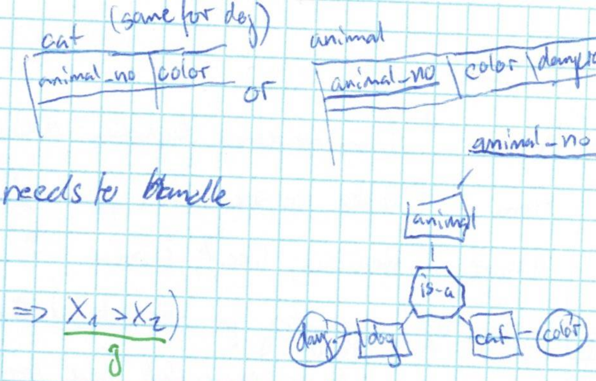
Solution Patterns:



DRC:

$\{ <\ldots,\ldots> \mid \exists / \forall\ (<\ldots,\ldots> \in r\ \wedge, \vee \ldots) \}$

1. Specify the Sets
2. Specify anything except the Set the Predicate needs to handle
3. Specify the Output (the ones which aren't in ∃ unbound Var.)
4. Specify the Quantifiers $\forall x (\underbrace{<\ldots> \in r}_{f} \Rightarrow \underbrace{x_1 > x_2}_{g})$

f	g	f ⇒ g
T	T	T
T	F	F
F	T	T
F	F	T

$\exists x(a_1, a_2 \ldots, a_n)$

ER-Diagram:

1. Umwandeln von allen Entities zu Relations → Spezialfälle sind Generalizations & Weak Entities
2. Relationships ebenfalls zu Relations umwandeln. Dabei werden die Primary keys folgendermaßen gesetzt:
   - → N:M → Kombination beider Keys
   - → 1:N / N:1 → Primary Key der N Seite
   - → 1:1 → Arbitrary one
   - \+ Immer alle Attribute der Relationship
3. Bei Relations von Entities & Relationships die denselben key haben, wird die Relationship in die Entity Relation gemerged (alle Attribute der Relation die nicht in der Entity Relation sind werden darin aufgenommen).
4. Foreign Keys sind Attribute einer Relation, die nicht Attribute der Entity selbst oder der Relationships selbst waren & keinen Primary Key der Relation darstellen (also ein Primary Key einer 1:N Beziehung z.B).

Relational Algebra:

Relational Operations:

$\Pi_{x,\ldots}$ → Removes Duplicates → $x, \ldots$ Specify Column

$\sigma_{Predicate}(R)$ → Returns Tuples that fullfill the Condition

* CP → all possible Combinations → $R_1(A,B) \times R_2(C,D) = R_3(AC, AD, BC, BD)$ (doesn't need same # of attributes / name)

(Renaming) ρ → $\rho_{After \leftarrow Before}(R)$

Set Operations:

same # of tuples…

⇒ they need the same relational schema (= same # of attributes & same name of attributes)

- * Joins: Theta: $R_1 \bowtie_{Predicate} R_2$, Equi: $R_1 \bowtie_{R_1.A = R_2.B} R_2$, Natural $R_1 \bowtie R_2$, Outer (Full, Right & Left; just fill up non dominant with NULL)
- Union: "everything once" → no duplicates
- Intersection: "every row where R_1 and R_2 are equal" (the rows which both have) $R_1 \cap R_2$
- Difference: "Extracts from R_1 everything which is not in R_2" $R_1 \setminus R_2$
- Division: Used for "for all" queries → only needs the same name of attributes; extracts all rows whose column values match those in the second Relation, BUT only…

SQL:

Select * (does not remove duplicates; distinct, aggregate functions)

From (Tables we select Data from & specify the joins; f.e country C join geoDesert GD on C.name=GD.code)

Where (Predicates; like "%", between, *)

group by (aggregating for Attribute(s) which is present in multiple tuples (merges to 1 tuple) all other attributes have to be aggregated here too or specified by an aggregation function in the Select clause)

Having (filter option in the end; 1. Where, 2. group by, 3. Having <filter statement>)

Query Intersect Query

Query Union Query

Query Minus (=Difference) Query

} Will need the same schema just as in RA

Remove Duplicates

Nested Queries:

Correlated: For every tuple of the outer query the inner query needs to be executed; then it is returned to the outer query and from there to the final result
Uses the Exist operator

Uncorrelated: Inner query executes, then outer executes and returns the valid tuples to final result

Decomposition Algorithm: 3NF => Others P. 6 in Summary.

candidate key $K = AC$

$R_0 = (A, B, C, D)$

$F = \{A \to B, C \to D\}$

1. Minimal Basis
2. go through the set holding the FDs. Convert every FD into a subschema → $R_1(A,B)$, $R_2(C,D)$
3. If we lose the candidate key bzw. no relation holds the candidate key anymore we create a new relation R_K which holds the key → $R_1(A,B)$, $R_2(C,D)$, $R_3(A,C)$
4. Remove redundant schemas → if there is a schema which is a subset of another schema, then drop it.

Basics:

=> When we Normalize = Split up tables we have to make sure 2 things:

1. Dependency Recoverability → Done via Minimal Basis → Done by Internet tool
2. Information Recoverability → That means that the relations we split into can do a "lossless join" to become the former relation again.

No FD may be completely lost bzw. has to be found in one Relation

Check Information Recoverability (lossless):

$$\left.\begin{array}{l} R1 \cap R_2 \hat{=} R_1 - R_2 \\ R_2 \cap R_1 \to R_2 - R_1 \end{array}\right\} \text{ bzw } \left.\begin{array}{l} R1 \cap R_2 \to R_1 \\ R_1 \cap R_2 \to R_2 \end{array}\right\} \text{ is in } F^+$$

(either one of them)

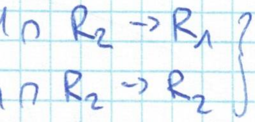
closure of F; F^+ = set of all regular FDs that can be derived from F

1. $Attr(R1) \cup Attr(R2) = Att(R)$ (Union of R1 and R2 must be equal to attribute of R; Each Attribute of R must be either in R1 or R2).
2. $Attr(R1) \cap Attr(R2) \neq \emptyset$ (Intersection Attr. must not be NULL)
3. $Attr(R1) \cap Attr(R2) \to Attr(R1)$ or $Attr(R1) \cap Attr(R2) \to Attr(R2)$ (Common Attribute must be key for at least one Relation)

Structures:

ISAM:

Read what happens when a page overflows; if it is treated as an overflow page then just create a new page and add the pointer from the overflowing page to that one; remember that there is only space for 1 index block, n attributes in the block and n+1 pointers.



B-Tree: B-Tree Visualization

$i = 1$ ≙ Max. Degree 3 which means a node holds 2 keys & 3 Pointer

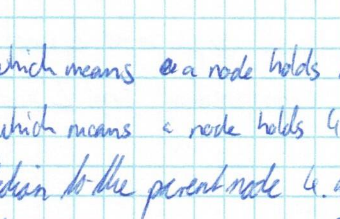
$i = 2$ ≙ Max. Degree 5 which means a node holds 4 keys & 5 Pointer

overflow: 1. get Median, 2. Split the inner node, 3. Move up the Median to the parent node 4. adjust pointer

delete: When I delete something in the root I might have to push down a node

↳ If there is an underflow pull down from parent ↳ (prev than i)

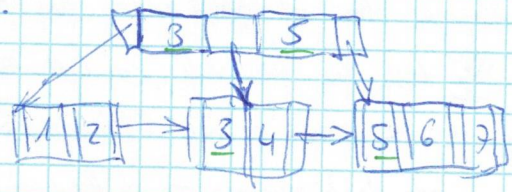
↳ Page (8)



all values are unique

B+-Tree: B+-Tree Visualization → [...,...,...,...,...]

Zuerst Knoten eingeben, der Median kommt dabei in die Root und in die rechte Node kriegt den value zugewiesen.



1. Insert
2. Check if too many
3. Move Median up & move Median inclusive to new node
4. Check Parent node; if that overflows as well split it as well the same way

⇒ There are always all key values once in the leaves
⇒ inner nodes have always n values and n+1 pointers
⇒ Check that values to the right are actually bigger
⇒ They are ordered
⇒ No values double

Partitioning bzw. Hashing:

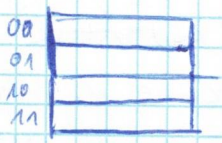
1. Write down the buckets and the values in the buckets
2. Write down local t'
3. Adjust global t
4. Adjust / assign pointers according to t'

⇒ When an overflow appears repeat 1-3 until the corresponding bucket has been split successfully
⇒ Then do step 4;
Pointers are assigned according to the local t' place the buckets are in (so multiple pointers might go to one bucket because even if there are 4 levels in the global scope the local scope is at 2 and we only read the corresponding first 2 numbers.)

calculate via the hash function what binary code there is:
$+1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$

directory

t=2

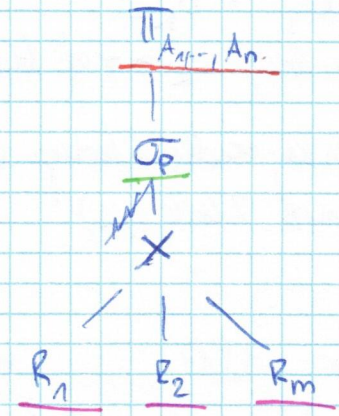


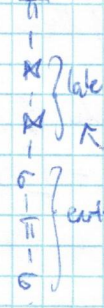
Optimization: P(11) - (12)

SQL gets translated to an RA canonical tree (P11)

Select $A_1, \dots, A_n$
from $R_1, \dots, R_m$
where Predicate

} Execution Plan →





Logical level basic techniques:

- Push Selection and Projection downward the relation s.t we make joins over smaller relations
- Do the smallest relations first
- Convert CP into join
- Insert useful Projections
- leave away unnecessary operators
- 1. ~~Π~~ the data we need asap
  2. σ the data asap
  3. join

Query Processing (P13-17)

Estimating Costs:

1. Selectivities: How many tuples get through?

Selection:

Nbr of tuples using the filter option

$$*Sel_P = \frac{|\sigma_P(R)|}{|R|} \quad \begin{matrix} \text{(all the ones that pass)} \\ \text{(all the tuples)} \end{matrix}$$

Join:

$$Sel_{RS} = \frac{|R \bowtie S|}{|R \times S|} = \frac{|R \bowtie_{\frac{1}{2}} S|}{|R| \cdot |S|} \quad \begin{matrix} \text{(Size of Join)} \\ \text{(Size of CP)} \end{matrix}$$

Used in formulas for estimating Cost for selection and join

Easy cases:

$*Sel_{R.A=C} = \frac{1}{|R|}$ when A is primary key of R (because there is ~~a~~ always 1 value)

$Sel_{R.A=C} = \frac{1}{i}$ if i is the number of different values (bzw. attributes) in Relation R (assuming uniform distribution)

i=2 = 50% passage, i=3 = 33% passage, i=4 = 25% passage ---

$Sel_{R.A=S.B} = \frac{1}{|R|}$ if equi-joining via foreign key S.B

2. Choose a structure if there is one: (for selection Cost Estimate)

1. Brute Force → scan all the pages of R (linear search)
2. B^+-Tree index: $t + \lceil Sel_? * b_R \rceil$
   t: number of levels in the tree, b_R = # of blocks in R
3. Hash Index → If attribute is key exactly 1 lookup basically 1 lookup for every attribute value (at most 2 look ups)

3. Estimating cost for a join Operator (P17)

b_R: # of pages in R ← bzw. blocks ← outer ← $= \lceil \frac{\#\text{tuples}}{\text{how many tuples 1 block holds}} \rceil$ (same for b_S ? with inner numbers)

$b_R + k + \lceil b_R/(m-k) \rceil \cdot (b_S - k)$

b_R: Pages outer bzw. blocks

$\lceil b_R/(m-k) \rceil$: # of times we have to scan through S

m: total memory we have

k: memory we have for inner loop

m-k: total memory we have assigned to outer loop

⇒ If we swap around inner and outer just swap b_R and b_S values.

Recovery: (24-26)

write, BOT or commit

- log file

Write the logs: for Undo operations CLR (Compensation Log Record) we create a

due to LSN if it crashes before all TAs commit we can Redo as many times as we want because we save LSN on top of the page and see how far we came.

[LSN, TA, PageID, Redo, Undo, PrevLSN]

if you want the same change again

if you want the change reverted

↳ LSN of the last TA of the same TA

bzw. CLR

Write logs after recovery: <LSN, TA identifier, affected Page, Redo Information, PrevLSN, Undo NxtLSN>

(in reverse order; ↳ just do those according to checkpoint in rev. order)

T_2 i.e. (P_C)

Put Undo info into the redo here (shift) ⇒ we are never going to redo an undo CLR therefore no undo info into field needed

debugging

Undo NxtLSN: Where to continue with the undo; if we crash during recovery (to the next CLR to

- Depending on the checkpoint (need ↳ and if we have one) to store some log entries, since some TA's went through and some did not.
  Any way we have to deal with the CLRs up from that checkpoint depending on what kind of checkpoint we have.

1. Write logs
2. Determine CLRs (for the non-committed
3. If we have a checkpoint write the CLRs for that case.