Introduction to the Cost Model

The *cost model* evaluates the ***space used*** (number of pages) and the ***number of read/write accesses*** to disk of the several operations (*equality search* for a single value, *range search* for an interval of values, etc). Parameters used for performance evaluation:

- File R
- Number of records $N_{rec}(R)$
- Length of single record L_r
- Number of pages $N_{pag}(R)$
- Size of single page D_{pag}

Serial (Heap) and Sequential Organizations

Both are very simple data organizations where records are stored contiguously; both **do not require auxiliary data structures** to access the records and the space occupied is equal to the memory needed to store the record one after another. Main difference: in the sequential organization records are ordered on a key K, in the serial (heap) organization they're not. s

Serial (Heap) organization. Efficient (standard organization of DBMS) only for small or very small files, otherwise performs badly for big files. As for the ***space used*** we have:

- Space: $N_{pag}(R) = N_{rec}(R) * L_r \,/\, D_{pag}$

As for the costs:

- ***Equality search***: we need to **full scan the file** until either we find a suitable result or we reach the end of the file. On average, the cost for equality search is:

  $C_s = \left\lceil \frac{N_{pag}(R)}{2} \right\rceil$ if we're looking for a key attribute

  $C_s = N_{pag}(R)$ otherwise

- ***Range search***: we need to **full scan the file** (records are not sorted, there could be interesting records everywhere on the file). The cost is $N_{pag}(R)$ because we need to read all the records.
- ***Insert***: very efficient, it's enough to insert record at the end of the file. The cost is **2** (1 to read the last page plus 1 to write the last page).
- ***Update and Delete***: implies the cost of searching for the record to update or delete plus the rewrite; the cost is the cost of search plus 1: $C_s + 1$.

Sequential Organization

Records are ordered on a key K and stored contiguously. Equality and range searches are both faster, but it's **difficult to maintain the file ordered** when there are many *insertions* and *updates*. For these reasons, sequential organization is rarely used in DBMS. As for the ***space used*** we have:

- Space: $N_{pag}(R) = N_{rec}(R) * L_r \,/\, D_{pag}$

As for the costs:

- ***Equality search***: if we can use *binary search* the average cost is:

  $$C_s = \log_2 N_{pag}(R)$$

  Otherwise the cost is the same as before, $N_{pag}(R)$.

- ***Range search***: if we need to find the records whose key value k is $k_1 \leq k \leq k_2$, first we compute the ***selectivity factor*** (estimated number of records that satisfy the condition):

  $$s_f = \frac{k_2 - k_1}{k_{max} - k_{min}}$$

  Then we compute the cost:

  $$\log_2 N_{pag}(R) + s_f * N_{pag(R)} - 1$$

  It's the cost of the equality search plus the number of pages belonging to the selectivity factor minus 1.

- ***Insert***: if pages are full, we need to rewrite all the pages from that of insertion to the end of file:

  $$C_s + N_{pag}(R) + 1$$

- ***Update and Delete***: same as before, $C_s + 1$.

Note (E_{rec}): in both cases to compute the number of records satisfying a certain range condition we have:

$$E_{rec} = N_{rec}(R) \times s_f$$

Once E_{rec} has been computed, we can compute the corresponding number of pages applying the previous formula:

$$N'_{pag}(R) = E_{rec} \times L_r / D_{pag}$$

Sort-merge algorithm for sorting files

Since *sequential organization* requires sorted records before performing any operation on them, we use an external sorting algorithm called ***sort-merge***. Suppose we have $N_{pag}(R)$ pages and B the buffer pages available:

1. **Sort phase.** $B < N_{pag}(R)$ pages are read into the buffer, sorted and written to the disk into a file called *run*. We have $S = N_{pag}(R) \,/\, B$ runs in total, each with B pages (except the last one).

2. **Merge phase.** Perform multiple merge passes: in each merge pass $Z = B - 1$ runs are merged and the number of runs is reduced by the Z factor ($S = S/Z$). Z is called **merge order**.

As for the **cost of *sort-merge***, it depends on the number of sort passes and merge passes:

$$C = 2 \times N_{pag}(R) \times (1 + log_{B-1}S)$$

…where $1 + log_{B-1}S$ is the total number of passes to sort the file. Note that if the number of pages is less than the squared number of pages of the buffer ($N_{pag}(R) < B^2$) then we **only need one merge pass**; the cost is:

$$C = 4 \times N_{pag}(R)$$

Note: the cost of reading/ writing data on a temporary file is the cost of accessing every page: $N_{pag}(R)$. Of course if we're interested in writing/reading only a subset of such pages, we must compute the number of pages satisfying the condition of the subset ($N'_{pag}(R)$ using E_{rec}).

Organizations based on unique keys

These organizations aims at quickly locating a record using a given key. They can be divided into ***primary organizations*** (require a certain physical organization of the files on the disk) and ***secondary organizations*** (no requirements).

Primary organization: *Hashing*. A record is read in a page whose address is obtained by applying a hashing function H to the record key value. Different keys can be mapped to the same address (***collision***); when a page is full and there's another key mapped to it, we have an ***overflow***. We have:

- ***Static hashing organization***. This organization has a *primary area* with M pages, each with a certain capacity c (the number of records of a page). The ***loading factor*** of the primary area is:

$$load_f = \frac{N_{records}}{M \times c}$$

A record in inserted in the primary area using this hashing function:

$$H(k) = k \bmod M$$

… where M is the number of pages of the primary area (also a **prime number**). When a page is full (c records are allocated) an overflow occurs; to manage it, either we find some free page in the primary area we can use (**open addressing overflow**) or we use a separate *overflow area* where we put every overflow (**chained overflow**). In general, when the primary area is well designed (80% page occupancy) we can assume that there are no overflow so a record is retrieved with **only 1 page access**.

Observations: the higher the loading factor, the higher the cost of operations; the higher the page capacity, the lower the percentage of overflows. In general it's convenient to have $load_f < 0.7$ and $c \gg 10$. **Static hashing organization is *not suitable* for range search**. Static hashing organization also needs several **reorganizations of the hash file** (= dump the file and reload it) when the overflow percentage is high and when many deletions have been performed (and we want to reuse the space left by logical deletion).

- ***Dynamic hashing organization.*** Dynamic hashing organizations can have auxiliary data structures (*virtual hash*) or not (*linear hash*).

Virtual hash. The idea is to start with a small data area with a limited number of M pages (for example $M = 7$); an **auxiliary bit-array**, B, is used to say whether the corresponding page has some records (1) or not (0). The hashing function in the general case is:

$$H_j(k) = k \; mod \left(2^j \times M\right)$$

Where j is the number of duplications performed on the small initial file. To insert a new record, suppose we're in this situation:

0	1	2	3	4	5	6
112 1176	519 3277 848	723	6856	6647 1075 7830	2385 2665 2840	286
1	1	1	1	1	1	1

We want to insert a record into page 5…

$$H_0(3820) = 3820 \; mod \; (2^0 \times 7) = 5$$

…but that page is full: we simply duplicate the initial file…

0	1	2	3	4	5	6	7	8	9	10	11	12	13
112 1176	519 3277 848	723	6856	6647 1075 7830	2385 2665	286						3820 2840	
1	1	1	1	1	1	1	0	0	0	0	0	1	0

…and every record causing the overflow is mapped with both the old and the new function:

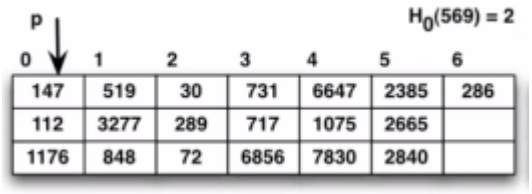
$$H_1(3820) = 3821 \; mod \; (2^1 \times 7) = 12$$

As for the **memory requirements**, Virtual Hashing requires the data area and the space for the bit-array B. The memory is not well used: we have many empty pages and the data area is frequently duplicated.

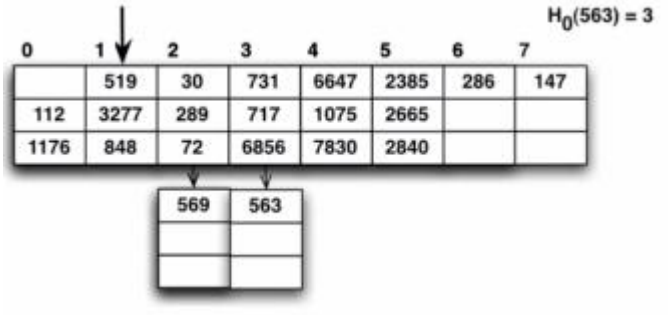
Linear hash. The idea is to increase the size of the initial file *linearly* when there's an overflow. Suppose we start with a file with $M = 7$ pages of capacity

$c = 3$. A pointer p points to the **first page that will be duplicated when there's an overflow**: that page is duplicated even if the overflow happens in any other part of the file; once the overflow happens, p is increased by 1 position (when p reaches the latest available page, it starts again from the beginning). For example, suppose we want to write the record 569; the hashing function is the same as before:

$$H_0(569) = 569 \; mod \; (2^0 \times 7) = 2$$



Since we have an overflow, we add the record with an auxiliary structure, we duplicate the page pointed by p, we split the records in page p between the original page and the new duplicated page using the hash function $H_1(k) = k \; mod \; 2^1 M$ and we increase p by 1:



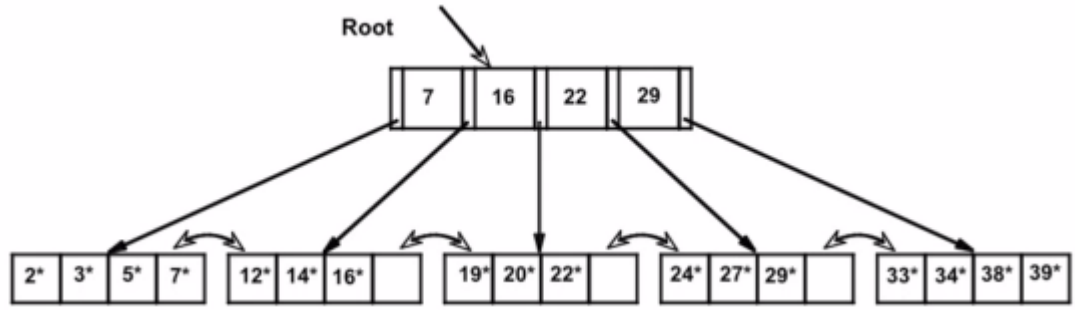
Conclusions. Both virtual and linear hashing are unsuitable to perform range search. It's also difficult to evaluate the performances in the worst case. They're very efficient but used rarely.

Primary organization: *Tree structures*. Tree structures are the **best solutions for range and equality searches**. The most important variant is called B^+*-tree* (multi-way trees where **nodes have many children**), as the original B-tree structure is not used anymore.

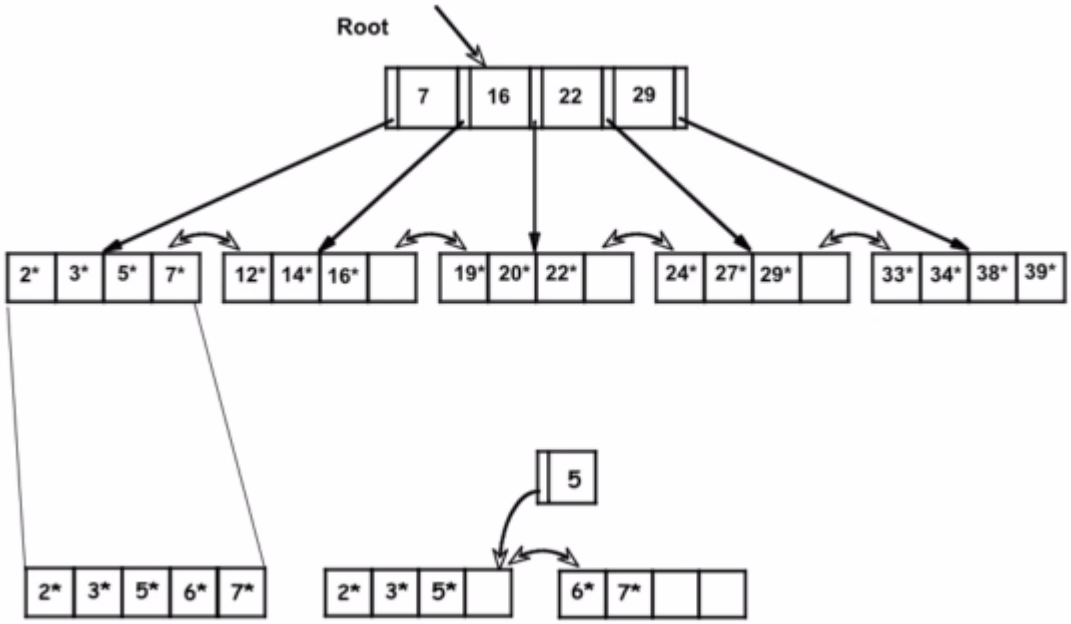
B^+-tree organization. Suppose we have a m-order B^+-tree, where $m \geq 3$:

- *Non-Leaf nodes* contain the key values (k) used to search for data entries in the leaves;
- *Leaf nodes* are pages containing the records (both key and data, k^*), which are **sorted** and organized into a **doubly linked list**.

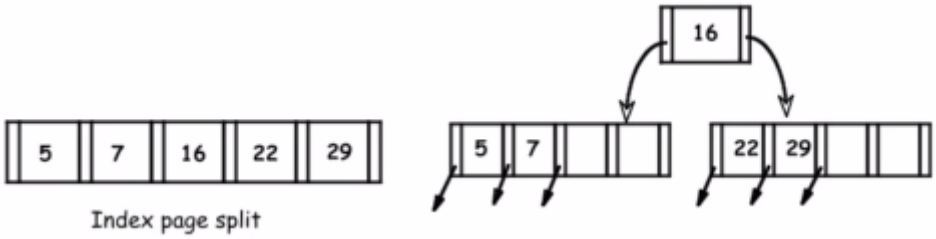
Every node has at least $\frac{m}{2} - 1$ elements, at most $m - 1$ elements. An example is:



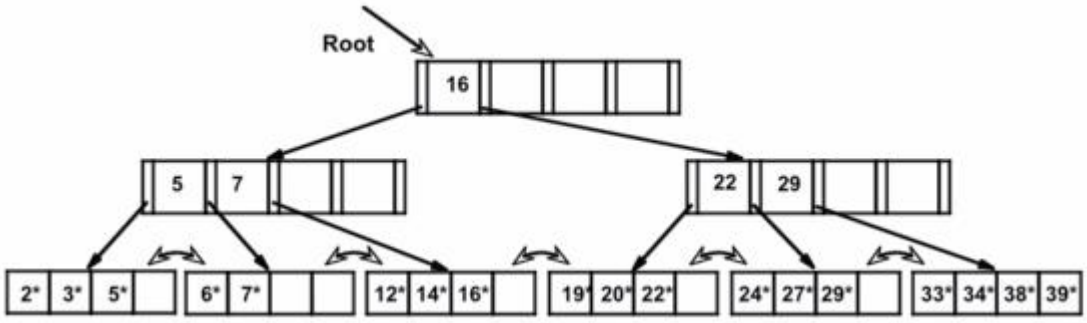
Insertion. To insert a record in a full page we must preserve the characteristics of the B^+-tree structure. Suppose we want to insert 6^*: we insert it into a leaf (a page) which is then split into two other leaves (pages) of acceptable capacity; the median element is put into the parent node:



Note that if the root node is full, we need to split it too:



The result is:



Deletion. To delete a single record, we start from a leaf node: if the number of elements of the node after deletion is $\geq \frac{m}{2} - 1$ then deletion ends. If it's instead $< \frac{m}{2} - 1$ we must perform a **merging** (the node is merged with a brother that has $\frac{m}{2} - 1$ elements; the overall number of nodes decreases) or a **rotation** (we get some elements from a brother which has elements between $\frac{m}{2} - 1$ and $m - 1$ and we move them into the sub-used node; the overall number of nodes is the same as before).

Costs. The **cost of searching** in a B^+-tree depends on the height (number of levels) of the tree, h:

$$1 \leq C \leq h$$

The height h can be estimated between:

$$\log_m(N+1) \quad \leq \quad h \quad \leq \quad 1 + \log_{\lceil \frac{m}{2} \rceil}\left(\frac{N+1}{2}\right)$$

… where m is the order of the tree, N is the number of records.

Secondary organization: *Index organization*. An index is a data structure containing a collection of:

(key k_i, pointer to the record with key k_i)

Using an index we aim to retrieve records with as few accesses as possible. Given a certain file of records, we can have **as many indexes as we need** each with a different search key. Indexes can be **clustered or unclustered** depending on the order of the records:

- If the records of the file are **ordered on the same** key of the index -> index is **clustered** (the index is perfectly ordered, the data is *almost* ordered as there could've been insertions);
- If the records of the file are not sorted on the key of the index -> index is **unclustered**;

As an example, suppose we have a table ordered with respect to the *Code*: an index on *Code* is clustered, an index on any other attribute (*City*, *BirthYear*, etc) is unclustered. Indexes can also be ***dense* or *sparse***:

- ***Dense* index**: the number of elements of the index is equal to the number of records; usually clustered indexes are dense.
- ***Sparse* index**: the number of elements of the index is less than the number of records (for example the index points to the largest record of the page). Usually unclustered indexes are sparse.

As for the **cost of search**, it depends on the clustered/unclustered index chosen: given $N_{rec}(R)$ records, we have:

Unclustered costs: when searching for a record, the cost is always the sum of the cost of accessing the index plus the cost of accessing the data:

- **Unclustered equality search**: $1 + 1$;
- **Unclustered range search**: $C_s = s_f(\psi) \times \left(N_{leaf}(idx) + N_{rec}(R)\right)$

Clustered costs: again, when searching for a record, the cost is always the sum of the cost of accessing the index plus the cost of accessing the data:

- **Clustered equality search**: $1 + 1$;
- **Clustered range search**: $C_s = s_f(\psi) \times \left(N_{leaf}(idx) + N_{pag}(R)\right)$

In both cases we have the selectivity factor defined as:

$$s_f(\psi) = \frac{k_2 - k_1}{k_{max} - k_{min}}$$

And the **number of leaves in the index** defined as:

$$N_{leaf}(idx) = \frac{(L_k + L_{RID}) \times N_{rec}(R)}{D_{pag} \times f_r}$$

In other words, the number of leaves in the index is the number of records of the files multiplied by the length of a single value of the key (L_k) plus the length of the record identifier (L_{RID}); this is divided by the dimension of the page ($\left(D_{pag}\right)$ times the fraction of pages left free for possible insertion (f_r). To sum up, the cost of a range search is almost the same when dealing with clustered and unclustered indexes, but:

$$\text{Clustered} \rightarrow pages$$

$$\text{Unclustered} \rightarrow records$$

As for the **number of records satisfying the equality or range search condition**, the formula is the same as before:

$$E_{rec} = \lceil s_f \times N_{rec}(R) \rceil$$

Observation: when is it convenient to use a clustered or unclustered index? If the interval of a range search is large (20-30% of all the records), it's better to use a **sequential scan** of the file. As a general rule use an index only if:

$$Cost\ of\ range\ search\ with\ c/unc\ index < N_{pag}(R)$$

Organizations based on non-unique keys

To retrieve records of a table that satisfy a query which involves non-key attributes (= attributes that do not have a unique value for each record: different records have the same value for a certain non-key attribute, like the *city* or the *age*).

Inverted indexes. Inverted indexes are used when dealing with queries with conditions on non-key attributes (equality and range searches on on-key attributes, conditions with Boolean operators AND, OR, NOT on non-key attributes), only when the selectivity factor is very small ($< 10\%$) and only if there's an acceptable number of indexes on the table[1]. An example of index on a non-key attribute (*quantity*) is:

Index on Quantity

Quantity	RID
1	5
2	1
2	8
2	9
5	3
5	7
5	10
8	2
8	6
10	4

Sales

RID	Date	Product	City	Quantity
1	20090102	P1	Lucca	2
2	20090102	P2	Carrara	8
3	20090103	P3	Firenze	5
4	20090103	P1	Arezzo	10
5	20090103	P1	Pisa	1
6	20090103	P4	Pisa	8
7	20090103	P2	Massa	5
8	20090104	P2	Massa	2
9	20090105	P4	Massa	2
10	20090103	P4	Livorno	5

An inverted index is a **list of triples**:

- Key value k;
- List of the records with key k (RID list);
- Length of the list (n).

Inverted Index on Quantity

Quantity	n	RID list
1	1	5
2	3	1, 8, 9
5	3	3, 7, 10
8	2	2, 6
10	1	4

As we can see, for every *quantity* we have a list of records identified by that *quantity* and the length of that list n. Note that the inverted list is **sorted on the non-key attribute**, in this case *quantity*. **Advantages**:

- We can access immediately the records matching the queries;
- Queries with COUNT with conditions are satisfied using only the index;
- The organization of the file is independent from the organization of the index.

Cost estimation. Given R, suppose R has $N_{pag}(R)$ pages, $N_{rec}(R)$ records (each sized L_r) and $N_I(R)$ indexes, each with $N_{leaf}(I)$ leaf nodes and $N_{key}(I)$ distinct keys. Then the **total memory occupied by the inverted indexes** is:

$$N_I(R) \times N_{rec}(R) \times L_{RID}$$

[1] Inverted indexes can be created for any non-key attribute *but* too many indexes can deteriorate the overall performance.

...which is the number of indexes on R $N_I(R)$ times the number of records of R $N_{rec}(R)$ times the number of bytes to represent the RID of a record L_{RID}.

Equality search costs are always "cost of accessing the index plus cost of accessing the data". The **cost of accessing the index** is always the same:

$$C_I = s_f(\psi) \times N_{leaf(I)} = \frac{N_{leaf(I)}}{N_{key}(I)}$$

Where $N_{leaf}(I)$ is:

$$N_{leaf}(I) = \frac{N_{rec}(R) \times L_{RID} + N_{key}(I) \times (L_I + L_r)}{D_{pag} \times f_r}$$

The cost of accessing the data is instead estimated using $E_{rec} = s_f(cond) \times N_{rec}(R)$. If the index is clustered then $C_D = N_{pag}(R) \times s_f(cond)$, if the index is unclustered then $C_D = \phi(\mathrm{E_{rec}}, \mathrm{N_{pag}}(R))$ where ϕ is the **Cardenas formula** (the probability that a certain number of records will go in a certain page):

$$\phi(k, n) = n\left(1 - \left(1 - \frac{1}{n}\right)^k\right)$$

For range search, the cost of accessing the index is always the same:

$$C_i = s_f(\psi) \times N_{leaf(I)}$$

The cost of accessing the data is instead the number of keys inside the interval (N-list) multiplied by the number of pages needed to visit each RID list:

$$C_D = N_{List} \times N_{page} to\ visit\ each\ RID\ list$$

Where $N_{list} = N_{key}(A_i) \times s_f(cond)$. If the index is clustered, $N_{page} = \frac{N_{pag}(R)}{N_{key}(A_i)}$, if the index is unclustered $N_{page} = \phi\left(\frac{N_{rec}(R)}{N_{key(A_i)}}, N_{pag}(R)\right)$.

As for **insertion and deletion**, the $N_I(R)$ indexes on a table R must be updated whenever a record is inserted or deleted: the operation requires $N_I(R)$ reads and writes.

Bitmap index. Used when a non-key attribute is used in many different records: for each value of a non-key attribute we insert, in correspondence with the index of a certain record, 1 if the record has that attribute value, 0 if not.

Sales

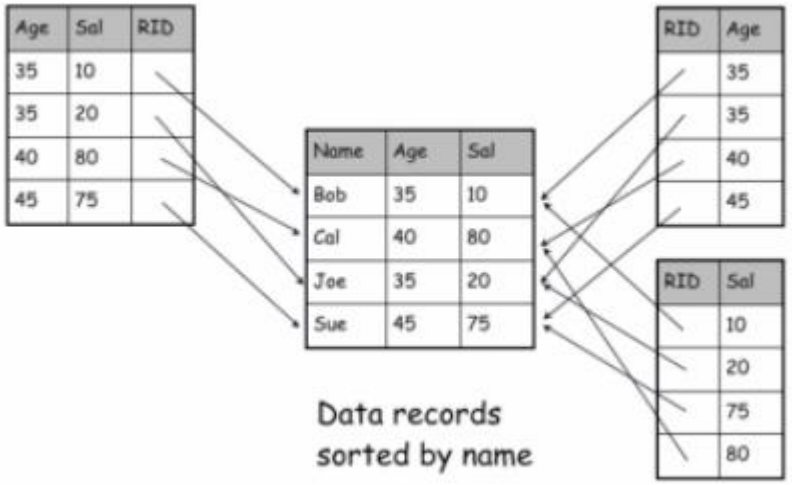
RID	...	Quantity
1	...	2
2	...	8
3	...	5
4	...	10
5	...	1
6	...	8
7	...	5
8	...	2
9	...	2
10	...	5

Quantity bitmap index

1	2	5	8	10
0	1	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	0	0	1
1	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	1	0	0	0
0	1	0	0	0
0	0	1	0	0

Bitmap indexes are useful when the attribute has a low selectivity or when queries perform COUNT on the data (= counting the number of "1" in a column).

Multiattribute indexes. Are indexes built on **more than 1 attribute.** For example, given the following data table, a multi-attribute index on the *Age* and *Salary* is:



Transaction management

A **transaction** is a *unit of work* composed by a **sequence of *reads* and *writes*** that starts with ***beginTransaction*** and *ends* with either ***commit*** or a ***rollback*** *(abort)* operation. Transaction guarantees:

- **Atomicity**: only committed transactions (terminated normally) change the database; when transactions are aborted (system failure, required by the transaction, etc) the state of the database remains unchanged (= like before the transaction).
- **Isolation**: when a transaction is executed concurrently with others, the final effect must be like it was executed alone on the database (**serializability property**: as if transactions were executed one after the other).

- **Durability**: committed transactions survive system failures (commitment is irrevocable).
- **Consistency**: every integrity constraint is satisfied before and after the execution of a transaction (be it committed or rolled-back).

Difference between *commit* and *abort*: commit must be explicitly written by the programmer of the transaction, abort might happen as a side effect (forced by a system failure, indirect consequence of a read/write/commit request, written explicitly by the programmer).

Types of failures. The impact of transaction failures can be:

- **No damage on temporary/permanent memory** when failures depend only on the transaction (***transaction failures***: error in the transaction, transaction performs an operation that is not possible at the moment). Neither the temporary memory nor the permanent memory are damaged (other transactions executed in parallel)
- **Damage to the temporary memory** when failures depend on the system (***system failures***: crash of the system);
- **Damage to both the temporary and the permanent memories** when failures depend on catastrophic events (***hardware or media failure***: everything is lost).

Transactions are managed by the **Transaction and Recovery Manager**. It's responsible for the execution of transactions (*read or write of an entire page*[2], *commit or abort of a transaction*), the management of log file and backup file, the application of techniques to prevent data loss and restart the system after a failure.

Operation: reading a page.

$$r_i[x] \quad i = transaction\ identifier, x = page\ read$$

Meaning: bring page x from the disk to the buffer pool (if it's not already in the buffer pool).

Operation: writing a page.

$$w_i[x] \quad i = transaction\ identifier, x = page\ read$$

Meaning: write page x on the buffer pool (note: x is not written immediately on the disk but only when the buffer manager decides to do it; consequence: the **write is lost in case of system failure**).

Strategies for protection from failures.

- Backup is simply a dump of the DB done on removable devices (every day, every week, etc).
- To reduce the number of operations that must be done to recovery from a system failure we can use **periodic checkpoint**: this way, when there's a

[2] Assumption: we're not working on read/write of single records but of single pages.

failure, the system does not need to read the *whole* log and recover the *whole* backup but only the part since the last periodic checkpoint.

- **Undo-redo management**. To undo the operations performed by an aborted transaction, the DBMS maintains a **log** – a list of the operations performed by every transaction in the DB. The log is also used to recover from system crashes: all **transactions that were "active" at the time of the crash are aborted** once the system is restarted. The log records sequentially the following records:
  - $(T, begin)$
    Transaction with identifier T begins
  - $(T, write,\ Page, oldValue, newValue)$
    Transaction with identifier T performs a write; $Page = oldValue$ is substituted by $Page = newValue$.
  - (T, commit) or (T, abort)
    Transaction with identifier T is committed or is aborted.

Reads are not recorded on the log since they have no impact on the memories of the DB. An example of log is:

Operation	Data	Information in the log
beginTransaction		(*begin*, 1)
$r[A]$	$A = 50$	No record written to the log
$w[A]$	$A = 20$	$(W, 1, A, 50, 20)$ — Old and new value of A are written to the log
$r[B]$	$B = 50$	No record written to the log
$w[B]$	$B = 80$	$(W, 1, B, 50, 80)$
commit		(*commit*, 1)

Each entry of the log is identified by a **LSN** (Log Sequence Number).

Checkpoints. Used to reduce the amount of time of recovery, a checkpoint in a certain point in time means "**write all modification of the buffer pool on the disk**". There are different approaches to perform a checkpoint:

- **Commit consistent CKP**: new transactions are put on hold; all transactions in execution are completed and all pages modified in the buffer pool are written on the disk. When all this is done, we insert in the log a record that says "we've done a checkpoint in this moment". Inefficient (wait for all active transactions to end, so the system is put on hold for potentially a lot of time).
- **Buffer consistent CKP**: new transaction are put on hold; instead of waiting for the end of all the active transaction, we start writing the dirty pages

from buffer to disk, and then we add a checkpoint in the log plus the list of the currently active transactions.

- **Fuzzy CKP (Aries method)**: the flush of pages from buffer to disk is simply executed without putting new transaction on hold or waiting for the currently active transaction to be completed.

Undo-redo procedures. We can perform two operations:

- **Undo**: if an aborted transaction has written a page on the buffer pool and on the disk before being aborted, then we have to *undo* such operation to maintain the consistency of the DB (memory is the same as before the failed transaction);
- **Redo**: if a transaction is committed but there's a system failure right before the pages written on the buffer are saved on the disk, we perform a *redo*.

We have several ***recovery algorithms***: undo-redo, undo-no redo, no undo-redo, no undo-no redo (rarely used).

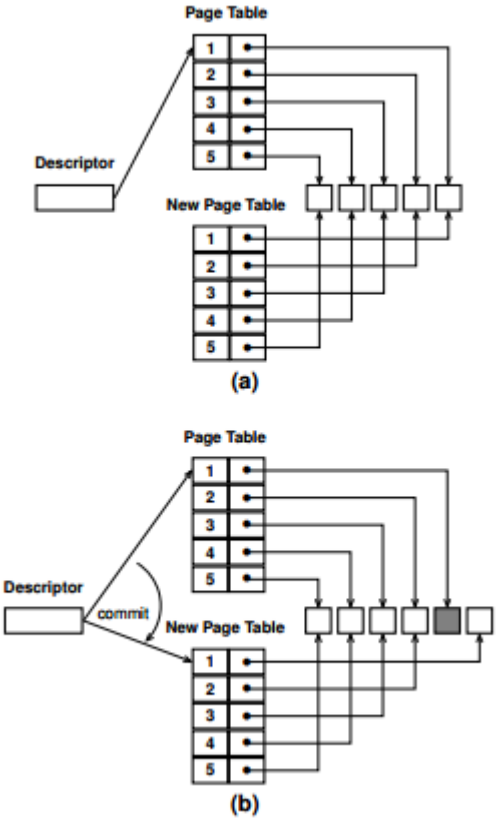
Undo. Suppose a transaction modifies a page: when this modified page is written from the buffer to the disk of the DB? It depends on *when* the modified page is written on the disk: *before* or *after* the commit? There are two possibilities:

- **Deferred update:** the buffer manager writes the modified page on the disk *after* the transaction is committed -> **no need for an undo operation**.
- **Immediate update:** the buffer manager can write the modified page on the disk any time, even before the transaction is committed -> **Undo might be required**. Immediate update can be applied only if we know that **Write Ahead Log (WAL)** policy is applied (before writing the modified page on the disk, the log is updated with the old version of the page).

Redo. When is a transaction considered terminated? Again, we have two possibilities:

- **Deferred update**: a transaction is terminated with success *after* every modified page has been written from the buffer to the disk -> **no need for a redo operation**.
- **Immediate update**: a transaction is terminated with success *before* its modified pages are written from the buffer to the disk -> **redo might be required**. Immediate update can be applied only if we know that the **Commit rule** is applied (the new pages are written in the log before the transaction is considered committed).

NoRedo-noUndo. It's a strategy not really used in practice but still interesting, and it's based on the ***shadow pages*** **technique**. An auxiliary structure, called *Page Table*, is created: it contains a pointer for each page of the DB. When a transaction starts, the Page Table is duplicated into a New Page Table **viewed only by the current transaction**; if a transaction modifies a certain page, instead of modifying the page pointed by its pointer, a new page is allocated and the pointer points to that new page with the new value. Other transactions do not view this modification until the commitment, when the original Page Table is simply substituted with the New Page Table.



Since neither redo nor undo are needed, the **log is not needed** as well.

Conclusion: the most efficient policy is undo-redo.

Recovery procedure. A transaction failure is recorded in the Log as

$$(T, abort) \quad transaction\ T\ is\ aborted$$

Once a transaction failure happens, depending on the failure we have two approaches:

- With a **system failure**, a *warm restart* is executed: from the CKP record, **transaction terminated must be redone** and **transactions not terminated must be undone**.
- With a **disk failure**, a cold restart is executed: using the backup copy of the DB, we redo the committed transactions.

The recovery algorithm happens in two phases: ***rollback*** and ***rollforward***. Given the **empty sets** L_r (transactions to be redone) and L_u (transactions to be undone):

1. **Rollback**: the log is read bottom-up (from the end to the beginning).

- With a $commit$: transaction is added to L_r (no action)
- With a $begin$: transaction is removed from L_u (no action)
- With a $write$: write must be undone (page = the old value)
- With a CKP {transactions}: transactions not belonging to L_r are added to L_u

Rollback populates both L_r and L_u. **When L_u becomes empty rollback is over.**

2. **Rollforward.** The log is read onward from the **first record after the checkpoint** to redo all the operations of the transactions successfully terminated ($T_i \in L_r$).

- With a $commit$: transaction is removed from L_r (no action)
- With a $begin$: transaction is removed from L_u (no action)
- With a $write$: write must be redone (page = the new value)

Rollforward empties L_r (L_u is already empty and not touched). Once L_r is empty rollforward is over.

Example. Suppose we have this log:

Log					
LSN	**Record**	**LSN**	**Record**	**LSN**	**Record**
1	(*begin*, 1)	6	(*begin*, 3)	12	(*begin*, 5)
2	$(W, 1, A, 50, 20)$	7	$(W, 3, A, 20, 30)$	13	$(W, 5, E, 50, 30)$
3	(*begin*, 2)	8	(CKP, $\{2, 3\}$)	14	(*commit*, 2)
4	$(W, 2, B, 10, 20)$	9	$(W, 2, C, 5, 10)$	15	$(W, 3, B, 20, 30)$
5	(*commit*, 1)	10	(*begin*, 4)	16	(*commit*, 4)
		11	$(W, 4, D, 5, 30)$		

To perform *rollback* and *rollforward*, we create a table with the Log Sequence Number, the operation we must analyze, L_r, L_u and the action.

LSN	Operation	Lr	Lu	Action

Rollback: starting from the record with the highest LSN (16 in our case), we have:

LSN	Operation	Lr	Lu	Action
16	(commit, 4)	{4}	{}	-
15	(W,3,B,20,30)	{4}	{}	Undo: B=20
14	(commit, 2)	{4,2}	{}	-

13	(W,5,E,50,30)	{4,2}	{}	Undo: E=50
12	(begin, 5)	{4,2}	{}	-
11	(W,4,D,5,30)	{4,2}	{}	Undo: D=5
10	(begin,4)	{4,2}	{}	-
9	(W,2,C,5,10)	{4,2}	{}	Undo: C=5
8	(CKP, {2,3})	{4,2}	{3}	-
7	(begin,3)	{4,2}	{}	

Since L_u is empty the rollback phase is over. The rollforward is:

LSN	Operation	Lr	Lu	Action
9	(W,2,C,5,10)	{4,2}	{}	Redo: C=10
10	(begin, 4)	{4,2}	{}	-
11	(W,4,D,5,30)	{4,2}	{}	Redo: D=30
12	(begin,5)	{4,2}	{}	-
13	(W,5,E,50,30)	{4,2}	{}	Redo: E=30
14	(commit,2)	{4}	{}	-
15	(W, 3, B, 20, 30)	{4}	{}	Redo: B=30
16	(commit,4)	{}	{}	-

Once L_u is empty, rollforward is done. Of course this operation is more complex if performed at record or field level.

Concurrency management.

Concurrent transactions are interleaved (one operation of the first transaction is executed between two operations of the second transaction, etc): we have better response time but also potential unpredictable results and interferences. To avoid this we could adopt **serial execution of transactions** (one transaction after the other). There's a better solution: using **serializable transactions**. A set of transactions $T = (T_1, T_2, \dots, T_n)$ with read/write operations and terminating with commit/abort is serializable if their interleaved execution has **the same effect of a serial execution** of those transactions. The concurrency manager – also known as *serializer* – is responsible of **scheduling transactions** (ordering their execution) **such that their execution is serializable**: for example, given transactions $T = (T_1, T_2, \dots, T_n)$ with the following operations…

$$T_1 = r_1[x] \quad w_1[x] \quad w_1[y] \quad c_1$$
$$T_2 = r_2[x] \quad w_2[y] \quad c_2$$
$$T_3 = r_3[x] \quad w_3[x] \quad c_3$$

…a valid schedule could be $H_1 = r_1[x]r_2[x]w_1[x]w_2[x]r_3[x]w_3[x]c_3w_2[y]w_1[y]c_2c_1$.

C-Equivalent histories. Operations of different transactions can also be **conflicting** (for example *read-write* and *write-write* over the same data; read-read is *never* in conflict): two schedules H and L are **c(onflict)-equivalent** if:

- H and L are defined over the same set of transactions;
- If the **order of conflicting operations is the same.**

For example, consider the following histories:

T1	T2	T3
r1[x]		
	r2[x]	
w1[x]		
		r3[x]
		w3[x]
		c3
	w2[y]	
w1[y]		
c1		
	c2	

H1

T1	T2	T3
	r2[x]	
	w2[y]	
	c2	
r1[x]		
w1[x]		
		r3[x]
		w3[x]
		c3
w1[y]		
c1		

H2 equivalent to H1

T1	T2	T3
r1[x]		
	r2[x]	
w1[x]		
		r3[x]
		w3[x]
		c3
w1[y]		
	w2[y]	
c1		
	c2	

H3 non-equivalent to H1

These three histories have the same transactions (T_1, T_2, T_3) but only $H1$ and $H2$ are equivalent; in fact, the conflicting operations are in the same order:

$$w_1[x]r_3[x] \quad r_3[x]w_3[x] \quad w_2[y] \, w_1[y]$$

$H3$ does not have this order ($w_1[y]$ is executed before $w_2[y]$) and therefore it's not equivalent to $H1$ or $H2$.

C-serialization. Suppose we have a set of transactions T and a **serial schedule** of them:

T1	T2	T3
	r2[x]	
	w2[y]	
	c2	
r1[x]		
w1[x]		
w1[y]		
c1		
		r3[x]
		w3[x]
		c3

If a schedule H is c-equivalent to the serial schedule, H is $\boldsymbol{c-serializable}$. Note that a c-serializable schedule is always serializable, but not viceversa: **c-serializability is stricter** than the general definition of serializability.

Serialization graph. To check if a history is c-serializable we use the *serialization graph*, a **direct graph** where:

- Node = transaction T_i of H;
- Arc $T_i \rightarrow T_j$ = at least one operation of T_i **precedes and is in conflict with** an operation of T_j. If this happens then T_i and T_j are called *conflicting transactions*.

Example. Suppose we have the following histories, $H2$ and $H3$

T1	T2	T3
	r2[x]	
	w2[y]	
	c2	
r1[x]		
w1[x]		
		r3[x]
		w3[x]
		c3
w1[y]		
c1		

Storia H2

T1	T2	T3
r1[x]		
	r2[x]	
w1[x]		
		r3[x]
		w3[x]
		c3
w1[y]		
	w2[y]	
c1		
	c2	

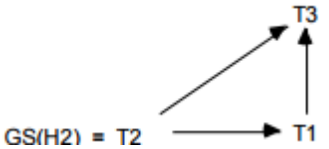
Storia H3

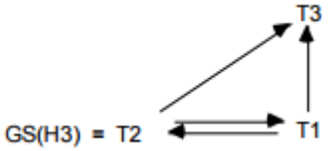
In both cases we have 3 nodes (3 transactions: $T_1, T_2 T_3$). The conflicting operations (read-write or write-write on the same object) of $H2$:

- T2 reads x and T3 writes on x: T2 -> T3;
- T1 reads x and T3 writes on x: T1 -> T3;
- T2 writes on y and T1 writes on y: T2 -> T1.

The conflicting operations of H3 are instead:

- T2 reads x and T1 writes on x: T2 -> T1
- T2 writes on y and T1 writes on y: T2 -> T1
- T2 reads x and T3 writes on x: T2 -> T3
- T1 writes on x and T3 writes on x: T1 -> T3
- T1 reads on x and T3 writes on x: T1 -> T3
- T1 writes on y and T2 writes on y: T1 -> T2





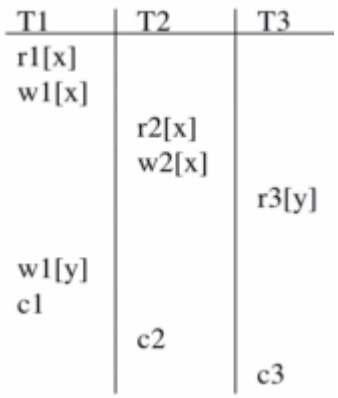
By looking at the serialization graph we can tell if a history is c-serializable: **a history is c-serializable if its serialization graph is acyclic**. A serial schedule can be obtained with a ***topological ordering*** on the graph (for example for H2, which has no loops, we have T2 -> T1 -> T3).

Concurrency control technique: locking. The most used strategy is called *locking* and it's based on the idea of **locking the data when it's used by a transaction**. Suppose a transaction wants to write or read some data: it must acquire a lock on that data containing the *transaction ID*, the *data ID* and the *mode* of the lock (**shared (S)**: the transaction wants to read the data; **exclusive (X)**: the transaction wants to write the data). Once acquired, the transaction can continue its execution (otherwise it's suspended until the lock on the data is obtained). Another transaction can be executed concurrently with this one only if their locks are not conflicting: for example, if T1 has a shared lock on $data$ and T2 wants to read $data$, it can be done; if T1 has an exclusive lock on $data$, instead, T2 is suspended.

These basic ideas are implemented in the ***Strict Two Phase locking protocol***:

1. If T1 wants to read he data, first obtains a shared (S) lock. If T1 wants to write the data, first obtains exclusive (X) lock.
2. Different transactions are executed because their **locks are not in conflict** (locks on the same object and one of them is X lock)
3. Hold all locks until end of transaction (commit), otherwise **isolation between transactions cannot be guaranteed.**

Theorem: if a serializer implements ***strict 2PL schedule***, the resulting schedule is c-serializable but not viceversa: for example this schedule is c-serializable but not 2PL.



T1 must perform a read on data x and requires an S lock on x; that lock is granted (there's no other lock on x); then, T1 requires an exclusive lock on x because it must perform a write; again it's granted (it's just an "upgrade" of the previous lock); next, T2 starts and requires an S lock on x, which cannot

be granted because T1 has an exclusive lock on x. Therefore these two transactions cannot be executed and therefore this is not 2PL but only c-serializable.

Deadlocks. 2PL is simple but **can cause deadlocks** (T1 has locked data A and needs a lock on data B, while at the same time T2 has a lock on B and needs a lock on A). The scheduler needs to detect or prevent deadlocks.

- **Deadlock prevention**: the scheduler checks for classical deadlock situations by looking at the **age of the transactions:**
  - **Wait-die (non preemptive or "older first")**: an older transaction waits a younger transaction, otherwise it's aborted and restarted with the old start time so it's not interrupted anymore.
  - **Wound-wait (preemptive or "younger first")**: a younger transaction waits an older transactions, otherwise the older transaction is aborted and restarted with the old start time so it's not interrupted anymore.

In both cases deadlocks cannot arise.

Example. Suppose we have the following transactions:

$T1 = w[x]\, w[y]$

$T2 = w[y]\, w[x]$

With T1 starting before T2. An execution with deadlock is:

T1	T2
lock-x on x	
w[x]	
	lock-x on y
	w[y]
	wait for lock-x on y
wait for lock-x on y	

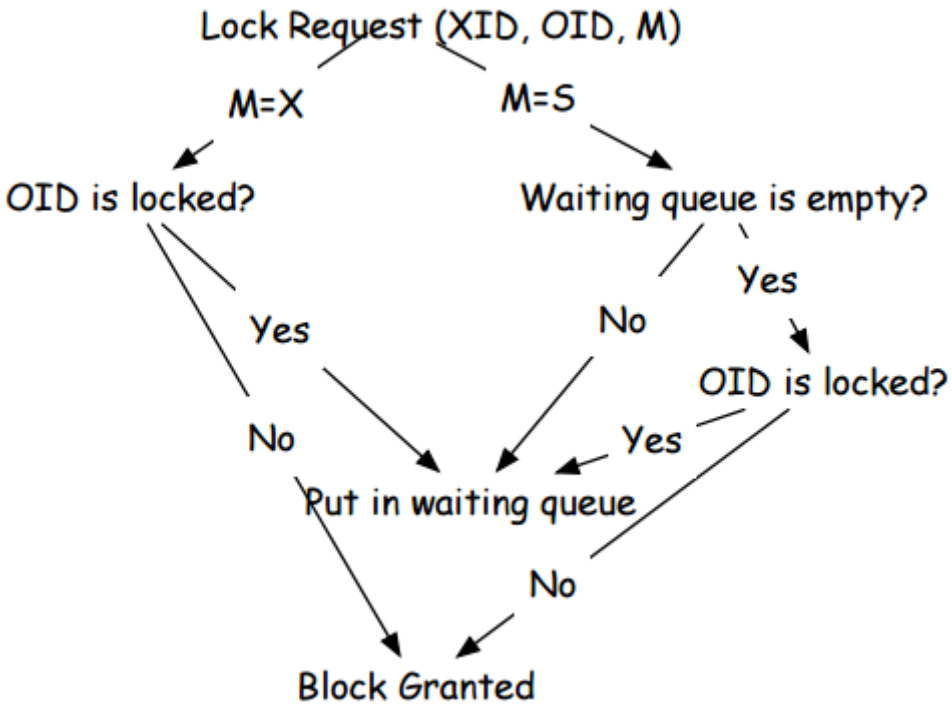
As we can see, after $w[y]$ T2 requires a X-lock on x, which is already locked by T1. T1 requires an X-lock on y, which is locked by T2. We can solve the deadlock depending on the chosen technique:

- *Older first*: T1 waits for T2; since there's a deadlock, T2 is aborted, T1 completes its execution, T2 is restarted with the old time frame (so it cannot be stopped by another transaction);
- *Younger first*: T2 waits T2; since there's a deadlock, T1 is aborted, T2 completes its execution, T1 is restarted with the old time frame (as before).

- **Deadlock detection**: the system checks for a deadlock by **building the *wait-for graph*** where:
    - Nodes: the active transactions;
    - Edges: depend on which transaction is waiting for which (if T_1 is waiting for T_2: T_1 -> T_2).

  When a transaction releases the data needed by T1, the corresponding edge is deleted; if a new transaction requires a lock on an object already locked by another, a new arc is added. A deadlock is present if there's a **loop in the graph**: once a loop is found, a transaction inside the loop is aborted and restarted (the cycle is broken).

Implementation of the concurrency manager (or serializer). The serializer manages **lock transactions** (locks a certain data (S or X), unlocks a certain data, unlock the whole data set); to do so, the serializer needs the ***lock table*** where for each *object identifier* (the data) there's a *set of locks* and a *queue of locking requests*. But what is the **granularity of locks**? Should we lock a field, a record, a page or a file? If the grain is fine then concurrency is increased but the management is expensive; if the grain is coarse then concurrency decreases but the management is simpler. The serializer must also implement **rules for locking**, like this:



Finally, when a transaction releases a lock, the list of locking transactions is updated: the first transaction of the waiting queue for that data is

considered and, if it can be reactivated, it's added to the list of locking transactions; if not, get to another transaction until we can find one.

Concurrency in real systems. In real systems what we've previously seen is too simple. The main feature of real systems is their granularity: locks are reduced as much as possible and are applied at the smallest possible level of granularity. The **containment hierarchy** of the granularity is:

DB -> Files (tables) -> Pages -> Records -> Fields

In the containment hierarchy we can have either low (fields) or high (DB) lock granularity , with the same pros and cons as before; note that if we lock a record, we lock every field of that record too (a locked object has all its components locked too). To lock some part of an object, an intention lock on the whole object is required: intention lock can be *shared*, *exclusive* or *shared-exclusive* (to read some parts and write some other parts of some object).

Exercise 3.3 – Heap organization

Let $R(K, A, B, other)$ be a relation with $N_{rec}(R) = 100'000$, a key K with integer values in the range $(1{,}100'000)$ and the attribute A with integer values uniformly distributed in the range $(1{,}1000)$. The size of a record of R is $L_r = 100\ bytes$, Suppose R is stored with a heap organization, with data unsorted both respect to A and K in pages of size $D_{pag} = 1024\ bytes$. Estimate the cost of the following SQL queries and consider for each of them the cases that *Attribute* is K or A, and assume there are always records that satisfy the condition.

Query #1.

```
SELECT *
FROM R
WHERE Attribute = 50;
```

This query return all the records whose attribute *Attribute* equals to 50. If the $Attribute = K$ then the query returns a **single key**; the cost of an equality search into a heap file is:

$$Equality\ search\text{:}\quad C_s = \left\lceil \frac{N_{pag}(R)}{2} \right\rceil$$

Now, $N_{pag}(R) = N_{rec}(R) \times L_r \,/\, D_{pag} = 100'000 \times \frac{100}{1024} = 9765$, and thus $\frac{9765}{2} = 4882$. If $Attribute = A$ then the query will return at least 1 result, and the cost of an equality search is:

$$Equality\ search\text{:}\quad C_s = N_{pag}(R)$$

…which is $N_{pag}(R) = N_{rec}(R) \times L_r \,/\, D_{pag} = 100'000 \times \frac{100}{1024} = 9765$.

Query #2.

```
SELECT *
FROM R
WHERE Attribute BETWEEN 50 AND 100;
```

This query returns all the records whose attribute value is between 50 and 100, so it's a range search. If the $Attribute = K$ or the $Attribute = A$ the cost of a range search is the same:

$$Range\ search\text{:}\quad C_s = N_{pag}(R)$$

…which is $N_{pag}(R) = N_{rec}(R) \times L_r \,/\, D_{pag} = 100'000 \times \frac{100}{1024} = 9765$.

Query #3.

```
SELECT Attribute
FROM R
WHERE Attribute = 50
ORDER BY Attribute;
```

In this case we have an ORDER BY clause. Since the ORDER BY is on the attribute we're using for the search, the ORDER BY clause is **not significant** and therefore the cost of this query is the same as the first query.

Query #4.

```
SELECT *
FROM R
WHERE Attribute BETWEEN 50 AND 100
ORDER BY Attribute;
```

This query is the same as the second query but with an ORDER BY clause. Since we need to return the attribute ordered, the only way we can produce an answer for this query is to apply a sorting algorithm over a temporary file where the not-yet-ordered results are stored. The overall cost will be the sum of:

1. The cost of the range search;
2. The cost of writing on a temporary file the results of step 1;
3. The sorting on the temporary file;
4. The cost of reading the temporary file once it's been sorted.

Let's compute these terms:

1. The cost of a range search is $N_{pag}(R) = N_{rec}(R) \times L_r \,/\, D_{pag} = 100'000 \times \frac{100}{1024} = 9765$;
2. The cost of writing the on the temporary file is the number of pages written on the temporary file: $N'_{rec}(R)$. Its value depends on the **number of records satisfying the condition** "BETWEEN 50 AND 100", E_{rec}:

$$E_{rec} = N_{rec}(R) \times s_f$$

   - If $Attribute = K$ we have $s_f = \frac{100-50}{100000-1}$ and therefore $E_{rec} = 100'000 \times \frac{100-50}{100000-1} = 51$. To find the number of pages we use the previous formula:

$$N'_{pag}(R) = N_{rec}(R) \times \frac{L_r}{D_{pag}} = 51 \times \frac{100}{1024} = 5$$

   - If $Attribute = A$, we have $s_f = \frac{100-50}{1000-1}$ and therefore $E_{rec} = 100'000 \times \frac{100-50}{1000-1} = 5006$. To find the number of pages we use the previous formula:

$$N'_{pag}(R) = N_{rec}(R) \times \frac{L_r}{D_{pag}} = 5006 \times \frac{100}{1024} = 489$$

3. Sorting the temporary file, assuming we apply the sort-merge algorithm and we can use a certain number of buffers, then the cost of sorting is $4 \times N'_{page}(R)$.
4. The cost of reading the file once it's been sorted is again the number of pages satisfying the ranged search: $N'_{page}(R)$.