INTRODUCTION

A quick disclaimer before reading the notes: they were taken by me during the lectures, they do not replace the professor's work and are not sufficient for passing the exams. Moreover they might contain mistakes, so please double check all that you read. The notes are freely readable and can be shared (always remembering to credit me and to not obscure this page), but **can't** be modified. Thank you and hope these notes are useful!

-Francesca Cinelli

DMA V1

Introduction to databases and database management systems

↳ collection of related data items within a specific business process or problem setting

↳ software package used to define, create, use, and maintain a database

↳ database system

→ database approach: all data stored, managed centrally by DBMS

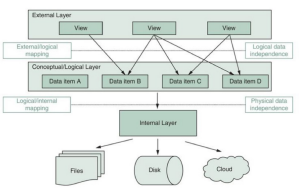
→ raw data + metadata
↳ data definitions that are stored in the catalog of DBMS

Database model / schema
- description of DB data at different levels of detail and specifies the various data items, relationships ecc.
- specified during design and doesn't change frequently
- stored in catalog (heart of DBMS)

Database state / instance
- data in DB at particular moment
- changes

Models
→ conceptual model: high-level description of data items with their characteristics and relationships
→ logical model: translation / mapping of the conceptual model toward a specific implementation environment
↳ relational
→ internal model: mapping of ↑ that represents the data's physical storage details
→ external model: subsets of data items in logical model, views, tailored toward the needs of specific applications or groups of users
↓ hides the rest of DB

3-layer architecture



Catalog
- contains data definitions / metadata of DB app.
- stores definitions of views, logical and internal data models and synchronizes these three data models to ensure their consistency

Database languages
→ data definition language (DDL) expresses external, logical and internal schemas } SQL
→ data manipulation language (DML) retrieve, insert, delete, modify data

Information
- structured: objects represented by short strings of symbols and numbers
- unstructured: texts written in a natural language
- semi-structured: have a structure but irregular / volatile

Information system
a set of data physically organized in secondary memory and managed in such a way as to allow its | creation, updating and interrogation
in homogeneous aggregates

Data independence
- physical - from logical and external
- logical - from external

Advantages of DB
-> no redundancy
-> integrity - correctness of data, constraints have to be satisfied
-> security - protect data from unauthorized access
-> concurrency control
  └ transaction = sequence of operations constituting a single logical transaction
    └ ACID
      · atomicity : all-or-nothing property
      · consistency: DB from one consistent state to another
      · isolation: effect of concurrent transactions should be the same as if they has been executed in i.
      · durability: the DB changes made by a transaction declared successful can be made permanent in all circumstances
-> backup and recovery - transaction log or dump

Performance indicators
- response time : time between a request and its termination
- throughput rate: transactions processed per unit of time
- space utilization: space used to store raw data + metadata

Relational model
- data + relationships = values
- no explicit pointers

object = record / tuple
attribute = info of interest

table = set of records of homogeneous type
r = instances ≠ R = schemas (to write qries)

Definitions
- domain : a set of values (columns of table)
- cartesian product: $D_1 \times D_2 \times \dots \times D_k$ the set $\{(v_1, v_2, \dots, v_k) \mid v_1 \in D_1, v_2 \in D_2, \dots, v_k \in D_k\}$
- mathematical relation: subset of ↑ of k domains (relation of degree k)
  - elements = tuples / ennuples
  - # tuples = cardinality
  - t[i] = i^{th} component of tuple t
- attribute = name + domain (set of attributes = schema)
- NULL values = lack of info or info not applicable, polymorphic value (doesn't belong to any domain but can replace values in any domain), 2 null values are different, null ≠ 0
- intra-relational constraints = defined on single attribute values or between values of same tuple or between tuples of the same relation
- inter-relational constraints = defined between multiple relations
- key = attribute / set of attributes that uniquely identifies a tuple -> if $t_1[x] \neq t_2[x]$
- primary key = most used key / one with smaller # of attributes -> can never be NULL
- foreign key = if is p.key in other relation
- functional dependencies = semantic link between 2 sets of attributes x and y belonging to R, $(X \rightarrow Y)$ (X: subsets of R, →: determines) $\rightarrow t_1[x]=t_2[x] \Rightarrow t_1[y]=t_2[y]$

Relational algebra 1

> notation for specifying queries about the contents of relations
> formal language to interrogate a relational DB, consisting of a set of unary and binary operators that if applied to one or two relation instances, generate a new relation instance
> procedural language - describes exact order in which operators have to be applied to relational instance to obtain the desired result
> 4 types of operators (single, set, combine, renaming) that can also be combined with eachother

Operators

1. Projection (single)
   - vertical cut on a relation, selects a subset of attributes
   - $\pi_{A_1, A_2, \ldots, A_k}(R)$
   - select the columns of R corresponding to attributes $A_1, A_2, \ldots, A_k$
   - ex. $\pi_{name}(Customer)$ -> | Rossi
Bianchi
Verdi |
   - no duplicates in result (might be loss of tuples)

2. Selection (single)
   - horizontal cut on the relation, selects all rows that meet some constraints
   - $\sigma_C(R)$
   - select the tuples of R that meet condition C
     - ↳ Boolean expression whose terms are in the form $A \theta B$ or $A \theta 'a'$
       - ↳ item in dom(A)
       - θ = comparison operator
       - A, B ⇝ attributes same domain
   - no loss of tuples

3. Union (set)
   - creates a new relation instance containing all tuples belonging to at least one of the operand instances
   - $R_1 \cup R_2$
   - performed on union compatible operands only
     - ↳ have same # attributes, corresponding attributes have same domain
   - can use projection first to make them compatible to union
   - is commutative

4. Difference (set)
   - creates a relation containing the tuples in the first operand relation but not in the second
   - $R_1 - R_2$
   - applied only to union compatible operands
   - not commutative

5. Intersection (set)
   - creates a relation containing the items that are present in both the operand relations
   - $R_1 \cap R_2 = (R_1 - (R_1 - R_2))$
   - applied only to union compatible operands
   - is commutative

Relational algebra 2

6. Cartesian product (combine)
   - creates a relation with tuples obtained by combining all the tuples in the first relation with all the tuples in the second relation
   - $r_1 \times r_2$
   - used when info needed is contained in multiple relations
   - associative and commutative

7. Natural join (combine)
   - selects the tuples in the result of the Cartesian product that satisfy the condition:
     * $R_1.A_1 = R_2.A_1 \wedge R_1.A_2 = R_2.A_2 \wedge \dots \wedge R_1.A_K = R_2.A_K$

     where $A_1, A_2 \dots A_K$ -> attributes in common, so duplicate attributes are dropped
   - $r_1 \bowtie r_2 = \pi_{xy}(\sigma_c(r_1 \times r_2))$
     - x: attributes r_1
     - y: attributes r_2 not in r_1
     - c: *
   - special cases:
     -> 2 relations have some attributes in common but the attributes have no values in common
     result = empty set
     -> 2 relations do not have attributes with the same name so condition * cannot be evaluated
     result = becomes a Cartesian product
   - associative and commutative

8. θ-join (combine)
   - selects the tuples resulting from the Cartesian product and satisfying the following condition:
     $A \theta B$
     where θ is a comparison operator ($\theta \in \{<, =, >, \leq, \geq\}$)
   - $r_1 \underset{A\theta B}{\bowtie} r_2 = \sigma_{A\theta B}(r_1 \times r_2)$

9. Renaming
   - creates a copy of the relation in which an attribute is renamed
   - $R_2 = \rho_{A \to B}(R_1)$

Relational algebra 3

-> join of a table with itself can be done
-> examples on slides 05

Designing a relational database - problems and constraints

- redundancy
  - waste of memory space
  - anomalies

  } happens when we have a database with a single schema and ≠ concepts

  -> update anomaly -> if some information changes we have to change that info everywhere
  -> insertion anomaly -> cannot insert some data if its not related to other data in table
  -> deletion anomaly -> if delete some data other important correlated data will be cancelled

solution:
A database schema is good if it has no redundancies and no anomalies.
To achieve this we need to correlate in the same schema only things that represent the same concept, so each concept has to be represented separetly in a distinct relation.
We have to make a database in which a single "table" can be achieved by join operations between multiple tables.
How to identify the concepts represented in a relation? By the key
→ an attribute/group of attributes that determine a particular functional dependence which is a particular kind of constraint

Constraints
↳ the representation in the database schema of a condition that is valid in the reality of interest
↳ an instance of the database is legal if it satisfies all constraints ("honest" representation of reality)
↳ DBMS allows us to define the constraints together with the DB schema, to verify if an instance is legal, to prevent insertion of tuples that would violate constraints
↳ functional dependencies express constraints between subsets of attributes of the schema itself

Functional dependencies

notation
- $R = A_1 A_2 \dots A_n$
- $(A, B, C, \dots)$ single attributes
- $(X, Y, \dots)$ sets of attributes
- $XY = X \cup Y$
- tuple = function that associates to each attribute A_i in R a value $t[A_i]$ in the domain
- $t_1[X] = t_2[X]$ if $\forall A \in X \; t_1[A] = t_2[A]$ (t_1 and t_2 coincide)
- $t_1[X] \neq t_2[X]$ if $\exists A \in X \; t_1[A] \neq t_2[A]$
- instance of R = set of tuples on R

functional dependence on R = ordered pair of non-empty subsets $X, Y \subseteq R$
- $X \to Y$ (X functionally determines Y or Y is functionally dependent on X)
- X = determinant, Y = dependent
- an instance r of R satisfies the f.d. $X \to Y$ if: $\forall t_1, t_2 \in r : t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y]$

note: if $t_1[X] \neq t_2[X]$ the dependence is satisfied whatever the values of $t_1[Y]$ and $t_2[Y]$

↳ implies
✓ = if $(t_1[X] = t_2[X])$ then $(t_1[Y] = t_2[Y])$
× = if $(t_1[X] \neq t_2[X])$ then $(t_1[Y] \neq t_2[Y])$

remark: $A \to B \equiv \bar{A} \cup B$
if $F = \{A \to B, B \to C\}$ is satisfied then $A \to C$ is also satisfied
if $A \to BC$ then $A \to B$ and $A \to C$ BUT if $AB \to C$ then not necessarily $A \to C$ and $B \to C$

- closure of F (F^+) = set of f.d. that are satisfied by each legal instance of R ($F \subseteq F^+$)
- a subset K of R is a key of R if $K \to R \in F^+$ and there is no proper subset $K' \subseteq K$ such that $K' \to R$
- $X \to Y \in F^+ \iff \forall A \in Y, \; X \to A \in F^+$ (Property of f.d.)

How to calculate F^+?

Closure of F

F^A is a set of functional dependencies on R so that:

- if $X \to Y \in F$ then $X \to Y \in F^A$
- if $Y \subseteq X \in R$ then $X \to Y \in F^A$ (reflexivity)] Armstrong axioms
- if $X \to Y \in F^A$ then $XZ \to YZ \in F^A$, for each $Z \in R$ (augmentation)] Armstrong axioms
- if $X \to Y \in F^A$ and $Y \to Z \in F^A$ then $X \to Z \in F^A$ (transitivity)] Armstrong axioms
- if $X \to Y \in F^A$ and $X \to Z \in F^A$ then $X \to YZ \in F^A$ (union rule) a
- if $X \to Y \in F^A$ and $Z \subseteq Y$ then $X \to Z \in F^A$ (decomposition rule) b
- if $X \to Y \in F^A$ and $WY \to Z \in F^A$ then $WX \to Z \in F^A$ (pseudotransitivity rule) c

proofs:

a if $X \to Y \in F^A$ by augmentation $X \to XY \in F^A$, if $X \to Z \in F^A$ by augmentation $XY \to YZ \in F^A$, since $X \to XY \in F^A$ and $XY \to YZ \in F^A$ by transitivity $X \to YZ \in F^A$

b if $Z \subseteq Y$ by reflexivity $Y \to Z \in F^A$, since $X \to Y \in F^A$ and $Y \to Z \in F^A$ by transitivity $X \to Z \in F^A$

c if $X \to Y \in F^A$ by augmentation $WX \to WY \in F^A$, since $WX \to WY \in F^A$ and $WY \to Z \in F^A$ by transitivity $WX \to Z \in F^A$

→ Let $X \subseteq R$, the closure of X w.r.t. F (X^+) is defined as $X_F^+ = \{A \mid X \to A \in F^A\}$

→ $X \subseteq X_F^+$

→ Lemma: let R be a schema and F a set of functional dependencies on R: $X \to Y \in F^A \iff Y \subseteq X^+$

Proof: let $Y = A_1, A_2, \dots, A_n$

if ($\Leftarrow$): since $Y \subseteq X^+$, for each $i = 1, \dots, n$ we have that $X \to A_i \in F^A$ by the union rule, $X \to Y \in F^A$

only if ($\Rightarrow$): since $X \to Y \in F^A$, for each $i = 1, \dots, n$, $X \to A_i \in F^A$ ($A_i \in X^+$), for each $i = 1, \dots, n$ $Y \subseteq X^+$

Theorem

$F^+ = F^A$

Proof: $F^A \subseteq F^+$

F_i^A = f.d. with at most i applications of Armstrong axioms

base case: $i = 0 \to F_0^A = F \subseteq F^+$

induction step: $i > 0 \to F_i^A \subseteq F^+ \Rightarrow F_{i+1}^A \subseteq F^+$ $\quad X \to Y \in F_{i+1}^A$

3 cases

- reflexivity $Y \subseteq X$ $t_1, t_2 \in r$ r = instance R
  $t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y] \Rightarrow X \to Y \in F^+$
- augmentation $V \to W \in F_i^A$ $Z \subseteq R$ $X = VZ$ and $Y = WZ$
  $t_1[X] = t_2[X] \to t_1[VZ] = t_2[VZ] \to t_1[V] = t_2[V] \wedge t_1[Z] = t_2[Z] \to$
  $\to t_1[W] = t_2[W] \wedge t_1[Z] = t_2[Z] \to t_1[WZ] = t_2[WZ] \to t_1[Y] = t_2[Y]$
- transitivity $X \to Z, Z \to Y \in F_i^A$ $\quad X \to Z, Z \to Y \in F^+$
  $t_1[X] = t_2[X] \to t_1[Z] = t_2[Z] \to t_1[Y] = t_2[Y]$

Proof: $F^+ \subseteq F^A$

$X \to Y \in F^+$

r	X^+	$R - X^+$
t_1	1.....1	1.....1
t_2	1.....1	0.....0

- prove r is legal

$V \to W \in F (\subseteq F^A)$ $t_1[V] = t_2[V] \Rightarrow V \subseteq X^+ \overset{\text{from Lemma}}{\Rightarrow} X \to V \in F^A \overset{\text{transitivity}}{\Rightarrow} X \to W \in F^A \Rightarrow W \subseteq X^+ \Rightarrow t_1[W] = t_2[W]$

$X \subseteq X^+ \Rightarrow t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y] \Rightarrow Y \subseteq X^+ \Rightarrow X \to Y \in F^A$

So $F^+ = F^A \to |F^+| >> 2^{|R|}$ ↳ lemma

Third Normal Form 1

->a relational schema is in 3NF if the only non-trivial f.d. that must be satisfied by any legal instance are those of type $k \to x$ where k contains a key or x is contained in a key

-> formally

$X \to Y$ non-trivial if $Y \notin X$

$X \to Y \in F^+$

-> R is in 3NF if $\forall X \to A \in F^+$, $A \notin X$, A belongs to a key (is prime) or/and X contains a key (it is a super key)

->to check 3NF we only need to check the dependencies $X \to A \in F^+$ obtained by decomposing the dependents of the dependencies in F

ex. $F = \{A \to B, B \to CD\}$ only check $A \to B$, $B \to C$, $B \to D$ but not $A \to C$, $A \to D$ that are obtained by transitivity

if A is the only key then the schema R for which F holds is not in 3NF

-> if the keys contain all dependents then for sure R will be in 3NF (since all dependents are prime)

Third Normal Form 2

->partial dependencies: when a dependency is a consequence of another dependency

$X \to A \in F^+ \mid A \notin X$ is a partial dependence on R if A is not prime and X is properly contained in a key of R ($X - k = \phi$)

->transitive dependencies: when a dependency is a consequence of two functional dependencies

$X \to A \in F^+ \mid A \notin X$ is a transitive dependence on R if A is not prime and for each key K of R we have that X is not properly contained in k ($X - k \neq \phi$) and $k - X \neq \phi$

-> schema is in 3NF iff there are neither partial nor transitive dependencies in F

-> goal: produce a schema in which every relation is in 3NF

-> if we produce schema that is not in 3NF we can decompose it so they are in 3NF

-> if we join decompositions we must produce a legal instance, we must preserve all dependencies in F^+ and not lose any information

Boyce-Codd Normal Form

a relation is in BCNF when every determinant in it is a superkey (a key is also a superkey)

-> if a relation respects BCNF it is also in 3NF, opposite not true

-> not always possible to decompose a schema in BCNF but always possible to do so with 3NF

Closure of X

·when obtaining schemas in 3NF we have to preserve all dependencies in F^+, but calculating it takes exponential time. But by the lemma $X \to Y \in F^+ \iff Y \subseteq X^+$ we only need to calculate X^+ to know that $X \to Y \in F^+$

Algorithm for calculating X^+

input: $R, F, X \subseteq R$ (x may be a single attribute)

output: X_F^+

```
begin
  Z = X
  S = A | Y → V ∈ F, A∈V ^ Y⊆Z
  while S ⊄ Z do:
    begin
      Z = Z ∪ S;
      S = A | Y → V ∈ F, A∈V ^ Y⊆Z
    end
end
```

insert into S the attributes that make up the right-hand parts of dep. in F whose left-hand part is contained in Z. At first Z is just X, so we insert attributes that are functionally determined by X; once these have entered Z, from these we add more (by transitivity)

Proof Let us denote by $Z^{(0)}$ the initial value of Z ($Z^{(0)} = X$) and by $Z^{(i)}$ and $S^{(i)}$ the values of Z and S after the i^{th} execution of the while loop; it is easy to see that $Z^{(i)} \subseteq Z^{(i+1)}$, for each i. Let j be such that $S(j) \subseteq Z(j)$ ($Z(j)$ = value of Z when algo terminates) we will prove that $A \in Z_j \iff A \in X^+$

part one: $Z_j \subseteq X^+$ by induction

base case: $i = 0$

since $Z^{(0)} = X$ and $X \subseteq X^+ \rightarrow Z^{(0)} \subseteq X^+$

induction step: $i > 0$ $Z^{(i+1)} \subseteq X^+$

$A \in Z_{i+1} = Z_i \cup S_i \longrightarrow A \in Z_i \subseteq X^+$

$\rightarrow A \in S_i \Rightarrow \exists\ Y \rightarrow V \in F$ s.t. $A \in V \wedge Y \subseteq Z_i$

$Y \subseteq Z_i \subseteq X^+ \Rightarrow X \rightarrow Y \in F^A$, $Y \rightarrow V \in F \subseteq F^A$ $\Rightarrow X \rightarrow V \in F^A \Rightarrow V \subseteq X^+$, $A \in V$ $\Rightarrow A \subseteq X^+$

$\Rightarrow Z_j \subseteq X^+$ (since we proved for any i)

part two: $X^+ \subseteq Z_j$

$A \in X^+ \Rightarrow X \rightarrow A \in F^+$ ($= F^A$)

consider

R	Z_j	R - Z_j
t_1	1 ... 1	1 ... 1
t_2	1 ... 1	0 ... 0

legal? ↓ yes!

$Y \rightarrow V \in F$ $t_1[Y] = t_2[Y] \Rightarrow Y \subseteq Z_j$, $Y \rightarrow V \in F$ $\Rightarrow V \subseteq S_j \subseteq Z_j$

$\Rightarrow t_1[V] = t_2[V]$

$X = Z_0 \subseteq Z_j \Rightarrow t_1[X] = t_2[X] \Rightarrow t_1[A] = t_2[A] \Rightarrow A \in Z_j$ (contradiction)

algorithm is polynomial time

$\Rightarrow Z_j = X^+$

Finding the keys of a schema

→ a subset K of a relation schema R is a key of R if:

1. $K \rightarrow R \in F^+$
2. there is no proper subset K' of K such that $K' \rightarrow R \in F^+$

remark 1 → start from those with higher cardinality, if their closure does not contain R → don't check c

remark 2 → attributes that never appear on the right of the functional dependencies of F must be in all the keys

remark 3 → brute force approach not best

remark 4 → every shortcut in calculations must be justified

→ if the closure of X contains R we have identified a superkey but we always have to check its minimality, meaning that we have to check that no subset of that closure is a key

Preserving F

Decomposition of R is a set of subsets $R_1, R_2, \dots R_k$ of R covering R that is $R = \bigcup_{i=1}^{k} R_i$

Equivalence, let F and G be two sets of f.d.; F and G are equivalent ($F \equiv G$) if $F^+ = G^+$, how to check?

Lemma: if $F \subseteq G^+$ then $F^+ \subseteq G^+$

Proof: $f \in F^+ \Rightarrow f \in G^+$

1. $f \in F \subseteq G^+$ $\quad \{f_1 \dots f_n\} \subseteq F \xrightarrow{A} f \in F^+$ $f_i \in F \subseteq G^+ = G^A$
2. $f \notin F \rightarrow f \in F^+ - F$ $\quad \{g_i^1 \dots g_i^{n_i}\} \subseteq G \xrightarrow{A} f_i$ → for every i $\xrightarrow{A} f \in G^+$

Preserving

$\rho = R_1, R_2 \dots R_k$ a decomposition of R

→ ρ preserves F if $F \equiv G = \bigcup_{i=1}^{k} \pi_{R_i}(F)$ where $\pi_{R_i}(F) = \{ X \to Y \in F^+ \mid XY \subseteq R_i \}$

check if F is preserved by checking $F \subseteq G^+$:

$\forall\, X \to Y \in F \Rightarrow Y \subseteq X_G^+ \Rightarrow X \to Y \in G^A \Rightarrow X \to Y \in G^+$

How To compute X_G^+ ?

Input: a schema R, a set F of functional dependencies on R, a decomposition $\rho = R_1, R_2, \dots, R_k$, a subset X of R
Output: the closure of X with respect to $G = \cup_{i=1}^{k} \pi_{R_i}(F)$, **(in variable Z)**
begin
$Z = X$
$S = \emptyset$
for $i = 1$ **to** k
 $S = S \cup (Z \cap R)^+_F \cap R_i$
while $S \not\subset Z$
 begin
 $Z = Z \cup S$
 for $i = 1$ **to** k
 $S = S \cup (Z \cap R)^+_F \cap R_i$
 end
end

Theorem: $Z_f = X_G^+$

Proof: $Z_f \subseteq X_G^+$ by induction

base case: $i = 0$

$Z_0 = X \subseteq X_G^+$

induction case: $i > 0 \quad Z_i \subseteq X_G^+ \Rightarrow Z_{i+1} \subseteq X_G^+$

$A \in Z_{i+1} = Z_i \cup S_i$

- $A \in Z_i \subseteq X_G^+$ by induction hypothesis
- $A \in S_i = \bigcup_{j=1}^{k} (Z_i \cap R_j)^+_F \cap R_j \Rightarrow \exists j$ s.t. $A \in (Z_i \cap R_j)^+_F \cap R_j \Rightarrow A \in R_j \wedge A \in (Z_i \cap R_j)^+_F$

$\Rightarrow \underbrace{(Z_i \cap R_j)}_{\subseteq R_j} \to \underbrace{A}_{\in R_j} \in F^A = F^+ \Rightarrow \underbrace{Z_i \cap R_j}_{\subseteq Z_i \subseteq X_G^+} \to A \in \pi_{R_j}(F) \subseteq G \subseteq G^+ \Rightarrow X \to A \in G^+ \Leftrightarrow A \in X_G^+ \Rightarrow Z_{i+1} \subseteq X_G^+$ → closure lemma $\Rightarrow X \to Z_i \cap R_j \in G^A$

Proof: $X_G^+ \subseteq Z_f$ $\quad Y \subseteq V \Rightarrow Y_G^+ \subseteq V_G^+$ (not part of program)

$X = Z_0 \subseteq Z_f \Rightarrow X_G^+ \subseteq (Z_f)_G^+$

$S' = \{ A \mid \exists\, Y \to V \in G \mid A \in V \wedge Y \subseteq Z_f \}$

Let $A \in S' \Rightarrow \exists\, j \in \{1 \dots k\}$ s.t. $Y \to V \in \pi_{R_j}(F) \Rightarrow S' \subseteq S_f \subseteq Z_f$

$X_G^+ \subseteq Z_f = (Z_f)_G^+$

Lossless join

Def: let R be a relation schema; A decomposition $\rho = R_1, R_2, \dots R_k$ has a lossless join if for each legal instance r of R we have $r = \pi_{R_1}(r) \bowtie \pi_{R_2}(r) \bowtie \dots \bowtie \pi_{R_k}(r)$

Thm: let R be a relation schema and let $\rho = R_1, R_2 \dots, R_k$ be a decomposition of R; for each legal instance r of R, denoted by $m_\rho(\rho) := \pi_{R_1}(r) \bowtie \pi_{R_2}(r) \bowtie \dots \bowtie \pi_{R_k}(r)$ we have:

1. $r \subseteq m_\rho(r)$
2. $\pi_{R_i}(m_\rho(r)) = \pi_{R_i}(r)$
3. $m_\rho(m_\rho(r)) = m_\rho(r)$

Proof:

1. $t \in r$. consider $t[R_i] \in \pi_{R_i}(r) \quad t \in \underbrace{\{t[R_1]\}}_{\in \pi_{R_1}(r)} \bowtie \dots \bowtie \underbrace{\{t[R_k]\}}_{\in \pi_{R_k}(r)} \quad \to t \subseteq m_\rho(r) \to r \subseteq m_\rho(r)$

$\pi_{R_1}(r) \bowtie \dots \bowtie \pi_{R_k}(r) = m_\rho(r)$

2. $\to \pi_{R_i}(r) \subseteq \pi_{R_i}(m_\rho(r))$

$t_i \in \pi_{R_i}(r) \quad \exists t \in r$ s.t. $t_i = t[R_i] \Rightarrow t \in m_\rho(r) \Rightarrow t_i \in \pi_{R_i}(m_\rho(r))$

$\to \pi_{R_i}(m_\rho(r)) \subseteq \pi_{R_i}(r)$

$t_i \in \pi_{R_i}(m_\rho(r)) \Rightarrow \exists t' \in m_\rho(r)$ s.t. $t_i = t'[R_i] \Rightarrow \exists t_1, \dots, t_k \in r$ s.t. $t'[R_i] = t_i[R_i] \in \pi_{R_i}(r)$

$t_i = t'[R_i] = t_i[R_i] \in \pi_{R_i}(r)$ contradiction

3. $m_\rho(m_\rho(r)) = \pi_{R_1}(m_\rho(r)) \bowtie \dots \bowtie \pi_{R_k}(m_\rho(r)) \overset{\text{by 2.}}{=} \pi_{R_1}(r) \bowtie \dots \bowtie \pi_{R_k}(r) = m_\rho(r)$

Algorithm

begin
construct a table **r** as follows:
r has R columns and k rows
at the intersection of the i-th row and j-th column put:
the symbol a_j **if the attribute** $A \in R_{i,j}$
the symbol b_{ij} otherwise
repeat
for every $X \to Y \in F$
if there are two tuples t_1 and t_2 in r such that $t_1[X]=t_2[X]$ and $t_1[Y] \neq t_2[Y]$
then for every attribute A_j in Y: **if** $t_1[A_j]=a_j$ **then** $t_2[A_j] = t_1[A_j]$
else $t_1[A_j] = t_2[A_j]$
until r has a line with all "a" **or** r has not changed;
if r has a row with all "a" **then:** it has a lossless join
else: it does not have a lossless join ⟶ has a lossy join
end

ex. $R=ABCD \quad \rho = \{ABC, ABD\} \quad F=\{AB \to C; AB \to D\}$

r	A	B	C	D
$R_1=ABC$	a_1	a_2	a_3	$b_{1,4}^{a_4}$
$R_2=ABD$	a_1	a_2	$b_{2,3}^{a_3}$	a_4

⇒ has a lossless join

Thm: let R be a relation scheme, F a set of functional dependencies on R and let $\rho = R_1, R_2, \dots, R_k$ be a decomposition of R; the algorithm correctly decides whether ρ has a lossless join.
S has a lossless join iff algorithm returns an instance r containing the tuple $t^a = (a \dots a)$

Proof:
$\overset{if}{\Rightarrow}$ Let r^0 be the initial instance
Let r^f be the final instance
$t_i^0[R_i] = (a, \dots, a) = t_i^f[R_i] \in \pi_{R_i}(r^f)$
$t^a \in \pi_{R_1}(r^f) \bowtie \dots \bowtie \pi_{R_k}(r^f) = m_\rho(r^f)$ but r^f is legal $\Rightarrow t^a \in r^f$
$\| r^f$ since assume has lossless join

Minimal covers

How to obtain a "good" decomposition?
Is it always possible to get it? Yes, sometimes even several
→ is the input to the decomposition algorithm
→ given F there can exist several equivalent minimal covers

Def: a minimal cover of F is a set of functional dependencies G, equivalent to F, such that:
- for each functional dependency in G, the dependent is a singleton (each dependent is non-redundant) → always possible thanks to decomposition rule
- for each dependency $X \to A$ in G, there not exists $X' \subset X$ such that $G \equiv (G - \{X \to A\}) \cup \{X' \to A\}$ (each determinant is non-redundant) → not possible to functionally determine A by a subset of X
- there not exists any $X \to A$ in G, such that $G \equiv G - \{X \to A\}$ (each dependency is non-redundant) → not possible to functionally determine A through other dependencies

How calculated?

in polynomial time in 3 steps:
1. using the decomposition rule, the dependants are reduced to singletons (right side of dependency)
2. every functional dependency $[A_1 A_2 \dots A_{i-1} A_i A_{i+1} \dots A_n \to A]$ in F such that $F \equiv F - \{\bullet\} \cup \{\bullet - A_i \text{ ('subset of X')}\}$ is replaced by "subset of X"; if the latter already belongs to F the original dependency is simply deleted; the process is repeted recursively on "subset of X"; the process ends when no f.d. can be further reduced
3. every functional dependency $X \to A$ in F such that $F \equiv F - \{X \to A\}$ is removed from F (because redundant)

2 and 3 are equivalence checks

2: $\bullet \in G^+$ and "subset of X" $\in F^+$ → not necessary

3: $X \to A \in G^+$ or $A \in X_G^+$

Decomposition algorithm

- given R and F which is a minimal cover, compute a decomposition $\rho = R_1, R_2, \dots, R_k$ such that
  - → for each $i = 1, \dots, k$ R_i is in 3NF
  - → ρ preserves F
  - → ρ has a lossless join

Algorithm

begin
$S = \emptyset$
$\rho = \emptyset$
for each $A \in R$, such that A is not involved in any functional dependency in F **do**
    $S = S \cup A$
if $S \neq \emptyset$ **then:**
    $R = R - S$
    $\rho = \rho \cup \{S\}$
if there is a functional dependency in F that involves all the attributes in R
  then: $\rho = \rho \cup \{R\}$
else:
    for each $X \rightarrow A \in F$ **do**
        $\rho = \rho \cup \{XA\}$
end

Theorem: the decomposition algorithm computes a decomposition ρ of R such that: each relational schema in ρ is in 3NF and ρ preserves F

Proof:

- ρ preserves F
  let $G = \bigcup_{i=1}^{k} \Pi_{R_i}(F)$; since for each f.d. $X \rightarrow A \in F$ we have that $XA \in \rho$, we have that this dependency of F will be in G, hence $F \subseteq G$ and $F^+ \subseteq G^+$
- each schema is in 3NF
  1. if $S \in \rho$, each attribute in S is part of the key and so S is in 3NF
  2. if $R \in \rho$, there is a functional dependency in F involving all attributes of R; as F is a minimal cover, this d. will have the form $R-A \rightarrow A$; as F is a minimal cover, it cannot exist a f.d. $X \rightarrow A$ in F, such that $X \subseteq R-A$ and, then, $R-A$ is a key of R; let $Y \rightarrow B$ be a d. in F; if $B = A$ and F is a minimal cover, then $Y = R-A$ (Y is a superkey); if $B \neq A$ then $B \in R-A$ and so B is prime
  3. if $XA \in \rho$, as F is a minimal cover, it cannot exist a f.d. $X' \rightarrow A$ in F such that $X' \subset X$ and hence X is a key of XA; let $Y \rightarrow B$ be any dependency in F, such that $YB \subseteq XA$; if $B = A$ then, as F is a minimal cover, $Y = X$ (Y is a superkey); if $B \neq A$ then $B \in X$, so, B is prime

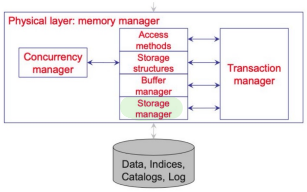
  it can be either 1+2 or 1+3 or 3 only

What about the lossless join?

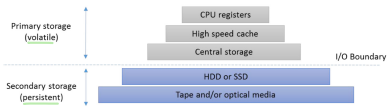
it can be demonstrated that, to have a lossless join, we just need to add to ρ a subschema containing one of the keys of R

Physical Organization

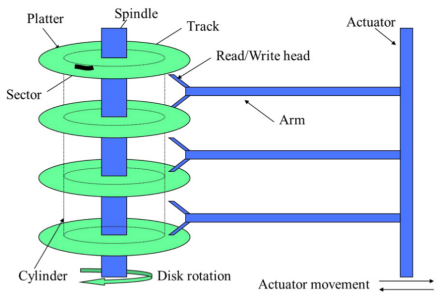
-> physical layer : memory management



storage hierarchy



- top, primary storage:
  - CPU
  - cache
  - central storage, RAM
  - contains DB buffer and runtime code of applications and DBMS
- bottom, secondary storage
  - persistent storage media
  - HDD, SSD
  - contains physical DB files



Hard Disk Drives

-> stores data on circular platters, covered with magnetic particles

-> contains a hard disk controller

-> they are directly accessible storage devices (DASD)

-> platters are secured on a spindle, that rotates at a constant speed

-> read/write heads can be positioned on arms, that are fixed to an actuator

reading from a block, or writing to a block implies

- positioning the actuator -> seek time: time to position the head on the correct track
- wait until the desired sector has rotated under the read/write head -> rotation delay
- transfer time: depends on block size, density of magnetic particles and rotation speed of disks
- response time: service time + queuing time
  - seek time + rotational delay + transfer time
- t_{rba} (random block access): expected time to retrieve/write disk block independently of previous r/w
  - t_{rba} = seek + ROT/2 + BS/TR ←
- t_{sba} (sequential block access): expected time to sequentially retrieve disk block with r/w head already in correct position
  - t_{sba} = ROT/2 + BS/TR ←

-> ROT: rotation time, BS: block size, TR: transfer rate

From logical to physical

Logical data model (general terminology)	Logical data model (**relational** setting)	Internal data model
Attribute type and attribute	Column name and (cell) value	Data item or field (bits or characters representing a specific value)
(Entity) record	Row or tuple	Stored record (collection of data items; represents all attributes of an entity)
(Entity) record type	Table or relation	Physical file or data set (similar entities, e.g., students)
Set of (entity) record types	Set of tables or relations	Physical database (collection of stored files)
Logical data structures	Foreign keys	Physical storage structures (logical interrelations like foreign keys)

The physical database

- a database consists of a set of files
- each file can be seen as a collection of pages/blocks, of fixed size
- each page stores several records (logical tuples)
- a record consists of several fields, with fixed/variable size (tuple's attributes)

Primary file organization methods

- linear search: each record in file is retrieved and assessed against search key
- hashing and indexing: direct access to the location containing the records corresponding to the search key

] search and manage data

1. Heap file

- simplest
- only linear search
- on average check $n/2$ blocks

search time: $\frac{n}{2} \cdot t_{sba}$

2. Sequential file

- records in ascending/descending order of search key
- linear and binary search

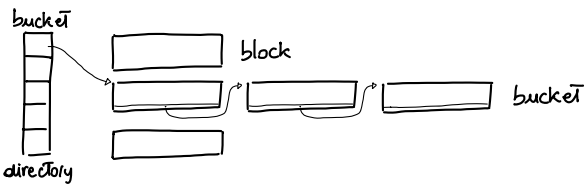
search time: $\log_2(n) \cdot t_{rba}$

- Selection criterion: record with search key value K_μ
- Set l = 1; h = number of blocks in file (suppose records are in ascending order of search key K)
- Repeat until h ≥ l
  - i = (l + h) / 2, rounded to nearest integer
  - Retrieve block i and examine key values K_j of records in block i
- if any $K_j = K_\mu$ ⇒ record is found!
- else if K_μ > all K_j ⇒ continue with l = i+1
- else if K_μ < all K_j ⇒ continue with h = i-1
- else record is not in file

- updating done in batch, since have to shift all
- often combined with indexes

3. Random file (hashing)

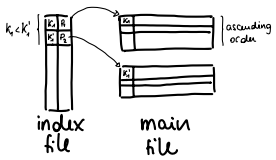
- assumes direct relationship between value of search key and physical location
- record retrieved with few block accesses
- hashing algorithm defines key-to-address transformation
  -> record physical address calculated from its key
- most effective when using primary key



- doesn't guarantee that all keys are mapped to different hash values
  - collision occurs when records assigned to same bucket (synonyms)
  - if more synonyms than slots for a bucket -> overflow -> more block accesses to retrieve overflow records
  - should distribute keys evenly
- popular hashing technique is division: $address(key_i) = key_i \bmod M$ where M is a prime number (close to, but larger, than # of available addresses), remainder of modulo division is the record address
- retrieving non-overflow record: $1 t_{rba}$ to first block of bucket, possibly followed by $m \cdot t_{sba}$
- additional block accesses for overflow records depending on percentage of overflow and handling technique
- required number of buckets b: $b = \lceil r/(bs \cdot lf) \rceil$ r = # records, bs = bucket size, lf = loading factor (how much we want the block to be filled)
- search time = $\frac{n}{2b}$
- open addressing: overflow records stored in next free slot after full bucket where a record would normally have been stored

4. Indexed sequential file indexed sequential access method - ISAM

- file divided into intervals or partitions
- each interval is represented by index entry (in index file)
  - search key value of first record in interval
  - pointer to physical position of first record in interval
    - block pointer
    - record pointer

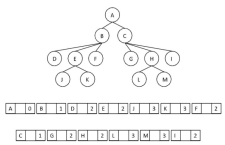


- dense or sparse indexing
  - dense: index entry for all possible value of search key
  - sparse: index entry for only some of search key values
    - each entry = group of records
  - dense is faster but more storage
    - still less storage than data/main file

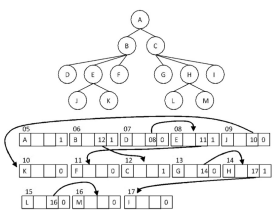
- data file is ordered on unique key, index is defined over this unique search key
- search time: $\log_2(nbi) + t_{rba}$ with $nbi << n$ (nbi = # of blocks in index) (index based search)
  - t_{rba}: access block in main file

Tree-structured indexes

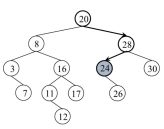
- properties:
  - all children of a node are the descendants
  - a node without children is a leaf node
  - a tree where all leaf nodes are at the same level is called balanced
  - nodes are stored in top-down-left-right sequence
    - letters are record keys, numbers are level in tree
    - can only navigate in sequential way



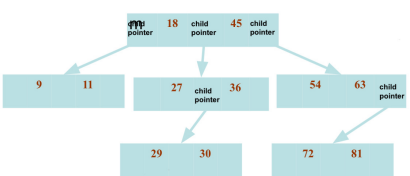
- trees can be implemented by linked lists
  - each node has pointer to next sibling
  - single bit often added if node is leaf node (0)
  - parent-child and sibling-sibling navigation is supported

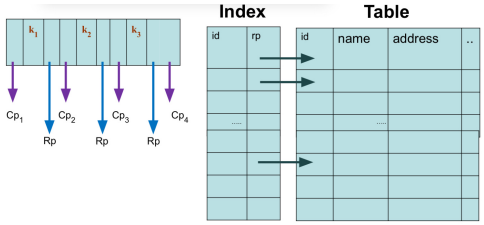


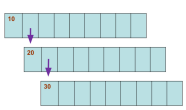
- binary search trees
  - left subtree only contain key values that are lower than key value in original node, right subtree higher



- multiway search trees
  - node has n key values and n+1 pointers, $n+1 \leq m$
  - all key values in subtree of P_i are less than K_i
  - all key values in subtree of P_{i+1} are bigger than K_i
  - a node can have from 1 to m-1 values and the number of sub-trees is from 0 to 1+i, where
    - i is the number of key values in the node
    - m is a fixed upper limit that defines how many key values can be stored in the node
  - an m-way search tree has m-1 keys
  - each node corresponds to a disk block
  - besides child pointers (Cp_i) we consider
    - record pointers (Rp) after every key
    - problem: no control in the way keys are inserted -> should fill up first node



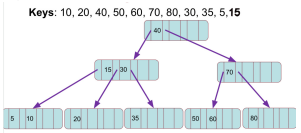




→ B-tree

- a self-balanced m-way search tree
- every node has at most m children
- every node except the root node and leaf nodes have at least m/2 children
- root node has at least two children if it is not a leaf node
- all leaf nodes are at same level (balanced)

} properties



- the creation process is bottom-up
- each node corresponds to a disk block and nodes are kept between half full and full
- every node contains a set of search key values, a set of tree pointers that refer to child nodes, a set of data pointers that refer to data records, or blocks with data records, that correspond to search key values and that are stored separately and are not part of B-tree
- search
  - recursively starting from root
  - cost ≤ tree height - 1 + 1
    - root node in RAM
    - accessing the data file
    - $h_{min} = \lceil \log_m (N+1) - 1 \rceil$ N = # nodes m = max # children for node
    - $h_{max} = \lfloor \log_d ((N+1)/2) \rfloor$ d = min # children of node