Advanced Databases

9

Physical data structures and query optimization

Study of “inside” DB technology: why?

- DBMSs provide “transparent” services:
  - So transparent that, so far, we could ignore many implementation details!
  - So far DBMSs have always been a “black box”
- So… why should we open the box?
  - Knowing how it works may help to use it better
  - Some services are provided separately

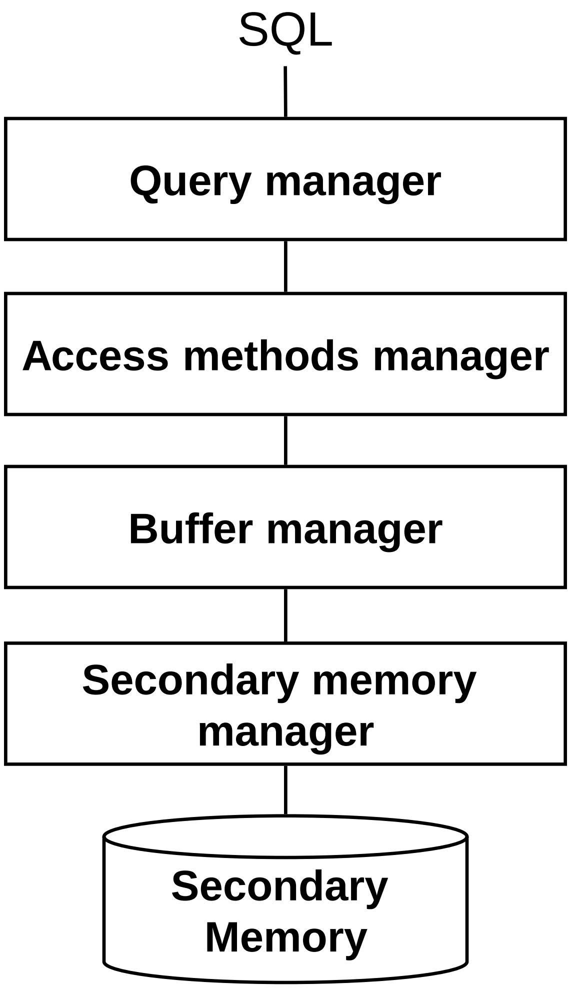
DataBase Management System — DBMS

A system (software product) capable of managing data collections which are:

- **large** ((much) larger than the central memory available on the computers that run the software)
- **persistent** (with a lifetime which is independent of single executions of the programs that access them)
- **shared** (in use by several applications at a time)

guaranteeing **reliability** (i.e. tolerance to hardware and software failures) and **privacy** (by disciplining and controlling all accesses).

Access and query manager



Technology of DBMSs - topics

- Query management ("optimization")
- Physical data structures and access structures
- Buffer and secondary memory management
- Reliability control
- Concurrency control
- Distributed architectures

Main and Secondary memory (1)

- Programs can only refer to data stored in main memory
- Databases must be stored (mainly) in 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 "main" and "secondary" terminology)

Main and Secondary memory (2)

- Secondary memory devices are organized in **blocks** of (usually) **fixed** length (order of magnitude: a few KBs)
- The only available operations for such devices are reading and writing one **page**, i.e. the byte stream corresponding to a block;
- For convenience and simplicity, we will use **block** and **page** as synonyms

Main and Secondary memory (3)

- Secondary memory access:
  - seek time (10-50ms) - *head positioning*
  - latency time (5-10ms) - *disc rotation*
  - transfer time (1-2ms) - *data transfer*

  as an average, hardly less than 10 ms
- 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: to create and delete files and for reading and writing 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.

DBMS and file system (2)

- The DBMS manages the blocks of allocated files as if they were a single large space in secondary memory.
- It builds in such space the physical structures with which tables are implemented.
- 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 in more than one file.

Blocks and records

- Blocks (the "physical" components of a file) and records (the “logical" components) generally have different size:
  - The size of a block depends on the file system
  - The size of a record depends on the needs of applications and is normally variable within a file

Block Factor

- The number of records within a block
  - S_R: Size of a record (assumed constant in the file for simpicity: "fixed length record")
  - S_B: Size of a block
  - if $S_B > S_{R,}$ there may be many records in each block:

$$\lfloor S_B / S_R \rfloor$$

- The rest of the space can be
  - used ("spanned" records (or "hung-up" records))
  - non used ("unspanned" records)

Physical access structures

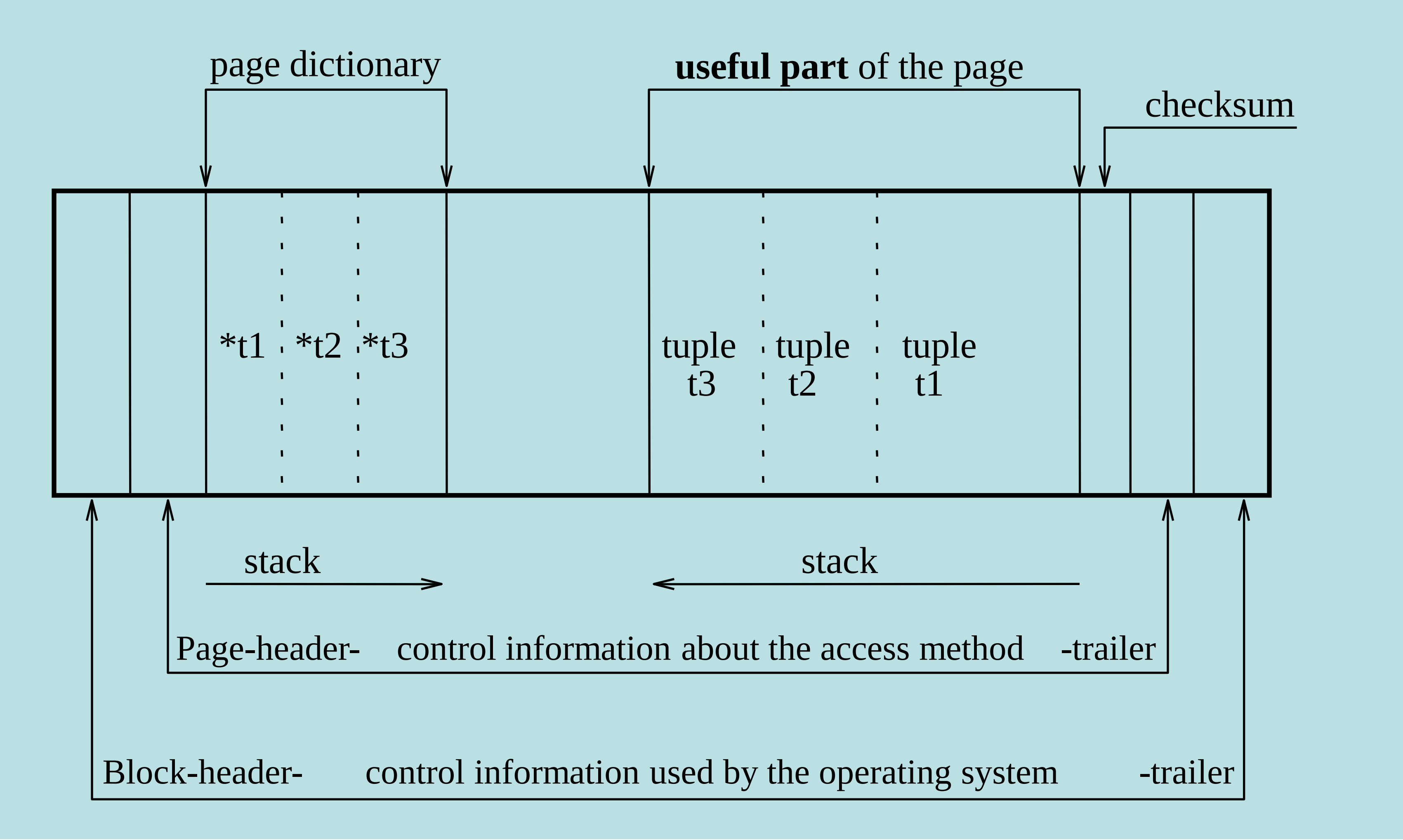
- Used for the efficient storage and manipulation of data within the DBMS
- Encoded as *access methods*, that is, software modules providing data access and manipulation primitives for each physical access structure
- Each DBMS has a distinctive and limited set of access methods
- We will consider three types of data structures:
  - Sequential
  - Hash-based
  - Tree-based (or index-based)

Organization of tuples within pages

- 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 item of useful elementary data contained in the page
  - A ***useful part***, which contains the data. In general, the page dictionary and the useful data grow as opposing stacks
  - A ***checksum***, to verify that the information in it is valid

- Tree structures have a different page organization

Organization of tuples within pages



Page manager primitives

- ***Insertion** and **update** of a tuple*
  - may require a reorganization of the page if there is sufficient space to manage the extra bytes introduced
- ***Deletion** of a tuple*
  - often carried out by marking the tuple as 'invalid'
- ***Access** to a **field** of a particular tuple*
  - identified according to the offset and to the length of the field itself, after identifying the tuple by means of its key or its offset

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 sequence depends on the value assumed in each tuple by a field that controls the ordering, known as the *key field*

“Entry-sequenced” sequential structure

- Optimal for carrying out sequential reading and writing operations
- Optimal for space occupancy, as it uses all the blocks available for files and all the spaces within the blocks
- **Non** optimal with respect to
  - searching specific data units or
  - updates that require more space

"Array" sequential structure

- Possible only when the tuples are of fixed length
- Made of n of adjacent blocks, each block with m slots available for tuples
- Each tuple has a numerical index i and is placed in the i-th position of the array

"Sequentially-ordered" sequential structure

- Each tuple has a position based on the value of the key field
- Historically, such structures were used on sequential devices (tapes). This had fallen out of use, but for data streams and system logs
- The main problems are insertions or updates which increase the physical space - they require reordering techniques for the tuples already present:
- Options to avoid global reorderings:
  - 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*

Hash-based access structures

- Ensure an efficient *associative* access to data, based on the value of a *key* field
- A hash-based structure has *B* blocks (often adjacent)
- A hash algorithm is applied to the key field and returns a value between zero and B-1. This value is interpreted as the position of the block in the file, and used both for reading and writing the block
- This is the most efficient technique for queries with equality predicates, but it is rather inefficient for queries with interval predicates

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 zero and *B* - 1.
- Optimal performance if the file is larger than necessary. Let:
  - *T* be the number of tuples expected for the file,
  - *F be* the average number of tuples stored in each page;

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

Collisions

- Collisions occur when the same block number is associated to too many tuples. They are critical when the maximum number of tuples per block is exceeded
- 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/(*F* x B) and of the average number *F* of tuples per page:

T/(FxB)	**1**	**2**	**3**	**5**	**10**	***F***
.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 records
- hash table with 50 positions:
  - 1 collision of 4 values
  - 2 collisions of 3 values
  - 5 collisions of 2 values

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 access based on equality for values of the key
- Collisions (overflow) can be managed by using the next available block or with linked blocks 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

- The most frequently used in relational DBMSs
  - SQL indexes are implemented in this way
- Gives associative access based on the value of a *key*
  - no constraints on the physical location of the tuples
- Note: the primary key of the relational model and the keys for hash-based and tree structures are different concepts

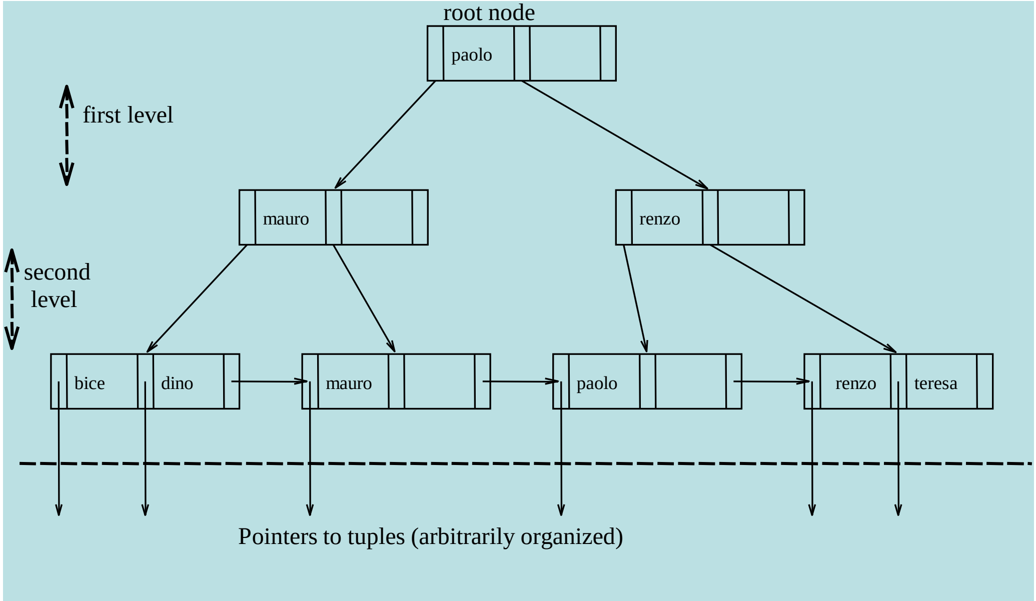
Index file

- Index: an auxiliary structure for the efficient access to the records of a file based upon the values of a given field or record of fields – called the index key.
- The index concept: index of a book, seen as a list of (term; page list) pairs, alphabetically ordered at the end of a book.
- The index key is not a key!

Types of indexes

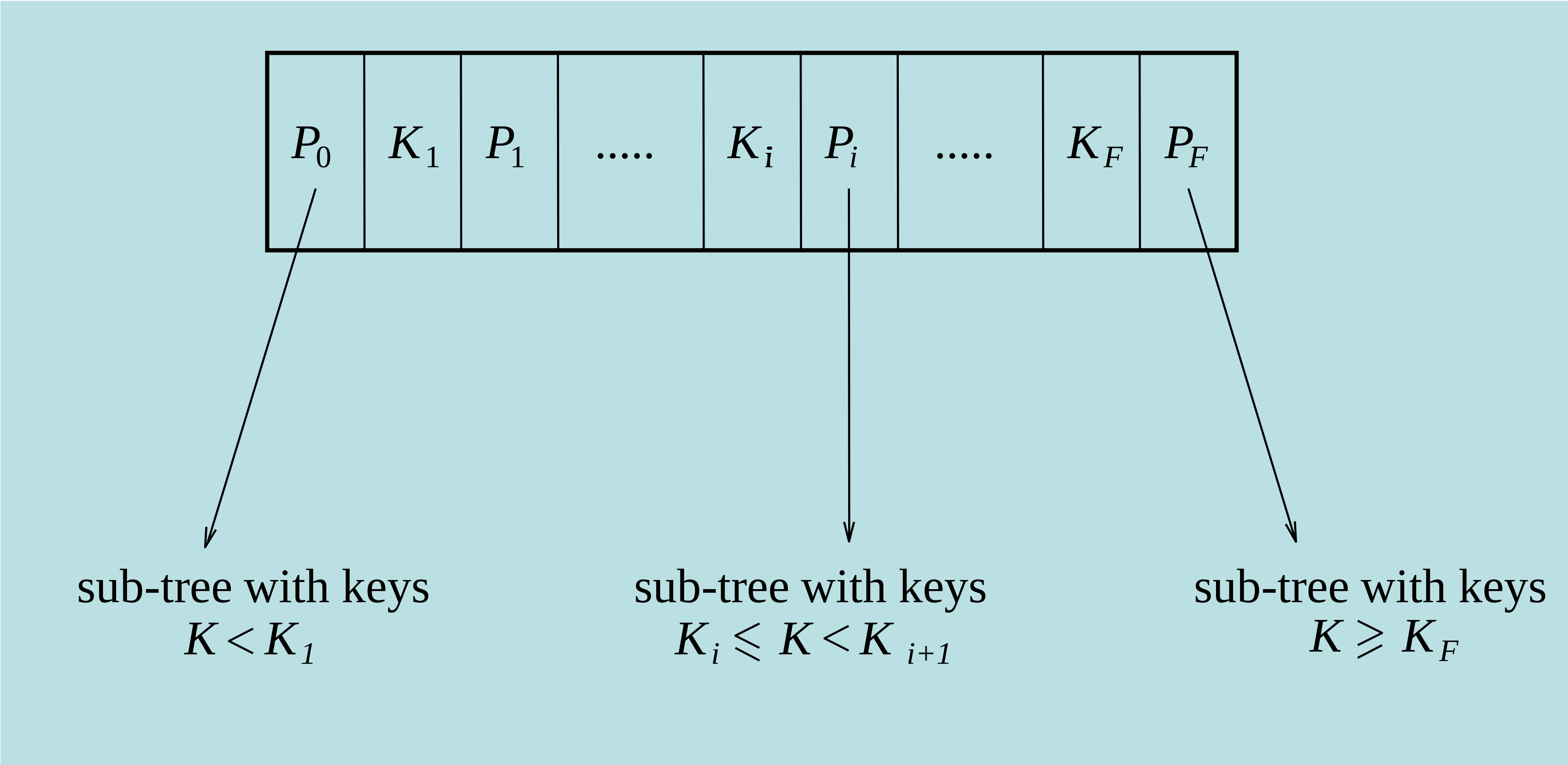
- Primary index:
  - Based upon the primary key
- Secondary index
  - Based upon other attributes (including secondary keys)
- Clustered index
  - One such that the records of the physical file are physically ordered according to the index key
- Dense index:
  - One having an index entry for every record of the file
- Sparse index:
  - Having less index entries than the number of records of the file

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.

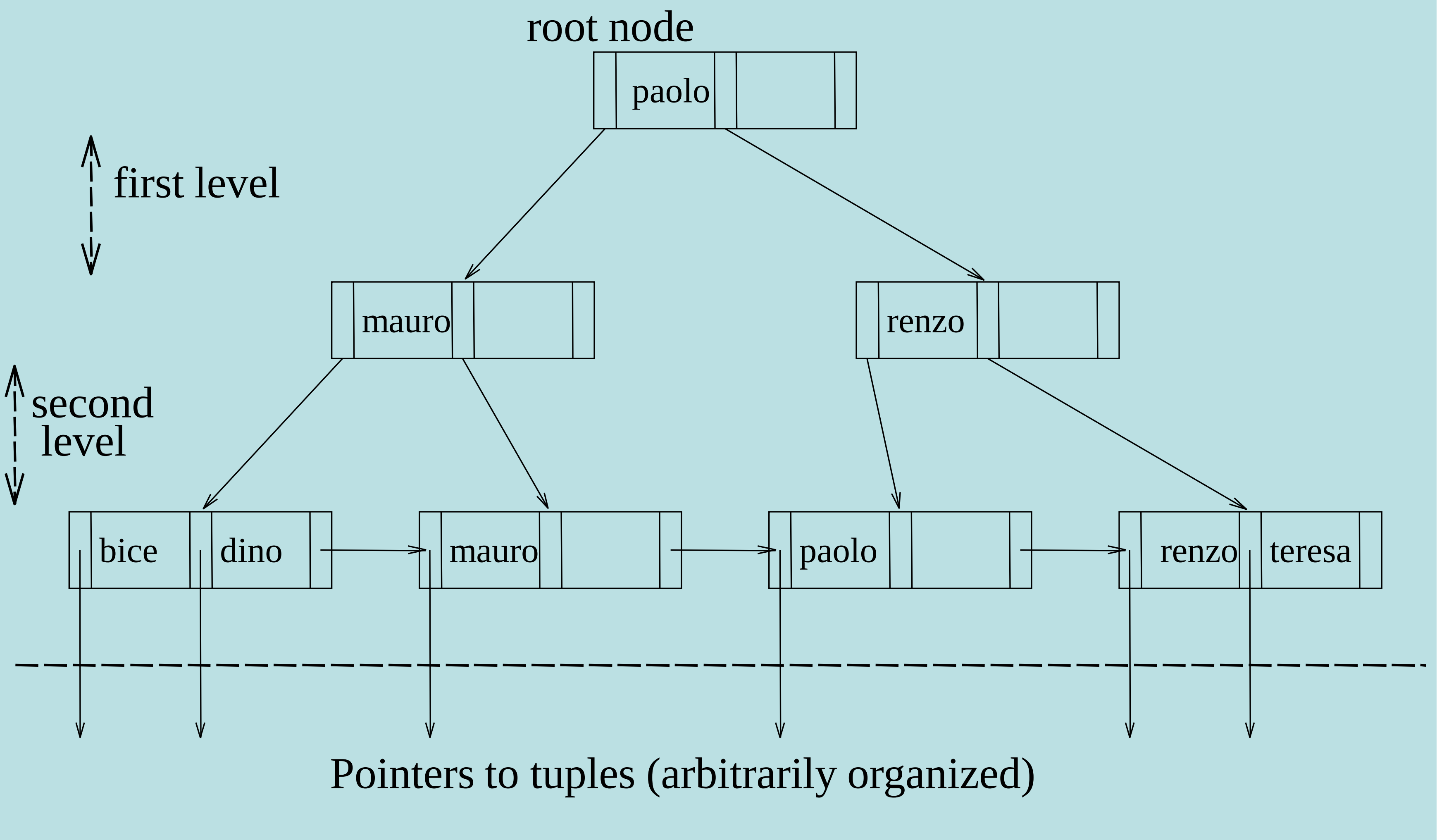
Structure of the tree nodes



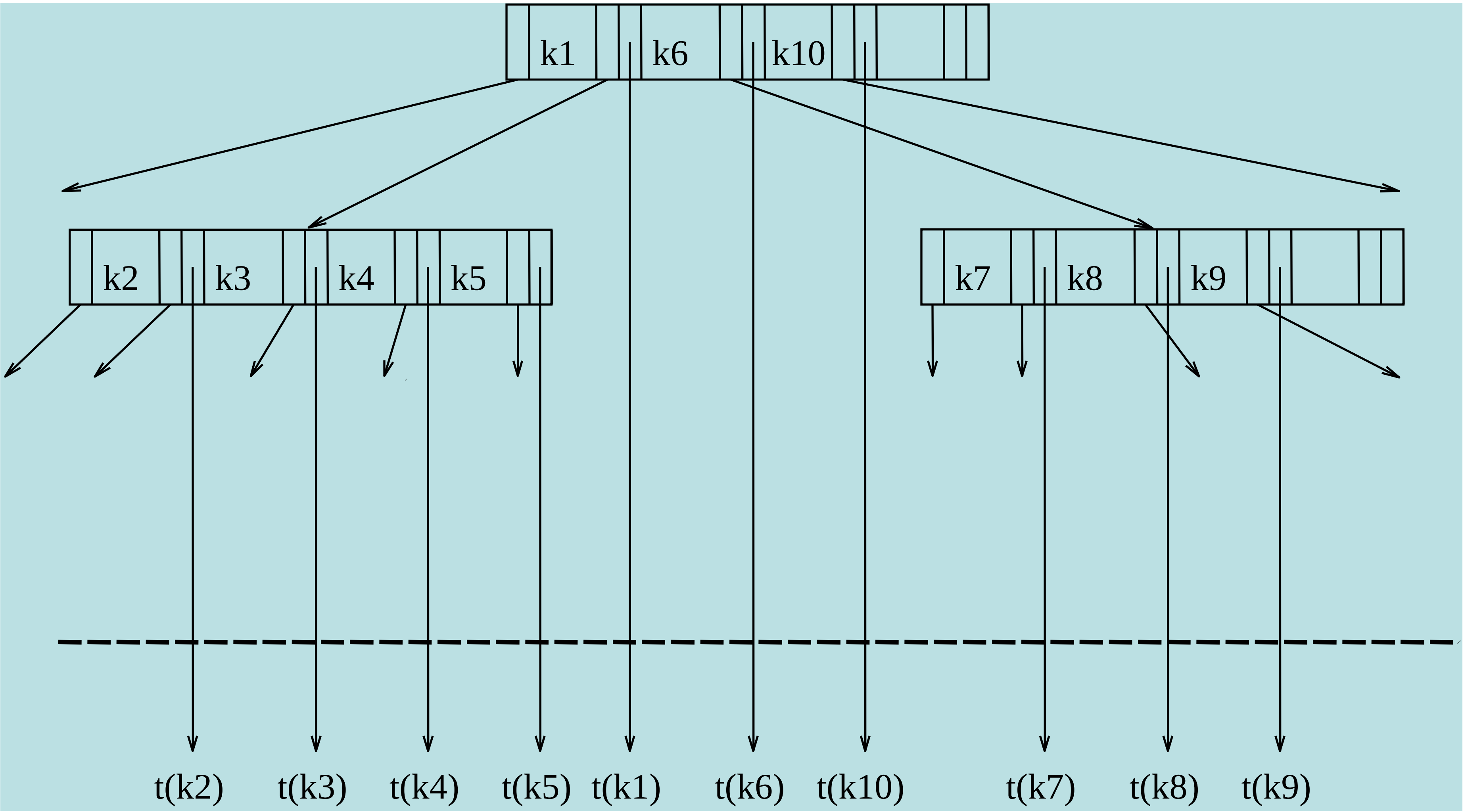
B and B+ trees

- **B+ trees**
  - The leaf nodes are linked in a chain in the order imposed by the key.
  - Supports interval queries efficiently
  - The most used by relational DBMSs
- **B trees**
  - No sequential connection for leaf nodes
  - Intermediate nodes use two pointers for each key value K_i
    - one points directly to the block that contains the tuple corresponding to K_i
    - the other points to a sub-tree with keys greater than K_i and less than K_{i+1}

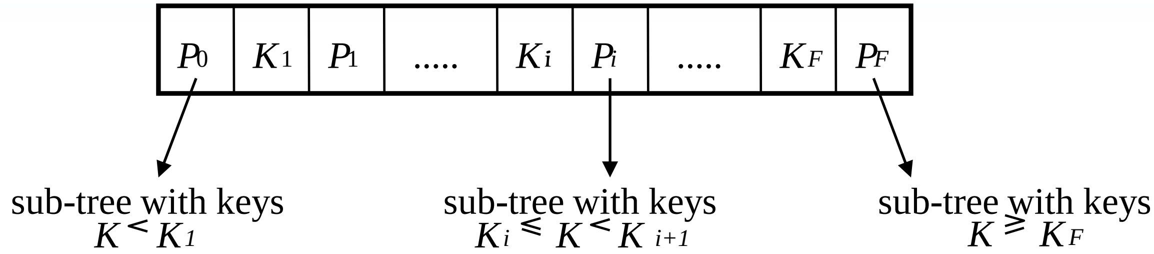
An example of B+ tree



An example of B tree



Search technique

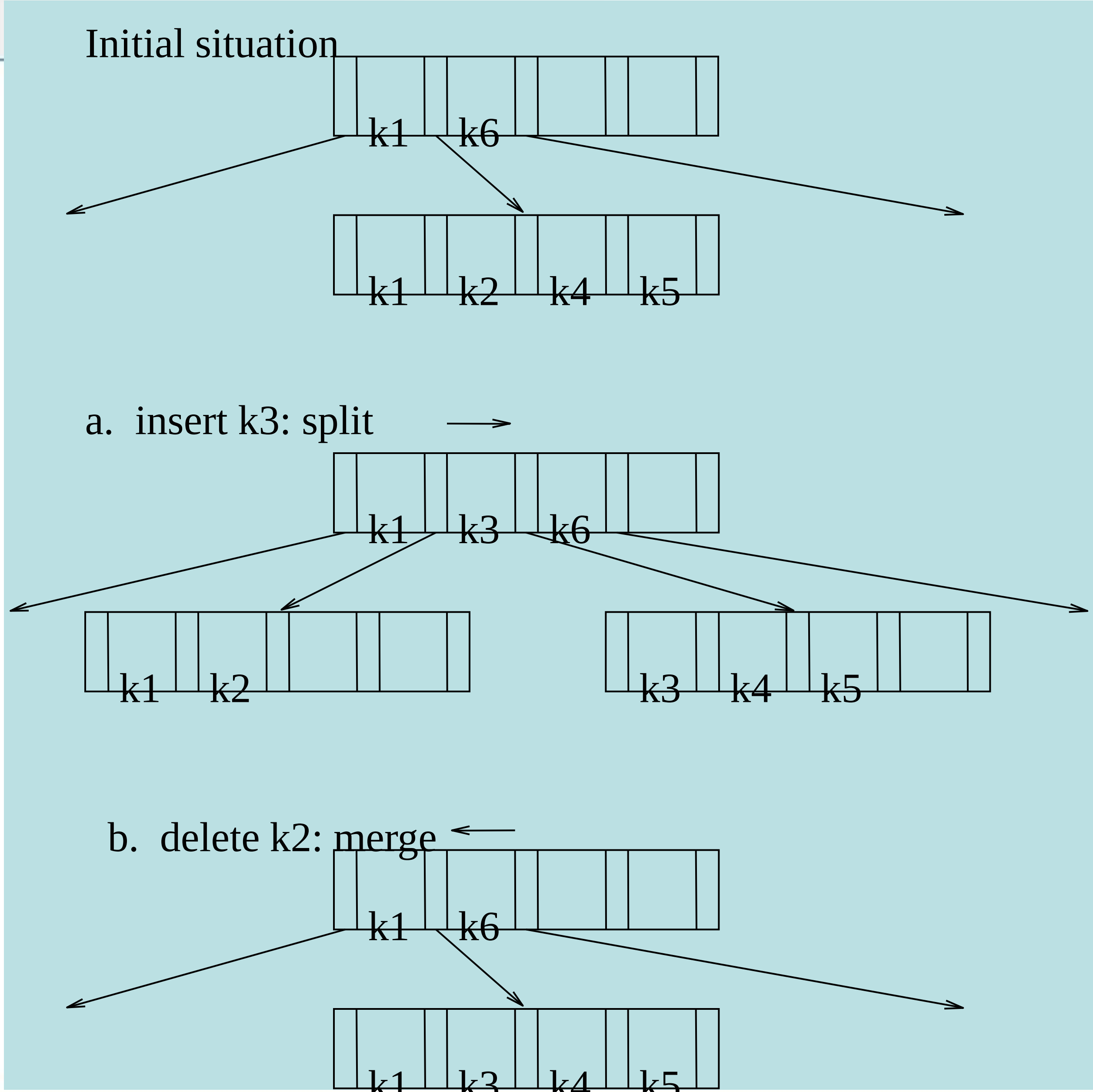


- Looking for a tuple with key value **V**, at each intermediate node:
  - **if $V < K_1$ follow P_0**
  - **if $V \geq K_F$ follow P_F**
  - **otherwise, follow P_j such that $K_j \leq V < K_{j+1}$**
- The leaf nodes can be organized in two ways:
  - In *key-sequenced* trees tuples are contained in the leaves
  - In *indirect trees* leaf nodes contain pointers to the tuples, allocated with any other 'primary' mechanism (entry-sequenced, hash, key-sequenced, ...)

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 superior node and thus could recursively cause another split
- MERGE: required when two “close” nodes have entries that could be condensed into a single node. Done in order to keep a high node filling and minimal paths from the root to the leaves.
  - Causes a decrease of pointers in the superior node and thus could recursively cause another merge

Split and merge



Index usage

- **Syntax in SQL:**
  - `create [unique] index` ***IndexName*** `on` ***TableName*(*AttributeList*)**
  - `drop index` ***IndexName***
- Every table should have:
  - A ***primary index***, with key-sequenced structure, normally unique, on the primary key
  - Several ***secondary indexes***, both unique and not unique, on the attributes most used for selections and joins
- They are progressively added, checking that the system actually uses them, and without excess

Query optimization

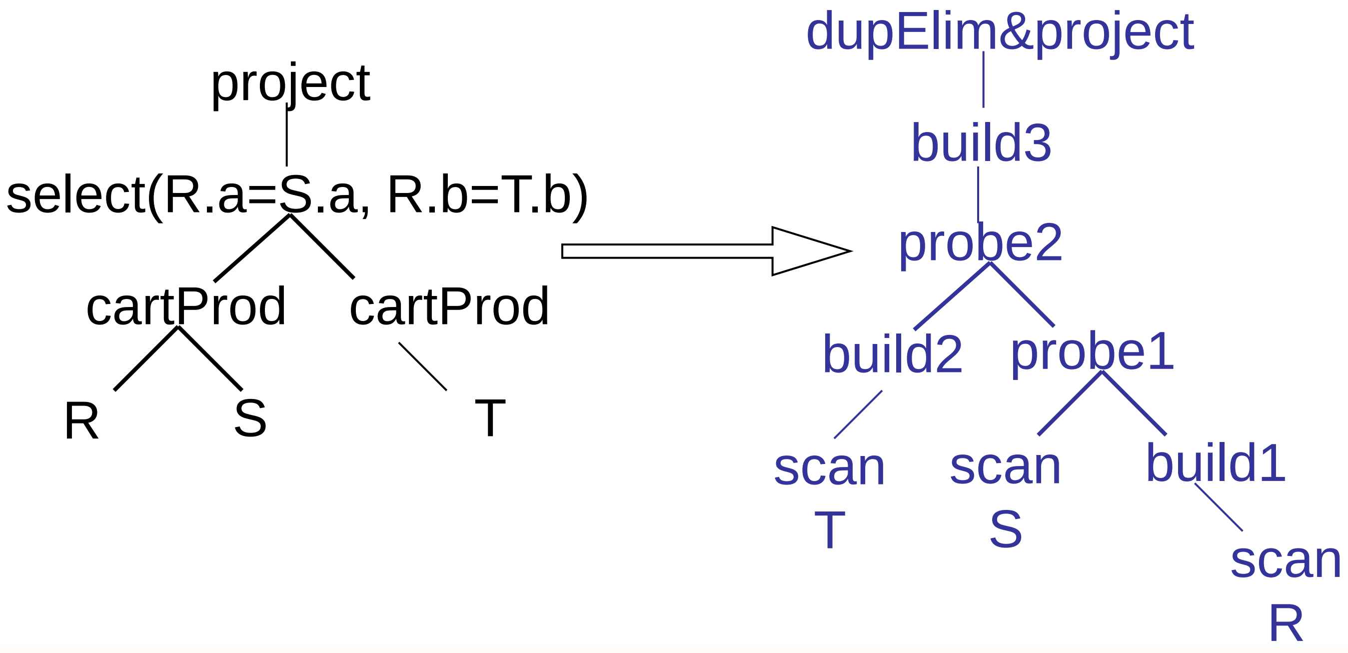
- Optimizer: an important module in the architecture of a DBMS
- It receives a query written in SQL and produces an access program in ‘object’ or ‘internal’ format, which uses the data access methods.
- Steps:
  - Lexical, syntactic and semantic analysis
  - Translation into an internal representation
  - Algebraic optimization
  - Cost-based optimization
  - 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

Query optimization input-output

- Input: query in SQL
  SELECT A FROM R,S,T WHERE R.A=S.A AND R.B=T.B
- Output: execution plan



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

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
  - the minimum and maximum 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

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 modifications of the tuples currently accessed during the scan
- Primitives:

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

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, which can not me transferred to main memory, performed by merging smaller parts with already sorted parts

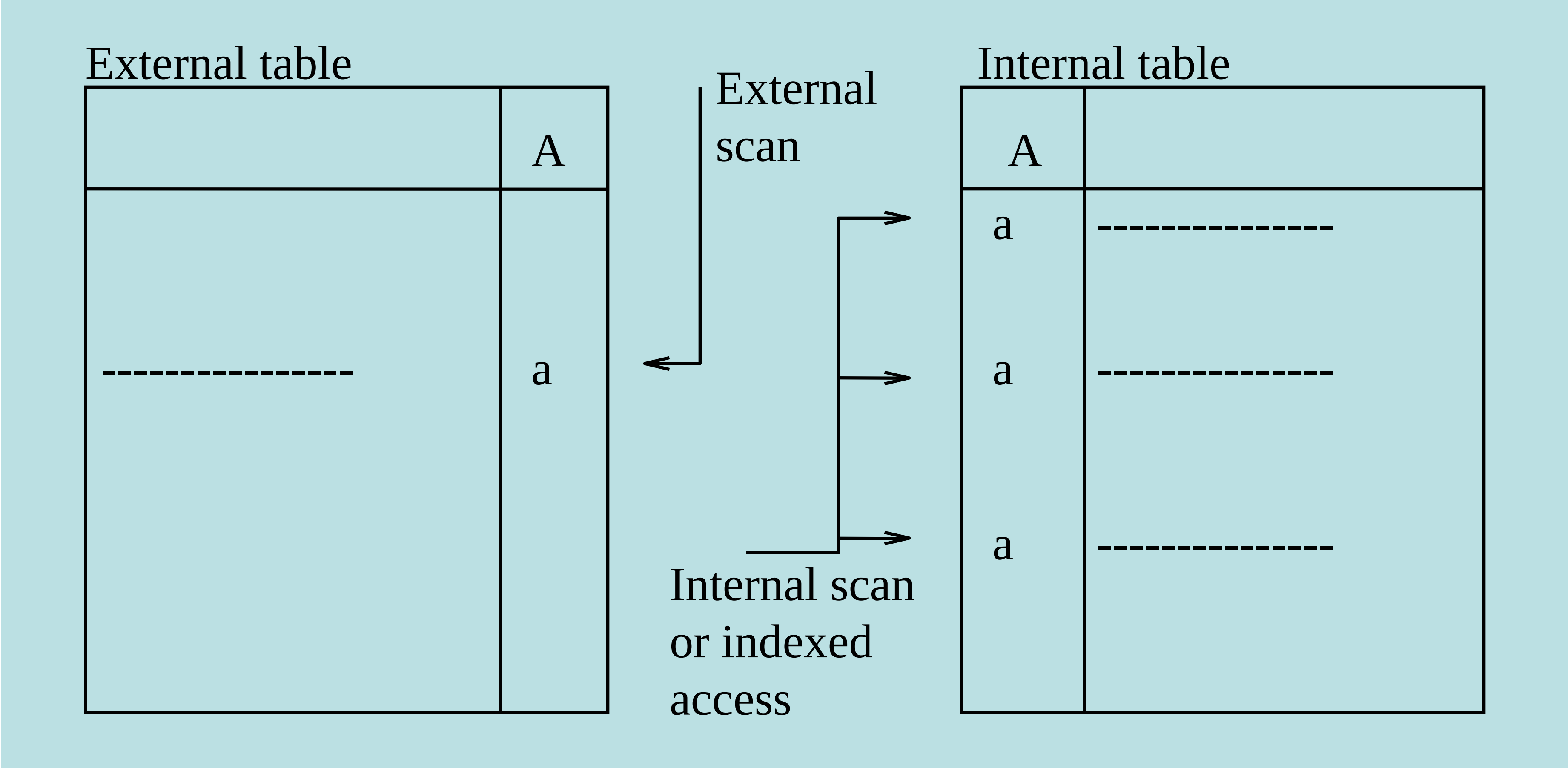
Indexed access

- Indexes are used when queries include:
  - simple predicates (of the type $\boldsymbol{A_i = v}$)
  - interval predicates (of the type $\boldsymbol{v_1} \le \boldsymbol{A_i} \le \boldsymbol{v_2}$)
- We say that such predicates are *supported* by the index
  - With conjunctions of supported predicates, the DBMS chooses the most selective supported predicate for the primary access, and evaluates the other predicates in main memory
- With disjunctions predicates:
  - if any of them is not supported a scan is needed;
  - if all are supported, indexes can be used only with duplicate elimination

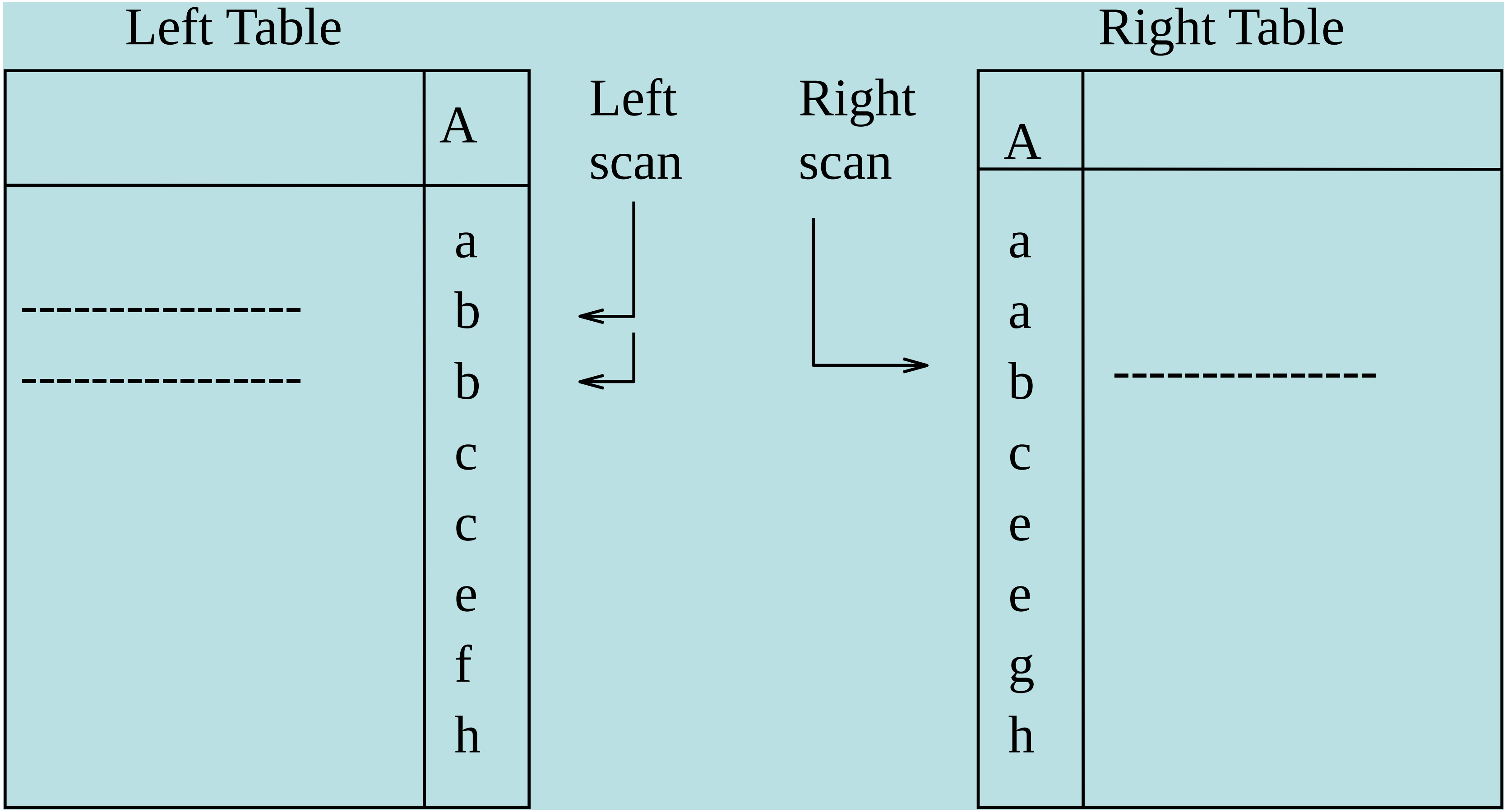
Join Methods

- Joins are the most frequent (and costly) operations in DBMSs
- There are several methods for join evaluation, among which:
  - *nested-loop*, *merge-scan* and *hashed*.
- These three methods are based on scanning, hashing, and ordering.

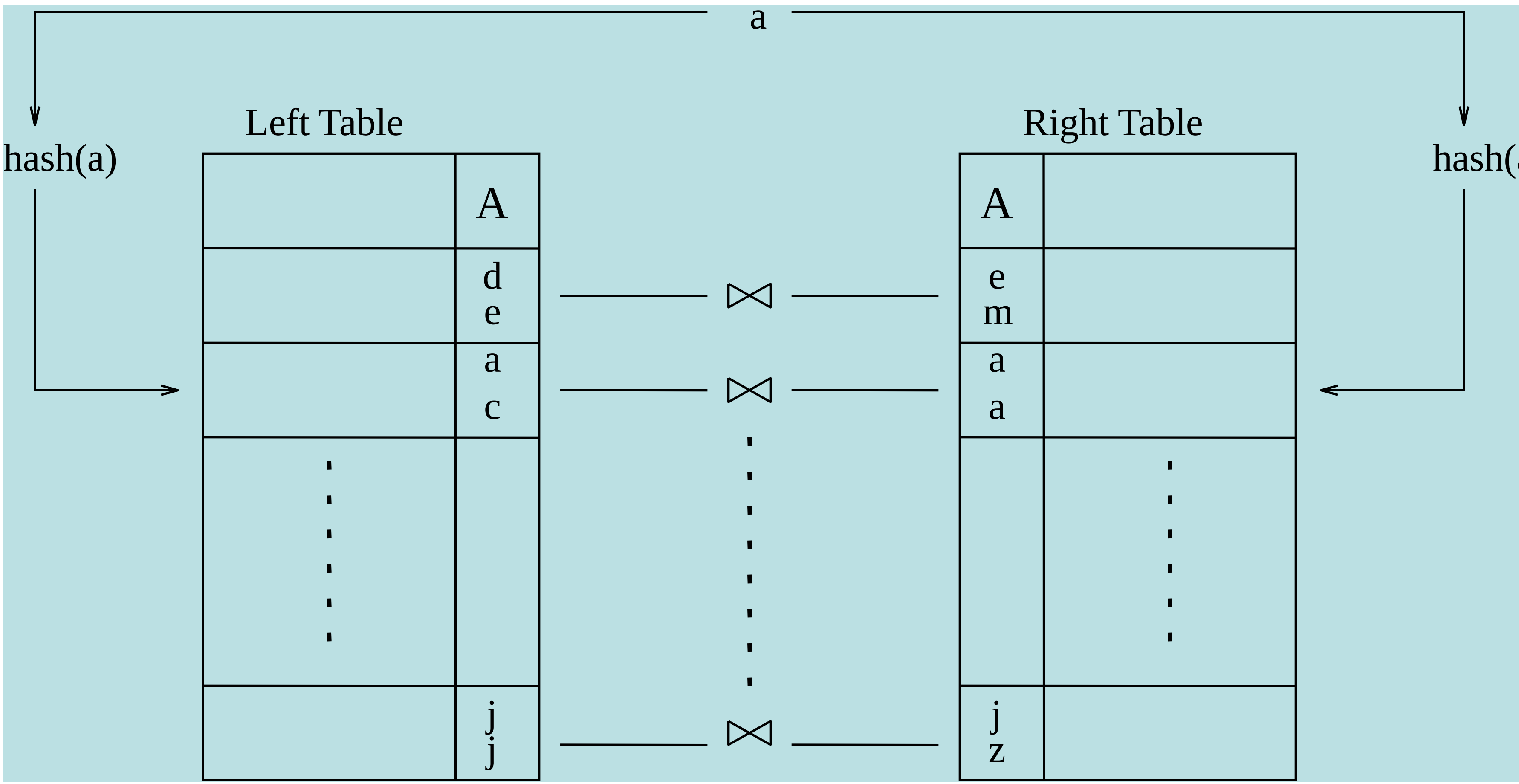
Nested-loop join



Merge-scan join



Hashed join



Cost-based optimization

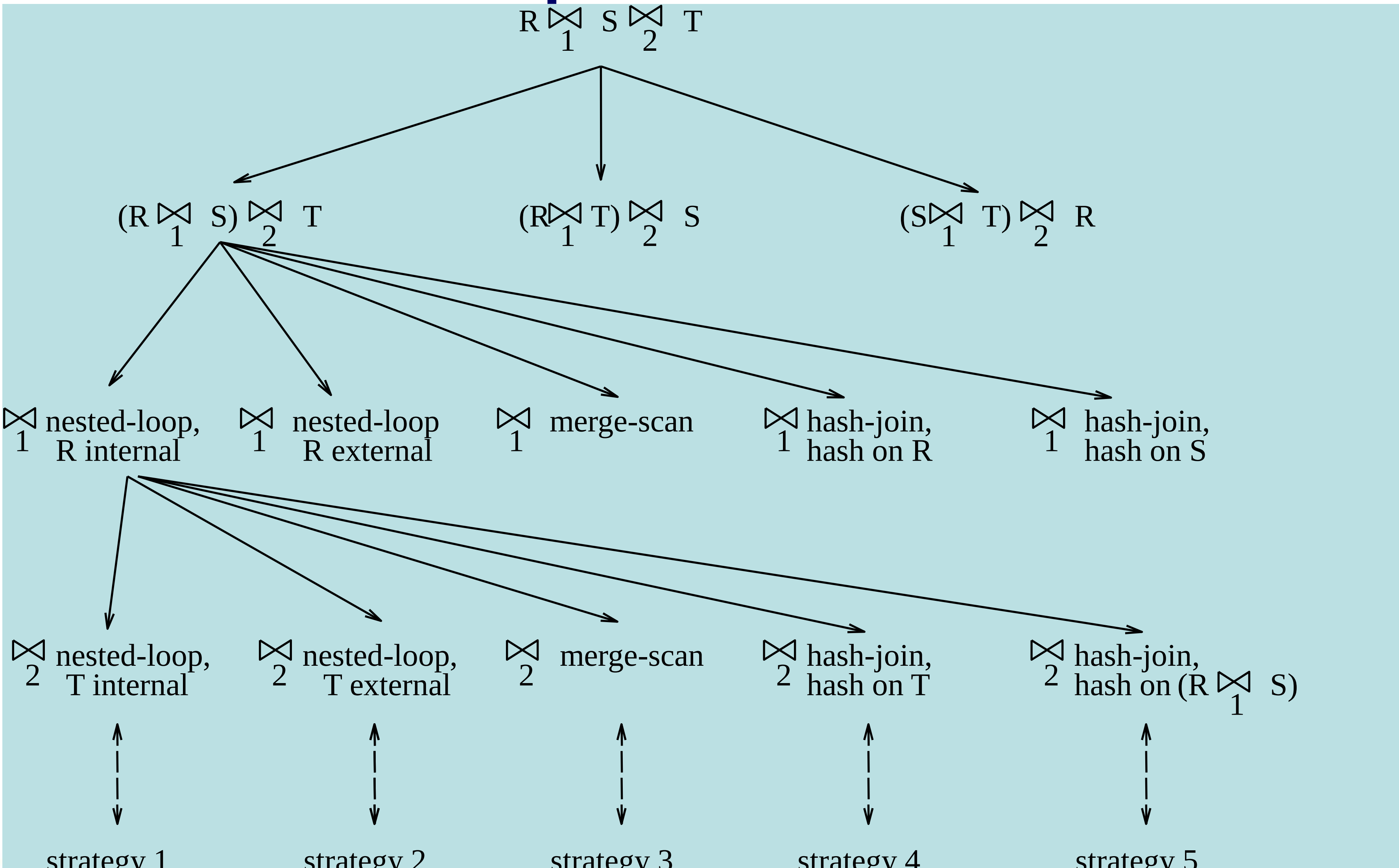
- An optimization problem, whose decisions are:
  - The data access operations to execute (e.g., scan vs index access)
  - The order of operations (e.g., the join order)
  - The option to allocate to each operation (e.g., choosing the join method)
  - Parallelism and pipelining can improve performances
- Further options appear in selecting a plan within a distributed context

Approach to query optimization

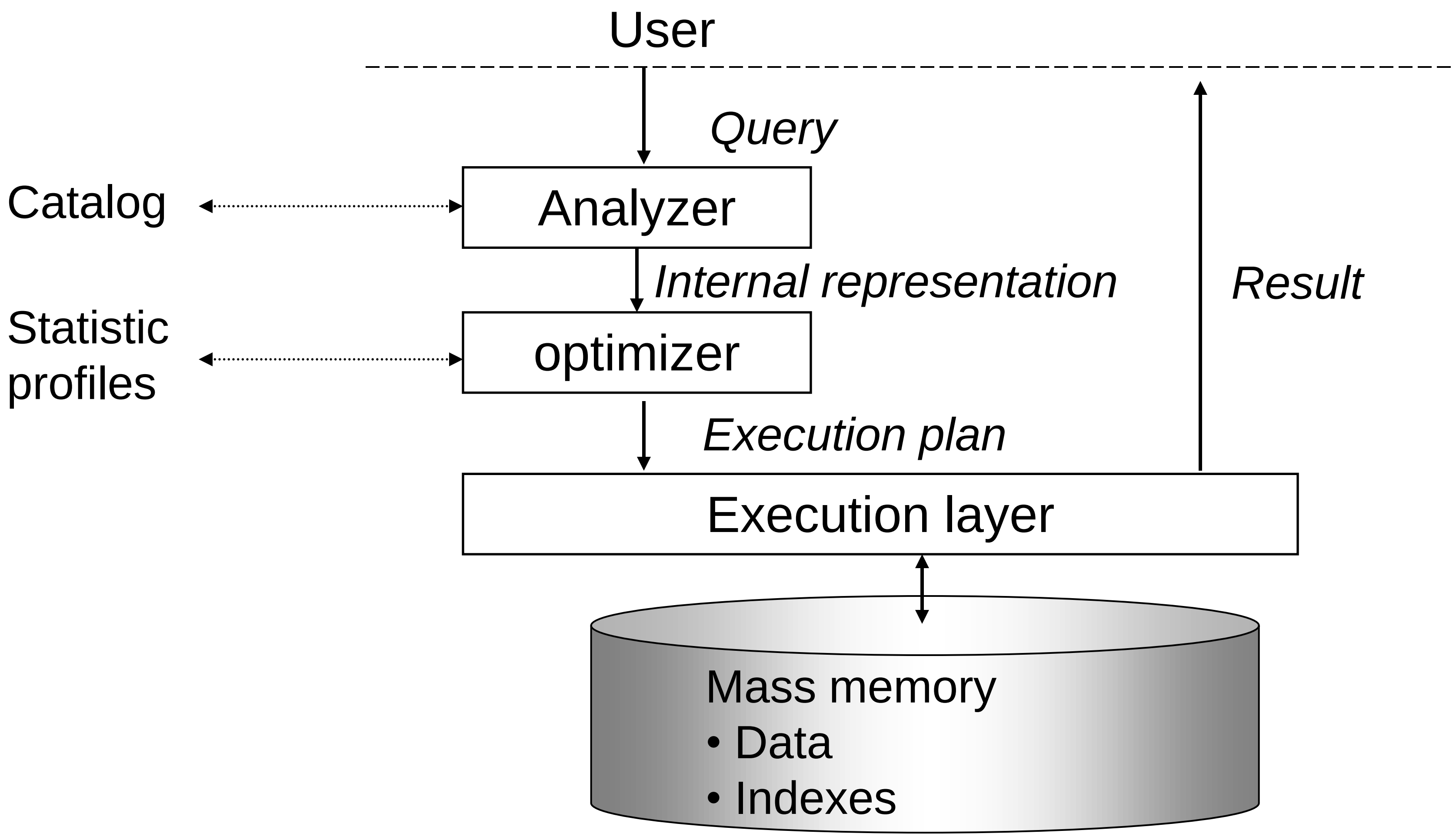
- Optimization approach:
  - Make use of profiles and of approximate cost formulas.
  - Construct a *decision tree*, in which each node corresponds to a choice; each leaf node corresponds to a specific *execution plan.*
  - Assign to each plan a cost:

    $$C_{total} = C_{I/O}\, n_{I/O} + C_{cpu}\ n_{cpu}$$
  - Choose the plan with the lowest cost, based on operations research (branch and bound)
- Optimizers should obtain ‘good’ solutions in a very short time

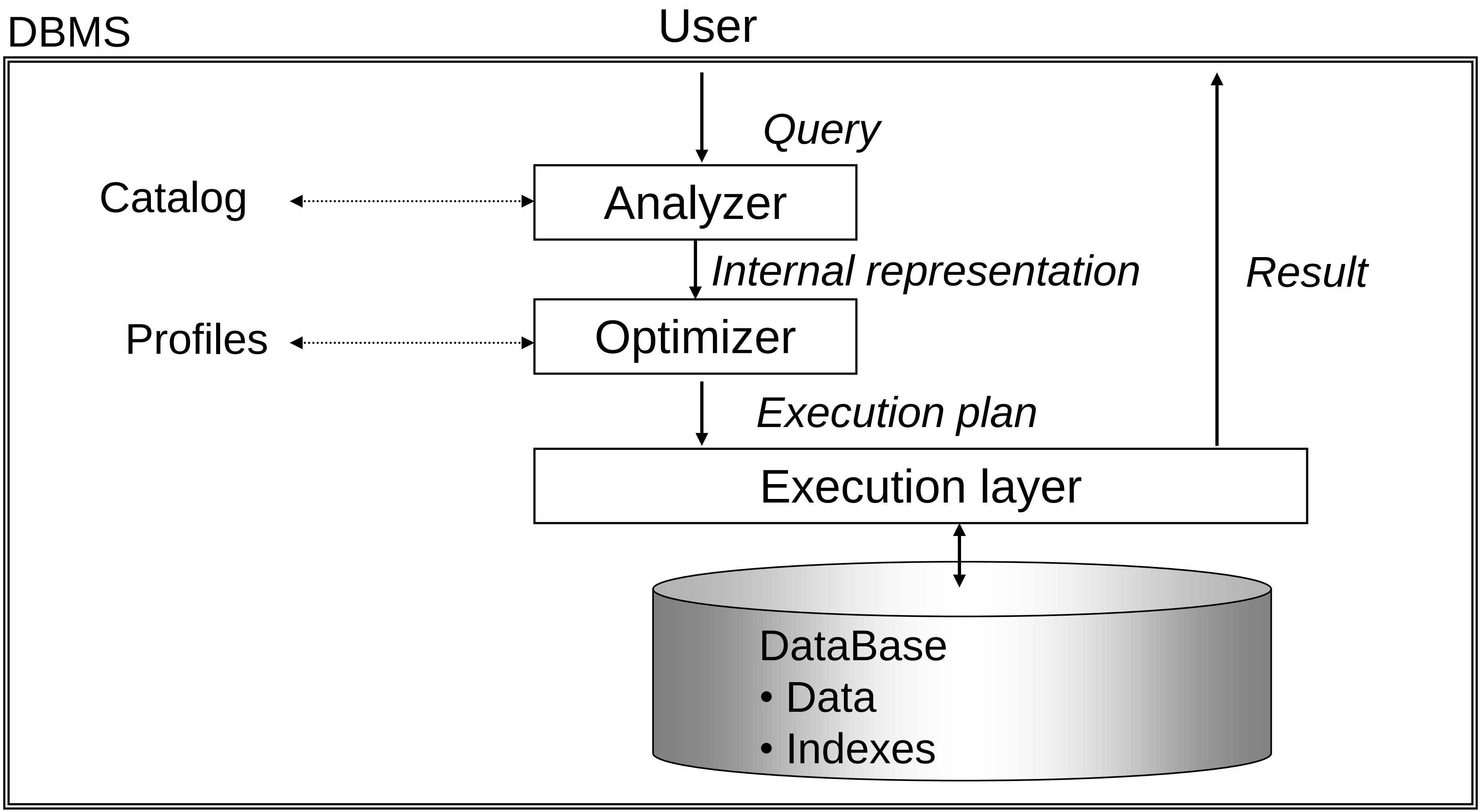
An example of decision tree



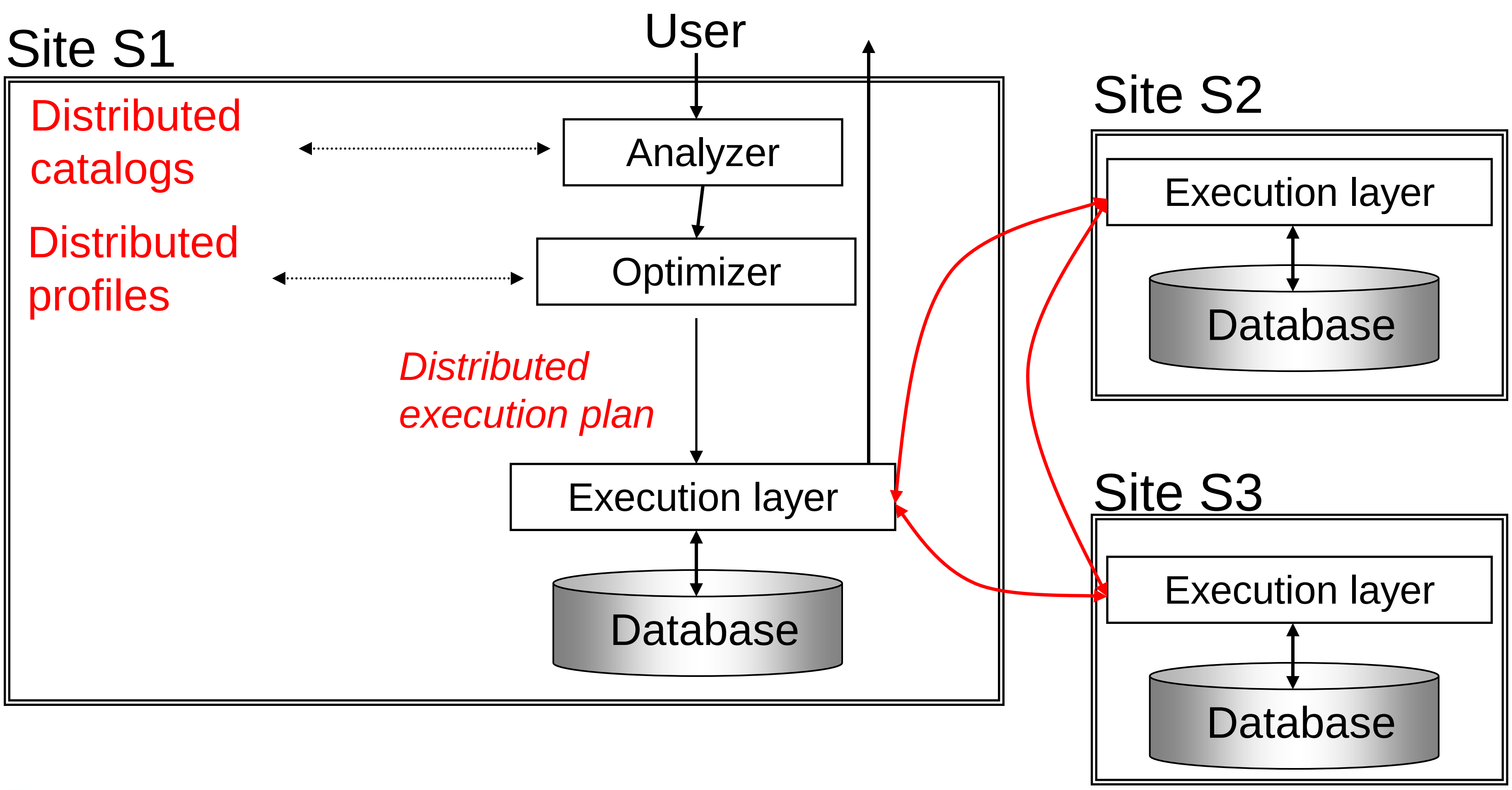
Query processing components



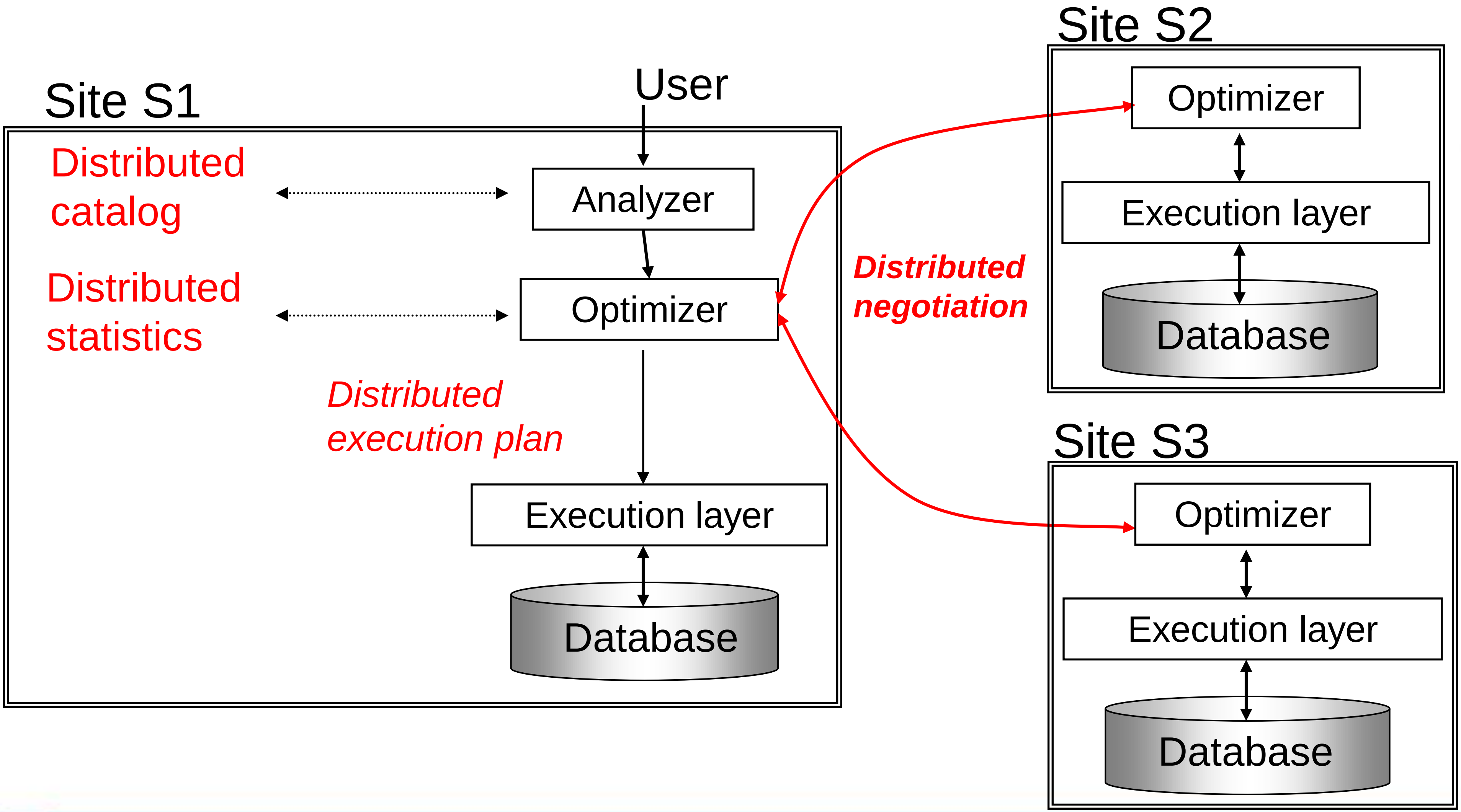
Centralized architecture (DBMS)



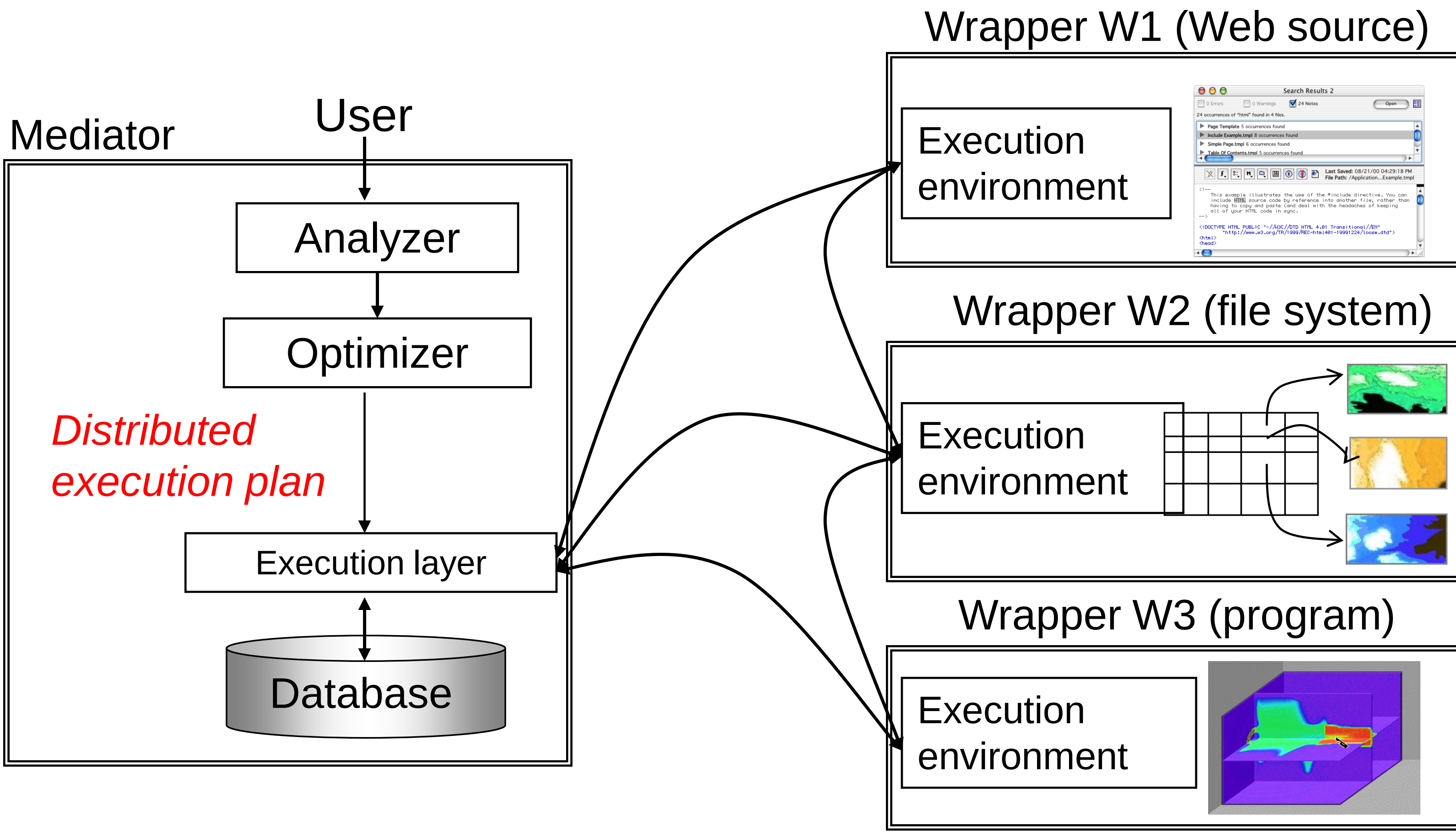
Distributed database with master-slave optimization



Distributed optimization with negotiation



Distributed system with mediator and wrappers



Overall view: components of a DBMS