# Normal Form

1st Normal Form - Atomic

2nd Normal Form -

Boyce-Codd Normal Form

3rd Normal Form

4th & 5th Normal Form

## Anomalies

* redundancy
* update anomaly
* delete anomaly
* insert anomaly

→ develop a theory to understand why this design may be better and how to find this decomposition.

# Functional Dependency

A → B: A functionally determines B, functional dependency

the instance of schema = multiset of tuples conforming that schema = table

## Checking B → C FD in R(A, B, C) by SQL code:

>> select B

>> from R

>> group by B

>> having count(distinct C)>1;

Good FD: Minimal redundancy, less probability of anomalies.

Bad FD: Redundancy, Possibility of data anomalies.

## Armstrong’s Rules

1. Split / Combine
2. Reduction
3. Transitivity

# Closures

{A1, … ,An} → B

{A1, …, An}+ = set of attributes B

Use a closure to find all FDs.

Algorithm

# Boyce-Codd Normal Form

Superkey: all attributes are functionally determined by superkey, X+ = all attributes

key: minimal superkey

find bad FD → decompose table into subtable until no bad FD → normalized!

## Good FD in BCNF:

X→A, if X is a superkey

## Bad FD in BCNF:

otherwise

Algorithm

# Buffer

* Disk:

slow(IO cost is high), durable, cheap

* RAM (Random Acess Menory) or Main Memory:

fast(10x sequntial, 100000x random), volatile, expensive

buffer: region of physical memory to store temporary data.

→ reducing the performance gap between disk and main memory.

disk IO slow → cache data using buffer

page: fixed-sized array of memory

file: variable length list of pages

Process Read / Write page in buffer without accessing the disk

* Read page
* Flush page
* Release page

commit; → flush

rollback; → release

# External Merge Algorithm

IO dominate algorithm cost

IO Aware algorithm?

input: 2 sorted lists (length M, N)

output: 1 sorted list (length M+N)

at least 3 buffer pages:

IOs: 2(M+N)

# External Merge Sort

1. split into chunks small enough to sort in the main memory

cf. run: each sorted file

1. external merge algorithm

# B+Trees & IO-Aware Index Structure

slack: extra space in the page

search key: any sub-set of fields, not the same as the relation’s key

index → data structure mapping searching keys to set rows in a database table

index covering: query can be answered using the index alone

needed attributes are the union of attributes in select and where clause

## How to build the index

* **create index** index name **on** table name(column name)

B Tree: Balanced Tree, one node = one page. store data either leaf or non-leaf. no duplication

B+Tree: leaves are linked. leaf level has all keys, some duplication. faster sequential processing.

order d: max 2d slots, 2d+1 pointers

## B-Tree operation

* insertion

empty slot, then just insert

full slot, then split the node: create a new node and distribute evenly (old + new) → promote center value to next higher node

* deletion

key in leaf: just delete.

key in non-leaf: delete the key and promote the next higher value to the slot.

\*next higher value: leftmost value in the right subtree.

## B+Tree operation

* insertion

insert at the bottom level

if the **leaf** overflow → split the node and **copy** the middle element to the next higher level node

if **non-leaf** overflow → split the node and **move** a middle element to the next higher level node

* deletion

delete the key from the leaf

if leaf underflow → merge with the adjacent leaf, delete the key in the non-leaf

if non-leaf underflow → merge with adjacent non-leaf, move down the key from the next higher node.

B+Tree with range queries: sequential traversal using links between the leaf nodes

# Transaction

## our model

shared: each process can read from / write to shared data in the main memory

disk: global memory can read from / flush to disk

log: assume that stable disk storage - spans(걸치다) both main memory and disk\

Transaction: sequence of multiple operations which reflects a single real-world transition.

→ ALL or NOTHING

SQL GRAMMAR:

START TRANSACTION

~~

OPERATIONS

~~

COMMIT; / ROLLBACK;

* Recovery & Durability: Keep the log to be able to rollback the TXN (all or nothing)
* Concurrency: good performance, as if alone. no waiting time for the user.
  + execution order is important! & one by one is inefficient!

## ACID

* **Atomicity**: all or nothing, commit → reflect changes; rollback → no changes.
* **Consistency**: always satisfy user-specific integrity constraints
* **Isolation**: executes concurrency, alone
* **Durability**: effect of TXN must continue to persist, write data to disk. when DBMS stop, all changes due to the completed TXN should all persist.

Atomicity & Durability for Recovery & Durability

Consistency & Isolation for Concurrency

## Challenges for ACID

* failures
* need to rollback
* concurrency
* performance

## Log for Atomicity and Durability

* the ordered list of modifications, archived in stable storage
* force to write the entries to the disk
* handled transparently
* Record UNDO minimal information <XID, location, old data, new data> because of memory and time limitation

## Write-Ahead Logging

ensure durability while maintaining our ability to UNDO

The TXN is Committed after all the logs are stored in stable storage (regardless of actual data saving)

Cannot rollback after commit

## Cluster Model

commit by replicating data or log into n other machines (fault resilient)

## Log Summary

So…? Choose the method to log!

WAL / Replication on n machines / Hybrid

## Concurrency by Consistency and Isolation

Consistency: consistent status

Isolation: as if alone

## Interleaving

might lead to an anomalous outcome, but performance!

→ as if the TXN executed serially!

check the interleaving is correct → comparing the result of serial execution

schedule: particular order of interleaving

serial schedule: not interleaved

equivalent schedule: same effect

serializable schedule: equivalent to some serial execution

## Conflicts

RW

* unrepeatable read

WR

* dirty read - reading uncommitted data
* inconsistent read - reading partial commits

WW

* Partially-lost update

conflicts are in both good and bad TXN, should avoid anomalies, not conflicts

Conflict Serializability

* involving the same actions in the same TXN
* conflicting action ordered in the same way
* Schedule S is conflict serializable if S is conflict equivalent to some serial schedule
* conflict serializable = serializable

Conflict serializable = good

Not conflict serializable = bad schedule

The conflict graph

T1 → T2: any actions in T1 precede and conflict with any actions in T2

Acyclic conflict graph = conflict serializable

## Concurrency

2-Phase Locking

* guarantee conflict serializability
* straightforward to implement & transparent to the user

X(exclusive) lock: lock before writing, others cannot obtain any lock

S(shared) lock: lock before reading, others cannot obtain X lock

Growing phase: get new locks, no release lock

Shrinking phase: releasing locks, no new lock

Strict 2PL

* allow only schedules its dependency(conflict) graph is acyclic → serializable schedules → isolation, consistency!

## Deadlock

the cycle of TXN waiting for locks to be released by each other

* deadlock prevention
* deadlock detection
  + waits-for graph, T1→T2: T1 is waiting for T2 to release a lock