SCALABLE DATA MANAGEMENT SYSTEMS

LAB 3 ANNOUNCEMENT

05. FEB. 2021

AGENDA

1) Announcement of Lab 3
2) Discussion of Lab 2

LAB 3 – OVERVIEW

Deadline: 26.02.2021, 09:50am (i.e., before the presentation of the next lab/solution of the last lab)

Points: 10 (reduced scope compared to other labs)

Part 1: MapReduce Operators

Part 2: MapReduce Execution Framework

Part 3: Using MapReduce

Advanced test specifications will be published as soon as possible

RECAP: MAPREDUCE PROGRAMMING MODEL

Data type: key-value *records (e.g., per line of file)*

Map function:

$$(K_{in}, V_{in}) \rightarrow list(K_{inter}, V_{inter})$$

Reduce function:

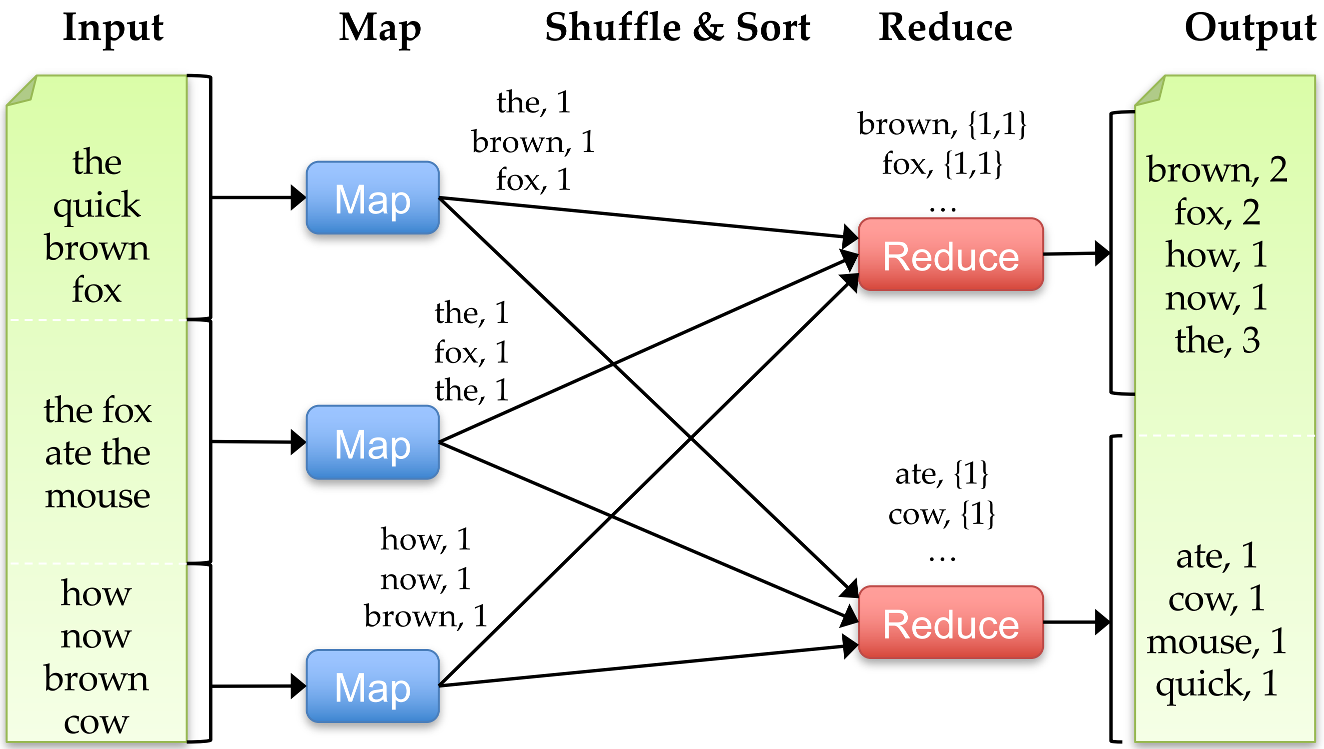
$$(K_{inter}, list(V_{inter})) \rightarrow list(K_{out}, V_{out})$$

EXAMPLE: WORD COUNT

```
def mapper(linenumber, line):
    foreach word in line.split():
        emit(word, 1)


def reducer(key, values): //e.g. ("foo",[1,1])
    emit(key, count(values))
```

RECAP: WORD COUNT EXECUTION



PART 1 – MAPREDUCE OPERATORS

Implement the Sort-, Mapper- and Reducer-Operator as Volcano-Iterator by implementing the open(), next() and close() methods for the classes

- Sort (found in the package de.tuda.dmdb.operator.exercise)
- Mapper (found in the package de.tuda.dmdb.mapreduce.operator.exercise)
- Reducer (found in the package de.tuda.dmdb.mapreduce.operator.exercise)

The next() methods return the next element according to the Iterator model, if all elements have been processed NULL should be returned. The open() and close() methods initialize/clean-up by calling open()/close() on the child-operators and/or initialize member variables.

Please pay attention to the hints and explanations on the following slides.

MAPREDUCE IN DMDB: DATA MODEL

- In Hadoop MapReduce: Map task takes a **set of data** and converts it into another set of data, where individual elements are broken down into **tuples** (key/value pairs). The Reduce task takes the **output** from a map as an **input** and combines those data tuples into a [usually] smaller set of tuples [https://www.tutorialspoint.com/hadoop/hadoop_mapreduce.htm]
- In DMDB:
  - **Set of Data** = Set/List of HeapTables
  - **Tuples** (key,value pairs) = Records/AbstractRecords (with two AbstractSQLValues/Columns, first being the key, second the value)
    This convention is defined as constants in the **`MapReduceOperator`** class
  - When creating a new (output) record for a new key and value, set their values as SQLValues in the record as defined by the `KEY_COLUMN` and `VALUE_COLUMN` constants
  - **Output/Input** = List of HeapTables (each HeapTable represents a partition)
  - In Hadoop MapReduce input data is split and passed to Mapper, our framework expects partitioned data

MAPREDUCE IN DMDB: MAPPER

- Implements map function/interface similar to http://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapreduce/Mapper.html
- The map(...) function can possibly map one <key,value> tuple to multiple new <key,value> tuples
- Instead of "Context" a `Queue<AbstractRecord>` is used to write results (use/pass the the member variable `nextList`)
- The map() method is called for each <key, value> tuple in the input. Implement this logic by completing the open(), next(), close() methods.
- The next() methods retrieves the next tuple and passes it to the `map(…)` function
- Keep in mind that the key and value are stored as columns of a record (you have to extract them)

MAPREDUCE IN DMDB: REDUCER

- Implements map function/interface similar to http://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapreduce/Reducer.html
- Instead of “Context” a `Queue<AbstractRecord>` is used to write results (use/pass the member variable `nextList`)
- The Reducer-Operator assumes that the input is sorted
- It groups tuples with the same key and invokes the reduce(…) method for the key and all its values (stored in an Iterable) – take advantage of the fact that the input is sorted
- You can use the member variable `lastRecord` to implement the grouping in the next() method
- The `next()` method should also return the next output record produced by the reduce function (again, keep in mind that `reduce(…)` should write its result records to `nextList`)

MAPREDUCE IN DMDB: SORT

- Sorting is a blocking operation
- The member variables `sorted` and `sortedRecords` are supposed to help you implement the `next()` method
- Keep it simple: You can use a sorted data structure for `sortedRecords` when initializing this member variable
- Side-Note: The Sort-Operator is used (e.g., in the next part to provide the Reduce-Operator with sorted input)

MAPREDUCE IN DMDB: HINTS FOR IMPLEMENTATION

- Keep in mind that you have to extract the key and value from a record
- The MapReduceOperator class defines helpful constants for this
- Keep in mind that the map() function can map one <key,value>-pair to multiple new records
- The member variables are supposed to facilitate the implementation

PART 2 – MAPREDUCE EXECUTION FRAMEWORK

Implement the MapReduce Execution Layer in DMDB by completing the code for the run() method in the following classes

- MapperTask
- ShuffleSortTask
- ReducerTask
- SinglePhaseTask

found in the package de.tuda.dmdb.mapreduce.task.exercise.

The above classes inherit from the Java Thread class and override its run() method. The classes define what actions are performed in the Map/Shuffle&Sort/Reduce phase during the execution of a MapReduce job. They create intermediate result for the next phase.

Please pay attention to the hints and explanations on the following slides.

MAPREDUCE IN DMDB: EXECUTION MODEL

- Tasks define what happens during the execution of a map-reduce job in DMDB, ie., they define the physical-execution plan for a phase
- A `Task` (ie. Thread) is created for each input partition/HeapTable and is executed in a separate thread (ie. node) in a Thread-Pool (ie. Cluster)
- You don’t have to deal with task creation, this is done in an `Executor`
- The definition of a MapReduce job is analogous to Hadoop MapReduce (compare test-case with tutorial at Apache Hadoop Tutorial)

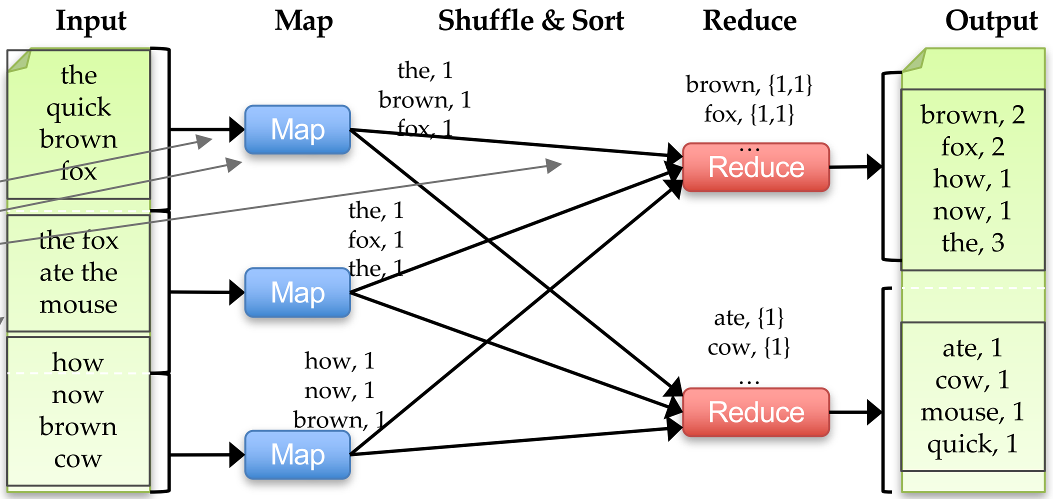
MAPREDUCE IN DMDB: EXECUTION MODEL

- Tasks are used to structure execution in execution unit/phases
- The general data processing model in a `Task` is based on Vulcano-Iterators (ie., chaining of operators)
- There is a 1:1 mapping between input partitions and tasks, i.e., for each input partition the executor will create a separate thread (i.e., task)

Logical Phases of a MapReduce job:

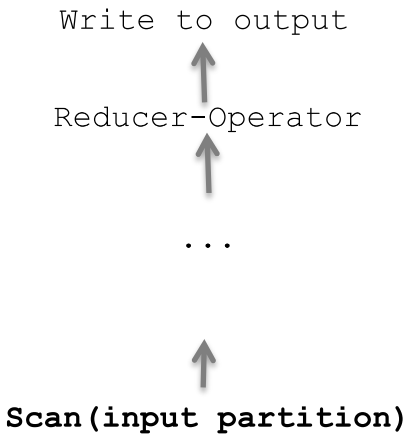
We use physical operators to define the actions, e.g. use TableScan to read input or use Mapper-Operator to do the map-action

Remember we use tables with key-value-records as input, there is one task for each table/partition



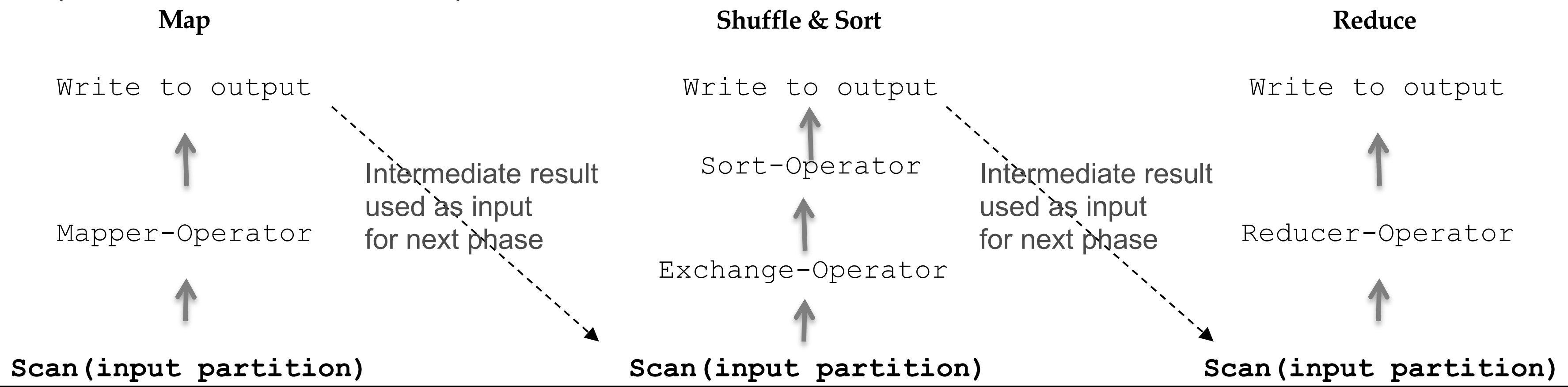
MAPREDUCE IN DMDB: SINGLEPHASETASK

- The SinglePhaseTask combines all logical phases into one execution unit
  - This is possible with Operator chaining / the Iterator model
  - It avoids overhead since it does not require to persist intermediate results
- Implement the correct operator change (extract):



MAPREDUCE IN DMDB: EXECUTION IN MULTIPLE PHASES

- The three tasks `MapperTask, ShuffleSortTask, ReducerTask` divide the execution into three physical phases (Apache Hadoop uses two phases)
  - Each task (ie., phase) creates an intermediate result
- Implement the correct operator chain for each task (see below)
- The `MultiPhaseExecutor` chains these tasks to execute the complete MapReduce job (have a look at the code):



MAPREDUCE IN DMDB: HINTS FOR IMPLEMENTATION

- You can use the following way to create a instance of a Mapper/Reducer in a Task:

```
MapperBase<? Extends AbstractSQLValue, ? Extends AbstractSQLValue, ? Extends
AbstractSQLValue, ? Extends AbstractSQLValue> mapper;
try{
mapper = this.mapperClass.newInstance();
} catch (…) {…}
```

- A tasks completes when all input records were processed
- Again:
  - Use a TableScan operator to read records from the input
  - Use HeapTable.insert() to write to the output
- Look at test cases to see how the code is used
- As always the provided test(s) are not exhaustive, it is recommended that you write additional test if required

PART 3 – USING MAPREDUCE

Use the MapReduce programming paradigm and the frame work that you implemented in the previous parts to solve the tasks described in the following classes

- DotProduct
- UserAverageSessionLength
- UserSessionCount
- WordCounter

found in the package de.tuda.dmdb.mapreduce.exercise.

The above classes inherit from the BaseMapReduceExercise class which defines how these classes are executed as MapReduce jobs in DMDB’s execution framework

Please pay attention to the hints and explanations on the following slides.

PART 3 – DotProduct

Implements a dot product of two vectors, e.g.: $[1\ 2] \cdot [5\ 3] = 11$

The input data encodes the first vector in the key field and the second vector in the value field:

KEY	VALUE
1	5
2	3

I.e., each column represents one vector.

The result should be a single value with key 1. E.g.:

KEY	VALUE
1	11

PART 3 – UserAverageSessionLength

Implement the following SQL-query as a map-reduce job:

```
SELECT userId,AVG(sessionLength) FROM user
GROUP BY userId.
```

The input data is `key=userId, value=sessionLength` [Double-Value]:

KEY	VALUE
3	6.0
3	10.0

You should output `userId,AVG(sessionLength)` [Double-Value]:

KEY	VALUE
3	8.0

PART 3 – UserSessionCount

Implement the following SQL-query as a map-reduce job:

```
SELECT userId,COUNT(sessionLength) FROM user
GROUP BY userId.
```

The input data is `key=userId, value=sessionLength` [Double-Value]:

KEY	VALUE
3	6.0
3	10.0

You should output `userId,COUNT(sessionLength)`:

KEY	VALUE
3	2

PART 3 – WordCounter

Implement a map-reduce Job, that counts how often each word was used in a given text.

The input data is `lineId,text`, e.g.:

KEY	VALUE
0	this is a sample text
1	having fun with text

You should output `word,numOccurrences`:

KEY	VALUE
text	2
this	1

PART 3 – HINTS

- Some examples have already been discussed in the lecture.
- If not otherwise stated, you can assume that `SQLInteger` is used as data type for numbers
- Make sure to produce the correct output format, if in doubt, study the basic tests.

GENERAL REQUIREMENTS & HINTS

- Please document your code!
- You should use the basic Unit-Tests in the package `de.tuda.dmdb.test` to validate your implementation.
- Have a look at the tests to also learn how the interfaces are used and how different components relate to each other.
- It is recommended to implement further tests to cover all scenarios/edge cases – the initially provided test cases don't do that.
- **The specification of advanced test cases (i.e., more information about what the tests do) will be published around 1 week after the lab announcement.**

GENERAL REQUIREMENTS & HINTS

- Have a look at the Abstract and Base classes to see what functions can be reused and to get some inspiration.
- **Feel free to add additional helper methods in the classes** to structure your code and make it modular.
- **In general, you do not need to add additional classes/files**! This either means we have forgotten something (=> pls. come and talk to us to fix it for everybody) or you might need to have another look at the code base
- Use meaningful standard JAVA exception classes to throw exceptions (e.g., IllegalArgumentException, RuntimeException)

GENERAL REQUIREMENTS & HINTS

Reference environment:

- Java 11 (we use openjdk11)
- Junit 5
- Gradle 6.7

EXERCISE SUBMISSION PROCESS

1. Fork project from `sdms_ws2020_students` group in GitLab
   (this will create a private copy that no other student will see)
2. Pull code from server to your Laptop
3. Add your Matriculation number to the `User.txt` file
4. Implement and test code (locally)
5. (Optional) Pull changes from the upstream repository
6. Each push to master branch triggers tests evaluation
   (you can push multiple times, until the deadline)

For more infrastructure details see Lab 0 slides