Normalize

Given a relation & a set of functional dependencies

Col A ⟶ Col B, C, D

or Col A, Col B → Col C.

Decompose/Normalize relation w/o info loss and so that functional dependencies can be enforced.

Rules:

1. No redundancy of facts
2. no cluttering facts
3. must preserve info
4. must preserve functional dependencies

Relations might have several problems:

1. Redundancy. same value for a particular column.
2. Insertion anomaly, insert only partial info leaving null value
3. Deletion anomaly: delete a row, while new info lost, we may just want to delete part of info
4. Update anomaly: only update some row, causing consistency problem.

We can decompose a relation into 2. However, if not done properly, we might have 1. info loss. eg. the connecting column is not unique, might causing extra rows (spurious tuples) that weren't in original table, this is also info loss 2. dependency loss, we can't have Col A, B → Col C when Col C is in another relation

How to decompose?

- Functional Dependencies:
  def: Let $X$ & $Y$ be sets of attributes in $R$
  $Y$ is functionally dependent on $X$ in $R$ iff for each $x \in R.X$ there is precisely one $y \in R.Y$.
- Full Functional Dependencies: extraneous attribute: remove it w/o changing closure.
  $Y$ is Full Functional Dependencies if $Y$ is functional dependent on $X$ & $Y$ is not functional dependent on any proper subset of $X$

prime attribute: if it is a member of some candidate key.

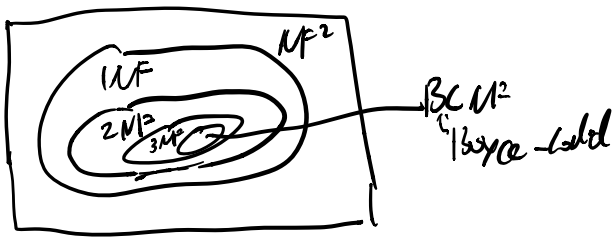
We use keys to enforces full functional dependences $X \to Y$

- In a relation, the values of the key are unique
  (let $X$ be the key, if $X \to Y$)

How well a relation keys act

4 normal forms

Whole set of data structures is called non-first normal form NF²




1NF: $R$ is in 1NF iff all domain values are atomic
2NF: 1NF && every non key attribute is fully dependent on the key

3NF: 2NF && any Nonkey attribute is non-transitively dependant on the key

★ BCNF: every determinant is a candidate key

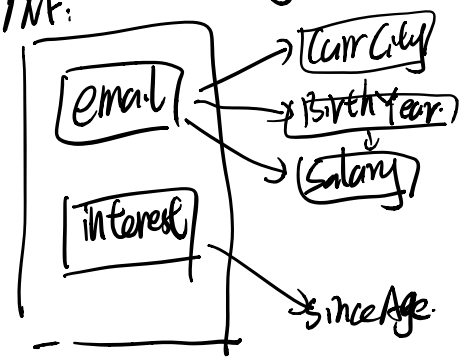
determinant: a set of attributes on which some other attribute is fully functionally dependent.

or 1NF: All attributes must depend on the key

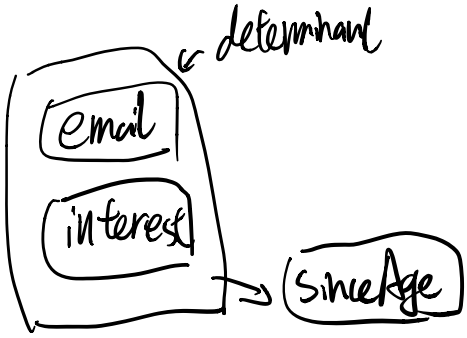
2NF: All attributes must depend on the whole key

3NF: All attributes must depend on the whole key & nothing but the key

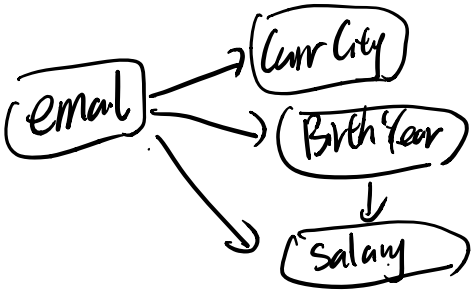
e.g. 1NF:




BCNF:




2NF:




(key is email)

transitively dependent:

email → BirthYear → Salary

!! 3NF & BCNF

BirthYear → Salary

& Email → Curr City, Birth Year

How to compute w/ Functional Dependencies:

Armstrong's rules:

reflexivity: if $Y \subseteq X$ (Y is part of X), then $X \to Y$ — this is trivial.

e.g. Email, Interest → Interest

⇒ (1) $X \to Y$, $Z \subseteq X$ ⇒ $X \to Z$

(2) $X \to Y$, $X \to Z$ ⇒ $X \to YZ$

(3) $X \to Y$, $WY \to Z$ ⇒ $WX \to Z$

augmentation: if $X \to Y$ then $W, X \to WY$

if $col A \to col B$ then $col A, C \to col B, C$

transitivity: if $X \to Y$ & $Y \to Z$ then $X \to Z$

$F^+$: extends to all FD implied by F. $X^+$: all attributes can be determined by X.

How to guarantee lossless Joins — $F^+ = E^+$? F ≡ E (equivalent)

Join field must be a key in at least one of the relations

How to guarantee preservation of Functional Dependency

the meaning implied by the remaining functional dependencies must be the same

In theory, there is relation that is 3NF but not BCNF, while (and can only decompose to 3NF)

being lossless & dependency preserving.

- It can only happen when the relation has overlapping keys

(really never happen in practice though...)

minimal FD set: 1. every dependency has 1 attribute in right hand side. 2. cannot replace any $X \to A$ w/ $Y \to A$ where $Y \subsetneq X$ & still have a set of dependencies equivalent to F 3. cannot remove any dependencies and still stay equivalent

# Efficiency

Access time

RAM: main memory, volatile, fast small & expensive — $3 \times 10^{-7}$ sec (50 nanosec)

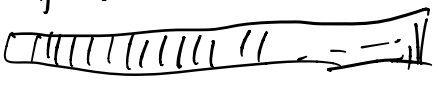
Disk: second memory, permanent slow big & cheap — $10^{-2}$ sec. (10 millisec)

- CPU application can only query & update data in main memory
- Data must be written to secondary memory after updated.
- Only a tiny fraction of a real database in main memory

RAM — CPU
↕ bus
DISK

We only need to consider I/O time not CPU time. ← too small, $3 \times 10^{-7}$ sec.

Records: Regular User(
e.g. Email varchar(50) ← 159 bytes
sex char(1)
Birthdate datetime ← 8 bytes
CurrentCity varchar(50)
Hometown varchar(50)); ← where record start etc.




Assume block size 4k. W/o meta data & 8% filled. ⇒ 3200 bytes
⇒ ~~100~~ records for each block
↑ blocking factor

Spanned representation: record broken up into 2 blocks
usually don't do that

blocks have pointer, connect to next block of this file

Transport data between disk & main memory.

Seek time: 3~8 ms  Transfer time: a 5~1 ms
rotational delay: 2-3 ms ⇒ Total 5~12 ms or cools

bulk transfer: transfer multiple block w/o additional seek time or Rotational delay, which need buffer space. Then, we need buffer management: e.g. LRU strategy

(that is in I/O)

won't work for nested loop join.

when we run out of buffer space, rewrite earliest used space

double buffering, 2 buffer, CPU read 1 & fill other one data concurrently.

Pin-Count: # times a page requested, if >0 will stay in buffer.

dirty bit: page updated? 1:0.

## Heap - unsorted file.

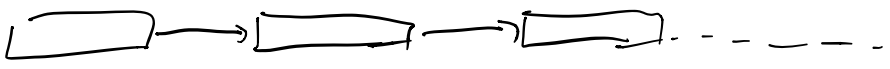
a file of data unsorted

Any record can be any block

to find page, we need to go over all data blocks (N), on average $\frac{N}{2}$

## Sorted file.

data sorted



binary search

## Indices.

primary index: specified on ordering key field of an ordered file of records

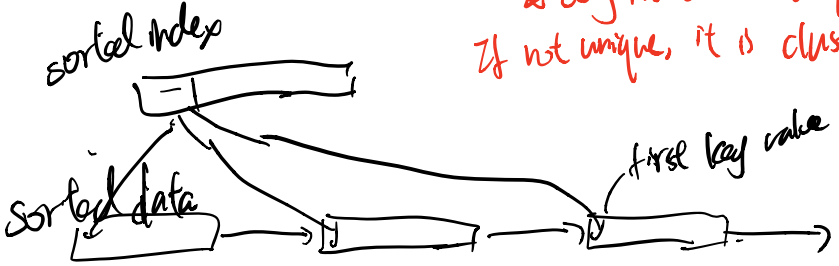
Physically order the records

so every records have unique value on such file.

If not unique, it is clustering index

primary or clustering but not both

non dense.




key value: email varchar(50) & pointer: 4 bytes = 54 bytes

~80% filled & 4kbs/block. ⇒ 3200 bytes

3200 ÷ 54 ≈ 60 fanout: ~60 (how many pointers in 1 index block) that is each index block contains around 60 data blocks indices. (which is Blocking factor)

have ~ 200k data blocks ⇒ 200k ÷ 60 ≈ 3334 index block. (Sparse index: only pick first key)

If we pick every key value on block, we will get dense

index.

If we have $n$ = # index blocks

Look up Time: ← access the value

$\log_2(n) + 1$

Dense index look up time > Sparse.

Secondary index: - specified on non ordering field. - Point queries only Have to be dense if unique value.

indices on non-key value, which is not sync order with key value (can be key as well) And it has to be dense index as records are not sorted.

So we get value of each record, pick up the key & ptr in index blocks. then sort. So you can only go to a point as data block is not sorted for this field.

Multi-level index. $\log_{\text{fanout}}(n) + 1$

index of index ....

good: fast of look up time.

bad: easy to overflow hard to adjust

$B^+$-Tree.

Static hashing (only fixed # of buckets)

Very large key space $\xrightarrow{\text{hash function}}$ address Point directly

good hash function:

distribute values uniformly over address space.

fill buckets as much as possible

Avoid collisions

hash field space: space of elements needed to hash

B Tree: balanced. used as primary file organizations