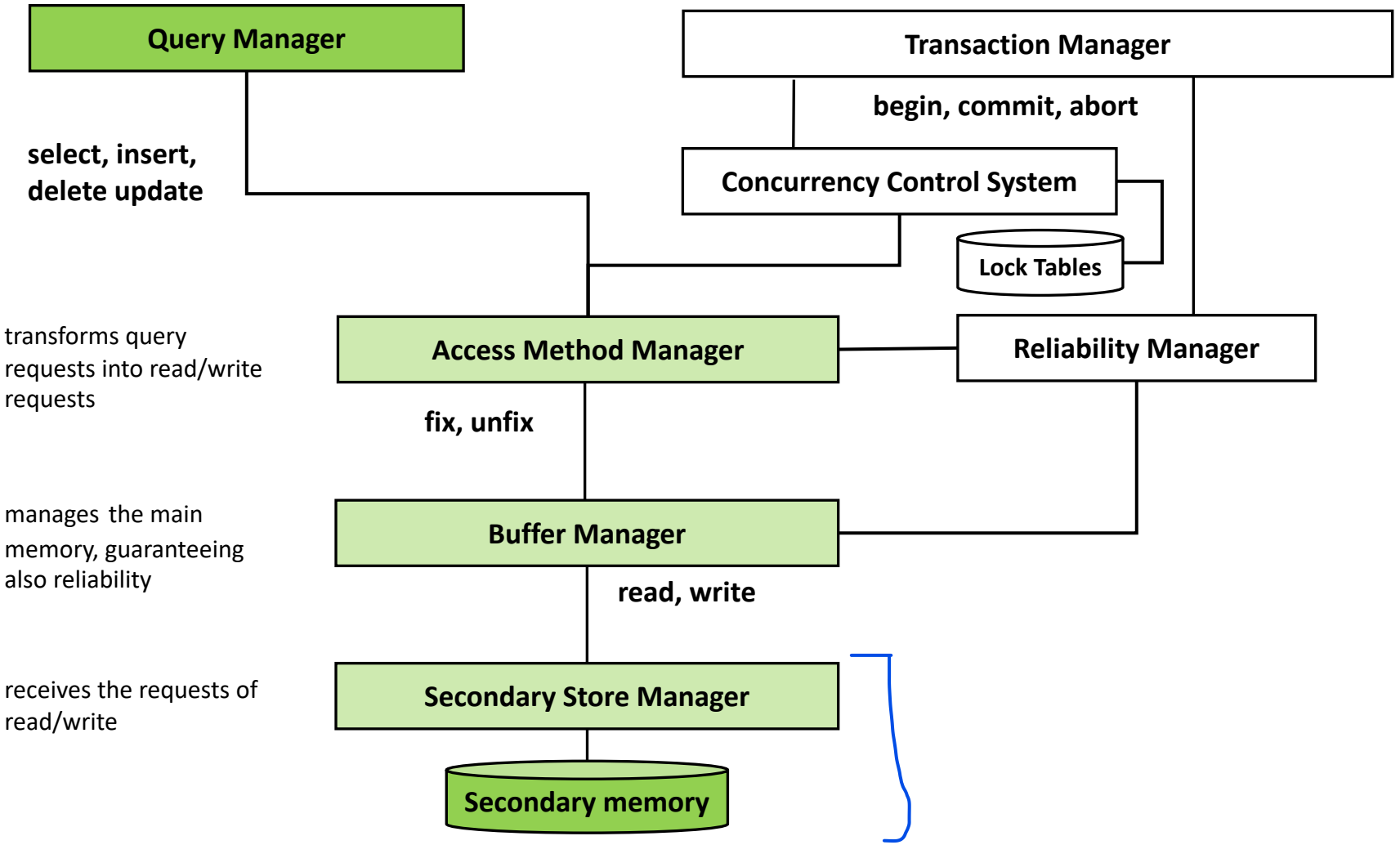
Databases 2

Physical data structures and query optimization

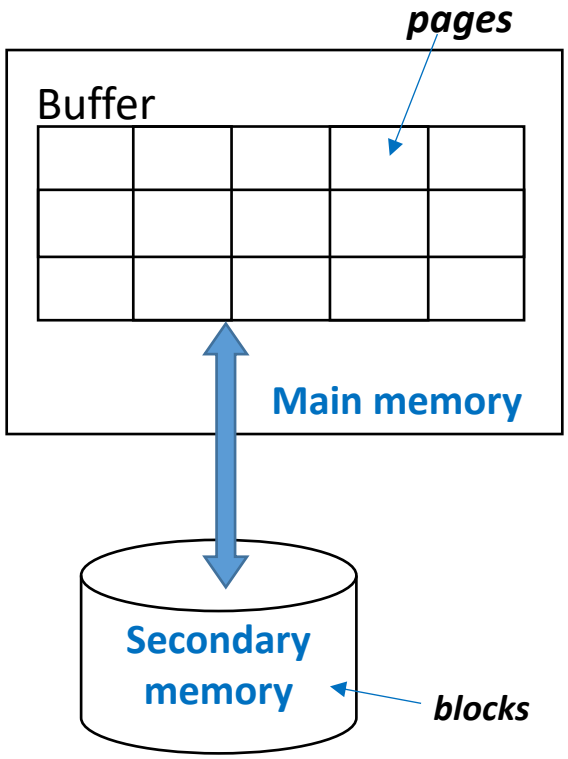
Query management in the DBMS architecture



DATA ACCESS and COST MODEL

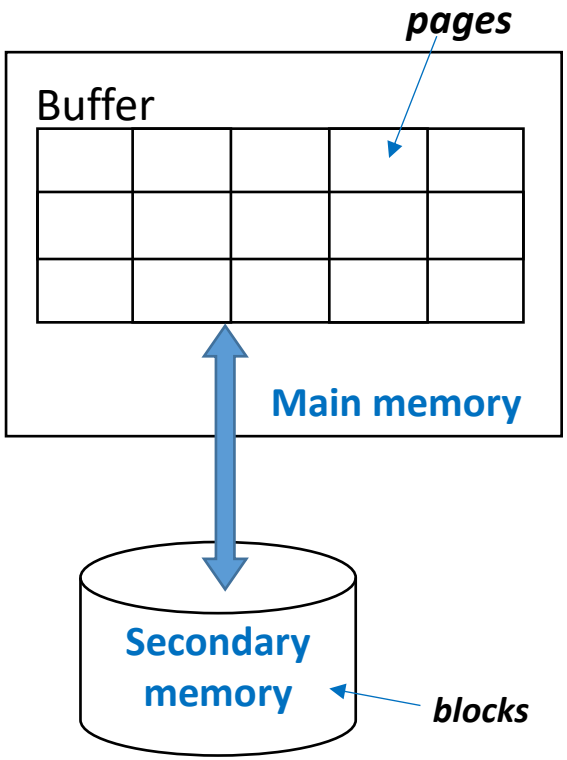
Main and Secondary memory

- 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
- Page: unit of storage in main memory
- Block: unit of storage in secondary memory



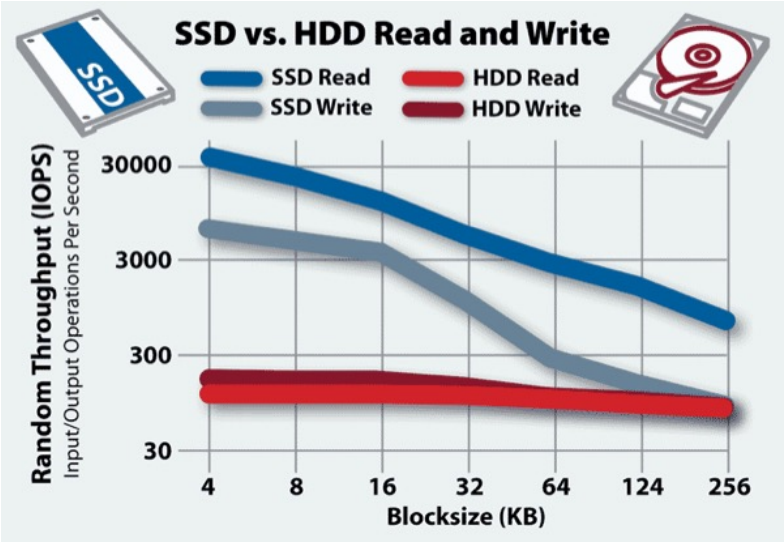
Pages and blocks

- Secondary memory devices are organized in blocks of (usually) fixed length
  - order of magnitude: a few KBytes
- We make the *assumption* that the size of a page equals the size of a block
  - In real systems a page may contain several blocks
- I/O operation: moving a block from secondary to main memory and vice-versa



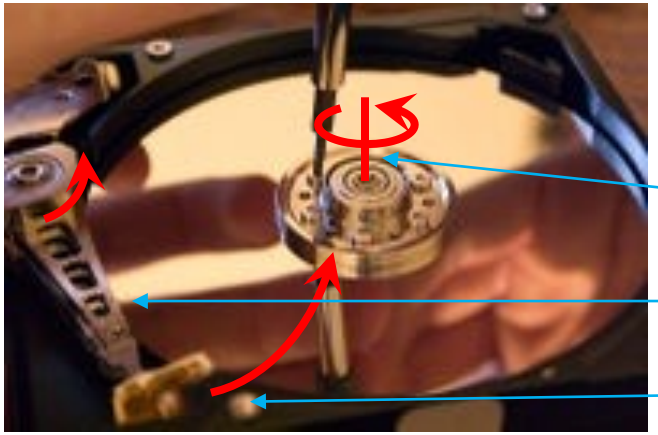
I/O operations

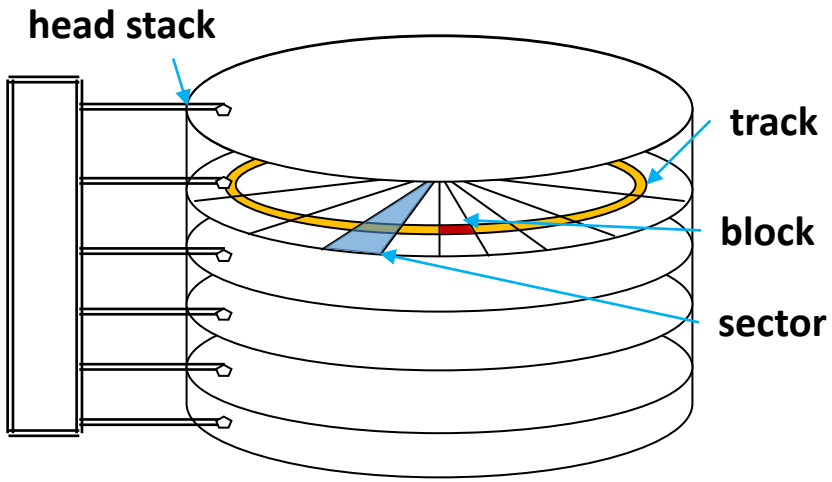
- How long does it take to read a block from a disk?
  - It depends on the technology
    - Mechanical hard drives
    - Solid State Drives (SSD)
      - Fast read/write speed
  - SSD are more expensive
    - For very large datasets, cheap storage is the only viable option



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 on the correct track
  - latency time (2-8ms) - disc rotation on the correct sector
  - 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
- **Cost of a query = #blocks** to be moved to execute a query

DBMS and file system

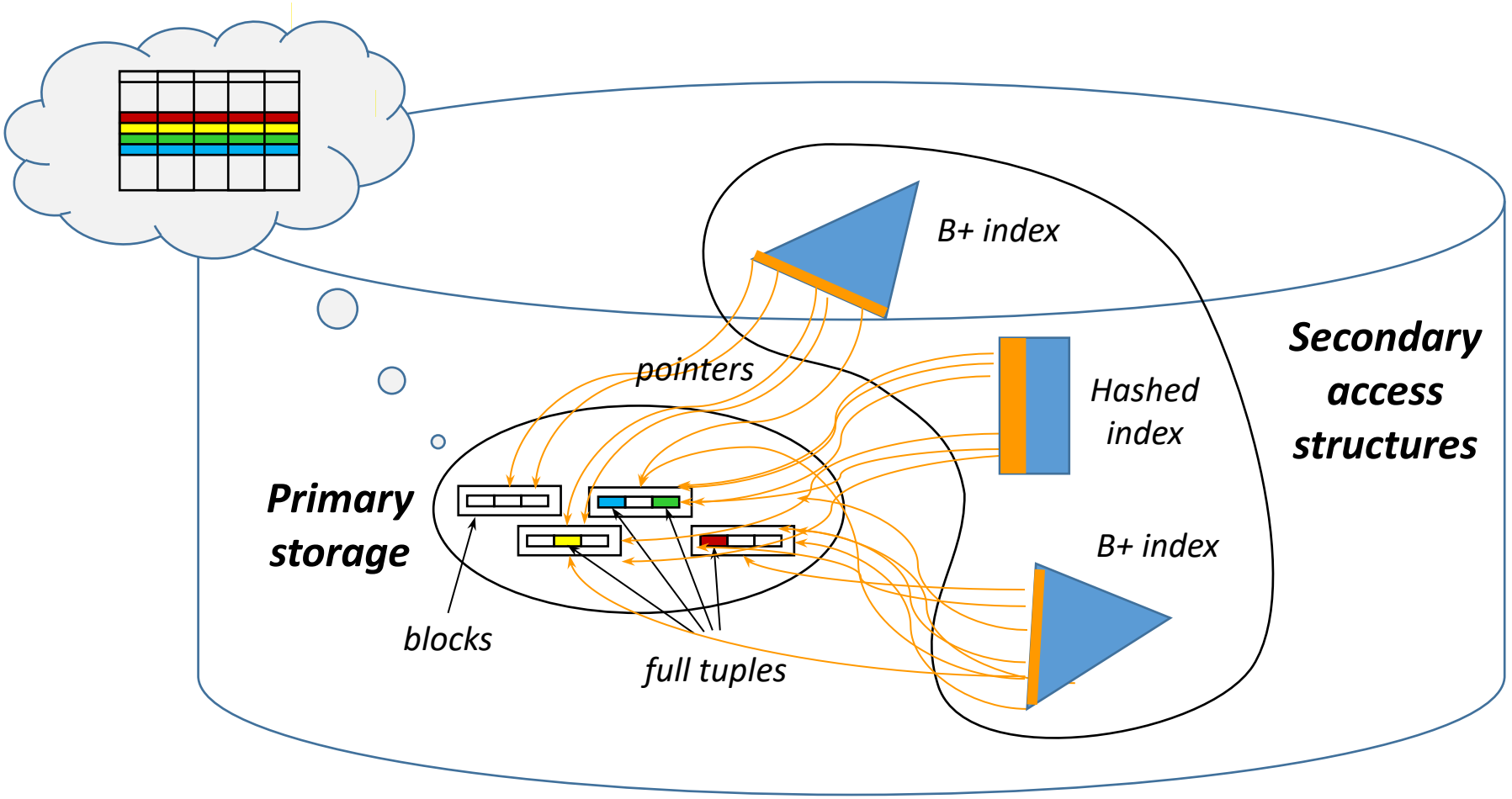
- File System (FS): component of the OS which manages access to secondary memory
- DBMSs make limited use of FS functionalities (create/delete files, read/write blocks)
- The **DBMS directly manages the file organization**, both in terms of the distribution of records within blocks and with respect to the internal structure of each block
  - A DBMS may also control the physical allocation of blocks onto the disk (for faster sequential reads, fine grain control of write operation execution time, reliability, etc.)

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 data structure**
- Access methods have their own data structures to organize data
  - Each table is stored into exactly one **primary** physical data structure, and contains actual tuples
  - may have one or many optional **secondary** access structures contain indices that point to actual tuples

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**	Not used
Hash-based structures	In some DBMSs (e.g., Oracle hash clusters, IDM DB2 “organize by hash” tables)	Frequent
Tree-based structures	Obsolete/rare	**Typical**

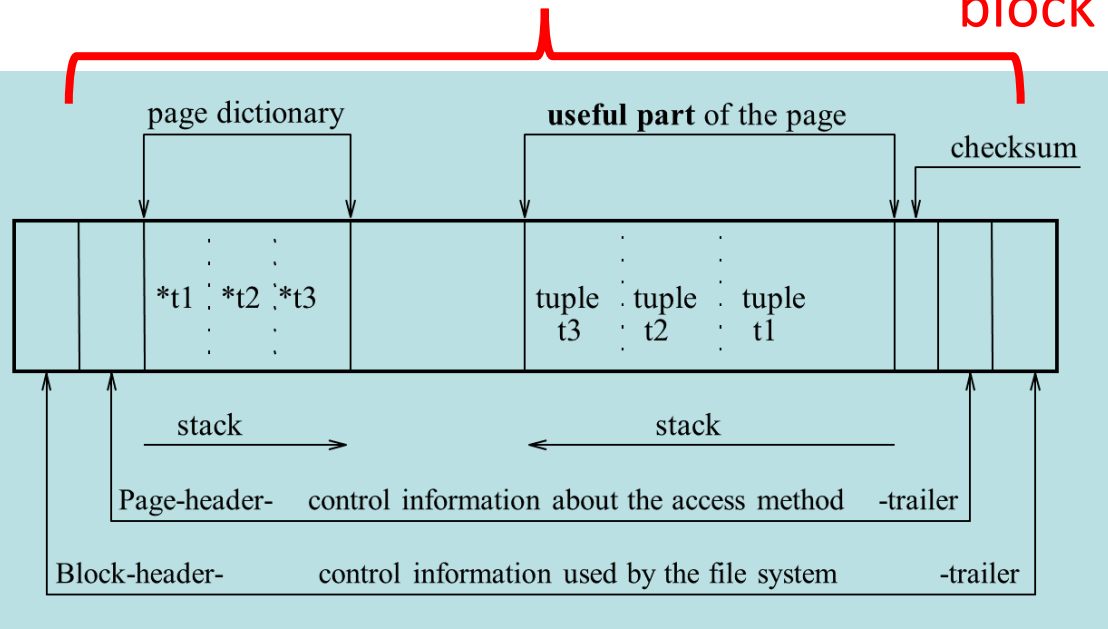
- https://docs.oracle.com/cd/B19306_01/server.102/b14231/hash.htm
- http://db2portal.blogspot.com/2012/07/db2-hashing-and-hash-organized-tables.html
- https://oracle-base.com/articles/8i/index-organized-tables

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 in blocks

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

How many tuples in a block: Block Factor

- Block factor B: the number of tuples within a block
- Fundamental for estimating cost of queries
  - SR: Average size of a record/tuple (assuming "fixed length record")
  - SB: Size of a block
    - if SB > SR, there may be many tuples in each block:
  - $B = \lfloor SB / SR \rfloor$ ($\lfloor x \rfloor$ = floor(x))
- The rest of the space can be
  - Used: tuples spanned between blocks (hung-up tuples/ records)
  - Not used: unspanned records

Buffer manager primitives

- Operations are performed in main memory and affect pages
- In our cost model we assume pages of equal size and organization as blocks
- Operations are:
  - Insertion and update of a tuple
    - may require a reorganization of the page or usage of a new page
    - also update to a field may require reorganization (e.g., varchar)
  - Deletion of a tuple
    - often carried out by marking the tuple as ‘invalid’ and triggering later reorganization of page asynchronously
  - 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 (access to next element is supported)
- Blocks can be contiguous on disk or sparse
- Two cases:
  - Entry-sequenced organization: sequence of tuples dictated by their order of entry
  - Sequentially-ordered organization: tuples ordered according to the value of a key (one or more attributes)

Entry-sequenced sequential (a.k.a. heap) structure

- Effective for
  - Insertion, which does not require shifting
  - Space occupancy, as it uses all the blocks available for files and all the space within the blocks
  - Sequential reading and writing (**select * from T**)
    - Especially if the disk blocks are contiguous (seek & latency times reduced)
    - Only if all (or most of) the file is to be accessed
- Non-optimal for
  - Searching specific data units (**select * from T where…**)
    - May require scanning the whole structure
    - But can be used efficiently **with indexes**

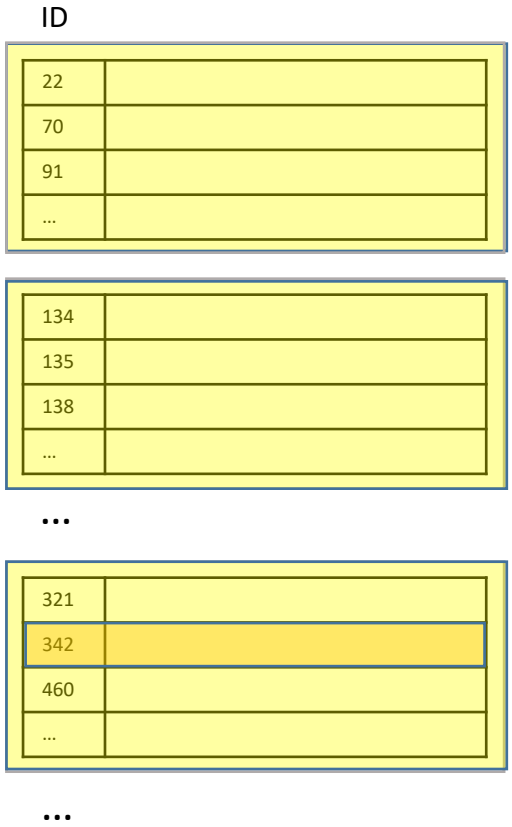
    it is a linear search: scan until I find
  - Updates that increase the size of a tuple (“shifts” required)
    - Shift may require storage in another block
    - Alternative approach: delete old version and insert new one

Sequentially-ordered sequential structure

- Tuples are sorted based on the value of a key field
- Effective for
  - Range queries that retrieve tuples having key in an interval
  - Order by and group by queries exploiting the key
- Problem: insertions & updates that increase the size
  - Reordering of the tuples within the block, if space allows
  - Techniques to avoid global reordering:
    - Differential files + periodic merging
    - Free space in the block at loading → 'local reordering' operations
    - Overflow file: new tuples are inserted into blocks linked to form an overflow chain (general principle used also in other organizations)

Example of query on an entry-sequenced sequential structure

Proceed sequentially until we found the tuple we were looking for or we have scanned the entire table




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

#I/O operations = # of read blocks

Comparison

	Entry-sequenced	Sequentially-ordered
INSERT	Efficient	Not efficient
UPDATE	Efficient (if data size increases → delete + insert the new version)	Not efficient if data size increases
DELETE	"Invalid"	"Invalid"
TUPLE SIZE	Fixed or variable	Fixed or variable
SELECT * FROM T WHERE key …	Not efficient	More efficient

Entry-sequenced organization is the most common solution, but **PAIRED** with secondary access structures

Hash-based access structures

- Efficient associative access to data, based on the value of a key
- A hash-based structure has N_B buckets (with N_B << number of data items)
  - A bucket is a unit of storage, typically of the size of 1 block
  - Often buckets are stored adjacently in the file

How hash-structures work: "I give you the key, the hash table gives me the bucket where the value is stored"

- 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 in the hash structure (hash table)
- Efficient for
  - Tables with small size and (almost) static content
  - Point queries: queries with equality predicates on the key
- Inefficient for
  - Range queries and full table queries
  - Tables with very dynamic content

How hash-based structures work

hash(fileId, Key): BlockId

- The implementation consists of two parts
  - folding, transforms the key values (e.g., strings) so that they become positive integer values, uniformly distributed over a large range
  - hashing transforms the positive 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 (=bucket) 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 last bucket
  - Open hashing (separate chaining):
    - a new bucket is allocated for the same hash result, linked to the previous one

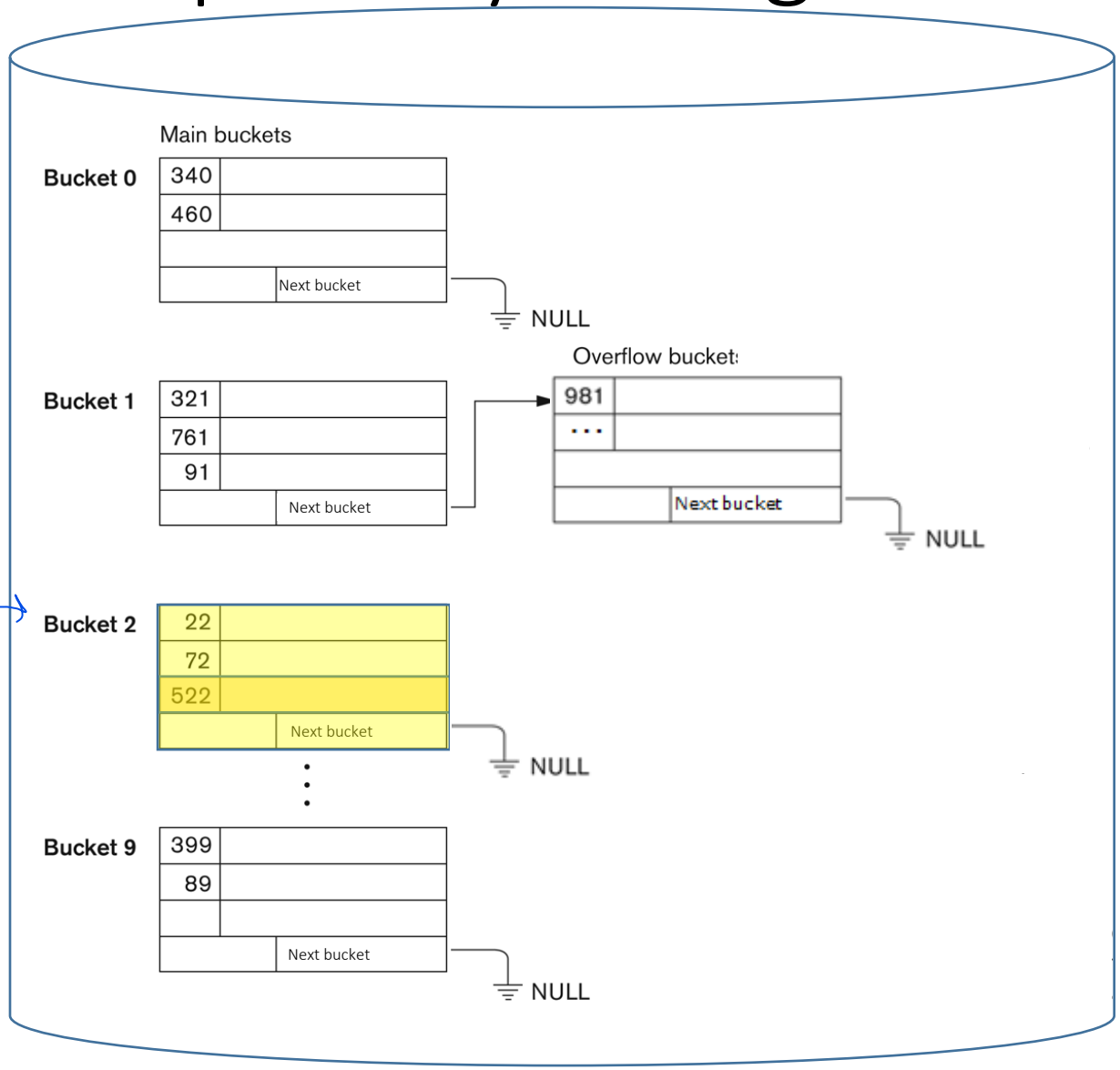
Hash table for primary storage

Hash function
h(K) = K mod 10

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

h(522) = 2

#I/O operations = 1



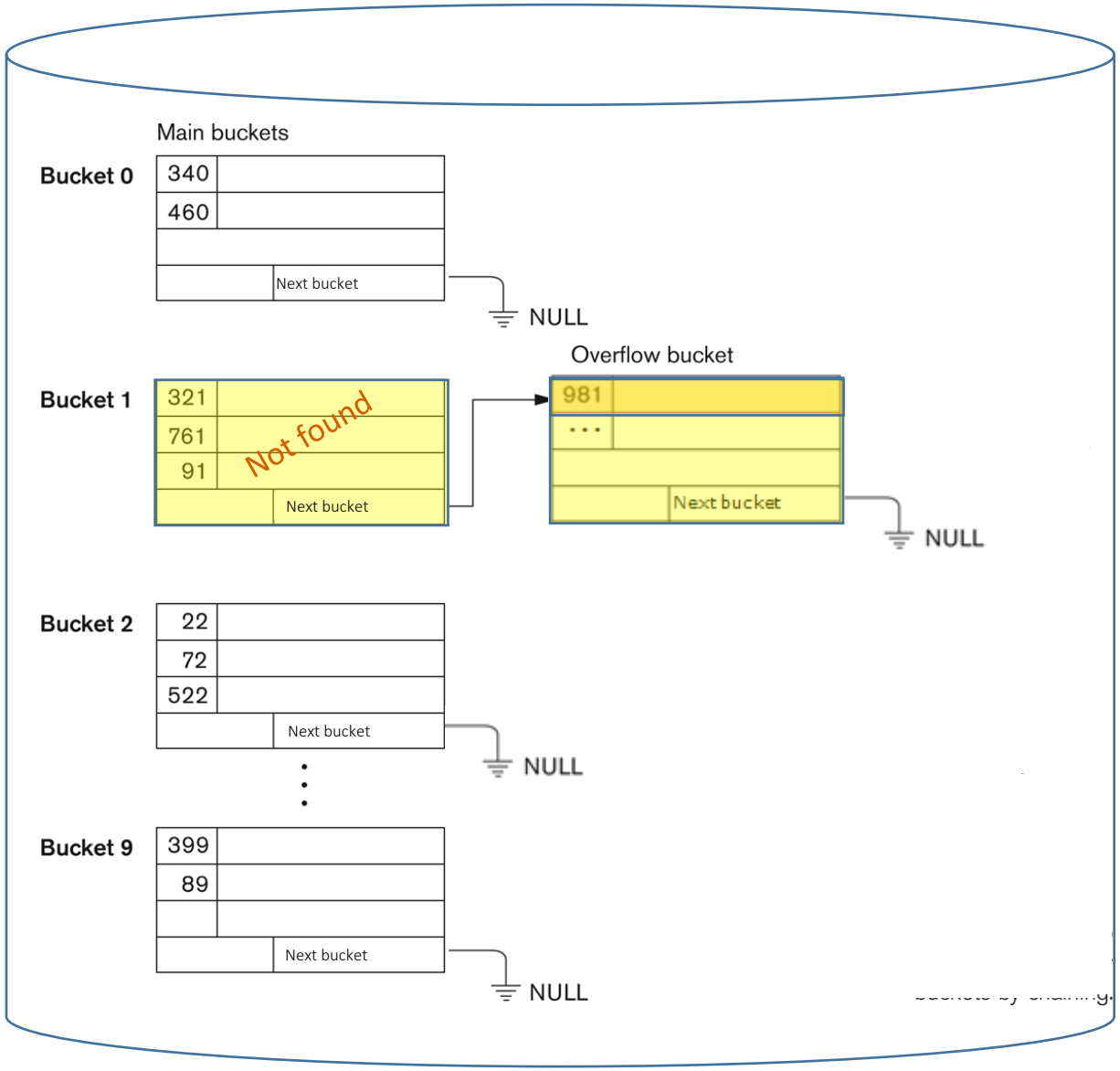
Hash table for primary storage

Hash function
h(K) = K mod 10

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

h(981) = 1

#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 average length of the overflow chain

Overflow chains

- The average length of the overflow chain is a function of
  - the **load factor** → occupied/available slots → T / (B x N_B) → average occupancy of a block (%)
  - the **block factor** B, where:
    - T is the number of tuples
    - N_B is the number of buckets
    - B is the number of tuples within a block (1 bucket <-> 1 block)
- Example (B=3, load factor=70%) → #I/O operations = 1 + 0.3 (with overflow chains)

Typical occupancy range

$T/(B \times N_B)$

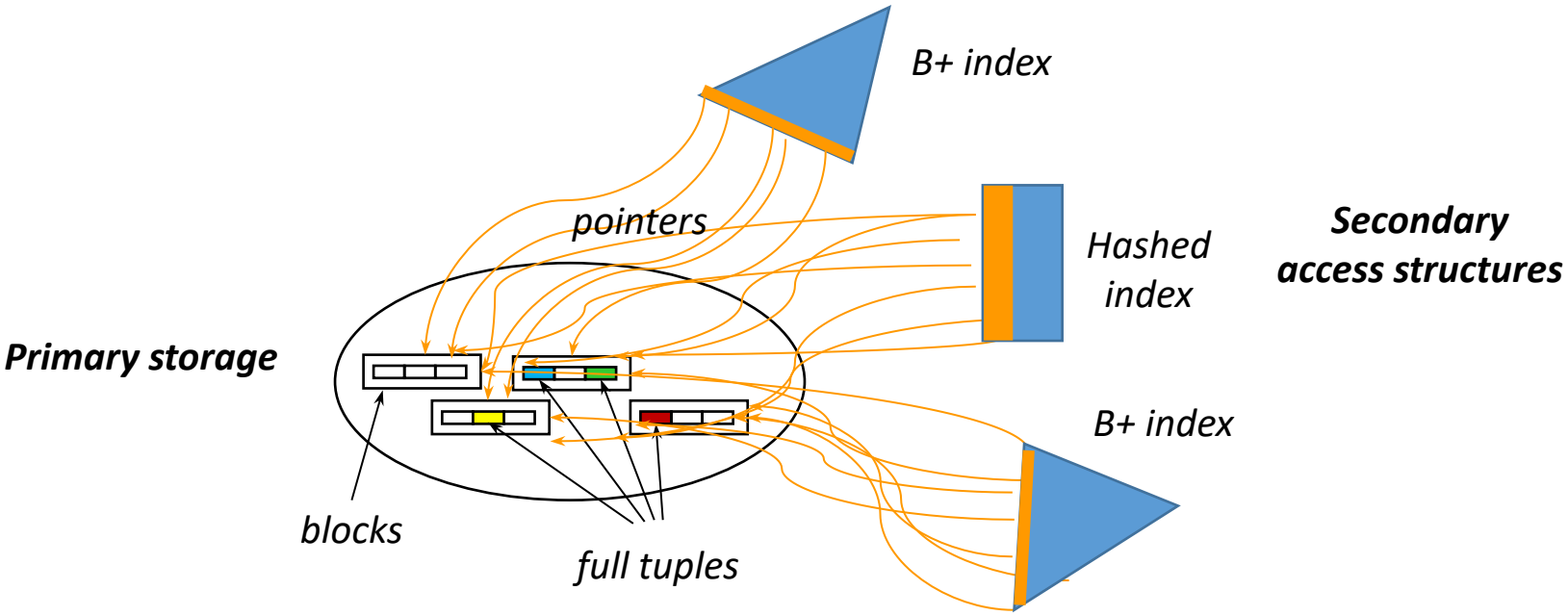
B

	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

average # of accesses to the overflow list (computed from real access statistics)

Indexes

- Data structures that help efficiently retrieve tuples on the basis of a **search key (SK)**
- They contain records of the form [search key, pointer to block]
- Index entries are sorted w.r.t. the key
- The index concept: analytic index of a book → pair (term - page number) alphabetically ordered at the end of a book



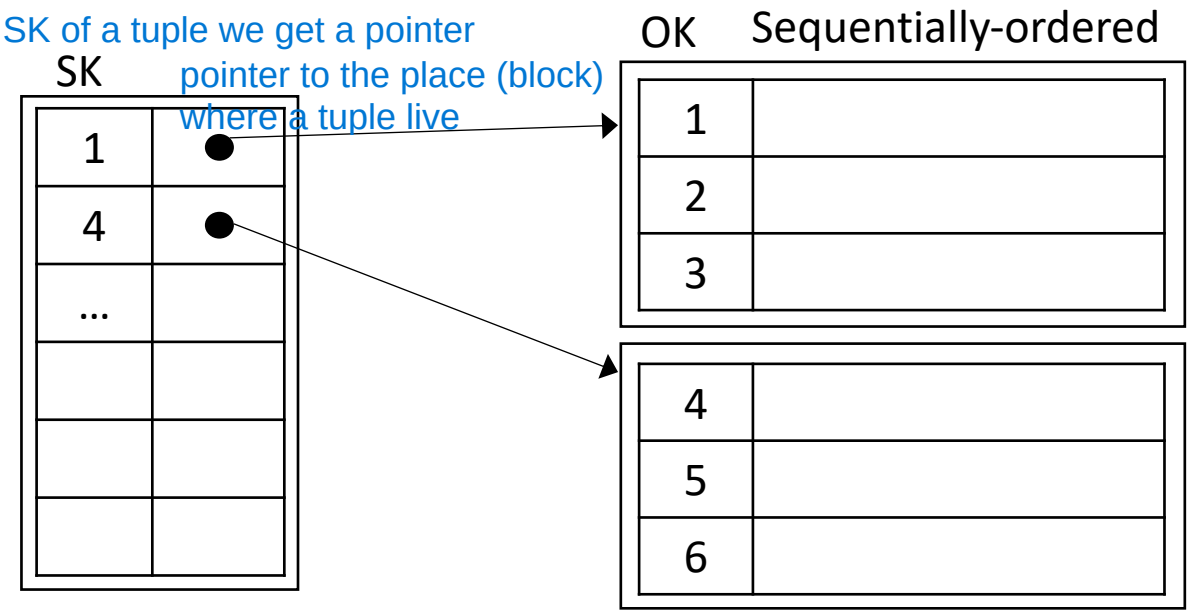
Indexes can be:
- dense: if contains all values of SK
- sparse: not all values of SK are there but only a few of them

Dense vs. sparse index

- Dense index:
  - An index entry **for each search-key value** in the file
  - Performance is good: only a search on the index is needed + access to the tuple
  - Can be used also on entry-sequenced primary structures
- Sparse index:
  - Index entries **only for some search-key values**
  - Requires less space, but is generally slower in locating the tuple
  - Requires sequentially ordered data structures and **SK = OK (ordering key)**
  - Performance is worse: search on the index **+ block scan**
  - Good trade-off: one index entry for each block in the file

given the SK of a tuple we get a pointer

pointer to the place (block) where a tuple live

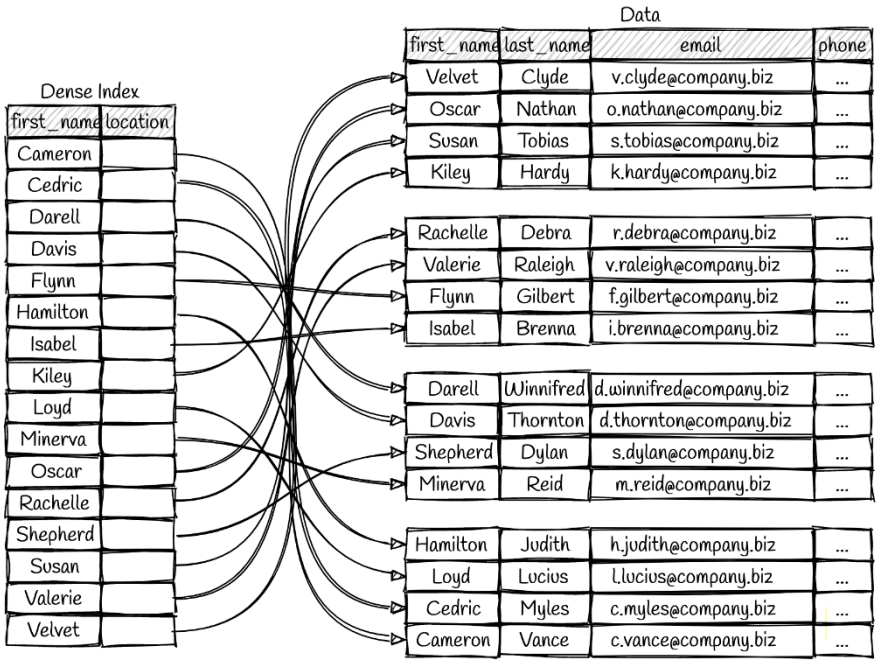


Tuple 3 may exist or not Impossible to know from the index

N.B. We just have pointer to blocks not inside it

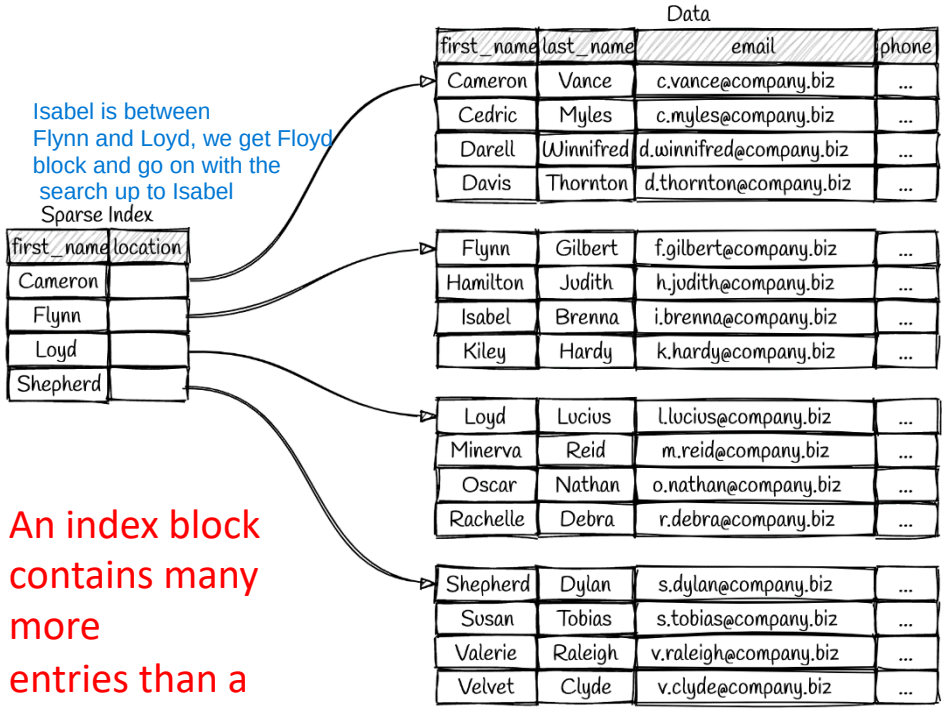
Dense vs Sparse index

Unordered structure --> we need dense index



Entry-sequenced organization

Ordered structure --> we can use sparse index



An index block contains many more entries than a table block

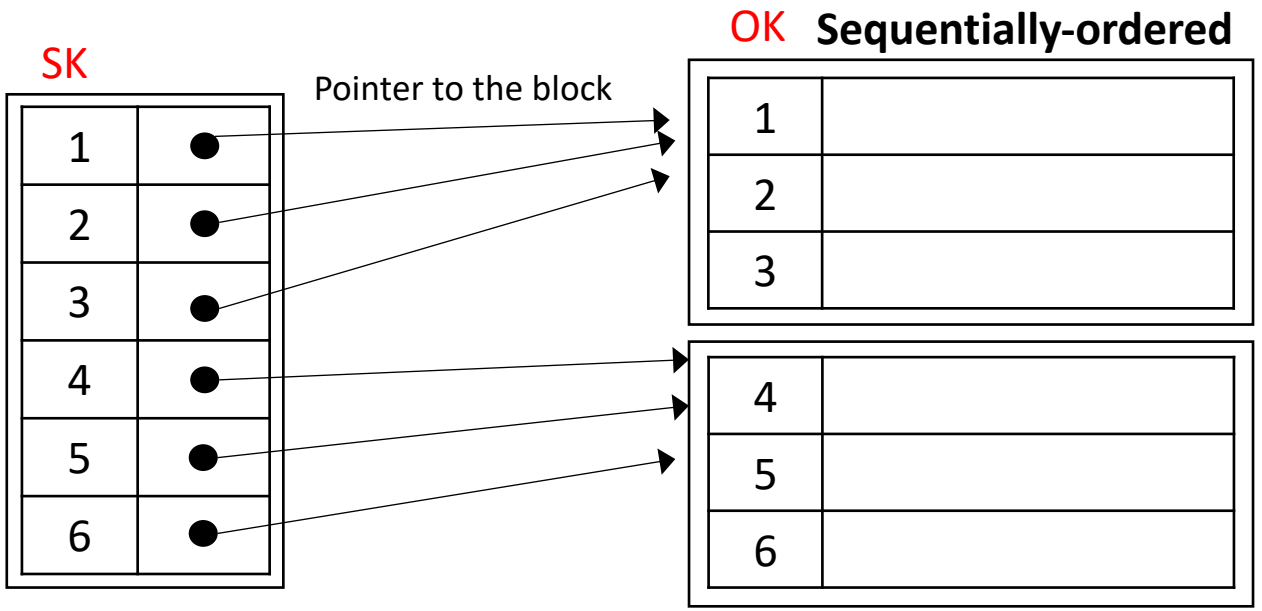
Sequentially-ordered organization

Source: https://yetanotherdevblog.com/dense-vs-sparse-indexes/

Not very important

Primary index

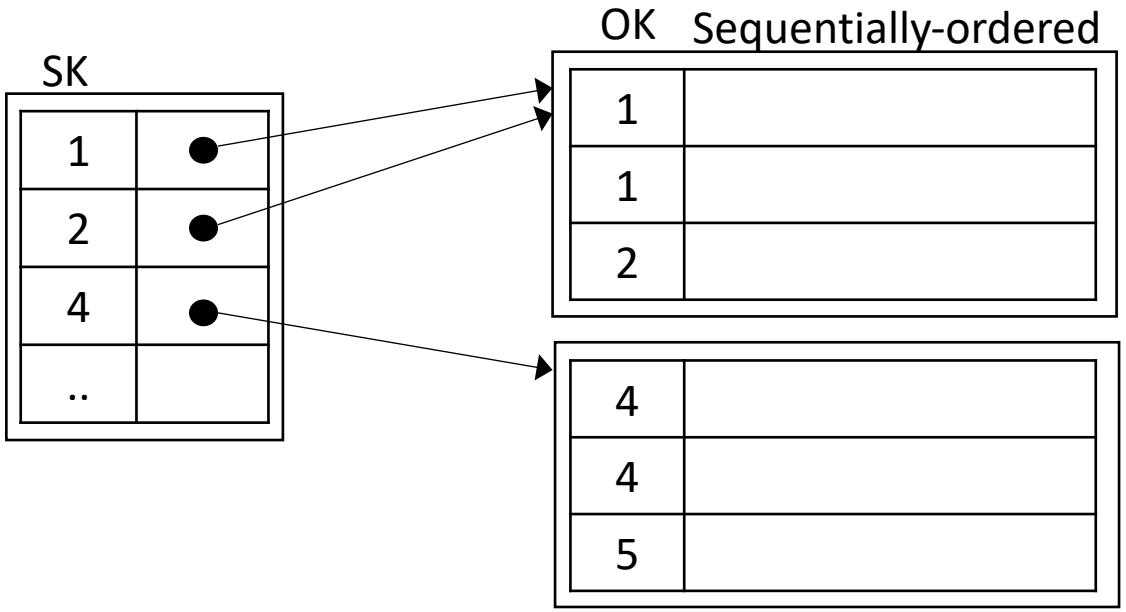
- An index defined on **sequentially ordered structures**
- The search key (SK) is **unique** and coincides with the attribute according to which the structure is ordered (ordering key - OK): **SK = OK**
  - **Only one** primary index per table can be defined
  - Usually on the primary key, but not necessarily
  - The index can be dense or sparse



Clustering index

skipped?

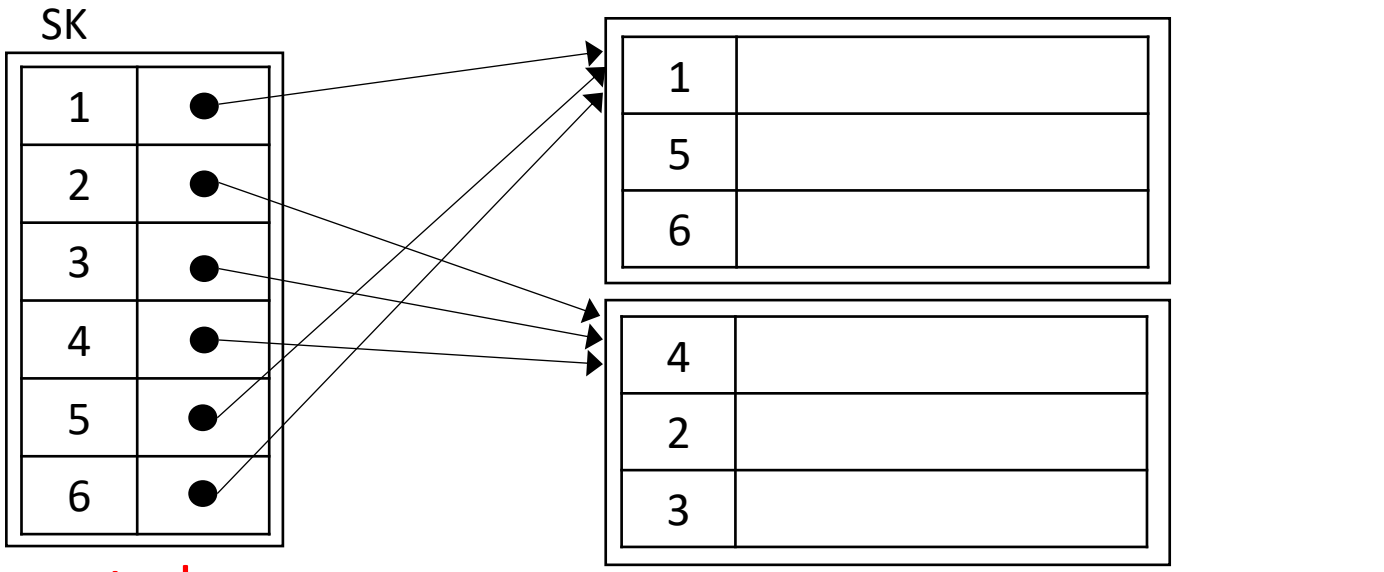
- A generalization of primary index (SK = OK) in which the ordering key **can be not unique**
- The pointer of key X refers to the block containing the first tuple of X key
- Could be sparse or dense, but typically sparse: one entry in the index corresponds to multiple tuples



not important

Secondary index

- Secondary index:
  - The search key specifies an order different from the sequential order of the file **(SK != OK)**
  - Multiple secondary indexes can be defined for the same table, on different search keys
  - It is necessarily **dense**, because tuples with contiguous values of the key can be distributed in different blocks



A search key is not a primary key!

- Don’t get confused:
- **Primary key**: set of attributes that univocally 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
- **Search key**: set of attributes used in an index to speed up locating a tuple
  - Physical implementation of access structures
  - Defines a common access path
    - Each search key value is associated with one or more pointers
    - May be unique (one pointer) or not unique (multiple pointers or pointer to block)

Summary

Type of Index	Type of structure	Search Key	Density	How many
Primary	Sequentially ordered with SK = OK	Unique	Dense or sparse	One per table
Secondary	Entry sequenced or Sequentially ordered with SK != OK	Unique and non unique	Dense (not possible to scan primary data structure wrt SK)	Many per table
Clustering	Sequentially ordered with SK = OK	Non unique	Typically sparse	One per table

Warning: the terminology is not univocal. Check the meaning of such terms as primary, secondary, key, clustering/clustered. They depend on the context and sometimes on the system

Pros & cons of indexes

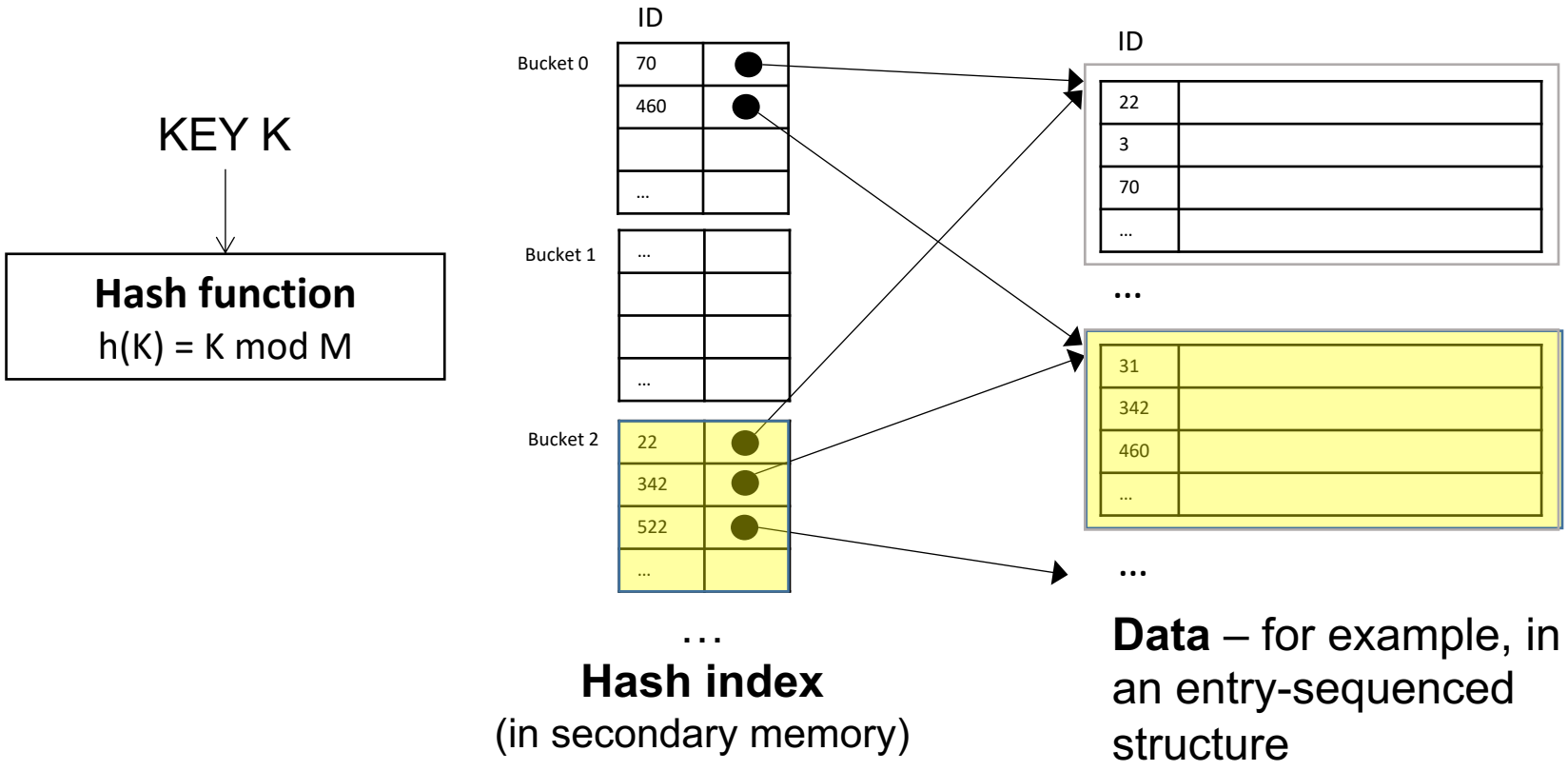
- Smaller than primary data structures, can be loaded in main memory
- Support point/range queries and sorted scans efficiently
  - But less efficient than hash structures for point queries
- BUT: adding indexes to tables means that the DBMS has to **update the index** too after an insert, update, or delete operation
- Indexes do not come for free, they may slow down data changing operations

In data warehouse (OLTP database) data are static since there they're only used for analytics porpuses, in that case the use of indexes is very useful since it speeds up operations (like Group by), however in normal databases (OLAP) many indexes can slow down update operations

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)

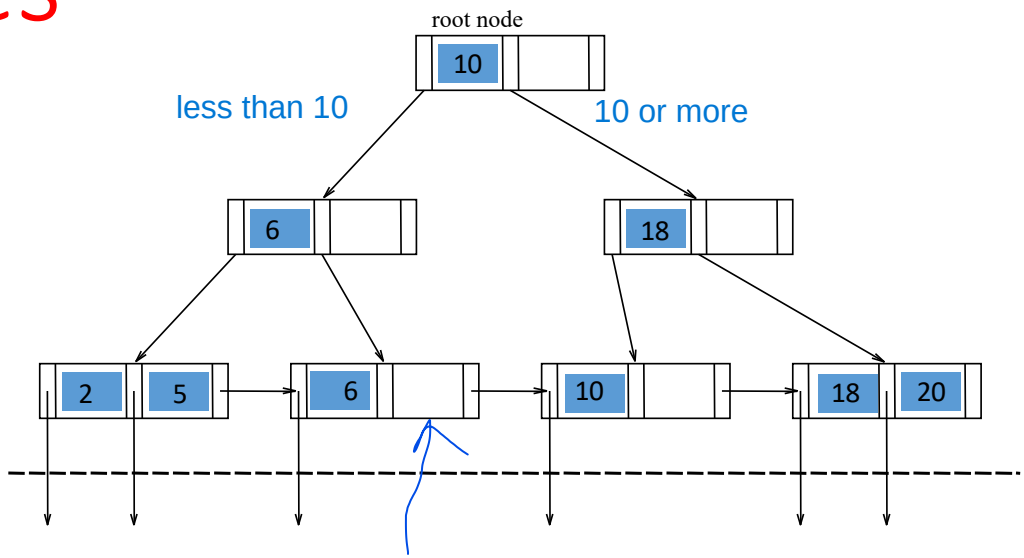
About hashing

- In a huge database, it is inefficient to search all the index values and reach the desired data
- 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 (index) structures
  - SQL indexes are implemented in this way
- Support associative access based on the value of a key search field
- **Balanced trees (B trees)**: the lengths of the paths from the root node to the leaf nodes are all equal
  - Balancing improves performance
- Two main file organizations:
  - **B trees**: key values stored *also in internal nodes*
  - **B+ trees** (most used): all key values stored *in leaf nodes*

B+ tree structures



- Evolution from B trees
- Multi-level index
  - one root node
  - several intermediate nodes
  - several leaf nodes
- Each node is stored in a block

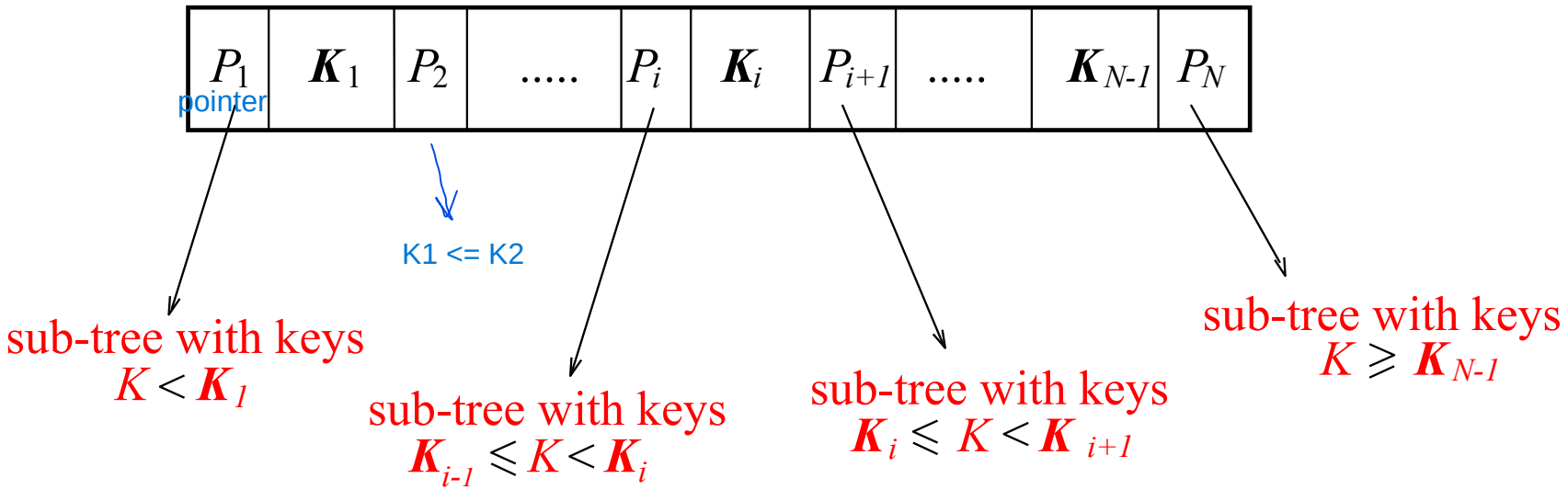
so accessing one of them costs 1

- In general, each node has a large number of descendants (fan out or degree), and therefore the majority of blocks are leaf nodes
- The fan out depends on the size of the block, the size of key values and the size of pointers

Examples of insert/delete in a B+ tree (visualization tool)
https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html

Structure of the B+ tree internal nodes

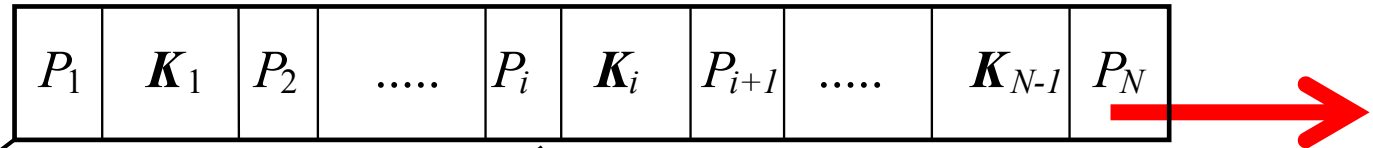
- Internal nodes form a multilevel (sparse) index on the leaf nodes
- Each node can hold up to N-1 search-keys and N pointers
- At least [N/2] pointers (except for the root node) are maintained to ensure that each internal node is at least half full
- Search-keys in a node are sorted $K_i < K_j$ with i<j



Structure of the B+ tree **leaf nodes**

- Each node can hold up to N-1 search-keys
- At least [(N-1)/2] key values should be present in each node (condition ensured by proper maintenance algorithms)
- Search-keys in a node are sorted $K_i < K_j$ with i<j
- The set of leaf nodes forms a **dense** index (each existing key value appears in a node)

While internal nodes indexes are sparse



Pointer(s) to block(s) of primary storage (actual tuple) containing tuples with search-key K_1

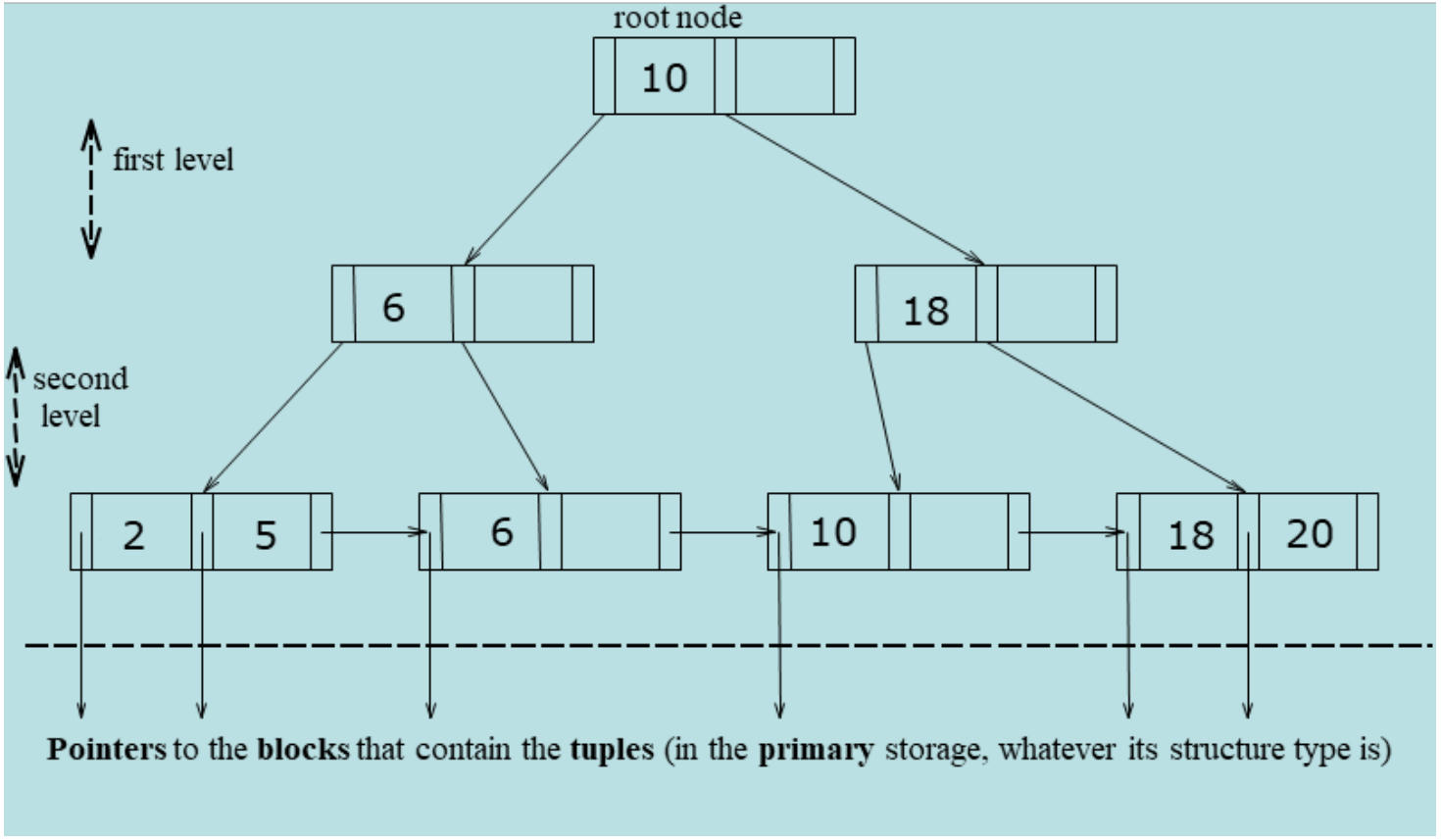
Pointer(s) to block(s) of primary storage containing tuples with search-key K_i

Pointer to the next leaf node

This allows sequential processing

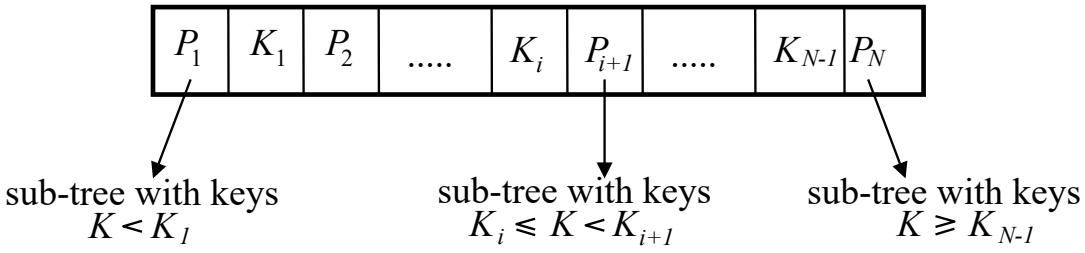
An example of secondary B+ tree

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



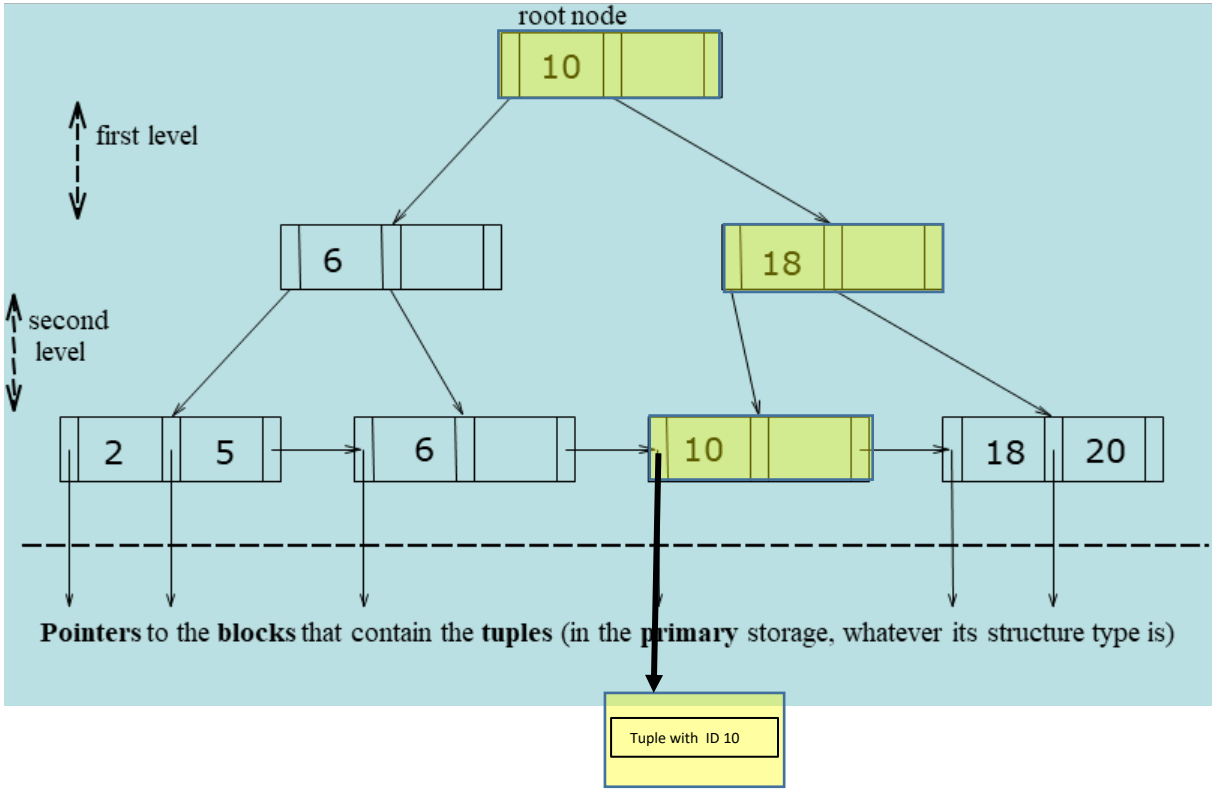
Search technique / lookup

- Looking for a tuple with key value V
- At each intermediate node:
  - if V < K_1 follow P_1
  - if V >= K_{N-1} follow P_N
  - otherwise, follow P_{j+1} such that K_j <= V < K_{j+1}



B+ tree index: query with equality predicate

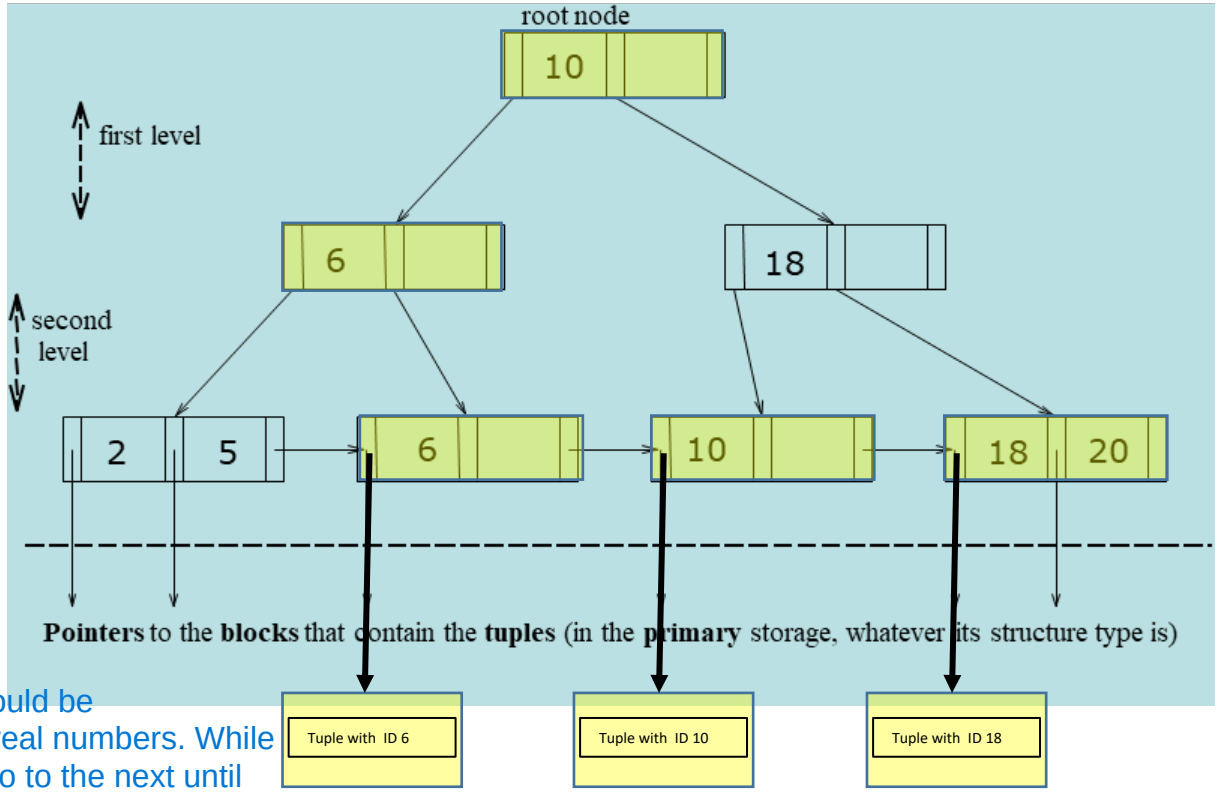
```
SELECT *
FROM  Student
WHERE Student.ID = ’10’
```



- #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 index: query with interval predicate

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

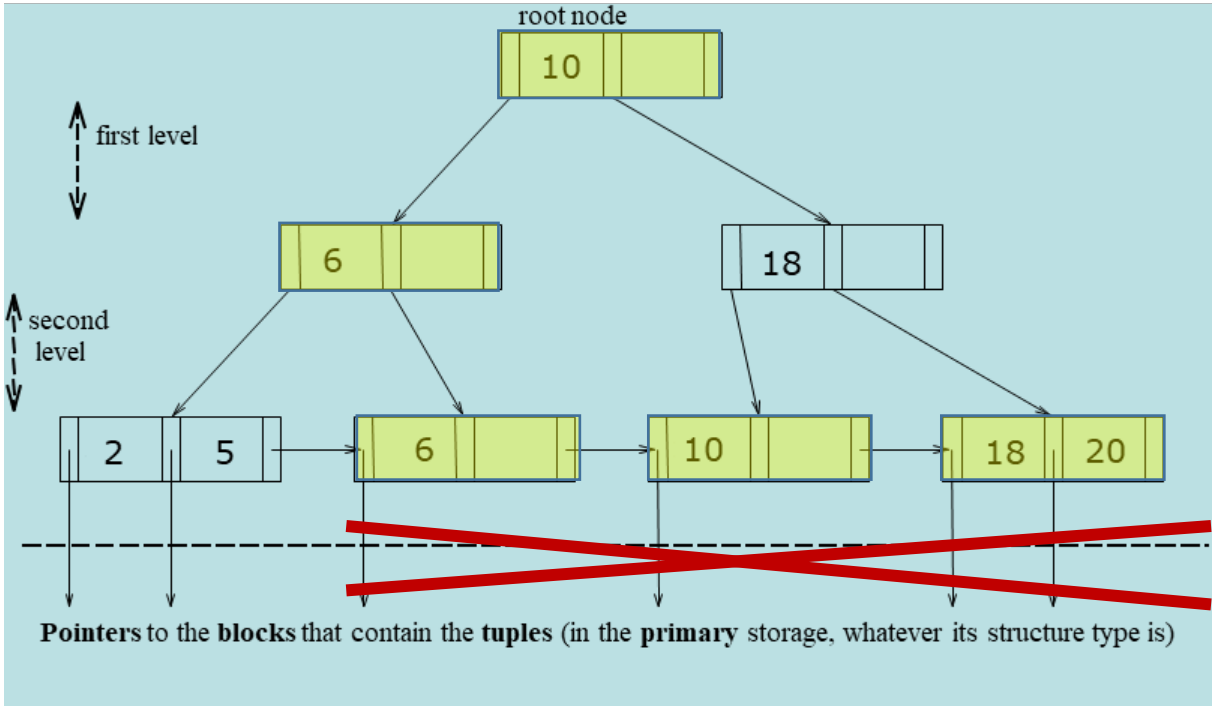


N.B. In case of hash-based structure for finding everything between 6 and 19 we should access every number between 6 and 19, it feasible here but could be infeasible in case of strings or real numbers. While using the B+ tree we just can go to the next until we end the interval

- #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 index: 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 (dense index) 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

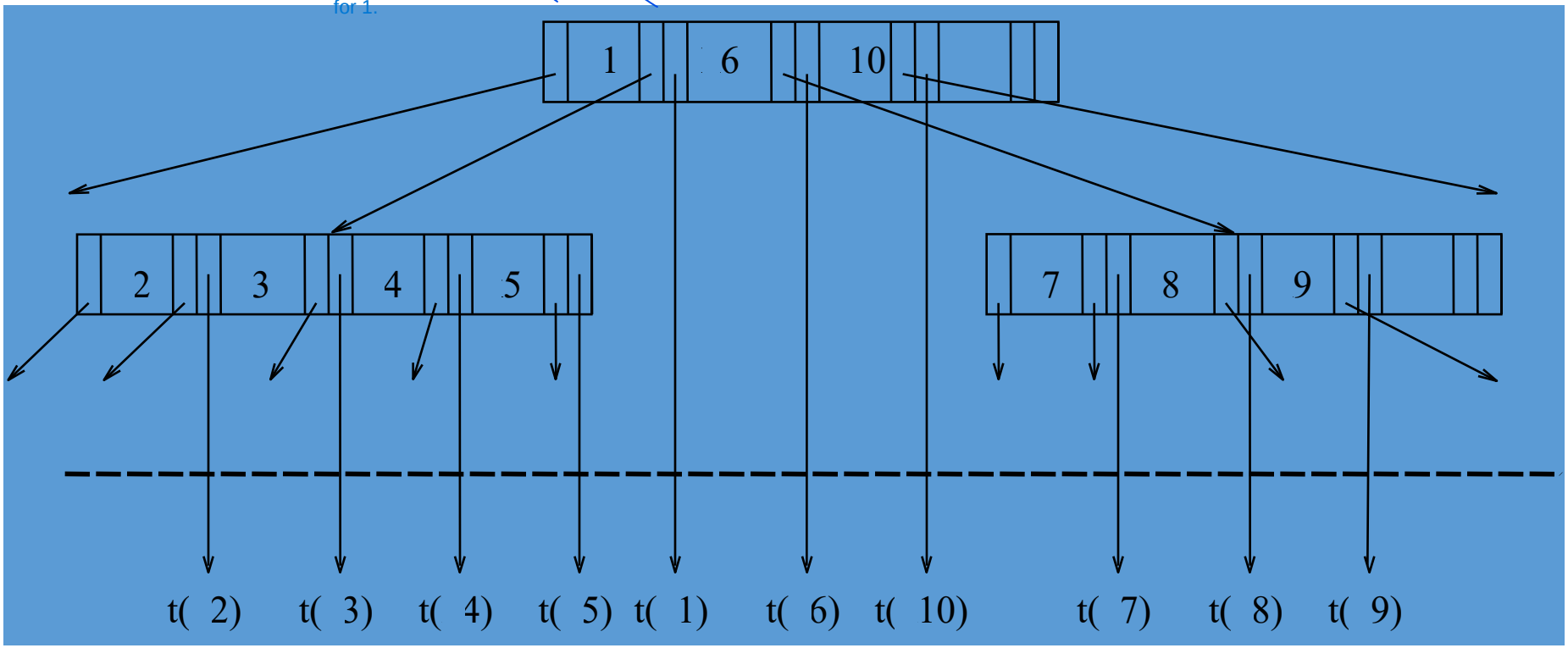
- Eliminates the redundant storage of search-key values
  - Search key values appear only once
  - Intermediate nodes for each key value K_i have:
    - One pointer to the sub-tree with keys between K_i and K_{i+1}
    - One (or more) pointer(s) to the block(s) that contain(s) the tuple(s) that have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

We notice we have more pointers

An example of secondary B tree

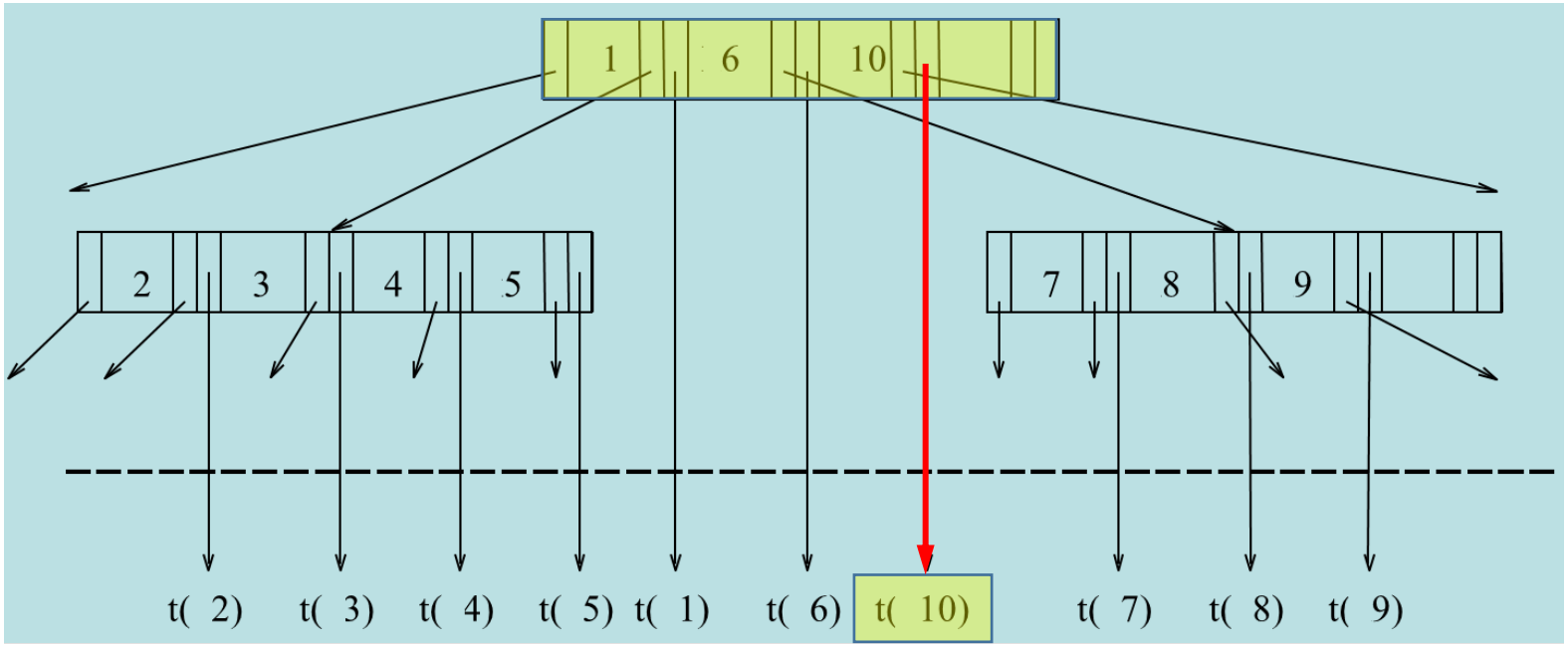
those are
left and right pointers
for 1.

pointer to the actual
data with SK value of 1, which we only had in the leaves in B+ trees



Query on 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

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 sequentially ordered (normally on unique values, typically the primary key)
  - Several secondary indexes, both unique and not unique, on the attributes most used for selections and joins
- 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 a secondary index to any column that is heavily used as a secondary key
4. Add a secondary index to a Foreign Key if it is frequently accessed
5. Add secondary indexes 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

Databases 2

INTRODUCTION TO OPTIMIZATION

COSTS OF DIFFERENT ACCESS MODES

Query optimization

- We learned about tree & hash indexes
  - How does the DBMS know when to use them?
- The same query can be executed by the DBMS in many ways
  - How does the DBMS decide which option 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 (similar to algebraic trees)
  - Algebraic optimization
  - Cost-based optimization
    - The query may be “rewritten” by the DBMS, e.g., turning joins into nested queries
  - Code generation

A simple example of query optimization

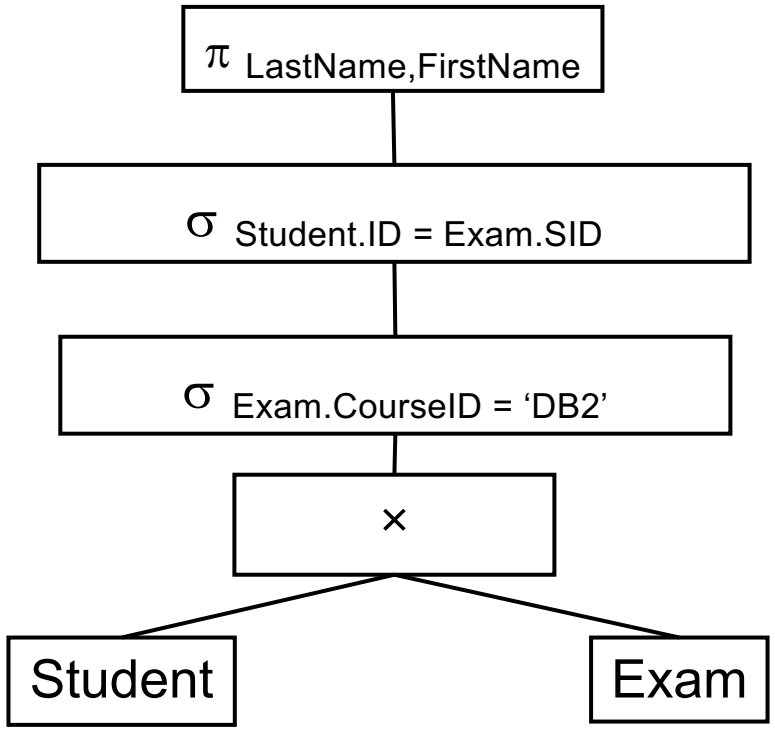
STUDENT (ID, FirstName, LastName, Email, City, Birthdate, Sex)

EXAM (SID, CourseID, Date, Grade)

- Query

```
SELECT LastName, FirstName
FROM Student, Exam
WHERE
  Student.ID = Exam.SID
AND
   Exam.CourseID = 'DB2'
```

Query Tree




```
SELECT LastName, FirstName
FROM Student, Exam
WHERE Student.ID = Exam.SID AND
Exam.CourseID = 'DB2'
```

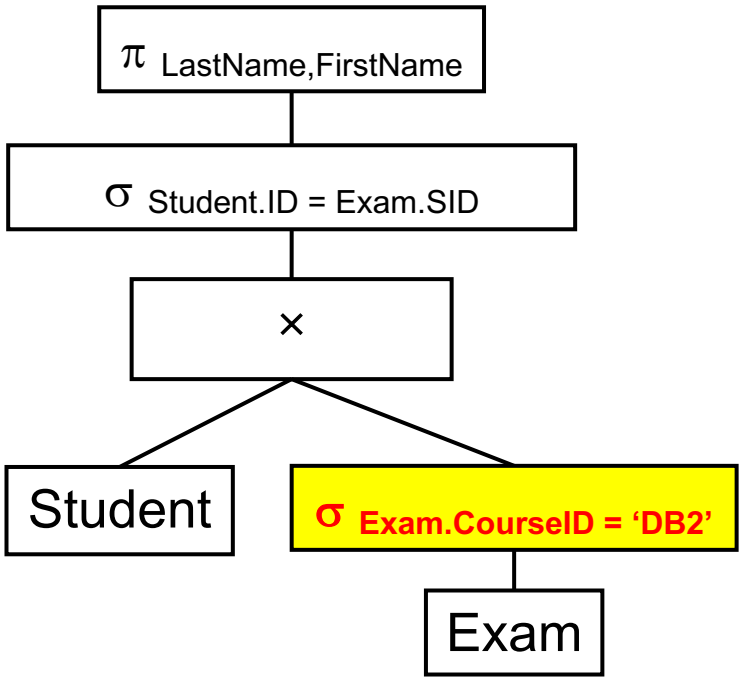
- Many ways to optimise queries:
  - Changing the query tree to an equivalent but more efficient one
  - Exploiting database statistics
  - Choosing efficient implementations of each operator

Optimisation Example

Push the selection down

- Equivalent query:
  - Selecting Exam entries with CourseID = 'DB2'
  - Taking the Cartesian product of the result with Student

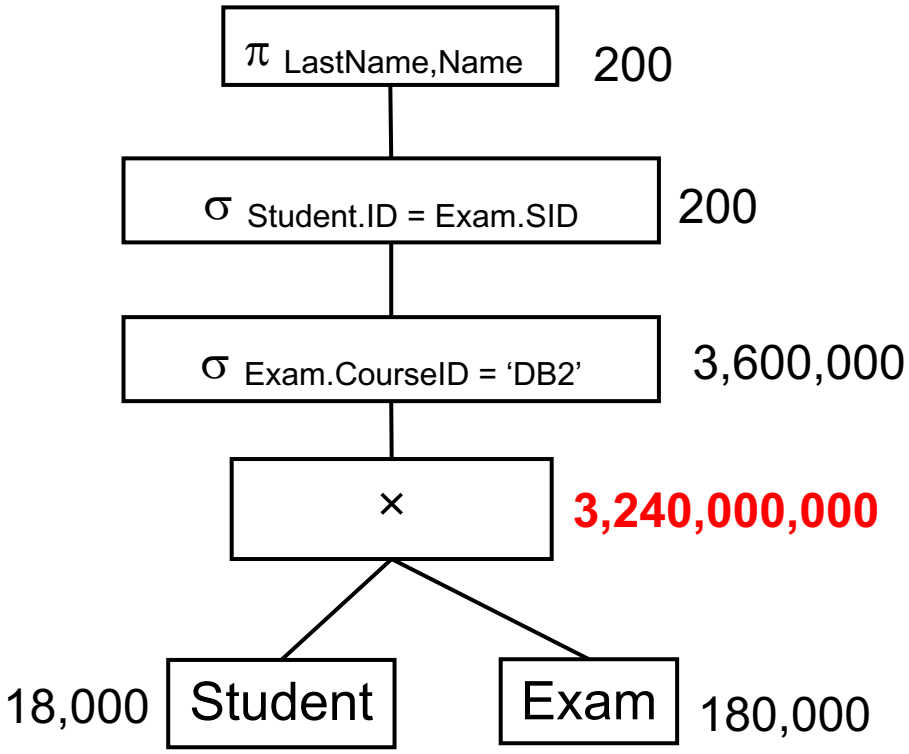
```
SELECT LastName, FirstName
FROM Student, Exam
WHERE Student.ID = Exam.SID AND
Exam.CourseID = 'DB2'
```

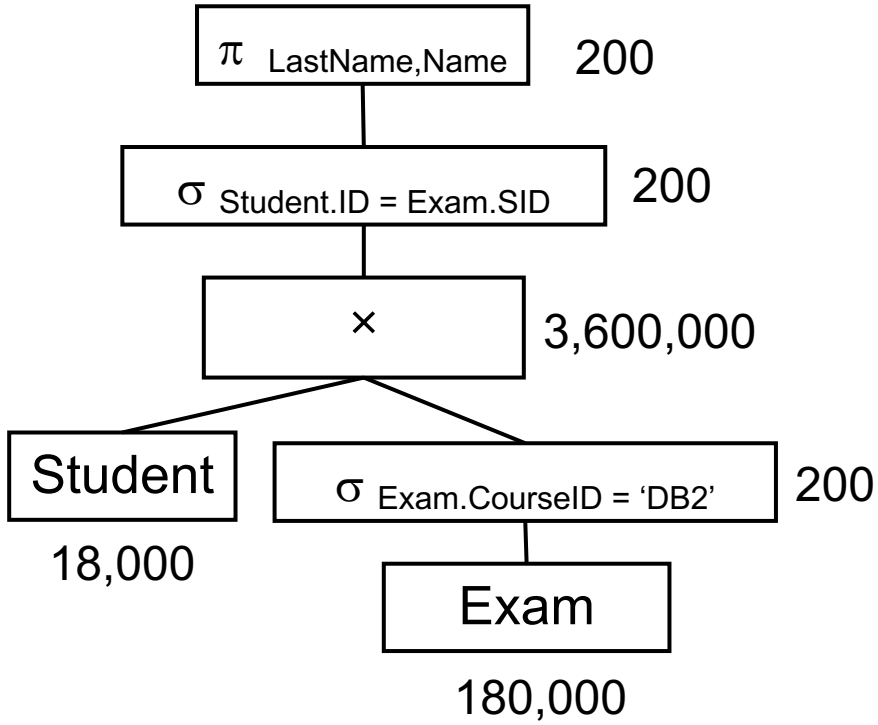


Optimisation Example

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

```
SELECT LastName, FirstName
FROM Student, Exam
WHERE Student.ID = Exam.SID
AND Exam.CourseID = 'DB2'
```





Relation profiles

- Profiles are stored in the data dictionary and contain quantitative information about tables:
  - the cardinality (number of tuples) of each table T
    - Select count (*) from T may be executed directly on the dictionary without accessing the table data
  - the size in bytes of each attribute A in T
  - the number of distinct values of each attribute A in T: val(A)
  - the minimum and maximum values of each attribute A in T
    - Queries with WHERE conditions using < and > may be executed directly on the dictionary
- 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 (attribute A has N values) and
  - the values are homogeneously distributed over the tuples, then
  - the selectivity of a predicate in the form A=v is 1/N
- Example: val(City)=200 → selectivity for City = 1/200 = 0.5%
  - 18K students → 90 students per city (on average)
- If no data on distributions are available, we will always assume homogeneous distributions!

Optimizations

Operations

- Selection
- Projection
- Sort
- Join
- Grouping

Access methods

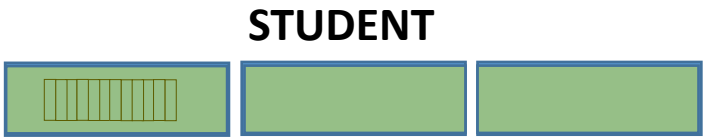
- Sequential
- Hash-based indexes
- Tree-based indexes

Running example

- Block size 8KB (8192 bytes)
- STUDENT (ID, Name, LastName, Email, City, Birthdate, Sex)
  - 150K tuples
  - Tuple size is 95 bytes (4+20+20+20+20+10+1)
  - 8192/95 ≈ 87 tuples per block (block factor)
  - 150000/87 ≈ 1.7K blocks
- EXAM (SID, CourseID, Date, Grade)
  - 1.8M tuples
  - Tuple size is 24 bytes (4+4+12+4)
  - 8192/24 ≈ 340 tuples per block (block factor)
  - 1.8M/340 ≈ 5.3K blocks
- (These numbers are not necessarily realistic…)

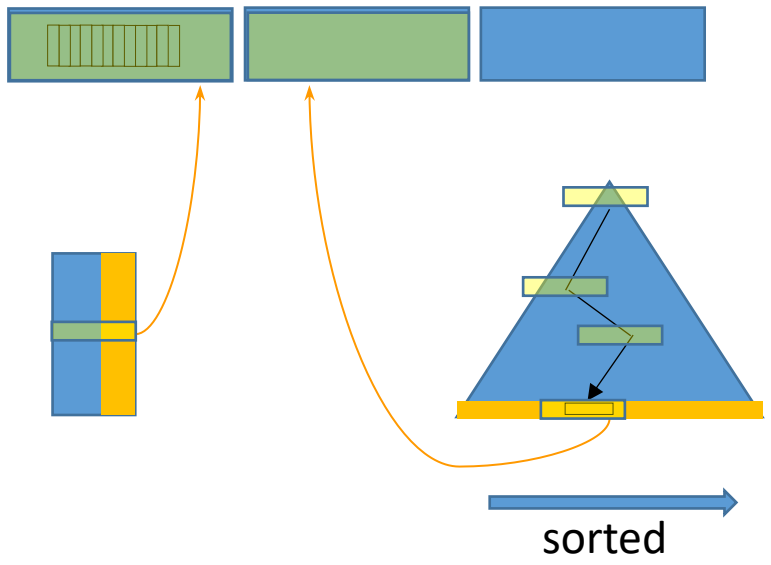
Sequential scan

- 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: A = v)



Hash and tree based access

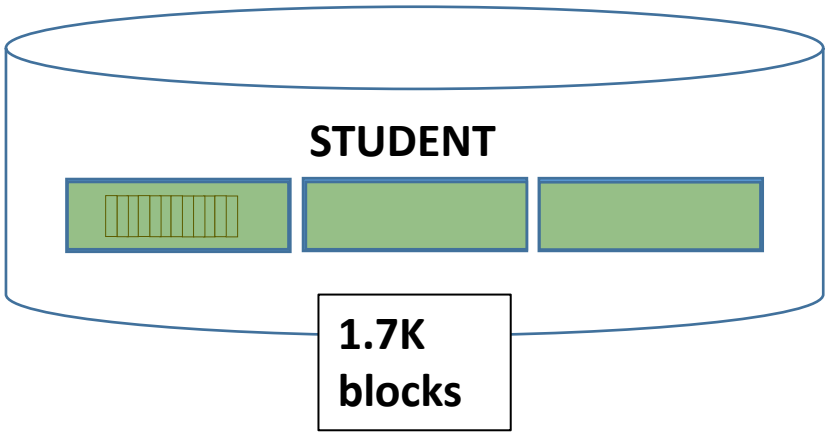
- Hashing supports:
  - equality predicates
- Tree-based indexes support:
  - selection or join criteria
  - ORDER BY
  - GROUP BY
  - other operations involving sorting (such as UNION or DISTINCT)



Sequential scan

- 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 because one can stop when ID=500 (order is by ID)

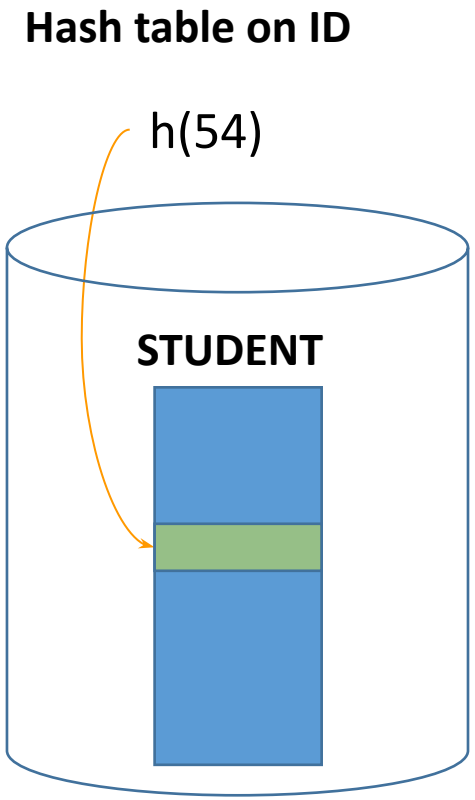
Cost of lookups: equality (A=v)

- Sequential structures with no index
  - Lookups are not supported (cost: a full scan)
    - Sequentially-ordered structures may have reduced cost
- Hash/Tree structures
  - Supported if A is the search key attribute of the structure
  - The cost depends on
    - the storage type (primary/secondary)
    - the search key type (unique/non-unique)

Equality lookup on a primary hash

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

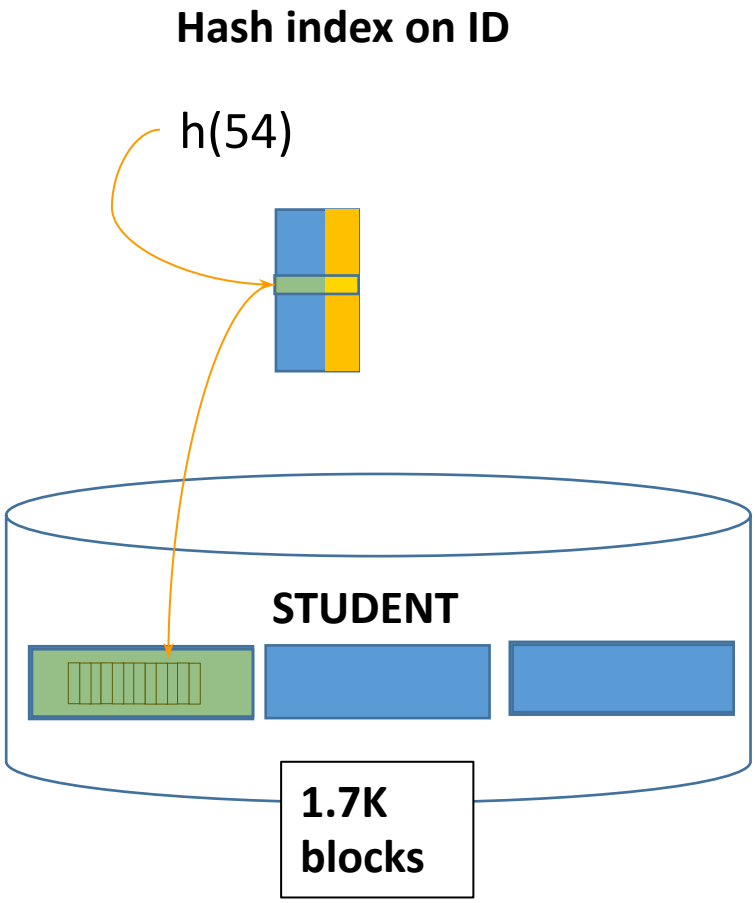
- Query predicate on search key
- Cost:
  - no overflow chains
    - cost = 1
  - With overflow chain with cost 0.3
    - cost = 1.3



Equality lookup on a secondary hash

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

- Query predicate on search key
- Cost:
  - no overflow chains
    - cost = 1 + 1
  - With overflow chain with cost 0.3
    - cost = 1.3 + 1

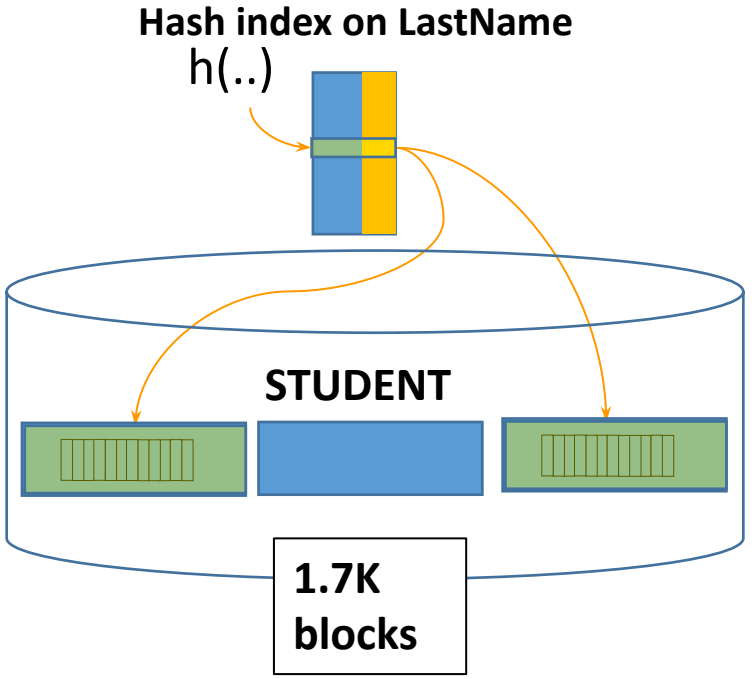


Equality lookup on a secondary hash (non unique search key)

```
SELECT *
FROM STUDENT
WHERE LastName='Rossi'
```

Note: you cannot assume that the 'Rossi' tuples are in the same block of primary storage

- Cost with/without overflow chains:
  - 1 [+ the size of overflow chain (e.g., 0.3)]
  - + cost of accessing blocks in primary storage
- Suppose val(LastName)=75K
  - STUDENT contains 150K tuples → on average each lastname appears twice
  - On average 2 blocks per lastname
- TOTAL COST (with overflow chain) =
  - 1.3 (cost to access the hash index) + 2 (cost to access 2 tuples of STUDENT needed for the SELECT *) = 3.3

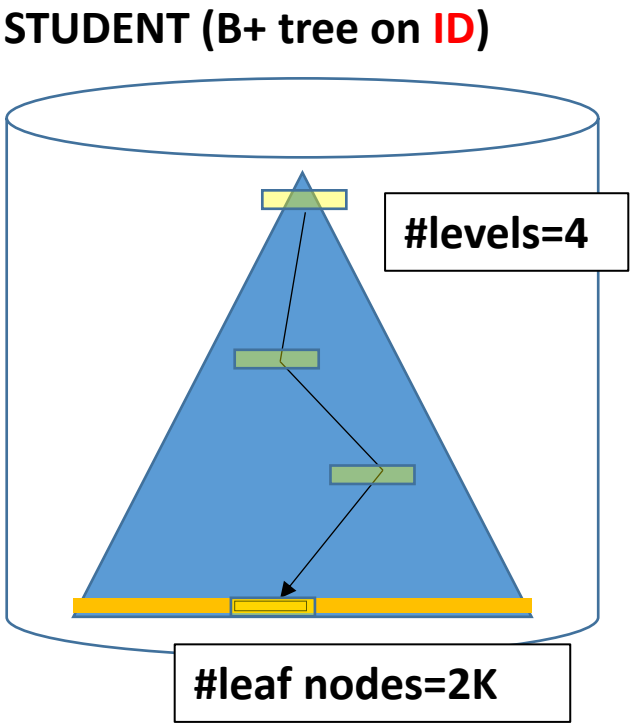


Equality lookup on a primary B+

- Query predicate **on unique** search key
  - Only 1 tuple per key value

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

- Cost:
  - 3 intermediate levels + 1 leaf node = 4
- Remember: primary B+ leaves contain the whole tuples

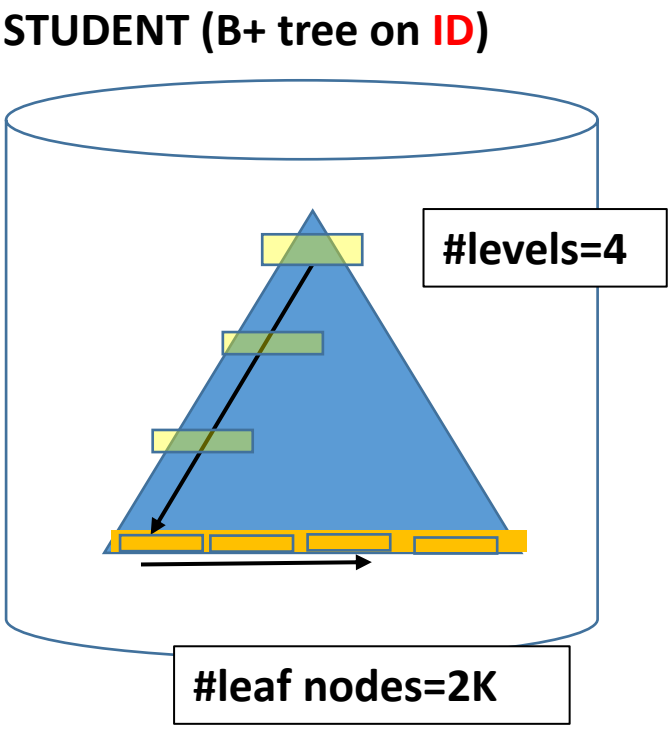


Equality lookup on a primary B+

- Query predicate **NOT** on search key

```
SELECT *
FROM STUDENT
WHERE city='Milan'
```

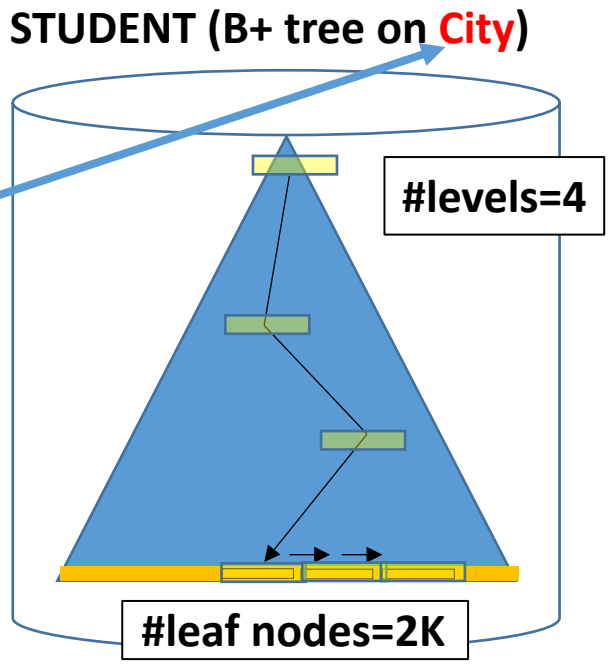
- All the tuples are in the leaf nodes, we need to reach & scan all of them
- Cost:
  - 3 intermediate levels + 2K leaf nodes



Equality lookup on a primary B+

- Query predicate **on non unique** search key
  - Multiple tuples per key value

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



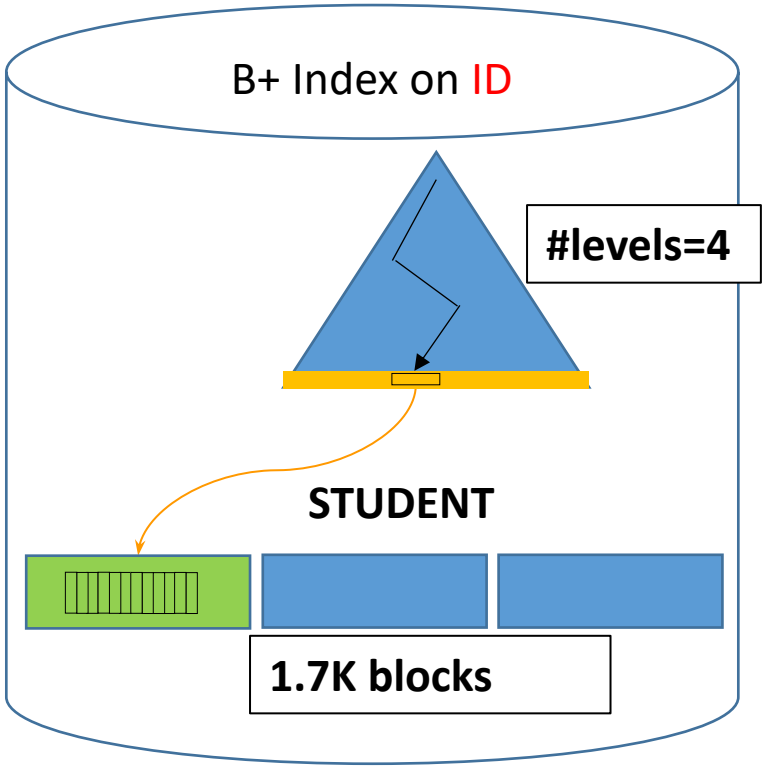
- Cost:
  - #levels to access the leaf = 4
  - How many Milan students blocks?
  - With statistics: val(City) = 150 → 150K / 150 = 1K tuples/City
  - #tuples in 1 leaf node: 150K tuples / 2K nodes = 75 tuples/node
  - #blocks with Milan tuples:
    - 1K Milan tuples / 75 tuples/node = 13.33 → 14 leaf nodes sequentially chained
  - Total cost
    - = 3 intermediate levels + 14 leaf blocks = 17

Equality lookup on a secondary B+

- Query predicate **on unique** search key of STUDENT
  - Only 1 pointer per key value

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

- Cost:
  - 3 intermediate levels + 1 leaf node + 1 data block = 5

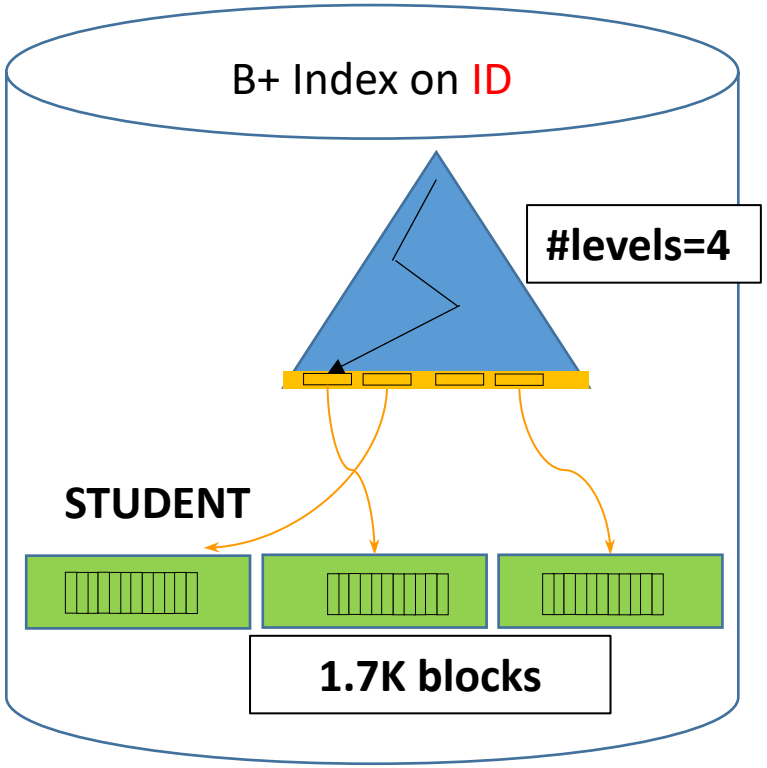


Equality lookup on a secondary B+

- Query predicate **not on** search key of STUDENT

```
SELECT *
FROM STUDENT
WHERE city='Milan'
```

- Cost:
  - Index is not useful for this query (full scan costs 1.7k)

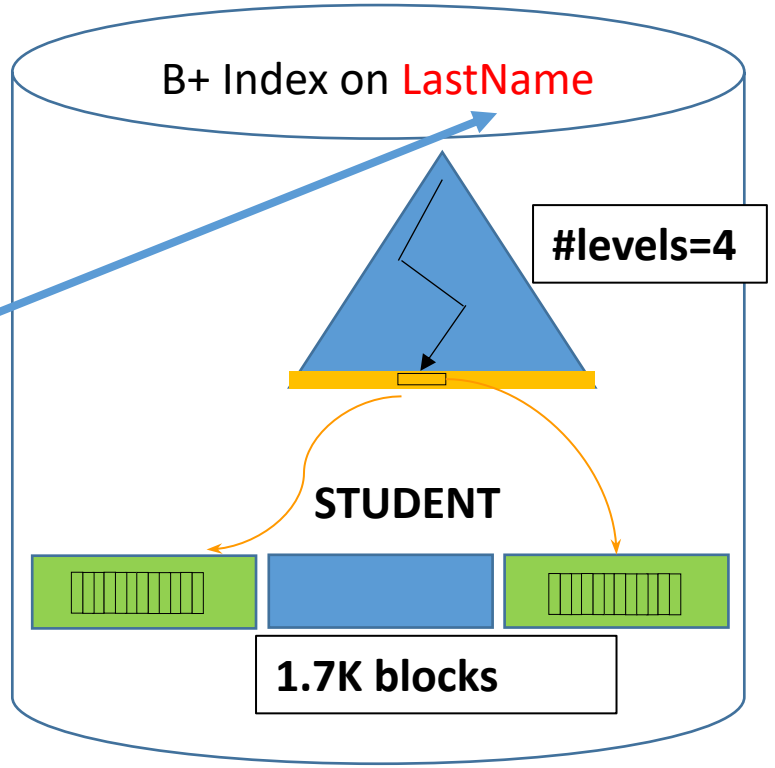


Equality lookup on a secondary B+

- Query predicate **on non unique** search key of STUDENT
  - Multiple pointers per key value

```
SELECT *
FROM STUDENT
WHERE LastName='Rossi'
```

- Suppose Val(LastName) = 75K → 2 tuples / Lastname
- Cost:
  - #of levels of the B+tree: 3 + 1 (2 pointers fit in 1 leaf node)
  - #of blocks to be accessed in STUDENT following the B+ tree pointers in the leaf: 2
  - Total cost = 3 +1 + 2 = 6



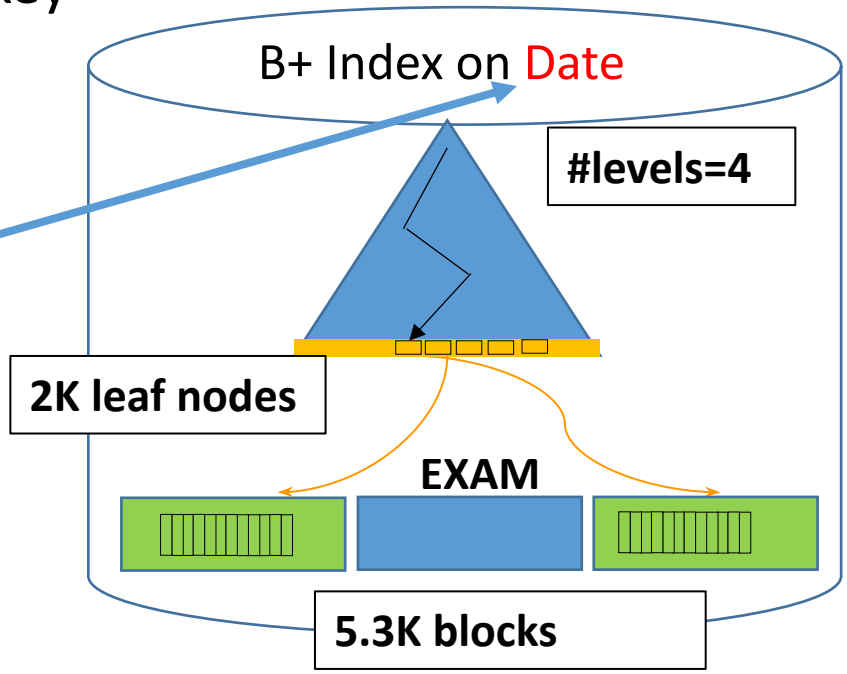
We assume that we reload a block for tuple Ti even if we have loaded it already for a tuple Tj (no caching)

Equality lookup on a secondary B+

- Query predicate **on non unique** search key of EXAM
  - Multiple pointers per key value

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

- Suppose val(Date) = 500
  → 1.8M / 500 = 3.6K tuples/date
- Cost factors =
  - #of intermediate levels **3**
  - pointers per leaf **1.8M / 2K = 900**
  - leaf nodes per date: ***3.6K / 900 = 4*** *(but in most of the cases, the given date may not be the left-most key in the block → on average, the 900 pointers will be in **5** blocks)*
  - blocks to access in EXAM (1 per pointer) = **3.6K**
- **Total cost = 3 + 5 + 3.6K ≈ 3.6K**



We assume that we reload a block for tuple Ti even if we have loaded it already for a tuple Tj (no caching)

Interval lookups A<v , v_1<= A <= v_2

- Sequential structures (primary)
  - Lookups are not supported (cost: a full scan)
    - Sequentially-ordered structures may have reduced cost
- Hash structures (primary/secondary)
  - Lookups based on intervals are not supported
- Tree structures (primary/secondary)
  - Supported if A is the search key
  - The cost depends on
    - the storage type (primary/secondary) – see next slides
    - the index key type (unique/non-unique)

Interval lookup on **primary** B+

- We consider a lookup for $v_1 <= A <= v_2$ as the general case
  - If A<v or v<A 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 is reached that stores tuples with $A = v_1$
  - 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 (estimated with statistics)

Interval lookup on **secondary** B+

- We still consider a lookup for for $v_1 <= A <= v_2$ 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 $A = v_1$
  - If all the pointers (up to v_2) are in that leaf block, stop
  - Else, continue in the chain of leaf blocks 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)

Conjunction / disjunction

- **Predicates in conjunction**

```
SELECT * FROM STUDENT
WHERE City='Milan' AND Birthdate=12/10/2000
```

  - If supported by indexes, the DBMS chooses the **most selective** supported predicate for the data access, and evaluates the other predicates in main memory
- **Predicates in disjunction**:

```
SELECT * FROM STUDENT
WHERE City='Milan' OR Birthdate=12/10/2000
```

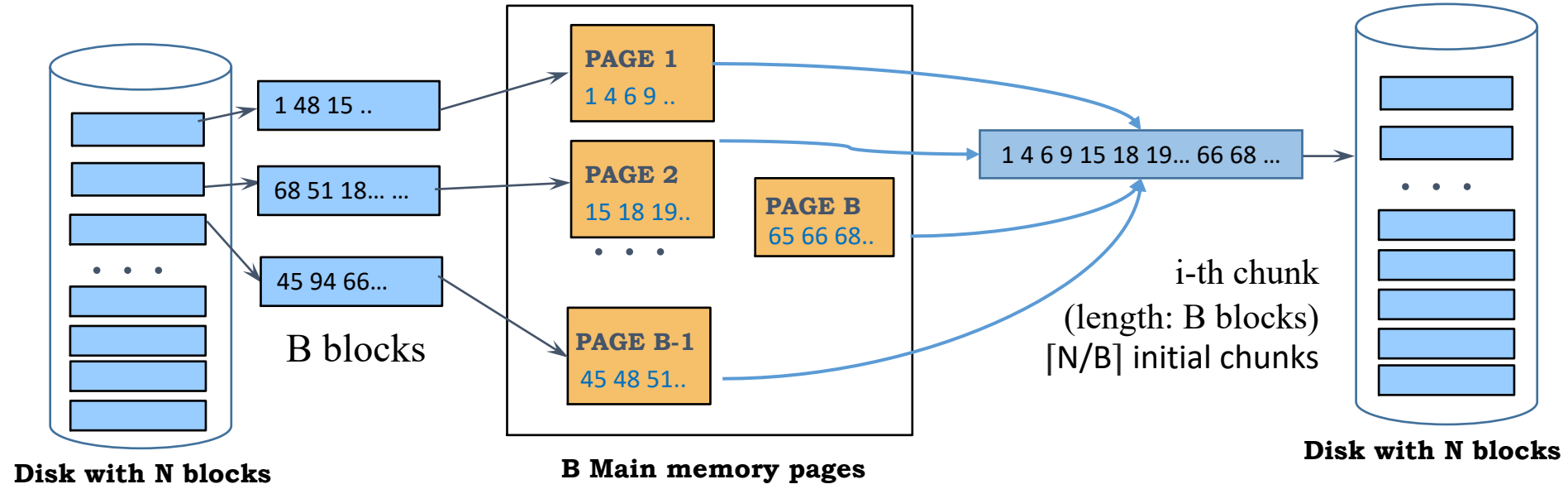
  - if any of the predicates is not supported by indexes, then a scan is needed
  - if **all** are supported, indexes can be used to evaluate all predicates and then duplicate elimination is normally required

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
  - External merge-sort:
    - Data do not fit into the main memory
    - Procedure:
      1. Load from disk as much data as can be stored in main memory and sort them, store such sorted chunks back to disk
      2. Then, merge sorted chunks parts using at least three pages: two for progressively loading data from two sorted chunks and one as an output buffer to store the merge-sorted data. Save the merge-sorted chunks back to disk
      3. Repeat step 2 until all data are sorted

External Merge Sort

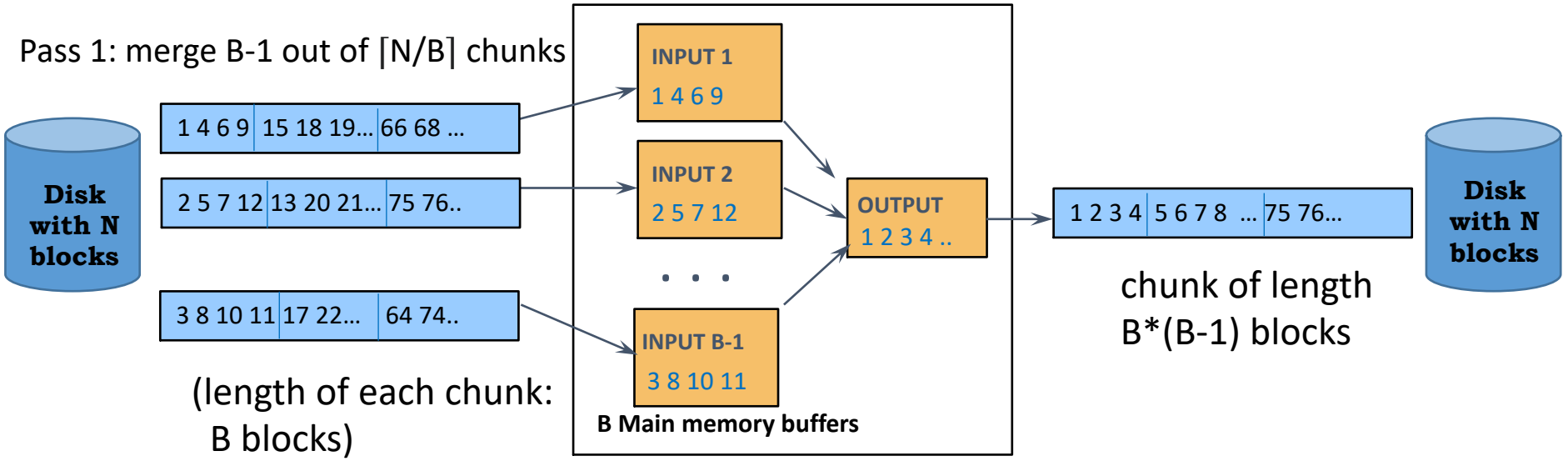
- To sort a file stored in N blocks using B buffer pages:
  - Pass 0:
    - read B blocks at a time into main memory and sort them
    - write sorted data to a chunk file (also called “run”)
    - → total chunks = ⌈N/B⌉



https://www.youtube.com/watch?v=Bp7fGofslng

External Merge Sort

- Pass 1, 2, 3, 4, …: Use B-1 blocks for input chunks, 1 block for OUTPUT
- Repeat
  - Read first block from each chunk into a buffer page
  - 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 chunk into the buffer
- Until all input buffer pages are empty

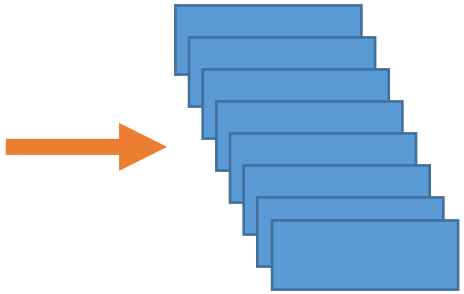


External Merge Sort

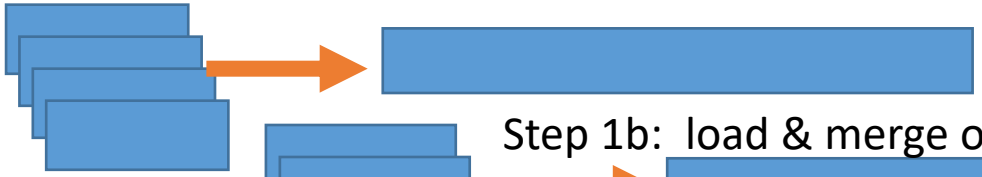
- In each pass, B-1 chunks are merged into a larger one
- Repeated passes are performed until all chunks have been merged into one
- A pass reduces the number of chunks by B-1 and creates correspondingly longer chunks
- E.g., B=5, N=40 → initial chunks = 40/5 =8 (having length 5 pages)
    - 40 read operations + 40 write operations
- In the next pass: with 4 input pages + 1 output page
    - First load 4 chunks and create a new chunk having length 5*4=20 pages
    - Load the next 4 chunks and create a new chunk having length 20 pages
        - 8/4 =2 sorted chunks of 20 pages each → 40 + 40 I/O operations
- In the next pass load the 2 final chunks into the final sorted chunk of length 20*2
        - $\lceil 2/4 \rceil$ =1 sorted chunks of 40 pages → 40 + 40 I/O operations
        - Total passes = 3

B=5, N=40

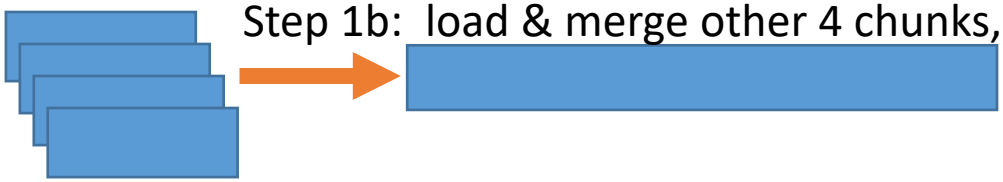
Step 0: create 8 (40/5) sorted chunks of 5 blocks



Step 1a: load & merge first 4 chunks, create one chunk of 5 x 4 = 20 blocks



Step 1b: load & merge other 4 chunks, create 2nd chunk of 20 blocks



Step 2: load & merge 2 chunks of 20 blocks, create one chunk of 40 blocks

External sorting

- Number of passes cost =
  - 1 (for initial step) + $\lceil \log_{B-1} \lceil N/B \rceil \rceil$ (for iterative merge of chunks)
  - In the example: $1 + \lceil \log_4 8 \rceil = 1 + 2$
- Cost = 2 (1 for reading + 1 for writing) * N * (# 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

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 various aspects…

Running example

- Block size 8KB (8192 bytes)
- STUDENT (ID, Name, LastName, Email, City, Birthdate, Sex)
  - 150K tuples
  - Tuple size is 95 bytes (4+20+20+20+20+10+1)
  - $t_{STUDENT} = 8192/95 \approx 87$ tuples per block (block factor)
  - $b_{STUDENT} = 150000/87 \approx 1.7K$ blocks
- EXAM (SID, CourseID, Date, Grade)
  - 1.8M tuples
  - Tuple size is 24 bytes (4+4+12+4)
  - $t_{EXAM} = 8192/24 \approx 340$ tuples per block (block factor)
  - $b_{EXAM} = 1.8M/340 \approx 5.3K$ blocks

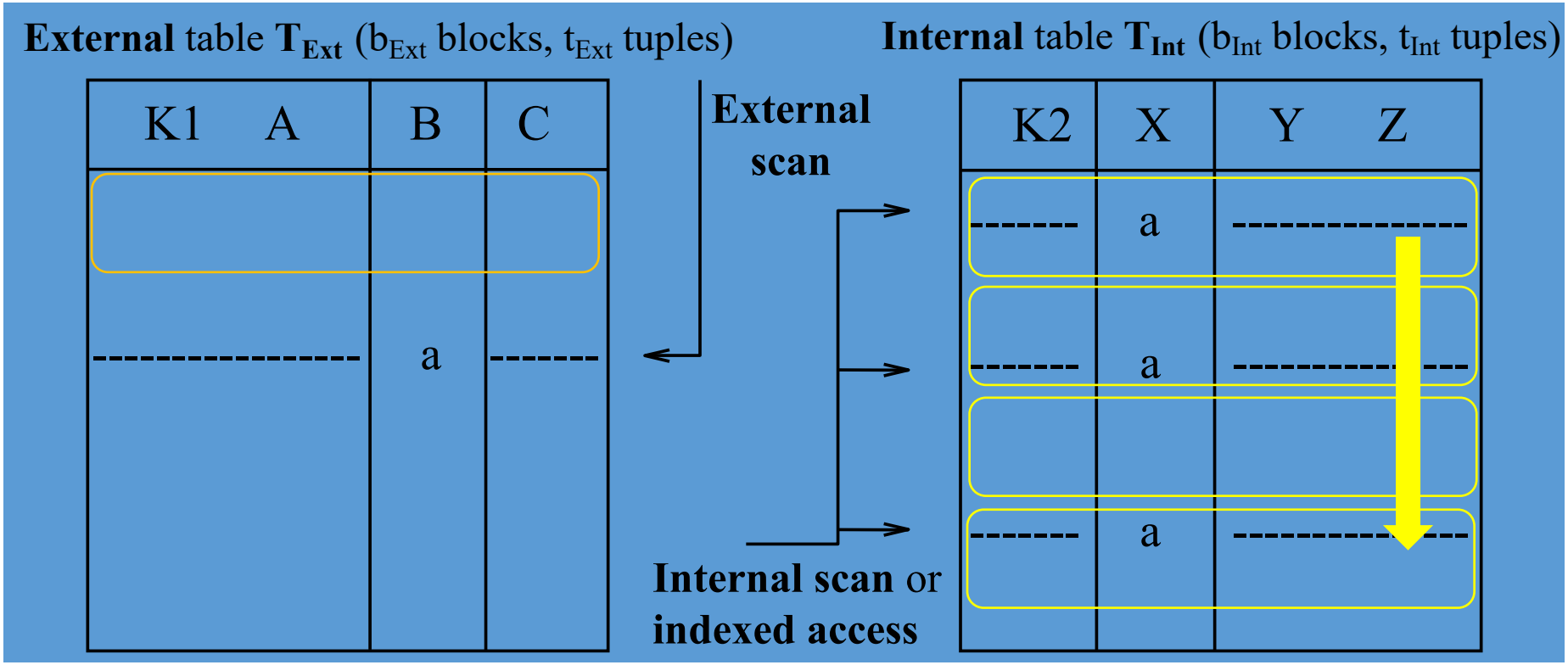
Equality Joins With One Join Column

```
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

- Scan the external table and for each block scan the internal table



Cost of simple nested-loop joins

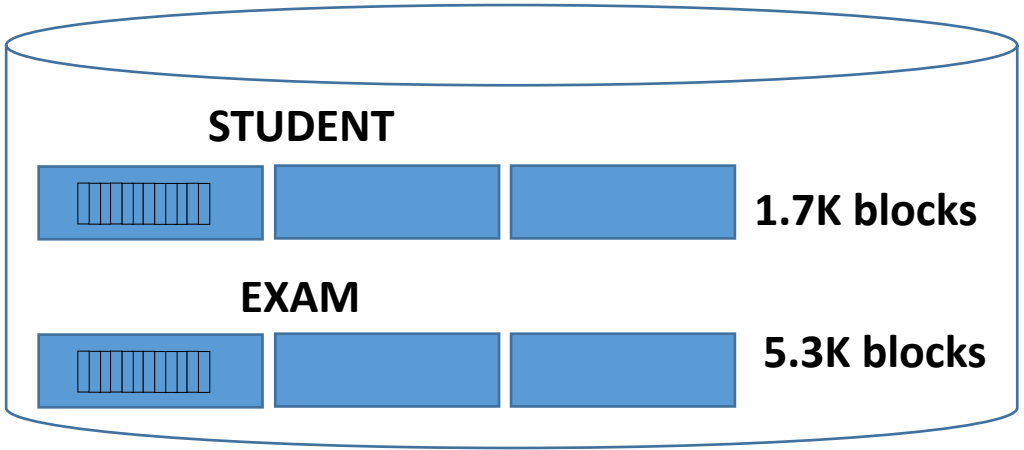
- A nested loop join compares the tuples of a block of table T_{Ext} with all the tuples of all the blocks of T_{Int} before moving to the next block of T_{Ext}
  - We always assume that the buffer does not have enough available free pages to host more than a few blocks
  - The cost is **quadratic** in the size of the tables: scan the external table and then scan the internal table once for each block of the external one
    - Cost = $b_{Ext} + b_{Ext} \times b_{Int} = b_{Ext} \times (1 + b_{Int}) \approx b_{Ext} \times b_{Int}$
  - If one of the tables is small enough to fit in the buffer, then it is chosen as internal table and scanned **only once**
    - Cost = $b_{Ext} + b_{Int}$

A simple Nested-Loop join

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

```
SELECT STUDENT.*
FROM STUDENT JOIN EXAM ON ID=SID
```

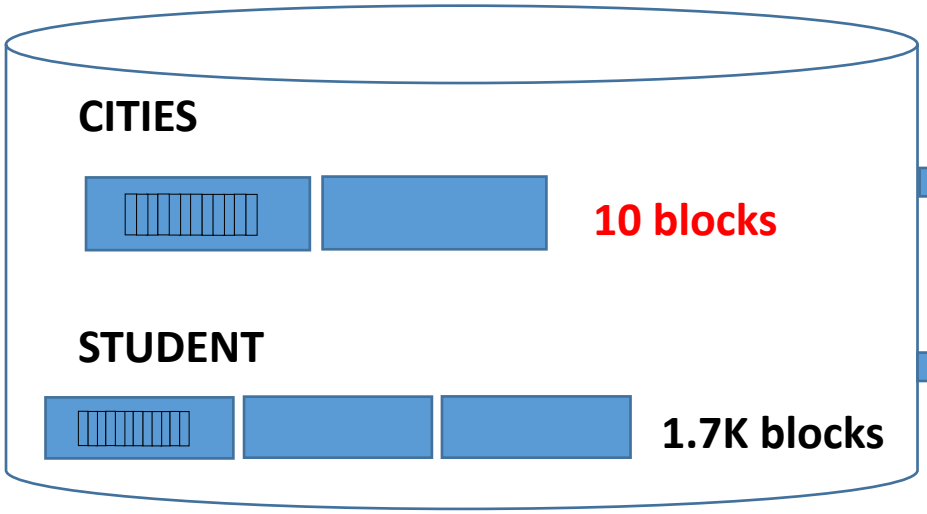
- Student external table, Exam internal table
  - Cost = 1.7K + 1.7K*5.3K ≈ 9M I/O accesses
- Exam external table, Student internal table
  - Cost = 5.3K + 5.3K*1.7K ≈ 9M I/O accesses



Nested-Loop join with cache

- STUDENT (ID, Name, LastName, Email, City, Birthdate, Sex)
- CITIES (City, Region, Description)
  - b_{CITIES} = 10 blocks, with block size 8 KB → 80 KB → cacheable

```
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**

Filtering by condition: option 1

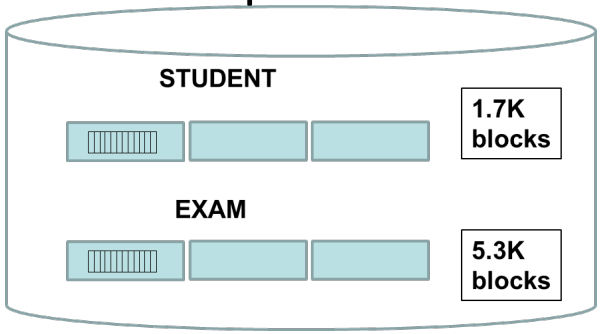
- STUDENT (ID, Name, LastName, Email, City, Birthdate, Sex)
- EXAM (SID, CourseID, Date, Grade)

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

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

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

- Read all the STUDENT blocks: **1.7K**
- **Filter** students from Milan → 150K tuples / 150 cities = **1K**
- For 1K students do the join with the tuples of all the 5.3K blocks in EXAM → this requires 1K scans



Total cost:
1.7K + 1K*5.3K
≈ **5.3M** I/O accesses

Filtering by condition: option 2

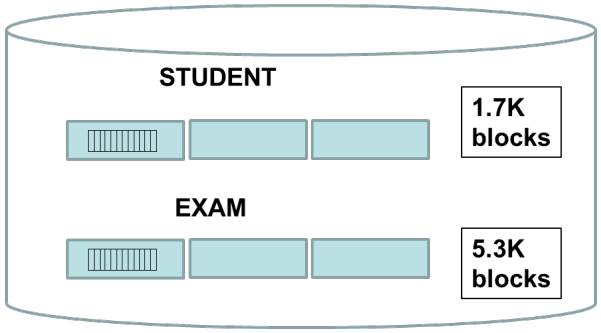
- STUDENT (ID, Name, LastName, Email, City, Birthdate, Sex)
- EXAM (SID, CourseID, Date, Grade)

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

val(City) = 150
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 do the join with 1.7K blocks in STUDENT



Total cost:
5.3K + **106K***1.7K
≈ **180M** I/O accesses
→ Pointless if there are more filtered tuples than blocks in T_{ext}! You never need to scan T_{int} more than b_{ext} times (as in the nested loop with no conditions)

Scan and lookup (indexed nested loop)

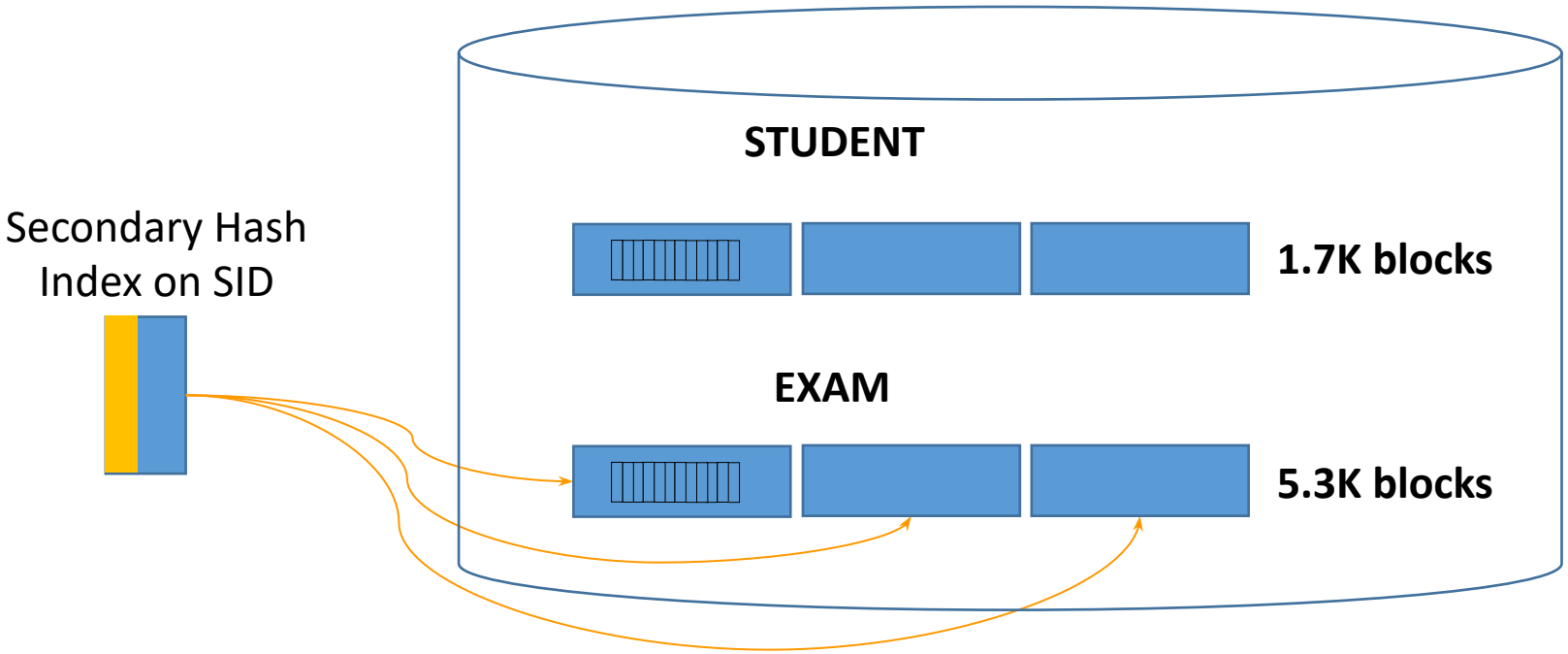
- If one table supports **indexed access in the form of a lookup based on the join predicate**, then this table can be chosen as **internal**, exploiting the predicate to extract the joining tuples without scanning the entire table
  - If both tables support lookup on the join predicate, the one for 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
    - Cost = $b_{Ext} + t_{Ext} \times cost\_of\_one\_indexed\_access\_to\_T_{Int}$
  - NOTE: as before, if the query has a filtering predicate on the external table then t_{Ext} is the subset of the tuples of the external table that satisfy the predicate (estimated)

Scan & lookup exploiting an index

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

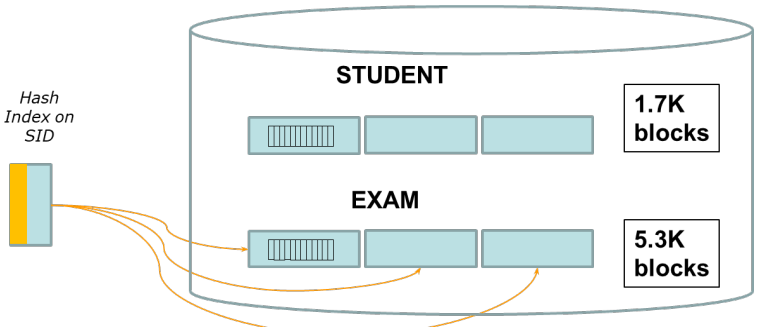
```
SELECT STUDENT.*
FROM STUDENT JOIN EXAM ON ID=SID
WHERE City='Milan' AND Grade='30'
```

val(City)=150



Filter + lookup: option 1 revised

- Improve option 1 (STUDENT as external table):
  - Scan Student, evaluate `City='Milan'` + lookup on Exam **exploiting the secondary hash index on EXAM**
- Scan all the STUDENT blocks: 1.7K
- Filter students from Milan → 150K tuples / 150 different cities = 1K
- Lookup with the hash index: for 1K students
  - 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 exams (and the **grades needed to evaluate the where condition**)



Hash index access for students from Milan

Scan of Student

Exam access for students from Milan

Total cost =

1.7K + 1K + 1K*12 =

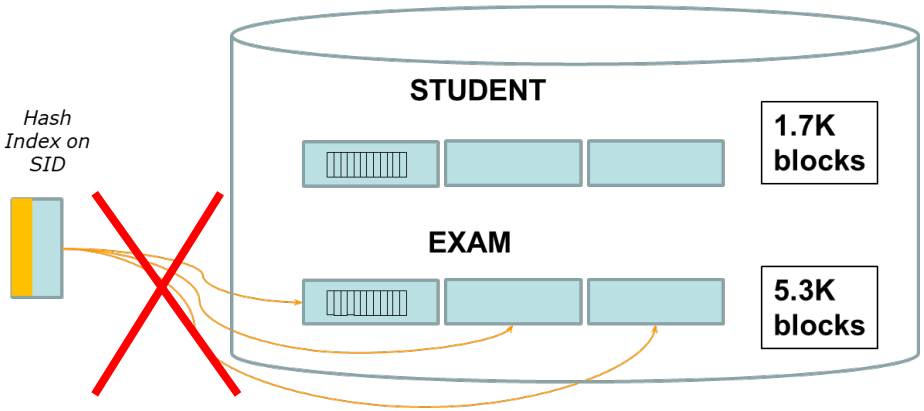
1.7K + 1K * (1+ 12) = 14.7K

Existence checking lookup

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

val(City)=150

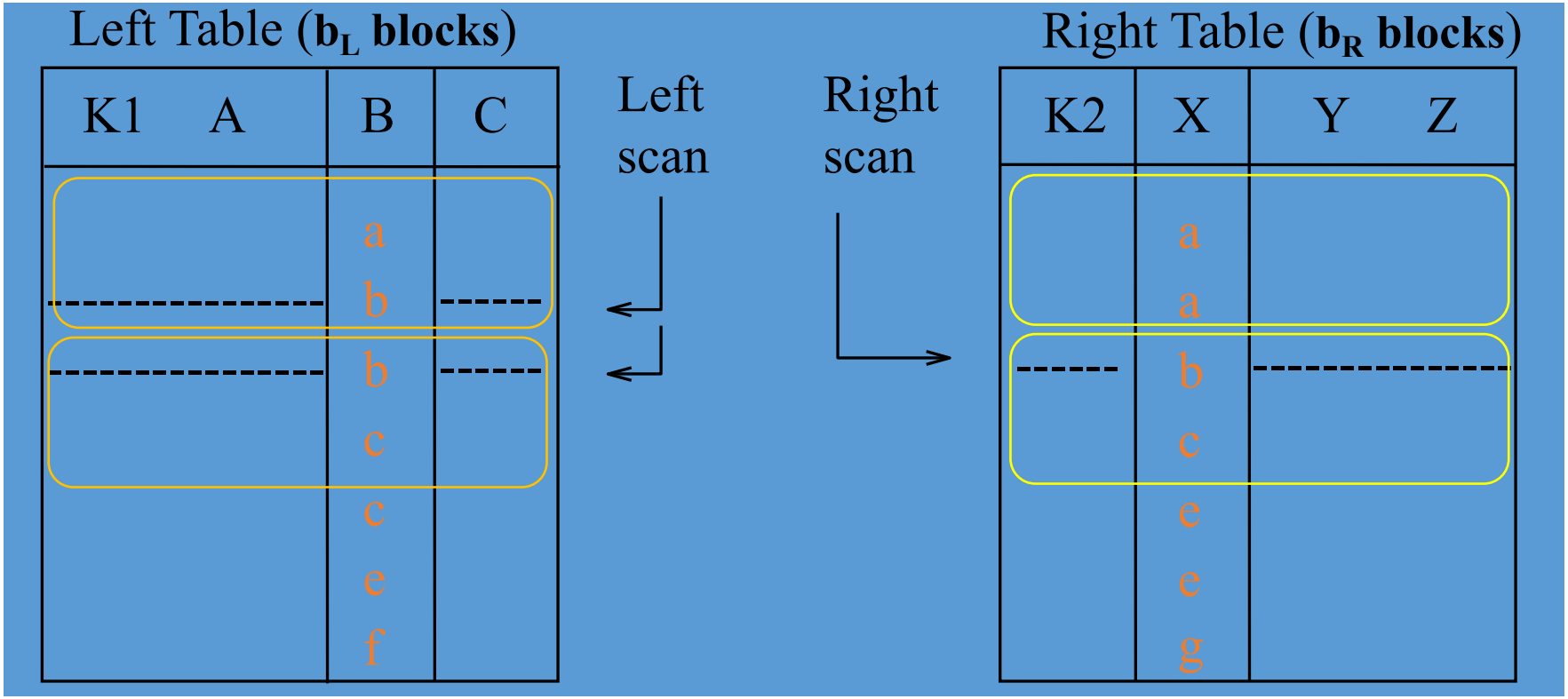
- Read all the STUDENT blocks: **1.7K**
- Filter students from Milan → 150K tuples/150 different cities = **1K**
- For 1K students use the hash index: for each student
  - access to the hash index (without overflow chain): **1**
- No need to access the full EXAM tuples to do the join, only check that the joined tuple pair exists



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

Merge-scan join

- This join is possible only if both tables are ordered according to the same key attribute, that is also the attribute used in the join predicate $L \bowtie_{B=X} R$



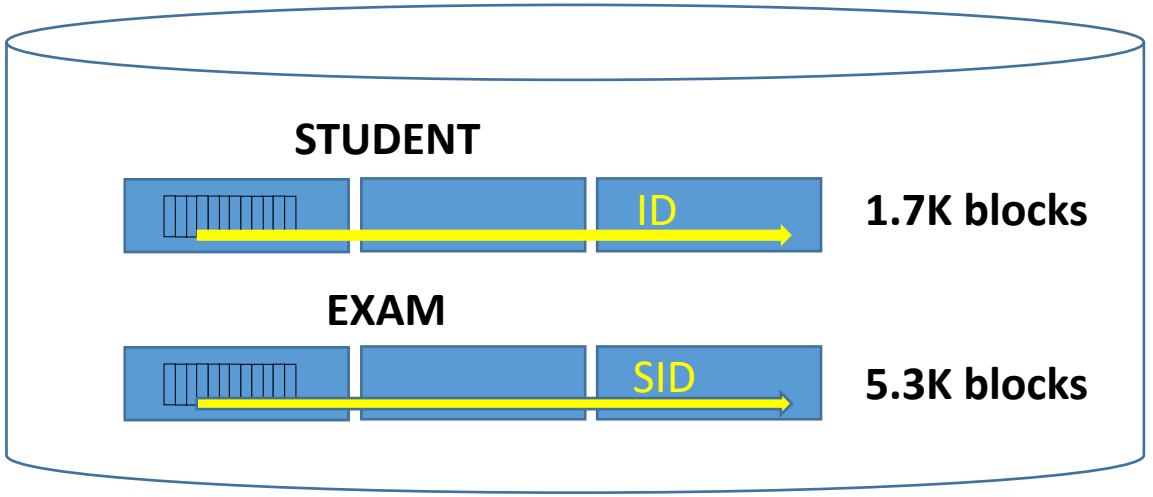
Sort + merge-scan join

- 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 of the join attribute
- Output result tuples <l, r>
- The cost is **linear in the size of the tables**
  - **$C = b_L + b_R$**
- The ordered full scan is possible for a table **if the primary storage is sequentially ordered wrt the join attribute or a B+ on the join attribute is defined**
- If a table needs to be sorted, add also the cost for sorting
  - $2 * b_L$ * (# of passes)
  - $2 * b_R$ * (# of passes)
    - # of passes cost = $1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil$, where N = # of blocks, B = # of buffer pages

Merge scan join on ordered tables

```
SELECT STUDENT.*
FROM STUDENT JOIN EXAM ON ID=SID
```

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

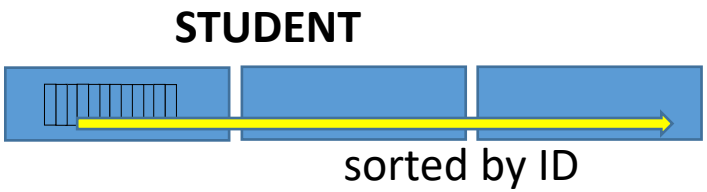


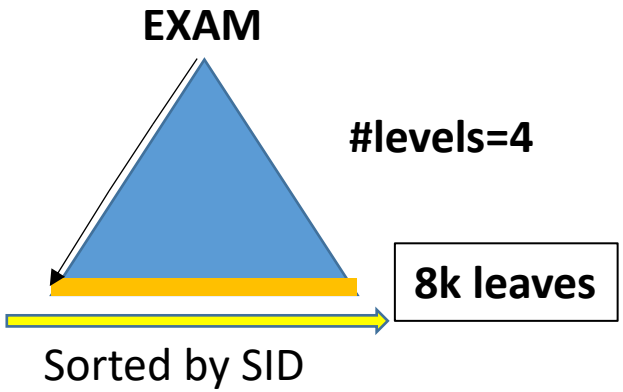
$C = b_{Student} + b_{Exam} = 1.7K + 5.3K = \mathbf{7K}$ I/Os

Indexed merge scan join

```
SELECT STUDENT.*
FROM STUDENT JOIN EXAM ON ID=SID
```

- Suppose
  - STUDENT: Sequentially-ordered by ID
  - EXAM: **primary** storage: B+ on SID (e.g., with 8k leaves)
- C = $b_{Student}$ (+ #intermediateNodes) + **#LeafNodes**$_{Exam}$
- C = 1.7k + 3 + 8k ≈ 9.7k

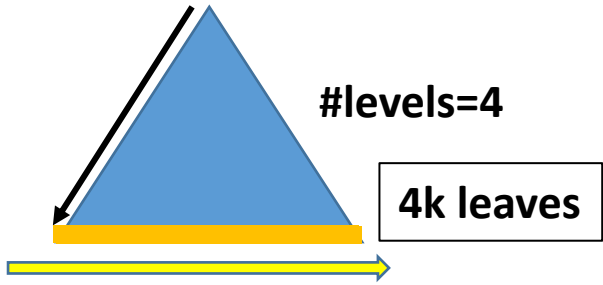


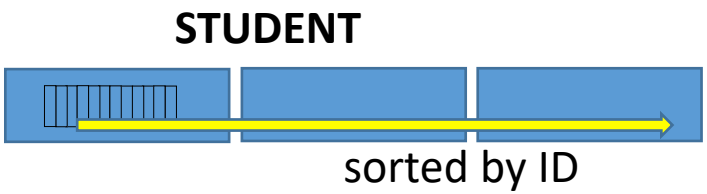


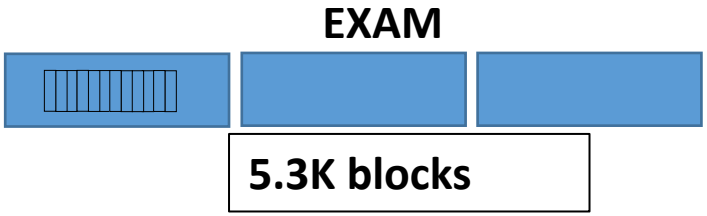
Indexed merge scan join

```
SELECT STUDENT.*
FROM STUDENT JOIN EXAM ON ID=SID
```

- Suppose
  - STUDENT: Sequentially-ordered by ID
  - EXAM: **secondary** index: B+ on SID
- C = $b_{Student}$ (+ #intermediateNodes) + **#LeafNodes**$_{Exam}$
- C = 1.7k + 3 + 4k ≈ **5.7k**
- NOTE:
  - Exam tuples are not needed, so B+ leaves are sufficient
  - One pointer <--> one tuple (SID is the PK)



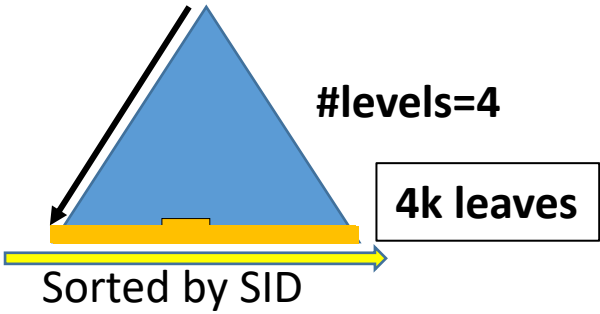


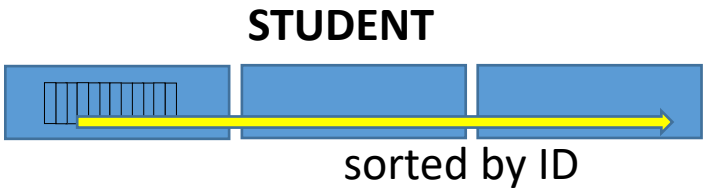


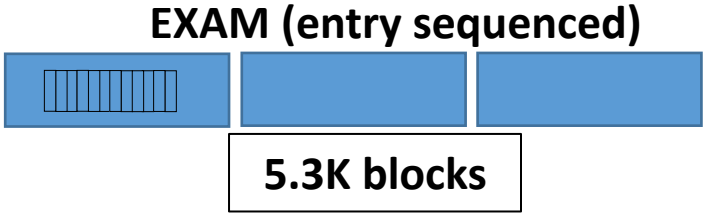
Indexed merge scan join

```
SELECT S.SID
FROM STUDENT S JOIN EXAM E ON ID=SID
WHERE GRADE <> '18' -- exam tuples must be accessed too
```

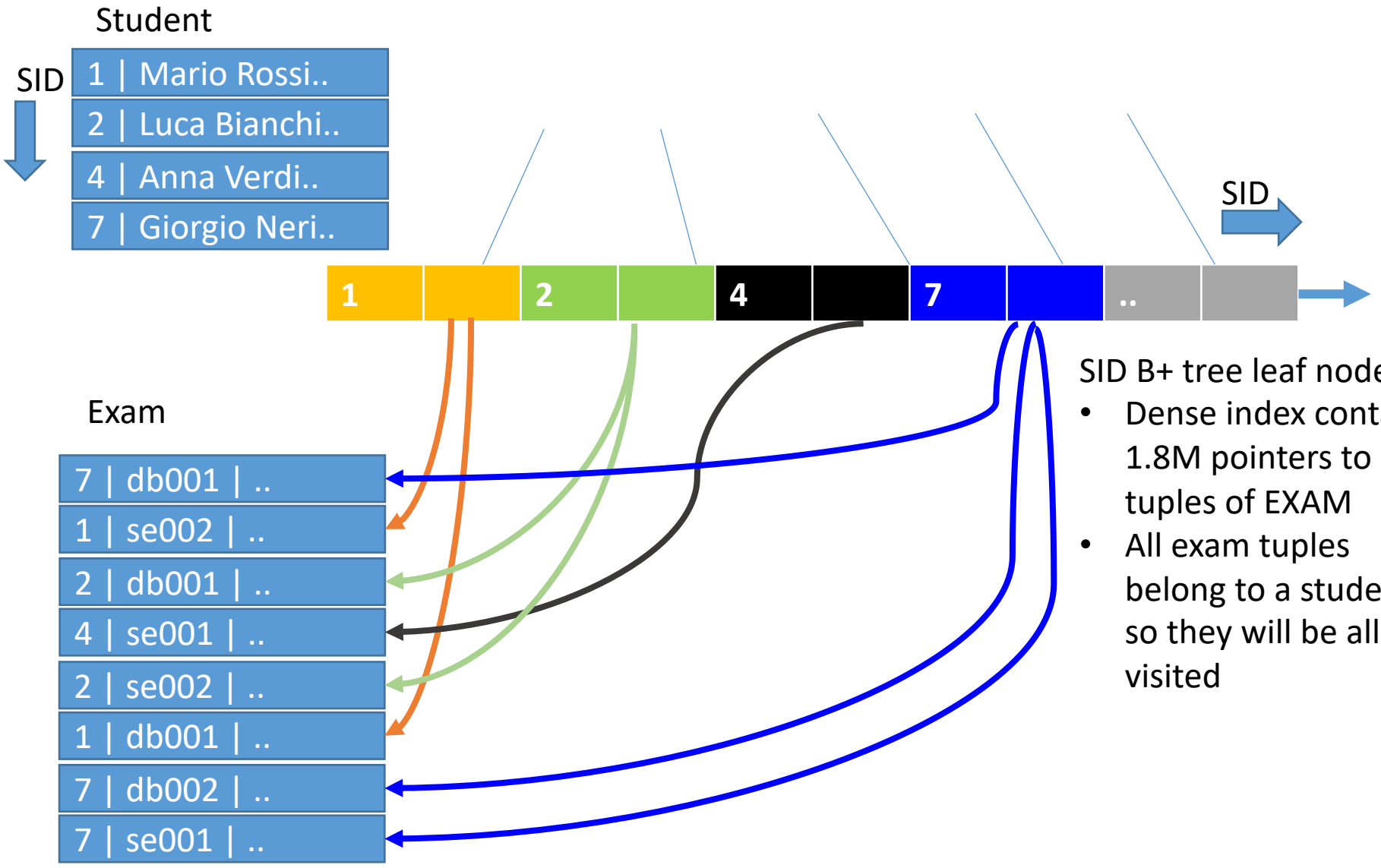
- Suppose
  - STUDENT: Sequentially-ordered by ID
  - EXAM: **secondary** index: B+ on SID
  - No statistics on grades
- C = $b_{Student}$ + #intermediateNodes + $\#LeafNodes_{Exam}$ + $\mathbf{\#Tuples}_{Exam}$
- C = 1.7k + 3 + 4k + 1.8M ≈ **1.8M**







Why 1.8M pointers to follow

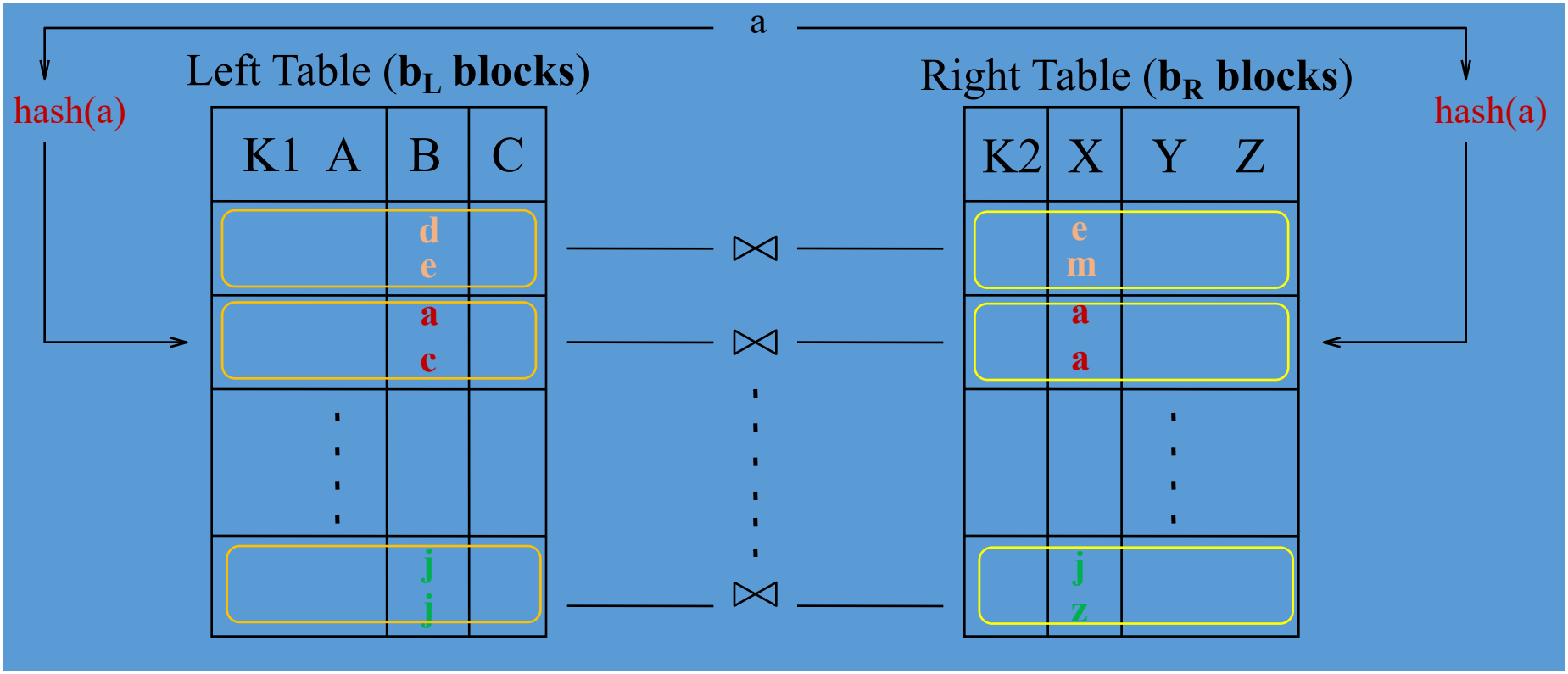


SID B+ tree leaf nodes

- Dense index contains 1.8M pointers to all tuples of EXAM
- All exam tuples belong to a student so they will be all visited

Hashed join

- This join is possible only if both tables are hashed on the same key (join) attribute
- Find the first/next key value v in the left table, use the hashed access to fetch the tuples with key = v from left and right table, add to the result the joined tuples that match



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

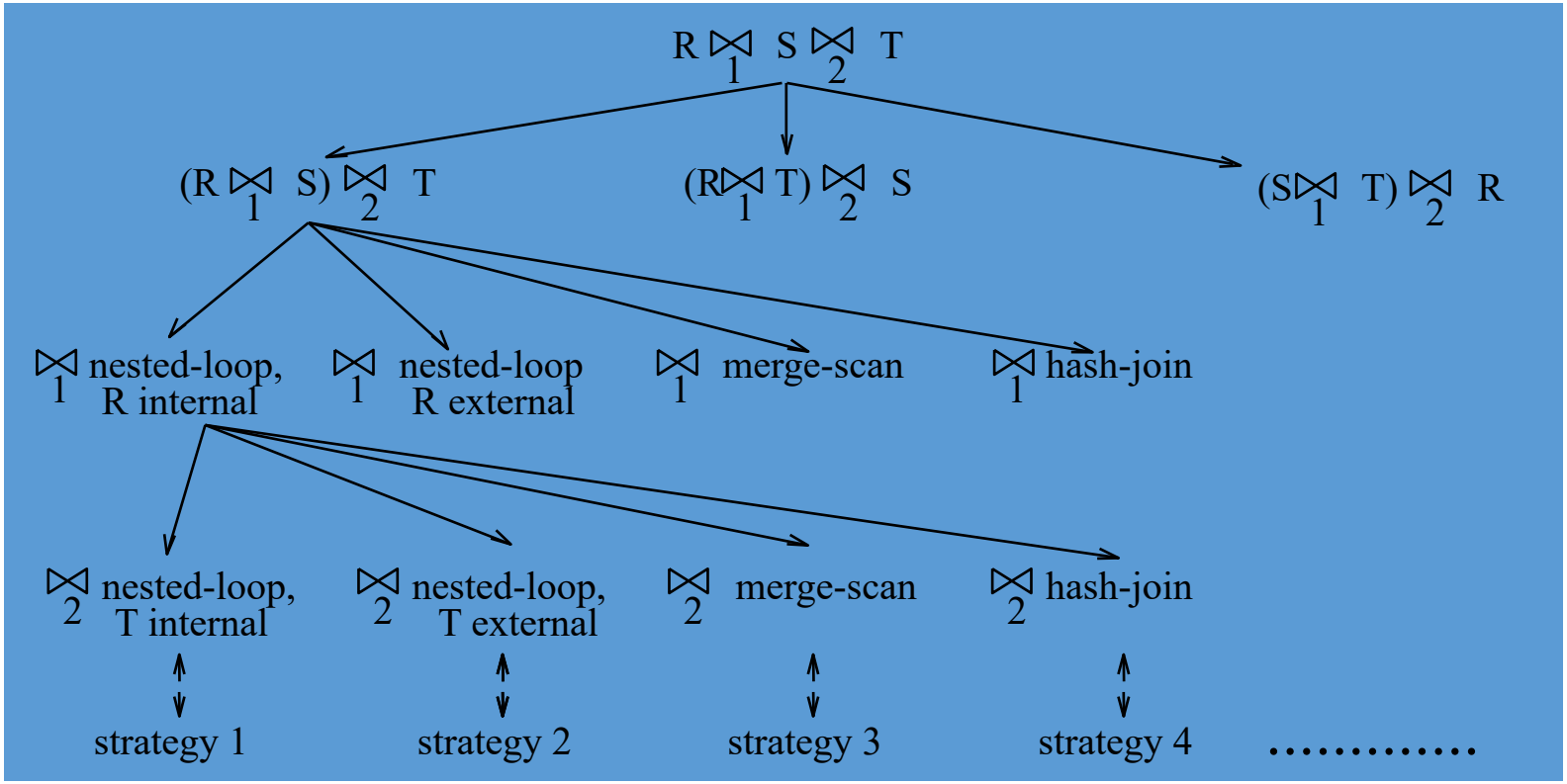
Cost-based optimization

- 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 for each operation (e.g., 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

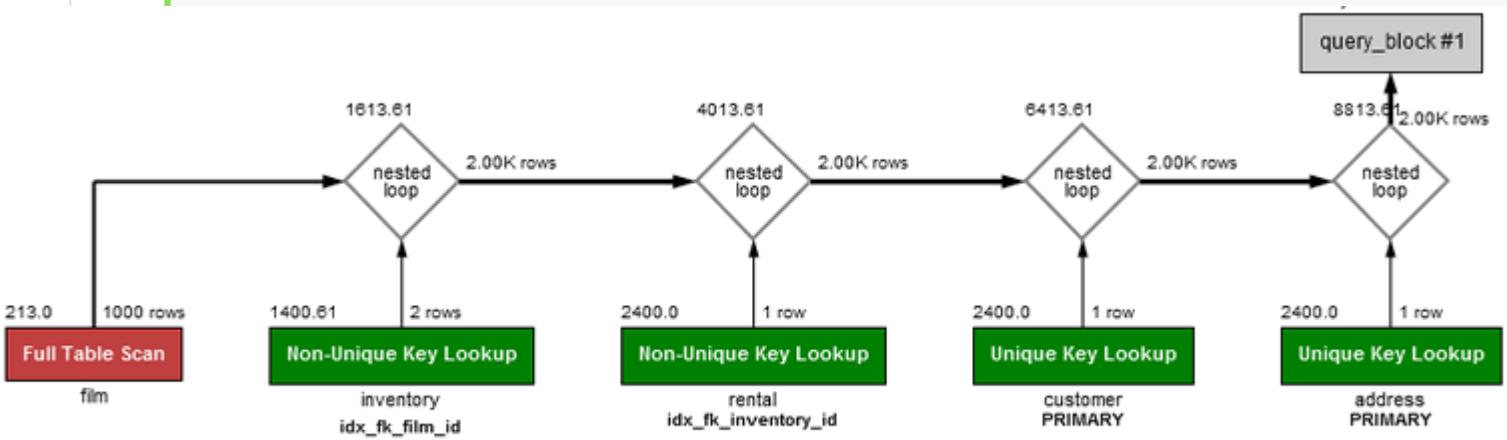
- **Compile and store**: the query is compiled once and executed many times (**prepared statement**)
  - The internal code is stored in the DBMS, together with an indication of the dependencies of the code on the particular versions of catalog used at compile time
  - On relevant changes of the catalog, 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: Elmasri-Navathe)

Query plan in MySQL workbench

```sql
SELECT CONCAT(customer.last_name, ', ', customer.first_name)
  AS customer, address.phone, film.title FROM rental
INNER JOIN customer ON rental.customer_id = customer.customer_id
INNER JOIN address ON customer.address_id = address.address_id
INNER JOIN inventory ON rental.inventory_id = inventory.inventory_id
INNER JOIN film ON inventory.film_id = film.film_id
WHERE rental.return_date IS NULL
AND rental_date + INTERVAL film.rental_duration DAY < CURRENT_DATE()
LIMIT 5;
```



https://dev.mysql.com/doc/workbench/en/wb-performance-explain.html

Determine the execution plan of your query

- “EXPLAIN PLAN” SQL statement
- Oracle: http://docs.oracle.com/cd/B28359_01/server.111/b28274/optimops.htm
- SQLite: http://www.sqlite.org/eqp.html
- MySQL: https://dev.mysql.com/doc/refman/8.0/en/execution-plan-information.html
- MS SQL server: http://msdn.microsoft.com/en-us/library/ms176005(v=sql.105).aspx