionUID = -3057724412864626584L;

        public long time0;

        public TimeSyncRequest(final long time0) {
            this.time0 = time0;
        }
    }

    private long getCurrentTime() {
        return System.currentTimeMillis();
    }

    @Override
    public Receive createReceive() {
        return receiveBuilder()
                .match(TimeSyncRequest.class, this::handle)
                .matchAny(object -> this.log().info("Unknown message: \"{}\"", object.toString()))
                .build();
    }

    protected void handle(TimeSyncRequest message) {
        final long time0 = message.time0;
        final long time1 = this.getCurrentTime();
        final long time2 = this.getCurrentTime();

        this.sender().tell(new Client.TimeSyncResponse(time0, time1, time2), this.self());
    }
}
```

Task 4: Replication and Partitioning

1. Assume you have a cluster of 12 801 nodes. The cluster runs a leaderless replicated, distributed database. Quorum reads and writes are used to ensure consistency and a gossip protocol ensures that all updates will eventually spread to all nodes in the cluster. For this task, assume also that the gossip protocol is perfect, i.e., newly written data is always gossiped to nodes that do not already know it.

   Which read (r) and write (w) values do we need to define in our quorum q(r,w) if writes should return as fast as possible and every successfully written value should reach all cluster nodes in not more than seven rounds of gossip? **4 points**

   Musterlösung:

   with 0 gossip rounds: 12801
   with 1 gossip rounds: 12801 / 2 = 6400.5 -> 6401
   with 2 gossip rounds: 12801 / 2 / 2 = 3200.5 -> 3201
   with 3 gossip rounds: 12801 / 2 / 2 / 2 = 1600.5 -> 1601
   with 4 gossip rounds: 12801 / 2 / 2 / 2 / 2 = 800.5 -> 801
   with 5 gossip rounds: 12801 / 2 / 2 / 2 / 2 / 2 = 400.5 -> 401
   with 6 gossip rounds: 12801 / 2 / 2 / 2 / 2 / 2 / 2 = 200.5 -> 201
   with 7 gossip rounds: 12801 / 2 / 2 / 2 / 2 / 2 / 2 / 2 = 100.5 -> 101

   in general:
   $w * 2^{rounds} >= 12801$
   $w >= 12801/2^{rounds} = 12801/2^7 = 12801/128 = 100 + 1/128$
   $w >= 101$

   consistency: r + w > n
   r > n - w = 12801 - 101 = 12700

   answer: q(12701,101)

   Grading:

   - 1P w = 101 correctly calculated
   - 1P r = 12701 correctly calculated
   - 1P gossip protocol correctly understood
   - 1P quorum consistency condition r + w > n

2. What is the disadvantage of the modulo operation when used for partitioning key spaces? What other approach prevents this issue? **2 points**

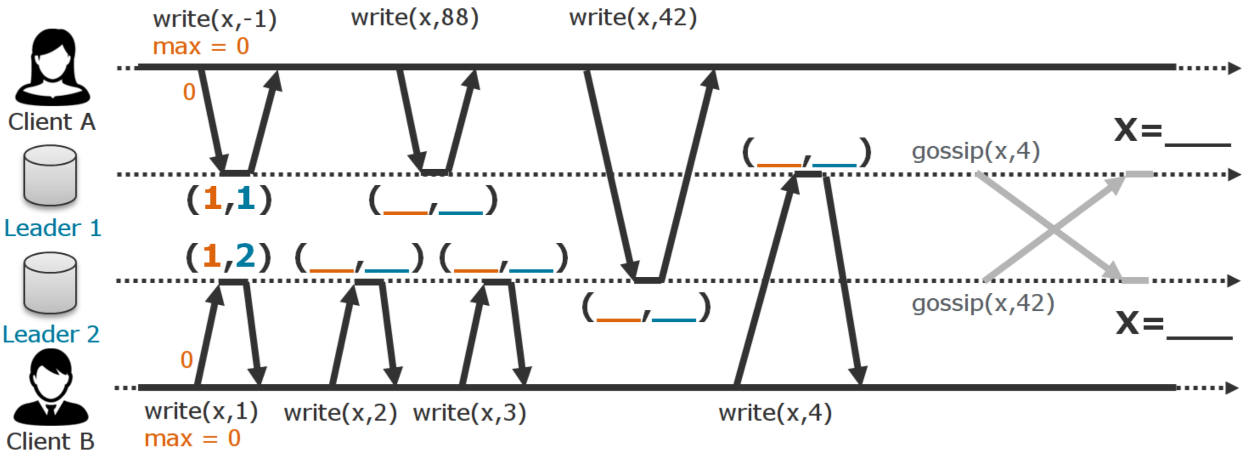
   Musterlösung:

   Grading:

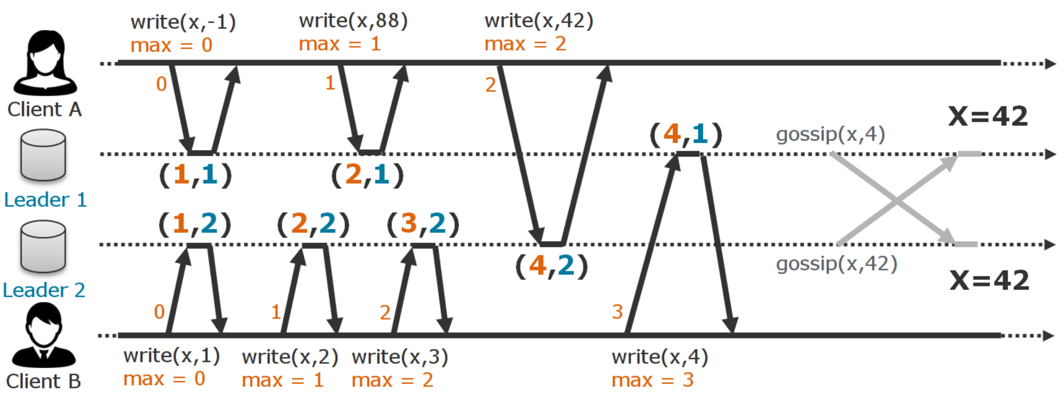
   - 1P modulo is not stable, i.e., it requires a lot of data shuffling if the number of partitions changes.
   - 1P alternative: consistent hashing (data stealing; fixed number of partitions per node)

Task 5: Consistency and Transactions

1. Given a database management system that implements *Lamport Timestamps* for causal write ordering. A lamport timestamp is a pair (c, i) with a write counter c and a node identifier i. Every write operation is associated with such a timestamp. We can use these timestamps for write ordering, because lamport timestamps are comparable: $(c, i) > (c', i')$ iff $(c > c') \vee (c = c' \wedge i > i')$. Add the lamport timestamps in the following example. What is the final value of the field x on leader 1 and on leader 2? **3 points**



Musterlösung:



Grading:

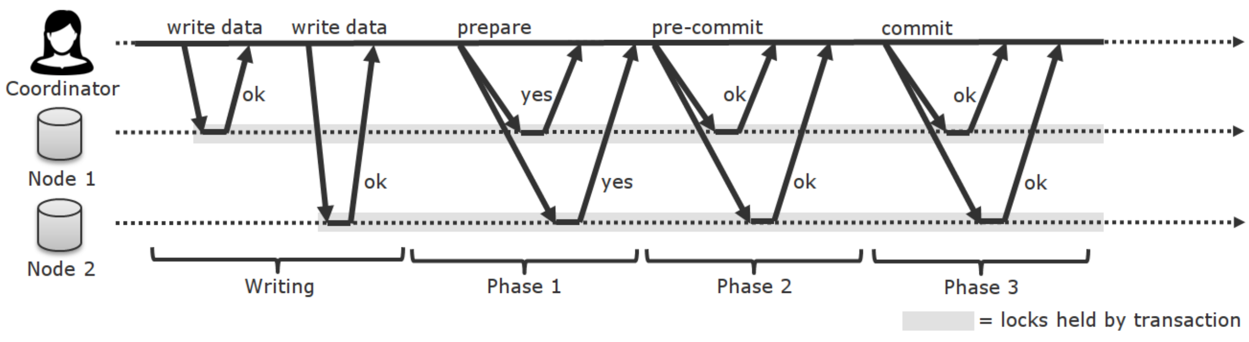
- 1P line is within phase 2
- 1P line is exactly within Node 1's pre-commit processing

2. The following figure depicts the *Three-Phase Commit* (3PC) protocol. If the coordinator in that protocol dies, another coordinator can take over and finish the transaction by either rolling it back or consistently committing it. Draw two vertical lines into the 3PC process:

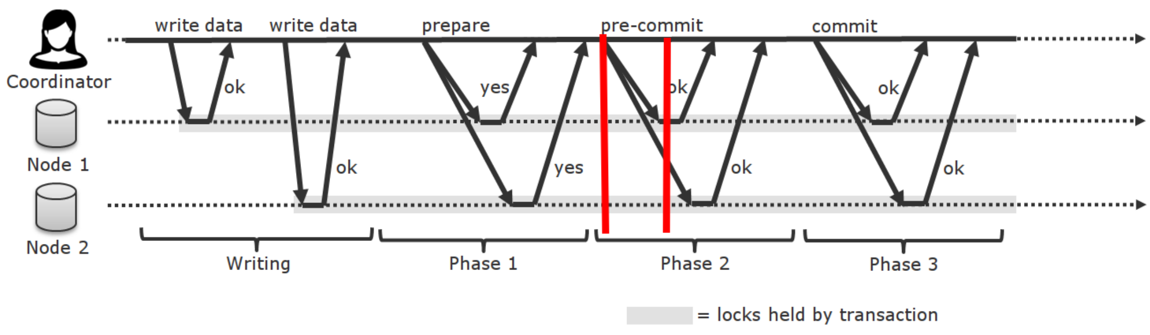
   I.) The first line should be at exactly that place up until which a new coordinator would definitely roll the transaction back, i.e., coordinator crashes left to that line would cause the transaction to be rolled back.

   II.) The second line should be at exactly that place from which onwards a new coordinator would definitely commit the transaction, i.e., coordinator crashes right to that line would cause the transaction to be committed.

   Coordinator crashed in between these two vertical lines could lead to both rollback and commit situations depending on what information has been lost. **2 points**



Musterlösung:



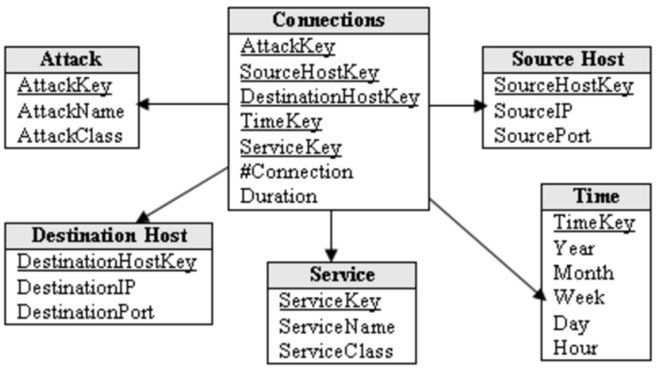
Grading:

- 1P first line on send of pre-commit

- 1P second line within Node 1's pre-commit processing, because if the coordinator dies before that *and* the pre-commit messages are lost, the new coordinator would still roll the transaction back.

Task 6: Data Warehousing

1. Consider the following data warehouse *star schema* and the join query on that schema. Rewrite the join query as a star join and explain why the star join makes sense especially in distributed data warehouses. **2 points**



$$((Connections \bowtie Attack) \bowtie Service) \bowtie Time$$

Musterlösung:

$((Connections \bowtie (Attack \times (Service \times Time))$

Or similar forms that use $\times$ between all dimension tables first and then $\bowtie$ with the facts table in the end.

Explanation: Star joins often produce smaller intermediate results on data warehouse star schemata and this is important especially for distributed systems, because intermediate results often have to be send (at least partially) over the network.

Grading:

- 1P correct star join
- 1P for explanation

2. What kind of schema mapping strategy is expressed by the following schemata and views? Write down the name of the strategy. **1 points**

Musterlösung:

Grading:

- 1P Global-as-View oder GaV

Global schema
Students(matrikel, first, last)
Registrations(matrikel, course)

Local schemata
ITSE_students(matrikel, first, last)
DE_students(matrikel, first, last)
DH_students(matrikel, first, last)
DS_students(matrikel, last, email)

HPI_registration(matrikel, course, lp)
PULS_registration(matrikel, course, accepted)

Views
```
V1: (SELECT * FROM ITSE_students, DE_students, DH_students)
      UNION
    (SELECT matrikel, null, last FROM DS_students);

V2: (SELECT matrikel, course FROM HPI_registration)
      UNION
    (SELECT matrikel, course FROM PULS_registration WHERE accepted=true);
```

Task 7: Distributed Query Optimization

1. Which of the following statements about distributed query optimization are true? Tick all true statements. **4 points**

☐ The query optimizer should, if possible, always try to push selection, projection and grouping operations to the nodes that hold the relevant data.

☐ The query optimizer does not need to push set operations, such as union, intersect and except, to data nodes, because evaluating set operations on data nodes does not reduce the network traffic.

☐ To minimize the network traffic for join operations, the query optimizer has to run a full reducer semi-join program.

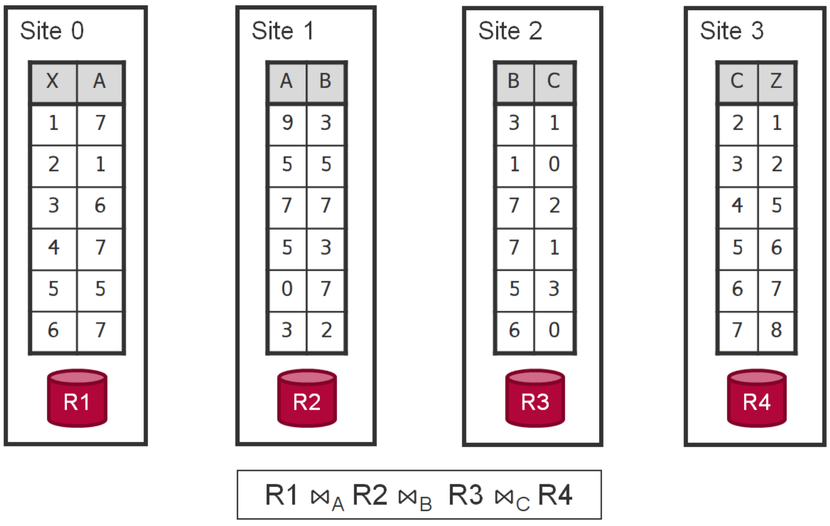
☐ Bloom filters can be used to approximate semi-joins: They might not remove all non-joining tuples, but all removed tuples are true non-joining tuples.

Musterlösung:

☒
☐ True for union, but pushing intersects and excepts to data nodes does reduce the network traffic, because only one side of these operations needs to be send to the other side's data node.
☐ A full reducer reduces all relations, such that we send a minimum amount of data if all relations have to be send to some third location that eventually executes the join. If we can join tuples on data nodes already, the full reduction increases the network traffic. The network traffic is also increased by the reduction process, if the final reduction is less than the traffic caused by the reduction process. Hence, the statement is wrong.
☒

Grading: 1P for each correctly ticked or non-ticked field

2. The following picture shows four relations that should be joined. Write down a *reducer program* of semi-joins that reduces R1. Then, write down the result of a full reducer program, i.e., the *reduced relations* R1, R2, R3, and R4. **3 points**



Site 0 (R1):

X	A
1	7
2	1
3	6
4	7
5	5
6	7

Site 1 (R2):

A	B
9	3
5	5
7	7
5	3
0	7
3	2

Site 2 (R3):

B	C
3	1
1	0
7	2
7	1
5	3
6	0

Site 3 (R4):

C	Z
2	1
3	2
4	5
5	6
6	7
7	8

$R1 \bowtie_A R2 \bowtie_B R3 \bowtie_C R4$

Musterlösung:

Reducer program for R1: $R1 \ltimes (R2 \ltimes (R3 \ltimes R4))$

Reduced relations:
$R1' = (1,7)(4,7)(5,5)(6,7)$
$R2' = (5,5)(7,7)$
$R3' = (7,2)(5,3)$
$R4' = (2,1)(3,2)$

Grading:

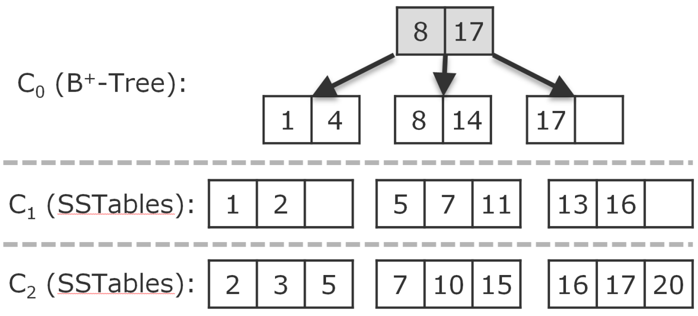
- 1P reducer program
- 0.5P for each correctly reduced relation

Task 8: SSTables and LSM Trees

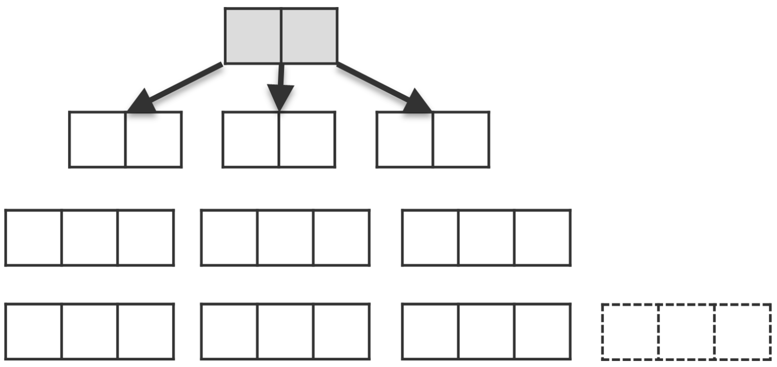
The following figures depict LSM tree instances with three levels, which are C_0, C_1, and C_2. In the LSM tree, we show only the keys and not their values. Assume that all values have the same size, so that a B^+-tree leaf can hold exactly up to two key-value pairs and each SSTable can hold exactly up to three key-value pairs. The maximum depth of the B^+-tree shall be two, which means that the depicted tree cannot grow any further. We now insert new elements into the LSM tree. Use the free space next to the figures for drafting and write your final results into the LSM tree templates below each instance figure.

1. Insert the key **18** into the depicted LSM tree instance: **1 points**

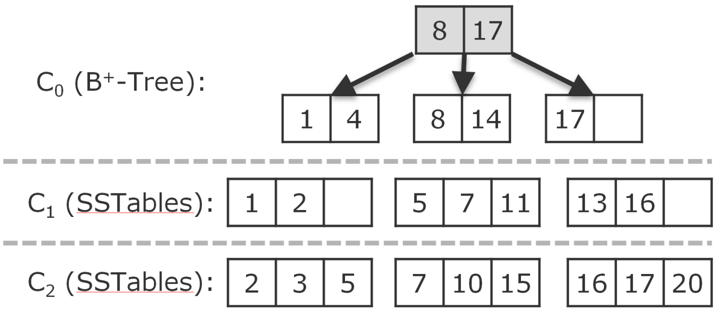
   LSM tree instance:

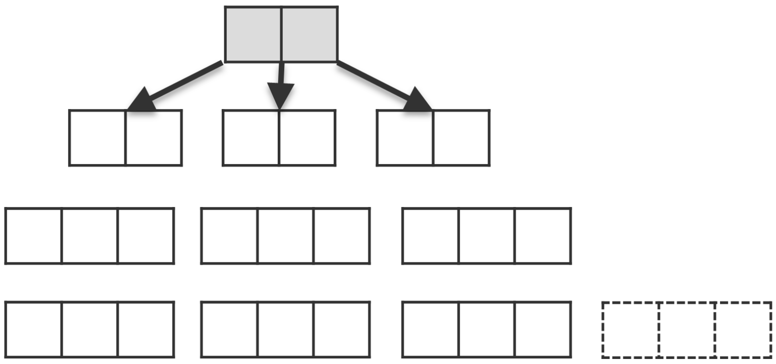


   LSM tree template:

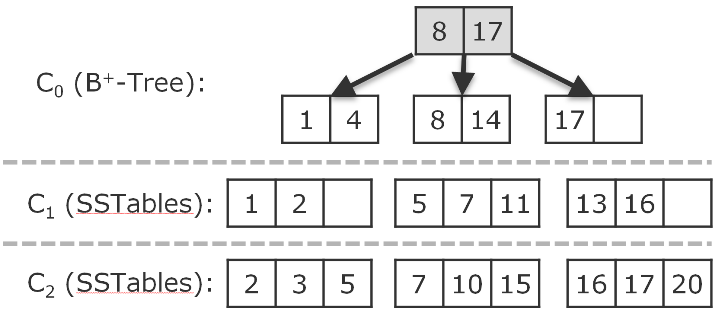


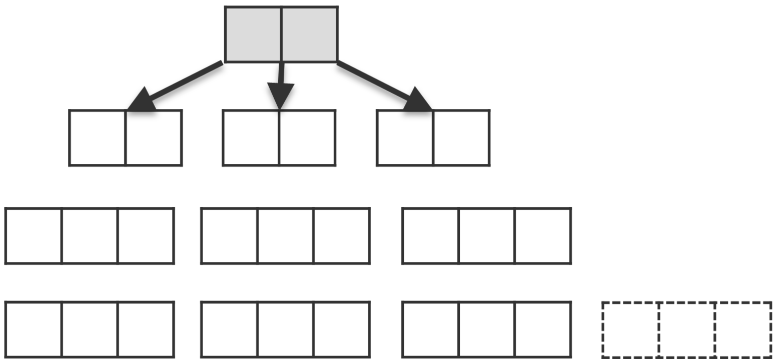
2. Insert the key **3** into the depicted LSM tree instance: **2 points**

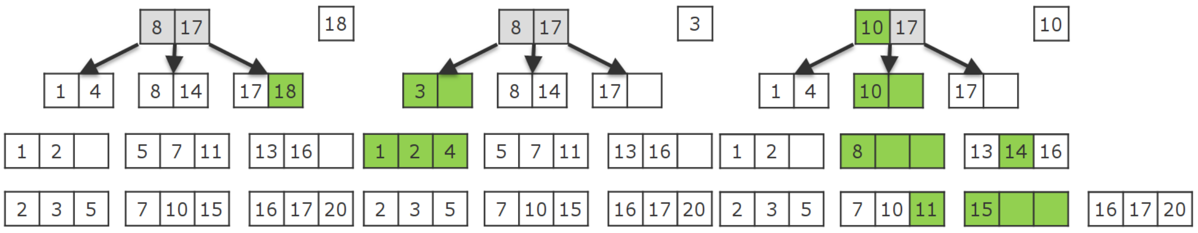




3. Insert the key **10** into the depicted LSM tree instance: **3 points**







Musterlösung:

Grading:

- 1P 18 correctly inserted into the B-tree and only the B-tree
- 1P 3 replaces the leftmost leaf
- 1P 1,2,4 is the new first SSTable; no new SSTable, no further merges, the key 1 from the leaf replaces the key 1 in the former SSTable
- 1P 10 is in the root and leaf of the B-tree
- 1P 8 and 13,14,16 are the new SSTables in level C1
- 1P 7,10,11 and 15 are the new SSTables in level C2; it does not matter in which order the SSTables are presented, i.e., whether 15 or 16,17,20 is in the dotted SSTable box - the SSTables can be placed in any order, because they are indexed and arbitrarily placed on disk anyway.

Task 9: Batch Processing

Suppose you are given three datasets: A `students` dataset that contains general information about students, a `courses` dataset that describes available courses for the students, and an `enrollment` dataset, which is basically a join table between `students` and their `courses`. The following code snippets read the three datasets into Spark `Datasets`:

```
val students = spark
  .read
  .option("quote", "\"")
  .option("delimiter", ",")
  .csv(s"data/students.csv")
  .toDF("ID", "Name", "Semester", "Supervisor")
  .as[(String, String, String, String)]

val enrollments = spark
  .read
  .option("quote", "\"")
  .option("delimiter", ",")
  .csv(s"data/enrollments.csv")
  .toDF("StudentID", "CourseID", "Credits")
  .as[(String, String, String)]

val courses = spark
  .read
  .option("quote", "\"")
  .option("delimiter", ",")
  .csv(s"data/courses.csv")
  .toDF("ID", "Title", "Teacher", "Topic")
  .as[(String, String, String, String)]
```

Use the three `Datasets` to solve the following tasks. You may use Spark's *Dataset* and/or *DataFrame* API but no SQL! Also have a look at the *Dataset* API documentation at the end of this task. If you are not sure about how a particular interface, call, or class works, make a good guess and provide a comment on how you *think* it works.

1. Write a Spark transformation pipeline that starts with the `enrollments` `Dataset`, then removes all those enrollments that grand less than 3 credits, then multiplies the credits by 2/3 in order to transform them into *SWS*s, and finally displays the results in tabular form with schema `(StudentID, CourseID, SWS)` on the standard output. **4 points**

   Musterlösung:

```
enrollments
  .filter(e => Integer.valueOf(e._3) >= 3)
  .map(e => (e._1, e._2, Integer.valueOf(e._3) * 2/3))
  .toDF("StudentID", "CourseID", "SWS")
  .show()
```

   Grading:

   - 1P filter; missing the integer conversion is ok
   - 1P map: 0.5P for multiplication, 0.5P for not missing the other values
   - 0.5P toDF renaming
   - 0.5P show
   - We give only 0.5P for renaming and showing, because the next task gives points for the same two operations.

2. Translate the following SQL query into a Spark transformation pipeline. **4 points**

```
SELECT Semester, COUNT(ID) AS Students
FROM students
WHERE Semester <= 10
GROUP BY Semester;
```

   Musterlösung:

```
students
  .groupByKey(row => row._3)
  .mapGroups((key, iter) => (key, iter.size))
  .filter(t => Integer.valueOf(t._1) <= 10)
  .toDF("Semester", "Students")
  .show()
```

   Grading:

   - 1P groupByKey

- 1P mapGroups
- 1P filter
- 0.5P toDF renaming
- 0.5P show or some other action
- We give only 0.5P for renaming and showing, because the previous task gives points for the same two operations.
- The action does not matter here, because the SQL query also does not imply an action.

3. Rank all teachers by the number of enrollments that they got for all their courses. You can assume that teachers have unique names in `courses.Teacher` and that the `enrollments` file stores all enrollments for all courses ever given. Teacher that never gave a course do not exist in the files and will not appear in the ranking. Report a list of `(Teacher, SumEnrollments)` that is sorted by `SumEnrollments`.
4 points

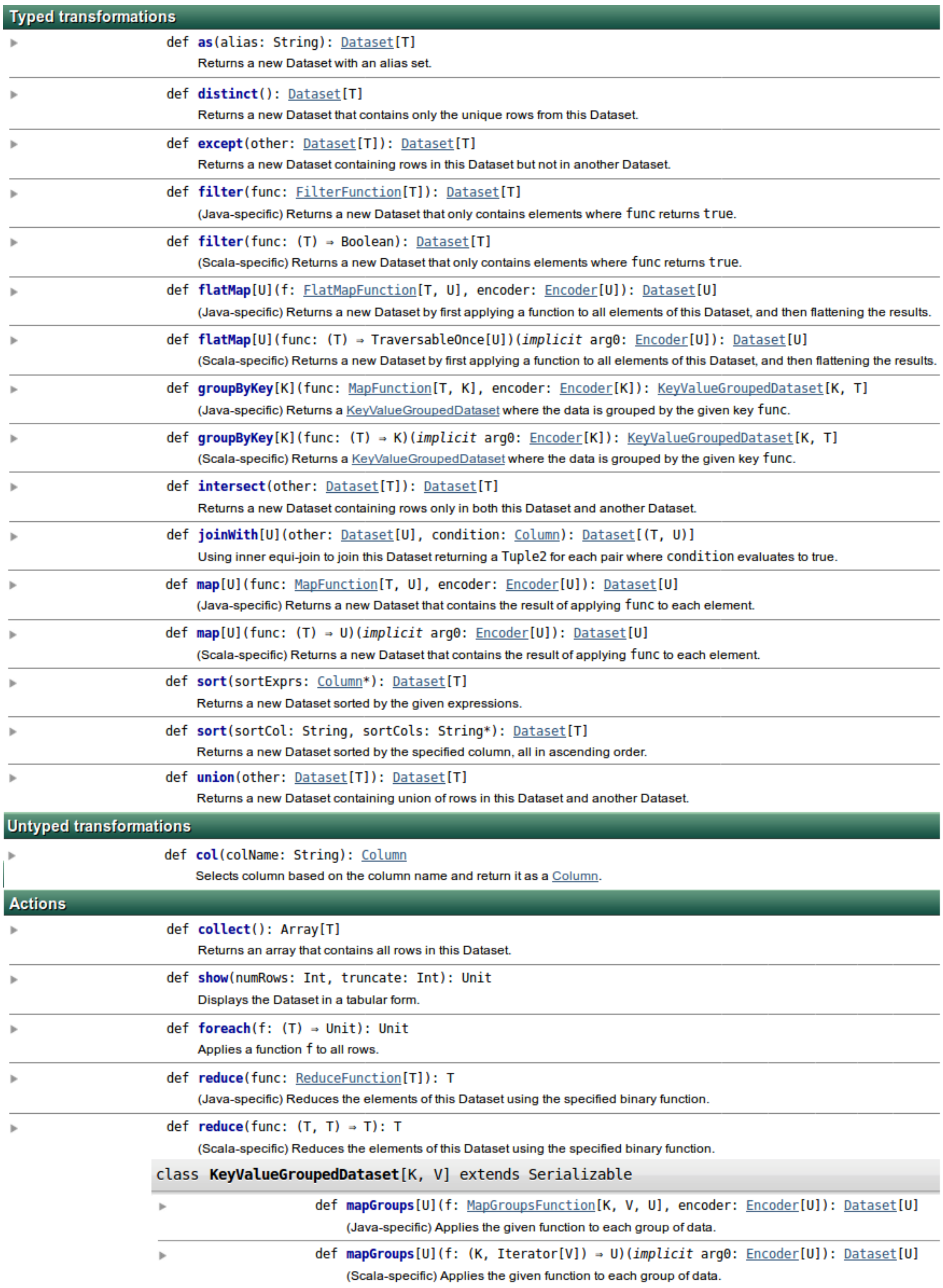
Musterlösung:

```
enrollments
  .joinWith(courses, col("CourseID") === col("ID"))
  .map(s => (s._1._1, s._1._2, s._1._3, s._2._2, s._2._3, s._2._4))
  .groupByKey(row => row._5)
  .mapGroups((key, iter) => (key, iter.size))
    .toDF("Teacher", "SumEnrollments")
    .as[(String, Integer)]
  .sort(col("SumEnrollments"))
  .show()
```

It does not matter if `col(ID)`, `$ID or t._1` or `=`, `==`, `or` `===` is used.

Grading:

- 1P joinWith
- 1P groupByKey
- 1P mapGroups
- 1P sort
- Bonus point: 1P for map after join



Typed transformations

`def as(alias: String): Dataset[T]`
Returns a new Dataset with an alias set.

`def distinct(): Dataset[T]`
Returns a new Dataset that contains only the unique rows from this Dataset.

`def except(other: Dataset[T]): Dataset[T]`
Returns a new Dataset containing rows in this Dataset but not in another Dataset.

`def filter(func: FilterFunction[T]): Dataset[T]`
(Java-specific) Returns a new Dataset that only contains elements where `func` returns `true`.

`def filter(func: (T) ⇒ Boolean): Dataset[T]`
(Scala-specific) Returns a new Dataset that only contains elements where `func` returns `true`.

`def flatMap[U](f: FlatMapFunction[T, U], encoder: Encoder[U]): Dataset[U]`
(Java-specific) Returns a new Dataset by first applying a function to all elements of this Dataset, and then flattening the results.

`def flatMap[U](func: (T) ⇒ TraversableOnce[U])(implicit arg0: Encoder[U]): Dataset[U]`
(Scala-specific) Returns a new Dataset by first applying a function to all elements of this Dataset, and then flattening the results.

`def groupByKey[K](func: MapFunction[T, K], encoder: Encoder[K]): KeyValueGroupedDataset[K, T]`
(Java-specific) Returns a KeyValueGroupedDataset where the data is grouped by the given key `func`.

`def groupByKey[K](func: (T) ⇒ K)(implicit arg0: Encoder[K]): KeyValueGroupedDataset[K, T]`
(Scala-specific) Returns a KeyValueGroupedDataset where the data is grouped by the given key `func`.

`def intersect(other: Dataset[T]): Dataset[T]`
Returns a new Dataset containing rows only in both this Dataset and another Dataset.

`def joinWith[U](other: Dataset[U], condition: Column): Dataset[(T, U)]`
Using inner equi-join to join this Dataset returning a `Tuple2` for each pair where `condition` evaluates to true.

`def map[U](func: MapFunction[T, U], encoder: Encoder[U]): Dataset[U]`
(Java-specific) Returns a new Dataset that contains the result of applying `func` to each element.

`def map[U](func: (T) ⇒ U)(implicit arg0: Encoder[U]): Dataset[U]`
(Scala-specific) Returns a new Dataset that contains the result of applying `func` to each element.

`def sort(sortExprs: Column*): Dataset[T]`
Returns a new Dataset sorted by the given expressions.

`def sort(sortCol: String, sortCols: String*): Dataset[T]`
Returns a new Dataset sorted by the specified column, all in ascending order.

`def union(other: Dataset[T]): Dataset[T]`
Returns a new Dataset containing union of rows in this Dataset and another Dataset.

Untyped transformations

`def col(colName: String): Column`
Selects column based on the column name and return it as a Column.

Actions

`def collect(): Array[T]`
Returns an array that contains all rows in this Dataset.

`def show(numRows: Int, truncate: Int): Unit`
Displays the Dataset in a tabular form.

`def foreach(f: (T) ⇒ Unit): Unit`
Applies a function `f` to all rows.

`def reduce(func: ReduceFunction[T]): T`
(Java-specific) Reduces the elements of this Dataset using the specified binary function.

`def reduce(func: (T, T) ⇒ T): T`
(Scala-specific) Reduces the elements of this Dataset using the specified binary function.

`class KeyValueGroupedDataset[K, V] extends Serializable`

`def mapGroups[U](f: MapGroupsFunction[K, V, U], encoder: Encoder[U]): Dataset[U]`
(Java-specific) Applies the given function to each group of data.

`def mapGroups[U](f: (K, Iterator[V]) ⇒ U)(implicit arg0: Encoder[U]): Dataset[U]`
(Scala-specific) Applies the given function to each group of data.

Extra page 1

Extra page 2

Extra page 3