SCALABLE DATA MANAGEMENT SYSTEMS

LAB 1 ANNOUNCEMENT

12. NOV. 2021

TECHNISCHE UNIVERSITÄT DARMSTADT

LAB 1 – OVERVIEW

Deadline: 17.12.2020, 09:50am (i.e., before the presentation of the next lab/solution of the last lab)

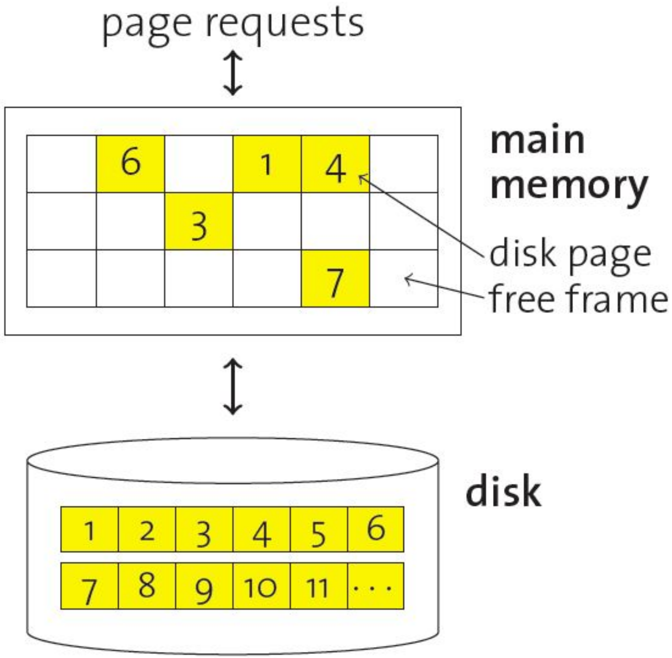
Part 1: Buffer-Management

Part 2: B+-Tree

Part 3: Operators

BUFFER POOL – LECTURE RECAP

Page table



The buffer pool is an in-memory cache of database pages

Page table keeps track of which pages are loaded in memory

One **entry in page** table is called “frame”

Operations on page table:

- PIN (load new page)
- UNPIN (free cached page)

REPLACEMENT STRATEGIES – LECTURE RECAP

If buffer is full when a new page is requested via a PIN operation,
then a victim frame needs to be selected (called page eviction)

Goal is to **evict pages that are not likely to be used in near future**

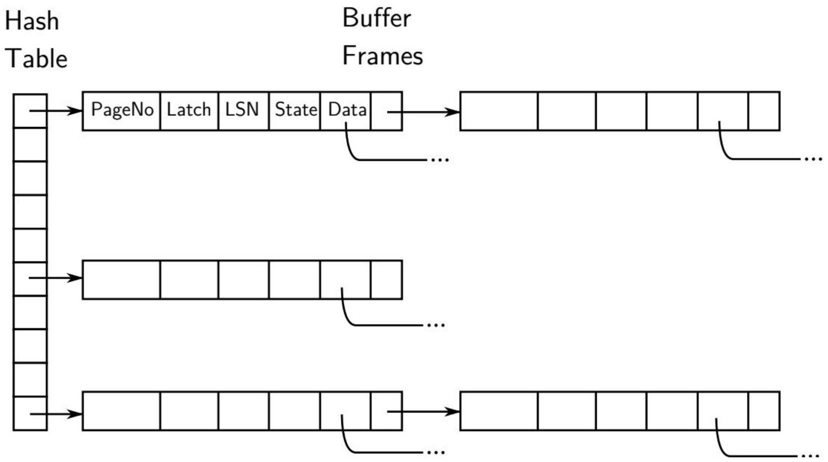
Typical eviction strategies

- LRU / LRU-k
- Clock
- …

The BufferManager should use the `replacer` member to maintain the pinCount of pages (`fix()`/`unfix()`) and to determine pages to be evicted (`evict()`)

BUFFER POOL: PAGE TABLE – LECTURE RECAP

Page table is implemented as a hash table with additional counters for occupied / free slots



- **PageNo:** the page number
- **Latch:** a read/writer lock to protect the page under concurrent access
- **LSN:** log sequence number
- **State:** flag to indicate if page is clean/dirty/newly created etc.
- **Data:** pointer to the actual page in memory

BUFFER POOL: PIN OPERATION – LECTURE RECAP

```
function PIN(pageno)
    if buffer caches page with pageno then
      pinCount(pageno) =  pinCount(pageno) + 1;
      return memory address of frame holding page;

    if buffer is full
      select a victim-frame v using replacement policy;
      if dirty(v) then write v to disk;
    end

    read page with pageno from disk into frame v;
    pinCount(pageno) = 1;
    dirty(pageno)  = false;
    return address of frame v;
```

Page already in buffer?

Page needs to be evicted?

Load page from disk Into buffer

BUFFER POOL: PIN OPERATION – LECTURE RECAP

```
function PIN(pageno)
    if buffer caches page with pageno then
      pinCount(pageno) =  pinCount(pageno) + 1;
      return memory address of frame holding page;

    if buffer is full
      select a victim-frame v using replacement policy;
      if dirty(v) then write v to disk;
    end

    read page with pageno from disk into frame v;
    pinCount(pageno) = 1;
    dirty(pageno)   = false;
    return address of frame v;
```

In our case the pinCount corresponds to, e.g., the refBit of the clock (use fix()/unfix())

Page already in buffer?

Page needs to be evicted?

Load page from disk Into buffer

We don‘t maintain a dirty flag – instead, always write pages out with the diskManager

BUFFER POOL: UNPIN OPERATION – LECTURE RECAP

```
function UNPIN(pageno, dirty)
1.   pinCount(pageno) = pinCount(pageno) - 1;
2.       if(dirty)
3.         dirty(pageno) = true
```

In our case the pinCount corresponds to, e.g., the refBit of the clock (use fix()/unfix())

We don‘t maintain a dirty flag

PART 1: BUFFER MANAGEMENT

Complete the following classes to implement the buffer management in DMDB (see next slides for details):

- BufferManager

- ClockReplacement

- LRUReplacement

PART 1: BUFFER MANAGEMENT

BufferManager (All methods should maintain the buffer's meta-data where required, e.g., how many frames are free (see BufferManagerBase)):

- AbstractPage pin(Integer pageId): Pin a page in the buffer pool and load the page from disk if not already available in the buffer.
- void unpin(Integer pageId): Unpin page in the buffer pool. Unpinned pages are marked for eviction.
- AbstractPage createPage(EnumPageType type, byte[] data): (≊ loadPage) Creates a page with a default page type and from the passed serialized data. The pageId of the page is inferred from the serialized data.
- AbstractPage createDefaultPage(EnumPageType type, int slotSize): Creates a page for a given page type and slot size.

PART 1: BUFFER MANAGEMENT

ClockReplacement (Add any required data structures that you need as member variables):

- fix(Integer pageId): Fixes a frame, indicating that it should not be evicted until it is unfixed. (Think: set a page's refBit=1.)
- unfix(Integer pageId): Unfixes a frame, indicating that it can now be evicted. (Think: set a page's refBit=0)
- Integer evict(): Determine the page that should be evicted next from the buffer

PART 1: BUFFER MANAGEMENT

LRUReplacement (Add any required data structures that you need as member variables):

- fix(Integer pageId): Fixes a frame, indicating that it should not be evicted until it is unfixed.
- unfix(Integer pageId): Unfixes a frame, indicating that it can now be evicted.
- Integer evict(): Determine the page that should be evicted next from the buffer

There are multiple ways to implement the LRU-strategy, e.g. it is up to you how you determine least recently used pages. But, you should NOT change existing classes (e.g. add new member to the page class).

PART 1: HINTS

- Note that the interface of the DiskManager changed compared to the last lab (see DiskManagerBase) – this should not be a problem
- We also changed the PAGE_SIZE member in AbstractPage to store the page size in bytes instead of kbytes.
- Note that the BufferManager implementation is single-threaded. It can be assumed that when a new page is pinned, old pages are not needed any more.
- We provide a DummyBufferManager that is used in the tests of the following parts, such there is no dependency to your BufferManager implementation

PART 2: B+ TREE VS. B TREE

Inner nodes only store “surrogate keys”=separators to enable search (i.e., inner nodes do not store pointers to tuples)

All **keys of tuples in table are stored on leaf level**

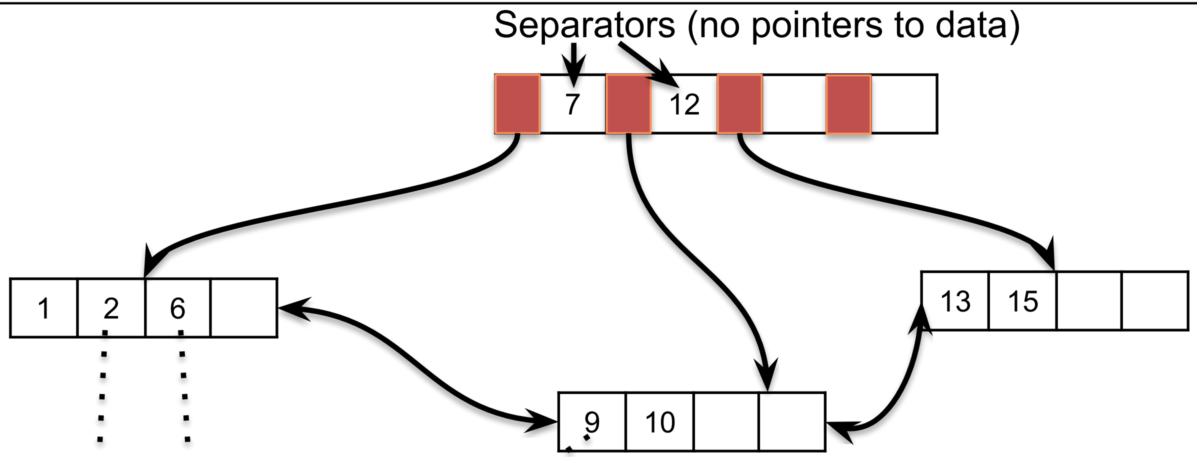
Leaf level is connected as a double-linked list (for range searches)

> *Note: For this project only point lookups are supported, therefore leaves do not need to be linked.*

Advantages over B tree:

- Higher fan-out for inner nodes (for same page size)
- Range searches can be implemented as sequential scan of leaf level

PART 2: B+ TREE EXAMPLE



Index

cid	last_name	first_name	age
6	Binnig	Carsten	33
2	Kemper	Alfons	NULL
9	Eickler	André	NULL
...	...	...	...

Table

PART 2: BASICS

Reference to child node (=Pageno)

Node structure (Inner Node)

C_0	R_0	C_1	R_1	C_2	R_2	C_3	free

Reference key ($R_0<R_1<R_2$)

Index-Entry (nodeRecPrototype)

Reference to a record (=RID)

Node structure (Leaf)

D_0	K_0	D_1	K_1	D_2	K_2	D_3	free

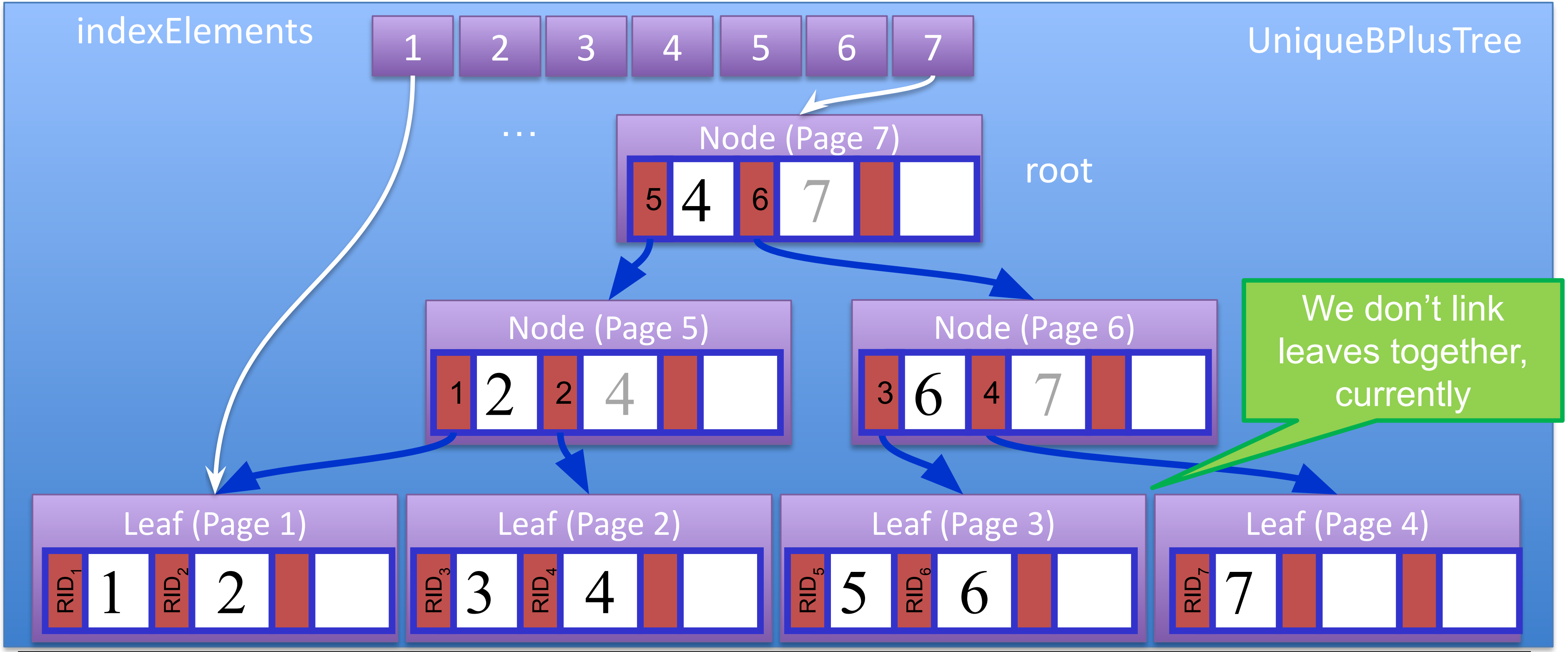
Key of a record($K_0<K_1<K_2<K_3$)

Index-Entry (leafRecPrototype)

C_0 references a subtree with values $\leq R_0$

C_1 references a subtree with values $> R_0$ and $\leq R_1$

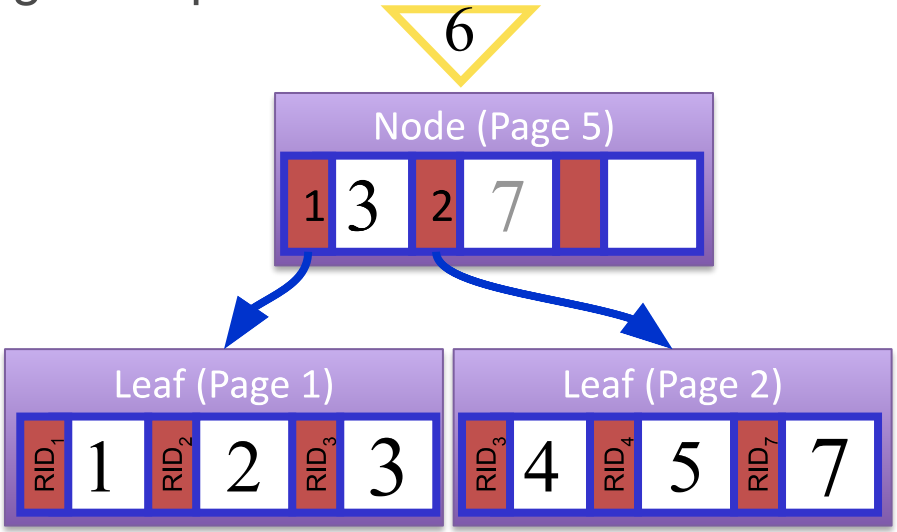
PART 2: EXPLANATION



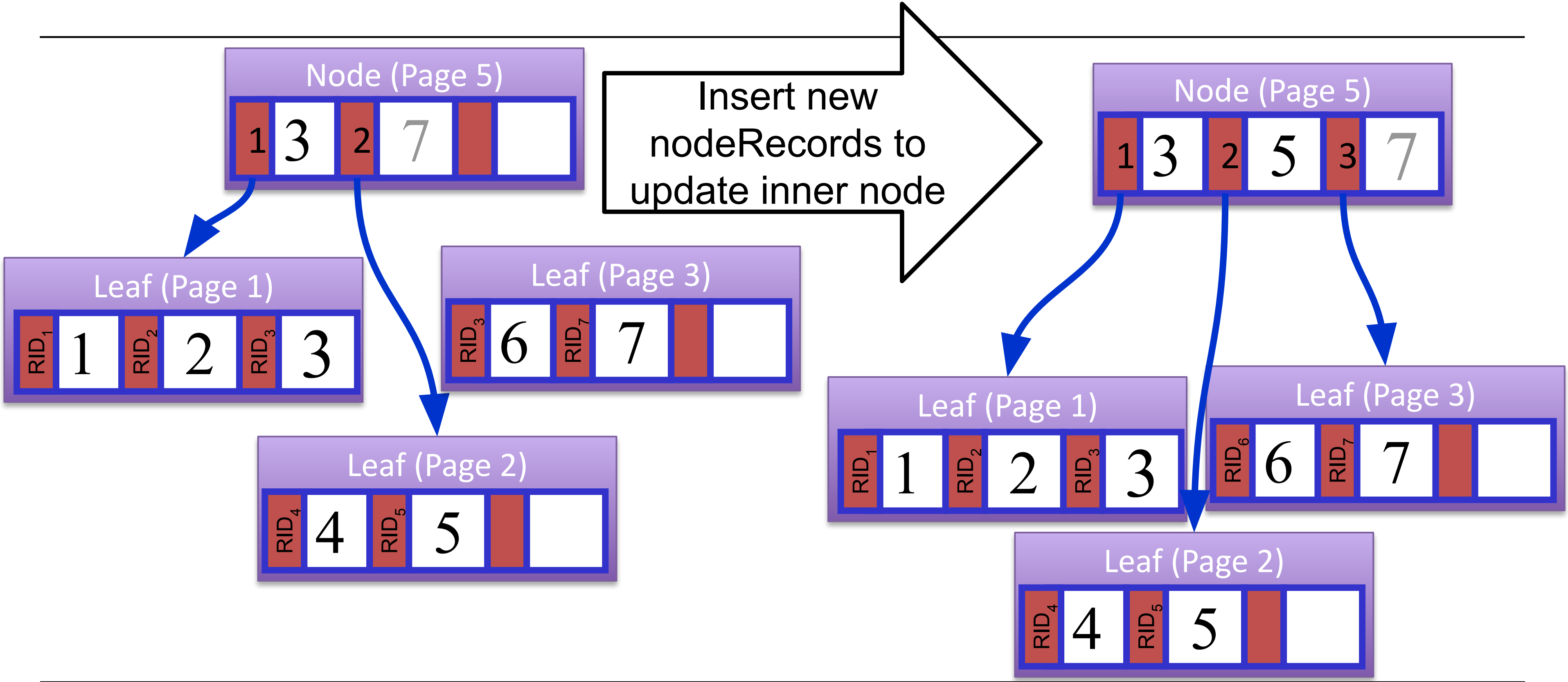
PART 2: EXPLANATION

Remarks/Recap on index re-organization when inserting:

- New records are always inserted at Leaf Level
- Splits are propagated upwards



PART 2: EXPLANATION



PART 2: EXPLANATION

The B+-Tree is implemented through the JAVA Class UniqueBPlusTree.

The UniqueBPlusTree uses the following index element classes:

- Node, used to represent the tree's inner nodes
- Leaf, used to represent the tree's leaf nodes

Have a look at the member variables and helper functions in the base classes, they will help you implement the solution. Some remarks are given on the next slide.

PART 2: B+ TREE INDEX

a) Implement the **insertion of records** into the UniqueBPlusIndex by implementing the insert method of the classes

- Leaf
- Node
- UniqueBPlusTree

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

All methods return true if the insertion was successful, otherwise false.

PART 2: B+ TREE INDEX

b) Implement the **lookup of records** in the UniqueBPlusIndex by implementing the lookup method of the classes

- Leaf
- Node
- UniqueBPlusTree

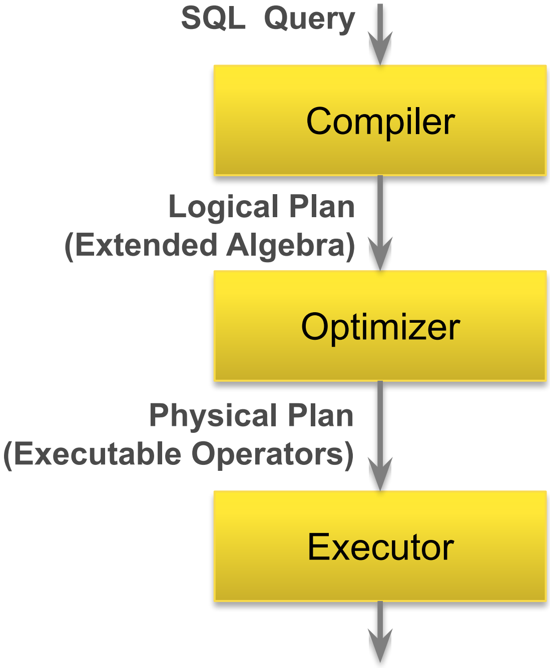
found in the package de.tuda.dmdb.access.exercise

The methods find a record based on its key and return it. If the record does not exist, NULL is returned.

PART 2: HINTS

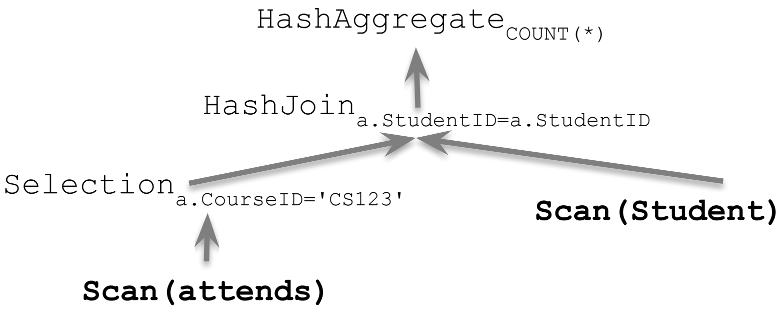
- Each instance of the class Leaf or Node corresponds to a Page, and inherit a function getIndexPage() from the base class AbstractIndexElement, which is used to access the Page.
- All inner nodes and leaves of the UniqueBPlusTree are managed in a HashMap (attribute indexElements) based on their Pageno. (see helper getIndexElement)!
- The indexElements (nodes & leafs) keep a reference to “their” tree via the inherited member uniqueBPlusTree.
- The class UniqueBTree keeps a reference to an record table through the attribute table.

PART 3: OPERATORS




```
SELECT COUNT(*)
FROM Student s, attends a
WHERE s.StudentID=a.StudentID
AND a.CourseID='CS123'
```

$\chi_{\texttt{COUNT(*)}}(\sigma_{\texttt{a.CourseID='CS123' AND a.StudentID=a.StudentID}}$ (Student X attends))



PART 3: OPERATORS

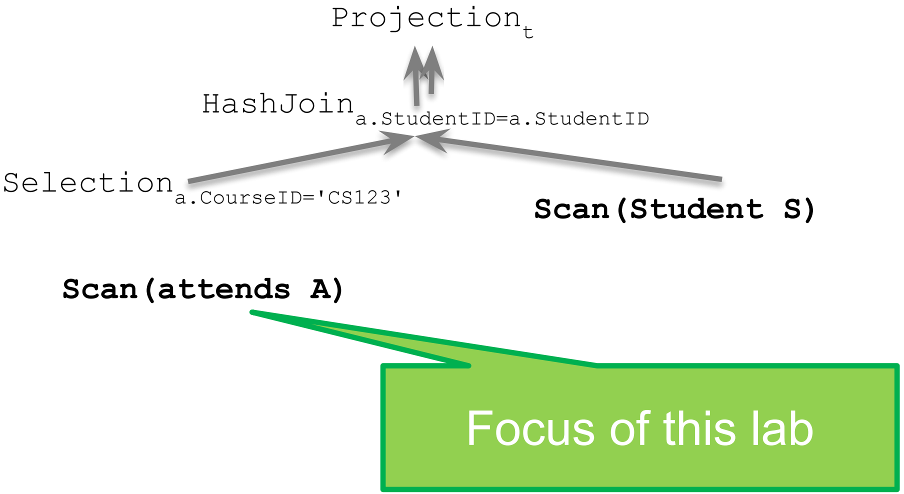
```
for t in child.Next():
  emit(projection(t))
```

```
for t1 in right.Next():
  buildHashTable(t1)
for t2 in left.Next():
  if probe(t2): emit(t1⋈t2)
```

```
for t in child.Next():
  if evalPred(t): emit(t)
```

```
for a in A:
  emit(a)
```

```
for s in S:
  emit(s)
```



PART 3: OPERATORS

We use the **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, locks, etc.

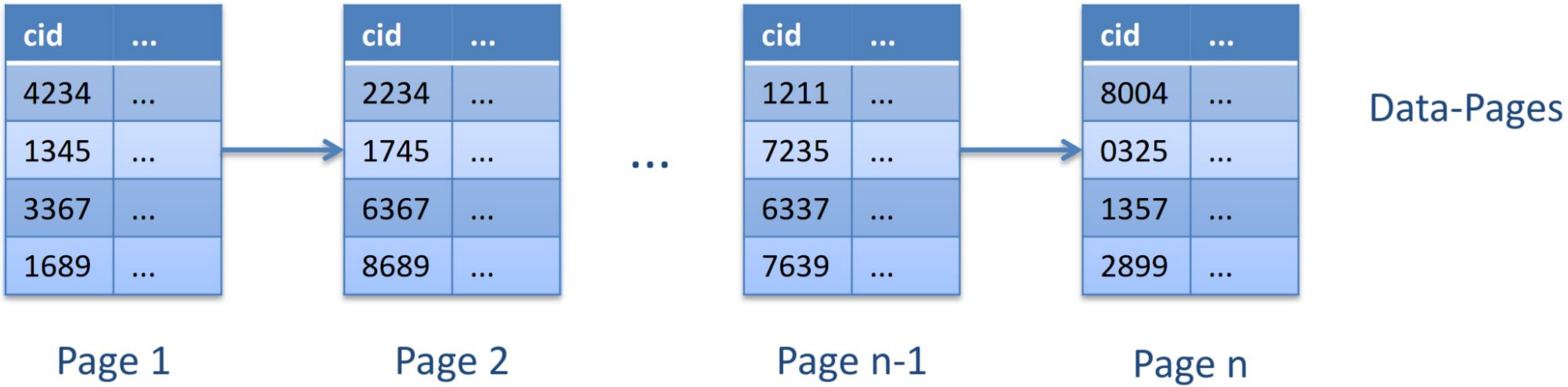
Evaluation is driven by the top-most operator which receives open(), next(), next(), ... calls from DBMS and propagates

PART 3: EXPLANATION HEAPTABLE

Heap Table (aka Heap File) maintain data in random order

Collection of multiple data pages (which, e.g., form a DB-table)

In order to find a record we might need to scan all data pages



PART 3: OPERATORS

Complete the class TableScan to allow DMDB to read an entire Heap-Table.
You have to implement methods open(), next() and close()

Hints:

- The next() method determines the next record and returns – how to determine the next record?
- Once a page has been read entirely it should be unpinned
- The HeapTable maintains an array (Vector) of pages that belong to it – you can use `table.getPageNumber(x)` to get the page at position x of this array
- Once you have the pageNumber you can retrieve the page by using the bufferManager member

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 often means you might need to have another look at the code base
- Use meaningful standard JAVA exception classes to throw exceptions (e.g., IllegalArgumentException, RuntimeException)

EXERCISE SUBMISSION PROCESS

1. Fork project from `sdms_ws2021_students` group in GitLab under your own namespace
2. Pull code from server to your computer
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