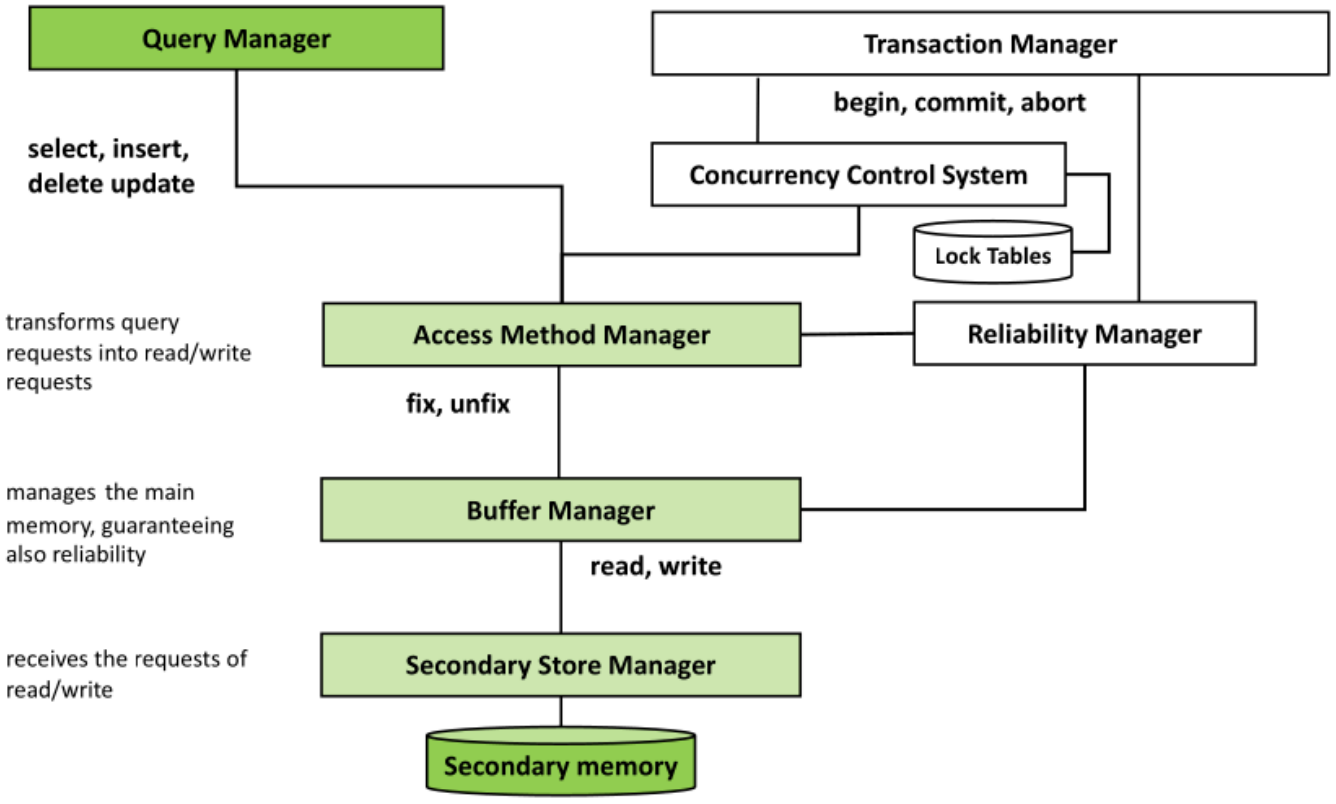
04.Physical Databases



We want to manage data that is so big that cannot reside in the computer memory. Need a manager to retrieve data from the secondary memory.

DATA ACCESS and COST MODEL

We want to manage the cost as we want to assign at each operation a cost and choose the operation that resolve a problem with the smallest cost.
Main memory: RAM, sometimes also called buffer. Typically organized in pages of the same size
Secondary memory: disk that is organized in blocks of the same size
Pages and Blocks need to be written/load in secondary/main memory to be stored/worked on.
Every time we write something in the secondary memory we are doing an IO operation and is the only cost that we are considering as it cost two order of magnitude more that any operation in main memory. We count the number of blocks that we read/write

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

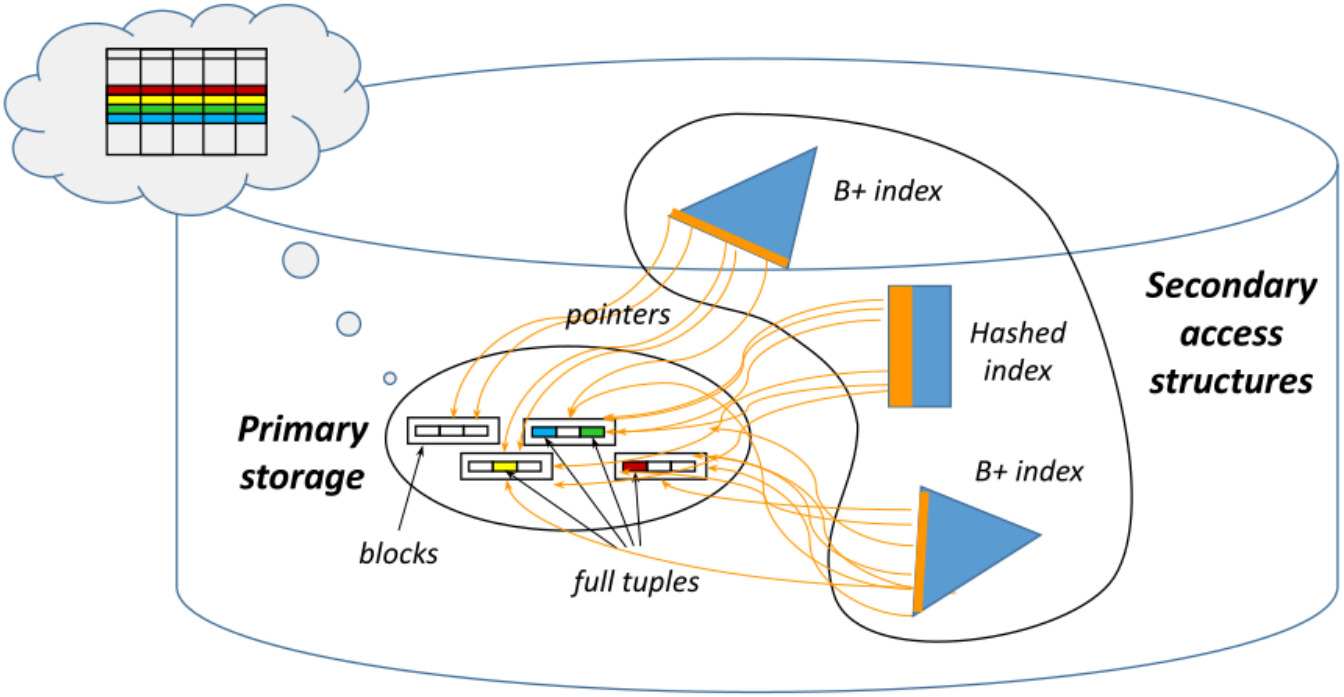
- transfer time (~1ms) - data transfer
  The cost of an access to secondary memory is 4 orders of magnitude higher than that to main memory
  In "I/O bound" applications the cost exclusively depends on the number of accesses to secondary memory
  Cost of a query = #blocks to be moved to execute a query

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

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 may have one or many optional secondary access structures(only pointer to data, convenient way to know where data is)



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

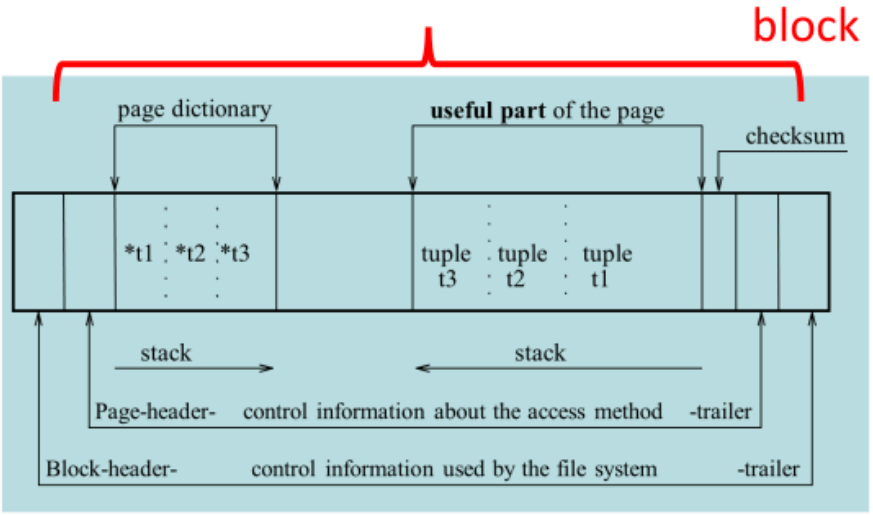
	Primary	Secondary
Sequential structures	Typical	Not Used
Hash-based structures	In some DBMSs (e.g., Oracle hash clusters, IDM DB2 “organize by hash” tables)	Frequent
Tree-based structures	Obsolete/rare	Typical

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), it cannot be anticipated, we can only do average consideration if the size is not fixed beforehand.

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(number used for checking that the data is correct)

How many tuples in a block: Block Factor

Block factor B: the number of tuples of the same type 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:$B = \lfloor SB/SR \rfloor$ ($\lfloor \mathrm{x} \rfloor = \mathrm{floor(x)}$)
  • The rest of the space can be:
- Used: tuples spanned between blocks (hung-up tuples/ records). Store just a part of a tuple and divide it in multiple blocks. Needed if tuples are larger than block size.
- 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(append them in the order they arrive)
- 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
- 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)
  The cost of the research is the number of blocks that you need to access to find the tuple.

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-Sequence	Sequentially-Ordered
INSERT	Efficient	Not Efficient
UPDATE	Efficient(if data size increase→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. Primary structure where we store the data but is inefficient and we have secondary structures that contains the data addresses and it is efficient.

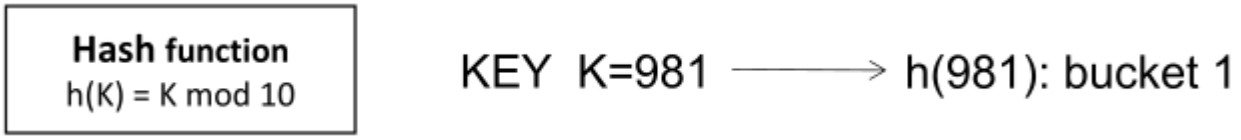
Hash-based access structure

Efficient associative access to data, based on the value of a key
A hash-based structure has NB buckets (with N_B << number of data items)

- A bucket(unit of storage) is a unit of storage, typically of the size of 1 block
- Often buckets are stored adjacently in the file
  A hash function maps the key field to a value between 0 and NB–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
  Given the key the hash table return the bucket. The number of buckets is much smaller than the number of objects that I want to store. We use an hash function to correlate the key to the index where we store the bucket.

$$hash(fileId, Key) : BlockId$$

  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

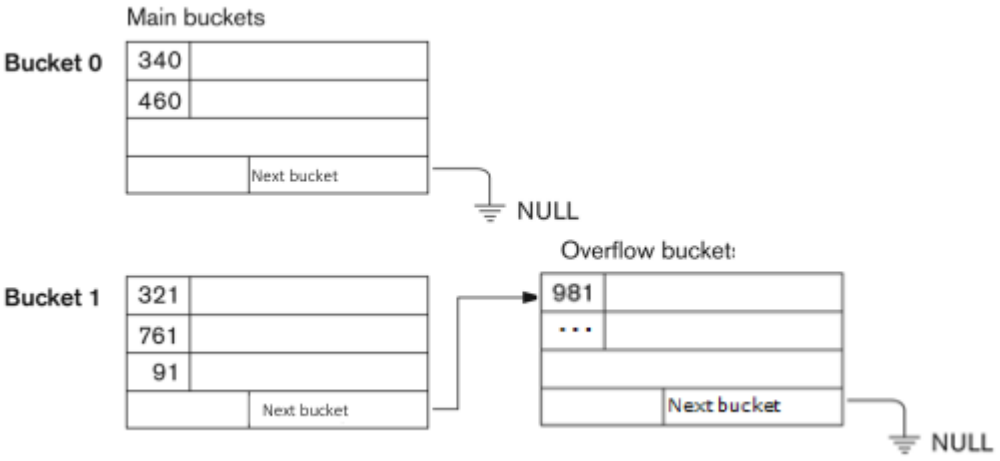


Collisions

When two keys (tuples) are associated with the same bucket

- E.g., K=983, K=723 with h(K)=K mod 10 à 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
  In DB we use chaining: if we have consumed all the bucket space I'll continue to blocks

that are attached to the datablocks to extend the bucket storage. I am assuming that I need extra space to store all my tuples.



We can estimate the cost of accessing the tuple by considering the average length of the overflow chain

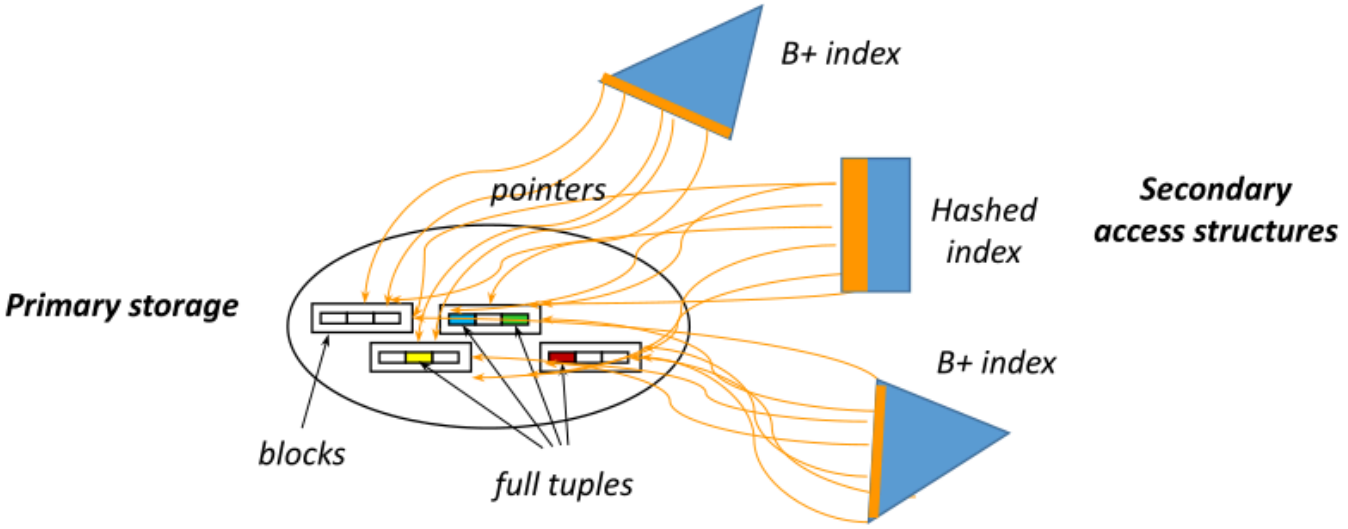
Overflows Chains

The average length of the overflow chain is a function of:

- the load factor $\rightarrow$ occupied/available slots $\rightarrow T/(BxN_B) \rightarrow$ 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 $\leftrightarrow$ 1 block)
  The more the blocks is smaller the more overflows chains will be big.

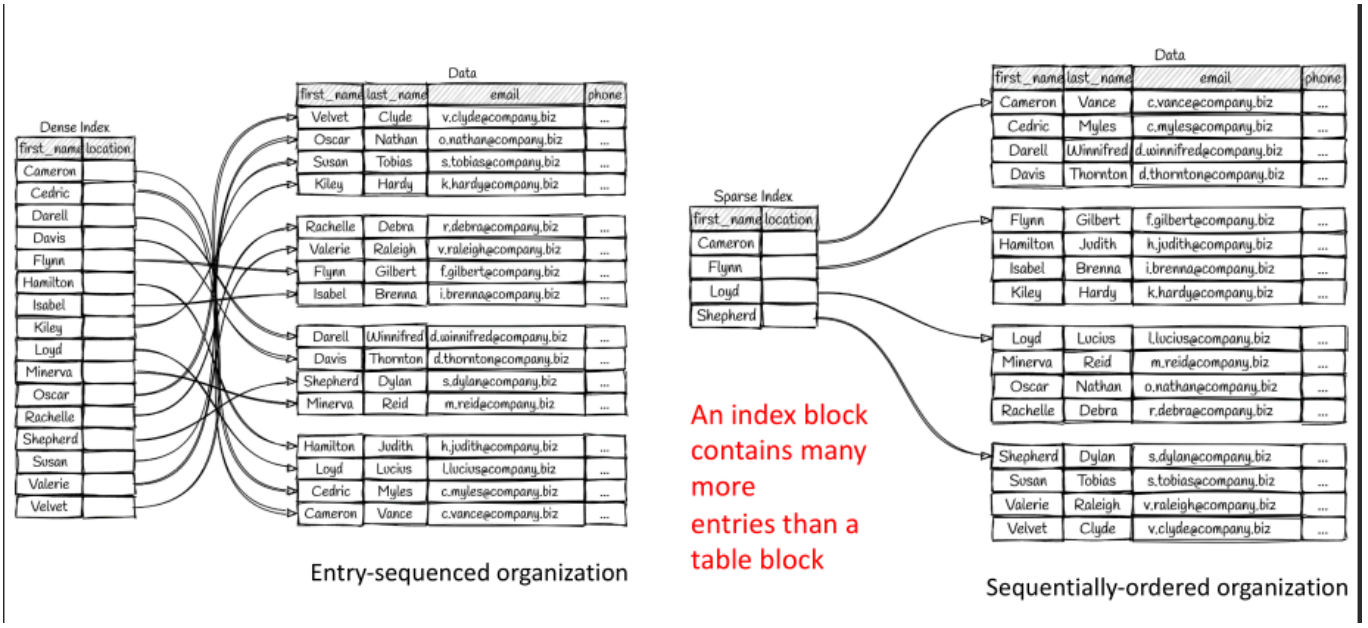
Indexes

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 $\rightarrow$ pair (term - page number) alphabetically ordered at the end of a book
Searches through indexes works by index keys(value used to see which key you are searching)



An index is dense if it contains all the values for the search. An index is sparse if it contains only some values we are searching
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

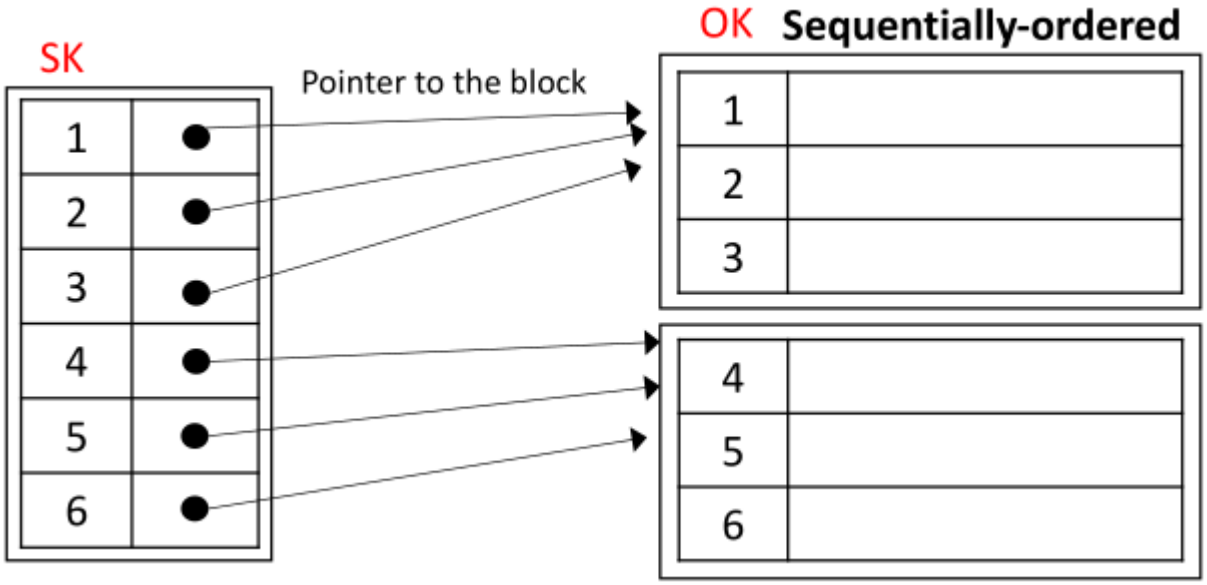


We can use sparse indexes if the structure is sequentially ordered and the structure indexes ordes is the same of the research one.

Primary index

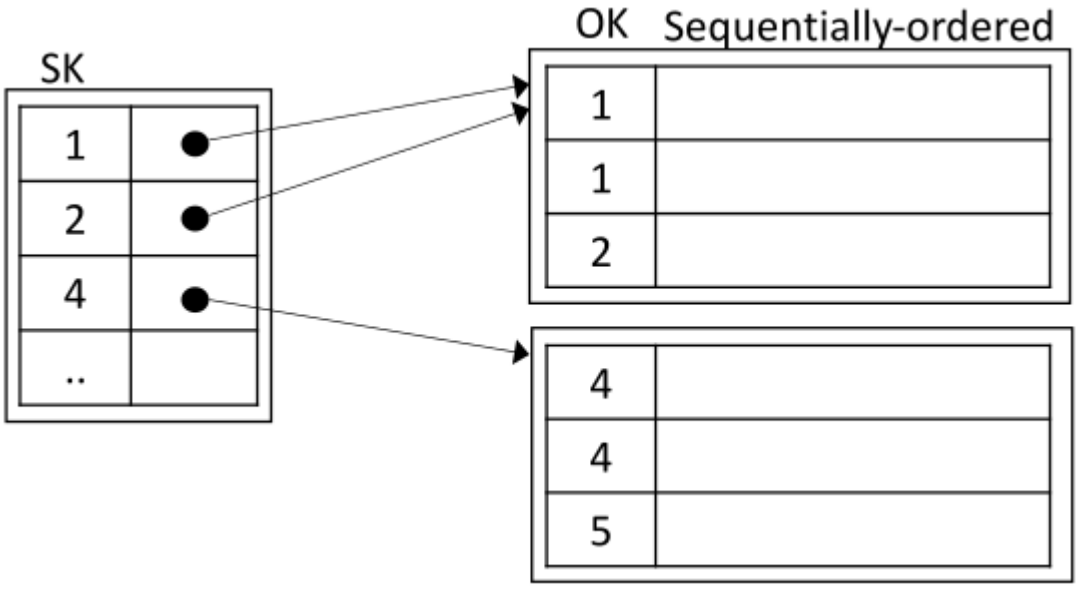
An index defined on sequentially ordered structures
The search key (SK) is unique and coincides with the attribute according to which the structure is ordered (ordering key - OK): SK = OK

- 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

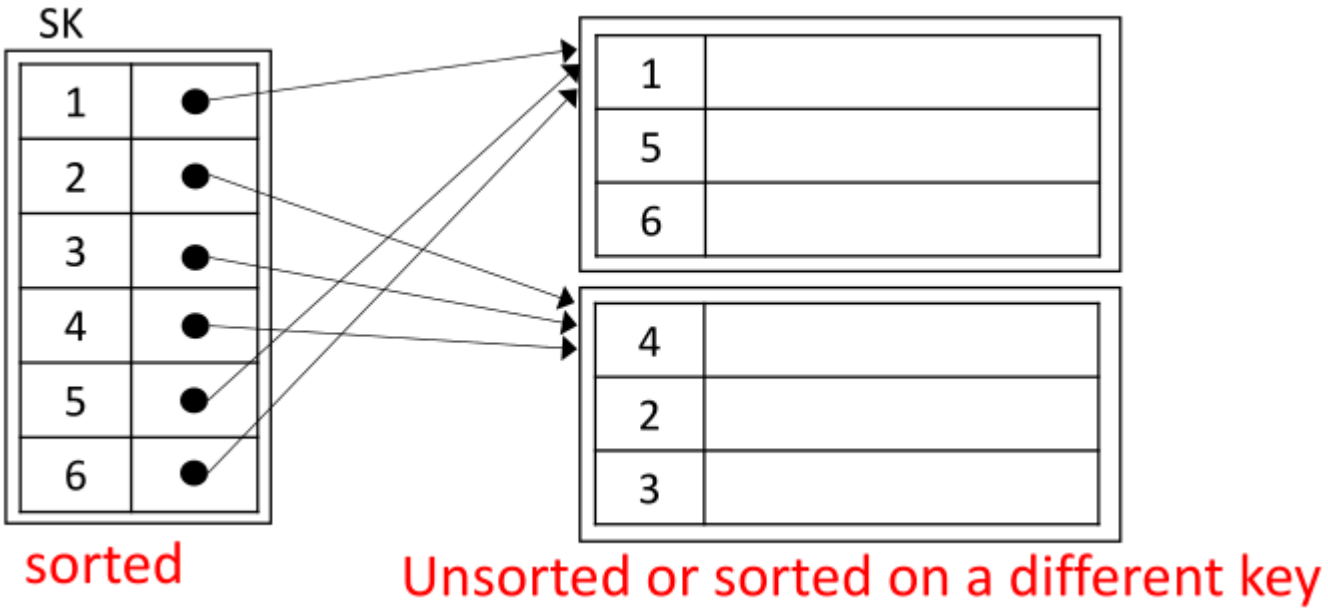
A generalization of primary index (SK = OK) in which the ordering key can be not unique
The pointer of key X refers to the block containing the first tuple of X key
Could be sparse or dense, but typically sparse: one entry in the index corresponds to multiple tuples



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

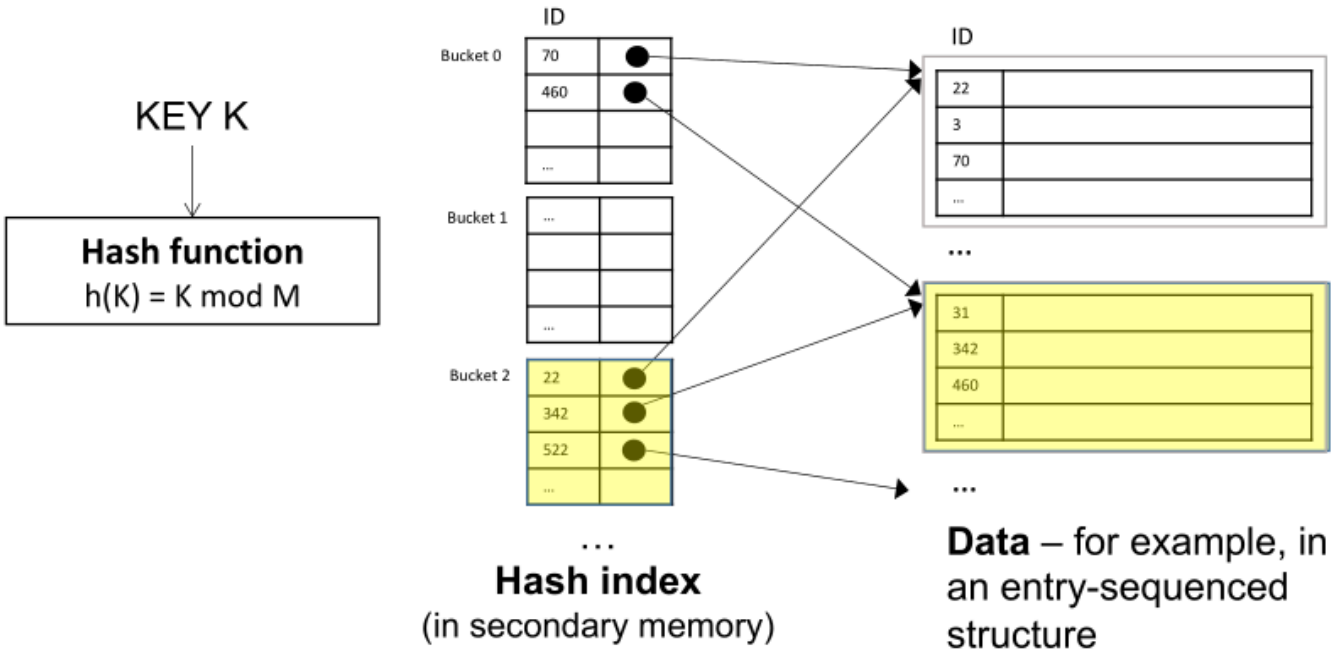
Pros & cons of indexes

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
When you update the data you need to update the indexes
You use the indexes as much as possible when you just do plain analytics query
Usually when there is a primary key the database under the hood creates an index.

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




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

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

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
  A tree is balanced if all the paths from the root t the leaves are of more or less of the same length.

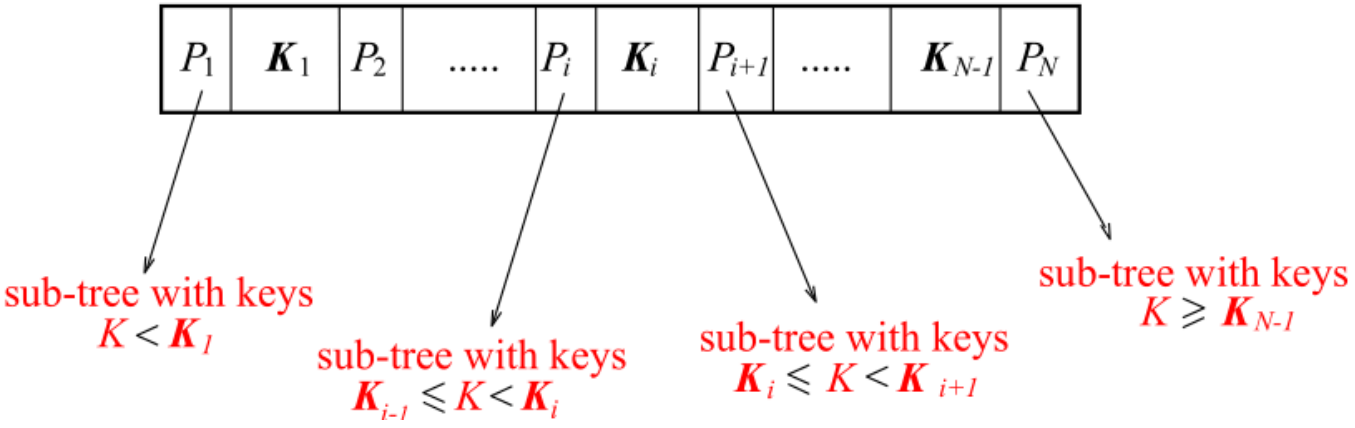
B+ trees

Information is only stored in the leaves and the values are consecutive in respect to the search other and the leaves are linked to the next one(can scan sequentially the leaves once found the starting one). So range queries are very efficient.
Evolution from B trees
Multi-level index:

- one root node
- several intermediate nodes
- several leaf nodes
  Each node is stored in a block. In general, each node has a large number of descendants (fan out 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.

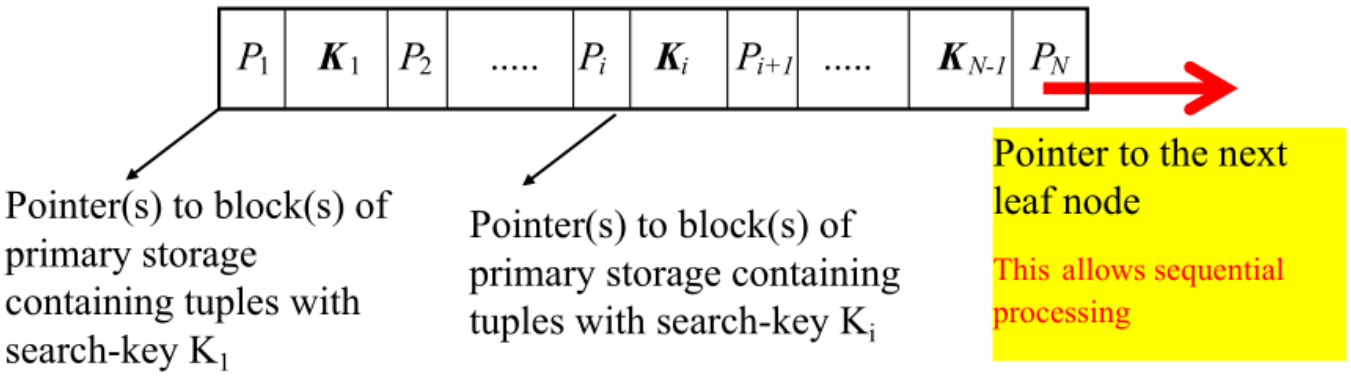
Structure of the B+ tree internal nodes

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)



Search technique / lookup

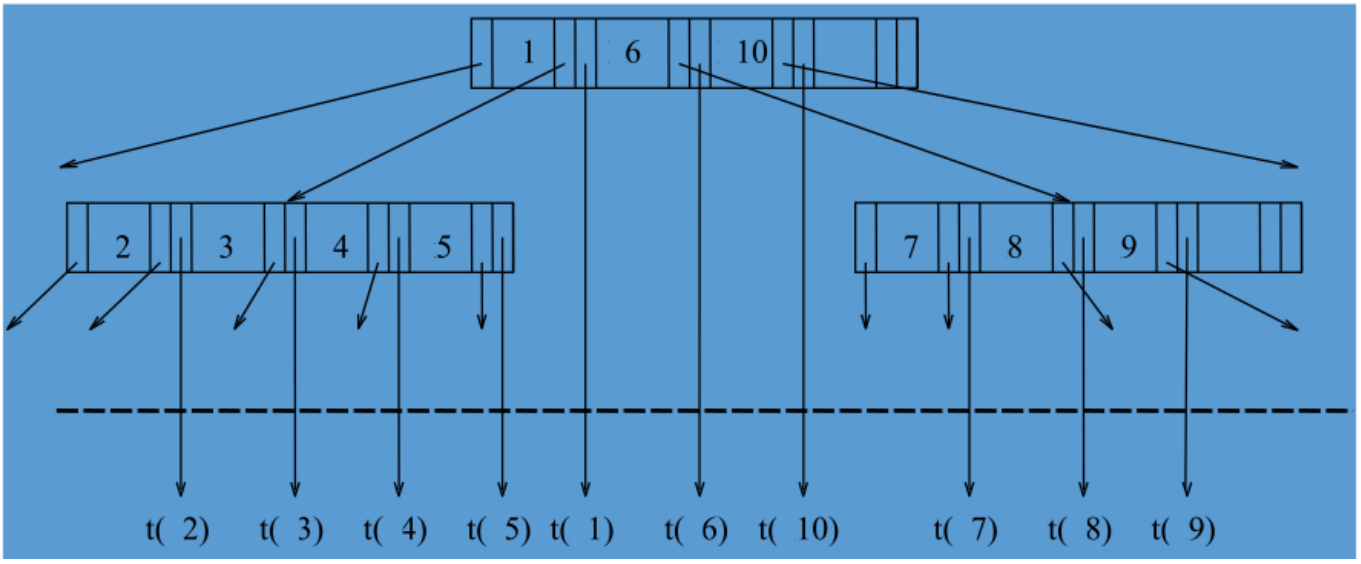
Looking for a tuple with key value V
At each intermediate node:

- if $V < K_1$ follow P_1
- if $V \geq K_N - 1$ follow P_N
- otherwise, follow $P_j + 1$ such that $K_j \leq V < K_j + 1$
  B+ tree index: query with equality predicate: #I/O operations = #levels to reach the leaf + 1
  access to the block containing the tuple
  B+ tree index: query with interval predicate: #I/O operations = #levels to reach the leaf + # of leaf nodes that are visited + #accesses to the blocks containing the tuples
  B+ tree index: query with interval
  predicate

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



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 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

Everything that need ordering needs an index. Need to see the trade off between indexing cost and speedup

INTRODUCTION TO OPTIMIZATION COSTS OF DIFFERENT ACCESS MODES

Query optimization

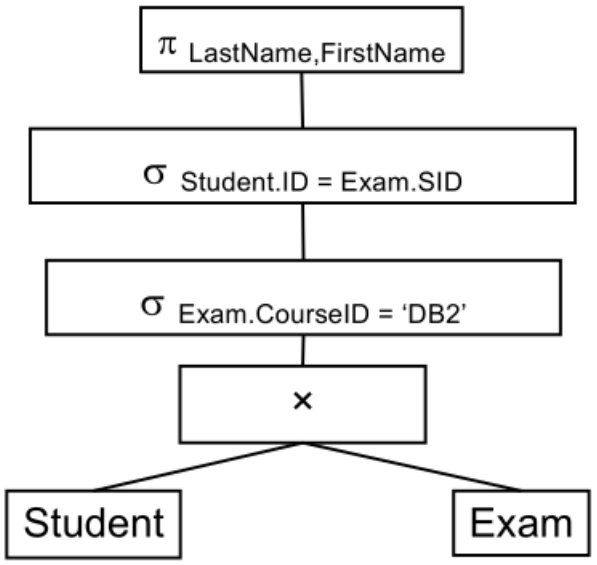
Optimizer:

- 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
  Example of query optimization:

```
STUDENT (ID, FirstName, LastName, Email, City,
Birthdate, Sex)
EXAM (SID, CourseID, Date, Grade)

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

Query Tree



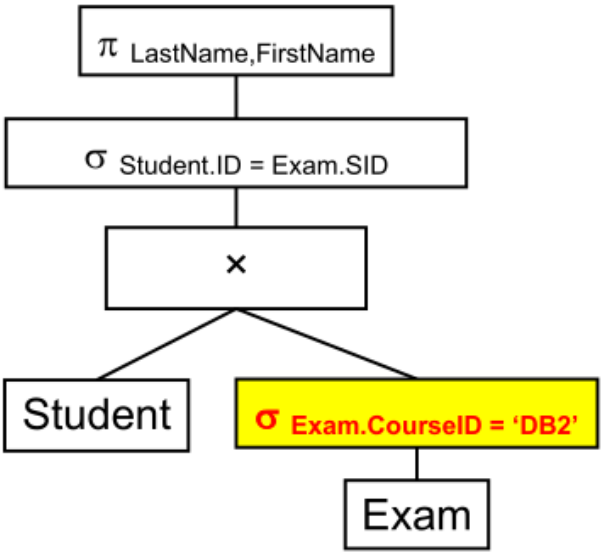

```
SELECT LastName, FirstName
FROM Student, Exam
WHERE Student.ID = Exam.SID AND
Exam.CourseID = 'DB2'
```

- 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

- 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'
```



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

Optimizations

OperationsAccess methods
• Selection
• Projection
• Sort
• Join
• Grouping• Sequential
• Hash-based indexes
• Tree-based indexes

Operations	Access methods
Selection	Sequential
Projections	Hash-based indexes
Sort	Tree-based indexes
Join	
Grouping	

Sequential scan

Performs a sequential access to all the tuples of 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)
  Can have estimate of how much blocks you need to scan to find the required data, but it is only an estimate. You can only scan sequentially, cannot jump(binary search, to do so you need indexes)

Example

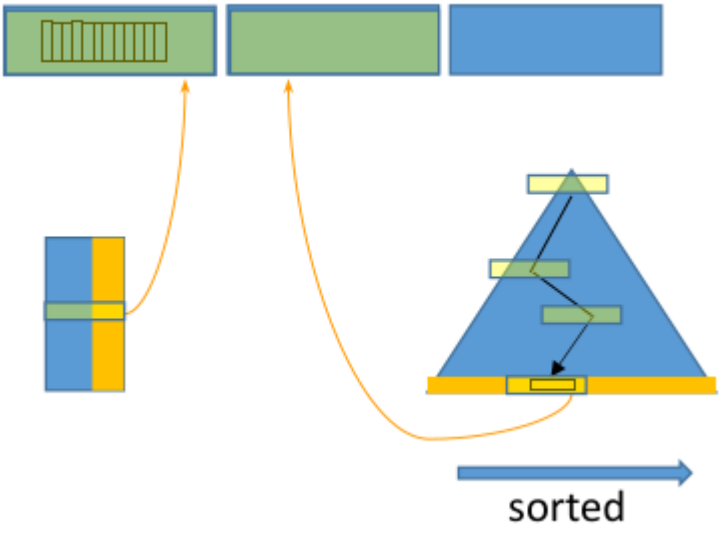
```
SELECT *
FROM STUDENT
WHERE ID<500
```

Cost: < 1.7K I/O accesses

Hash and tree based access

Hashing supports equality predicates
Tree-based indexes support:

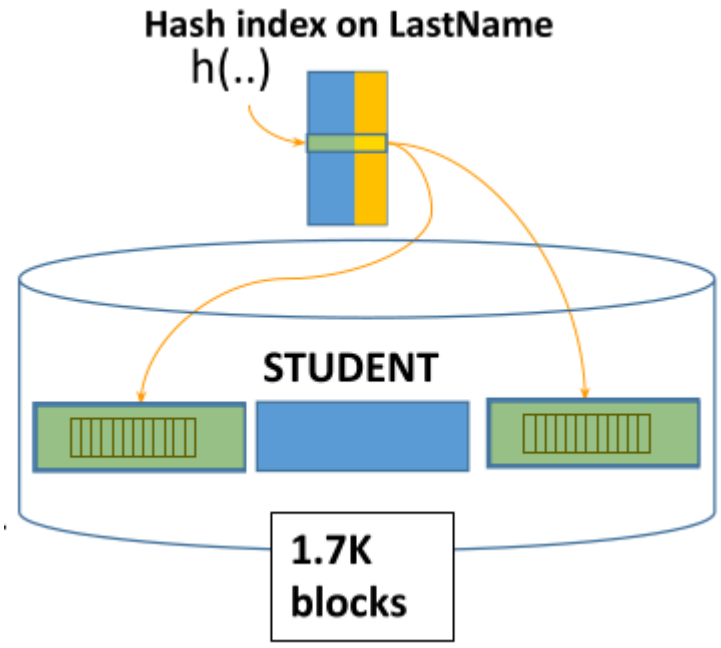
- selection or join criteria
- ORDER BY
- GROUP BY
- other operations involving sorting(such as UNION or DISTINCT)



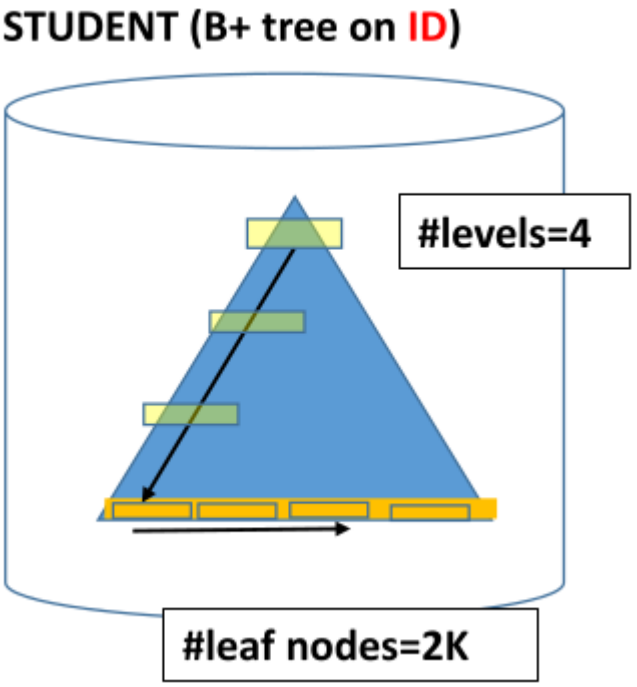
Cost of lookups: equality (A=v)

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)
  If there is a primary hash the cost is only the access cost(access plus overflow chain). If there is a secondary hash the cost is the access to each structures. With non unique search key you need to take in consideration the cost needed to access all the various tuple in different blocks(the overflow chain value is considered only once as it is the probability to have to access the overflow chain).



Equality lookup on a primary B+



Not common in practice. The leafs contains the data.

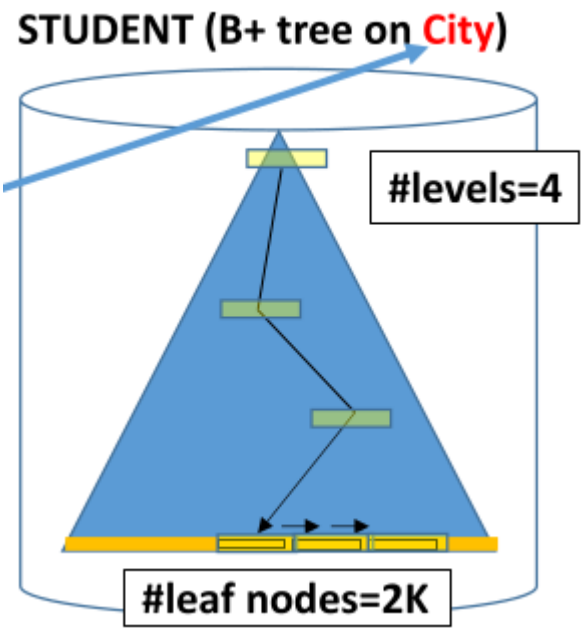
Example

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(~2K)
  Query predicate **on non unique** search key. Multiple tuple per key value:

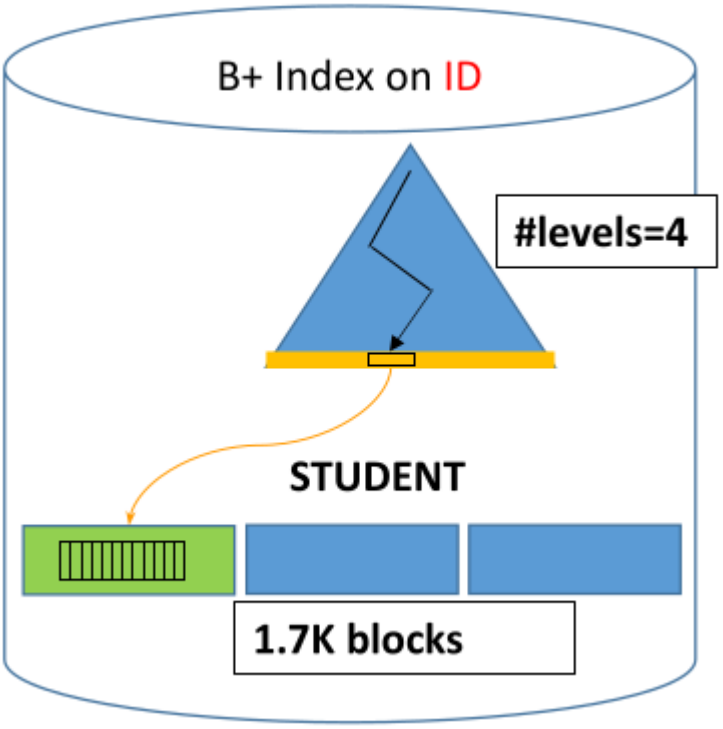



```
SELECT *
FROM STUDENT
WHERE City='Milan'
```

Cost:

- #levels to access the leaf = 4
- 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+



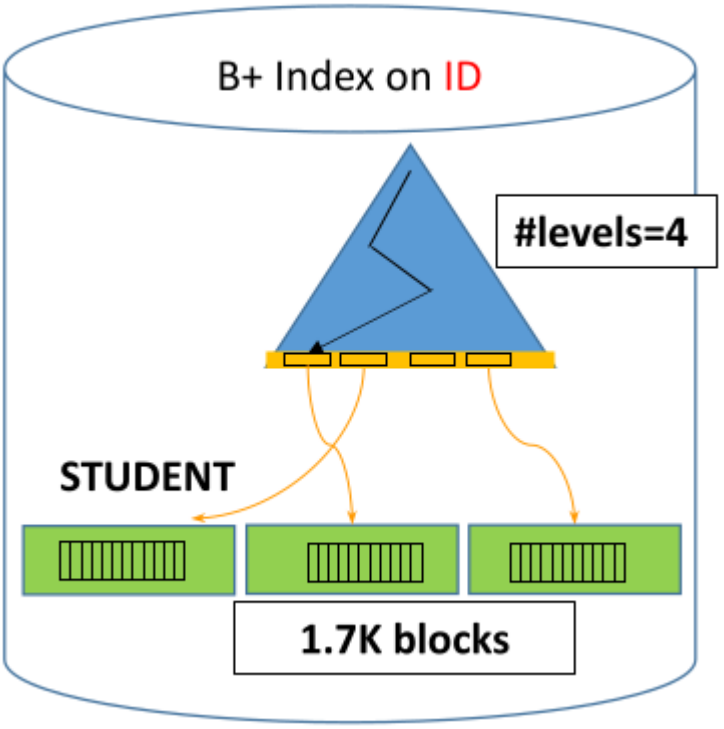
Query predicate on unique search key of STUDENT

- Only 1 pointer per key value

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

Cost:

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

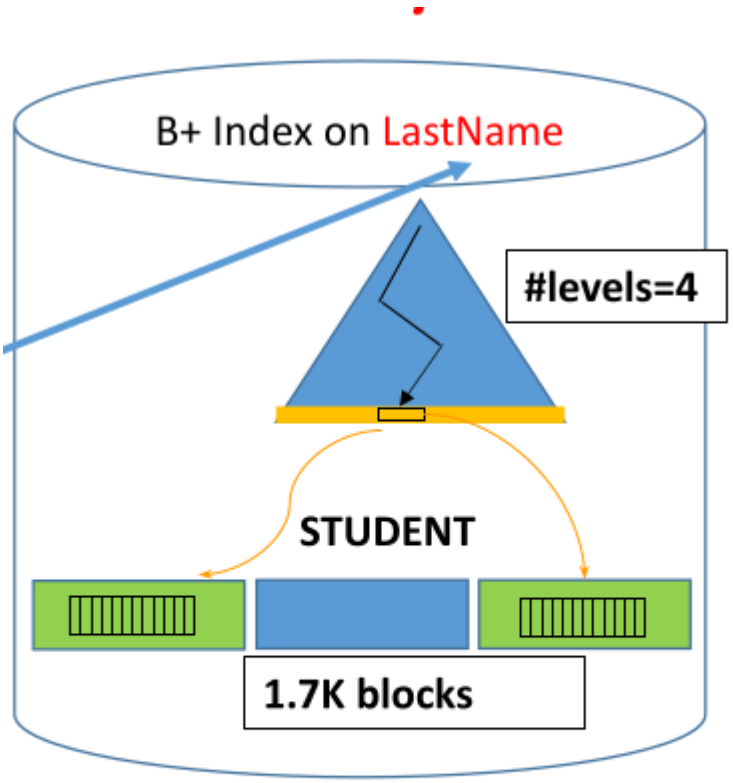


Query predicate not on search key of STUDENT

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

Cost:

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



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

Query predicate on non unique search key of STUDENT

- Multiple pointers per key value

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

Suppose Val(LastName) = 75K $\rightarrow$ 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

Interval lookups A<v , v1<= A <= v2

Common query where we need to look up an interval.
We have statistic that tells us the minimum and maximum value in a DB.
Sequential structures (primary), support only partially with assumption on data distribution:

- 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), support it:
- 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 v1 <= A <= v2 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 =v1
  - If the searched tuples are all stored in that leaf block, stop
  - Else, continue in the leaf blocks chain until v2 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 secondaty B+

We still consider a lookup for for v1 <= A <= v2 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 =v1
  - If all the pointers (up to v2) are in that leaf block, stop
  - Else, continue in the chain of leaf blocks until v2 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:

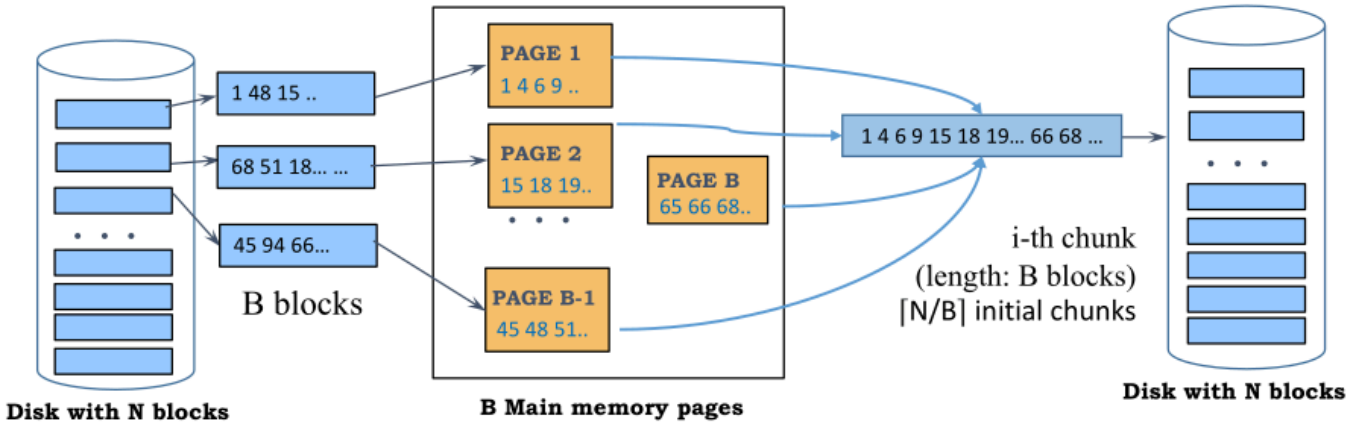
- Sort in main memory, typically performed by means of ad-hoc algorithms (merge-sort, quicksort, …)
- Sort of large files, performed with different algorithms, such as
- External merge-sort:
  - Data do not fit into the main memory
  - Procedure:
    1. Load from disk as much data as can be stored in 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

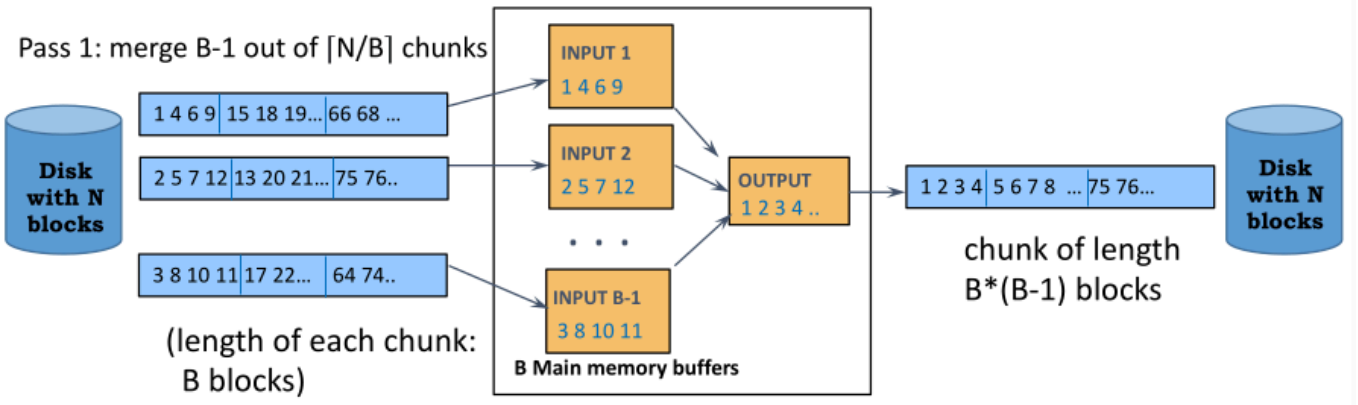
To sort a file stored in N blocks using B buffer pages:

- 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⌉



- 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



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
- ⌈2/4⌉ =1 sorted chunks of 40 pages → 40 + 40 I/O operations
- Total passes = 3

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(we use blocks and not tuples in this case, the cost is the product between the number of blocks in each DB). Nested-loop: scan the external table and for each block scan the internal table. We have to pay a cost equal to b_{EXT} X b_{INT}. If one of the table is small enough to fit in the buffer it is chosen as internal table and scanned only once $\Rightarrow$ $\text{Cost}=b_{EXT}+b_{INT}$
- These three join methods are based on scanning, ordering and hashing
  The “best” strategy is chosen based on various aspects…

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
  - $Cost = b_{Ext} + t_{Ext} \times costOfOneOndexedAccessToT_{Int}$
  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'
```

Merge-scan Join

Assume that both table are sorted by the same attribute. These are the attributes where we do the join. We will have a linear scan of both table and find the tuple as we go. The cost is linear in the number of blocks in both table.
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 *bL* (# of passes)
- 2 *bR* (# of passes)
  - #of passes cost = 1+ ⌈log B-1⌈N/B⌉ ⌉, where N = # of blocks, B = # of buffer pages
  If we have a secondary structure we extra cost if we have to access another structure as we have to check for every row(we have to follow all the pointer to see if the data respect the constraints)

Hashed Join

Assume that the join attributes have an hash index on both table and they are using the same hash table in both function ⇒ we have the same value in both tables for the same hash. Do a linear scan of both tables and select the cost from both table. Cost is the sum of the number of blocks in both tables.

Cost Based Optimization

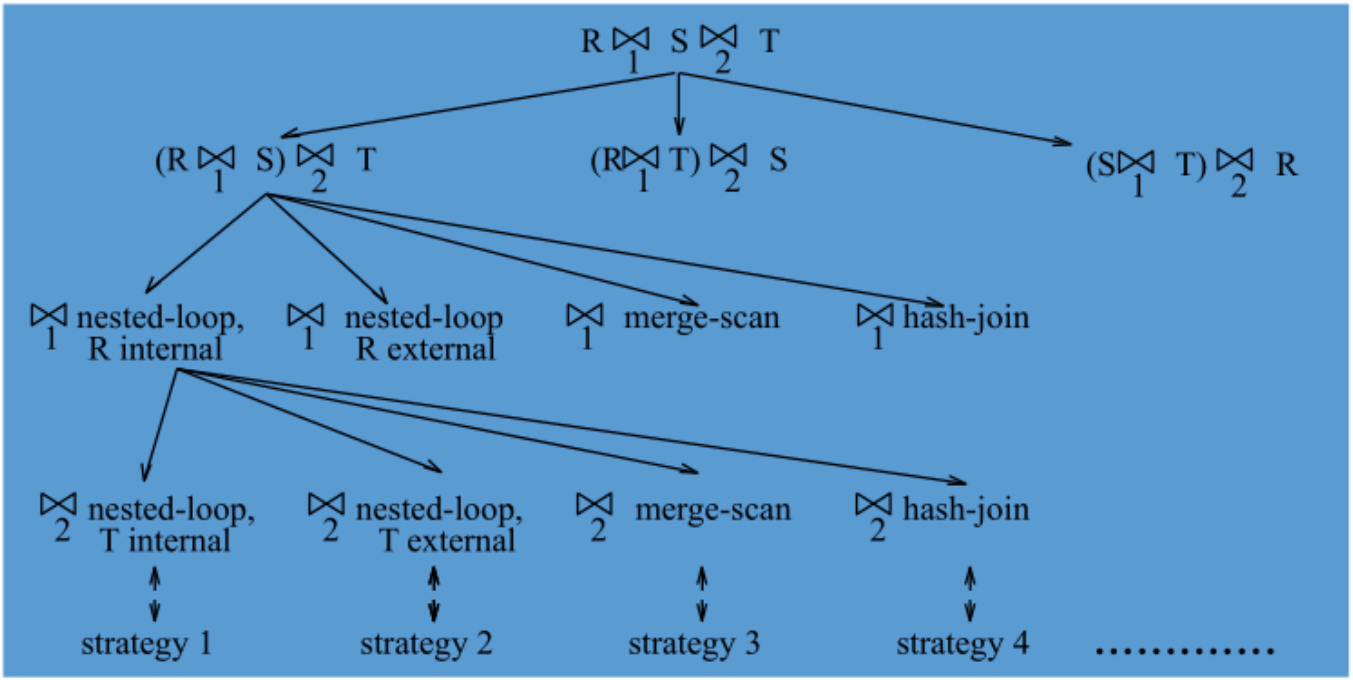
An optimization problem, whose decisions are:

- 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



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