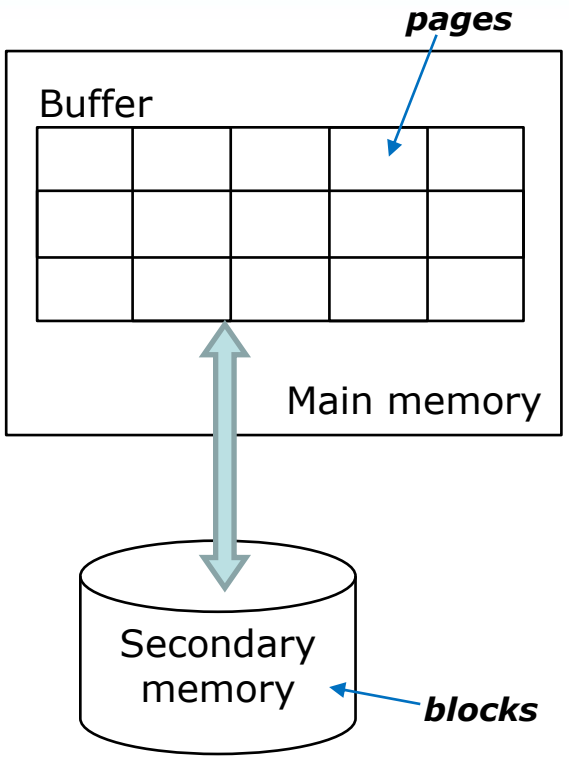
Databases 2

8 Physical data structures and query optimization

DATA ACCESS and COST MODEL

Main and Secondary memory (1)

- Programs typically refer to data stored in main 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
  - (which explains the terms "main" and "secondary")



Main and Secondary memory (2)

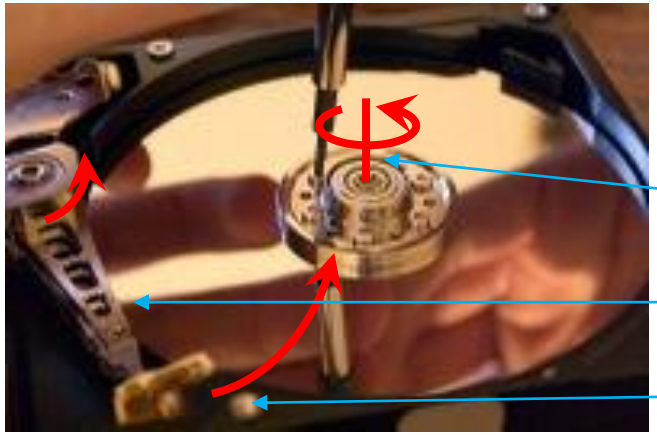
- Secondary memory devices are organized in **blocks** of (usually) **fixed** length (order of magnitude: a few KBytes)
  - The size of a block is decided when the disks are formatted
- Typically, the only available input/output operations for such devices are reading and writing **one block at a time**
  - **i/o operation**: moving a block (also called a ***page***) from secondary to main memory and vice-versa, respectively;
- Consolidated terminology uses *page* and *block* to refer to the transferred units w.r.t. the buffer management/access structures and the disk management/file system respectively. For convenience and simplicity, we will consider them synonyms, and mostly use **block** in the following, unless we need to distinguish.

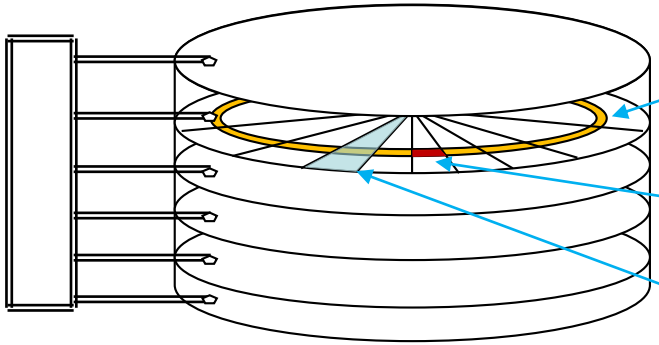
Main and Secondary memory (2)

- How long does it take to read a **block** from a disk?
- In principle, this is strictly dependent on the technology
- However, generally speaking, the larger the memory size (and the cheaper the device), the slower the access
  - For very large datasets, cheap storage is the only viable option
- For decades:
  - Mechanical hard drives
- More recently:
  - Solid State Drives (SSD)

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)





Main and Secondary memory (3)

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

  on average, reading one block takes **hardly less than 10 ms overall**
- 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 (1)

- The File System (FS) is the component of the Operating Systems which manages access to secondary memory
- DBMSs make **limited use of FS functionalities**:
  - only to create/delete files and to read/write *single blocks* or *sequences of consecutive 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** (so as to support faster sequential reads)

Random and Sequential access to blocks

- We have seen that reading one block from a disk takes at least 10ms, while the actual data transfer takes 1ms
- This is **true for random access** to blocks (i.e., when the position of each next block to read is unknown and arbitrary)
  - the seek and latency cost are paid for each access
- However, for the **sequential scan of large files**, if the next blocks to be read are deliberately allocated so that they will "pass under" the heads just after the transfer of the previous ones is complete, then the latency and seek cost is paid "only once", at the beginning of the scan
  - Only the first access is a random one

Random and Sequential access to blocks

- **Random** access transfer time
  - T_R =~ 10-15 ms
  - 1 block → 1 T_R 100 blocks → 100 T_R
- **Sequential** access transfer time
  - T_s =~ 1 ms
  - 1 block → 1 T_R 100 blocks → 1 T_R + 99 T_s =~ 10 T_R
- If reading many sequential blocks (suitably allocated onto the disk) in a sequence, T_s can be 5 to 15 times smaller than T_R

DBMS and file system (2)

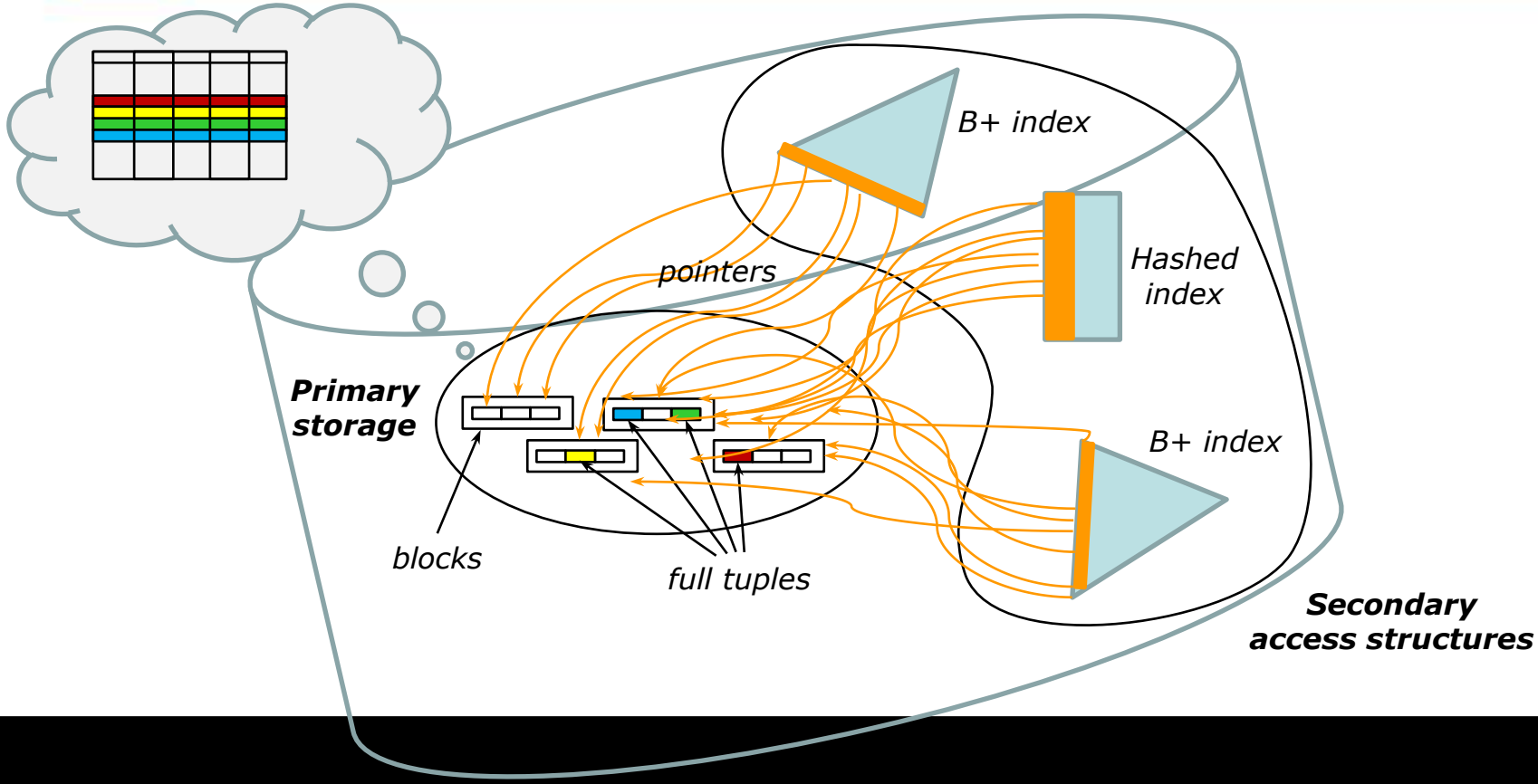
- The DBMS manages the blocks of its files as if they were a single large space in secondary memory
- In this space, the DBMS builds the physical structures where tables are stored
- A file is typically dedicated to a single table, but…
  - …it may happen that a file contains data belonging to more than one table, and that the tuples of one table are split into more than one file

PHYSICAL ACCESS STRUCTURES

Physical access structures

- Used for the efficient storage and manipulation of data within the DBMS
  - The DBMS does not simply rely on the file system
- Encoded as ***access methods***, that is, software modules that provide data access and manipulation **primitives** for each physical access structure
- Each DBMS has a distinctive and limited set of access methods
- 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

- The **primary** structure contains all the tuples of a table
  - Its main purpose is 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
  - Their main purpose is to speed up the search for specific tuples within the table, according to some search criterion
- We will study **three 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**

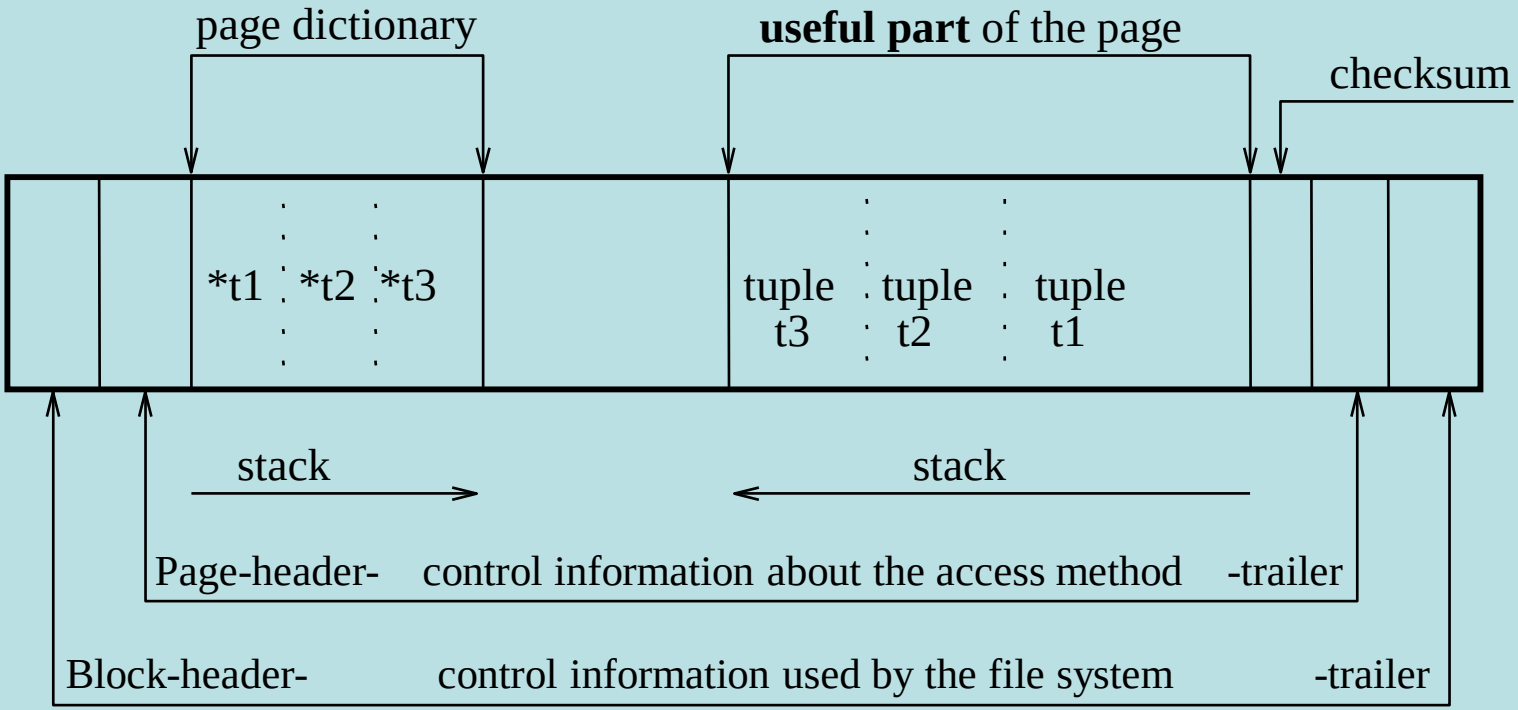
Physical access structures

- We will first see how tuples are stored in the blocks of a primary structure (sequential or hash-based)
- Then, we'll discuss the organization of the blocks within a tree-based secondary structure
- Secondary hash structures and primary tree structures will be quickly addressed

Blocks and tuples

- Blocks (the "physical" components of files) and tuples (the "logical" components of tables) generally have different sizes:
  - 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
    - Tables have tuples of variable size as they contain optional (possibly `null`) values and attributes of type `varchar` (or other types of non-fixed size)
    - In most cases, the size of a tuple is an *average* size

Organization of tuples within pages



Organization of tuples within pages (blocks)

- Each access method has its own **page** organization
- In the case of *sequential* and *hash-based* methods each **page** has:
  - An initial part (***block header***) and a final part (***block trailer***) containing control information used by the **file system**
  - An initial part (***page header***) and a final part (***page trailer***) containing control information about the **access method**
  - A ***page dictionary***, which contains pointers 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
- *Tree-based* structures have a different internal page organization

Block Factor

- The block factor **B** is the number of tuples within a block
  - $\mathbf{S_R}$: Average size of a tuple ("fixed length record" assumption)
    - In reality, tuples have large variability in their size
  - $\mathbf{S_B}$: Size of a block
  - if $S_B > S_R$, there may be many tuples in each block:

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

- 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 (there is enough space to store the extra bytes) or even usage of a new page (if more space is required)
- ***Deletion*** *of a tuple*
  - often carried out by marking the tuple as 'invalid'
- ***Access*** *to a* ***field*** *of a particular tuple*
  - after identifying the tuple by means of its key or its offset, the field is identified according to the offset and the length of the field itself

Sequential structures

- Characterized by a sequential arrangement of tuples in the secondary memory
- Three cases: **entry-sequenced**, **array**, **sequentially-ordered**
  - In an ***entry-sequenced*** organization, the sequence of the tuples is dictated by their order of entry
  - In an ***array*** organization, the tuples (all of the same size) are arranged as in an array, and their positions depend on the values of an index (or indexes)
  - In a ***sequentially-ordered*** organization, the tuples are ordered according to the value of a ***key*** (typically one field, but may be obtained by combining more than one attribute)

"Entry-sequenced" sequential structure

- **Optimal** for **space occupancy**, as it uses all the blocks available for files and all the space within the blocks
- **Optimal** for 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 (highly selective access)
    - May require scanning the whole structure
  - Updates that increase the size of a tuple
    - All following tuples must be "shifted" on

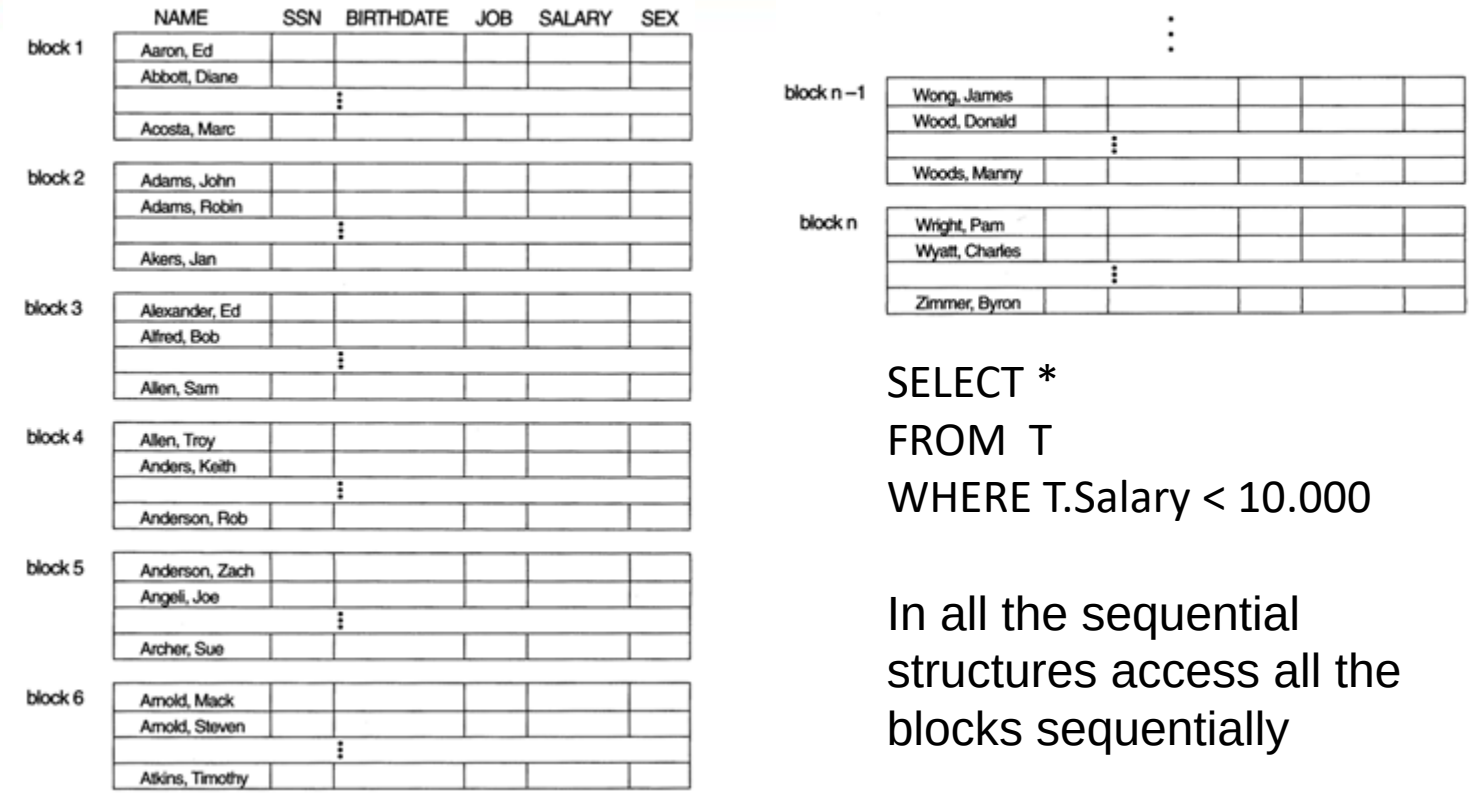
"Array" sequential structure

- Each tuple has a numerical index *i* and is placed in the *i*-th position of the array
- 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
  - Otherwise, a lot of disk space is wasted

"Sequentially-ordered" sequential structure

- Each tuple has a position based on the value of the **key field**
- Historically, these structures were used on sequential devices (tapes) This has fallen out of use, but for data streams and system logs
- Main problem: *insertions* and *updates that increase the data size* - they require reordering techniques for the tuples already present:
- Options to avoid global reordering (or to limit the impact):
  - Differential files (example: yellow pages)
  - Leaving a certain number of slots free at the time of first loading, followed by 'local reordering' operations
  - Integrating the sequentially ordered files with an *overflow file*, where new tuples are inserted into blocks linked to form an *overflow chain*

Example:




```
SELECT *
FROM  T
WHERE T.Salary < 10.000
```

In all the sequential structures access all the blocks sequentially

Hash-based access structures

- Ensure an efficient ***associative*** access to data, based on the value of a ***key*** field to which the hash is applied
- A hash-based structure has N_B ***buckets***, each typically of the size of 1 block (the buckets are often all adjacent in the file)
- A **hash** algorithm is applied to the key field so as to compute a value between 0 and N_B-1
  - This value is interpreted as the index of a bucket (and as the position of the corresponding block in the file)
- This is the most efficient technique for queries with equality predicates, but it is rather inefficient for interval queries

Features of hash-based structures

- Primitive interface: `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
- Optimal performance if the file is larger than necessary. Let:
  - *T* be the number of tuples to be stored in the hash structure,
  - *B* be the block factor (average number of tuples per page);

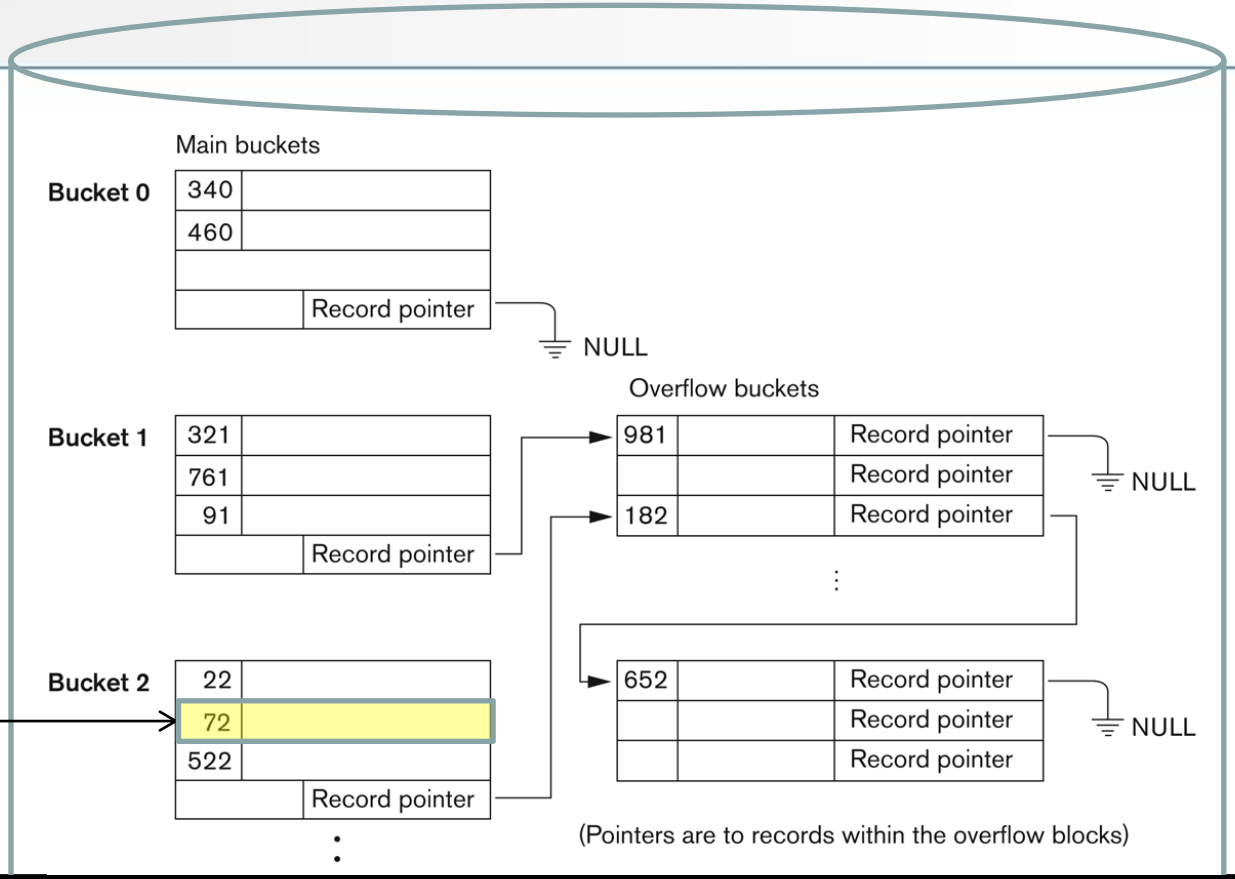
  then a good choice for N_B is T/(0.8 x B), using only 80% of the available space

8 Physical data structures and query optimization

SELECT *
FROM T
WHERE T.ID = ’72’

KEY K=72

Hash function
h(K) = K mod 10

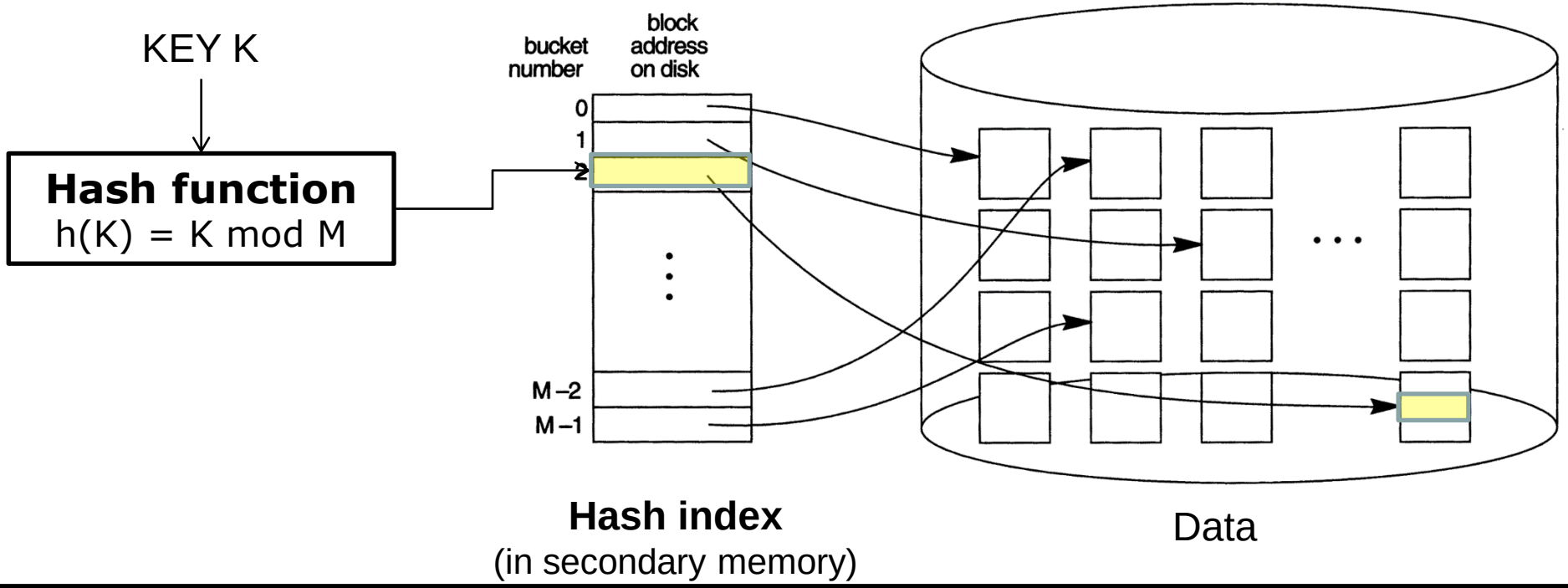


Secondary hash-based structures

- A secondary hash-based index is shaped and managed exactly like a primary one, but for the fact that instead of the actual tuples its nodes only contain **key values and pointers**
- The pointers to the blocks of the primary storage are arranged within buckets (ideally, all buckets of size 1 block)
  - On average, a bucket of a secondary hash index has more free space than the corresponding primary storage, if the hash function is the same (as the key field is much smaller than the full tuple)

Hash-based index

SELECT * FROM T WHERE T.ID = ’72’



Collisions

- A collision between two tuples occurs whenever the same bucket is associated with them. Collisions become critical when the maximum number of tuples per block is exceeded (bucket size > block size)
- Collisions are solved by adding an overflow chain
  - This gives the additional cost of scanning the chain
- The average length of the overflow chain is a function of the ratio $T/(B \times N_B)$ and of the block factor B:

$T/(B \times N_B)$	1	2	3	5	10	*B*
.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	

An example

- 40 tuples
- Hash function: M mod 50
  - Values between 0 and 49
- The hash table therefore has 50 buckets:
  - 1 collision of 4 values
  - 2 collisions of 3 values
  - 5 collisions of 2 v
- Some buckets are empty

M	M mod 50
60600	0
66301	1
205751	1
205802	2
200902	2
116202	2
200604	4
66005	5
116455	5
200205	5
201159	9
205610	10
201260	10
102360	10
205460	10
205912	12
205762	12
200464	14
205617	17
205667	17

M	M mod 50
200268	18
205619	19
210522	22
205724	24
205977	27
205478	28
200430	30
210533	33
205887	37
200138	38
102338	38
102690	40
115541	41
206092	42
205693	43
205845	45
200296	46
205796	46
200498	48
206049	49

About hashing

- Performs best for direct (random) access when searching tuples based on equality of values of the key field
- Collisions (overflow) are typically managed writing in the next block with space available; sometimes, linked blocks go into an area called **overflow file**
- **Inefficient** for access based on interval predicates or based on the value of non-key attributes
- Hash files "degenerate" if the extra-space is too small (should be at least 120% of the minimum required space) and if the file size changes a lot over time

Tree structures

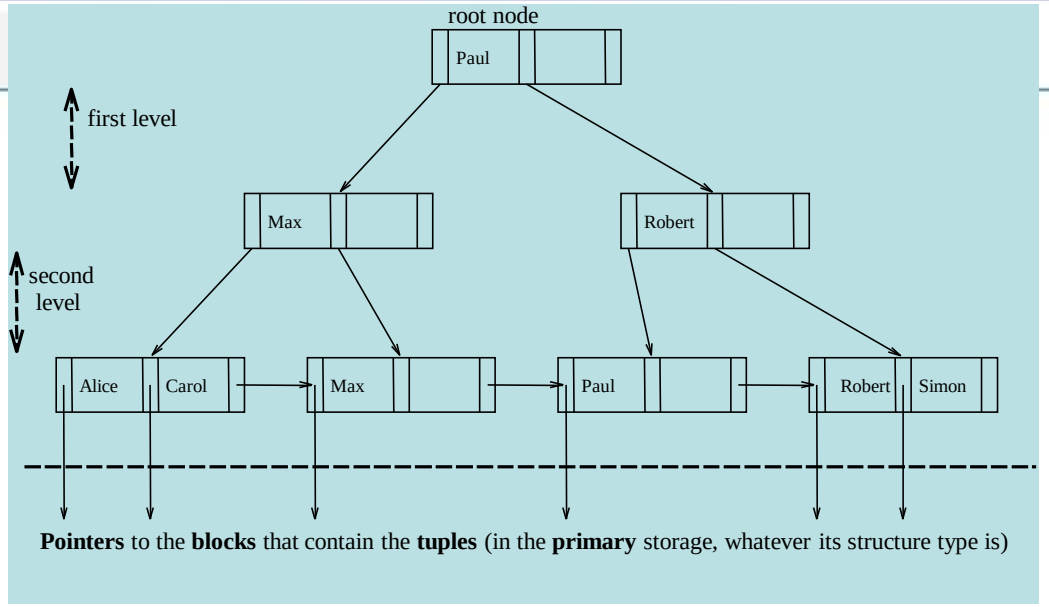
- Most frequently used in relational DBMSs for secondary structures
  - SQL indexes are implemented in this way
- Gives associative access based on the value of a ***key*** field
  - no constraints on the physical location of the tuples
- N.B.: the primary key of the table (w.r.t. the relational model) and the key fields for sequentially-ordered, hash-based and tree-based structures are different concepts

Secondary access structures (index files)

- **Index**: an auxiliary structure for the efficient access to the tuples of a primary structure, based on the values of a given attribute or set of attributes – called the index **key**
- The index concept: analytic index of a book, seen as a pair (term-page list), alphabetically ordered, at the end of a book
- Once more: the index key is not a primary key!

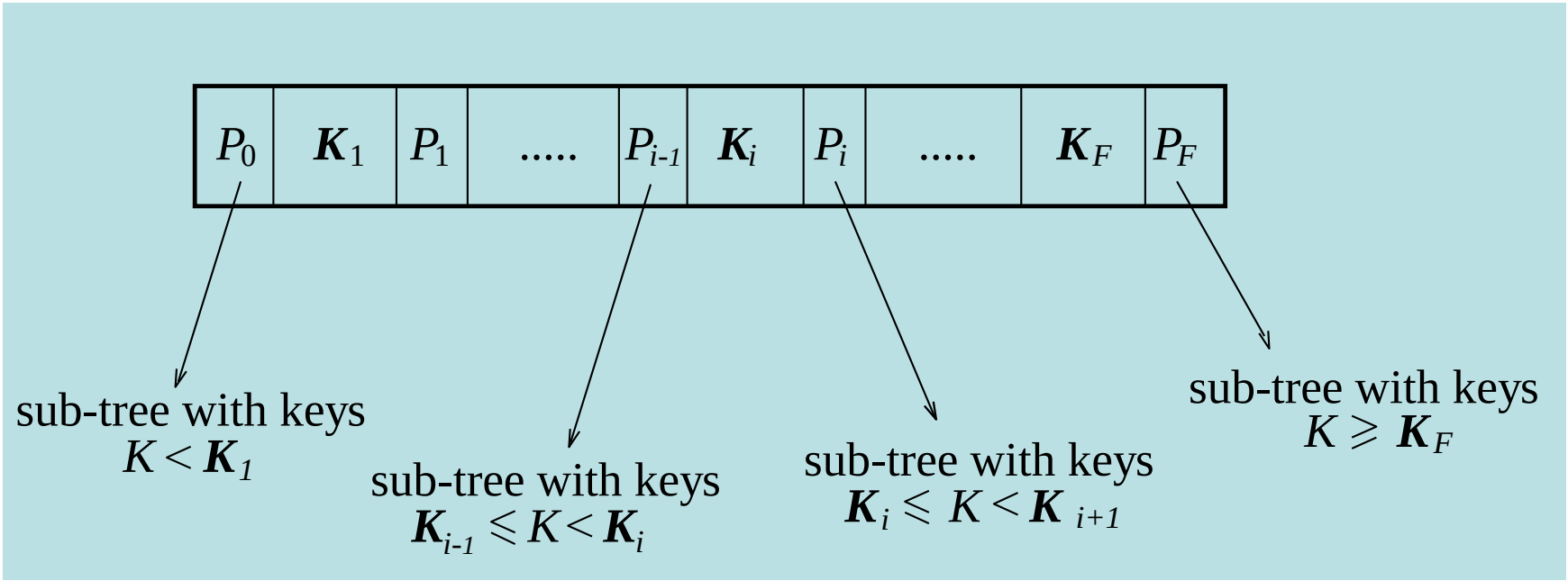
Tree structures

- Each tree has:
  - one root node
  - several intermediate nodes
  - several leaf nodes
- Each node corresponds to a block



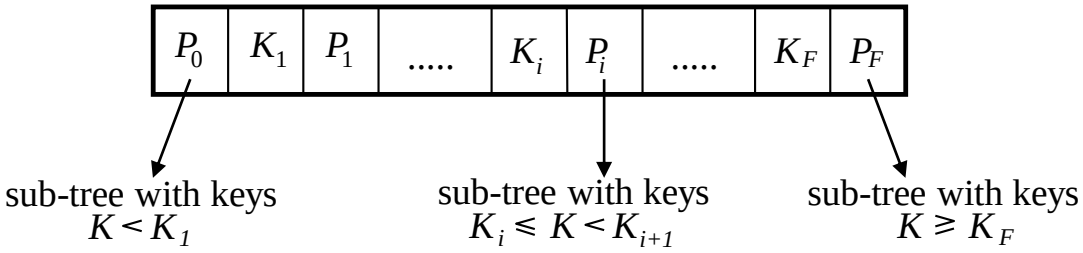
- The links between the nodes are established by **pointers** to mass memory
- In general, each node has a large number of descendants (**fan out**), and therefore the majority of pages are leaf nodes
- In a ***balanced tree***, the lengths of the paths from the root node to the leaf nodes are all equal. Balanced trees give **optimal** performance.

Structure of the tree nodes



Search technique

- 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 $K_j \leq V < K_{j+1}$**



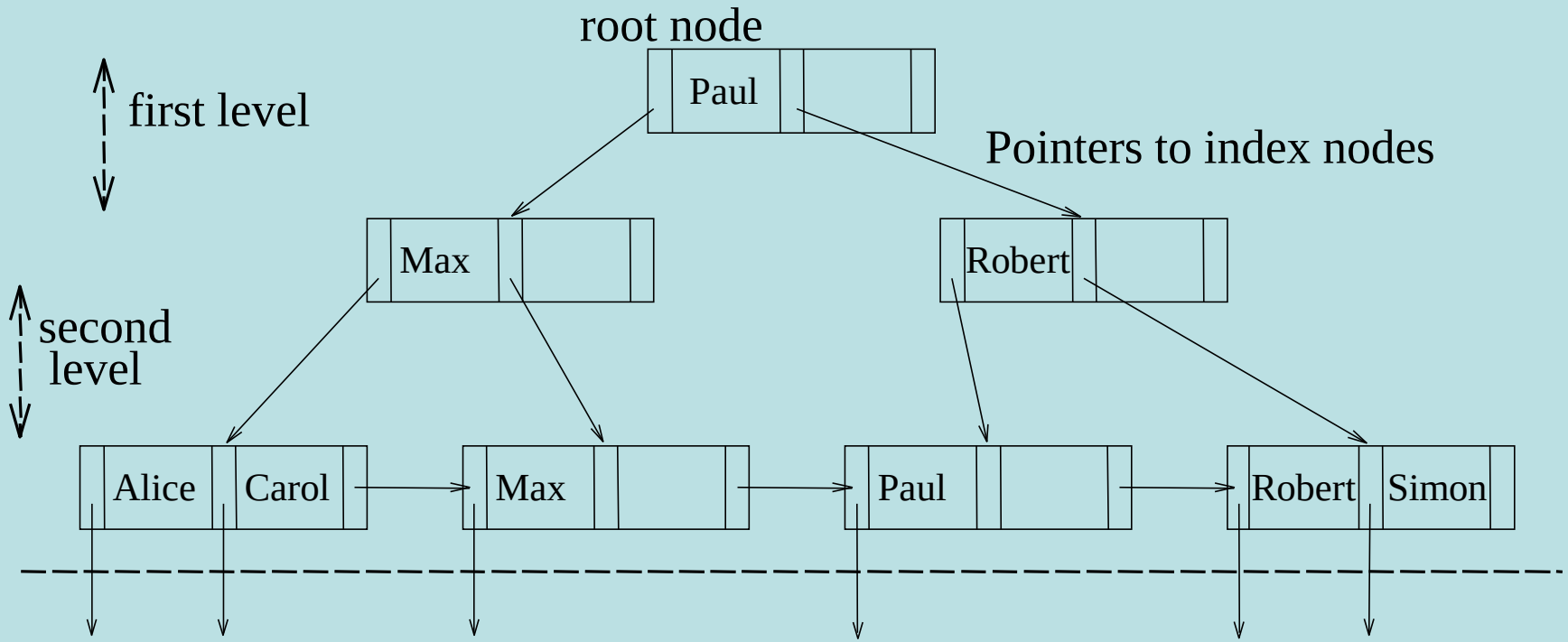
- The nodes can be organized in two ways:
  - Pointers to the tuples are only contained in the leaf nodes (**B+** trees)
  - Pointers to the tuples can also be contained in intermediate nodes
    - **B** trees: in this way, for tuples with $V=K_i$, we don't need to descend the tree down to the leaves

B and B+ trees

- **B+ trees**
  - The leaf nodes are linked in a chain ordered by the key
  - Supports interval queries efficiently
  - The most used by relational DBMSs
- **B trees**
  - No chain links for leaf nodes (ant thus no support for intervals)
  - Intermediate nodes use distinct pointers for each key value K_i
    - one points to a sub-tree with keys between K_i and K_{i+1}
    - the other(s) directly point(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)

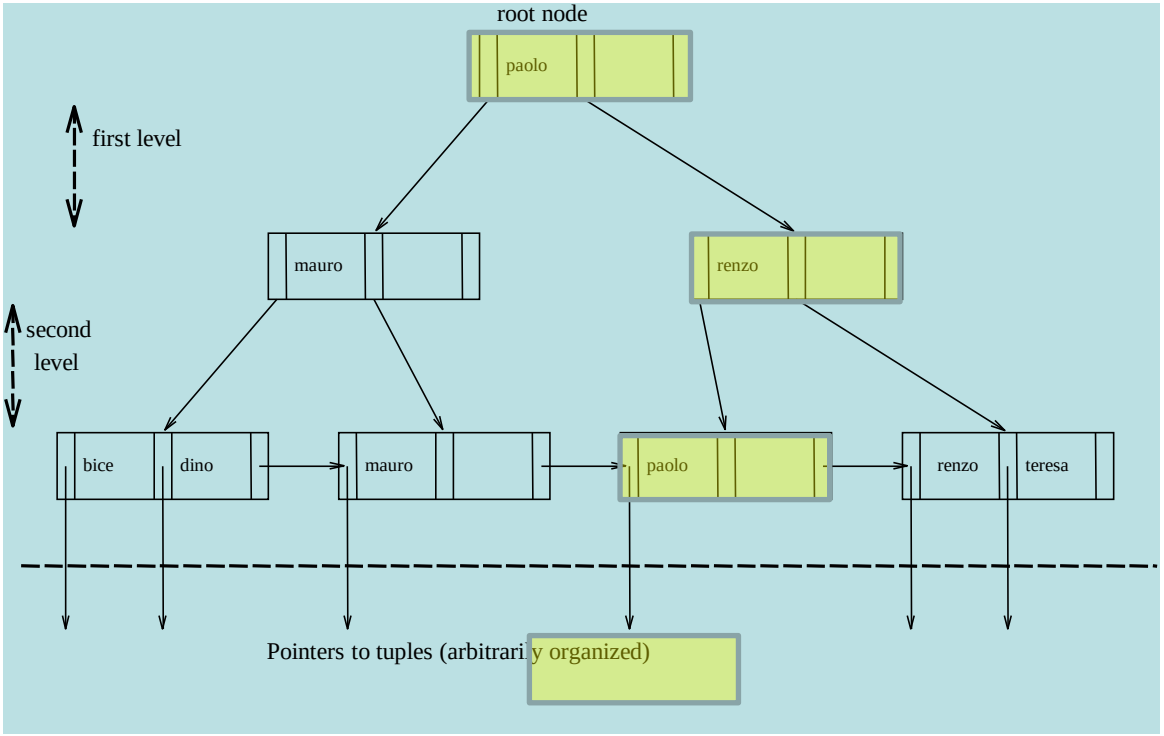
An example of secondary B+ tree



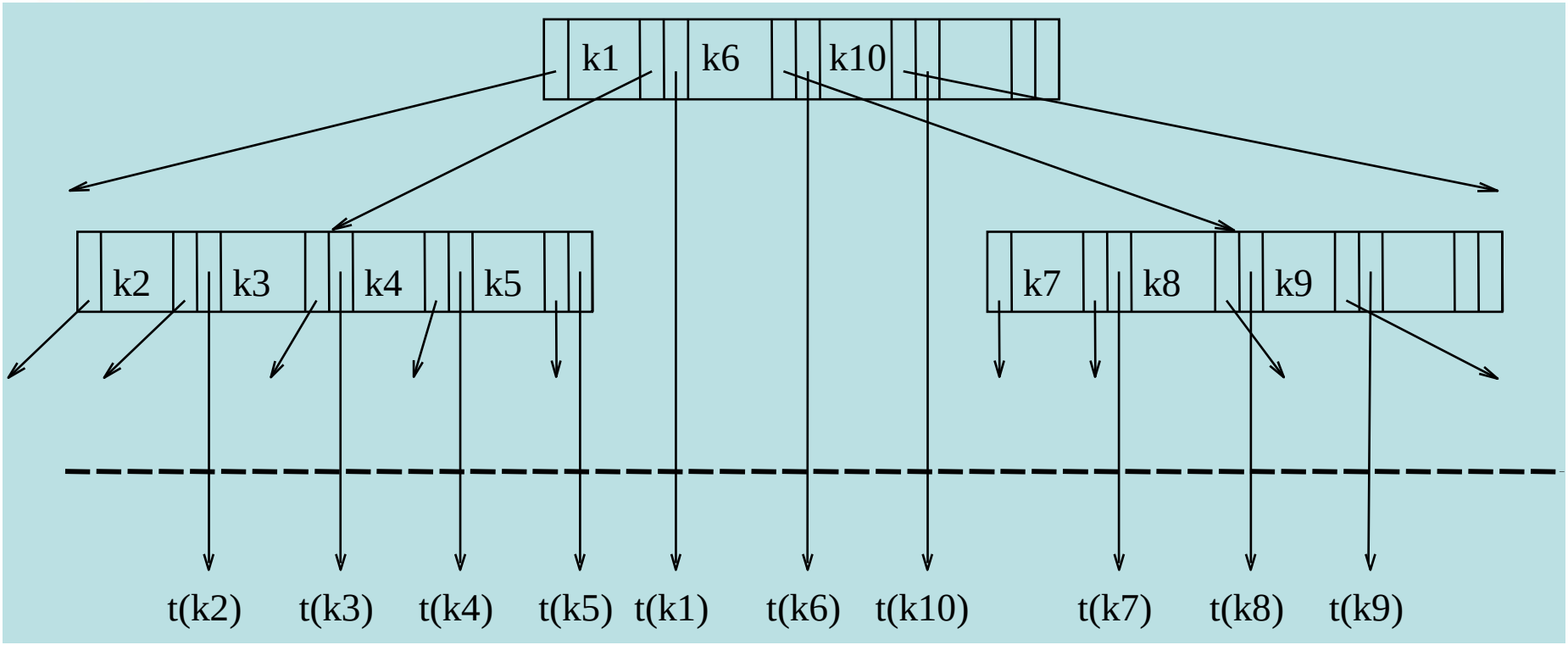
Pointers to the **blocks** that contain the **tuples** (in the **primary** storage, whatever its structure type is)

Example – B+ tree

```
SELECT *
FROM  T
WHERE T.NAME = 'paolo'
```

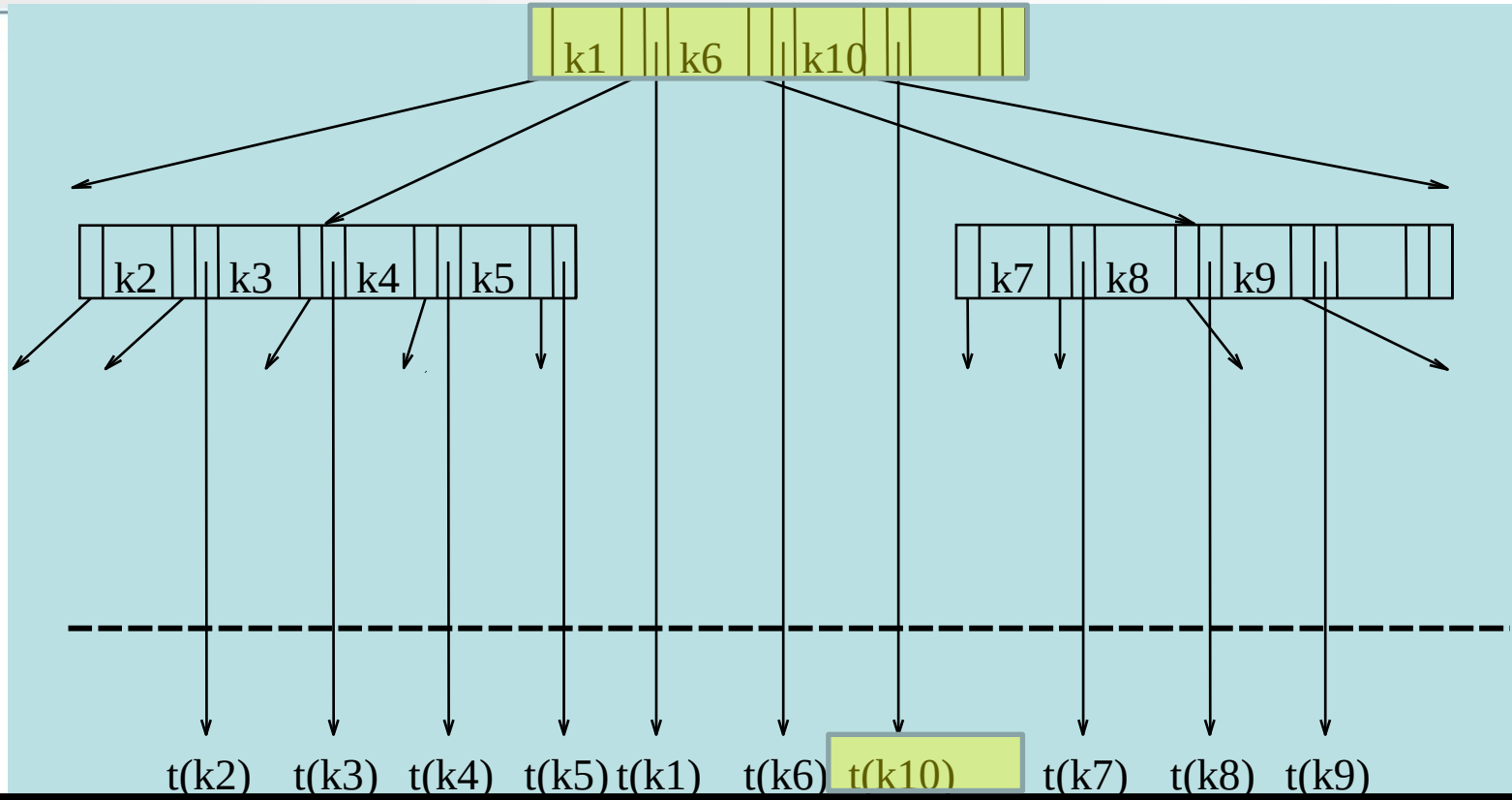


An example of secondary B tree



8 Physical data structures and query optimization

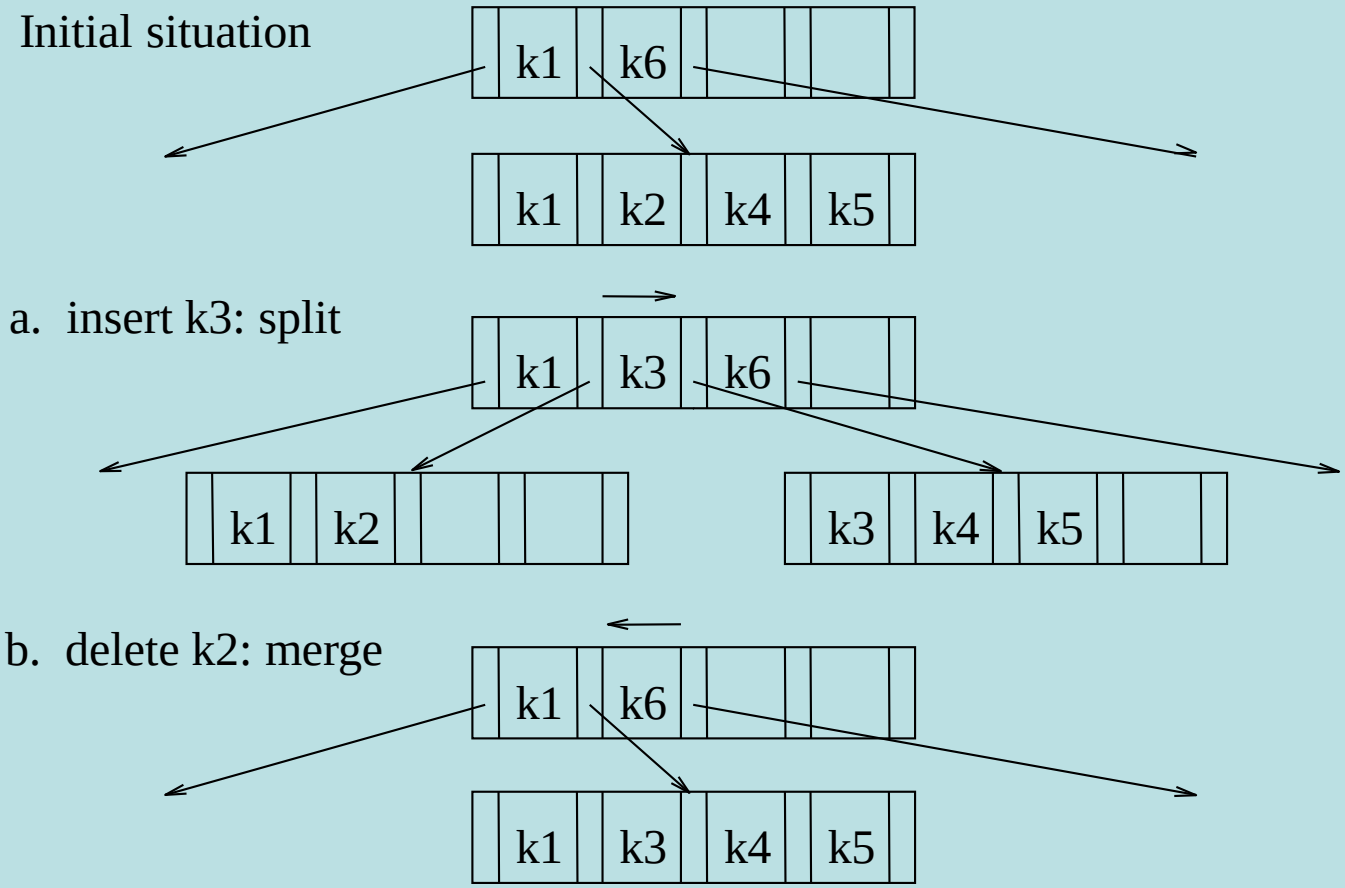
```
SELECT *
FROM  T
WHERE T.ID
      = 'k10'
```



Split and Merge operations

- **SPLIT**: required when the insertion of a new tuple cannot be done locally to a node
  - Causes an increase of pointers in the parent node and thus could recursively cause another split
- **MERGE**: required when two "close" nodes have entries that could be condensed into one node. Done in order to keep a high node filling ratio and minimal paths from the root to the leaves.
  - Causes a decrease of pointers in the parent node and thus could recursively cause another merge

Split and merge



Primary vs. Secondary tree structures

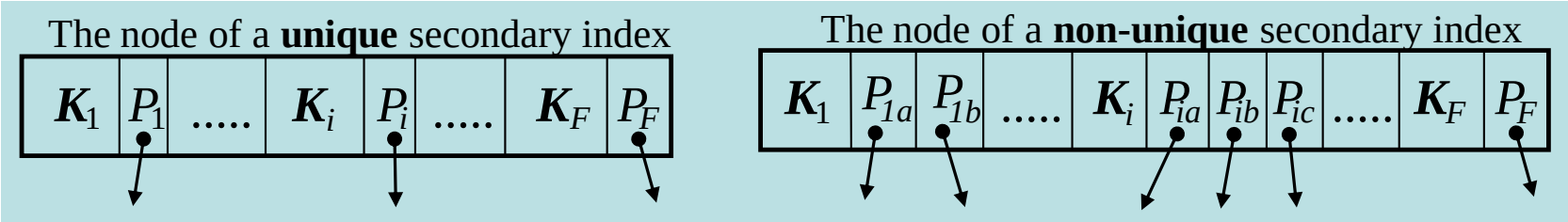
- Trees can be also used as **primary** access structures:
  - Tuples are stored in the tree nodes or in a file ordered according to the index key (also: clustered index)
    - Possibly a "*sparse*" index: with less index entries than the number of contained tuples, as tuples are nevertheless ordered
- 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" !)

Key values for key-based structures

- Many structures are built around the notion of a key:
  - Sequentially ordered
  - Hash-based (primary and secondary)
  - Tree-based (primary and secondary, B and B+)
- **Key values** may be:
  - **Simple** (taken just from one field/attribute of the table), or **Composite** (more than one field, juxtaposed)
  - **Unique** (if the key values are candidate keys for the table), or **Non-unique** (if several tuples might have the same value)
    - E.g.: (*Surname,Name,Birthdate*) may be a composite, non unique key for which the ordering depends on the Surname first, then on the Name, …

Unique vs. Non-unique secondary indexes

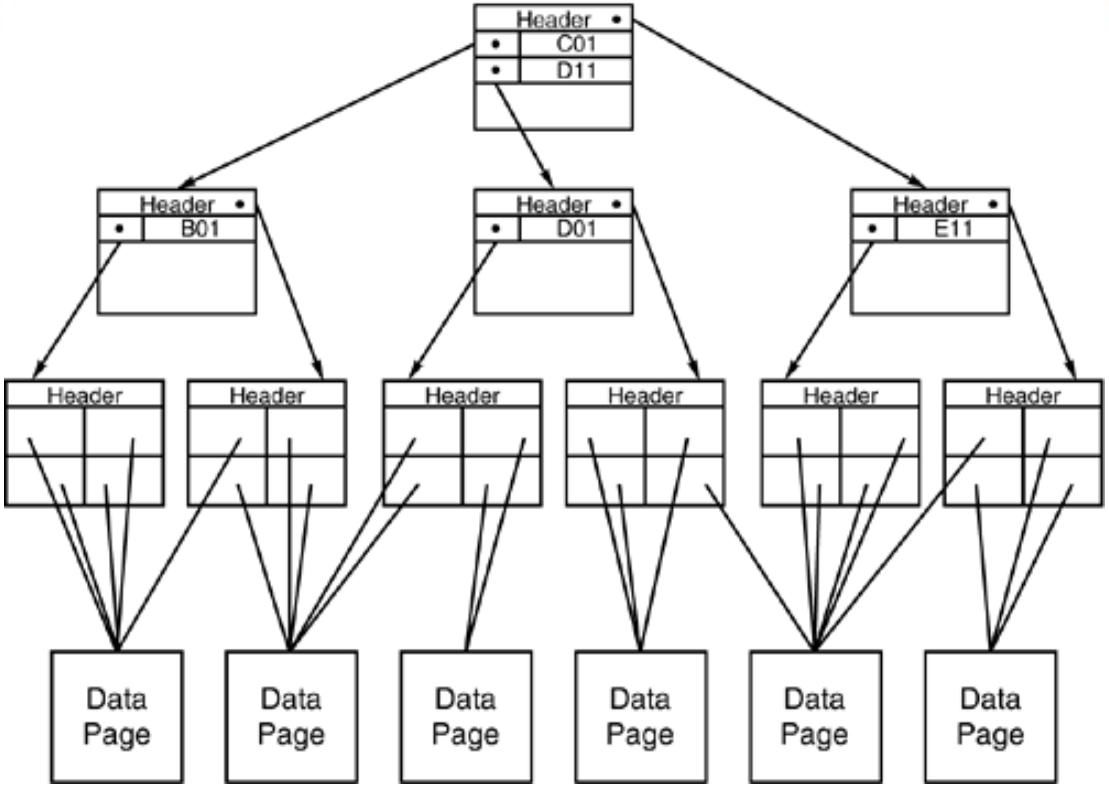
- A secondary index can be built over a (set of) attribute(s) that may or may not be unique for the indexed table
  - If they are **unique**, each key value is guaranteed to be associated with exactly one pointer (the pointer to the block that contains the only tuple with that value for the key)
  - If they are **non-unique**, key values may be associated with several pointers, as many as the blocks that contain all the tuples with that value for the key



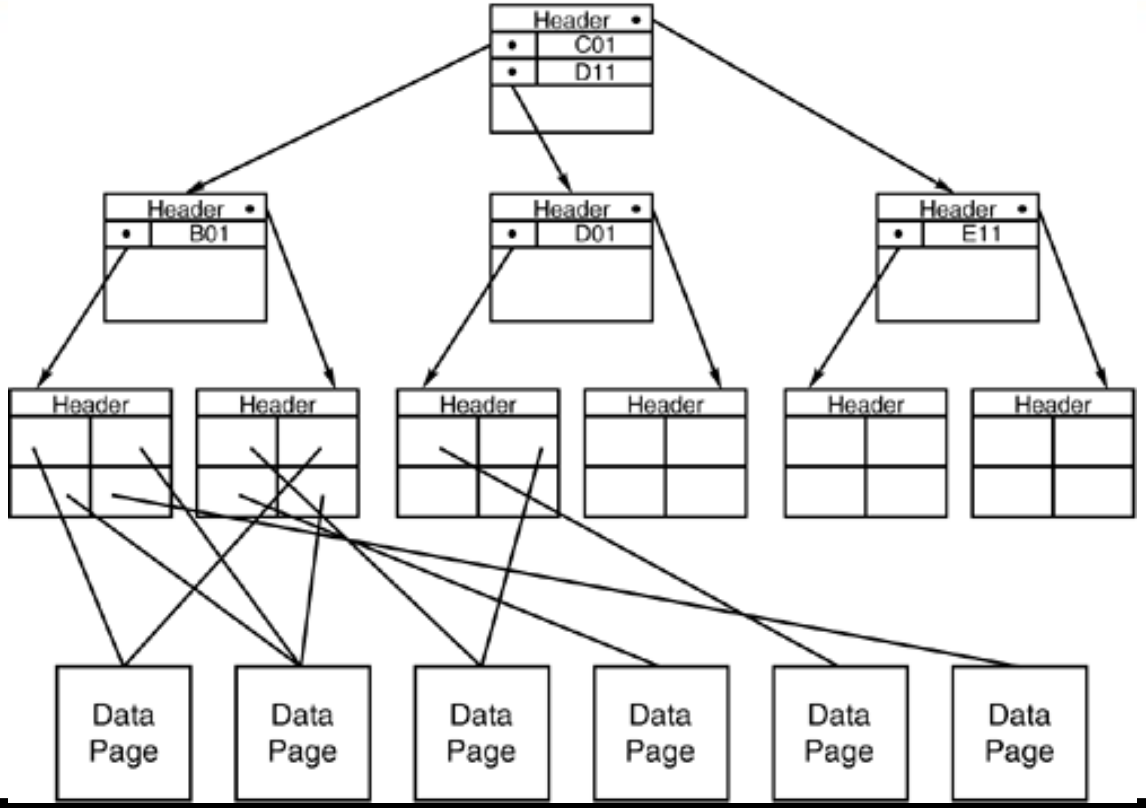
Architecture of indexes

- **Clustered index**:
  - It determines also the physical order of the data in a table.
- **Non-clustered index**:
  - The index specifies the logical order, but data are not stored in the same order

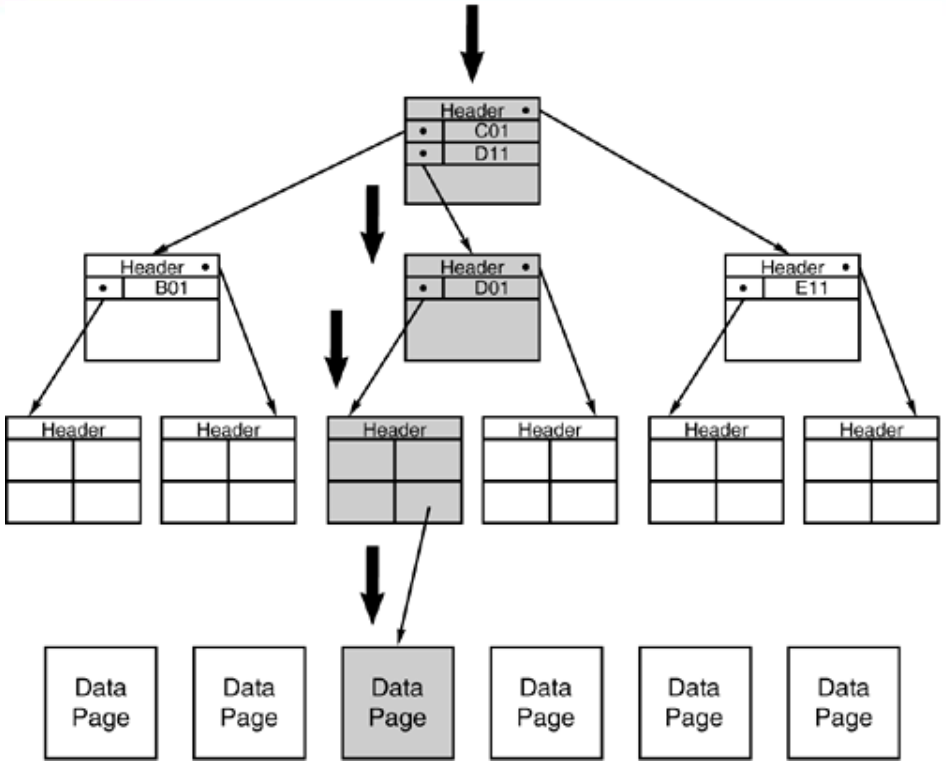
Clustered index architecture



Non-clustered index architecture



Example with equality predicate



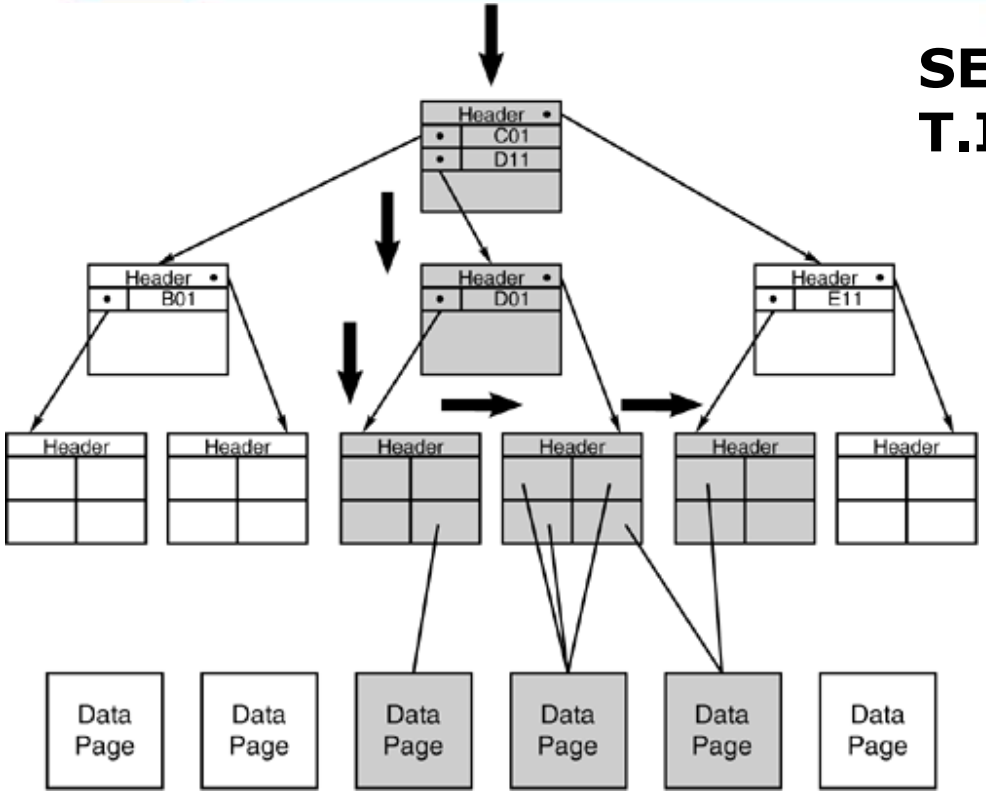
SELECT * FROM T
WHERE T.ID='D01'

Search D01:
# of accesses = tree depth

(here you will find the pointer to the whole tuple)

+ 1 access to the corresponding tuple

Example with interval predicate



SELECT * FROM T WHERE T.ID>='D01' and T.ID<='E11'

Search D01:
# of accesses = tree depth

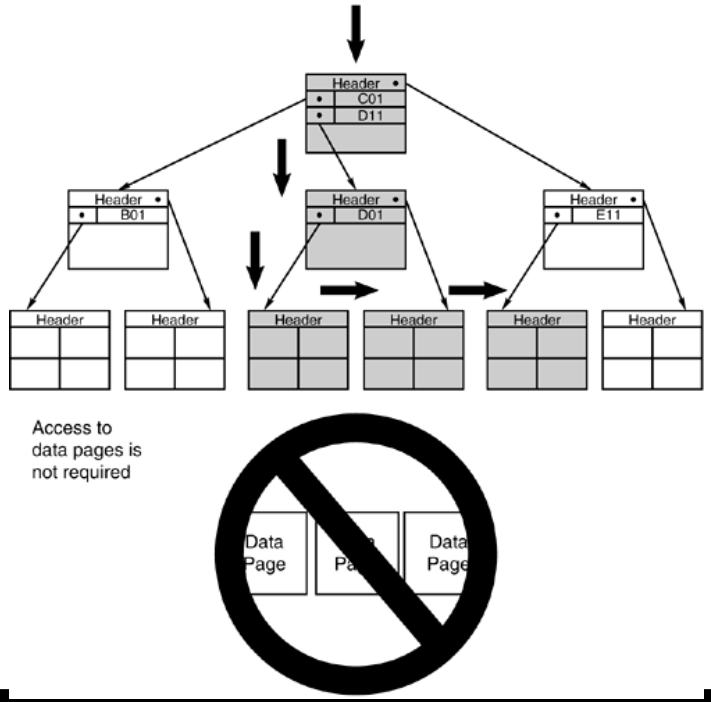
Navigate linked leaves until E11
→ Add #accesses to linked leaves

For each ID between D01 and E11 access the corresponding tuple (less accesses in case of clustered index)

Query examples – tree access only

SELECT T.ID FROM T WHERE T.ID>='D01' and T.ID<='E11'

We can access only
the B+ tree:



Similar example:
SELECT V.* FROM T, V
WHERE T.ID=V.ID // join on ID

Indexes in SQL

- **Syntax in SQL:**
  - `create [unique] index` ***IndexName*** `on` ***TableName*****(*****AttributeList*****)**
  - `drop index` ***IndexName***
- 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 most used for selections and joins
- Secondary structures are progressively added, checking that the system actually uses them, and, ideally, without excess

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 file organization.

(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**.

ACCESS TO DATA

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 an access program in 'object' or 'internal' format, which 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

Internal representation of queries

- A tree representation, similar to that of relational algebra:
  - Leaf nodes correspond to the physical data structures (**tables, indexes, files**).
  - Intermediate nodes represent physical data access **operations** that are supported by the access methods
    - Typical operations include sequential scans, orderings, indexed accesses and various methods for evaluating joins and aggregate queries, as well as materialization choices for intermediate results

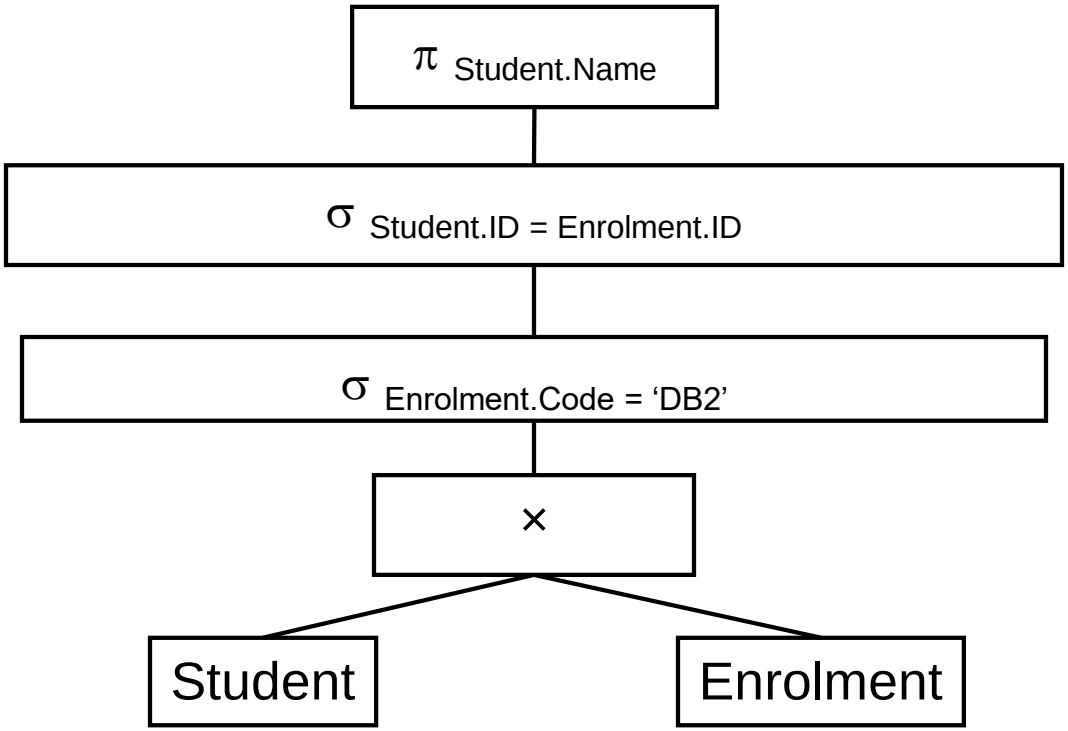
Example: SQL → Relational Algebra

- SQL statement

```
SELECT Student.Name
FROM Student, Enrolment
WHERE
  Student.ID = Enrolment.ID
AND
  Enrolment.Code = DB2'
```

- Relational Algebra
  - Take the product of Student and Enrolment
  - select tuples where the IDs are the same and the Code is DB2
  - project over Student.Name

Query Tree


π Student.Name
σ Student.ID = Enrolment.ID
σ Enrolment.Code = 'DB2'
×
Student
Enrolment

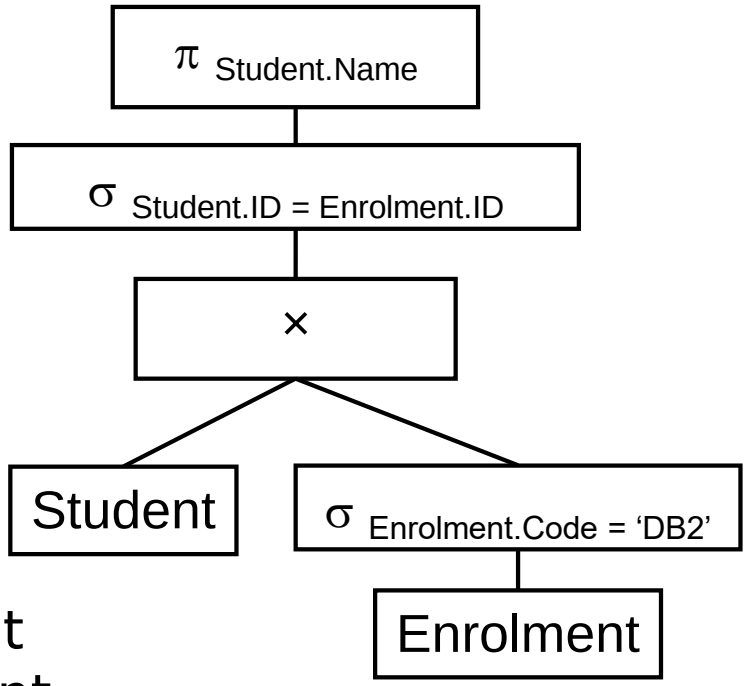
Optimisation

- There are often **many ways to express the same query**
- Some of these will be more efficient than others
- Need to find a good version

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

Optimisation Example

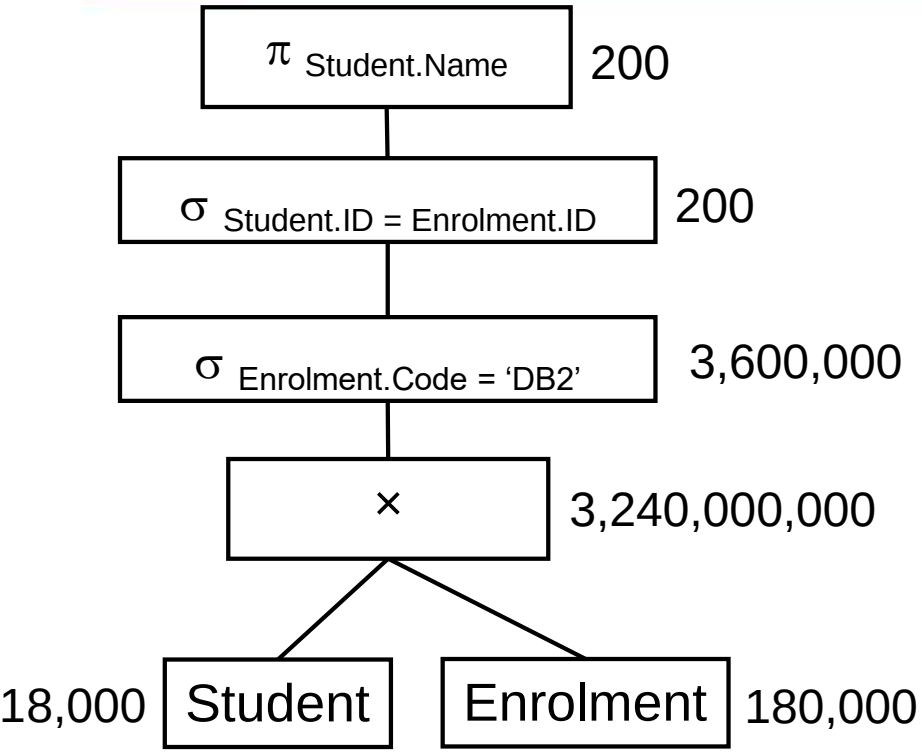
- In our query tree we have:
  - Take the product of Student and Enrolment
  - Then select those entries where the Enrolment.Code equals ‘DB2’
- This is equivalent to
  - Selecting those Enrolment entries with Code = ‘DB2’
  - Then taking the product of the result of the selection operator with Student

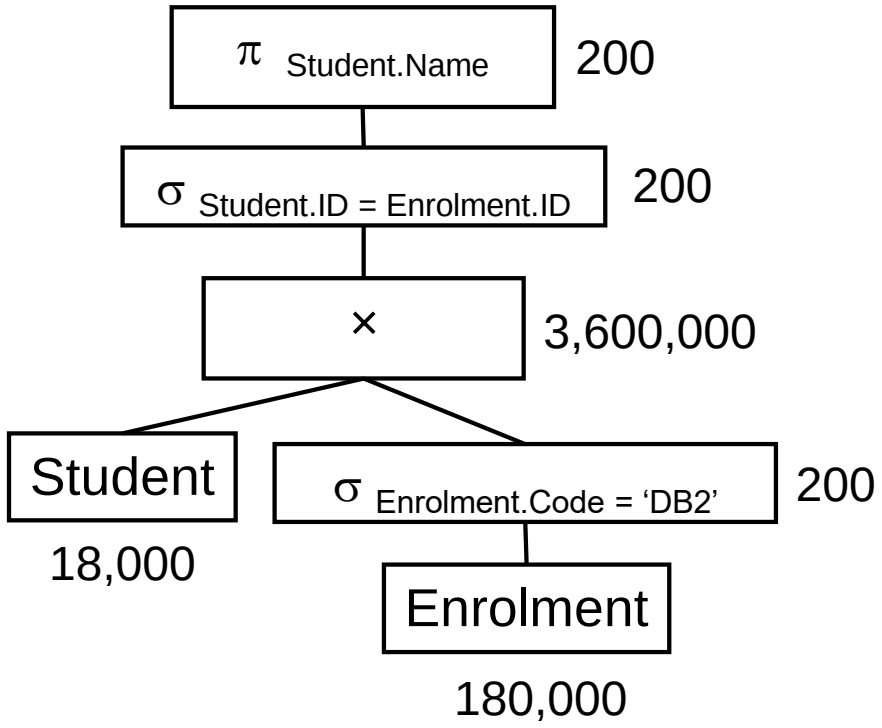


Optimisation Example

- To see the benefit of this, consider the following statistics
  - The university has 18,000 students
  - Each student is enrolled in at about 10 modules
  - Only 200 take DB2
- From these statistics we can compute the sizes of the relations produced by each operator in our query trees

Original and Optimized Query Tree





Relation profiles

- Profiles contain quantitative information about tables and are stored in the data dictionary:
  - 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: a measure of the "filtering power" of 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

- If no data on distributions are available, we will always assume homogeneous distributions!

Optimizations

Access method

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

Operations

- Selection
- Projection
- Sort
- Join
- …

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: $\boldsymbol{A_i = v}$)
  - Sort (ordering)
  - Insertions, deletions, and updates of the tuples currently accessed during the scan
- Primitives:

  Open, next, read, modify, insert, delete, close

Hash and indexes

- **Hashing:**
  - For *equality predicates*
  - As auxiliary data tables to speed up other operations

- **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

- If a table is only stored in a primary sequential structure then the only way to select/extract some tuples is a full scan
- A full scan obviously costs as many i/o operations as the size of the structure, expressed in blocks
  - **The cost is independent of the size of the "useful" data extracted to be included in the query result**
    - i.e., the access cost is independent of the selectivity of the access
  - If the structure is sequentially ordered, the search cost may be reduced
  - Also, searching a unique key typically takes less (in average)

Indexed access (lookup)

- Indexes are used when queries include predicate-based lookups:
  - simple predicates (of the type $A_i = v$ or $A_i > v$)
  - interval predicates (of the type $v_1 \leq A_i \leq v_2$)
- These predicates are said to be *supported* by indexes built on A_i
  - With **conjunctions** of supported predicates, the DBMS chooses the **most selective supported predicate** for the data access, and **evaluates the other predicates in main memory**
  - With **disjunctions** of predicates:
    - if any of the predicates is not supported, 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 (unordered)
  - Lookups are not supported (cost: a full scan)
- Hash and Tree structures
  - The lookup is supported only if A_i is the key attribute on which the structure is built
    - Otherwise, a full scan is required
  - The cost depends on
    - the storage type (primary/secondary)
    - the structure type (tree/hash)
    - the key type (unique/non-unique)

Cost of equality lookup on a primary Hash

- A lookup for $A_i = v$ needs to find the block(s) storing the tuples with the same hash for A_i as those with v
- The **hash function** is applied to v
  - the right bucket is therefore immediately identified
- The bucket contains the searched tuples
  - **If** the hash has no overflow chains, the **cost** is **1**
  - **Else**, the **cost** is **1 +** the **average** size of the **overflow** chain

Cost of equality lookup on a secondary Hash

- The lookup for $A_i = v$ works exactly in the same way, only the aim is to find the block(s) storing the **pointers to the tuples** with the same hash for A_i as those with v
- The **hash function** is applied to v
  - the right bucket is identified, and the corresponding block(s) are retrieved
- The bucket contains **the pointers** to the searched tuples
  - **All pointers** pointing to the tuples with the searched key value are to be followed so as to retrieve the tuples

Cost: **1** block for the initial bucket block **+ average overflow** blocks, if any **+ 1 block per pointer** to be followed

Cost of equality lookup on a primary B+

- A lookup for $A_i = v$ needs to find the block(s) storing all tuples indexed by the key value v
- The **root** is read first
  - then, a node per **intermediate** level is read, until…
- …the first **leaf** node is reached, that stores tuples with $A_i = v$
  - **If** the searched tuples are all stored in that leaf block, stop
  - **Else**, continue in the leaf blocks chain until v changes to v'

Cost: **1** block **per** intermediate **level +** as many **leaf blocks** as necessary to store all the tuples with $A_i = v$

If attribute A_i is a **unique** key, the cost is equal to the tree **depth**

Cost of equality lookup on a secondary B+

- A lookup for $A_i = v$ needs to find the block(s) storing all **the pointers that point to all** tuples with the key value v for A_i
- The **root** is read first
  - Then, a node per **intermediate** level is read, until…
- …the first **leaf** node is reached, that stores **pointers** $A_i = v$
  - **If** the **pointers** are all stored in that leaf block, stop
  - **Else**, continue in the leaf blocks chain until v changes to v'

Cost: **1** block **per** intermediate **level +** as many **leaf blocks** as necessary to store the pointers pointing to the tuples with $A_i = v$ **+ 1 block per** each **pointer** (to be followed to retrieve the pointed tuple!)

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

- 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 key attribute of the structure
  - The cost depends on
    - the storage type (primary/secondary)
    - the key type (unique/non-unique)

Cost of interval lookup on a primary B+

- We consider a lookup for $v_1 \leq A_i \leq v_2$ 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 store all the tuples in the interval

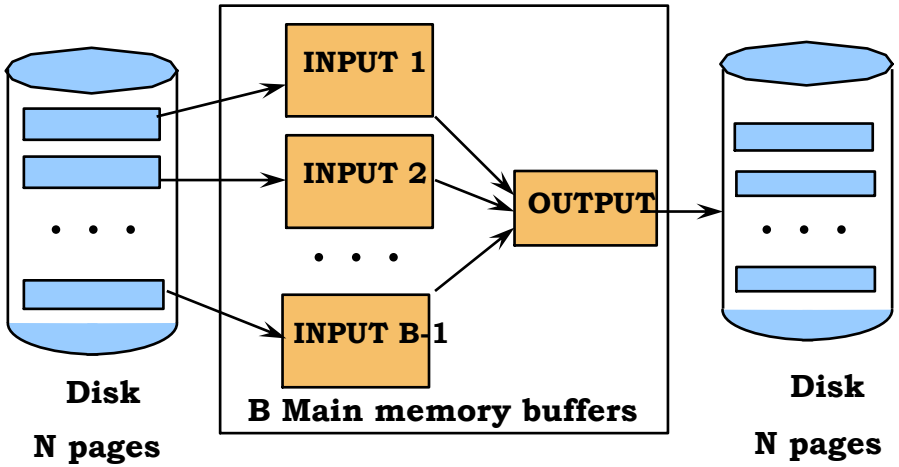
Cost of interval lookup on a secondary B+

- 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** 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 store all pointers to the tuples 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
  - Sort of **large files**, performed by merging smaller parts with already sorted parts



External sorting

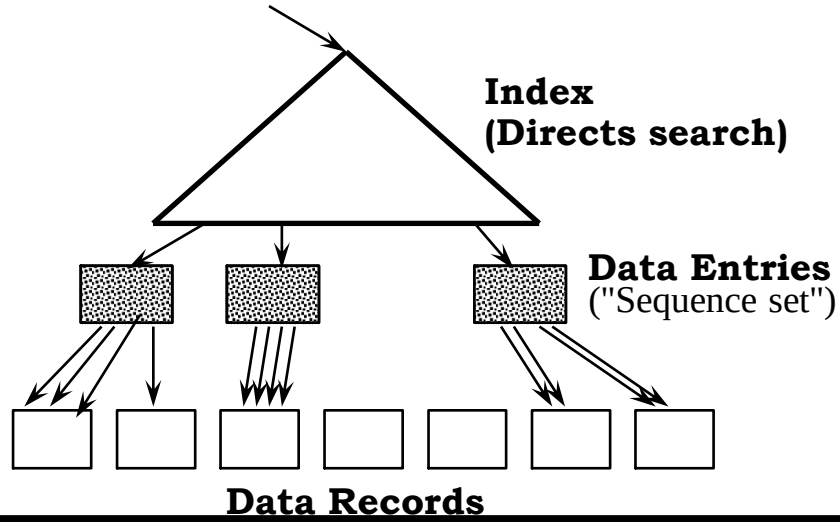
- Cost = 2N * (# of passes)

# of passes

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

Using B+ Trees for Sorting

- Scenario: Table to be sorted has B+ tree index on sorting column(s).
- Idea: Can retrieve records in order by traversing leaf pages.
- Cases to consider:
  - B+ tree is clustered
    - ***Good idea!***
    - *Better than external sorting*
  - B+ tree is not clustered
    - ***Could be a very bad idea!***



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, hashing, and ordering
- The “best” strategy is chosen based on several aspects…

Example

Students(SID, Name, GPA, Age, Advisor)
Professors(PID, Name, Dept)

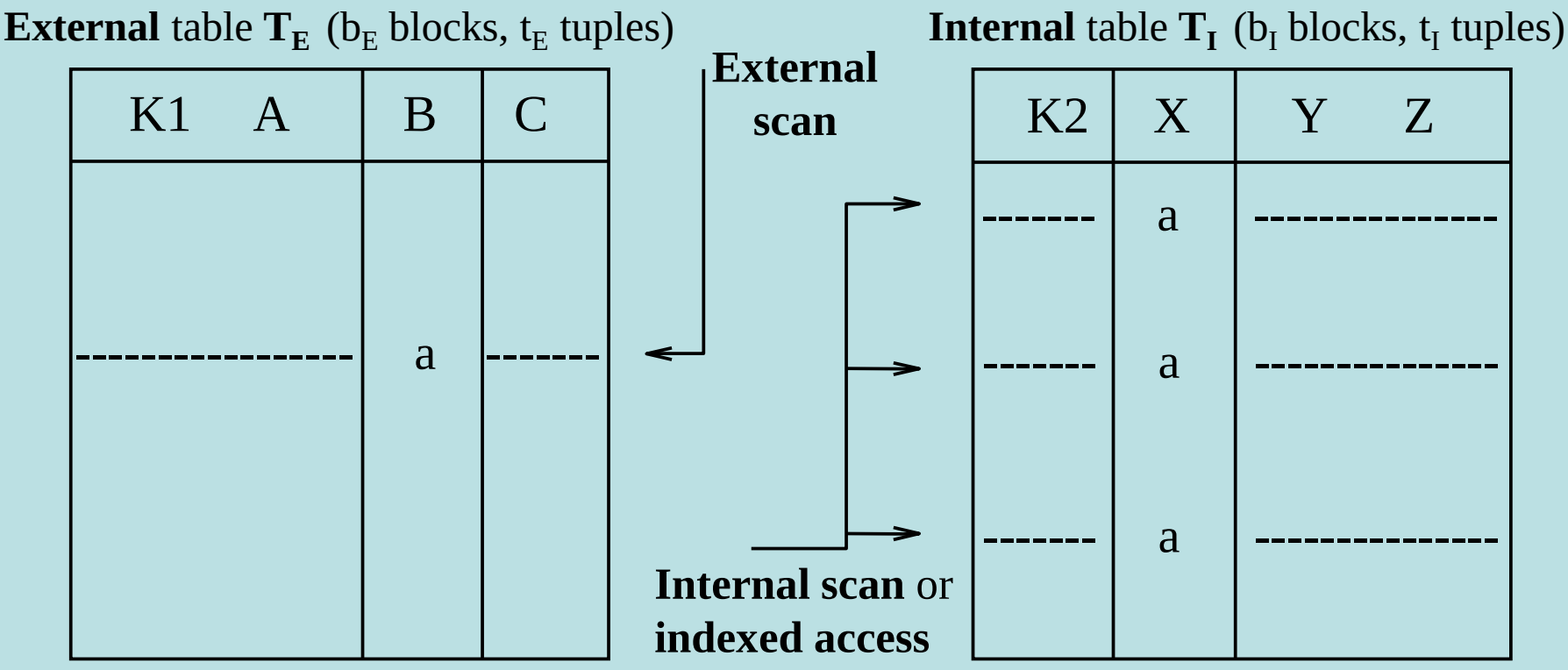
- Students:
  - Each tuple is 40 bytes long,
  - $p_{Students}$ = 100 tuples per page (Block factor)
  - $b_{Students}$ = 1000 pages (blocks) total.
- Professors:
  - Each tuple is 50 bytes long,
  - $p_{Professors}$ = 80 tuples per page (Block factor)
  - $b_{Professors}$ = 500 pages (blocks) total.

Equality Joins With One Join Column

```
SELECT *
FROM   Students, Professors
WHERE  Advisor=PID
```

- In algebra: R⋈S. Common! Must be carefully optimized.
  - R × S is large → R × S followed by a selection is inefficient
- Assume:
  - b_R blocks in R (Students), p_R tuples per page,
  - b_S blocks in S (Professors), p_S tuples per page.
- *Cost metric*: # of I/Os (we will ignore output costs)

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_E with all the tuples of all the blocks of T_I before moving to the next block of T_E
  - 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
    - $C = b_E + b_E \times b_I = b_E \times (1 + b_I) =\sim b_E \times b_I$
  - **IF** one of the tables is small enough to fit in the buffer, then it is chosen as internal table ($C = b_E + b_I$)

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
- In this case, the cost is that of a full scan of the **blocks** of the external table + the cost of lookup for each **tuple** of the external table onto the internal one, to extract the matching tuples
  - $C = b_E + \mathbf{t_E} \times cost\_of\_one\_indexed\_access\_to\_T_I$

Examples of Indexed Nested Loop

$$b_E + \mathbf{t_E} \times cost\_of\_one\_indexed\_access\_to\_T_I$$

- Case of Hash-index on *PID* of Professors (as inner):
  - Scan external table Students: $b_{Student}$=1000 I/Os
  - Students has $t_{Student} = p_{Student} * b_{Student}$ =100*1000 tuples
  - For each tuple of Students: 1.2 I/Os to get data entry in hash index + 1 I/O to get (the exactly one) matching Professors tuple

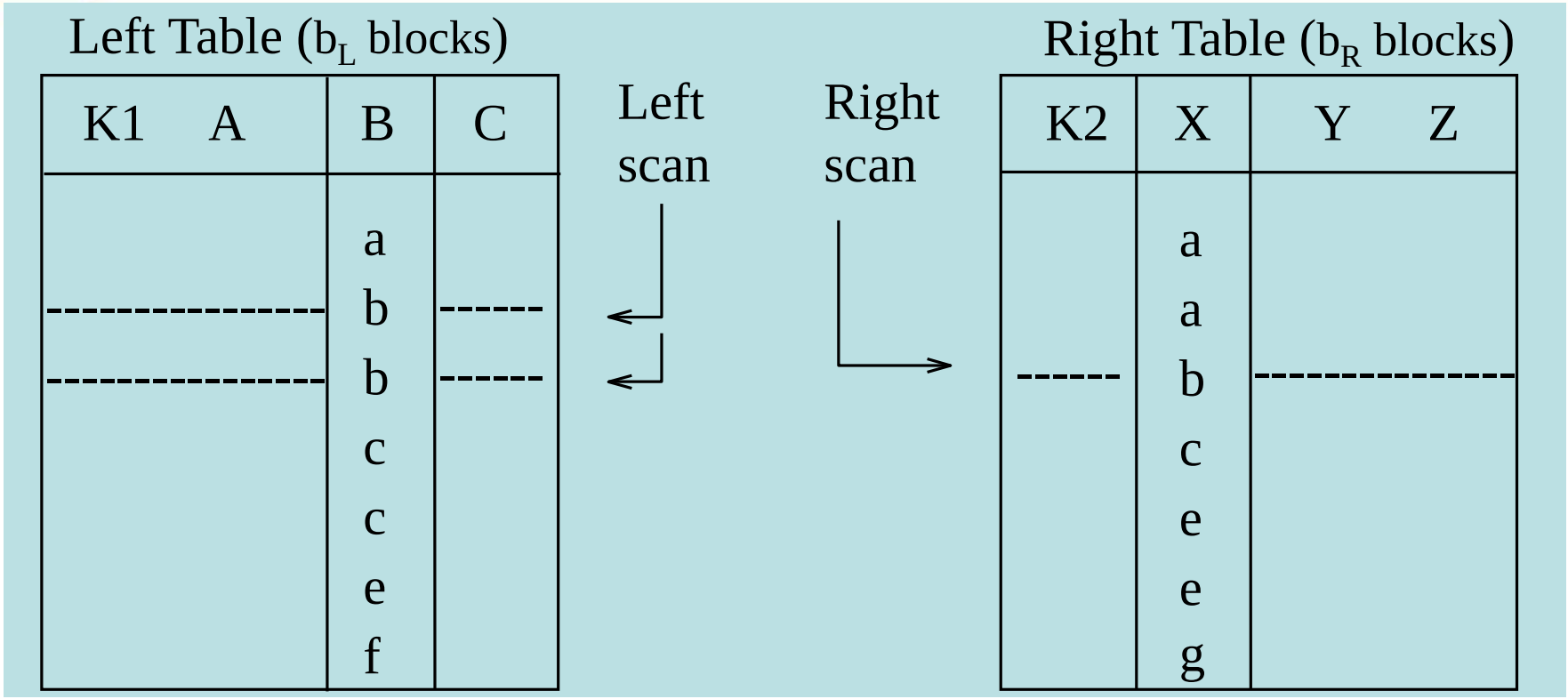
Total cost: 1000 + 220,000 I/Os

Examples of Indexed Nested Loop

$$b_E + \mathbf{t_E} \times cost\_of\_one\_indexed\_access\_to\_T_I$$

- Case of Hash-index on *SID* of Students (as inner):
  - Scan Professors: $b_{Professor}$=500 page I/Os
  - Professor has $t_{Professor}$ = 80*500 tuples
  - For each tuple of Professors: 1.2 I/Os to find index page with data entries, plus cost of retrieving matching Students tuples. Assuming uniform distribution, 2.5 students per professor (100,000 / 40,000), the cost of retrieving them is 1 or 2.5 I/Os depending on whether the index is clustered.

Merge-Scan join



Sort-Merge scan Join ($L \underset{i=j}{\bowtie} R$)

- Sort L and R on the join column, then scan them to do a "merge" (on join col.), and output result tuples.
  - Advance scan of L until current L-tuple >= current R tuple, then advance scan of R until current R-tuple >= current L tuple; do this until current L-tuple = current R-tuple.
  - Output all pairs of tuples <l, r> with the same value in R and S.
  - Then resume scanning L and R.
- L is scanned once; each R group is scanned once per matching L tuple. (Multiple scans of an R group are likely to find needed pages in buffer)

Cost of M-S joins

- 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
- A merge-scan join never needs to consider any tuple twice, as the tuples are compared in order
  - The cost is **linear** in the size of the tables
    - If the primary storage is sequentially ordered or B+, an ordered full scan of both tables is possible:

$$C = b_L + b_R$$

Example: Cost of M-S joins

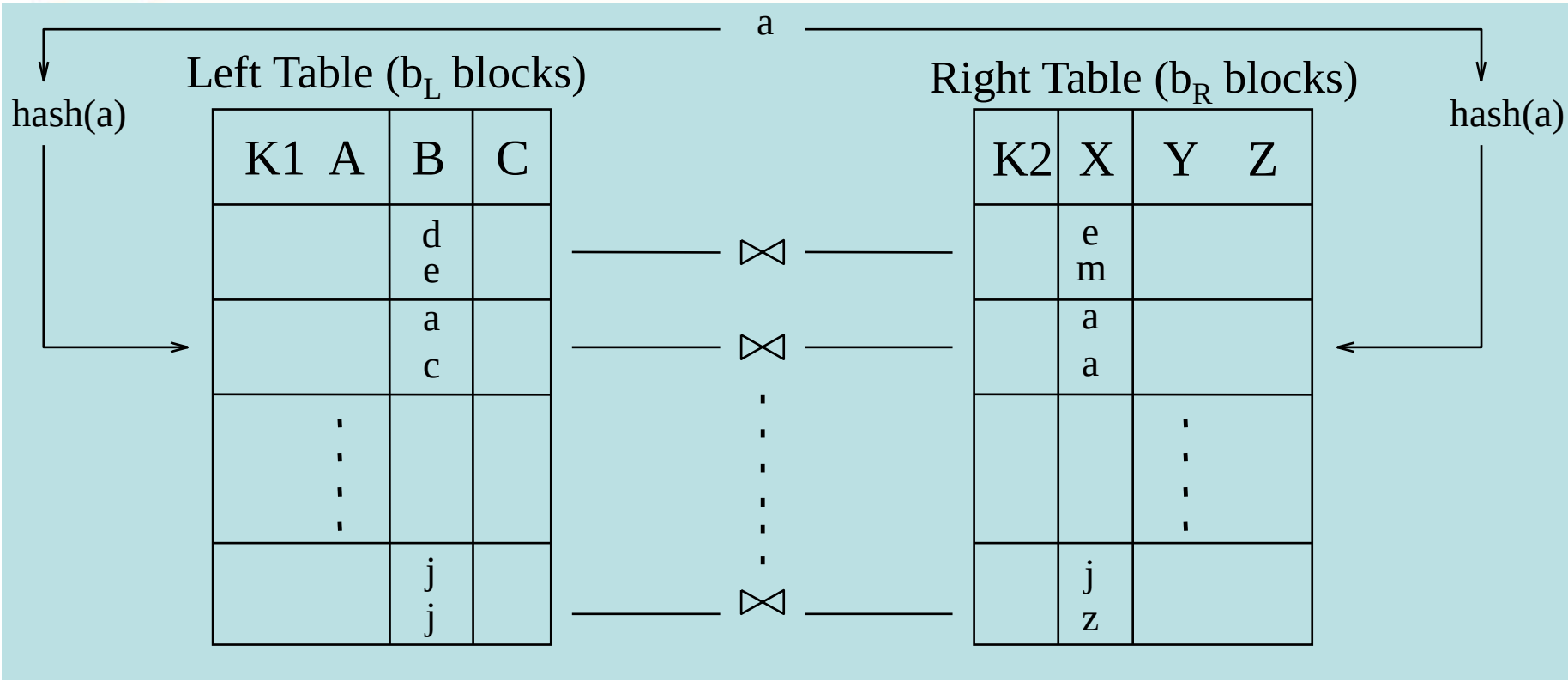
- Suppose that Students and Professors are sorted by professor-ID (e.g., in case of "sequentially-ordered" sequential structure or in case of primary B+ structure)
  - The cost is C = $b_{Students} + b_{Professors} = 1000 + 500 = 1500$ I/Os

[If tuples are not ordered by the key, a sorting algorithm (it has a cost) can be applied

Cost of the sorting algorithm is

$2 * b_{Students} * \#passes + 2 * b_{Professors} * \#passes$]

Hashed join


a
hash(a)
Left Table (b_L blocks)
K1 A
B
C
d e
a c
j j
Right Table (b_R blocks)
K2
X
Y Z
e m
a a
j z
hash(a)

Cost of Hashed joins

- This join is possible only if both tables are hashed according to the same key attribute, that is the same used in the join predicate
- The matching tuples can only be found in corresponding buckets, so there is no point in comparing tuples contained in non-corresponding buckets
  - The cost is **linear** in the size of the hashes
    - 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

QUERY OPTIMIZATION

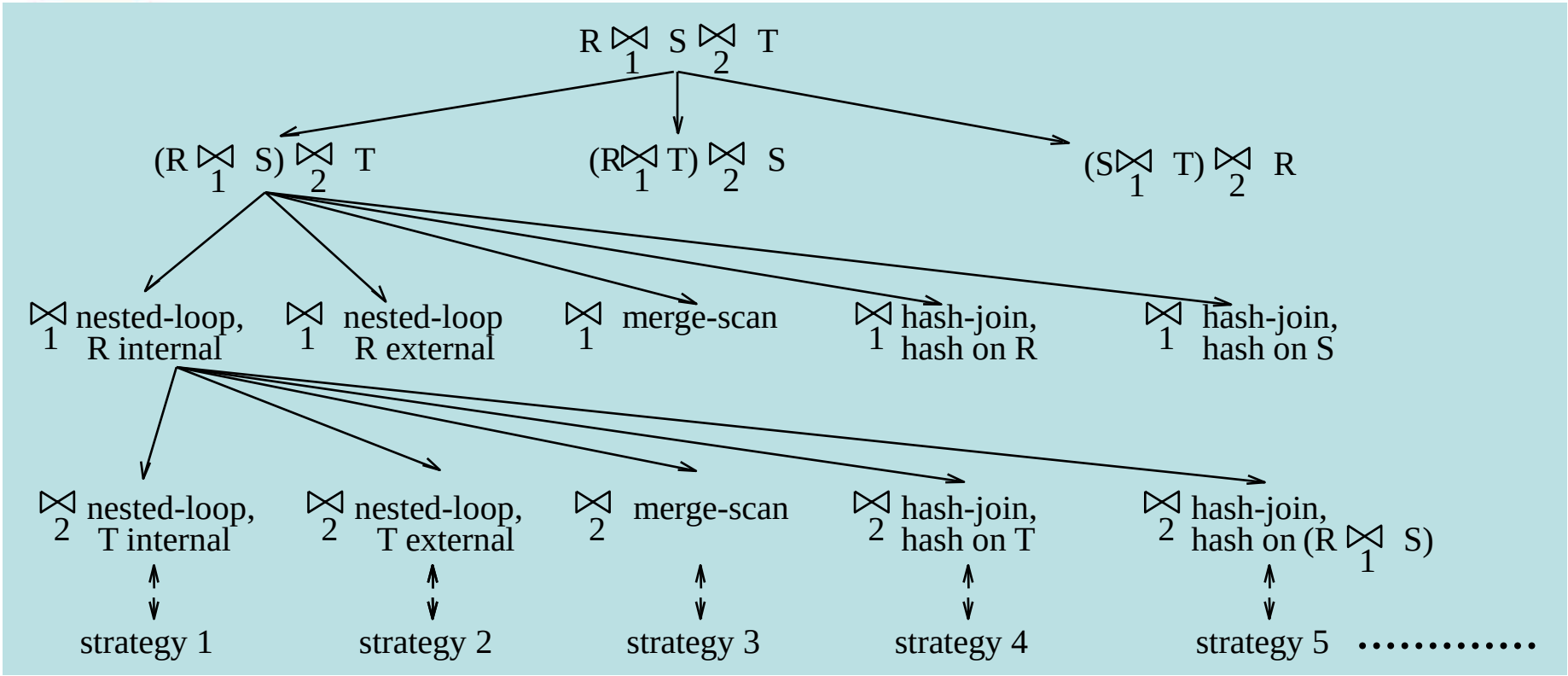
Cost-based optimization

- 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

Example of graphical query plan in MS SQL SMS