SCALABLE DATA MANAGEMENT SYSTEMS

TECHNISCHE UNIVERSITÄT DARMSTADT

LAB 2 ANNOUNCEMENT

17. DEC. 2021

AGENDA

1. Announcement of Lab 2
2. Discussion of Lab 1
3. Paper Exercise 7 solution
4. New bonus paper reading

LAB 2 – OVERVIEW

Deadline: 04.02.2022, 09:50am (i.e., before the presentation of the next lab)

Part 1: OLAP

Part 2: Exchange operator and its variants

Part 3: Distributed query processing

BITMAP-INDEXES

(SEE CHEE YONG CHAN, YANNIS E. IOANNIDIS: BITMAP INDEX DESIGN AND EVALUATION. SIGMOD CONFERENCE 1998: 355-366)

A bitmap-index stores a bit-vector B_v for each distinct value v of a single attribute (e.g., 2 values for height)

- Each bit-vector has the size of the table (#of tuples in table)
- The bit at position n of one bit-vector is set to “1” if the tuple at row ID n (=RID) has the value represented by the bit-vector

Bitmap indexes can efficiently support multidimensional queries

- Use one bitmap-index per attribute and distinct value
- Boolean operations to combine bitmap-indexes (AND, OR, NOT) are efficient because 64 bits can be combined in 1 CPU cycle
- SIMD operations allow for even higher parallelism

EXAMPLE: BITMAP-INDEX

Index: D4.brand -> {RID}

RID	D4.id	D4.product	D4.brand	D4.group
0	1	Latitude E6400	Dell	Computers
1	2	Lenovo T61	Lenovo	Computers
2	3	SGH-i600	Samsung	Handheld
3	4	Axim X5	Dell	Handheld
4	5	i900 OMNIA	Samsung	Mobile
5	6	XPERIA X1	Sony	Mobile

Index: D4.group -> {RID}

B_{Dell}	B_{Len}	B_{Sam}	B_{Sony}
1	0	0	0
0	1	0	0
0	0	1	0
1	0	0	0
0	0	1	0
0	0	0	1

Bitmap Index: D4.brand

B_{Com}	B_{Hand}	B_{Mob}
1	0	0
1	0	0
0	1	0
0	1	0
0	0	1
0	0	1

Bitmap Index: D4.group

EXAMPLE: BITMAP-INDEX

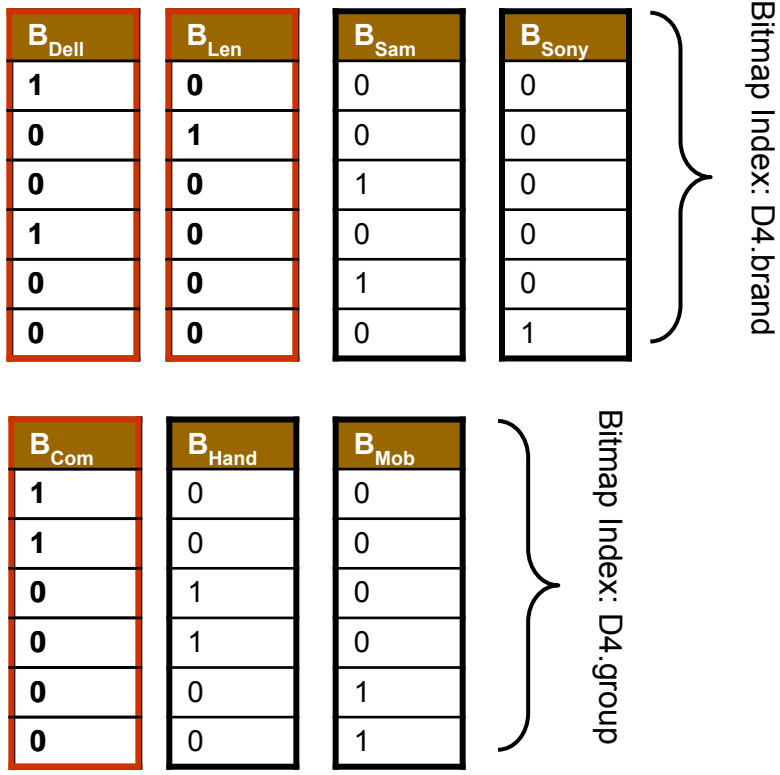
Query:

```
SELECT SUM(F.price)
FROM F, D4 WHERE F.D4 = D4.id
AND D4.group = 'Computer'
AND (D4.brand = 'Dell'
OR D4.brand = 'Lenovo')
```

Index Operation: $B = B_{Com} \wedge (B_{Dell} \vee B_{Len})$

```
B = [110000] ∧ ([100100] ∨
    [010000]) = [110000]
```

=> RIDs 0,1 of D4 need to be read



EXAMPLE: RANGE-ENCODED BITMAP-INDEX

Query: $2011 \leq D2.year \leq 2014$

RIDs: $B_{2011} \wedge \text{NOT } B_{2015} = [1101\ldots1] \wedge [0111\ldots0] = [0101\ldots0]$

RID	D2.year
0	2015
1	2014
2	2010
3	2011
…	…
999	2015

B_{2015}	B_{2014}	B_{2013}	B_{2012}	B_{2011}	B_{2010}
1	1	1	1	1	1
0	1	1	1	1	1
0	0	0	0	0	1
0	0	0	0	1	1
…	…	…	…	…	…
1	1	1	1	1	1

PART 1.1 – NAÏVE BITMAP INDEX

Implement the following methods of the class NaiveBitmapIndex that uses the vanilla/naïve bitmap approach (one bitmap for each distinct value):

- void bulkLoadIndex(TableScanBase tableScan)
  Reads the table of the NaiveBitmapIndex instance and initializes as well as sets all required bitmaps (member variable bitMaps) for the index key column (member variable keyColumnNumber) according to a naïve bitmap approach

- Iterator<RecordIdentifier> rangeLookup(T startKey, T endKey)
  Returns a list of Abstract Records stored in the table of the IndexInstance that fall into the specified range. StartKey and endKey are AbstractSQLValue instances that specify the lower- and upper bound of a range.
  A record falls into the range if startKey <= recordKey <= endKey

PART 1.2 – RANGE ENCODED BITMAP INDEX

Implement the following methods of the class RangeEncodedBitmapIndex that uses the range encoded bitmap index approach (no compression):

- void bulkLoadIndex(TableScanBase tableScan)
  Reads the table of the RangeEncodedBitmapIndex instance and initializes as well as sets all required bitmaps (member variable bitMaps) for the index key column (member variable keyColumnNumber) according to a naïve bitmap approach

- Iterator<RecordIdentifier> rangeLookup(T startKey, T endKey)
  Returns a list of Abstract Records are stored in the table of the IndexInsance that fall into the specified range. StartKey and endKey are AbstractSQLValue instances that specify the lower- and upper bound of a range.
  A record falls into the range if startKey <= recordKey <= endKey

REMARKS & HINTS

- Pay attention to the member variables and methods inherited from the super classes
- The class AbstractTable provides a mapping of rowNum to RecordIdentifiers (PageNumber, SlotNumber) (member variable recordIDMapping), along with helper methods to retrieve the Record Identifier for a given rowID
- The lookup results should be returned in table order (= insertion order).
- Have look at the JavaDoc for Java BitSets since they are used as bitmaps: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/BitSet.html (it contains information about setting bits, supported bit-operations, looping etc.)

PART 1.3: OLAP – INDEXES

Implement the following physical SQL-Operators as Volcano-Iterators by implementing the open(), next() and close() method for the following classes

- IndexScan (point lookup)
- RangeIndexScan
- GroupByAggregate
- Rollup

} Easy to implement, mostly wrappers around the index

found in the package de.tuda.dmdb.sql.operator.exercise.

All next() methods return the next element according to the Iterator model, otherwise (no further element to process) *null* should be returned. The open() and close() methods initialize/clean-up by calling open()/close() on the child(ren) and/or initializing member variables.

You can find some further hints and remarks on the following slides.

PART 1.3: OLAP OPERATORS

Recap **Volcano-based Iterator** Model:

Each operator implements the **following interface**

- **open():** Reset internal state and prepare to deliver first tuple
- **next():** Deliver next result tuple or indicate EOF
- **close():** Release internal data structures, I/O channels, locks, etc.

Evaluation is driven by the top-most operator which receives open(), next(), next(), ... calls from DBMS and propagates

REMARKS & HINTS

Hints **Scan** Operators:

- The B+Tree implementation is already provided in the code-base => the scan-operators can be implemented without implementing the Bitmap-Indices.

REMARKS & HINTS

Hints **GroupByAggregate** Operator:

The grouping is performed based on the attributes defined in the `groupByAttributes` list. You have to think of a mechanism to group the records and compute the aggregates per group.

Aggregates are computed using the passed function(s) in the `aggregateFunctions` list. Each aggregateFunction is a mapping: (oldAggregateValue, update) -> newAggregateValue. Have a look at the Java BiFunction functional interface and lambda expressions.

Each aggregate function should be applied on the corresponding attribute/column defined in the `aggregateAttributes` list.

As you can observe in the code, we assume that we only compute aggregates for integer values.

Note that the helper methods `append(…)` and `keepValues(…)` in the `AbstractRecord` class work in place and use references instead of copying. You might want to call `clone()` on a record and then apply these functions to create a new modified record.

Make sure to return the correct record format in the next method! (see source code)

SQL ROLLUP

Rollup: group-by for all combinations along a hierarchy

Example (IBM DB2):

```
select sum(revenue) as total, region, country, city
from Sales s, Customer c
where s.custKey = c.custKey
group by rollup(region, country, city)
```

Query **groups-by following combinations**:
(), (region), (region, country) and (region, country, city)

EXAMPLE: ROLLUP

We do not expect a particular order in the tests

region	country	city	total	
NULL	NULL	NULL	3.234.343	()
Europe	NULL	NULL	866.323	(region)
USA	NULL	NULL	2.368.020	
Europe	France	NULL	232.199	
Europe	Germany	NULL	634.124	(region, country)
…	…	NULL	…	
Europe	Germany	Munich	119.566	
Europe	Germany	Mannheim	35.234	
…	…	…	…	(region, country, city)
Europe	France	Paris		
…	…	…		

Our tests assume that the aggregate value is at the end, and that columns are ordered

REMARKS & HINTS

Hints **Rollup** Operator:

This operator is similar to the GroupBy-Operator (see hints of GroupBy-Opertor)
In addition:

- The member `combinationFunctions` defines how to aggregate intermediate aggregated values to create the hierarchies
- Additional helper methods might be useful to structure and re-use code
- The AbstractRecord class implements the comparable interface
- There is a special SQLNull value-type that you should use

Make sure to return the correct record format in the *next* method!

Attention: member function `keepValues` from `Record` does does not clone the `AbstractSQLValues` but only copies the references.

PART 2: EXCHANGE OPERATOR

Implement the following physical SQL-Operators as Volcano-Iterators by implementing the open(), next() and close() method for the following classes

- Send
- Receive
- HashRepartitionExchange
- GatherExchange
- ReplicationExchange

found in the package de.tuda.dmdb.sql.operator.exercise.

All next() methods return the next element according to the Iterator model, otherwise (no further element to process) *null* should be returned. The open() and close() methods initialize/clean-up by calling open()/close() on the child(ren) and/or initializing member variables.

You can find some further hints and remarks on the following slides.

PART 2: EXCHANGE OPERATOR

The Exchange (aka shuffle) operator is a meta-operator that is used to encapsulate parallelism

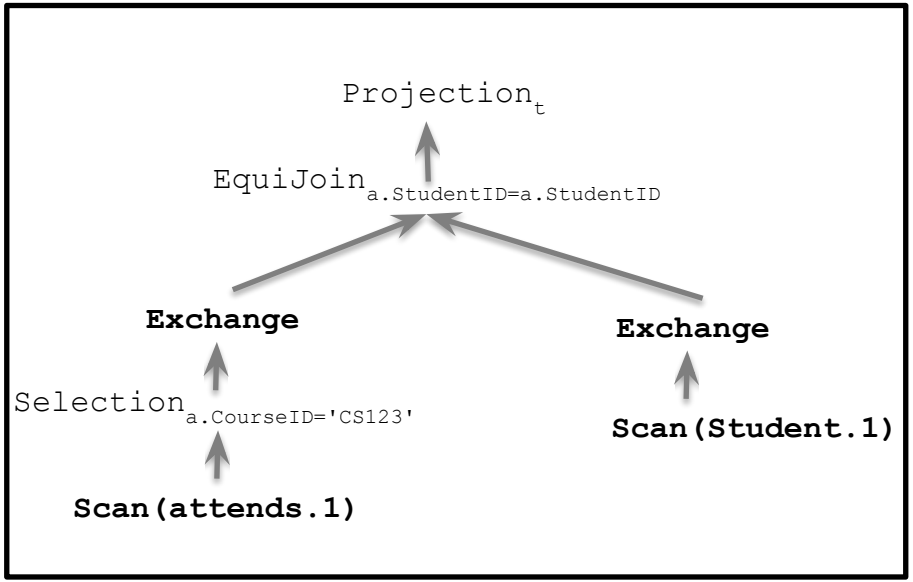
It implements the iterator interface.

It is injected into single-node/threads to enable parallelism:

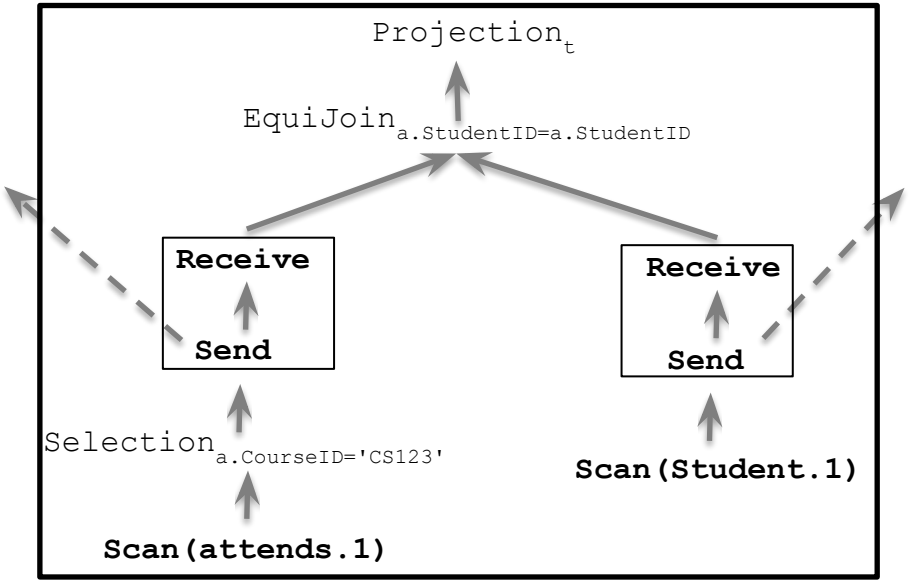
- A query operator plan is initiated for each thread
- The local plans are connect through the exchange operators transparently
  (it starts required threads/connections and buffers data)
- The basic relational operators can remain (largely) unchanged

EXCHANGE: WRAPS SEND/RECEIVE

The Exchange operator encapsulates the SEND- and RECEIVE-Operators



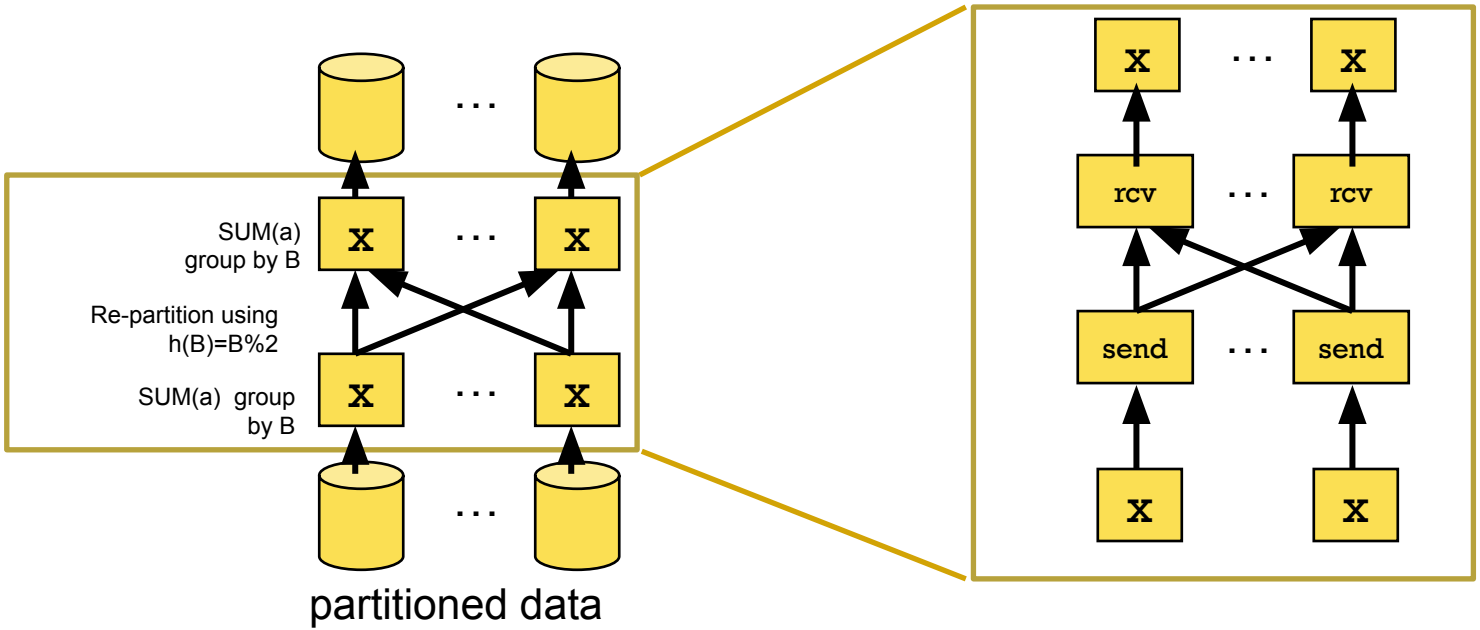
Node 1



Node 1

EXCHANGE FOR PARALLEL/DISTRIBUTED ALGORITHMS

Repartitioning, replication etc. for parallel/distributed processing is achieved with the exchange (send/receive) operators



NETWORK LAYER IN DMDB

- The Network Layer is implemented by classes in package de.tuda.dmdb.net
- A client-server communication pattern is implemented:

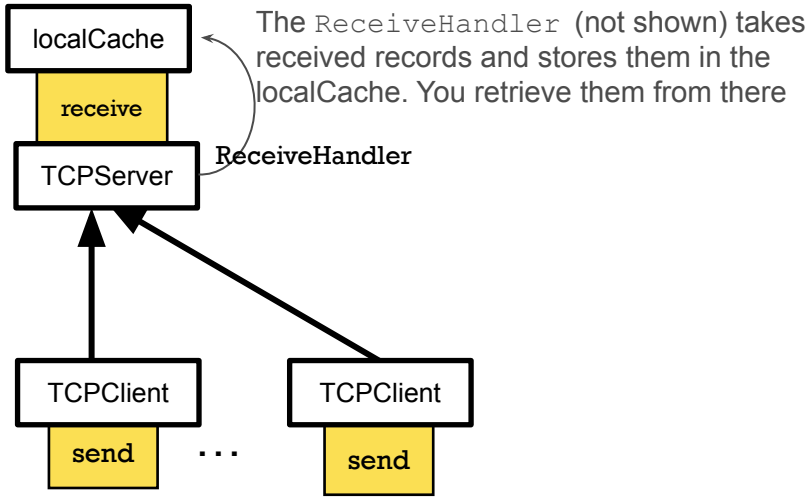
`TCPServer`:
  - Listens for incoming connections and dispatches client connections to a Thread pool
  - The server-component only provides an in-bound channel to receive records, nothing is sent back (implemented by the `ReceiveHandler`)
  - The server-component keeps looking for new records until the **connection is closed actively by a client**

`TCPClient`:
  - Connects to a specified server
  - The client-component only provides an out-bound channel to send records

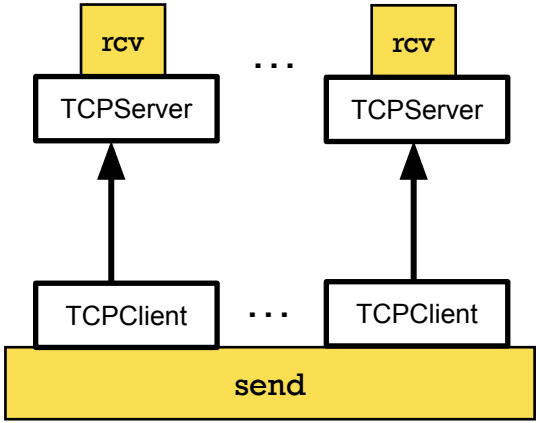
- Only a one-way communication from client to server is implemented/supported

USING THE NETWORK LAYER FOR SEND/RECEIVE

Receive-Operator uses the TCPServer (member variable `receiveServer`) to listen for incoming connections from multiple clients



Send-Operator uses (multiple) TCPClients (member variable `connectionMap`) to send records to different TCPServers (peers)



PART 2: HINTS FOR SEND/RECEIVE

- Pay attention to the member variables inherited from the super classes
- Make sure to **also** set-up/tear-down the network communication in the open(), close() methods (this involves freeing I/O resources). Especially in the RECEIVE-Operator, the order of the performed actions is important.
- The `TCPServer` in the RECEIVE-Operator encapsulates multiple threads (thread pool), pay attention to multi-threading were required (when accessing shared resources)
- The `ReceiveHandler` processes incoming records received by the TCPServer and puts them in the `localCache` of the `ReceiveOperator`. (You need to pass in the reference to the `localCache` when initializing the `TCPServer` for a `ReceiveOperator`)
- Close the connections to peers in the SEND-Operator when all records are processed
- Use the `distributionFunction` in the SEND-Operator to determine to whom a record must be sent. (see next slides)

PART 2: EXCHANGE OPERATOR VARIANTS

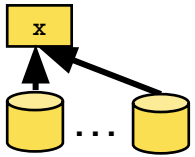
Exchange Operator variants differ in how they distribute data to other nodes/threads.

In DMDB, how data is distributed (i.e. to what nodes/threads) is defined by the `distributionFunction`.

The `distributionFunction` is a mapping function: AbstractRecord -> [list of nodeIds to which to send a record to].

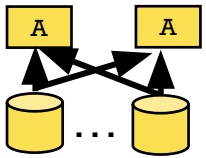
The Exchange Operator variants all define different distribution functions:

All data is **sent only to one** node (`coordinatorId`)



`GatherExchange`

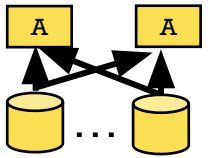
All data is **sent to every other node**



`ReplicationExchange`

A record is **sent to a specific node** as determined by a hash-function (%numReceivers)

Re-partition using h(B)=B%`numReceivers`



`HashRepartitionExchange`

PART 2: HINTS FOR EXCHANGE VARIANTS

- The distribution function of the specific exchange operator must be defined and set in the constructor
- Have a look at the Java Function functional interface and Lambda expressions
- Have a look at the test for the SEND-Operator.

PART 3: DISTRIBUTED QUERIES

Implement the following distributed queries by completing the compilePlan() method in the following classes

- DistributedCountPlan
- FragmentReplicateJoinPlan
- AsymmetricRepartitioningJoinPlan
- SymmetricRepartitioningJoinPlan
- SemiJoinReductionPlan

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

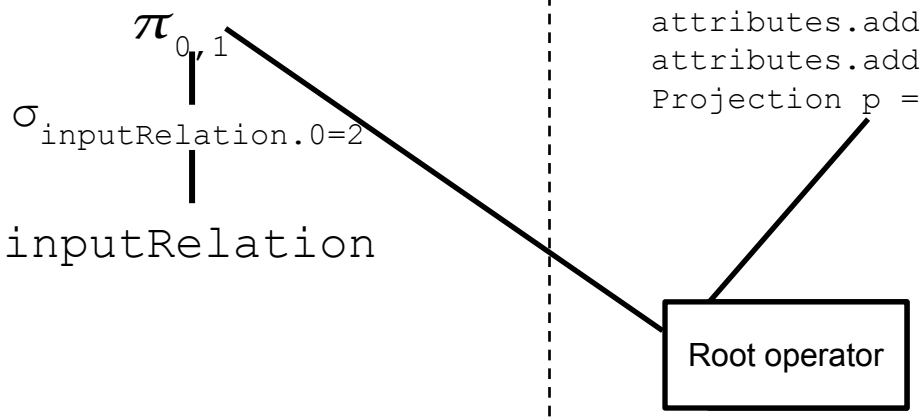
A query is implemented by defining the correct query operator plan that realizes the specific query/strategy. You define a plan by instantiating and chaining the required physical operators (this includes the exchange operator). You have to return the root operator (i.e., the top most operator) of your plan from the method.

You can find some further hints and remarks on the following slides.

EXAMPLE PLAN DEFINITION IN DMDB

Logical plan

$\pi_{0,1}(\sigma_{\text{inputRelation.0=2}}($ inputRelation$))$



DMDB code base

```
TableScan ts = new TableScan(inputRelation);
Selection s = new Selection(ts, 0, new SQLInteger(2));
Vector<Integer> attributes = new Vector<Integer>();
attributes.add(0);
attributes.add(1);
Projection p = new Projection(s, attributes);
```

PART 3: DISTRIBUTED QUERIES

Recap Semi-Join Reduction:

Idea: Before shipping data from Node 1 to any Node 2, reduce it to the necessary tuples that have a join partner on remote side

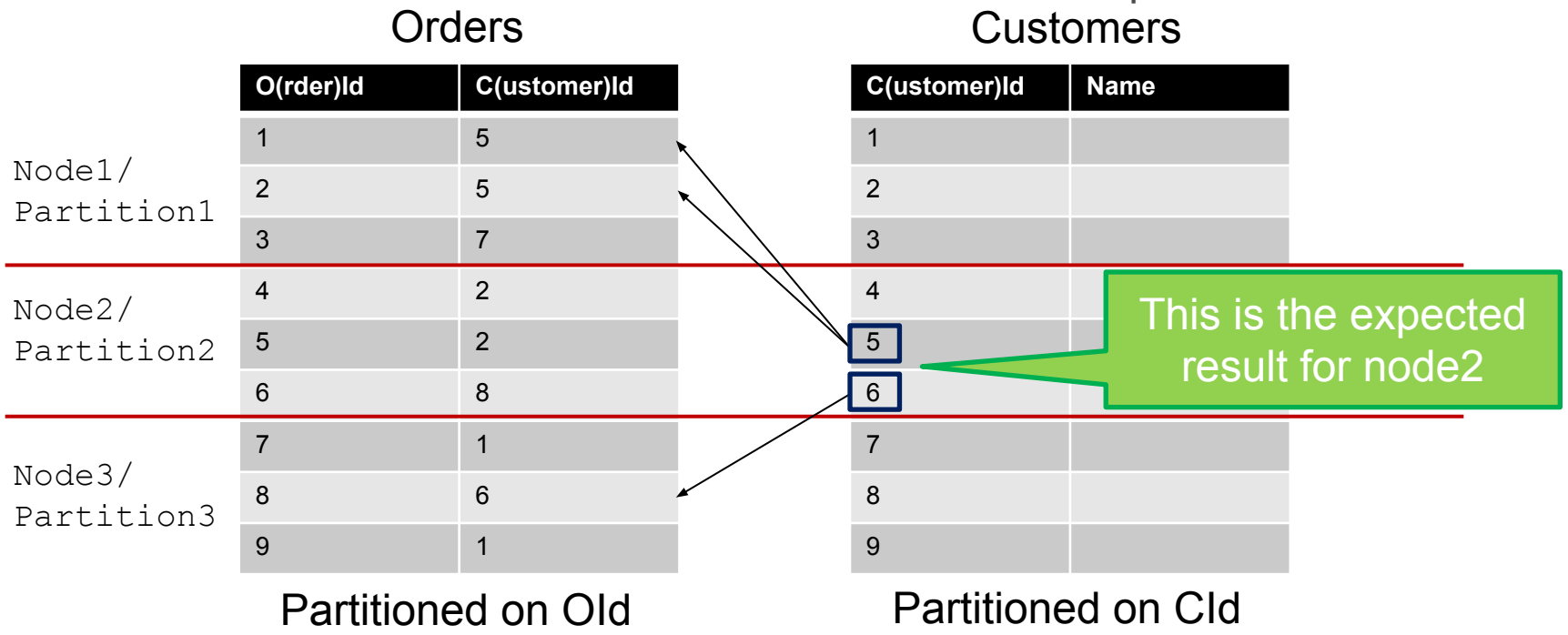
The Semi-Join Reduction in the lecture slides has 5 steps. You can stop at step 3 and return the local result. You have implemented all required basic operators (e.g. projection) to perform the required steps!

PART 3: HINTS

- All required basic operators have been implemented in the previous parts or are already provided in the common-base.jar
- Make sure to use the appropriate operators for the given strategy. E.g. use the SemiHashJoin operator for your semi-join reduction plan.
- `SemiJoinReductionPlan` stops after computing the local Semi-Join result, it assumes a setup where data is replicated
- Some tests only expect you to compute “local results”. Look at test cases to get a deeper understanding ;-)
  - E.g.: The `TestSemiJoinReductionPlan` only tests if you compute the correct Semi-Join result on each node

PART 3: HINTS

- Visualization of simulated `SemiJoinReductionPlan` setup in 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.

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_ws2021_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