

Distributed Data Analytics – Exam

Winter Term 2017/2018

Matriculation Number: _____

0	1	2	3	4	5	6	7	Σ
1	9	6	5	6	23	13	8	71

Important Rules:

- The exam must be solved within 180 minutes (09:00 – 12:00).
- 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 28 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. Assume that you wrote an algorithm in Akka. After careful analysis of the algorithm, you know that 90% of its execution time would profit from parallelization, while 10% of it is non-parallelizable. According to Amdahl's Law, how many cores must an idealized cluster, i.e., one with no distribution overhead, provide to achieve a speedup of 9? **2 points**

Musterlösung:

$$\text{speedup} = 9$$

$$p = 9/10$$

$$\text{speedup} = 1 / ((1-p)+p/s)$$

$$9 = 1 / ((1-9/10)+(9/10)/s)$$

$$s = 81$$

Grading:

- 1P correct formula
- 1P correct calculation

2. In distributed systems, faults occur more often than in non-distributed systems, because these systems are prone to additional types of faults. Name three types of faults that you have to deal with in distributed systems but not in non-distributed systems. **3 points**

Musterlösung:

- network faults
- clock deviation
- partial power failures
- nondeterministic behavior
- untrustworthy messages/information
- ...

Grading:

- 1P for each correct type of fault

3. To solve some particular task, it might be important that a group of Akka actors synchronizes their local times with a dedicated master actor. For this purpose, you implemented the network time protocol (NTP) in each actor: The actor frequently sends a time message to the master actor and receives a message with three time-stamps back. Given that one response contained $t_0 = 10:32:07$, $t_1 = 10:32:19$, and $t_2 = 10:32:20$, to what time must the actor set its current time, if the current time shows 10:32:29 upon message receipt? **2 points**

Musterlösung:

$$\text{delta} = ((t_1 - t_0) + (t_2 - t_3)) / 2$$

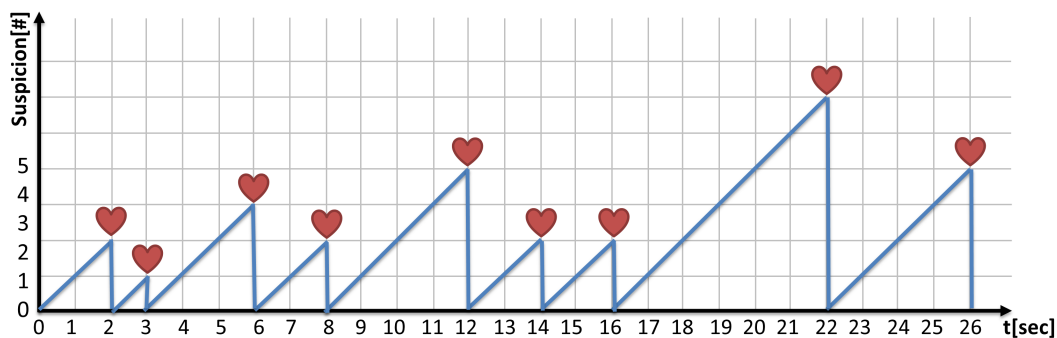
$$\text{delta} = ((19 - 7) + (20 - 29)) / 2$$

$$\text{delta} = 3 / 2 = 1.5$$

Set time to 10:32:30 or 10:32:31

Grading:

- 1P correct formula
 - 0.5P correct calculation
 - 0.5P correct interpretation i.e. +1.5
4. Many distributed systems use heartbeats to detect failed nodes: A monitored process p sends periodical heartbeat messages to the server process q. Based on these heartbeats, q calculates a continuous suspicion value for p. In the end, however, q must translate this suspicion into binary trust or distrust. One dynamic approach to this interpretation problem uses dynamic thresholds T_{high} and T_{low} . Assume that q initializes both T_{high} and T_{low} to 1 and then monitors the following suspicion progress. In what time intervals does q distrust p? **2 points**

**Musterlösung:**

[1, 2]

[5, 6]

[11, 12]

[20, 22]

Grading: 0.5P for each correct intervall

Task 2: Data Models and Query Languages

1. Many distributed storage engines use schema-on-read rather than schema-on-write. Name two advantages and two disadvantages for this strategy. **2 points**

Musterlösung:

Advantages:

Higher write throughput/efficiency
 Schema flexibility
 Easier data integration (simply copy data into a lake)
 No information loss during data collection
 ...

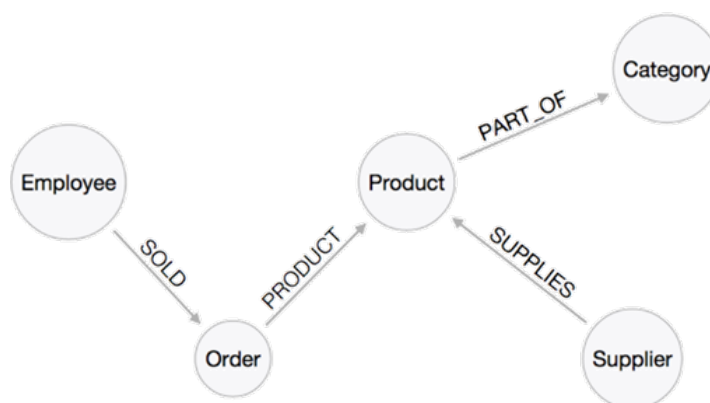
Disadvantages:

Lower read throughput/efficiency
 Lower compression potential
 Joins, aggregations, ... are a lot more difficult
 ...

Grading:

- 0.5P for each correct advantage/disadvantage

2. The following graph describes the schema of an enterprise resource planning (ERP) database. Assume that this database was created in Neo4J and you have to use Cypher to query its content. Write a query that searches for an Employee named "Mr. T" and list all Suppliers, whose products "Mr. T" has sold. **4 points**



Musterlösung:

MATCH (:Employee {name:"Mr. T"})-[:SOLD]->(:Order)-[:PRODUCT]->

(:Product)<-[:SUPPLIES]-(supplier:Supplier)

RETURN supplier

Grading:

- If bracket types are not correct or the syntax is a little bit wrong, we do not mark that as an error
- Because the depicted schema is so small, we cannot grade that arrows and nodes match the correct types of nodes
- 1P MATCHing of Employee with name Mr.T
- 1P arrows follow the join path correctly
- 1P MATCH and RETURN terms
- 1P RETURNing supplier

Task 3: Storage and Retrieval

Sorted String Tables (SSTables) are a popular approach to balance out write throughput and random-read performance for log-structured data. Assume that your data is a stream of stock-counts produced by a tool store. You might, then, encounter the following tasks.

1. SSTables are read-only so that a system needs to start a new SSTable whenever some value does not fit into an existing SSTable. To compact the data, the system frequently merges SSTables. Given the SSTable from 15/02/17 and the segment file from 16/02/17 below, calculate their merge. **2 points**

15/02/17

key	value
drill	2
grinder	8
hammer	13
knife	32
nail	934
pliers	21
saw	8
wrench	42

16/02/17

key	value
drill	1
drill	0
drill	15
hammer	13
nail	834
pliers	20
pliers	19
saw	19
wrench	40
wrench	41

Musterlösung:

key	value
drill	15
grinder	8
hammer	13
knife	32
nail	834
pliers	19
saw	19
wrench	41

Grading:

- -0.5P for every wrong element; not less than 0P

2. In a distributed setting, our log-data could be partitioned across multiple SSTables. Each node in the cluster holds its share of the set of SSTables. Our tool data might therefore be contained in the following SSTables, each placed on a different node:

key	value
drill	32
grinder	244
hammer	134

key	value
knife	245
nail	8335
pliers	153

key	value
saw	253
wrench	247

Given that these SSTables are indexed with a *sparse index*, how would you find the value of key “grinder”? Write down the index entries that you need for the look-up and describe the entire look-up procedure, i.e., all steps needed until the key’s value is read. **3 points**

Musterlösung:

Index Entries:

“drill” → (Partition-Host-IP, Partition-ID/Partition-Physical-Address)

“knife” → (Partition-Host-IP, Partition-ID/Partition-Physical-Address)

Procedure:

- look-up index entry by “drill” < “grinder” < “knife”
- read host IP and partition ID (or something that indicates that keys of the index point to SSTables on the nodes)
- go to host, go to partion, run linear-scan/binary-search for “grinder”

Grading:

- 1P reasonable indexes
- 1P finding the partition
- 1P finding the key-value pair inside the partition

Task 4: Replication

1. Many distributed systems that support leaderless replication spread new information by using quorum reads/writes and the gossip protocol. Assume that such a system is set up on 1000 replicas, defines a quorum as (10,5), and runs the gossip protocol with a frequency of 10 seconds.

- a) Discuss whether or not the described system is *quorum consistent*. **1 points**

Musterlösung:

It is not quorum consistent, because quorum consistency means that $r+w>n$ and in this setup $10+5<1000$

- b) If a write has succeeded on exactly 16 nodes, how long does it take for this write to reach all other nodes via gossiping (in the best case)? **3 points**

Musterlösung:

$2^{\text{rounds}} \geq \text{nodesreached}$ rounds $\geq \log_2(\text{nodesreached})$ Rounds to reach

16 nodes: rounds $\geq \log_2(16) = 4$

Rounds to reach 1000 nodes: rounds $\geq \log_2(1000) = 10$

Rounds from 16 to 1000 nodes: $10-4 = 6$

Time to reach all 1000 nodes: $6 \cdot 10\text{sec} = 60\text{sec}$

Grading:

- 1P for gossiping formula
- 1P for considering that 16 nodes know the write already
- 0.5P for current round calculation
- 0.5P for time calculation

2. The CAP-Theorem describes three properties of database management systems (DBMS) and claims that, if faults occur, only two of them can be guaranteed. Assume that we have a single-leader replicated DBMS that seeks to guarantee C and P of CAP. How does it ensure C and how does this (potentially) violate A? **2 points**

Musterlösung:

- For every write, the leader blocks until all replicas acknowledged the write
- This guarantees consistency, because there are no concurrent writes and writes either fail or succeed system-wide
- This also violates availability, because the system does not process other writes if one write is in progress

Grading:

- 1P for statement: leader blocks writes and waits until *all* followers acknowledged the write to ensure consistency
- 1P for statement: leader blocking writes might violate availability

Task 5: Actor Programming

1. The Actor model is a stricter message-passing model that treats actors as the universal primitives of concurrent computation. What are the three components that define an actor? **2 points**

Musterlösung:

State + Behavior + Mailbox

Grading: -1P for any missing component, but not less than 0P

2. Message-passing as used by the actor model is only one communication principle among others. Name one other communication principle and two properties, in which that principle differs in comparison to message-passing. **2 points**

Musterlösung:

Databases: data vs. messages, no response vs. maybe response, no recipient addressing vs. recipient addressing, persisting vs. volatile, usually disk-based communication vs. usually RAM-based communication, ...

or

Services: synchronous vs. asynchronous, function calls vs. messages, guaranteed response vs. maybe response, blocking vs. non-blocking, ...

Grading:

- 1P naming a databases or services
- 0.5P naming a difference

3. Actors in Akka live in hierarchies. Each such hierarchy is maintained by an *ActorSystem*. Which of the following statements on actor hierarchies and *ActorSystems* is *true*? Tick the true ones. **3 points**

An *ActorSystem* can span across several nodes in a cluster maintaining the lifecycle of all its actors and the nodes resources.

The *let it crash* philosophy of Akka says that, if an error occurs, user actors should not try to fix it but let it crash so that the *user guardian* can handle their errors on a more supervisory level.

Idle actors do not block CPU resources, because their *ActorSystem* dispatches its actors dynamically on a pool of threads so that there is no fix actor to thread assignment.

- The *Reaper Pattern* defines a dedicated reaper actor that knows all actors of the *ActorSystem* and initiates a clean shutdown by sending all these actors a *PoisonPill* message.
- The *dead letter box* is the message queue of the *root guardian* actor, who uses the dead letters to identify crashed actors.
- Messages that are delivered between actors of different *ActorSystems* must be serialized, but messages between actors of the same *ActorSystem* can be exchanged without serialization.

Musterlösung:

- Every ActorSystem is hosted on one node only.
- Any supervisor Actor could handle the error; if the user guardian has to take it, than it would probably need to restart most of the system.
-
- The Reaper does not send PoisonPills.
- It is neither the root guardian's message queue nor is it used to identify crashed actors. There are more reliable ways of supervision than dead letters!
-

Grading: 0.5P for each correctly ticked or non-ticked field

4. Assume someone built an Akka application that monitors arbitrarily many sensors. Whenever you add a sensor to the application, the application creates a new *SensorWatcher* actor that reads and forwards a new value of its assigned sensor every 10 seconds. This actor uses the following code snippet:

```
while (true) {
    String reading = this.sensor.read();
    this.forwardReading(reading);
    Thread.sleep(10000);
}
```

What issue does the described application have? Explain what happens if you add more and more sensors to it. **2 points**

Musterlösung:

At some point, all threads of the system are blocked by SensorWatcher actors. The system cannot process any forwarded reading and does not monitor any further sensors, because all its threads are mostly sleeping. The forwarded messages can, at some point, not be send any more, because the mailboxes of the reveiving actors are full.

Grading:

- 1P for recognizing that the sleep() will block the actors' threads
- 1P for explaining a plausible consequence

5. In this task, we build a multi-leader replicated in-memory key-value store with Akka. The key-value store should consist of arbitrary many leader-actors. Each leader holds a replica of the data in a local hash-map. When the leader receives a *ReadMessage* with a particular key, it replies with a *ReadResponseMessage* containing the key's value; when the leader receives a *WriteMessage* with a key-value mapping, it updates its local replica accordingly and replies with a *WriteResponseMessage*.

To keep the data consistent, leaders also forward *WriteMessages* as *PropagationMessages* to all other leaders. For conflict resolution, the system should implement *Lamport timestamps* and use the *last-write-wins* principle.

Complete the *LeaderActor* below by implementing the *LamportTimestampedValue*, *WriteMessage*, *WriteResponseMessage*, *createReceive()*-function, and the two *handle()*-functions for *Write*- and *PropagationMessages*. You do not have to implement the setup of the ActorSystem or any other actor that might as well be needed for the key-value store. Use the algorithm gaps for your solution and the extra pages at the end of this exam for drafts. **14 points**

```
import java.io.Serializable;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import akka.actor.AbstractLoggingActor;
import akka.actor.ActorRef;
import akka.actor.Props;

public class LeaderActor extends AbstractLoggingActor {

    public static Props props(int identifier) {
        return Props.create(LeaderActor.class, () -> new LeaderActor(identifier));
    }

    public static class LamportTimestampedValue implements Serializable {
        private static final long serialVersionUID = 1L;
        public String value;
    }
}
```

```
public static class ReadMessage implements Serializable {
    private static final long serialVersionUID = 1L;
    public int key;
    public ReadMessage(int key) {
        this.key = key;
    }
}

public static class ReadResponseMessage implements Serializable {
    private static final long serialVersionUID = 1L;
    public String value;
    public ReadResponseMessage(String value) {
        this.value = value;
    }
}

public static class WriteMessage implements Serializable {
    private static final long serialVersionUID = 1L;
    public int key;
    public String value;
}

public static class WriteResponseMessage implements Serializable {
    private static final long serialVersionUID = 1L;
}

public static class PropagationMessage implements Serializable {
    private static final long serialVersionUID = 1L;
    public int key;
    public LamportTimestampedValue value;
    public PropagationMessage(int key, LamportTimestampedValue value) {
        this.key = key;
        this.value = value;
    }
}
```

```
public static class PropagationMessage implements Serializable {
    private static final long serialVersionUID = 1L;
    public int key;
    public LamportTimestampedValue value;
    public PropagationMessage(int key, LamportTimestampedValue value) {
        this.key = key;
        this.value = value;
    }
}

private final Map<Integer, LamportTimestampedValue> replica = new HashMap<>();
private final List<ActorRef> otherLeaders = new ArrayList<>();
private final int identifier;

public LeaderActor(final int identifier) {
    this.identifier = identifier;
}

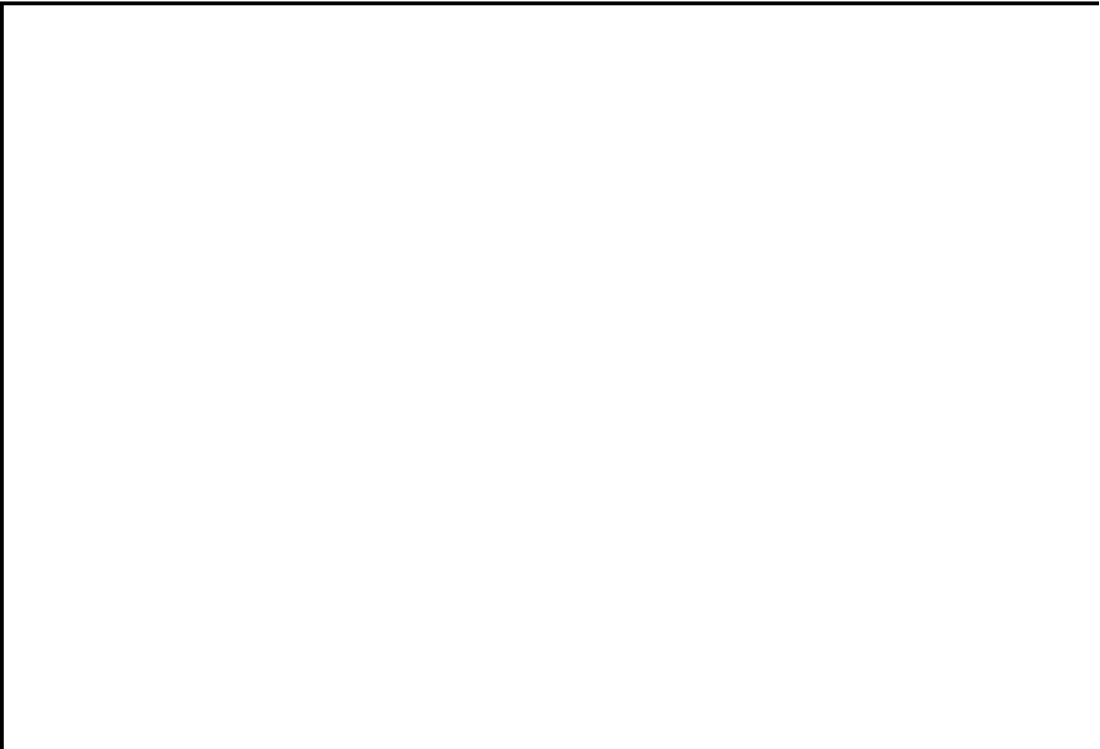
public void addLeaderRef(ActorRef otherLeader) {
    this.otherLeaders.add(otherLeader);
}

@Override
public Receive createReceive() {
    return receiveBuilder()
        .build();
}

private void handle(ReadMessage message) {
    this.getSender().tell(new ReadResponseMessage(this.replica.get(message.key).value),
        this.getSelf());
}
}
```



```
private void handle(WriteMessage message) {  
    LamportTimestampedValue oldValue = this.replica.get(message.key);
```



```
}  
private void handle(PropagationMessage message) {  
    LamportTimestampedValue oldValue = this.replica.get(message.key);
```



```
}  
}
```

Musterlösung:

Grading:

- Do not be strict about the syntax: missing ; or small mistakes are ok
- 2P for counter + identifier in LamportTimestampedValue
- 1P counter in WriteMessage
- 1P counter in WriteResponseMessage
- 3P for matches in createReceive()
- 5P handle WriteMessage
 - * 1P counter increment
 - * 1P Lamport Logic ($== \text{null}$, $< \text{counter}$, $== \text{counter} + < \text{identifier}$)
 - * 1P value update
 - * 1P tell PropagationMessage
 - * 1P tell WriteResponseMessage
- 2P handle WriteResponseMessage
 - * 1P Lamport Logic
 - * 1P value update

```
public static class LamportTimestampedValue implements Serializable {
    private static final long serialVersionUID = 1L;
    public String value;
    public int counter;
    public int identifier;
    public LamportTimestampedValue(String value, int counter, int identifier) {
        this.counter = counter;
        this.identifier = identifier;
        this.value = value;
    }
}
```

```
public static class WriteMessage implements Serializable {
    private static final long serialVersionUID = 1L;
    public int key;
    public String value;
    public int counter;
    public WriteMessage(int key, String value, int counter) {
        this.key = key;
        this.value = value;
        this.counter = counter;
    }
}
```

```
public static class WriteResponseMessage implements Serializable {
    private static final long serialVersionUID = 1L;
    public int counter;
    public WriteResponseMessage(int counter) {
        this.counter = counter;
    }
}
```

```

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

private void handle(WriteMessage message) {
    LamportTimestampedValue oldValue = this.replica.get(message.key);

    int counter = (oldValue == null) ? message.counter : Math.max(oldValue.counter, message.counter);
    counter++;

    LamportTimestampedValue newValue = new LamportTimestampedValue(message.value, counter, this.identifier);

    this.replica.put(message.key, newValue);

    for (ActorRef otherLeader : this.otherLeaders)
        otherLeader.tell(new PropagationMessage(message.key, newValue), this.getSelf());
    this.getSender().tell(new WriteResponseMessage(counter), this.getSelf());
}

private void handle(PropagationMessage message) {
    LamportTimestampedValue oldValue = this.replica.get(message.key);

    if (oldValue == null ||
        oldValue.counter < message.value.counter ||
        (oldValue.counter == message.value.counter && oldValue.identifier < message.value.identifier)) {
        this.replica.put(message.key, message.value);
    }
}
}

```

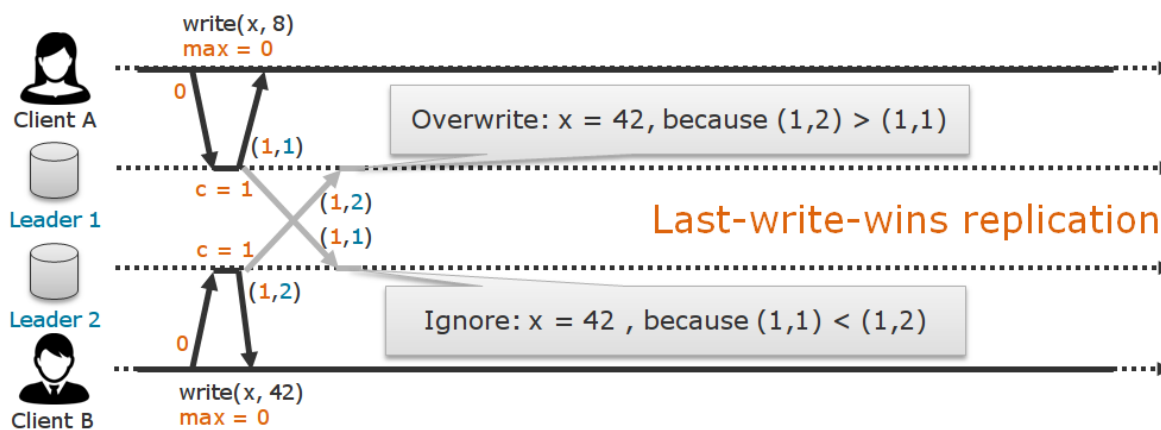
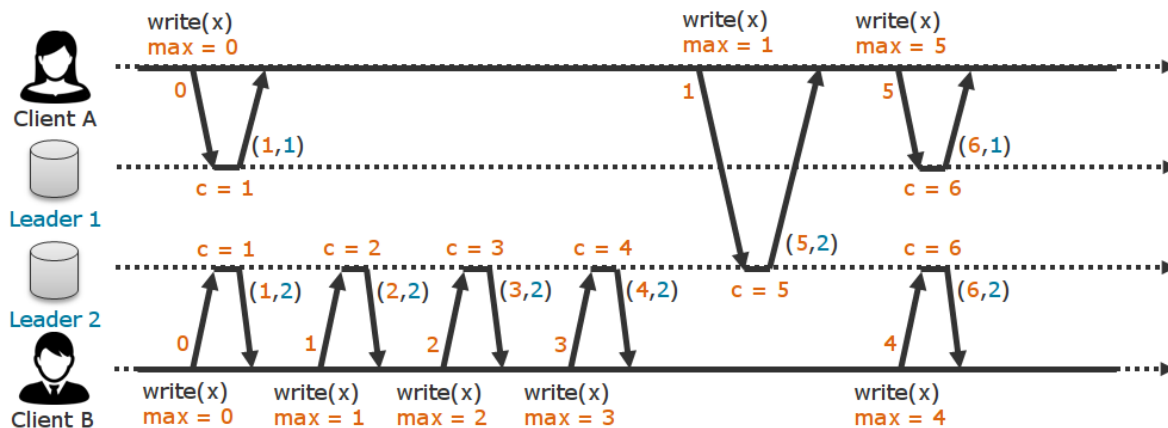
Lamport timestamps

distributed c

- Each node has a unique **identifier** and a **counter** for processed operations
- **Lamport timestamp**:
 - A pair (**counter**, **identifier**)
 - Globally unique for each event
 - Imposes a **total order** consistent with causality:
 - Order by counter
 - If counters are equal, use identifier as tie-breaker
- Achieving causal order consistency:

- Nodes store their current counter **c**
 - Clients store the max counter **m** seen so far (send with each event)
 - Nodes increment their counter as **c = max(c,m) + 1**
 - Counter moves past some events that happened elsewhere

Task 6: Batch Processing



1. Transformations in a batch processing pipeline can be categorized as either *narrow* or *wide*. Which of the following statements on narrow and wide transformations are *true*? Tick the true ones. **3 points**

- Narrow transformations preserve the size of their inputs, which means that number of input entries equals number of output entries.
- Wide transformations may produce outputs that are smaller or larger than their input size.
- sorted()* is a wide transformation.
- join()* is a wide transformation.
- map()* is a narrow transformation.
- distinct()* is a narrow transformation.

Musterlösung:

Narrow transformations are those where records in the same output partition originate from the same input partition. The number of records can, of course, become larger (flatmap) or smaller (filter).

distinct requires shuffling, i.e., it is not narrow

Grading:

- 0.5P for each correctly ticked or non-ticked field

2. Consider a relational dataset with the following schema:

Students(*ID*, *Name*, *Password*, *Gene*)

Your task is to find for each student its longest common substring in the *Gene*-attribute with any of the other students. Write *one* Spark job (= one transformation pipeline) that answers this task. The output should be printed to the console of the driver and show the values *ID* (the unique ID of every student) and *Substring* (the students longest common substring) for every student. You can use the *Dataset* and/or *DataFrame* API but no SQL!

Hints:

- Have a look at the *Dataset* API documentation two pages further on.
- 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.
- The function *longestCommonSubstring()* calculates the longest common substring of two given strings. This function is already given and does not need to be implemented.
- The code for reading the input dataset is also given and can be used to start the pipeline.

10 points

```
// longestCommonSubstring(a: String, b: String): String = { ... }
import de.hpi.dda.longestCommonSubstring

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

Musterlösung:

```
val students2 = spark
  .read
  .option("quote", "\"")
  .option("delimiter", ",")
  .csv(s"data/students.csv")
  .toDF("ID", "Name", "Password", "Gene")
  .as[(String, String, String, String)]

students
  .joinWith(students2, students.col("ID") != students2.col("ID"))
  .map(t => (t._1._1, longestCommonSubstring(t._1._4, t._2._4)))
  .groupByKey(t => t._1)
  .mapGroups{ case (key, iterator) => (key, iterator
    .map(t => t._2)
    .reduce((a,b) => { if (a.length > b.length) a else b })) }
  .toDF("ID", "Substring")
  .show()
```

Grading:

- 5P pipeline construction: join, map, groupby, aggregate, final action
- 5P correct/reasonable arguments/UDFs: in join, in map, in groupby, in aggregate, in outputting results to console

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.

Task 7: Stream Processing

1. Consider the following Apache Flink program.

```
import org.apache.flink.streaming.api.scala._
import org.apache.flink.streaming.api.windowing.time.Time

object WindowWordCount {
  def main(args: Array[String]) {

    val env = StreamExecutionEnvironment.getExecutionEnvironment
    val text = env.socketTextStream("localhost", 9999)

    val counts = text.flatMap { _.toLowerCase.split("\\W+") filter { _.nonEmpty } }
      .map { (_, 1) }
      .keyBy(0)
      .timeWindow(Time.seconds(5))
      .sum(1)

    counts.print()

    env.execute("Window Stream WordCount")
  }
}
```

This program is both data- and task-parallel. Explain both properties in the context of transformation pipelines. For each property, also give an example from the depicted Flink pipeline: Where is it data-parallel and where is it task-parallel?

4 points

Musterlösung:

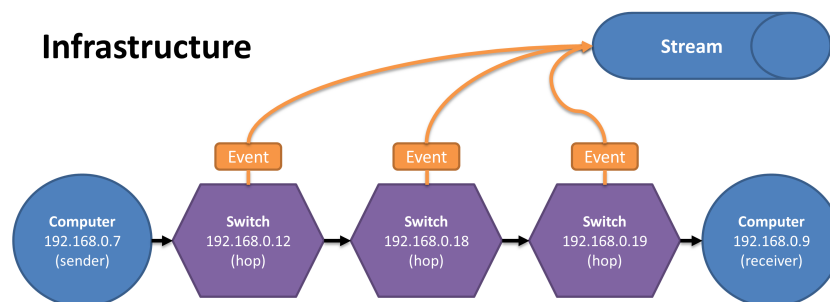
Data-parallel, because each transformation in the pipeline can be executed in parallel on different parts of the input stream, e.g., *flatMap* can be called on different lines simultaneously.

Task-parallel, because different transformation functions can calculate different subtasks in parallel, e.g., *flatMap* maps rows while *map* maps words to tuples.

Grading:

- 1P explain data-parallel
- 1P explain task-parallel
- 1P example data-parallel
- 1P example task-parallel

2. Assume you are doing network traffic analysis on the HPI intranet. All switches in the network emit an event for every package that they transmit. An event is a tuple of $(package_id, sender, last_hop, next_hop, receiver)$, where sender, hops, and receiver are IP-adresses of nodes in your network. Your infrastructure combines all these events into one stream of events. Given this event stream, your goal is to calculate for each package its path (= list of hops) through the network for a later bottleneck analysis. Because the stream of events is volatile, you need to apply windowing on the event stream.



- a) What kind of window do you need to record the paths?

1 points

Musterlösung:

Session Window

- b) What parameters does this window require, i.e., when does it start, how long is it, and when does it end?

3 points

Musterlösung:

Straggler events do not need to be considered. But you could do so by ending a window only if the chain of hops is without gaps.

Correct answer depends on first answer:

Session Window: *start event* (sender = last hop); *end event* (next hop = receiver); lasts for as long a message travels on the network

Tumbling Window: *event length* (time or size); start when previous window ended; ends when event length is reached

Hopping Window: *event length* (time or size); *hop length* (time or size); start when hop length reached; ends when event length is reached

Sliding Window: *event length* (time or size); start after each new event; ends when event length is reached

Extra page 1

Extra page 2

Extra page 3