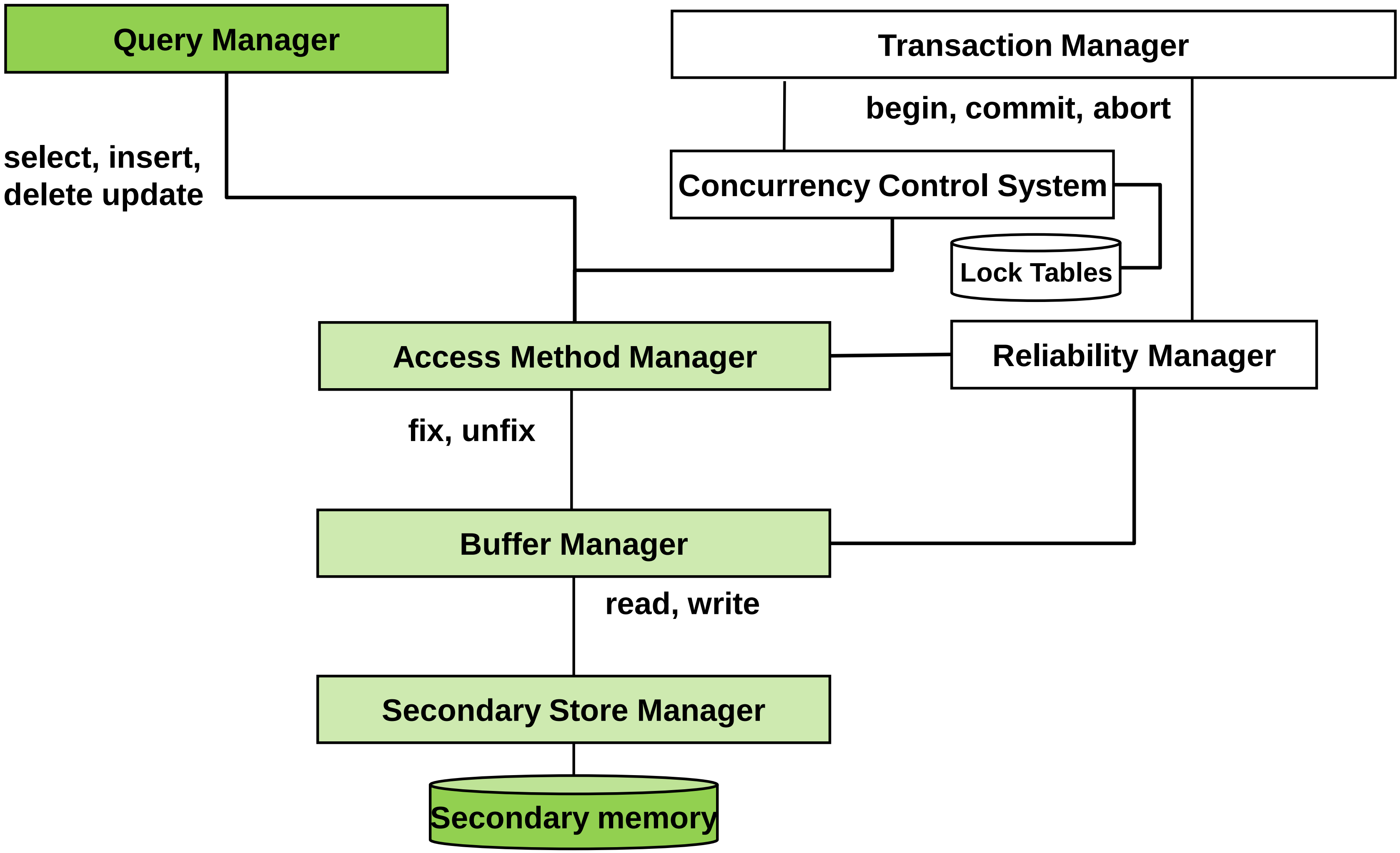
Databases 2

3

Physical data structures and query optimization

Query management in the DBMS architecture

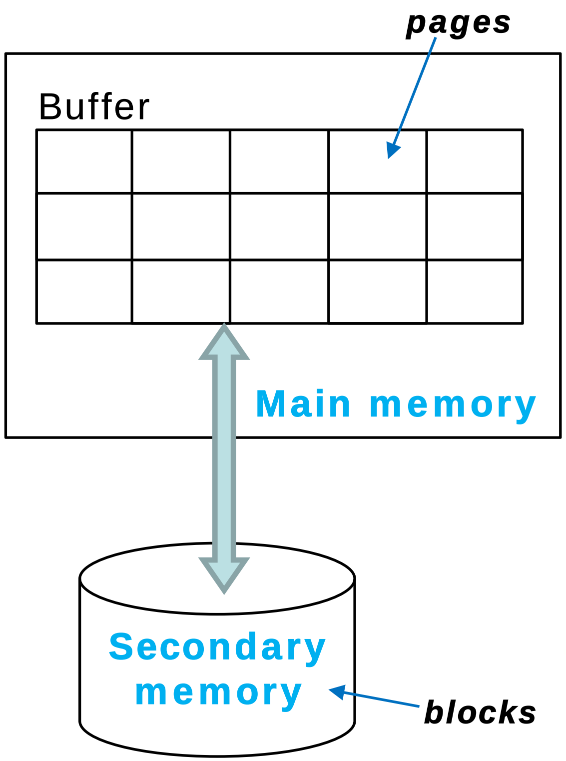


DATA ACCESS and COST MODEL

Main and Secondary memory (1)

- Databases must be stored (mainly) in files onto secondary memory for two reasons:
  - size
  - persistence

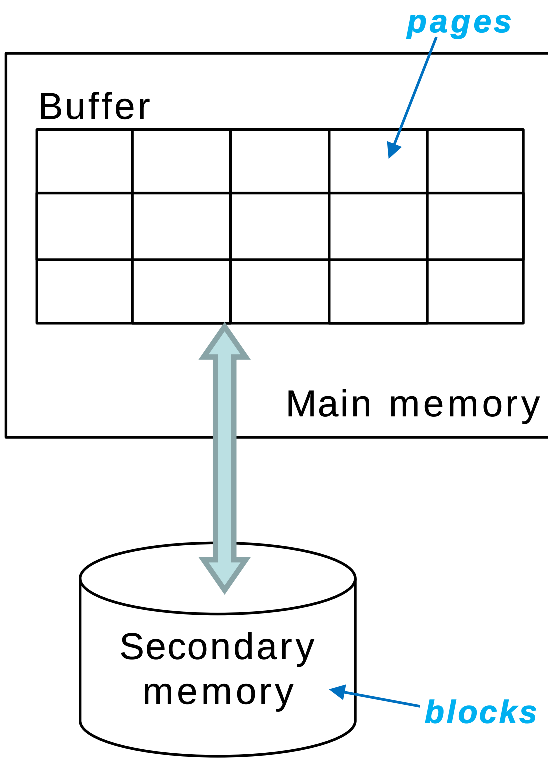
- Data stored in secondary memory can only be used if first transferred to main memory



Main and Secondary memory (2)

- Secondary memory devices are organized in **blocks** of (usually) **fixed** length
  - order of magnitude: a few KBytes

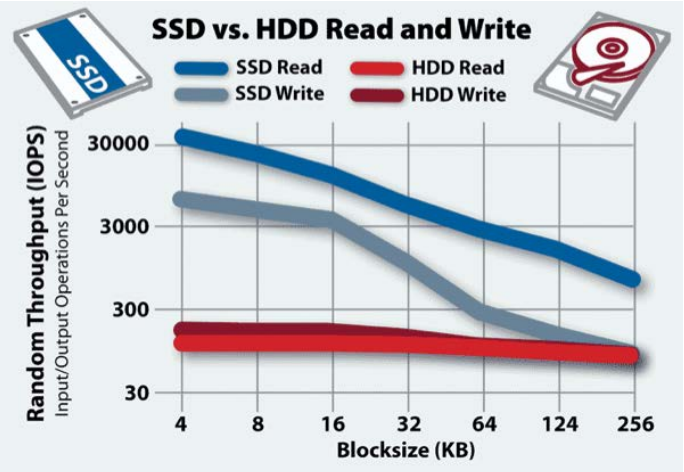
- **I/O operation**: moving a block from secondary to main memory and vice-versa



Main and Secondary memory (3)

- How long does it take to read a **block** from a disk?

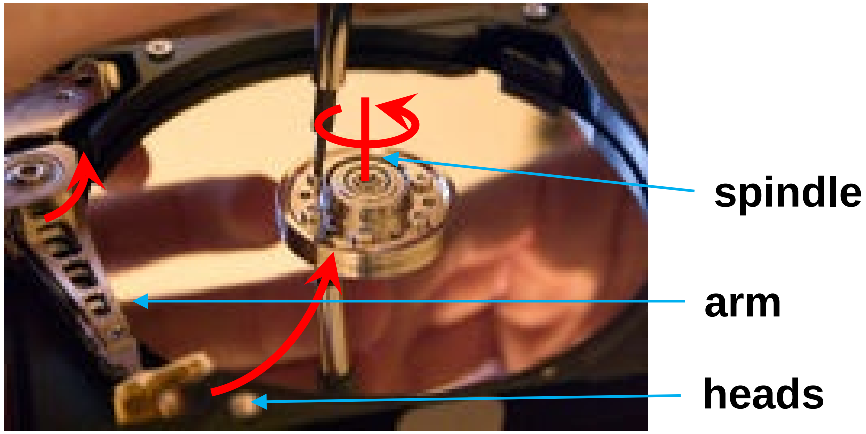
  - It depends on the technology
    - Mechanical hard drives
    - Solid State Drives (SSD)
      - Fast read/write speed

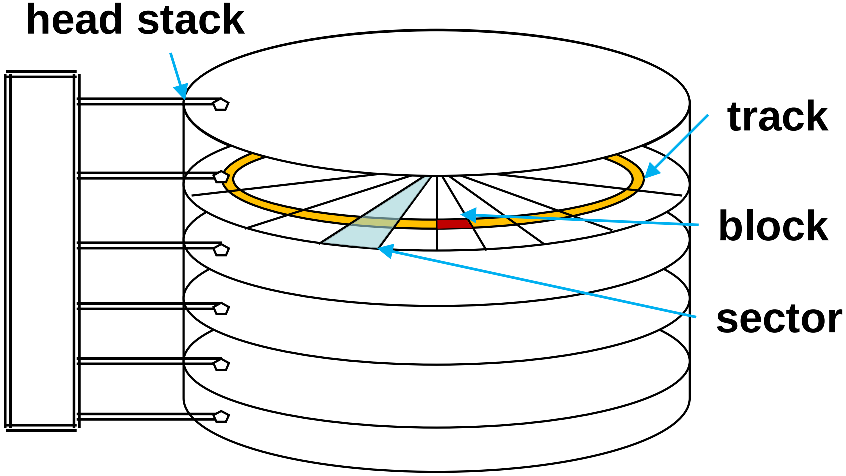


  - SSD are more expensive
    - For very large datasets, cheap storage is the only viable option

A mechanical hard drive («Winchester»)

- Several **disks** piled and rotating at constant angular speed
- A **head stack** mounted onto an arm moves radially in order to reach the **tracks** at varius distances from the rotation axis (**spindle**)
- A particular **sector** is «reached» by waiting for it to pass under one of the heads
- Several **blocks** can be reached at the same time (as many as the number of heads/disks)





→ in depth in the course *Computing Infrastructures, semester 2*

Main vs. Secondary memory access time

- Secondary memory access:
  - **seek time** (8-12ms) - *head positioning*
  - **latency time** (2-8ms) - *disc rotation*
  - **transfer time** (~1ms) - *data transfer*

The cost of an access to secondary memory is **4 orders of magnitude** higher than that to main memory

- In "I/O bound" applications the cost **exclusively** depends on the number of accesses to secondary memory

DBMS and file system

- File System (FS): component of the OS which manages access to secondary memory

- DBMSs make **limited use of FS functionalities**
- The DBMS directly manages the file organization, both in terms of the distribution of records within blocks and with respect to the internal structure of each block

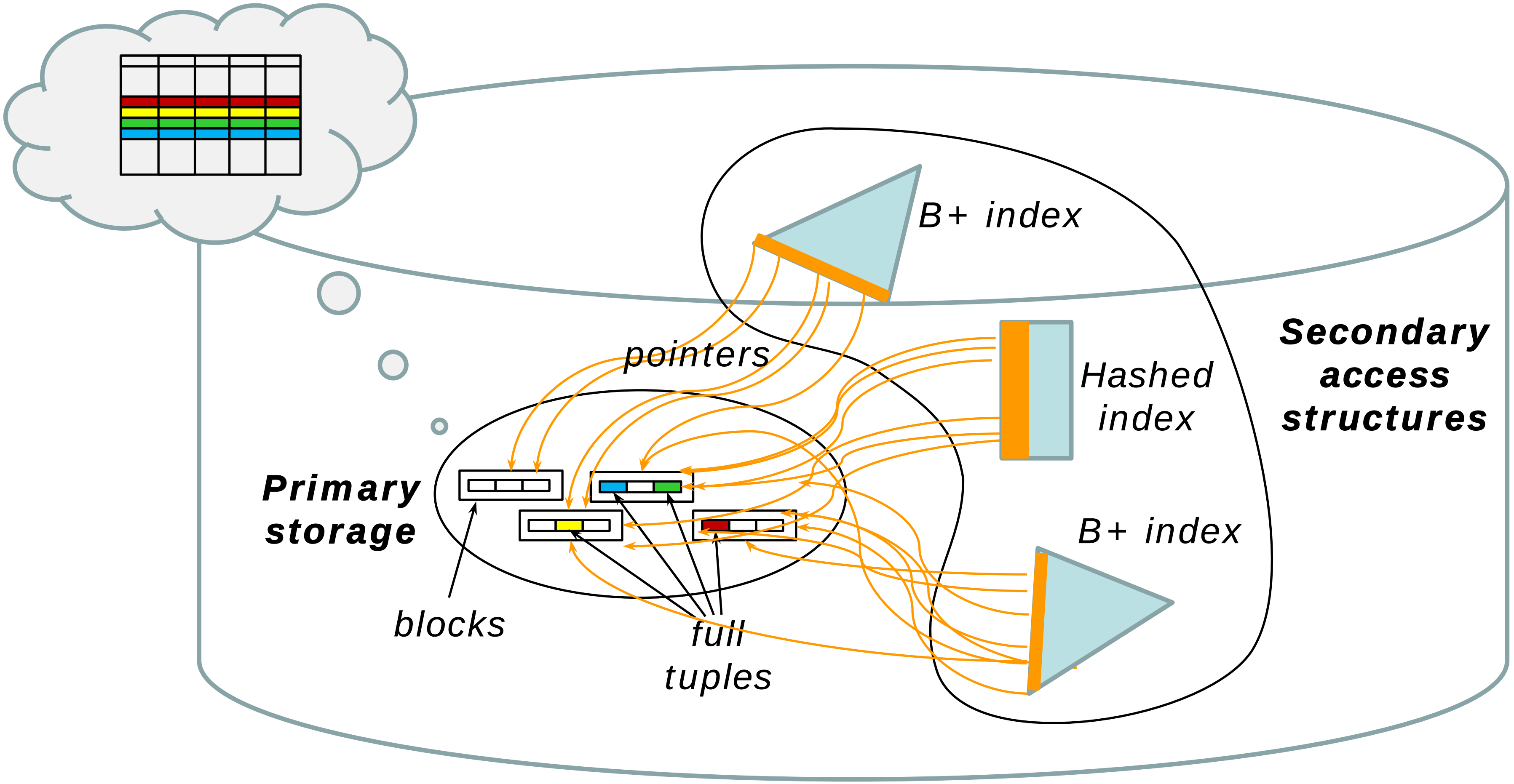
  - A DBMS may also **control the physical allocation of blocks onto the disk** (for faster sequential reads)

PHYSICAL ACCESS STRUCTURES

Physical access structures

- Each DBMS has a distinctive and limited set of access methods
  - ***Access methods***: software modules that provide data access and manipulation (store and retrieve) **primitives** for each physical access structure
- Access methods have their own data structures to organize data
  - Each table is stored into **exactly one** **primary** physical access structure, and
  - may have **one or many optional** **secondary** access structures

Primary and Secondary access structures



Physical access structures

- **Primary** structure: it contains all the tuples of a table
  - Main purpose: to store the table content
- **Secondary** structures: are used to index primary structures, and only contain the values of some fields, interleaved with pointers to the blocks of the primary structure
  - Main purpose: to speed up the search for specific tuples, according to some search criterion
- **Three main types of data access structures**:
  - **Sequential** structures
  - **Hash-based** structures
  - **Tree-based** structures

Physical access structures

- Not all types of structures are equally suited for implementing the primary storage or a secondary access method

	Primary	**Secondary**
Sequential structures	**Typical**	-
Hash-based structures	Frequent	Frequent
Tree-based structures	Frequent	**Typical**

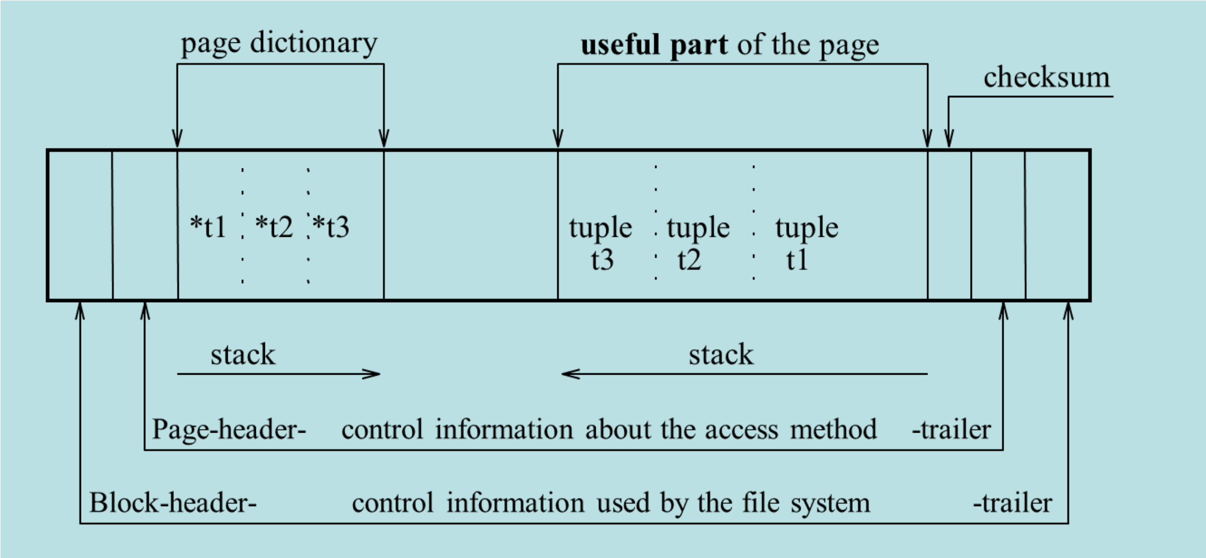
Blocks and tuples

- Blocks: the "physical" components of files
- Tuples: the "logical" components of tables

  - The size of a **block** is typically fixed and depends on the file system and on how the disk is formatted
  - The size of a **tuple** (also called record) depends on the database design and is typically variable within a file
    - optional (possibly `null`) values, `varchar` (or other types of non-fixed size) attributes

Organization of tuples within blocks / pages

Block for sequential and *hash-based* methods



- ***Block header*** and ***trailer*** with control information used by the **file system**
- ***Page header*** and ***trailer*** with control information about the ***access method***
- A ***page dictionary***, which contains pointers (offset table) to each elementary item of useful data contained in the page
- A ***useful part***, which contains the data
  - In general, page dictionaries and useful data grow as stacks in opposite directions
- A ***checksum***, to detect corrupted data

Block Factor

- Block factor **B:** the <u>number of tuples within a block</u>

  - $\mathbf{S_R}$: Average size of a tuple (assuming "fixed length record")
  - $\mathbf{S_B}$: Size of a block
    - if $S_B > S_R$, there may be many tuples in each block:

$$\mathbf{B} = \lfloor \mathbf{S_B} / \mathbf{S_R} \rfloor \qquad \left(\lfloor \mathbf{x} \rfloor = \mathrm{floor}(\mathbf{x}) \right)$$

- The rest of the space can be
  - non used (*unspanned* records)
  - used (records *spanned* between blocks (*hung-up* records))

Page manager primitives

- ***Insertion*** *and* ***update*** *of a tuple*
  - may require a reorganization of the page or
  - usage of a new page
- ***Deletion*** *of a tuple*
  - often carried out by marking the tuple as ‘invalid’
- ***Access*** *to a* ***field*** *of a particular tuple*
  - identified according to an offset w.r.t. the beginning of the tuple and the length of the field itself (stored in the page dictionary)

Sequential structures

- Sequential arrangement of tuples in the secondary memory
- Three cases:

1. ***Entry-sequenced*** organization: sequence of tuples dictated by their order of entry
2. ***Array*** organization: the tuples (all of the same size) are arranged as in an array, they can be accessed through an index
3. ***Sequentially-ordered*** organization: tuples ordered according to the value of a ***key*** (one or more attributes)

1. "Entry-sequenced" sequential structure

- **Optimal** for
  - **space occupancy**, as it uses all the blocks available for files and all the space within the blocks
  - carrying out **sequential** reading and writing
    - Especially if the disk blocks are arranged sequentially
    - Only if all (or most of) the file is to be accessed
- **Non-optimal** with respect to
  - Searching specific data units (may require scanning the whole structure)

2. “Array” sequential structure

- Each tuple has a numerical index *i* and is placed in the *i*-th position of the array
  - indexes are obtained by increasing a counter
- Made of *n* adjacent blocks, each block with *m* slots available to store *m* tuples
  - Stores overall up to $n \times m$ tuples
- Possible only when the tuples are of fixed length

3. "Sequentially-ordered" sequential structure

- Each tuple has a position based on the value of the **key field**

- Main problem: *insertions* of new tuples

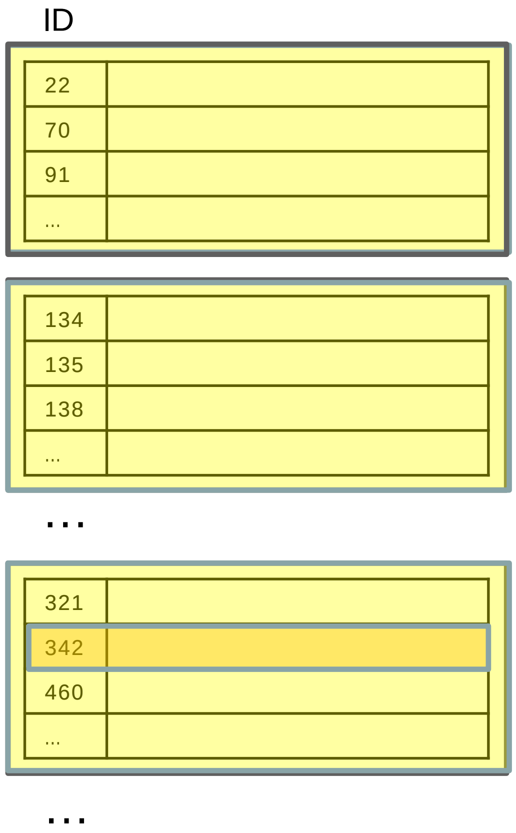
  and *updates that may increase the data size*

  - Reordering techniques for the tuples already present
  - Options to avoid global reordering:
    - Differential files (example: yellow pages)
    - Free slots at the time of first loading → 'local reordering' operations
    - *Overflow file*: new tuples are inserted into blocks linked to form an *overflow chain*

Comparison of sequential structures

	Entry-sequenced	**Array**	**Sequentially-ordered**
INSERT	Efficient	Efficient	**Not efficient**
UPDATE	Efficient (if data size increases → delete+insert the new version)	Efficient	Not efficient if data size increases
DELETE	"Invalid"	"Invalid"	"Invalid"
Tuple size	Fixed or variable	**Fixed**	Fixed or variable

Example of query on a sequential structure



SELECT * FROM Student WHERE Student.ID = ’342’

Hash-based access structures

- Efficient ***associative* access** to data, based on the value of a ***key*** field
- A hash-based structure has N_B ***buckets*** (unit of storage, typically of the size of 1 block - often all adjacent in the file)
- A **hash** function maps the key field to a value between 0 and $N_B - 1$
  - This value is interpreted as the index of a bucket
- Efficient technique for queries with *equality predicates* on the key

Features of hash-based structures

`hash(fileId, Key): BlockId`

- The implementation consists of two parts
  - *folding*, transforms the key values so that they become positive integer values, uniformly distributed over a large range
  - *hashing* transforms the positive binary number into a number between 0 and N_B–1, to identify the right bucket for the tuple

Hash function
h(K) = K mod 10

KEY K=981 ⟶ h(981): bucket 1

Collisions

- When two keys (tuples) are associated with the same bucket
    - E.g., K=983, K=723 with h(K)=K mod 10 → bucket 3
- When the maximum number of tuples per block is exceeded, collision resolution techniques are applied:
    - Closed hashing (open addressing): try to find a slot in another bucket in the hash table (not used in DBMS)
        - A very simple technique is *linear probing:* visit the next bucket, start again from 0 when you reach the end of the table
    - Open hashing (separate chaining):
        - a new bucket is allocated for the same hash result, linked to the previous one

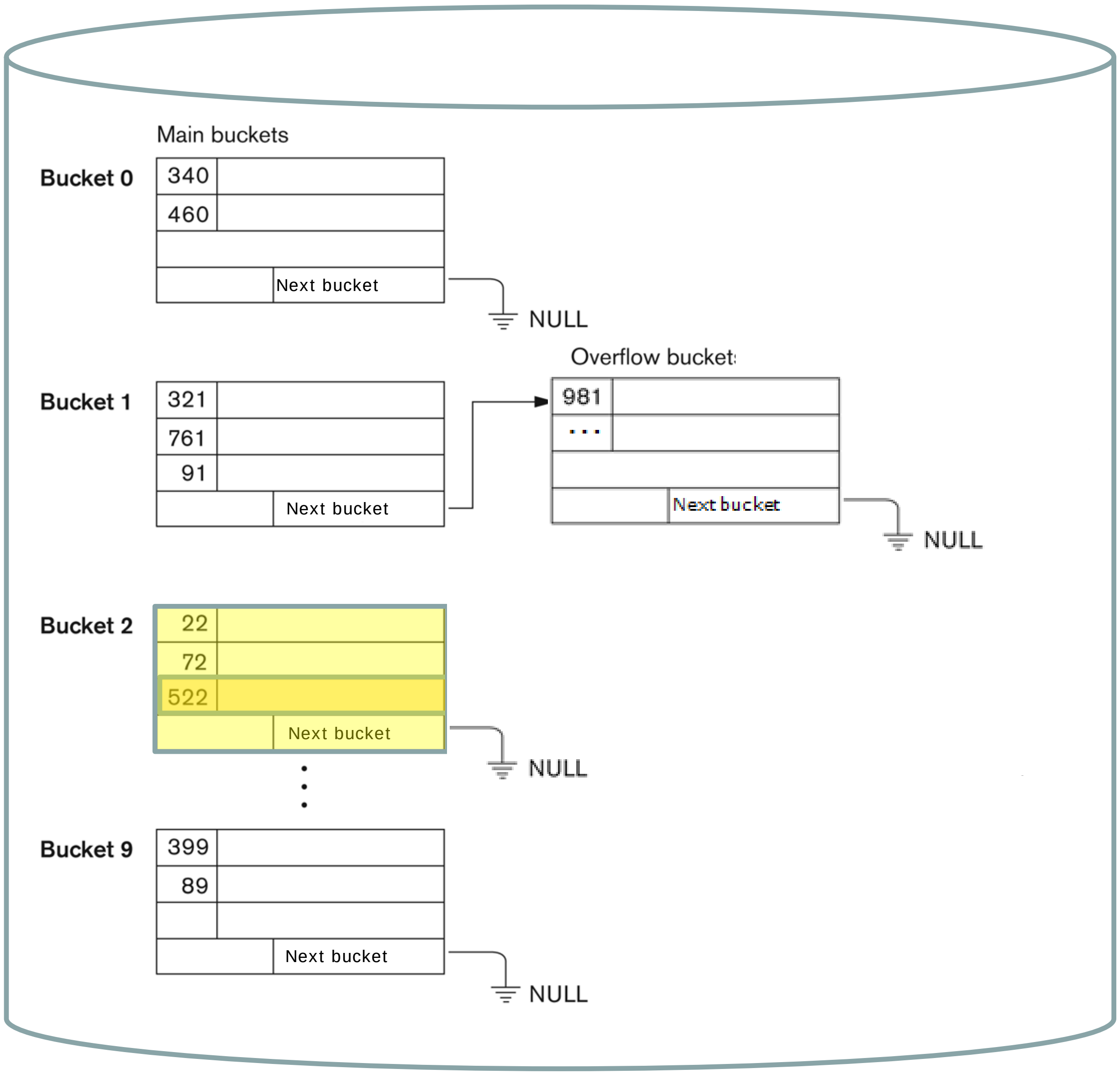
Hash table for primary storage

Hash function
h(K) = K mod 10

```
SELECT *
FROM  Student
WHERE Student.ID = '522'
```

h(522) = 2

#I/O operations = 1



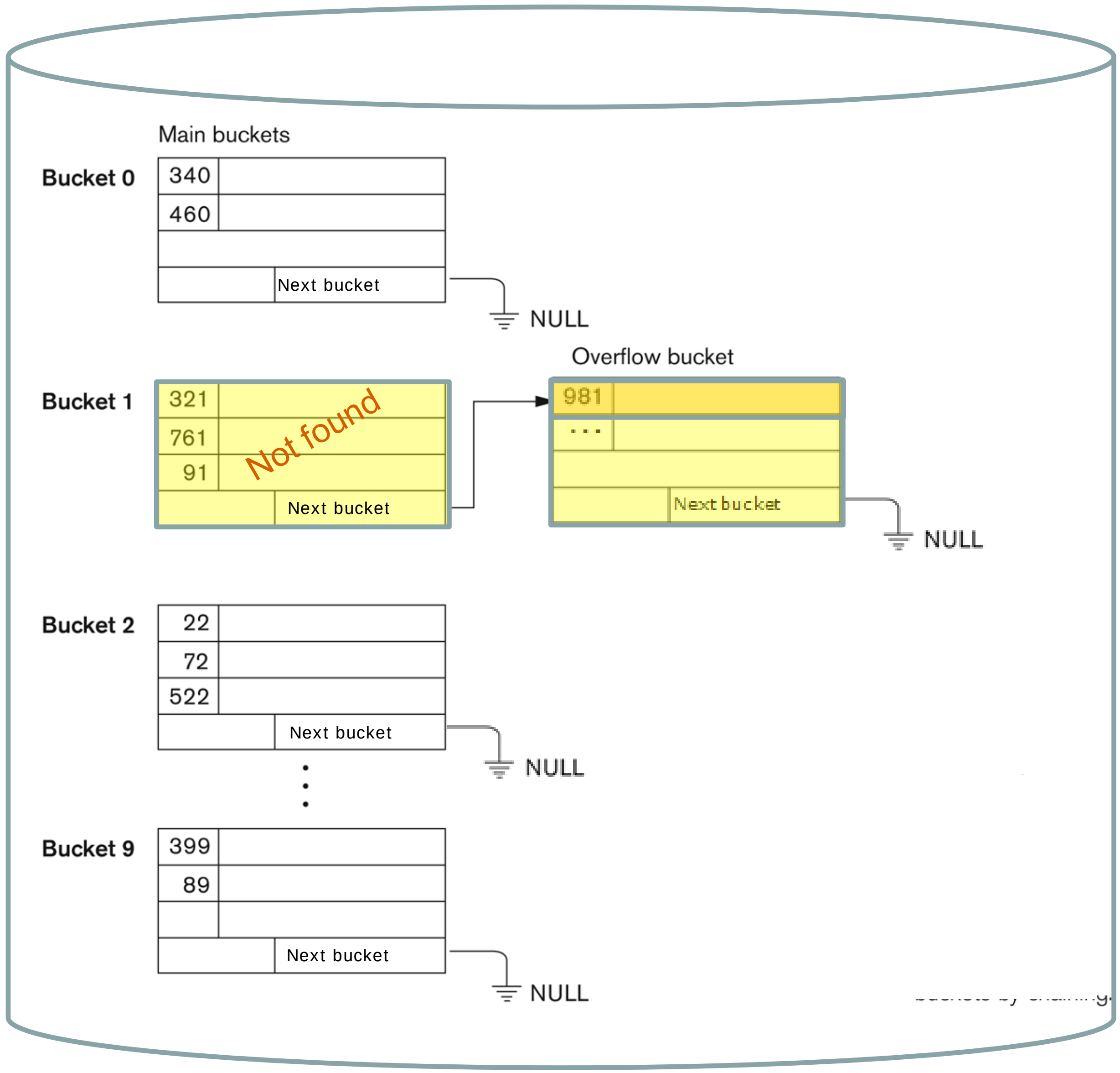
Hash table for primary storage

Hash function
h(K) = K mod 10

SELECT *
FROM Student
WHERE Student.ID = '981'

h(981) = 2

#I/O operations = 2



Hash-based primary storage - collisions

How many accesses are needed to find the tuple corresponding to a given key?

Most of the times 1 access, sometimes 2 accesses or more

⇨ We can estimate the cost of accessing the tuple by considering the <u>average length of the overflow chain</u>

Overflow chains

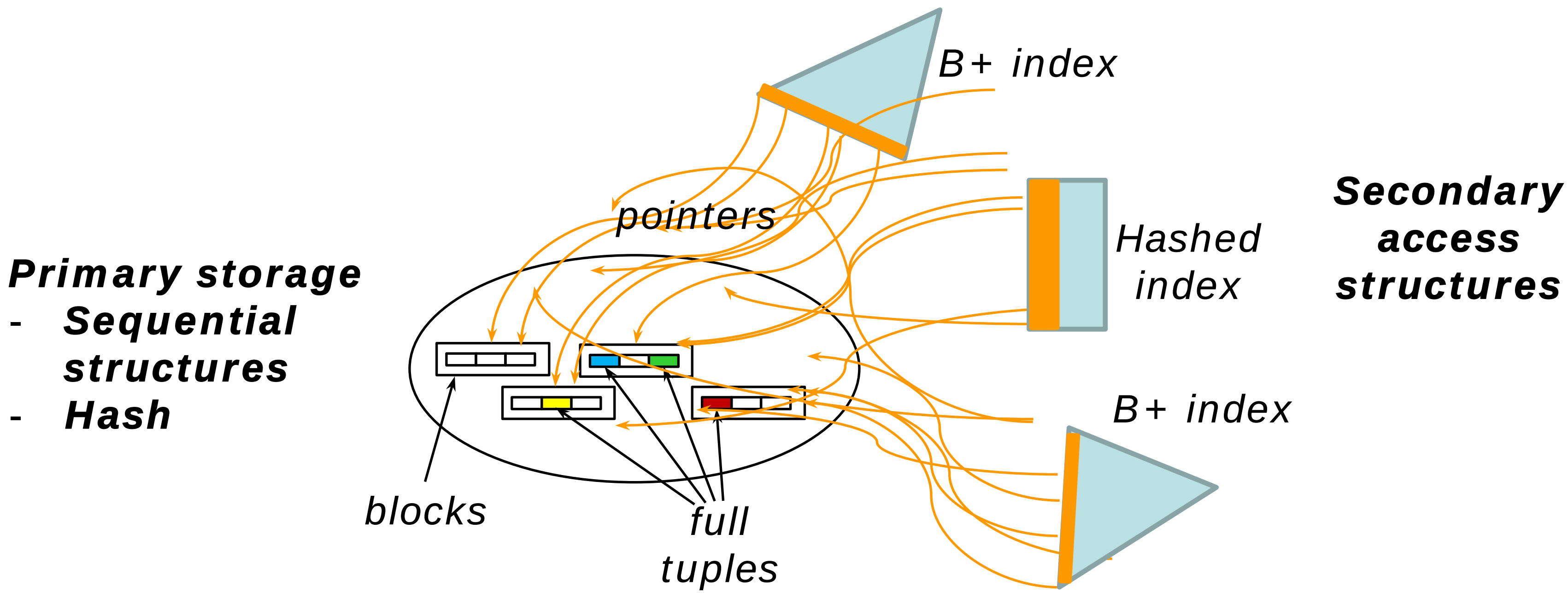
- The **average length of the overflow chain** is a function of
  - the load factor $T/(B \times N_B)$ and
  - the block factor B

$T/(B \times N_B)$ \ B	1	2	3	5	10
.5	0.5	0.177	0.087	0.031	0.005
.6	0.75	0.293	0.158	0.066	0.015
.7	1.167	0.494	0.286	0.136	0.042
.8	2.0	0.903	0.554	0.289	0.110
.9	4.495	2.146	1.377	0.777	0.345

- T is the number of tuples
- N_B is the number of buckets
- B is the number of tuples within a block

Example (B=3, load factor=70%) → #I/O operations = 1.3 (with overflow chains)

Indexes



- Data structures that efficiently retrieve tuples on the basis of a **search key**
- They contain records of the form

search-key	pointer

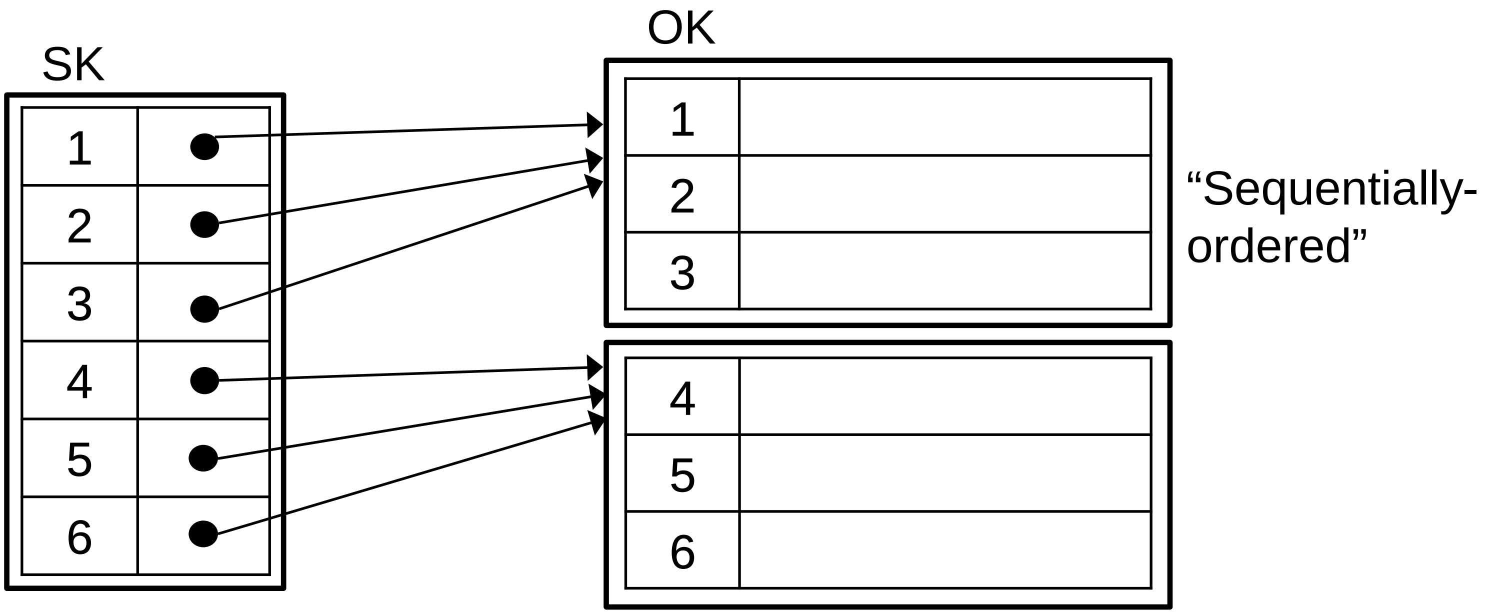
- The index concept: analytic index of a book → pair (term - page number) alphabetically ordered, at the end of a book

Primary index

- In case of “Sequentially-ordered” access structures

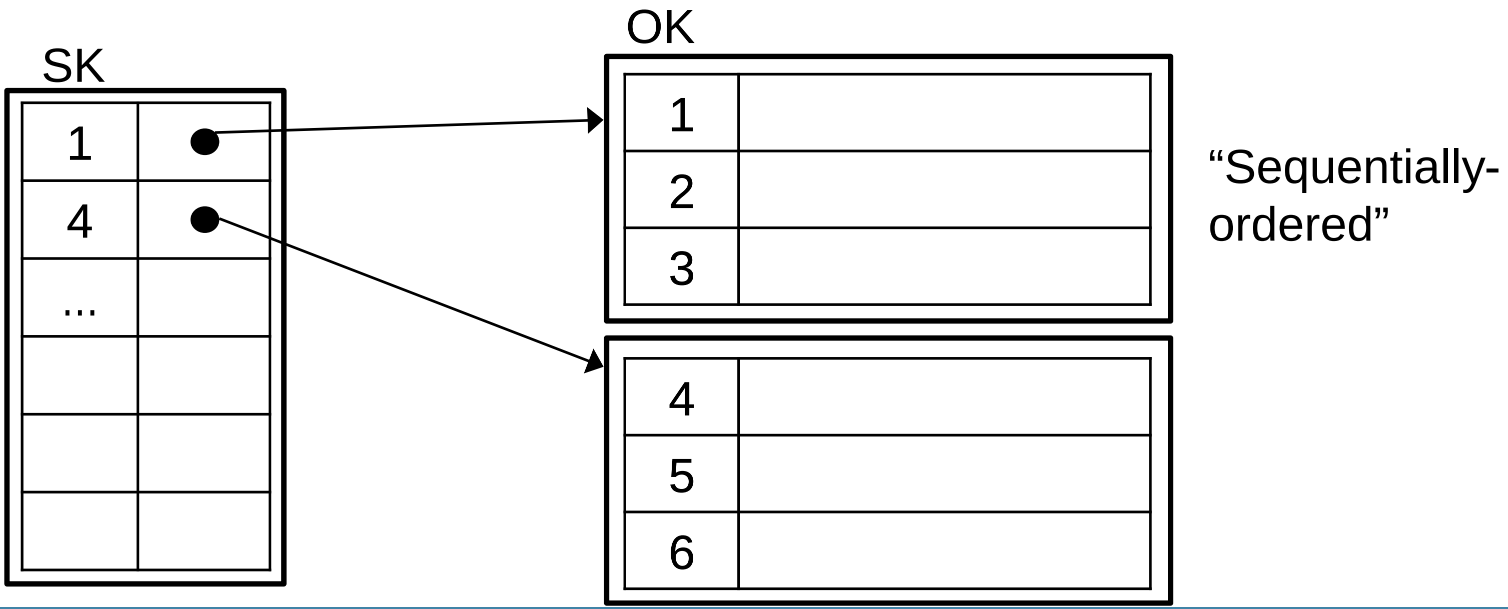
  it is possible to define a **Primary index:**

  - The search key (SK) is the same attribute according to which the structure is ordered (ordering key - OK)
  - Only one primary index can be defined
  - Usually on the primary key, but not necessarily



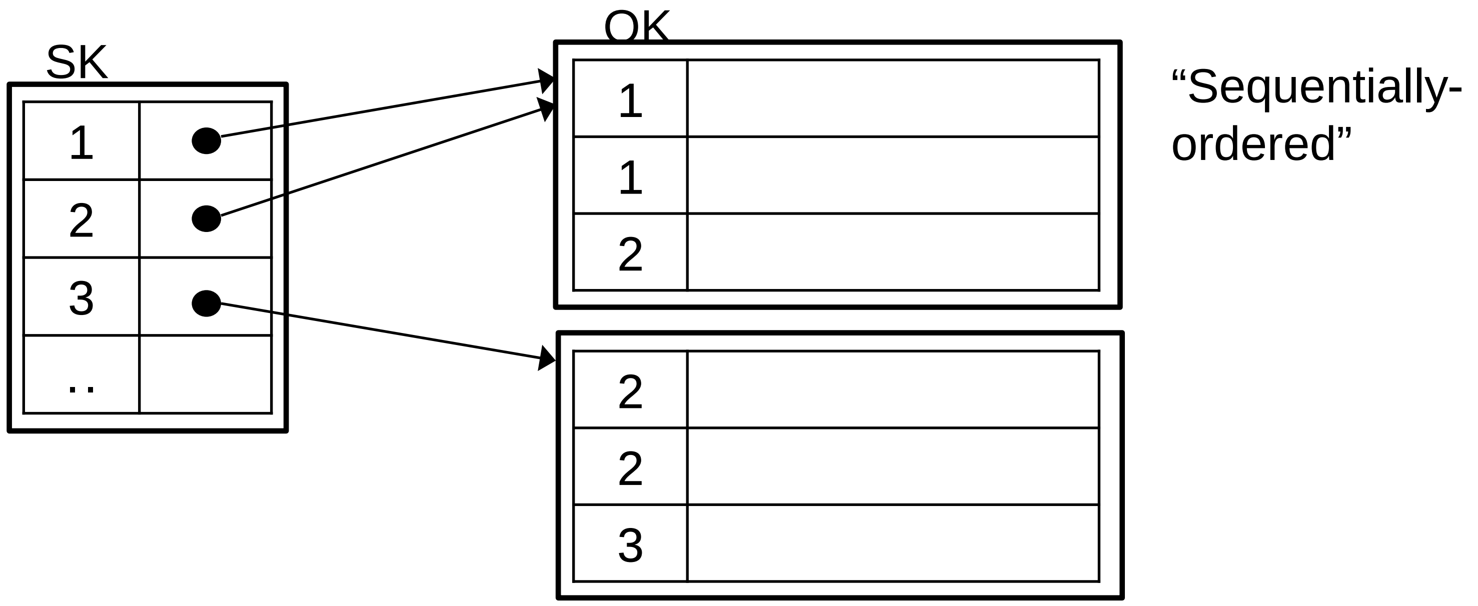
Dense vs. sparse index

- **Dense index:**
  - An index entry for each search-key in the file (previous example)
- **Sparse index:**
  - Index entries only for some search-key values
  - Applicable when tuples are sequentially ordered on search-key
  - Less space, generally slower in locating the tuple
  - Good trade-off: one index entry for each block in the file



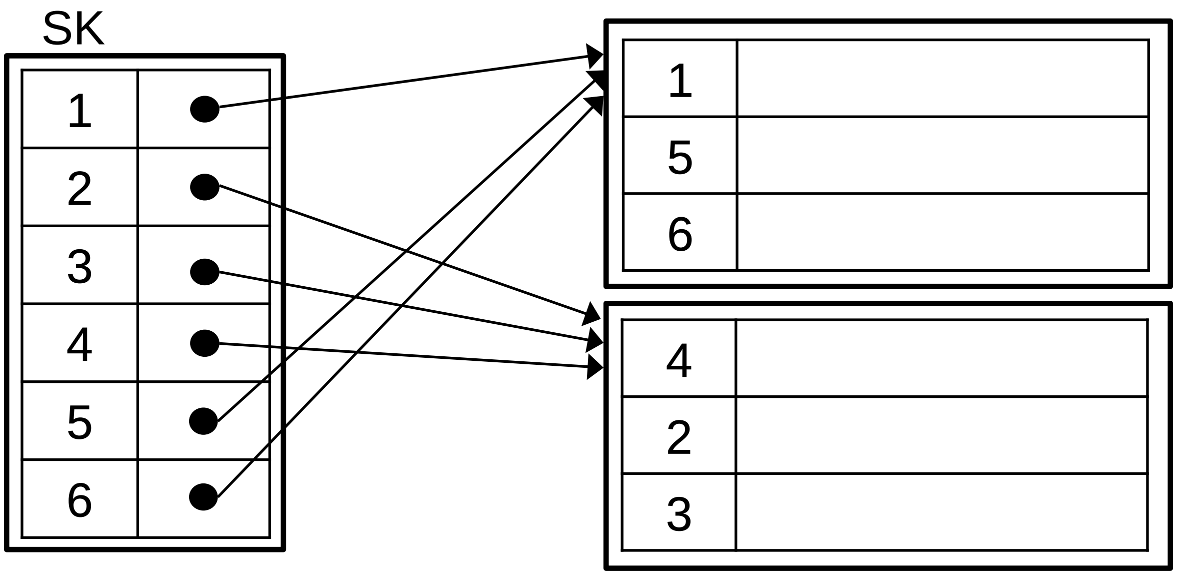
Clustering index

- If the ordering key field in the primary structure is not unique, a ***clustering index*** is used



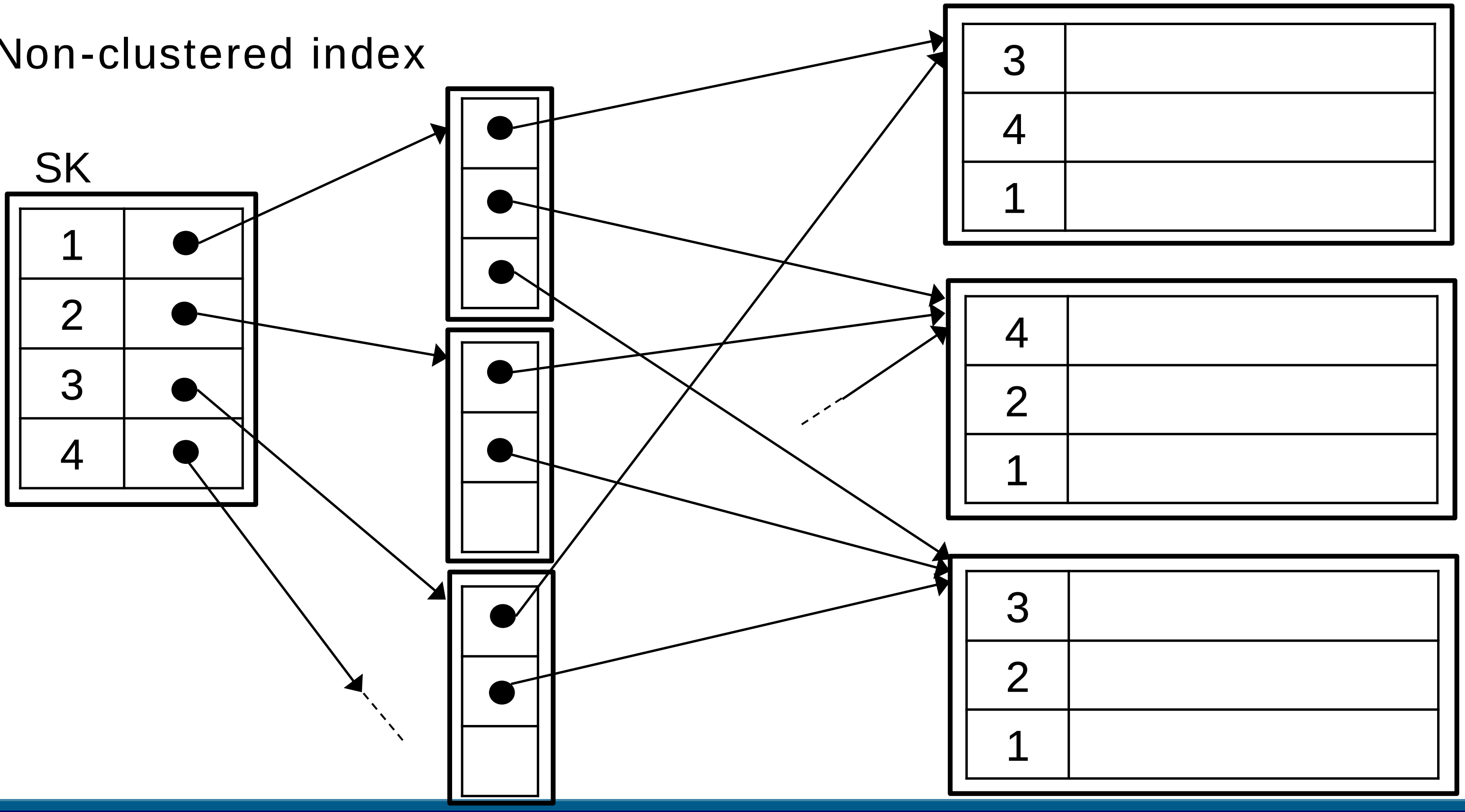
Secondary index

- **Secondary index:**
  - The search key specifies an order different from the sequential order of the file
  - Non-clustered index
  - Multiple secondary indexes can be defined, on different search keys



Non-unique secondary index

- Numerous tuples in the file have the same value
- Each entry in the index points to a bucket with pointers to the file
- Multiple secondary indexes can be defined, on different search keys
- Non-clustered index



A search key is not a Primary key!

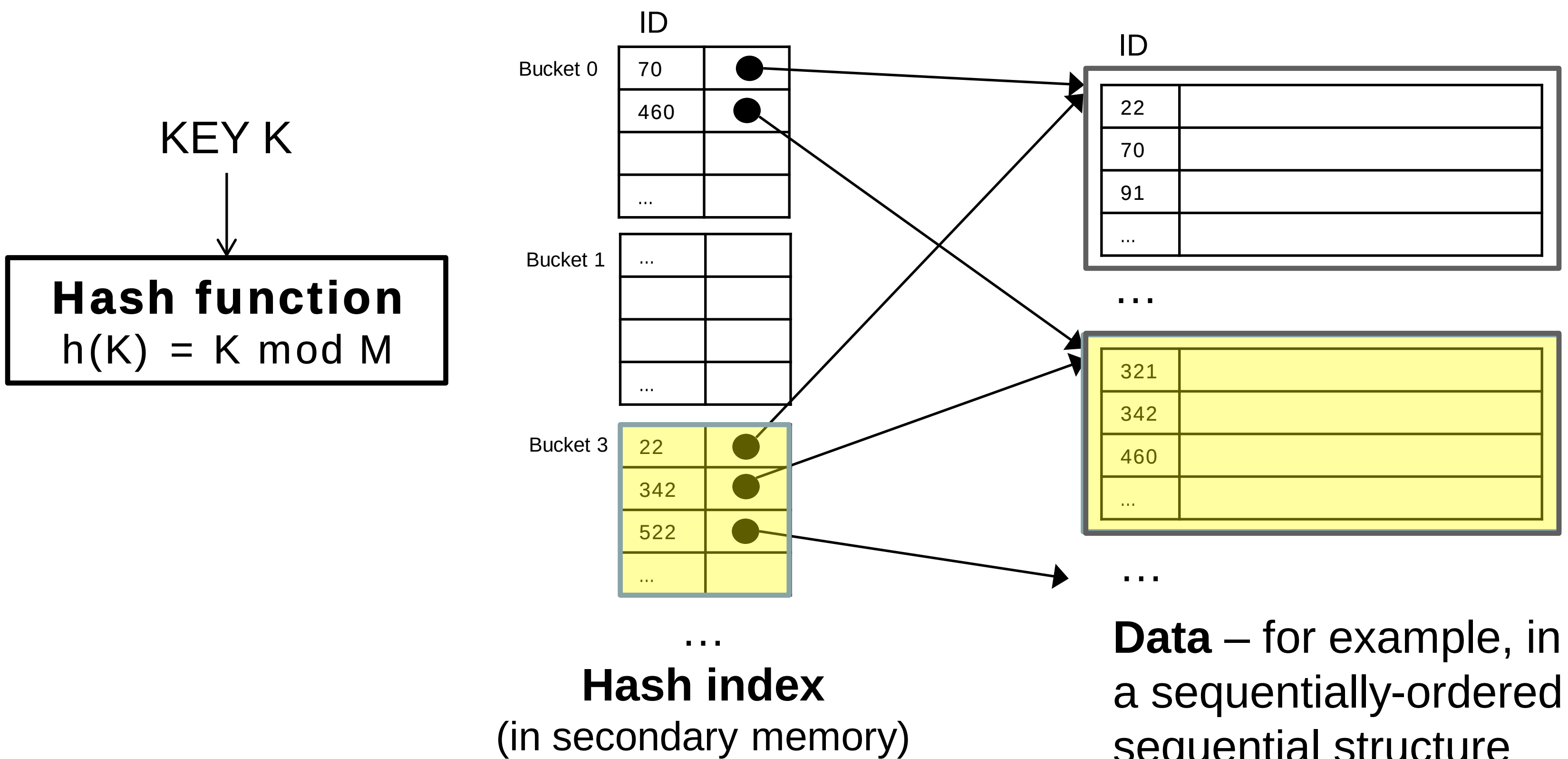
Don’t get confused:

- **Primary key**: set of attributes that uniquely identify a tuple (minimal, unique, not null)
    - Does not imply access path
    - In SQL “PRIMARY KEY” defines a constraint
    - Implemented by means of an index
- Indexes are associated with a **search key**, composed by one or more attributes
    - Physical implementation of access structures
    - They define a common access path
        - Each key is associated with one or more pointers
    - May be unique (one pointer) or not unique (more pointers)

Using hash-based structures as indexes

- Hash-based structures can be used for **secondary indexes**
- Shaped and managed exactly like a hash-based primary structure, but
  - instead of the tuples, the buckets only contain **key values and pointers**

Hash-based index



SELECT * FROM Student WHERE Student.ID = '342'

#I/O operations = 2 (without overflow chains)

About hashing

- Good performance for **equality predicates** on the key field
- **Inefficient** for access based on
  - interval predicates or
  - the value of non-search-key attributes

Tree-based structures

- Most frequently used in relational DBMSs for secondary structures
  - SQL indexes are implemented in this way

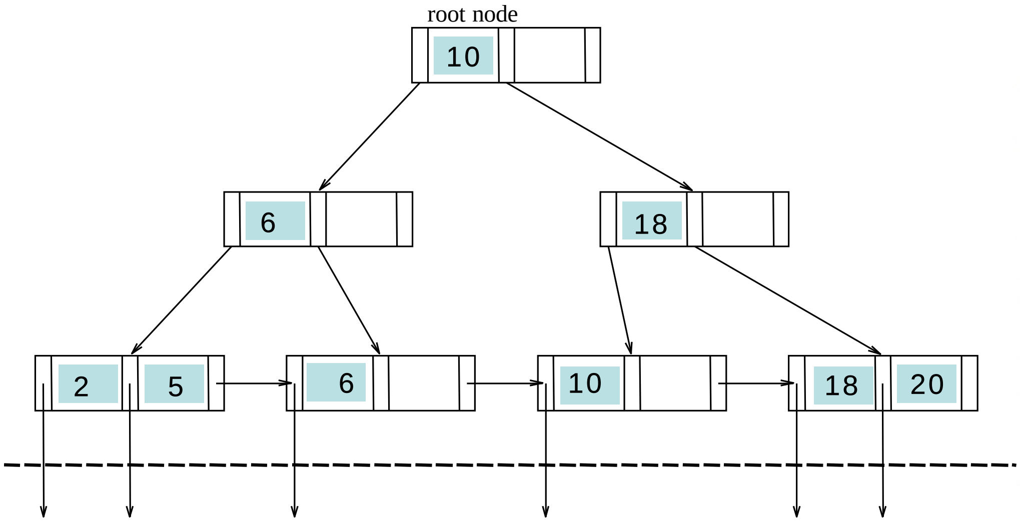
- Gives associative access based on the value of a ***key search*** field

- Two main file organizations (**B**alanced trees):
  - B+ trees (most used)
  - B trees

  - In a ***balanced tree***, the lengths of the paths from the root node to the leaf nodes are all equal. Balanced trees give **optimal** performance.

B+ tree structures

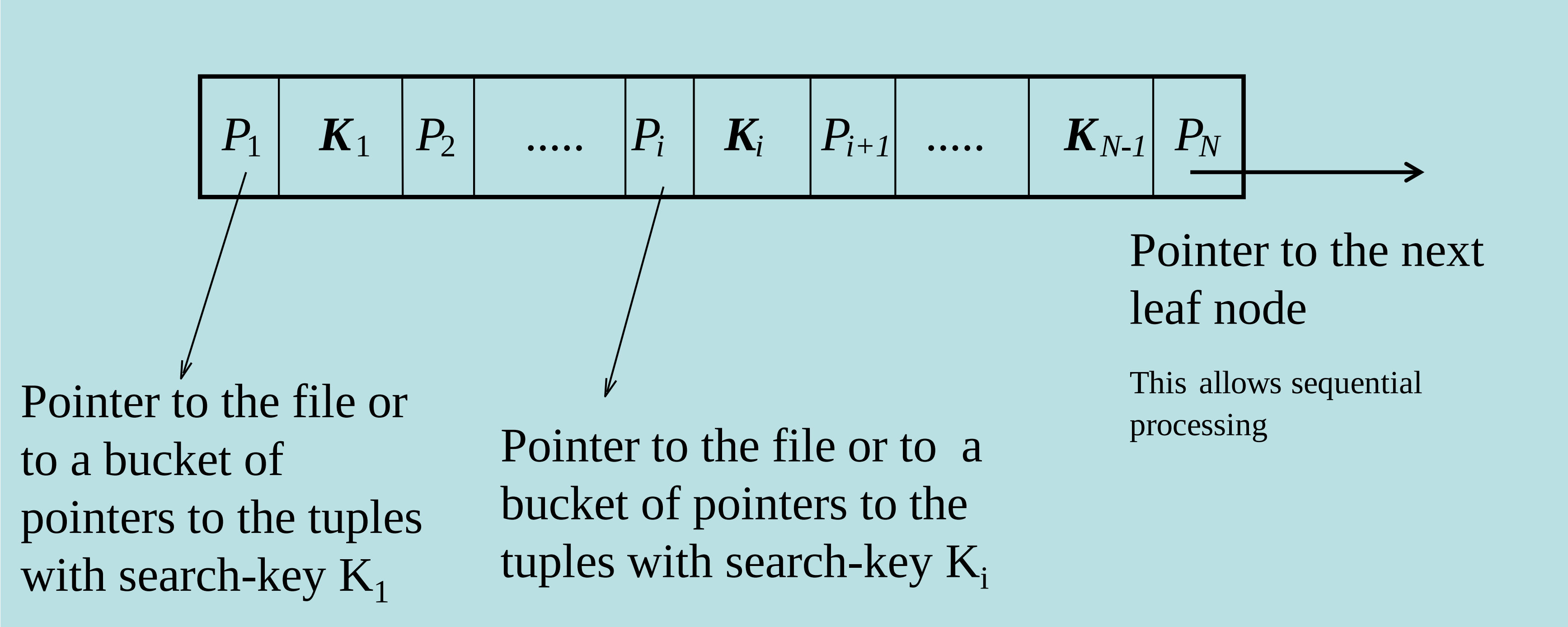
- Multi-level index
  - one root node
  - several intermediate nodes
  - several leaf nodes



- Each node is stored in a **block**
- In general, each node has a large number of descendants (**fan out**), and therefore the majority of blocks are leaf nodes

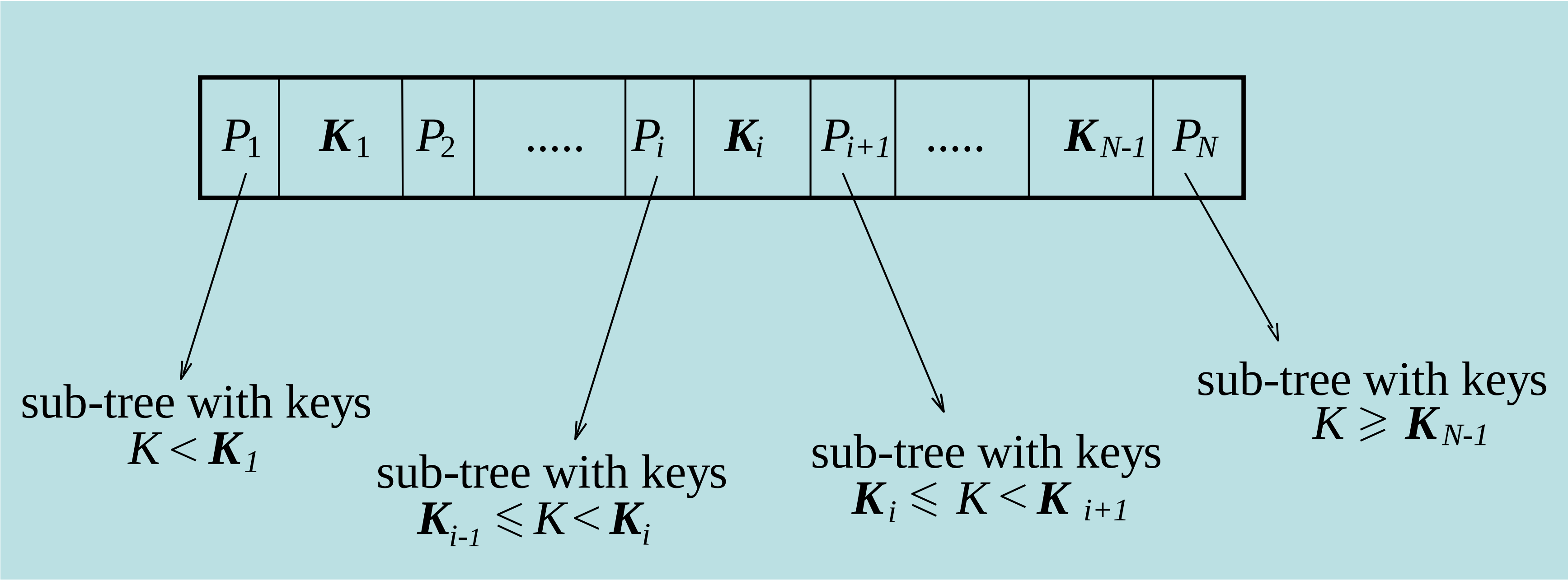
Structure of the B+ tree leaf nodes

- Each node can hold up to N-1 search-keys
  - At least ⌈(N-1)/2⌉
- Search-keys in a node a sorted $K_i < K_j$ with i<j
- The set of leaf nodes forms a dense index

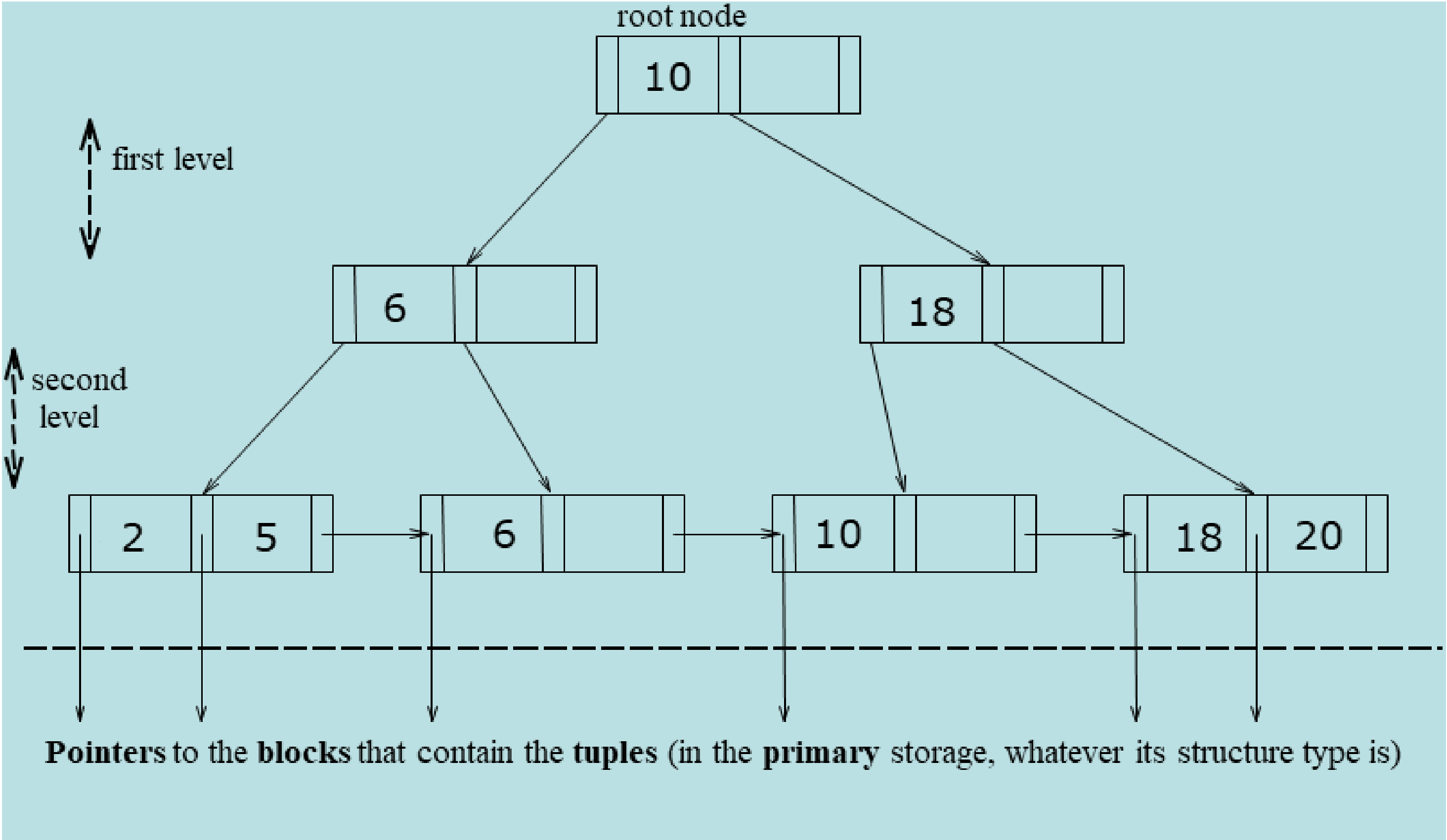


Structure of the B+ tree internal nodes

- Internal nodes form a multilevel (sparse) index on the leaf nodes
- Each node can hold up to N-1 search-keys and N pointers
  - At least ⌈N/2⌉ pointers (except for the root node)
- Search-keys in a node a sorted $K_i < K_j$ with i<j



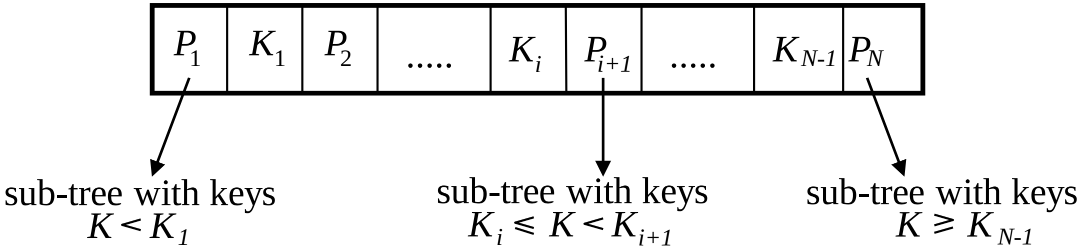
An example of secondary B+ tree



- The **leaf nodes are linked in a chain ordered by the key**
- Supports interval queries efficiently

Search technique / lookup

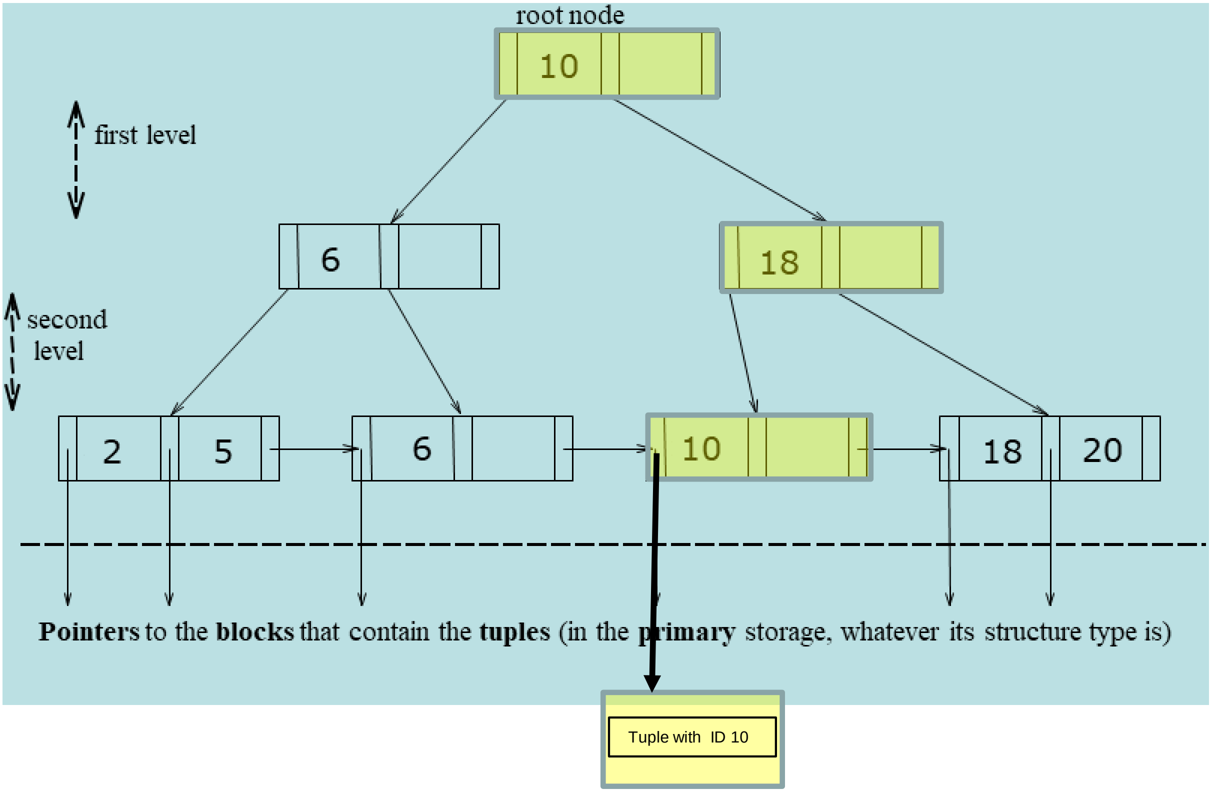
- Looking for a tuple with key value V, at each intermediate node:
  - if $V < K_1$ follow P_0
  - if $V \geq K_F$ follow P_F
  - otherwise, follow P_j such that $K_j \leq V < K_{j+1}$



Examples of insert/delete (visualization tool)

B+ tree – query with equality predicate

```
SELECT *
FROM  Student
WHERE Student.ID = '10'
```

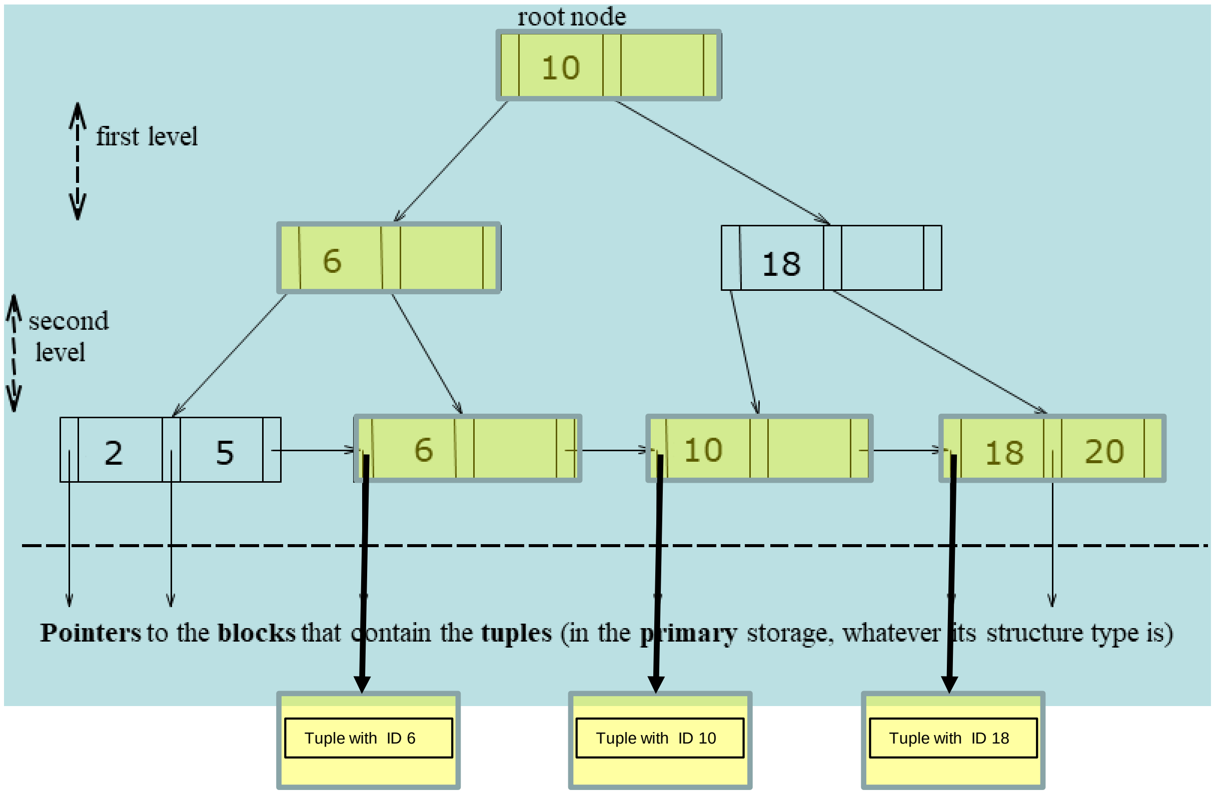


#I/O operations = #levels to reach the leaf +
1 access to the block containing the tuple

→ In the example: 4 I/O operations

B+ tree – query with interval predicate

SELECT *
FROM Student
WHERE Student.ID
BETWEEN ‘6’ and ’19’

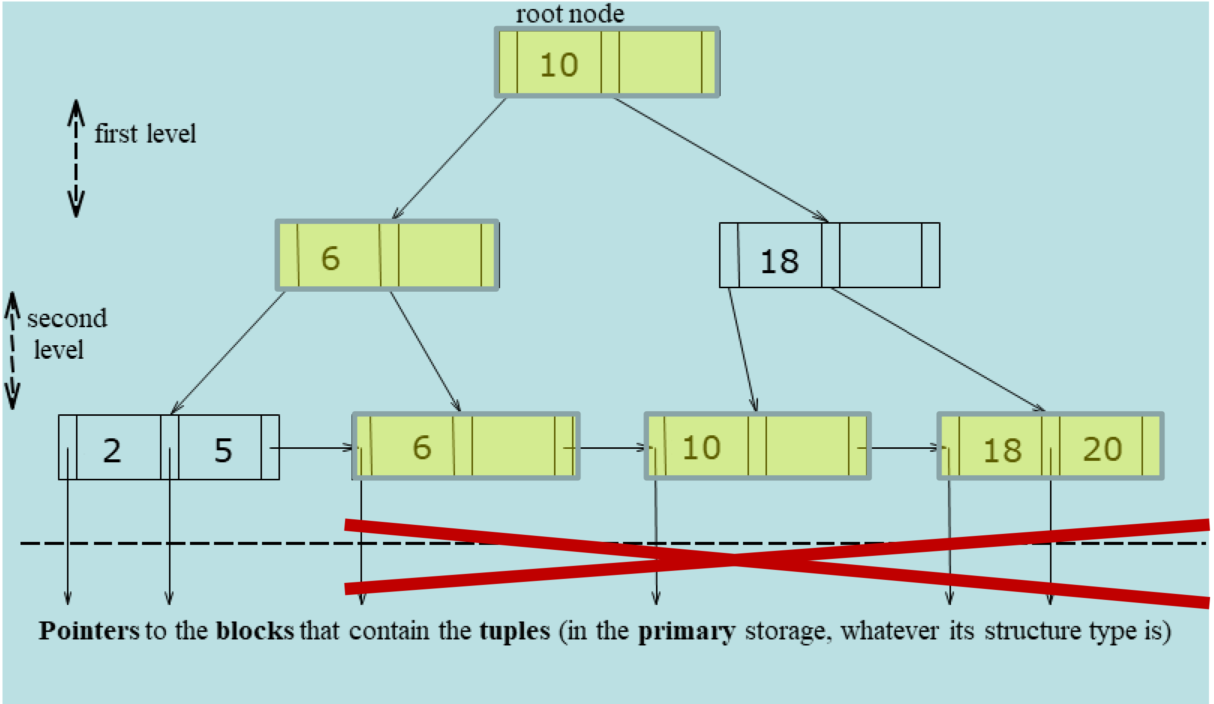


#I/O operations = #levels to reach the leaf + # of leaf nodes that are visited + #accesses to the blocks containing the tuples

→ In the example: 2 intermediate nodes + 3 leaf nodes + 3 data blocks = 8 I/O operations

B+ tree – query with interval predicate

SELECT Student.ID
FROM Student
WHERE Student.ID
BETWEEN '6' and '19'



The leaf nodes of the index contain all the ID values, sorted in ascending order!

We can access only the B+ tree

#I/O operations = #levels to reach the leaf + # of leaf nodes that are visited = 5 I/O

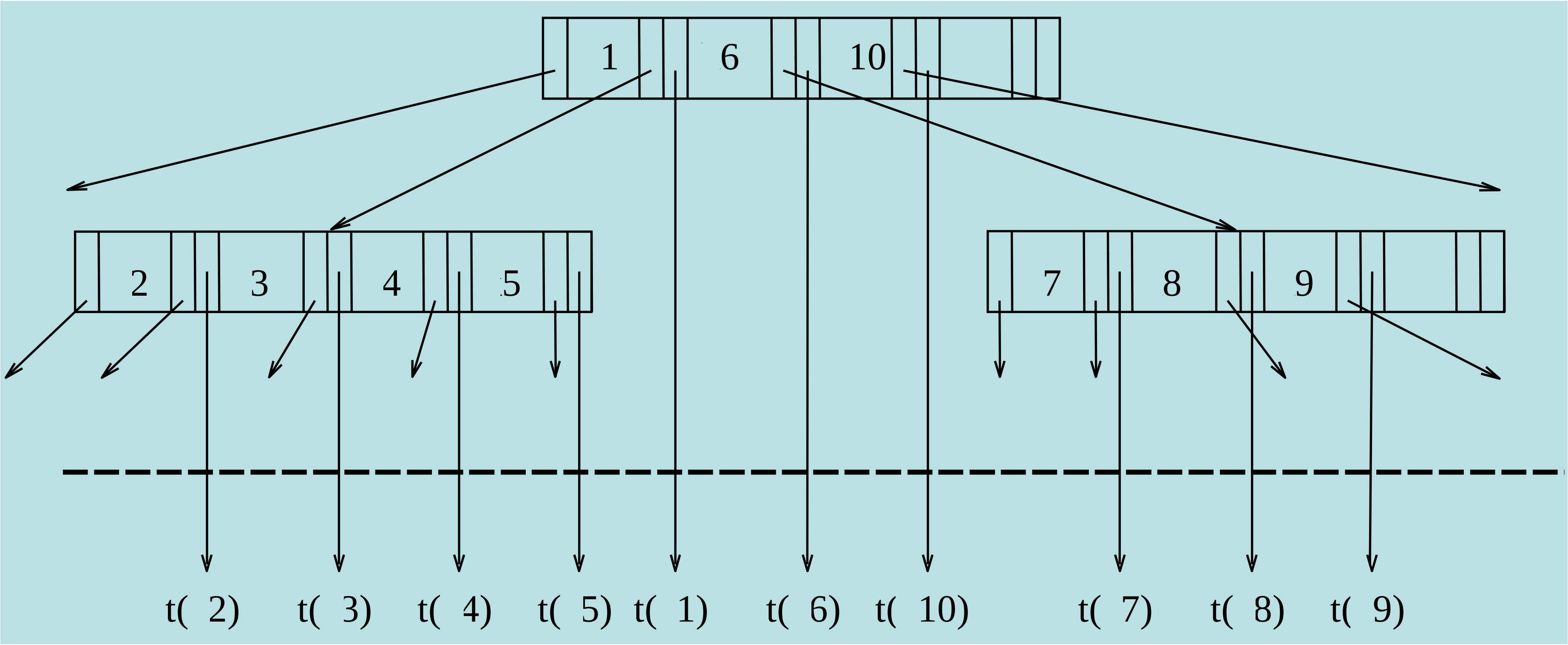
B-tree

- Eliminates the redundant storage of search-key values
  - They appear only once
  - Intermediate nodes for each key value K_i have:
    - One pointer to the sub-tree with keys between K_i and K_{i+1}
    - One (or more) pointer(s) to the block(s) that contain(s) the tuple(s) that have value K_i for the key

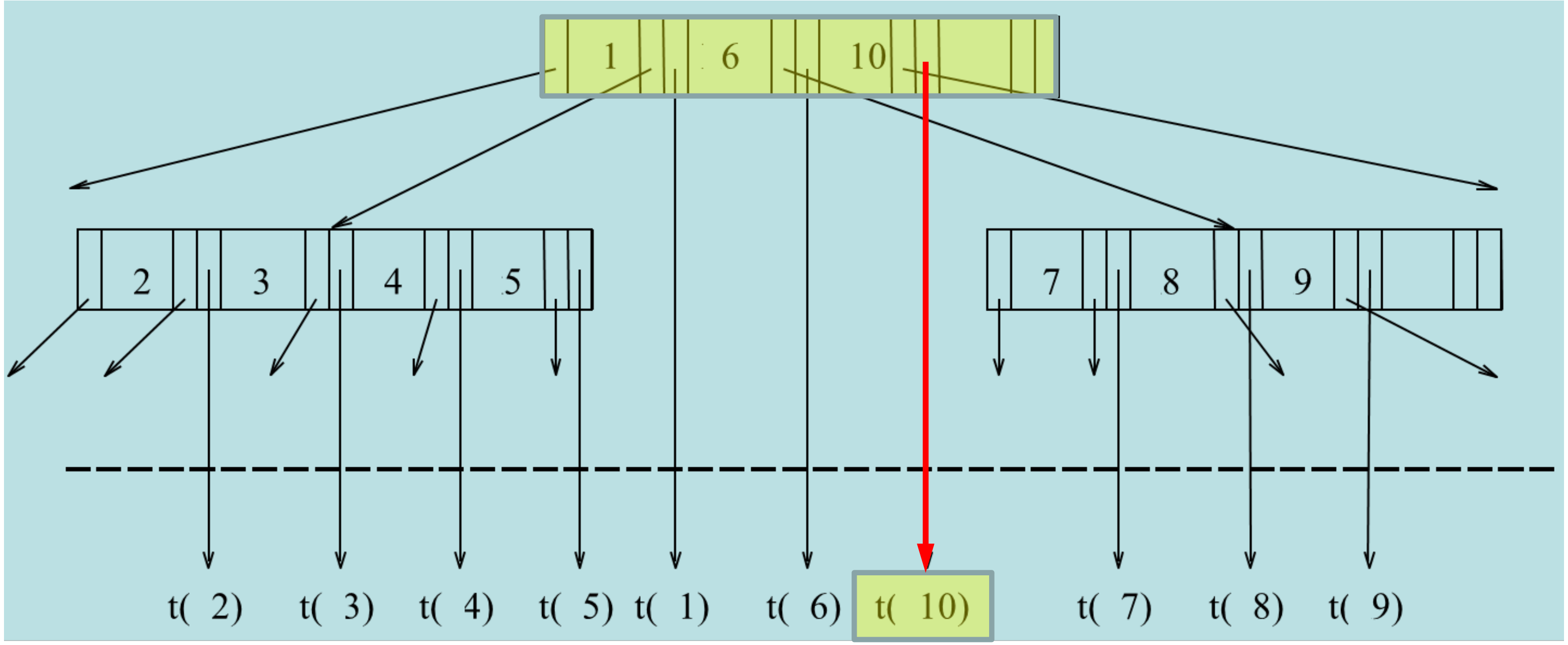
      (there can be more than one tuple if the key is not unique)

- Lookup can be slightly faster, but interval queries are less efficient

An example of secondary B tree



SELECT * FROM Student WHERE Student.ID='10'

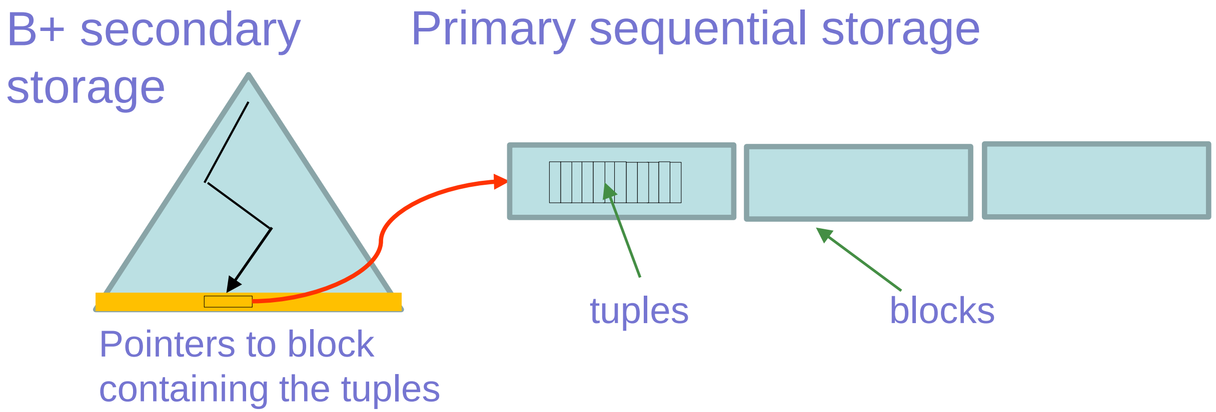


#I/O operations = # levels to find the ID in the tree +
1 access to the block containing the tuple

→ In the example: 2 I/O operations

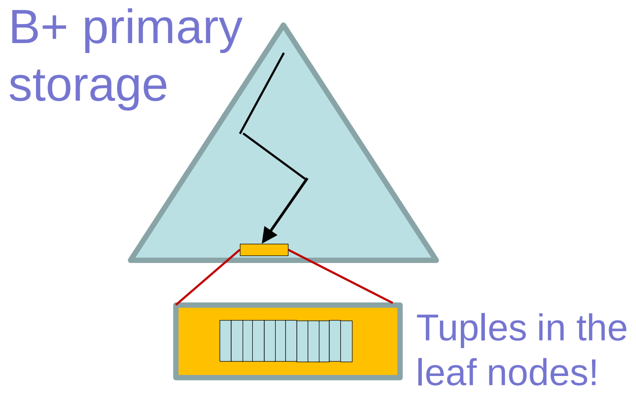
Primary vs. Secondary tree structures

- Trees are (more) often used as **secondary** access structures
  - Tuples are primarily stored within another access structure (hashed, sequential, a primary tree with a different key)
  - The tree nodes only contain **key values and pointers**
    - Necessarily a "*dense*" index: one index entry pointing to every tuple of the primary storage is required (or the tuple is "lost" !)



Primary vs. Secondary tree structures

- Trees can be also used as **primary** access structures:
  - Tuples are stored in the tree nodes (or in a file sorted according to the search key – see clustering index)



Indexes in SQL

- **Syntax in SQL:**

  create [unique] index *IndexName* on *TableName*(*AttributeList*)

  drop index *IndexName*

- Some examples:

  Oracle create index

  MySQL create index

Indexes in SQL

- Every table should have:
  - A suitable ***primary storage***, possibly key-sequenced (normally on unique values, typically the primary key)
  - Several ***secondary indexes***, both unique and not unique, on the attributes <u>most used for selections and joins</u>
- Secondary structures are progressively added, checking that the system actually uses them

Some guidelines for choosing indexes

(1) Do not index **small tables**.

(2) Index **Primary Key** of a table if it is not a key of the primary file organization.

(Some DBMS automatically create unique indexes on primary keys and unique keys)

(3) Add secondary index to any column that is heavily used as a secondary key.

(4) Add secondary index to a **Foreign Key** if it is frequently accessed.

Some guidelines for choosing indexes

(5) Add secondary index on columns that are involved in: **selection or join criteria; ORDER BY; GROUP BY;** other operations involving sorting (such as **UNION or DISTINCT**).

(6) Avoid indexing a column or table that is frequently **updated**.

(7) Avoid indexing a column if the query will retrieve a **significant proportion** of the records in the table.

(8) Avoid indexing columns that consist of **long character strings**.

INTRODUCTION TO OPTIMIZATION

COSTS OF DIFFERENT ACCESS MODES

Query optimization

- We learned about tree & hash indexes
  - How does DBMS know when to use them?
- The same query can be executed by the DBMS in many ways
  - How does DBMS decide which is best?

Query optimization

- **Optimizer**:
  - it receives a query written in SQL and
  - produces a program in an ‘internal’ format that uses the data access methods
- **Steps**:
  - Lexical, syntactic and semantic analysis
  - Translation into an internal representation
  - Algebraic optimization
  - Cost-based optimization
    - The query may be “rewritten” by the DBMS
  - Code generation

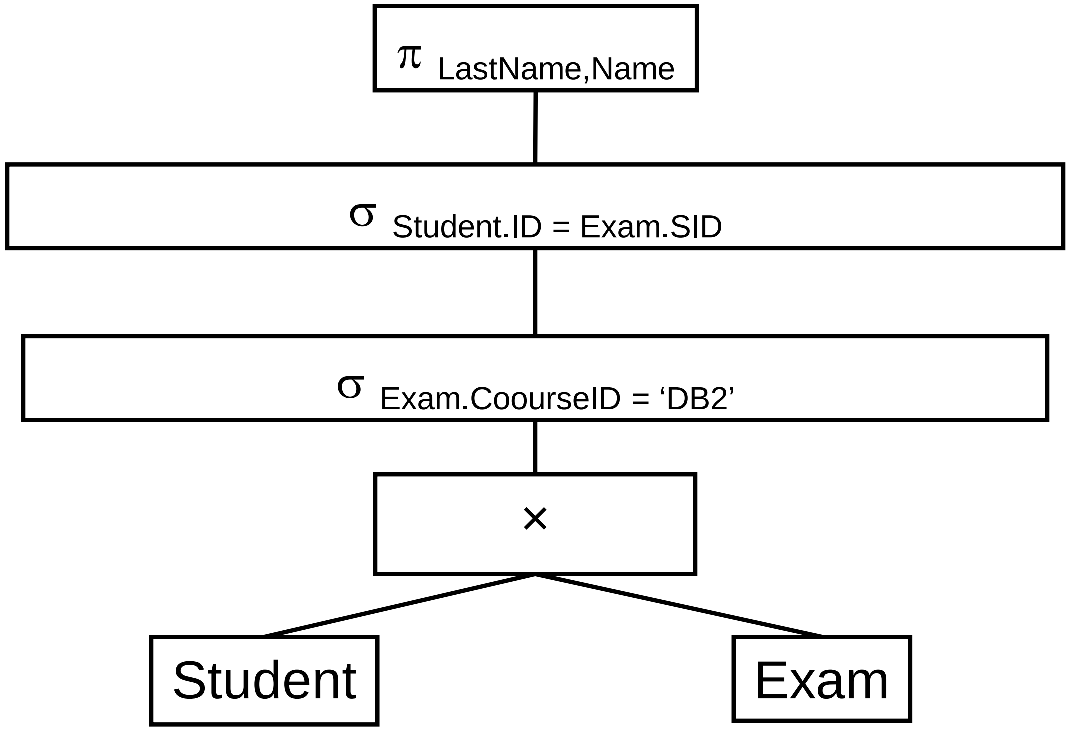
A simple example of query optimization

STUDENT (<u>Id</u>, Name, LastName, Email, City, Birthdate, Sex)

EXAM (<u>SId, CourseId</u>, Date, Grade)

```
SELECT LastName,Name
FROM Student, Exam
WHERE
  Student.ID = Exam.SID
AND
  Exam.CourseID = ‘DB2’
```

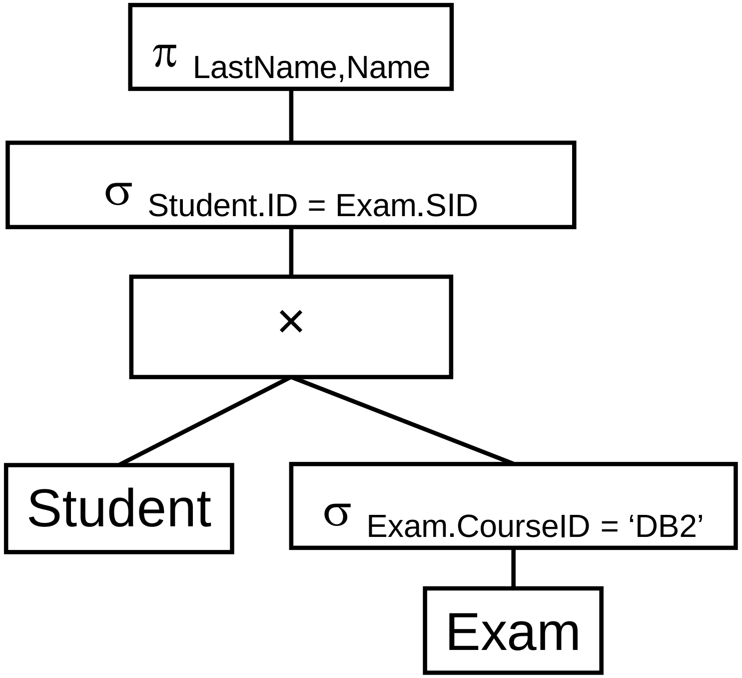
Query Tree



- **Many ways to optimise queries**:
  - <u>Changing the query tree</u> to an equivalent but more efficient one
  - Exploiting database <u>statistics</u>
  - Choosing efficient <u>implementations of each operator</u>

Optimisation Example

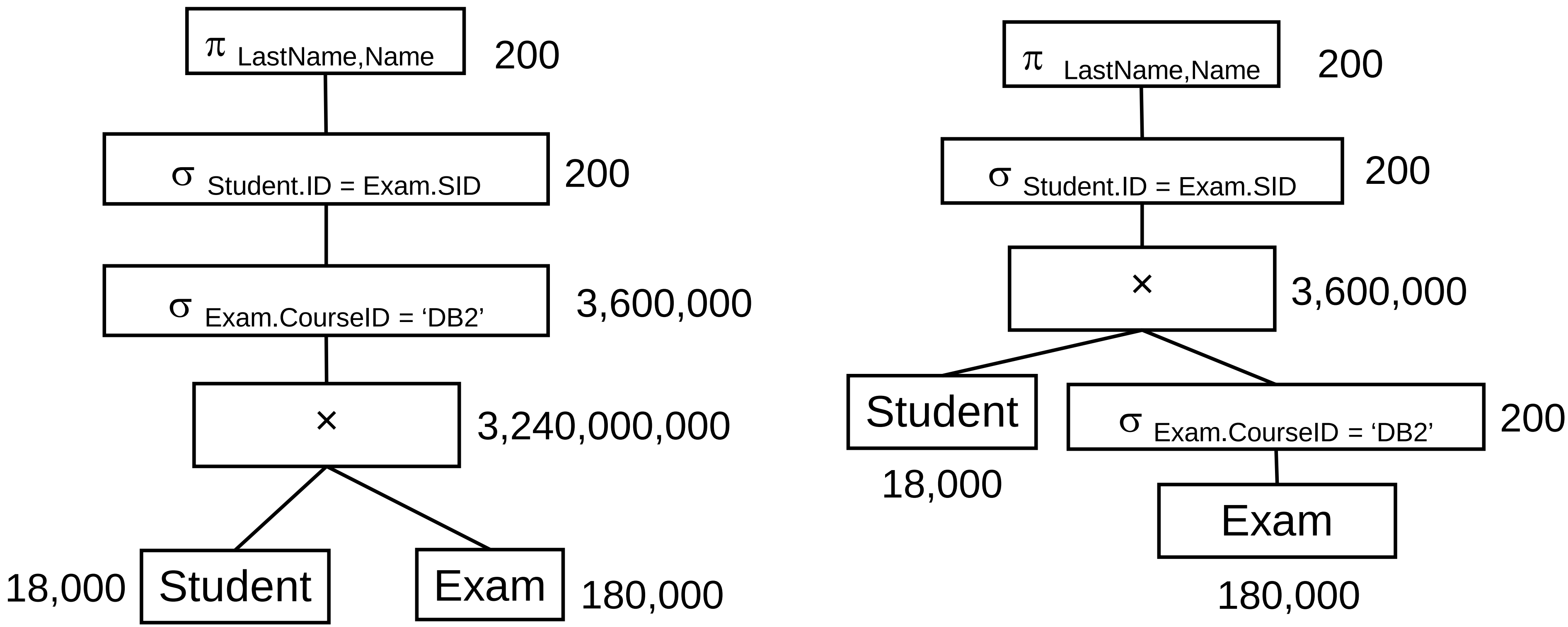
- Equivalent query:
  - Selecting Exam entries with CourseID = ‘DB2’
  - Take the product of the result with Student



Optimisation Example

- Consider the following statistics
  - The university has 18,000 students
  - Each student is enrolled in at about 10 exams
  - Only 200 take DB2

Original and Optimized Query Tree



Relation profiles

- Profiles are stored in the **data dictionary** and contain quantitative information about tables:
  - the cardinality (number of tuples) of each table T
  - the dimension in bytes of each attribute A_j in T
  - the number of distinct values of each attribute A_j in T (**val(**A_j**)**)
  - the **min**imum and **max**imum values of each attribute A_j in T
- Periodically calculated by activating appropriate system primitives (for example, the **update statistics** command)
- Used in cost-based optimization for estimating the size of the intermediate results produced by the query execution plan

Data profiles and selectivity of predicates

- **Selectivity =** probability that any row will satisfy a predicate
  - If **val(A)=N** and
  - the **values are homogeneously distributed** over the tuples, then
  - the **selectivity** of a predicate in the form **A=k** is **1/N**
- Ex: val(City)=200 → selectivity for City = 1/200 = 0.5%
  - 18K students → 90 students per city
- If no data on distributions are available, we will always assume homogeneous distributions!

Optimizations

Access methods

- Sequential
- Hash-based indexes
- Tree-based indexes

Operations

- Selection
- Projection
- Sort
- Join
- ...

Running example

STUDENT (<u>Id</u>, Name, LastName, Email, City, Birthdate, Sex)

150K tuples

1 t = 4+20+20+20+20+10+1 = 95 byte

Block size 8KB → 87 tuples/block → For 150K students: **1.7K blocks**

EXAM (<u>SId, CourseId</u>, Date, Grade)

1.8M tuples

1 t = 4+4+12+4 = 24 byte

Block size 8KB → 340 tuples/block → For 1.8M exams: **5.3K blocks**

Sequential scan

- Performs a sequential access to all the tuples of a table or of an intermediate result, at the same time executing various operations, such as:
  - Projection to a set of attributes
  - Selection on a simple predicate (of type: $\boldsymbol{A_i = v}$)

Hash and indexes

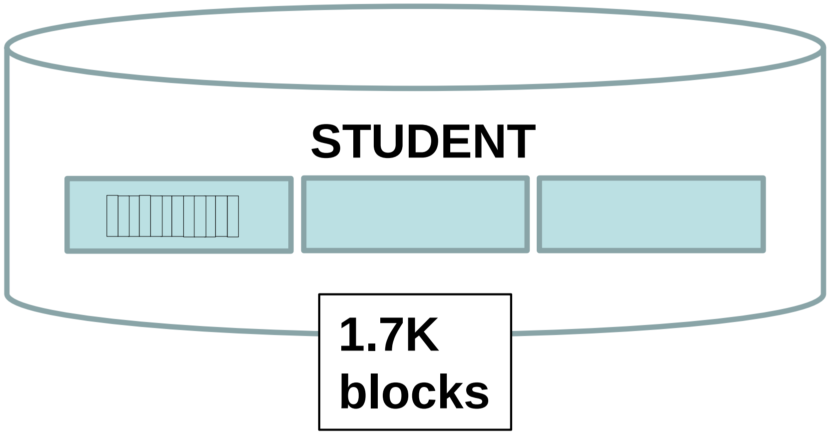
- **Hashing:**
  - For *equality predicates*

- **Tree-based indexes support:**
  - *selection or join criteria*
  - *ORDER BY*
  - *GROUP BY*
  - other operations involving sorting (such as *UNION or DISTINCT*)

Cost of a sequential scan for a query

- Example – suppose STUDENT sequentially ordered by ID:

SELECT *
FROM STUDENT
WHERE City="Milan"



- Cost: 1.7K I/O accesses

SELECT *
FROM STUDENT
WHERE ID<500

- Cost: **<** 1.7K I/O accesses, you can stop when ID=500

Indexed access (lookup)

- Indexes built on $\boldsymbol{A_i}$ can used for queries with:
  - simple predicates (of the type $\boldsymbol{A_i = v}$ or $\boldsymbol{A_i > v}$)
  - interval predicates (of the type $\boldsymbol{v_1} \leq \boldsymbol{A_i} \leq \boldsymbol{v_2}$)
- If there are two or more predicates in **conjunction** supported by indexes, the DBMS chooses the **most selective supported predicate** for the data access, and **evaluates the other predicates in main memory**
- If the predicates are in **disjunction**:
  - if any of the predicates is not supported by indexes, then a scan is needed
  - if all are supported, indexes can be used (**for all predicates**) and then duplicate elimination is normally required

Cost of lookups: equality ($A_i = v$)

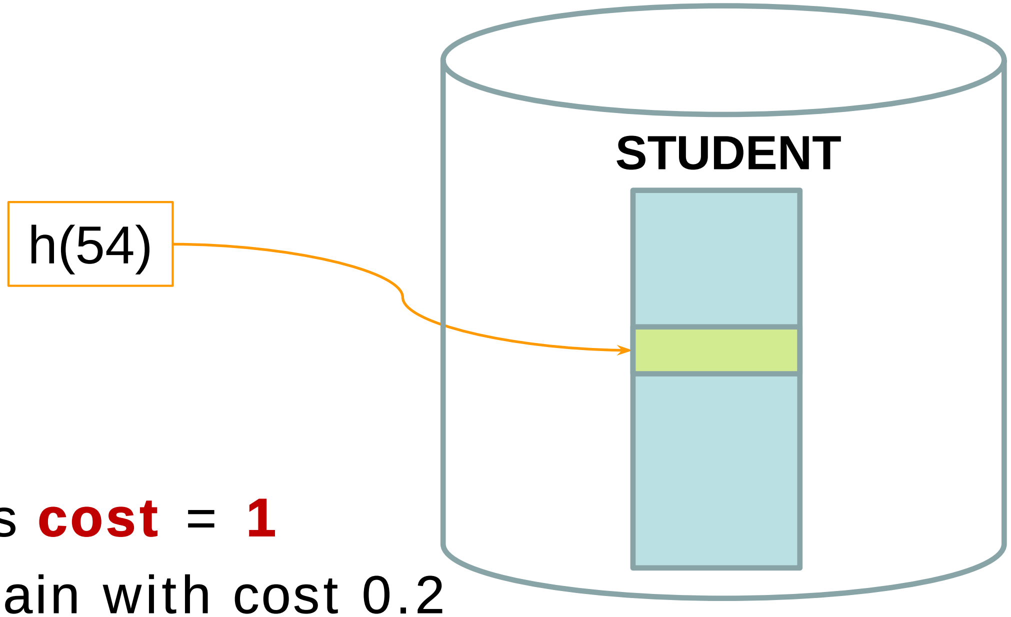
- **Sequential** structures
  - Lookups are not supported (cost: a full scan)
    - Sequentially-ordered structures may have reduced cost

- **Hash / Tree** structures
  - Supported if A_i is the index key attribute of the structure
  - The cost depends on
    - the storage type (primary/secondary)
    - the index key type (unique/non-unique)

Cost of equality lookup on a primary Hash

```
SELECT *
FROM STUDENT
WHERE ID='54'
```

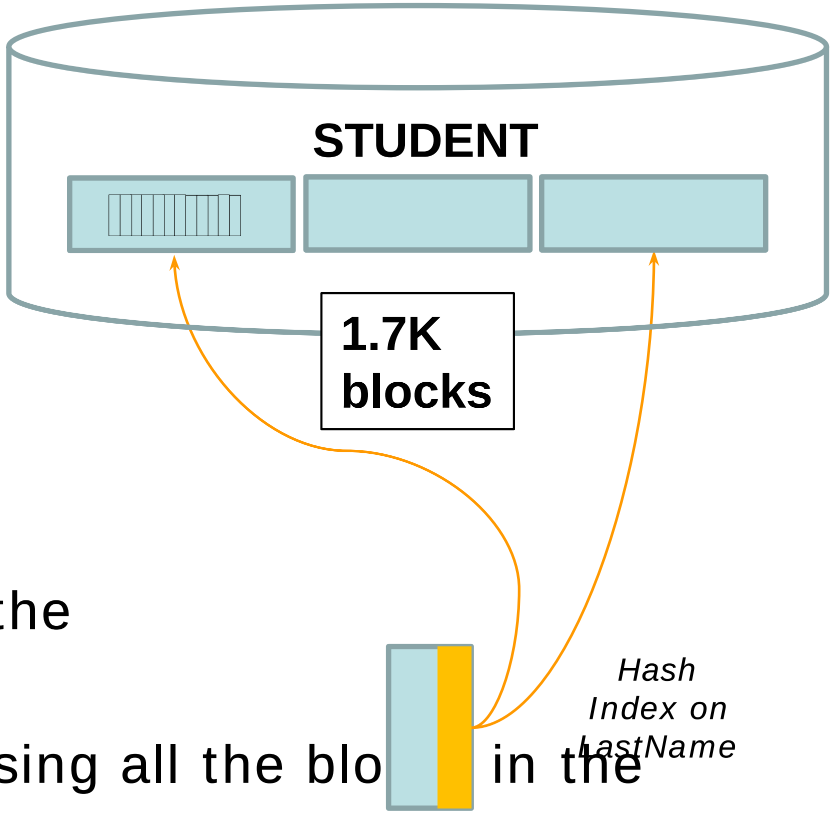
STUDENT (hash table built on ID!)



- Cost:
  - no overflow chains **cost** = **1**
  - With **overflow** chain with cost 0.2
    cost = **1.2**

Cost of equality lookup on a secondary Hash

SELECT *
FROM STUDENT
WHERE LastName="Rossi"



- Cost:
  - **Without** overflow chains:
    cost to access the hash index=**1**
  - **With** overflow chains:
    cost = **1 +** the **average** size of the **overflow** chain

  + we have to add the cost of accessing all the blo[cks] in the primary storage

Cost of equality lookup on a secondary Hash

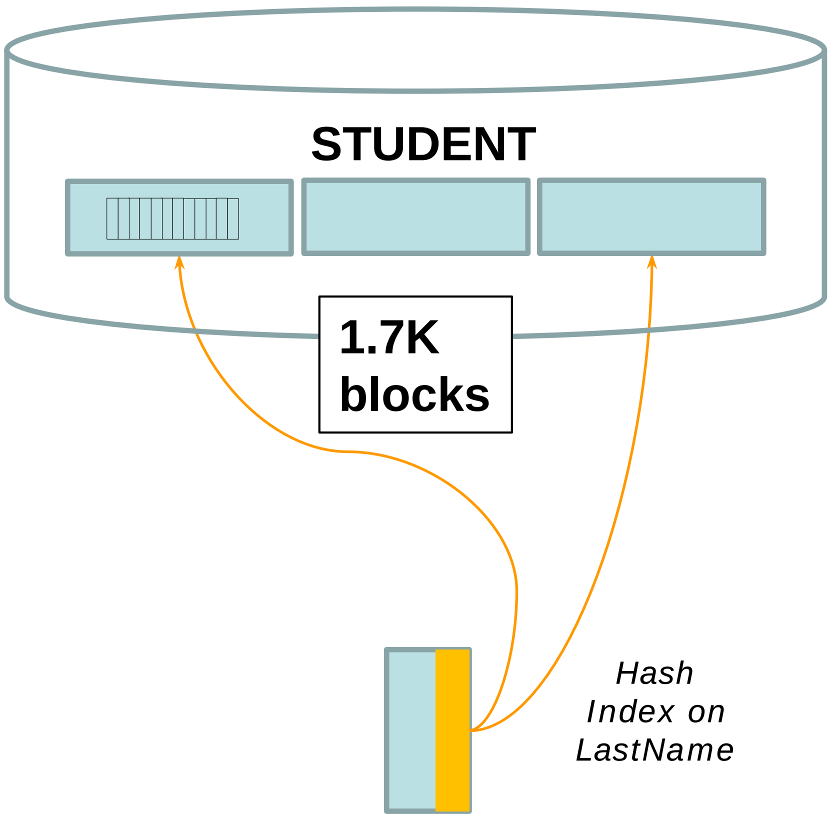
How many pointers for “Rossi”?

The system uses statistical data,
in particular:

the number of distinct values for attribute *LastName* in STUDENT:
val(*LastName*)

Suppose val(LastName)=75K

Since STUDENT contains 150K tuples,
it means that, in average, each lastname appears twice

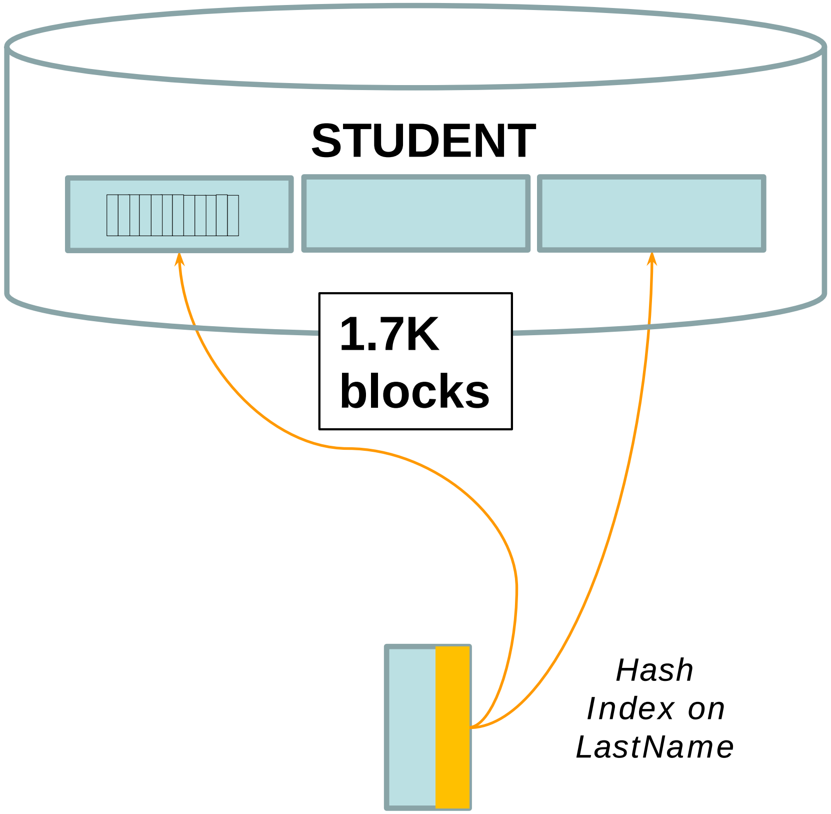


Cost of equality lookup on a secondary Hash

How many pointers for “Rossi”?

→ We expect to access *<u>in average</u>* 2 blocks for a given lastname

TOTAL COST (with overflow chain) =
1.2 (cost to access the hash index) **+**
2 (cost to access 2 tuples of STUDENT needed for the SELECT *) **= 3.2**



Cost of equality lookup on a primary B+

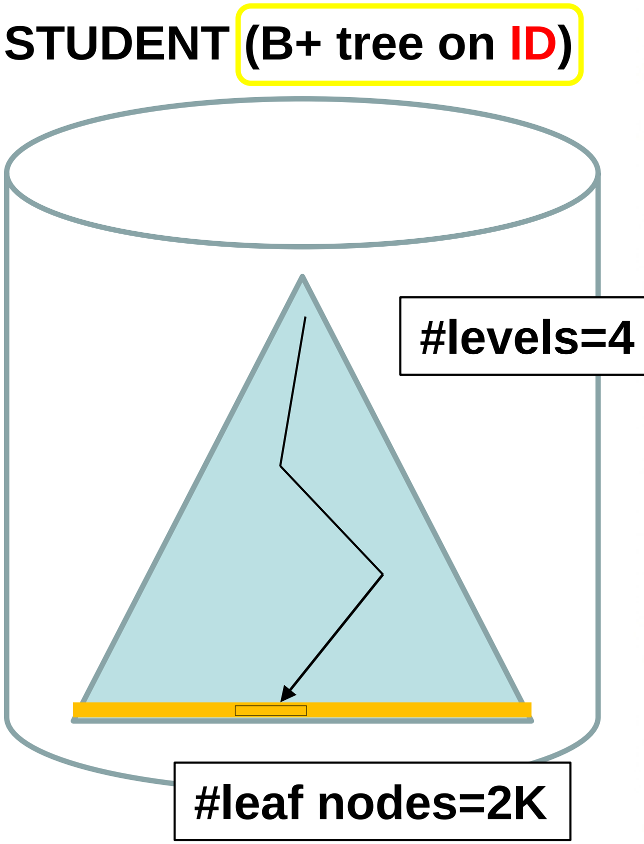
a) SELECT *
FROM STUDENT
WHERE ID=54

- Cost:
  - #levels to access the leaf = **4**

b) SELECT *
FROM STUDENT
WHERE City="Milan"

- Cost: all the tuples are in the leaf nodes!

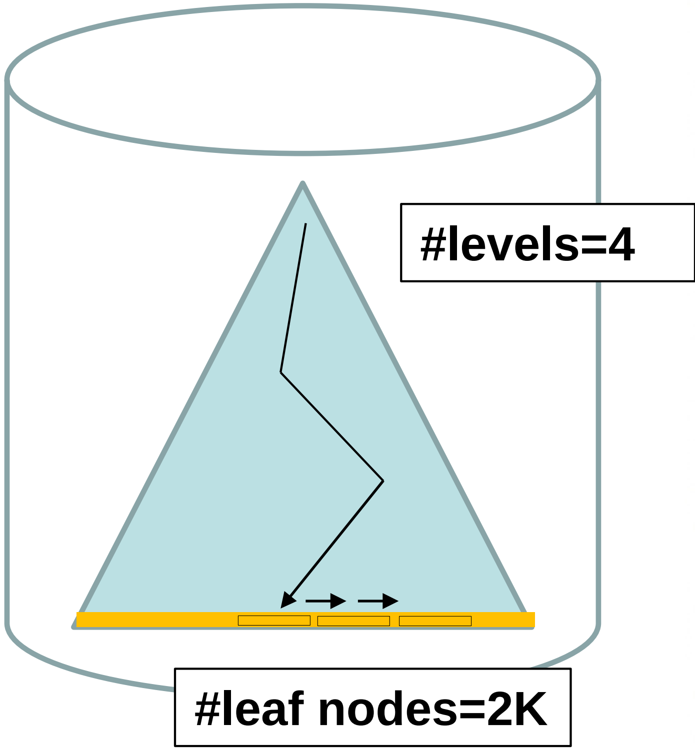
We need to reach them and scan all of them: **3** intermediate levels **+ 2K** #leaf nodes



Cost of equality lookup on a primary B+

```
SELECT *
FROM STUDENT
WHERE City="Milan"
```

STUDENT (B+ tree on City)



- With statistics: val(City)=150 → 150K/150 = 1K tuples/City
- Cost:
  #levels to access the leaf = **4**

  Q: Are all the tuples of Milan students contained in a block?

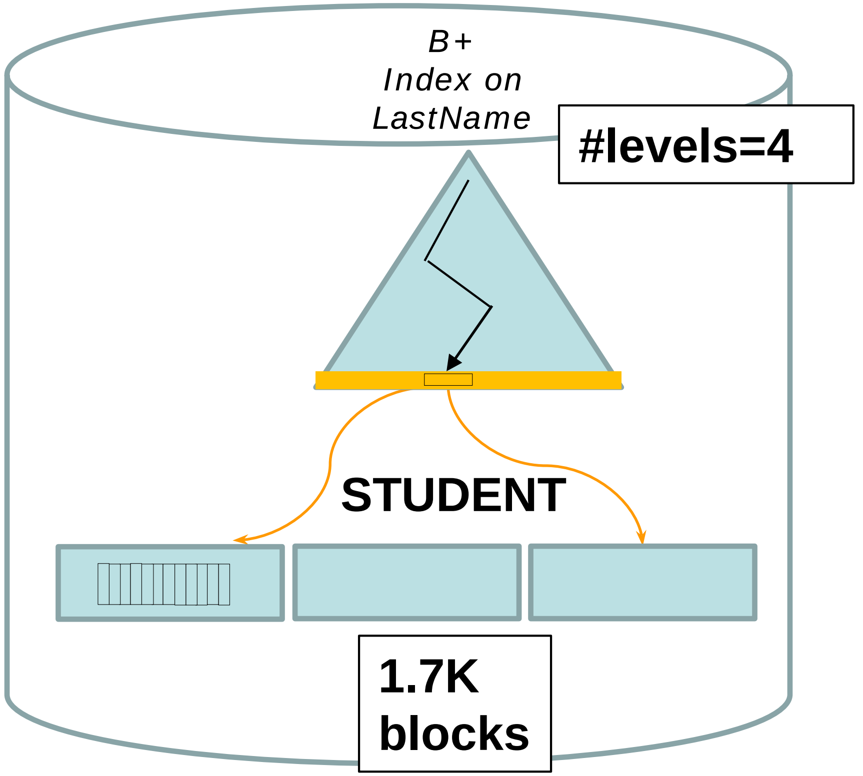
#Tuples in 1 leaf node: 150K tuples/2K nodes = 75 tuples/node

#blocks with Milan tuples: 1K Milan tuples/75 tuples/node = **14** nodes

Total cost = 3 intermediate levels + 14 = **17**

Cost of equality lookup on a secondary B+

SELECT *
FROM STUDENT
WHERE LastName="Rossi"



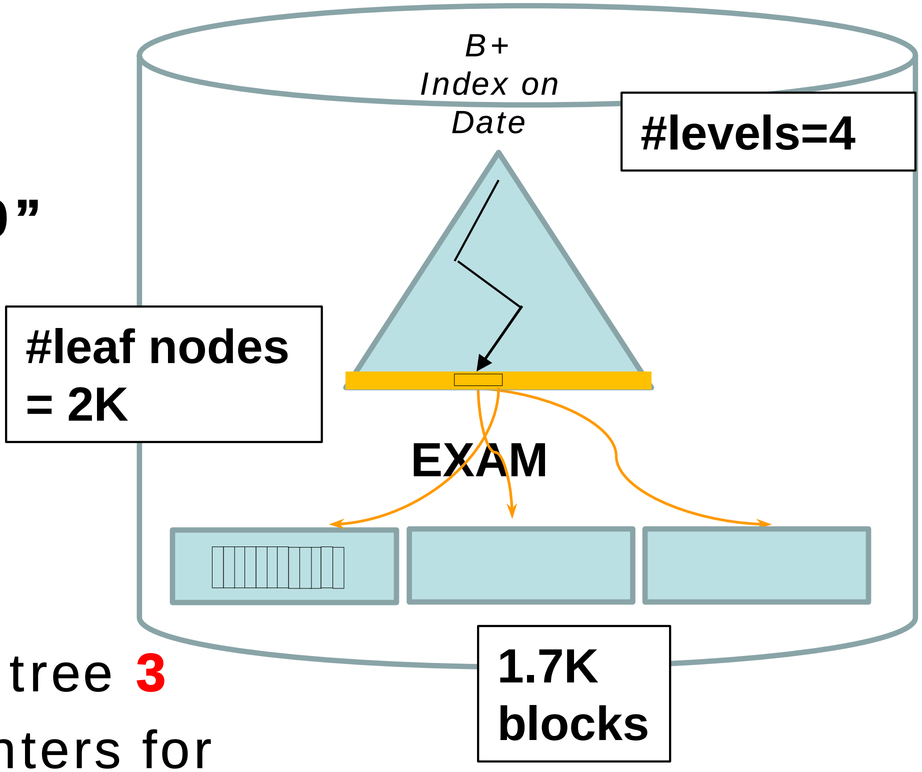
- Suppose val(LastName)=75K → 2 tuples/Lastname

- Cost:
  #of levels of the B+tree index **4**
  #of blocks to be accessed in STUDENT **2**

Total cost = 6

Cost of equality lookup on a secondary B+

SELECT *
FROM EXAM
WHERE Date="10/6/2019"



- Suppose val(Date)=500 → 1.8M/500 = 3.6K tuples/date
- Cost = sum of:
  - #of intermed. levels of the B+tree **3**
  - #of linked leaf nodes with pointers for the given date = 1.8M/2K = 900 pointers per leaf → *we need to access* ***3.6K/900=4*** *leaf nodes*
  - #of blocks to be accessed in EXAM = **3.6K**

→ Total=3.6K + 3 + 4 ~ 3.6K

Cost of lookups: intervals ($A_i < v$, $v_1 \le A_i \le v_2$)

- **Sequential** structures
  - Lookups are not supported (cost: a full scan)
    - Sequentially-ordered structures may have reduced cost
- **Hash** structures
  - Lookups based on intervals are not supported
- **Tree** structures
  - Supported if A_i is the index key attribute of the structure
  - The cost depends on
    - the storage type (primary/secondary)
    - the index key type (unique/non-unique)

Cost of interval lookup on a primary B+

- We consider a lookup for $v_1 \le A_i \le v_2$ as the general case
  - If $A_i < v$ or $v < A_i$ we just assume that the other edge of the interval is the first/last value in the structure
- The **root** is read first
  - then, a node per **intermediate** level is read, until...
- ...the first **leaf** node is reached, that stores tuples with $A_i = v_1$
  - **If** the searched tuples are all stored in that leaf block, stop
  - **Else**, continue in the leaf blocks chain until v_2 is reached

Cost: **1** block **per** intermediate **level +** as many **leaf blocks** as necessary to read all the tuples in the interval

Cost of interval lookup on a secondary B+

- We still consider a lookup for $v_1 \leq A_i \leq v_2$ as the general case
- The **root** is read first
  - then, a node per **intermediate** level is read, until...
- ...the first **leaf** node is reached, that stores **the pointers pointing** to the blocks containing the tuples with $A_i = v_1$
  - **If** all the **pointers** (up to v_2) are in that leaf block, stop
  - **Else**, continue in the leaf blocks chain until v_2 is reached

Cost: **1** block **per** intermediate **level +** as many **leaf blocks** as necessary to read all pointers in the interval **+ 1 block per** each such **pointer** (to retrieve the tuples)

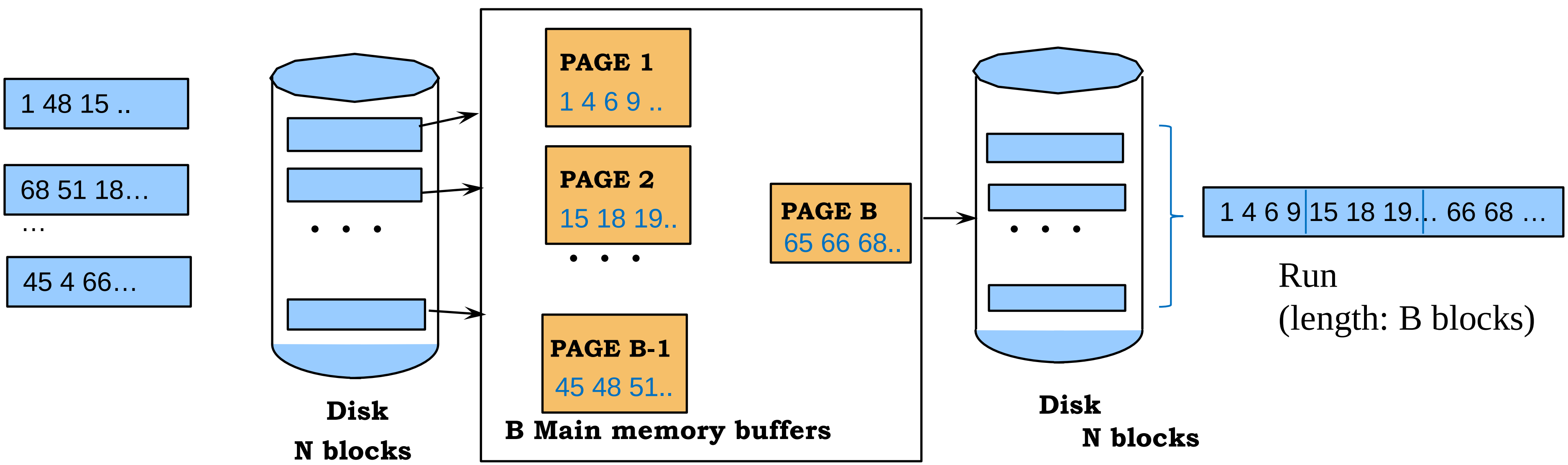
Sort

- This operation is used for ordering the data according to the value of one or more attributes. We distinguish:
  - Sort in **main memory**, typically performed by means of ad-hoc algorithms (merge-sort, quicksort, …)
  - Sort of **large files**, performed with different algorithms such as e.g., the external merge-sort:
    - Data do not fit into the main memory
    - Idea: first, sort chunks of data small enough to fit in main memory
    - Then, merge sorted parts

External Merge Sort

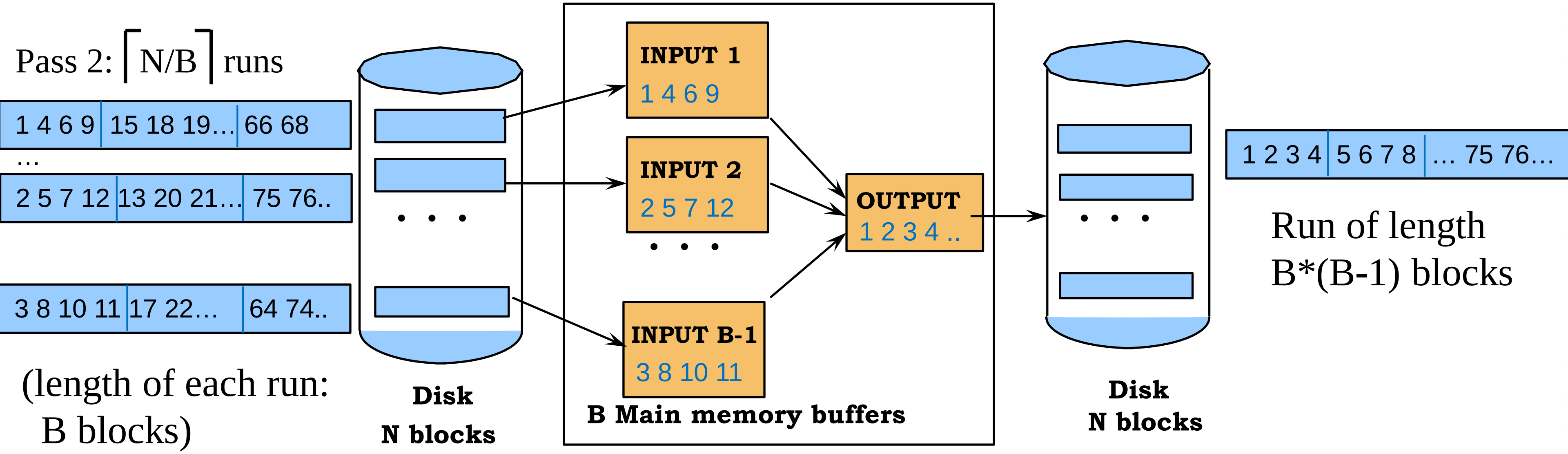
- To sort a file stored in N blocks using B buffer pages:
  - Pass 1:
    - read B blocks at a time into main memory and sort them;
    - write sorted data to a *run* file

    → total runs = $\lceil N/B \rceil$



External Merge Sort

- Pass 2, 3, 4, ...: Use B-1 blocks for input runs, 1 block for OUTPUT
- Repeat
  - Select the first record (in sort order) among all buffer pages and write it to the output buffer; if the output buffer is full, write it to disk
  - Delete the record from its input buffer page. If the buffer page is empty, read next block of the run into the buffer
- Until all input buffer pages are empty



External Merge Sort

- In each pass, contiguous groups of B-1 runs are merged
- Repeated passes are performed until all runs have been merged into one
- A pass reduces the number of runs by a factor of B-1 and creates runs longer by the same factor

- E.g., B=5, N=40 → initial runs=$\lceil 40/5 \rceil$=8 (having length 5 pages)
  - 40 read operations + 40 write operations
- In the next pass: with 4 input pages + 1 output page
  - First load 4 runs and create a new run having length 5*4=20 pages
  - Load the next 4 runs and create a new run having length 20 pages
    - $\lceil 8/4 \rceil$=2 sorted runs of 20 pages each → 40 + 40 I/O operations
- In the next pass load the 2 final runs into the final sorted run of length 20*2
  - $\lceil 2/4 \rceil$=1 sorted runs of 40 pages → 40 + 40 I/O operations
  - Total passes = 3

External sorting

- $\text{Cost} = 1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil$

In the example: $1+\lceil \log_4 8 \rceil$

→ Cost = 2N * (# of passes)

# of buffers in main memory

# of blocks

N	B=3	B=5	B=9	B=17	B=129	B=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

# of passes

Join Methods

- Joins are the **most frequent (and costly)** operations in DBMSs
- There are several join strategies, among which:
  - *nested-loop*, *merge-scan* and *hashed*
  - These three join methods are based on scanning, ordering and hashing
- The “best” strategy is chosen based on several aspects...

Running example

STUDENT (<u>Id</u>, Name, LastName, Email, City, Birthdate, Sex)

150K tuples

Each tuple is 95 bytes long

$t_{STUDENT}$ = 87 tuples per block (block factor)

$b_{STUDENT}$ = 1700 blocks

EXAM (<u>SId, CourseId</u>, Date, Grade)

1.8 M tuples

Each tuple is 24 bytes long

t_{EXAM} = 340 tuples per block (block factor)

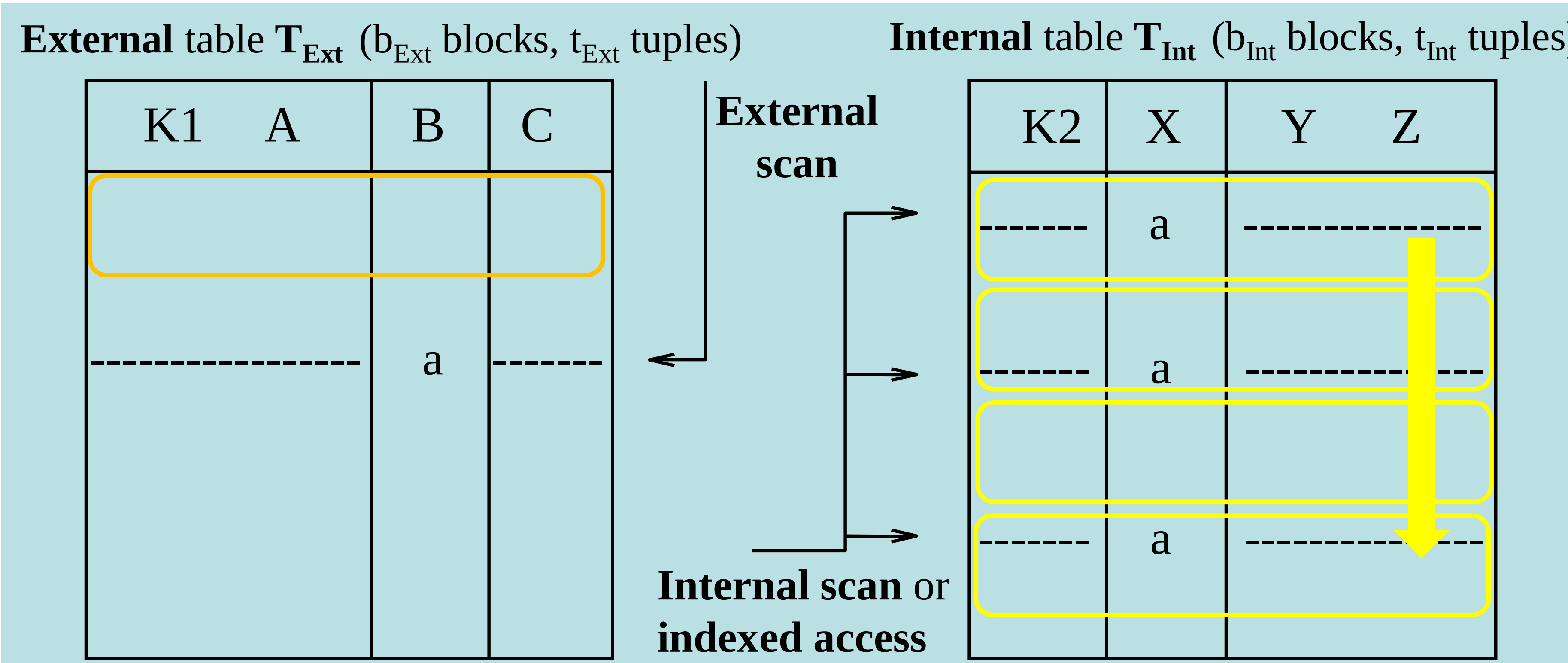
b_{EXAM} = 5300 blocks

Equality Joins With One Join Column

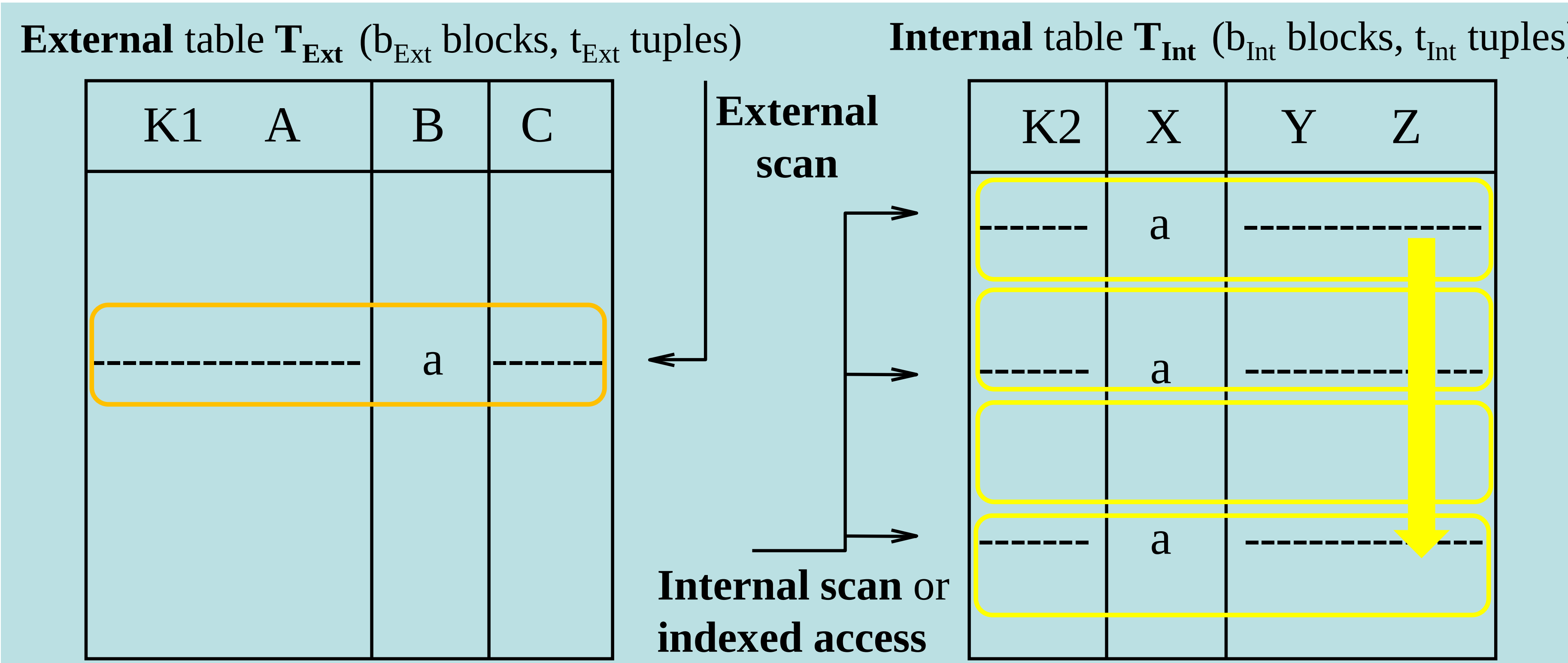
```
SELECT  *
FROM    Student, Exam
WHERE   ID=SID
```

- In algebra: Student $\bowtie_{ID=SID}$ Exam
  - Common! Must be carefully optimized
  - Student $\times$ Exam is large → Student $\times$ Exam followed by a selection is inefficient

Nested-Loop join (NL , S&L)



Nested-Loop join (NL , S&L)



Cost of simple Nested-Loop joins

- A nested loop join compares the tuples of a block of table T_{Ext} with all the tuples of all the blocks of T_{Int} before moving to the next block of T_{Ext}
  - We always assume that the buffer does not have enough available free pages to host more than a few blocks
  - The cost is **quadratic** in the size of the tables
    - $C = b_{Ext} + b_{Ext} \times b_{Int} = b_{Ext} \times (1 + b_{Int}) \approx b_{Ext} \times b_{Int}$
  - **IF** one of the tables is <u>small enough to fit in the buffer</u>, then it is chosen as <u>internal</u> table ($C = b_{Ext} + b_{Int}$)

A simple Nested-Loop join

STUDENT (<u>Id</u>, Name, LastName, Email, City, Birthdate, Sex)
EXAM (<u>SID, CourseID</u>, Date, Grade)

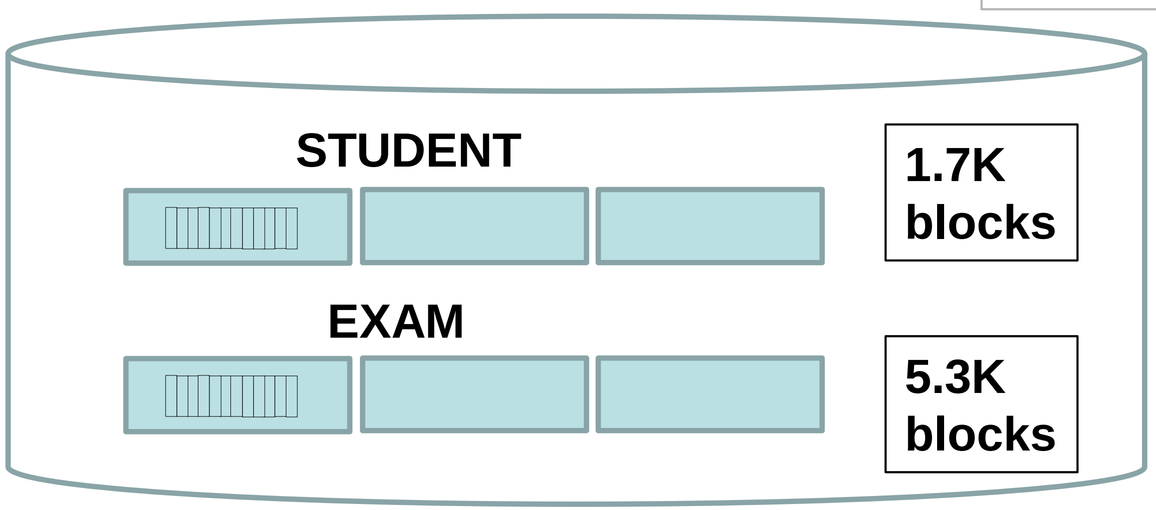
SELECT STUDENT.*
FROM STUDENT JOIN EXAM ON Id=SID

Cost = 1.7K + 1.7K*5.3K =~ **9M** I/O accesses

Student external table
Exam internal table

Cost = 5.3K + 5.3K*1.7K =~ **14M** I/O accesses

Exam external table
Student internal table



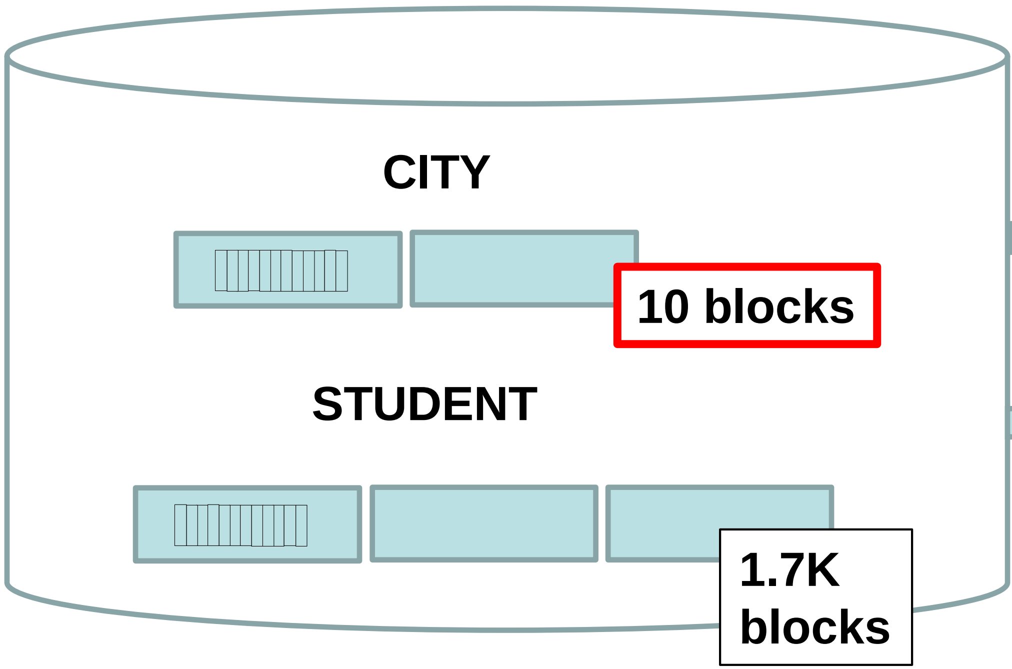
Nested-Loop joins with cache

STUDENT (<u>Id</u>, Name, LastName, Email, City, Birthdate, Sex)

CITIES (<u>City</u>, Region, Description)

b_{CITIES} = 10 blocks total!!!

SELECT STUDENT.* FROM STUDENT NATURAL JOIN City



Cache CITIES (10 blocks)
Cost = **10** I/O accesses

For all the 1.7K blocks of STUDENT, compute the join of their tuples with the cached tuples
Cost=**1.7K** I/O accesses for STUDENT

Total cost = 10 + 1,7K =~ **1,7K**

Scan and lookup (indexed nested loop)

- **IF** one table supports indexed access in the form of a lookup based on the join predicate, then this table can be chosen as internal, exploiting the predicate to extract the joining tuples without scanning the entire table
  - If both tables support lookup on the join predicate, the one for which the predicate is more selective is chosen as internal
- Cost:
  - full scan of the **blocks** of the external table +
  - cost of lookup for each **tuple** of the external table onto the internal one, to extract the matching tuples
    - C = b_{Ext} + $\mathbf{t_{Ext}} \times \mathit{cost\_of\_one\_indexed\_access\_to\_}T_{Int}$

Adding a condition – no indexes

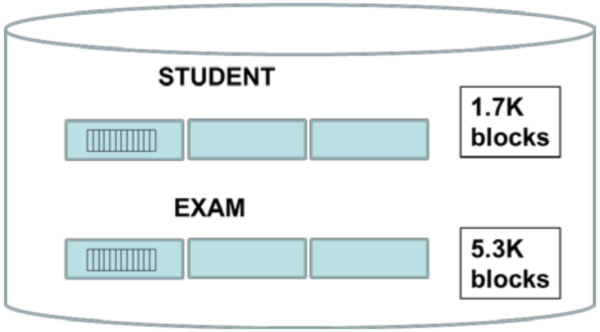
STUDENT (<u>Id</u>, Name, LastName, Email, **City**, Birthdate, Sex)
EXAM (<u>SID, CourseID</u>, Date, **Grade**)

SELECT STUDENT.*
FROM STUDENT JOIN EXAM ON Id=SID
WHERE City='Milan' AND Grade='30'

val(City)=150
val(Grade)=17

Option 1: Scan Student, evaluate City='Milan' + Lookup on Exam

- Read all the STUDENT blocks: **1.7K**
- Filter <u>students from Milan</u> → 150K tuples/150 different cities = **1k**
- For 1k students compute the join with the tuples of all the 5.3K blocks in EXAM



Total cost:
1,7K + 1K*5.3K
=~ **7M** I/O accesses

Adding a condition – no indexes

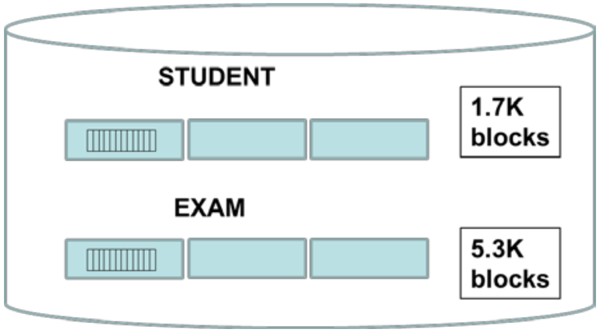
STUDENT (<u>Id</u>, Name, LastName, Email, **City**, Birthdate, Sex)
EXAM (<u>SID, CourseID</u>, Date, **Grade**)

SELECT STUDENT.*
FROM STUDENT JOIN EXAM ON Id=SID
WHERE City='Milan' AND Grade='30'

val(City)=150
val(Grade)=17

Option 2: Scan Exam, evaluate Grade='30' + Lookup on Student

- Read all the EXAM blocks: **5.3K**
- Filter Exams with Grade='30' → 1.8M/17 different grades=106K
- For 106K exams compute the join with 1.7K blocks in STUDENT



Total cost:
5,3K + **106K***1.7K

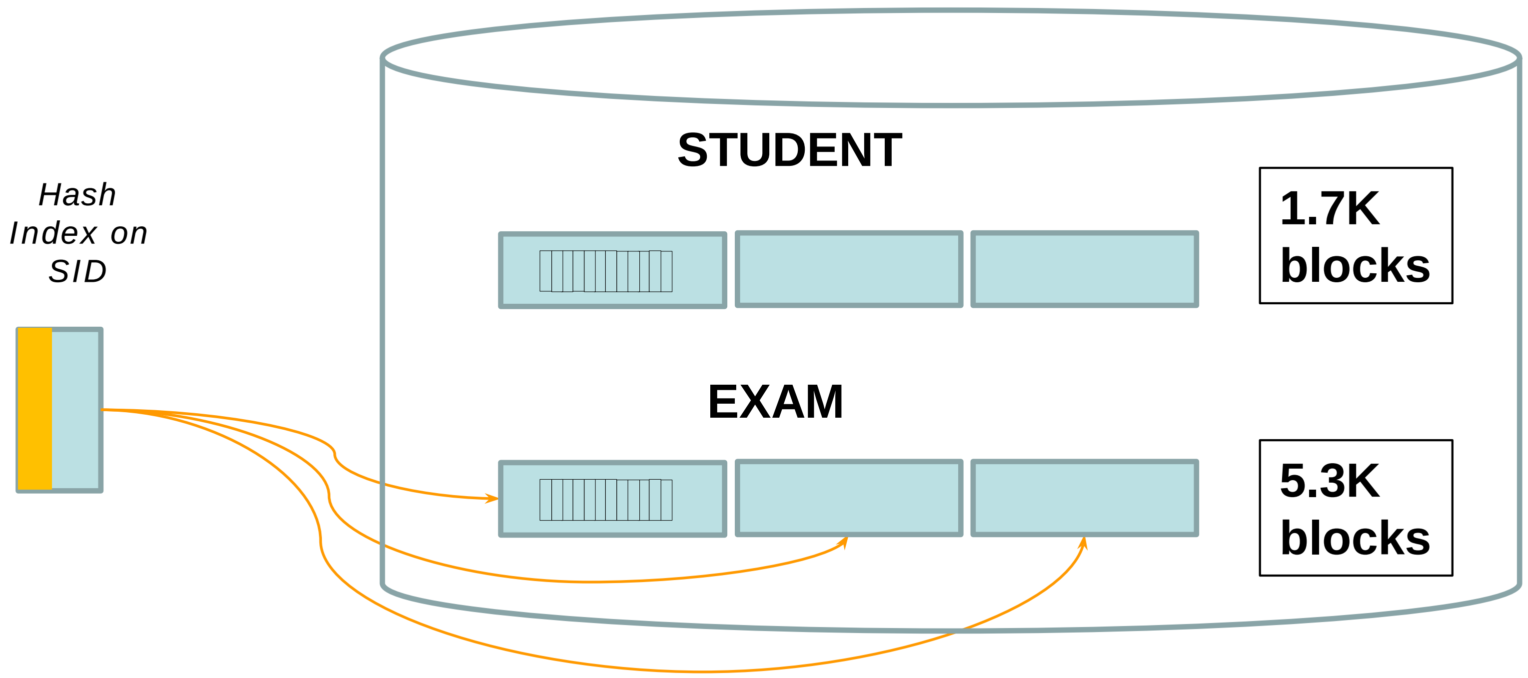
106K >> 5.3K!!!

Scan & Lookup exploiting an index

STUDENT (Id, Name, LastName, Email, City, Birthdate, Sex)
EXAM (SID, CourseID, Date, Grade)

SELECT STUDENT.*
FROM STUDENT JOIN EXAM ON Id=SID
WHERE City='Milan' AND Grade='30'

val(City)=150

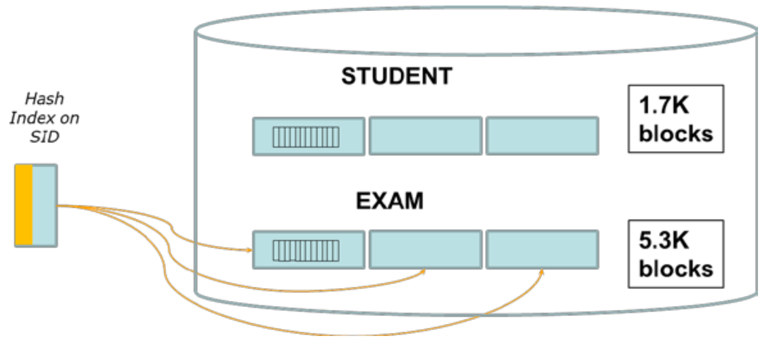


A simple Nested-Loop join

Improve option 1:

Scan Student, evaluate City='Milan' +
Lookup on Exam exploiting the hash index

- Read all the STUDENT blocks **1.7K**
- Filter students from Milan → 150K tuples/150 different cities = 1k
- For 1k students use the hash index: for each student
  - 1 access to the hash index (without overflow chain)
  - How many exams per student? 1.8M exams / 150K students=12
  - Follow the 12 pointers to retrieve the corresponding exams (and grades)



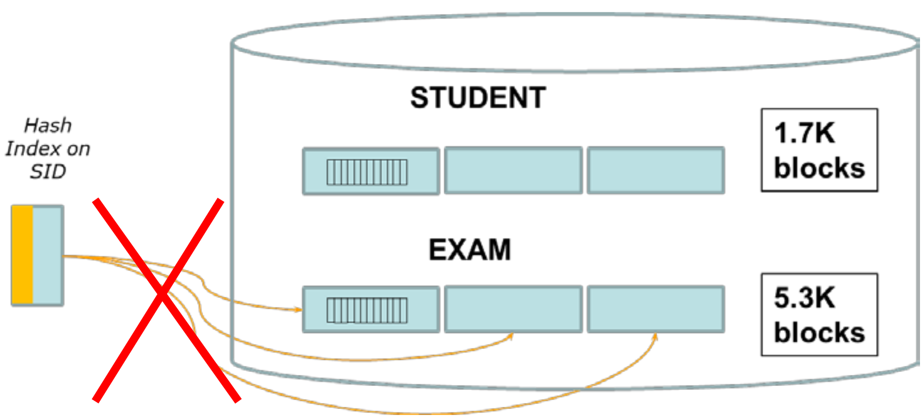
Total cost =
1.7K + 1K * (1+12) = 14.7K

A simple Nested-Loop join

SELECT STUDENT.*
FROM STUDENT JOIN EXAM ON Id=SID
WHERE City='Milan' ~~AND Grade='30'~~

val(City)=150

- Read all the STUDENT blocks **1.7K**
- Filter students from Milan → 150K tuples/150 different cities = 1k
- For 1k students use the hash index: for each student
  - 1 access to the hash index (without overflow chain)
  - We do not need to access the whole tuples to do the join

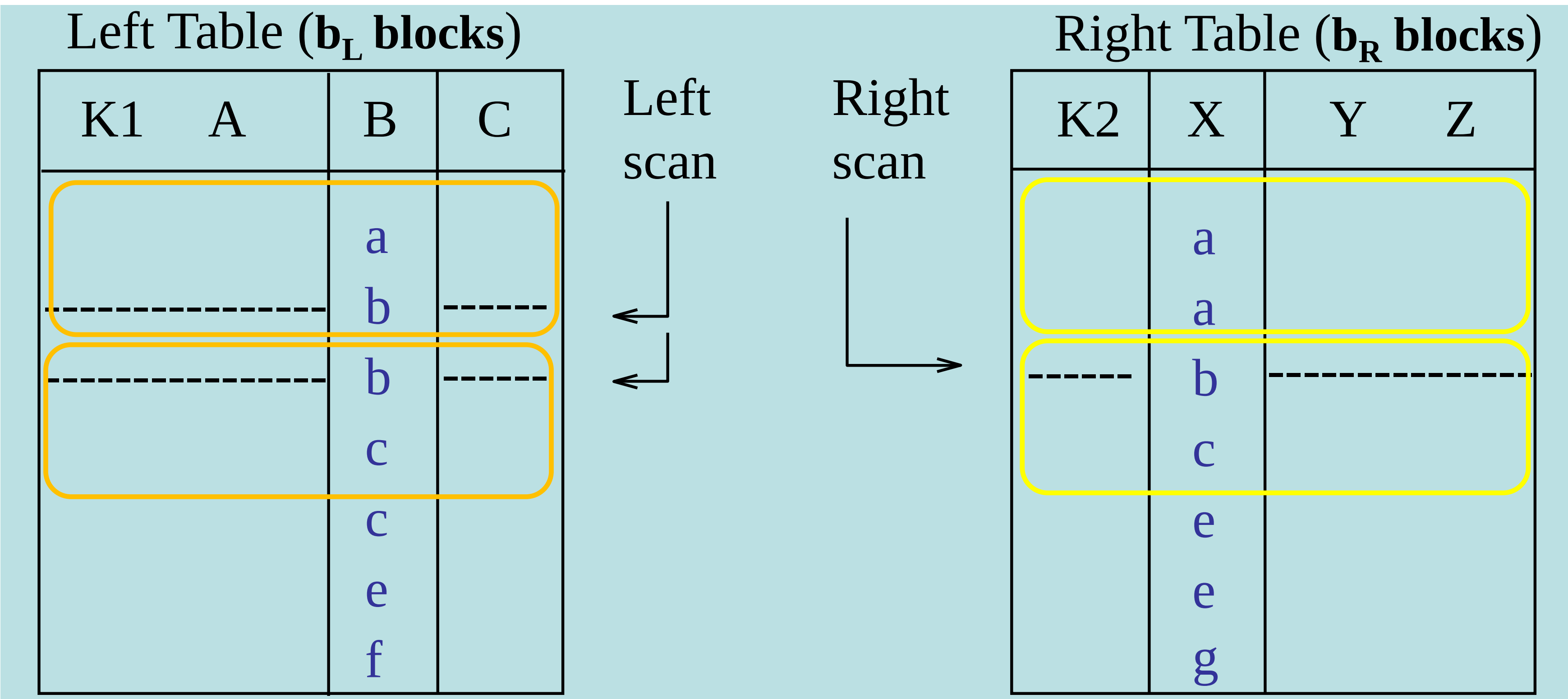


Total cost =
1.7K + 1K * (1~~+12~~) = 2.7K

Merge-Scan join

This join is possible <u>only if both tables are ordered according to the same key</u> attribute, that is also the attribute used in the join predicate

$$L \bowtie_{B=X} R$$



Sort-Merge scan Join

- (If not sorted) sort L and R on the join column
- Scan them to do a “merge’’, advancing on the tables with the least value
- Output result tuples <l, r>
- Cost?

Cost of M-S joins

- The cost is **linear** in the size of the tables

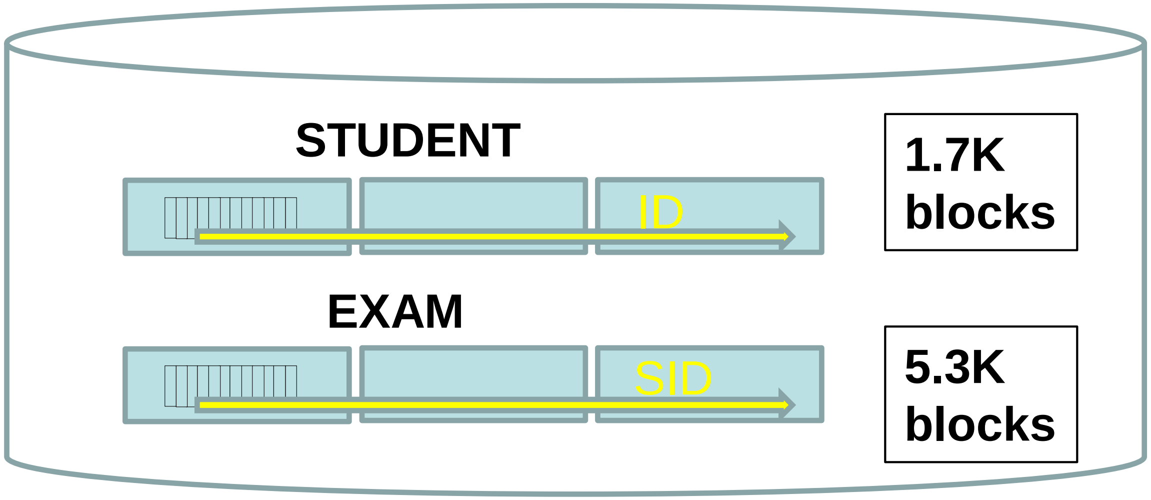
$$C = b_L + b_R$$

- If the primary storage is sequentially ordered wrt the join attribute or a B+ on the join attribute is defined, an ordered full scan of both tables is possible
- If one or both tables need to be sorted, add also the cost for sorting
  - $2\ b_L$ * (# of passes)
  - $2\ b_R$ * (# of passes)

Example: Cost of M-S joins

SELECT STUDENT.*
FROM STUDENT JOIN EXAM ON Id=SID

- Suppose
  - STUDENT: Sequentially-ordered by ID
  - EXAM: Sequentially-ordered by SID

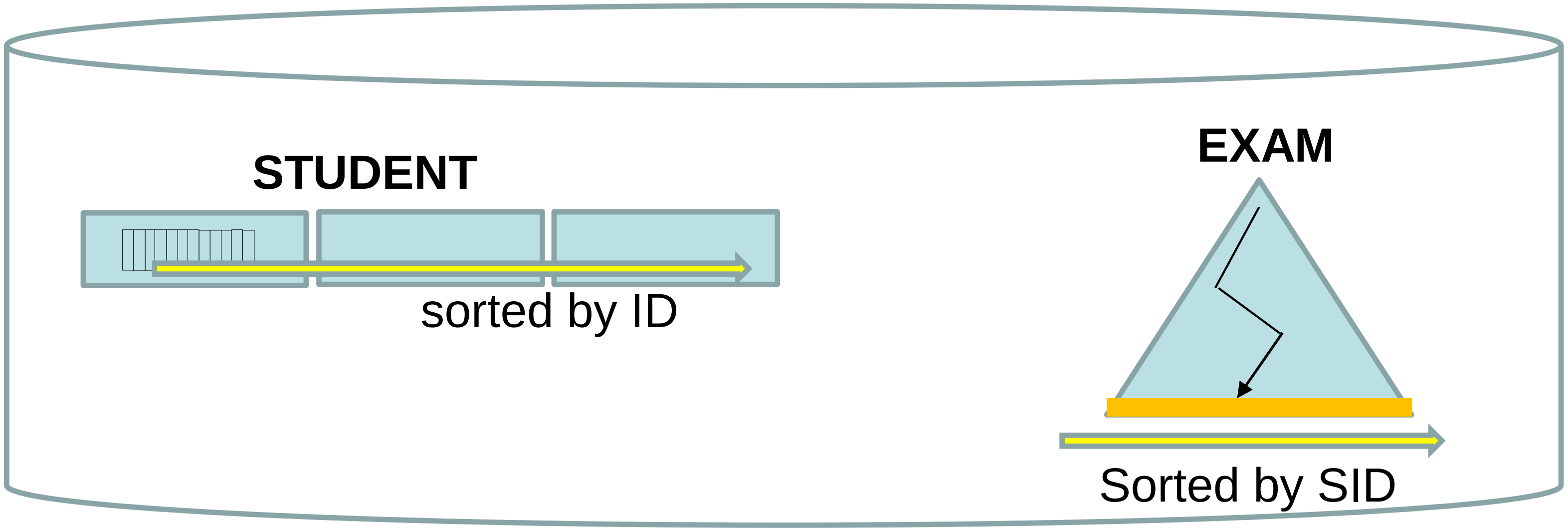


$$C = b_{Student} + b_{Exam} = 1.7K + 5.3K = \mathbf{7K} \text{ I/Os}$$

Example: Cost of M-S joins

SELECT STUDENT.*
FROM STUDENT JOIN EXAM ON Id=SID

- Suppose
  - STUDENT: Sequentially-ordered by ID
  - EXAM: primary storage: **B+ on SID**

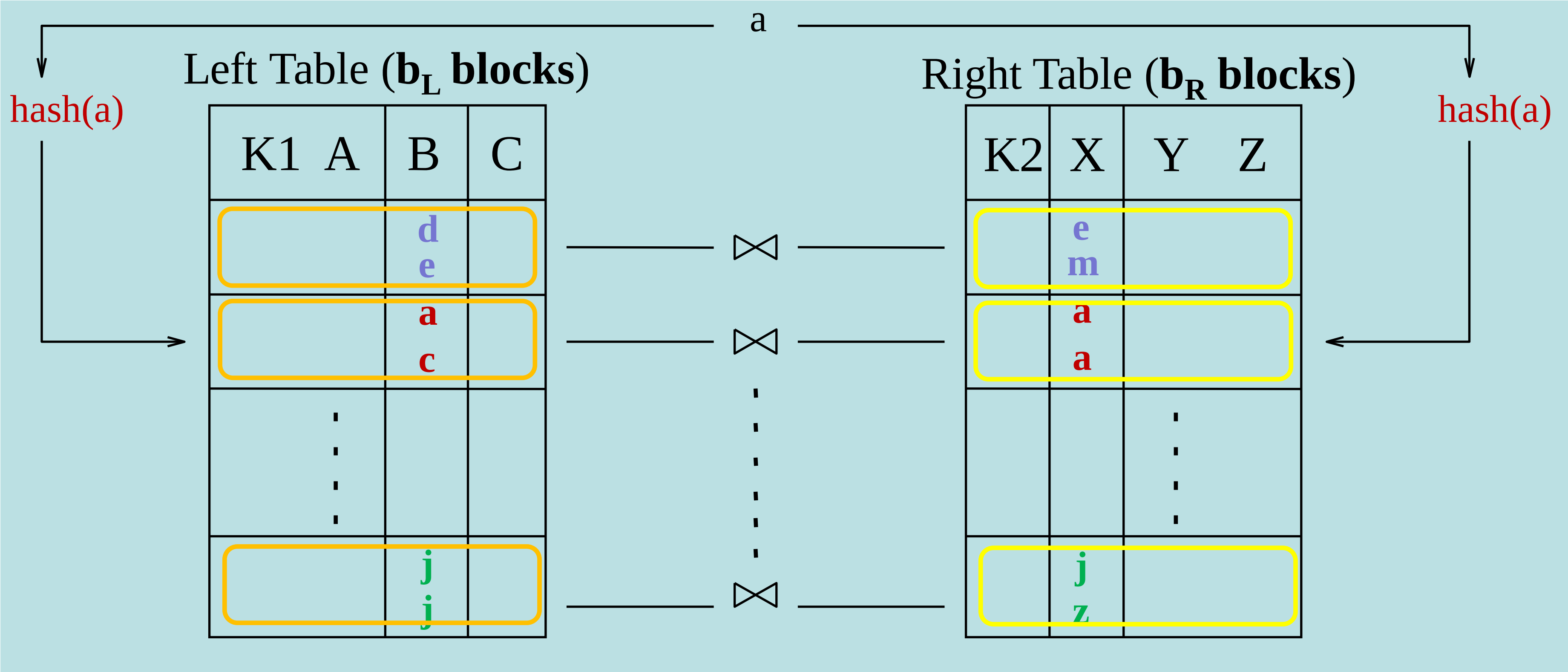


$$C = b_{Student} + \mathbf{\#LeafNodes}_{Exam}$$

Hashed join

This join is possible only if both tables are <u>hashed</u> according to the <u>same key (join) attribute</u>

The matching tuples can only be found in corresponding buckets



Cost of Hashed joins

- The cost is **linear** in the number of blocks of the hash-based structure
  - If the two hashes are both primary storages:

$$C = b_L + b_R$$

  - Note that the two hashes have the **same number of buckets**, but the number of blocks b_L and b_R may (slightly) differ due to overflows

COST-BASED OPTIMIZATION

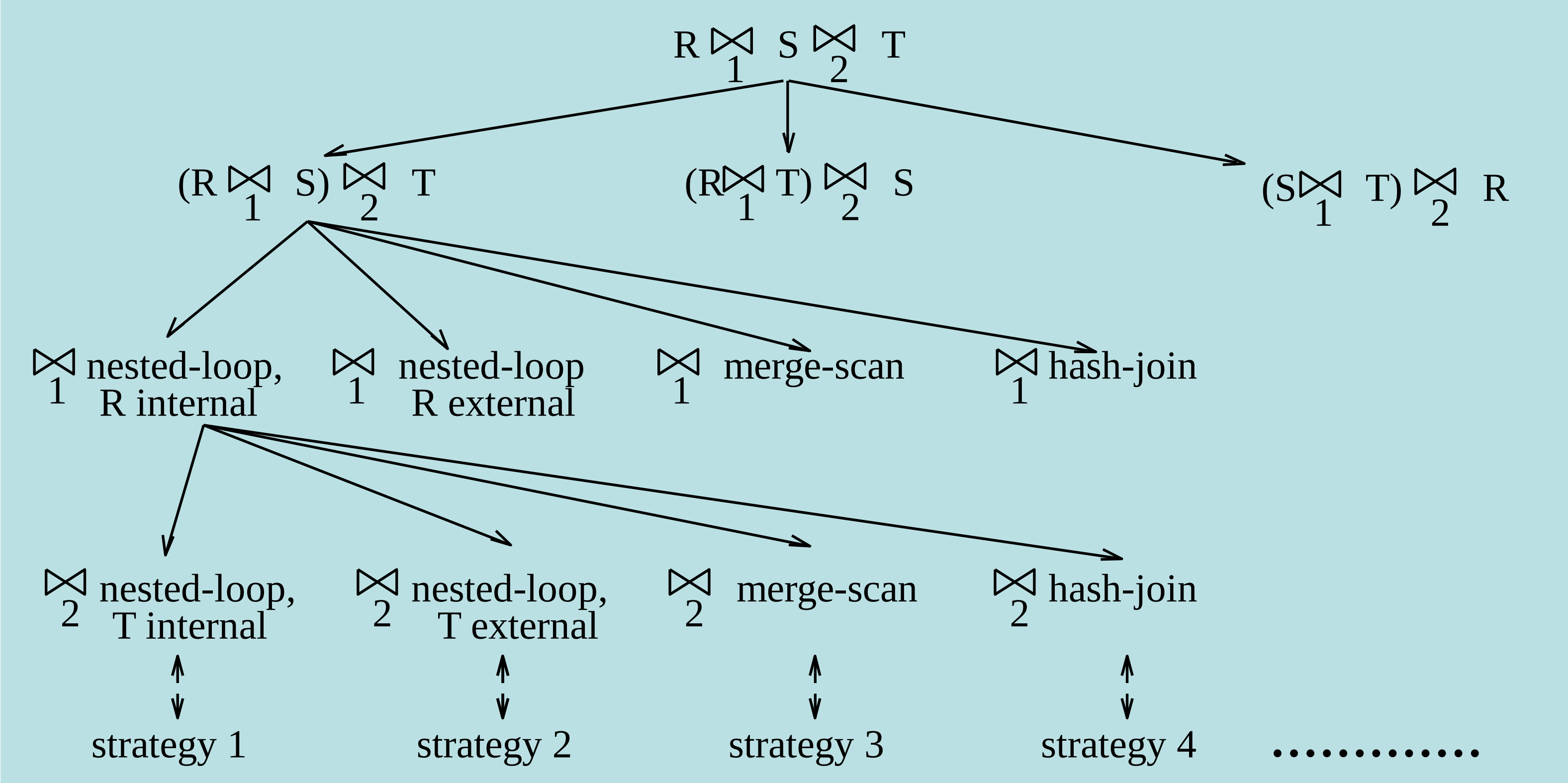
Cost-based optimization

- An optimization problem, whose decisions are:
  - The data access operations to execute (e.g., scan vs index access)
  - The order of operations (e.g., the join order)
  - The option to allocate to each operation (e.g., choosing the join method)
  - Parallelism and pipelining can improve performances

Approach to query optimization

- Optimization approach:
  - Make use of profiles and of approximate cost formulas
  - Construct a *decision tree*, in which
    - each node corresponds to a choice
    - each leaf node corresponds to a specific *execution plan*

An example of decision tree



Approach to query optimization

- Assign to each plan a cost:

$$C_{total} = C_{I/O}\, n_{I/O} + C_{cpu}\, n_{cpu}$$

- Choose the plan with the lowest cost, based on operations research (branch and bound)

- Optimizers should obtain ‘good’ solutions in a very short time

Approaches to query execution

- ***Compile and store***: the query is compiled once and executed many times
  - The internal code is stored in the DBMS, together with an indication of the dependencies of the code on the particular versions of catalog used at compile time
  - On relevant changes of the catalog, the compilation of the query is invalidated and repeated
- ***Compile and go***: immediate execution, no storage
  - Even if not stored, the code may live for a while in the DBMS and be available for other executions

Summary: Query Optimization

- Important task of DBMSs
- Goal is to minimize # I/O blocks
- Search space of execution plans is huge
- Heuristics based on algebraic transformation lead to good logical plan (e.g., apply first the operations that reduce the size of intermediate results), but no guarantee of optimal plan
- More details in other books (suggested: Elmasry-Navathe)

Determine the execution plan of your query

"EXPLAIN PLAN" SQL statement

- Oracle: http://docs.oracle.com/cd/E11882_01/server.112/e16638/ex_plan.htm
  - See also how the Query Optimizer works: http://docs.oracle.com/cd/B28359_01/server.111/b28274/optimops.htm
- SQLite: http://www.sqlite.org/eqp.html
- MySQL: http://dev.mysql.com/doc/refman/5.5/en/execution-plan-information.html
- MS SQL server: http://msdn.microsoft.com/en-us/library/ms176005(v=sql.105).aspx

Example of query plan in Oracle

ORACLE Database Express Edition

Home Logout Help

User: HR

Home > SQL > SQL Commands

Autocommit Display 10 Save Run

```
SELECT AVG(SALARY)
FROM (EMPLOYEES NATURAL JOIN DEPARTMENTS NATURAL JOIN LOCATIONS NATURAL JOIN COUNTRIES)
WHERE COUNTRY_NAME='Denmark'
```

Results Explain Describe Saved SQL History

Query Plan

Operation	Options	Object	Rows	Time	Cost	Bytes	Filter Predicates *	Access Predicates
SELECT STATEMENT			1	1	10	26		
SORT	AGGREGATE		1			26		
MERGE JOIN	CARTESIAN		253	1	10	6,578		
TABLE ACCESS	BY INDEX ROWID	DEPARTMENTS	1	1	1	7	"EMPLOYEES"."DEPARTMENT_ID" = ""DEPARTMENT_ID"	
NESTED LOOPS			11	1	5	286		
MERGE JOIN	CARTESIAN		107	1	4	2,033		
INDEX	FULL SCAN	COUNTRY_C_ID_PK	1	1	1	8	"COUNTRIES"."COUNTRY_NAME" = 'Denmark'	
BUFFER	SORT		107	1	3	1,177		
TABLE ACCESS	FULL	EMPLOYEES	107	1	3	1,177		
INDEX	RANGE SCAN	DEPARTMENTS_IDX1	2	1	0		"DEPARTMENTS"."MANAGER_ID" IS NOT NULL	"EMPLOYEES"."MANAGER_ID" = "DEPARTMENTS"."MANAGER_ID"
BUFFER	SORT		23	1	9			
INDEX	FAST FULL SCAN	LOC_CITY_IX	23	1	0			

* Unindexed columns are shown in red

Graphical query plan in MS SQL SMS