1. ANSI-SPARC architecture:
   1. Conceptual structure (schema) - just 1
   2. External structure**s** (ex. views) - there can be several
   3. Physical (internal) structure (data files, indexes etc.) - just 1
   4. No such thing as a *symbolic* structure
2. **Data independence** -
   1. Logical = apps + external schemas are not affected by changes in the *conceptual structure*
   2. Physical = - | | - *physical structure*
3. From the simplest to the most complex: field, record, table, database
4. **Data description model** can be used to describe
   1. the structures of the data
   2. relationships with other data
   3. consistency constraints
5. The rows in a **relational** table are not ordered, the records are distinct - but DBMSs allow duplicates
6. Degree of a relationship set = the no. of entity sets the participate in that rel
   1. The degree of a projection < .. of the original table
7. **Optimizer** - produces an efficient execution plan for query eval, based on storage information
8. Multiple candidate keys can be declared using UNIQUE; one of them can be chosen as the primary key
9. We can have a multiple FK that references a multiple PK
10. FK can be null
11. An att can be a PK and FK at the same time
12. **Key** = a restriction defined on an entity set; a set of attributes with distinct values in the entity’s set instances
13. **Integrity constraints** = conditions specified on the DB schema, restricting the data that can be stored in the DB (includes key constraints, FK constraints)
    1. **Key constraint** = a constraint stating that a minimal subset of atts in a relation is a unique identifier for every tuple in the relation
    2. **Superkey** = set of fields that contains the key
14. **3-valued logic** (true, false, unknown = null)
15. “expression = ANY (subquery)” ⇔ “expression IN (subquery)”
16. “Expression <> ALL (subquery)” ⇔ “expression NOT IN (subquery)”
17. Copy data from one table to another:

“INSERT INTO T2; SELECT \* FROM T1”

1. SELECT can contain arithmetic operations
2. SELECT, FROM - mandatory; HAVING - optional
3. **HAVING** cannot contain row-level conditions because it works with groups
4. Patterns:
   1. "%" - 0 or more arbitrary characters
   2. "\_" - any one character (used with LIKE)
5. **Can’t group** by a subquery
6. SELECT **R.\*** ⇔ select all the columns in R, ex.

SELECT **R.\***

FROM R

RIGHT JOIN whatever ON whatever

1. LEFT JOIN = inner join + the left operands which don’t match the condition (their columns are filled with NULL); same idea for RIGHT JOIN
2. If we have a **FROM query**, we need to give it an alias
3. Processing order:

from > where > group by > having > select > distinct > order by > top

1. **Functional dependency**:

left side = **determinant**; right side = **dependent**

1. Functional dep properties:
   1. K - key of R, Ꞵ - key of a subset of R => K -> Ꞵ
   2. Ꞵ ⊆ 𝛼 => 𝛼 -> Ꞵ (trivial func. dep., reflexivity)
   3. 𝛼 -> Ꞵ => Ɣ -> Ꞵ, for any Ɣ : 𝛼 ⊂ Ɣ
   4. a -> b and b -> c => a -> c (transitivity)
   5. a -> b => a u c -> b u c (augmentation)
2. **Prime attribute** if it’s included in a key K; non-prime otherwise
3. B is **fully functionally dependent** on A if
   * 1. A -> B
     2. B doesn’t depend on any proper subset of A
4. **Multivalued dependencies** - each value from the determinant is associated with a set of values, not with just one, denoted by “”
5. **Join dependency** (JD) - we project an R[A] onto R1[𝛼1] … Rn[𝛼n] => R satisfies the JD {𝛼1, …, 𝛼n} if R = R1 \* … \* Rn
6. **Normal forms**: (1NF ⊂ 2NF ⊂ 3NF ⊂ BCNF ⊂ 4NF ⊂ 5NF)
   1. **1NF** = no repeating attributes
   2. **2NF** = 1NF + trb **toata** cheia sa determine fiecare coloana non-prime (daca cheia e nu putem avea , trb )
   3. **3NF** = 2NF + no non-prime attr depends on other non-prime attr
   4. **BCNF** = every determinant (left) is a key (cea mai ok forma)
   5. **4NF** if for every multi-valued dep , either one is true
      1. b ⊆ a (trivial) or a U b = R
      2. a – superkey (caz particular de multi-valued 🡪 FD)
   6. **5NF** = every non-trivial join dependency (JD) is implied by the candidate keys in R
      1. A JD \*{𝛼1, …, 𝛼n} is trivial if any 𝛼i is the set of all atts of R
      2. … is implied by the candidate keys in R if ALL 𝛼i are superkeys in R
7. **Armstrong’s axioms:** reflexivity, augmentation, transitivity
8. Rules derived from AA
   1. a -> b and a -> c => a -> b u c (union)
   2. a -> b u c => a -> b and a -> c (decomposition)
   3. a -> b and b u c -> d => a u c -> d (pseudotransitivity)
9. **𝛼+ = closure of 𝛼 under a set of FDs**
10. **Example: Show that some FDs are in 𝛼+**

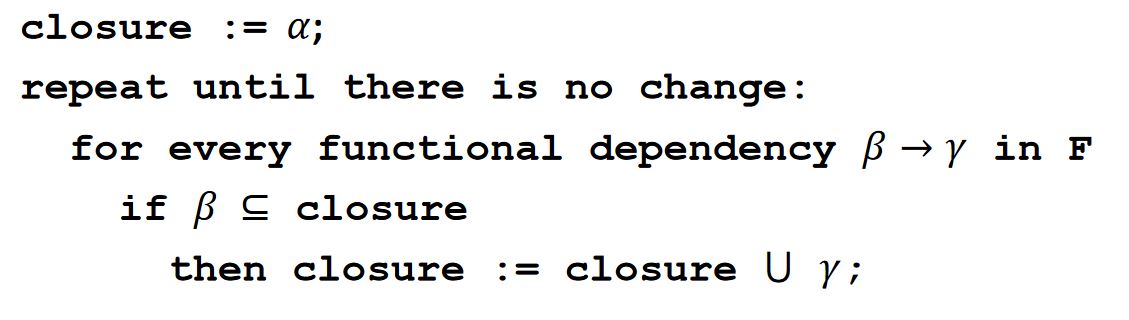
We have a relation R = {a … f} and 𝛼 = {a -> b, a -> c, cd -> e, cd -> f, d -> e}. Show that the following fds are in 𝛼+

*a -> bc*: union of a -> b and a -> c

*cd -> ef*: union of cd -> e and cd -> f

*ad -> e*: a -> c / augment with d => ad -> cd; from cd -> e through transitivity => ad -> e

*ad -> f*: pseudotransitivity a -> c and cd -> f

1. Algorithm for computing 𝛼+ 
2. **Example: Compute 𝛼+**

R = {a … f} and F = {a -> b, a -> c, cd -> e, cd -> f, d -> e}

𝛼 = {a, d} (given)

𝛼+ = {a, d}

a -> b => 𝛼+ = {a, b, d}

a -> c => 𝛼+ = {a, b, c, d}

cd -> e => 𝛼+ = {a, b, c, d, e} (because both c, d were in 𝛼+)

cd -> f => 𝛼+ = {a, b, c, d, e, f} (same reason)

d -> e => we don’t add it because E already is in 𝛼 => no change at this step => stop => 𝛼+ = R

1. F, G - sets of fds => F ≡ G (equivalent) if F+ = G+
2. **Minimal cover** Fm if
   1. Fm ≡ F
   2. Right side of every dependency in Fm has a single att
   3. Left side .. is irreductible (if we remove an att from it, we modify Fm’s closure)
   4. No dependency in Fm is redundant (if we remove it, we modify the closure)
3. **Example: compute Fm**

R = {a … d}; F = {a -> bc, b -> c, a -> b, ab -> c, ac -> d}

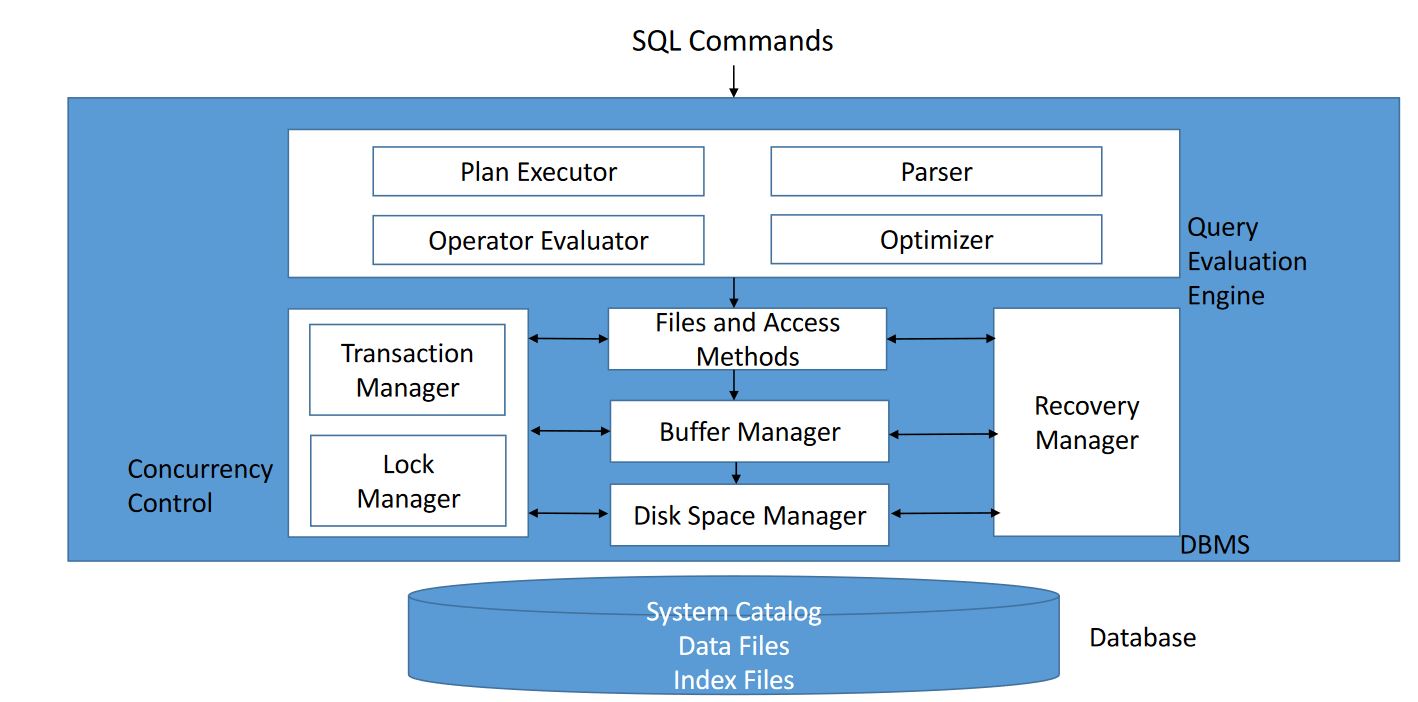
Use decomposition on a -> bc => a -> c, a -> b (which already was in, so we don’t add it again) => **{a->b, a->c, b->c, ab->c, ac->d}**

a -> c / augment with a => aa -> ac => a -> ac; we know ac -> d => a -> d (transitivity); now ac->d becomes redundant, we replace it with a -> d => **{a->b, a->c, b->c, ab->c, a->d}**

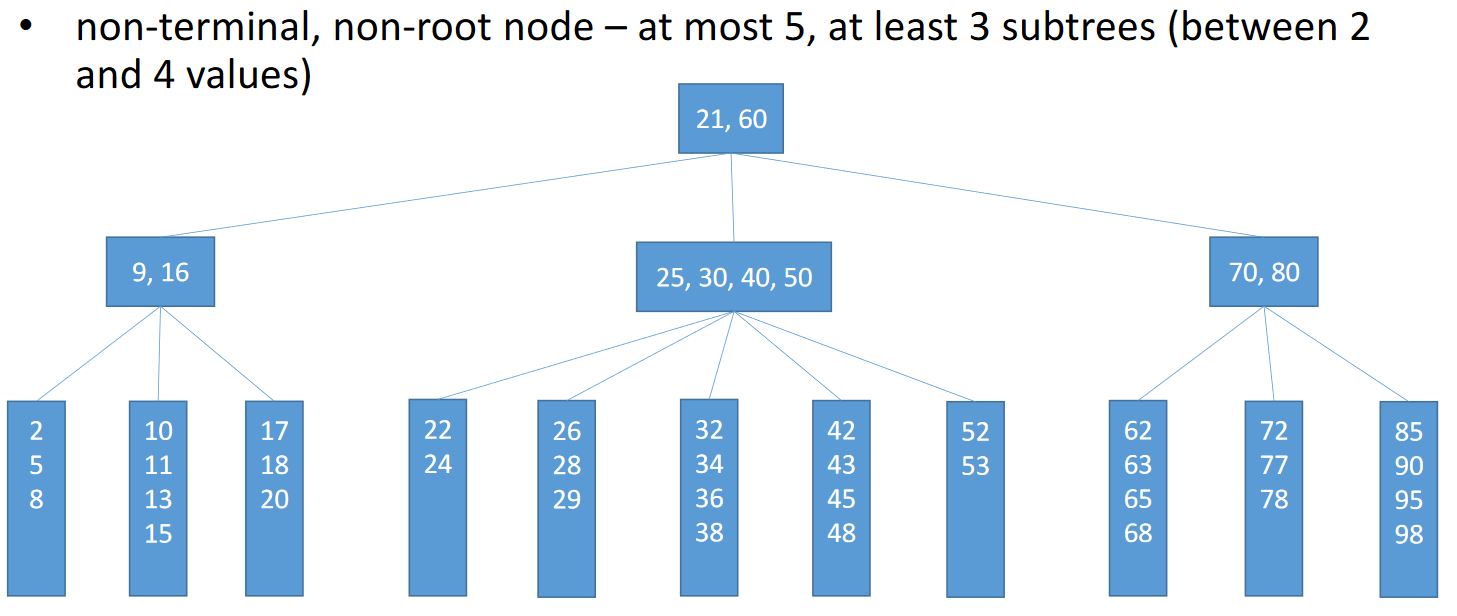
a -> c / augment with b => ab -> cb => (decomposition) ab -> c (it’s already in, we can remove it) => **{a->b, a->c, b->c, a->d}**

a -> c can be obtained as a -> b and b -> c (transitivity) => remove b -> c

Fm = **{a->b, a->c, a->d}**

1. Just by looking at an instance, we can **only** say that a func. dep is **satisfied** or not, we can’t conclude that it’s **specified on the schema**
2. Algebra symbols
   1. 𝜎𝐶(R) = select \* from R where C; distributive over set diff, intersection, union
   2. πlist\_of\_cols(R) = projection = select 𝛼 from R (𝛼 can specify multiple columns); it’s distributive only over **union**
   3. R1 X R2 = select \* from R1 cross join R2 (cross product); the schema will contain all the atts of R1 followed by the atts of R2
   4. R1 U R2, R1 - R2, R1 ∩ R2 = union, set diff, intersection
   5. = .. inner join on Θ (condition join)
   6. R1 \* R2 = natural join; schema will contain the union of the atts (= all atts, common ones appear **only once**); returns 1 relation instance
   7. R1 ⋉c R2 = left (outer) join on c; the reversed symbol is right (outer) join..
   8. R1 ⋈c R2 = full (outer) join on c; schema will contain all the atts of R1 followed by the atts of R2
   9. R1⊳R2 = left semi join; the reverse is right semi join
      1. Left.. = the tuples in R1 that are used in the natural join
      2. right.. = ..R2..
   10. R1 ➗ R2 = division
       1. = the tuples in R1 that are associated with all tuples in R2
   11. 𝛿 = duplicate elimination
   12. S {list} (R) = sort
   13. 𝛾{𝑙𝑖𝑠𝑡1} 𝑔𝑟𝑜𝑢𝑝 𝑏𝑦 {𝑙𝑖𝑠𝑡2} (R)
   14. 𝜌 = renaming operator (nush ce plm e cu asta)
3. All the operators (without duplicate elim, sort, group by) can be obtained from {𝜎,𝜋, ×, ∪, −} (= *independent set of operators*) (see [l7, p3](https://moodle.cs.ubbcluj.ro/pluginfile.php/31115/mod_resource/content/1/DB_CSEn_lecture7.pdf#page=3) on how to do it)
4. If there are multiple triggers defined for the same action, they are executed in a random order
5. DBMS structure 
6. DBMSs operate on data when it is in memory
7. **Seek time** = time required to move the disk head to the desired track (smaller platter size => decreased seek time)
8. **Disk access time** (magnetic disk) =

seek time + rotational delay + transfer time

1. **Block** = unit of data transfer between disk and main memory
2. **DSM** = disk space manager
   1. monitors disk usage
   2. it commands to (de)allocate, read / write a page
3. **BM = buffer manager**
   1. brings new data pages from disk to main memory as they are required + manages the available main memory
   2. **Replacement policies** - LRU (least recently used), MRU, random, clock replacement, toss-immediate
   3. **Buffer pool** (BP) is made of frames, which can fit a page
   4. Each frame has
      1. *pin\_count* = no. of current users on the crt frame; only those with 0 can be replaced; initial value = 0
      2. *dirty* = true if the page has been changed since it was brought to the frame; initially off
   5. When a page is requested
      1. If the page is in the BP, increase its pin count => done
      2. Else, select a frame (FR) to replace using the replacement policies; if the old frame was dirty, write it on the disk; pin\_count(FR)++, the new page is read by the BM in that frame
      3. If the BP is full, the operation may be aborted, or it waits
   6. It may pre-fetch pages
4. **Heap files**
   1. Simplest file structure, unordered records, best when we need to scan all the records
   2. Can use
      1. Doubly-linked list - it needs a header = stored by the dbms; it has 2 lists, for full / free pages
      2. Directory of pages
5. **Index**
   1. Stored on the disk, associated with a table or view
   2. Speeds up equality / range select queries on the search key
   3. If we change the data => the indexes need to be updated
   4. hashed files - very good when searching for equality
   5. **Data entry is:**
      1. A1 - the actual data record with search key value = k
      2. A2 - <k, rid>, rid = id of a data record with sk = k
      3. A3 - <k, list of rid> …
      4. In general, a2, a3 are smaller than a1
   6. **Clustered**
      1. the order of the data records is close to / same as the order of the data entries
      2. A1 - always clustered🡺cl. idxIncludes all the columns
      3. At most 1 / collection of records
      4. When creating a PK, if there’s no clus. on that table (and we don’t specify unclustered) => create a unique cl. Index
      5. Organized as a B+ tree
   7. **Unclustered**:
      1. A2, A3 – unclus (clus. only if the data records are ordered on the search key, which is uncommon)
   8. **Primary index** - contains the PK
   9. **Secondary** **index** = not primary; it can be unclus
   10. **Unique index** = contains a candidate key
   11. Duplicates = with same search key
       1. Primary / unique can’t contain duplicates
       2. Secondary can
   12. **covering index** = contains all the columns that are necessary in a query
   13. **sparse index** = has entries only for some values of the search key
   14. **dense index =** has one entry for every search key value
   15. **key columns** = columns in the search key
   16. **non-key columns** = columns specified in the INCLUDE clause when creating a non-clustered index
   17. **Composite SK** - contains several fields
   18. **Covering index** - contains all the columns that are necessary in a query
   19. **Filtered index** - can be created with a WHERE => only used when it satisfies that condition
6. **ISAM** = Indexed Sequential Access Method
   1. Not very good when doing lots of inserts / deletes 🡺 suitable when data size / distribution are relatively static
   2. Search complexity: logcm (c = children per index page, m = primary leaf pages)
7. **2-3 tree**
   1. All terminal nodes on the same level
   2. Values in the tree are distinct
   3. If a non-terminal node has 1 key => 2 subtrees:
      1. one with values < key
      2. the other > key
   4. If 2 keys (key1 < key2) => 3 subtrees:
      1. first < key1 < second < key2 < third
   5. insert:
      1. add value in corresponding node
      2. if 3 values, middle value goes to parent (and so on…)
   6. remove:
      1. replace with neighbour value
      2. if current is emptied
         1. if sibling has 2 values:
            1. gives 1 to parent
            2. parent gives 1 to emptied node
         2. else (sibling has 1 value)
            1. crt node merged with sibling + 1 val from parent
            2. continue for parent
8. **B-tree of order m**
   1. Generalization of 2-3 tree (= b-tree of order 3)
   2. All terminal nodes on the same level
   3. Every non-terminal, non-root node has between ceil and m **subtrees 🡺** at most m-1 values
   4. A node with p subtrees has p-1 ordered **values**
   5. 
   6. A block stores a node from a B-tree
9. **B+ tree**
   1. Variant of the B-tree:
      1. all keys are in terminal nodes (some of them appear in non-terminal to guide search)
      2. terminal nodes are maintained in a doubly linked list
   2. BENEFITS
      1. balanced index 🡺 Uniform search time
      2. top levels can be kept in main memory 🡺

few I/O ops needed for searching

* + 1. ideal for range selection, good for equality selections as well
  1. Much better than ISAM for insert / delete etc.

1. **Hash-based indexing** - ideal for equality selections
2. **Static hashing** - the classic one (SDA gen)
3. **Extendible hashing**
   1. *directory* which points to the buckets
   2. each bucket determined by the last k (= *depth)* digits of the values in base 2
   3. when a bucket becomes full, we split it into 2 buckets determined by the last k + 1 digits and we double the size of the directory
4. The client generates SQL statements and sends them to the server, which syntactically analyses + evaluates them and sends the results back
5. **Table scan** - high transfer time for large tables (the slowest)
6. **Index seek** – when we search for a key value k0 (k = k0) that we have indexed (the fastest); search can be:
   * 1. explicit (searching a table, evaluating a join)
     2. implicit (checking a key constraint)
7. **Index scan**
   1. the search condition can depend on
      1. a key attr: ineq./IS (NOT) NULL
      2. a non-key attr: eq. / ineq. / IS (NOT) NULL
   2. some mem. blocks may be read multiple times
8. Joins are used in practice more often than cross-products. The sooner we apply selections / projections in the join implementation, the better
9. **Cross-join**

foreach **n1** blocks in **R1**

read in memory the set **M1** of records

foreach **n2** blocks in **R2**

read in memory the set **M2** of records

foreach **r1** in **M1**

foreach **r2** in **M2**

add **(r1,r2)** to the result

1. Indexed nested loop join

foreach block in **R1**

read into main memory the set **M1** of records

foreach **r1** ∈ **M1**

determine **v**

use the *index* from **R2** on **B** to det the set records **r2** ∈ **R2**

with value **v** for **B**

foreach such record **s**

add **(r,s)** to the result

1. Merge join

* Sort the 2 tables based on the columns in the join condition
* Perform a merge (interclasare gen):

let r in R and s in S be 2 current records

if r.A = s.B

add (r', s') to the result

next(r)

next(s)

if r.A < s.B

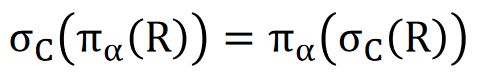
next(r)

if r.A > s.B

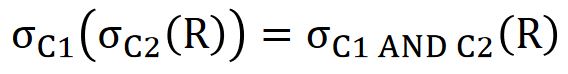
next(s)

A diagram of a diagram

AI-generated content may be incorrect.

1. hash join
   1. hash R and S on the join column
   2. tuples in partition Rx are compared only with tuples in partition Sx
2. **Algebra equivalences**
   1. 

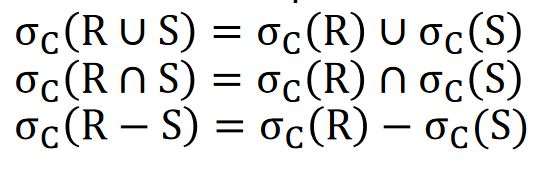
(second is more efficient)

* 1. 

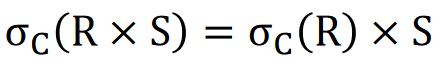
(second - more eff because it only does one pass of R)

* 1. 

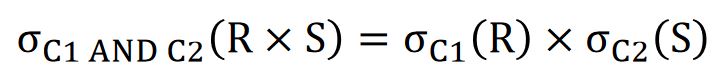
(condition join - more eff)

* 1. 

(if the schemas of R and S are compatible)

* 1. 

**(if C contains only atts from R)**

* 1. 

**(if C1 contains only atts from R, C2 .. S)**

* 1. 

**(if C2 is a join condition between R and S)**

* 1. 
  2. 

(𝛼1 = atts from R that appear in 𝛼 or C, 𝛼2 = … from S…)

1. Intersection and union are **associative** and **commutative**
2. Cross-product and natural join are **associative**
3. When using the cross-join algorithm, the order of the data sources is important
4. **Surrogate key** = key that isn’t obtained from the domain of the modeled problem
5. **DB design stages**
   1. Requirement analysis
   2. **Conceptual design** = high level description of data and integrity constraints
   3. **Logical design** = translate the conceptual DB design to a DB schema in terms of the model supported by the DBMS
   4. **Schema refinement** (normalization, eliminate redundancy)
   5. **Physical design** (ex. indexes)
6. We can have tables that reference themselves (recursive relationship); if there we have ON DELETE CASCADE => error; same if we have two tables that reference each other
7. The M:M table is also called *join table*
8. **Mapping inheritance**:
   1. A1: 1 table / class; subclasses have only their specific columns
   2. A2: 1 table for superclass, with all columns from all subclasses
   3. A3: 1 table / subclass, with their columns + the ones from super
9. **Reduce fragmentation**
   1. In a heap - create + drop cl. index
   2. In an index
      1. ALTER INDEX REORGANIZE (if small fragmentation)
      2. ALTER INDEX REBUILD (if large fragmentation)

OR

drop + recreate index

* + 1. make an index with a fill factor

1. **Indexed views**
   1. cannot reference other views
   2. the index must be clus + unique;
   3. only allowed aggregation operators are sum, count
   4. subqueries, outer joins, set operations etc are not allowed
   5. 
2. WAITFOR TIME ‘4:37’ => execution continues at 4:37
3. Traditional dbms:
   1. *one-shot query* (executed on the current instance of the data)
   2. *Human active, DBMS passive model (HADP)*
4. **Data stream**
   1. temporal sequence of values produced by a data source
   2. potentially infinite
   3. data is associated with timestamps
   4. we can have data **stream** management sys
5. **Event**
   1. elementary unit of information on a data stream
   2. the equivalent of a record
6. **Sliding window** 
   1. contiguous portion of the data stream
   2. it has a size and a step size
7. **Continuous query**
   1. perpetually running
   2. DAHP (= db active, human passive)
8. **Data Stream Management System**
   1. processes streams of data in a perpetual manner
   2. built around a query processing engine, which performs data manipulation operations

|  |  |  |
| --- | --- | --- |
|  | DBMS | DSMS |
| Permanent elems | Data | Continuous queries |
| Temporary elems | Queries | Data on streams |