Database Management Systems

Database Management System is basically a collection of data related to an enterprise with some programs to manage or manipulate it.

Program | Data

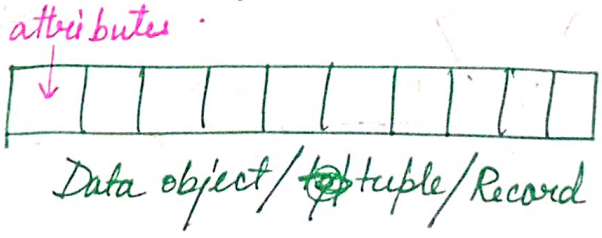
Isolation
Abstraction
(So that the low level details need not be known to the programmer)

— Convenience of Data
— Efficiency

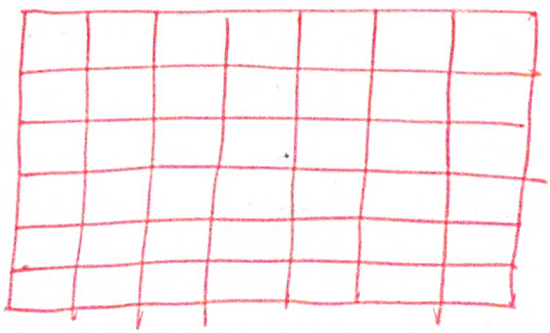
Database Model

• Relational Model

Every single unit of data is represented as a vector, where each component is called an attribute.



Multiple records gives rise to a "data table"



Eg.! Table / Students of DBMS

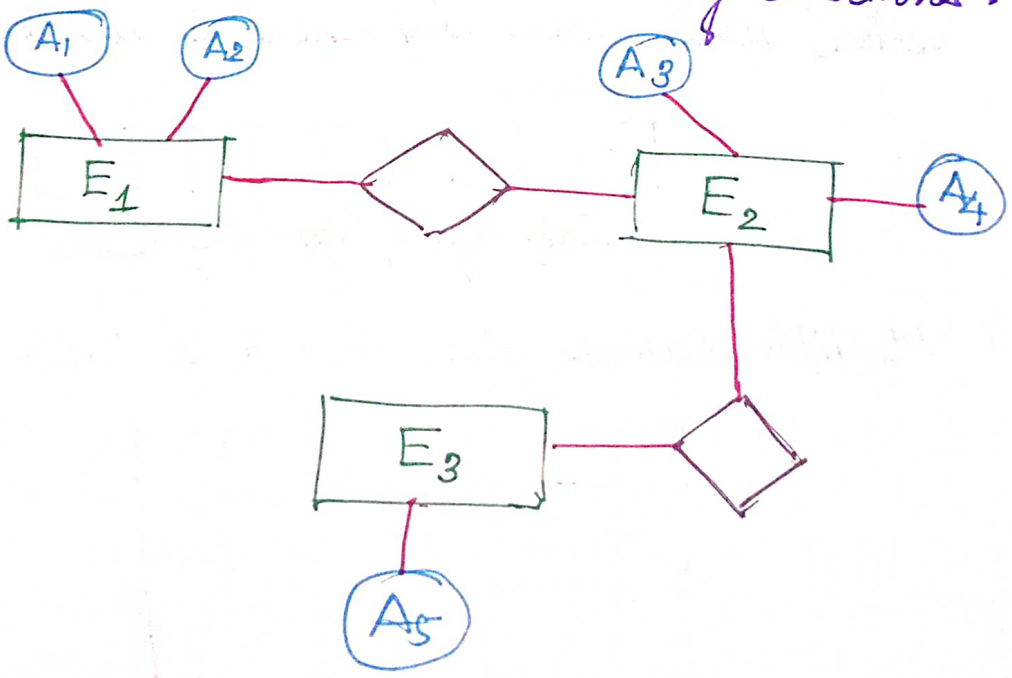
	RN	N	D	CG	SG	H	
S_1							
S_2							
S_3							
S_4							

We can name this relation "DBMS course students"

This is a subset of the cartesian product of everything.

- We will interchangibly use "Table" & "Relation"
- Structure of Record → Schema of data; All schema — Schema of DB
- Entity Relationship Model (ER-Model)

Data is represented in a diagramatic way. There are entities (described by a set of attributes) and these entities are related by relations.



Entity

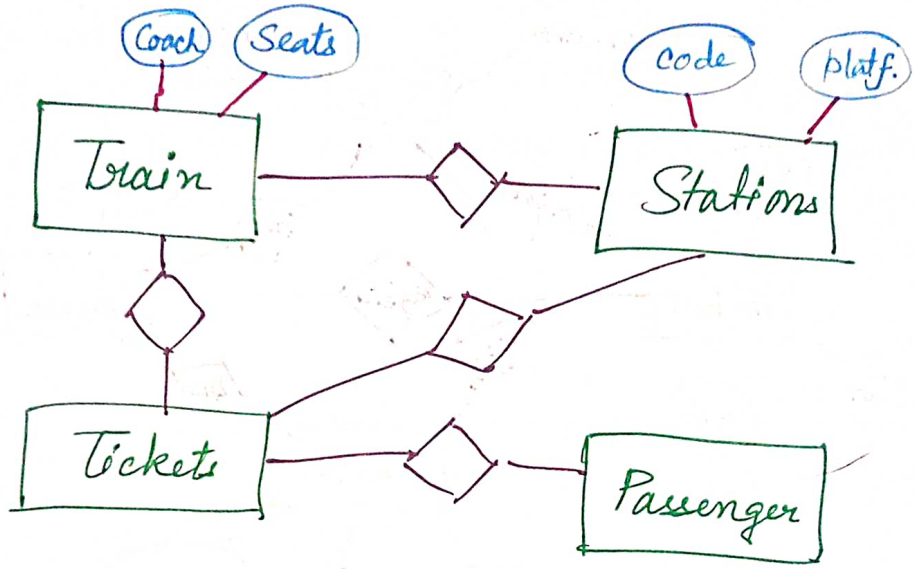
Attribute

Relation

Eg.: ER-Diagram of IRCTC Booking System

Entities

- Train
- Passengers
- Stops/Stations
- Ticket



We can convert an ER-Diagram to a Relational Schema & vice-versa.

(We can have entities as tables & or have a common table entry to identify relation).

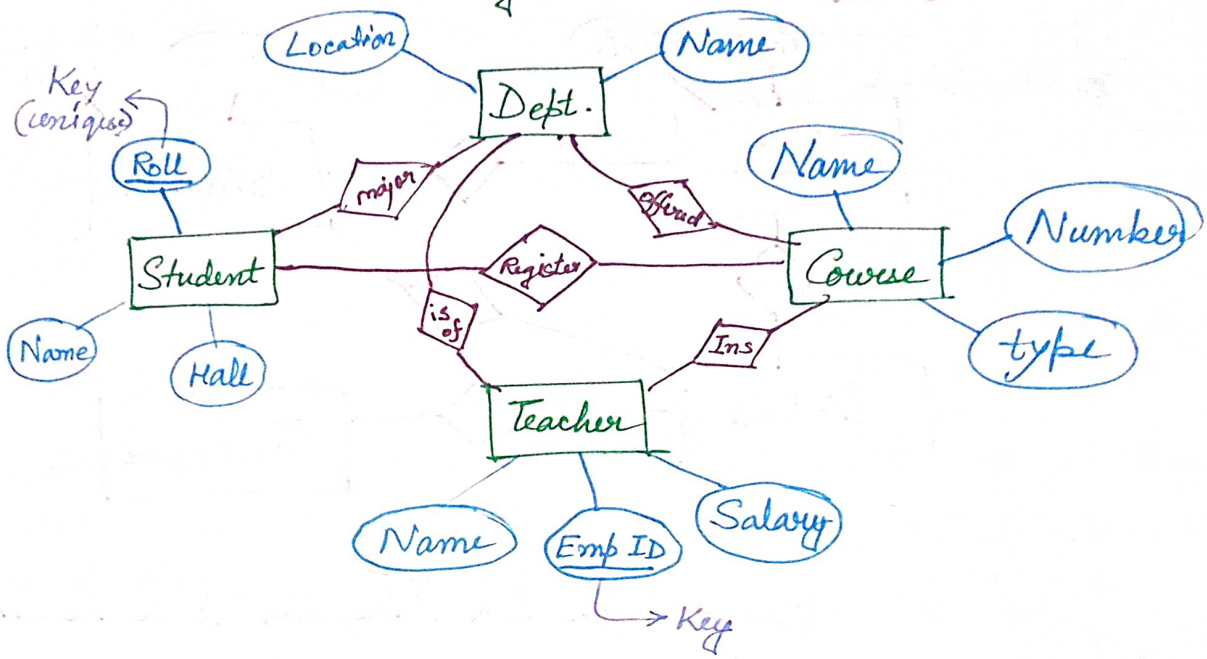
- E-R table & Relational schema is can be repeatation
- Relational schema cannot handle data without well defined set of attributes (unstructured data)
- Relational schema cannot handle data with varying number of attributes.

Semi Structure Data Model (XML)

Some markups or tags are used to describe data.

Used in E-commerce sites as Flipkart or Amazon

ER-Diagram



Step 1: Write down all entities and their attributes.
Step 2: Identify relations and incorporate them in the schema

Key: Unique attribute of an entity (no two instances of this entity has the same key)

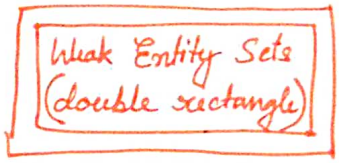
Superkey: Subset of attributes which has unique values for every instance of the entity

Minimal Superkey: Superkey whose no smaller subset as superkey (Candidate key)

- Superkey & Candidate Key are part of the schema
- There can be multiple Candidate Keys.
- Keys, when forced to be unique, ~~are called "Primary Key"~~ and is considered to be a part of the schema

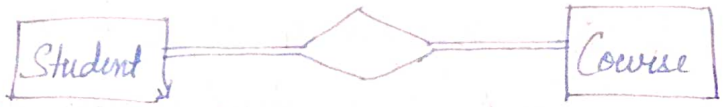
Primary Key: Candidate Key, which is forced to be unique.

Weak Entity Set: Entities which have no candidate key.



Participating Constraints:

(i) Total Participation: All ~~entity~~ instances of the entity has atleast one relation. (double line)



(ii) Cardinality

- Many to Many — —
- One to Many <——
- Many to One ——>
- One to One <——>

Converting ER-Diagram to a Table

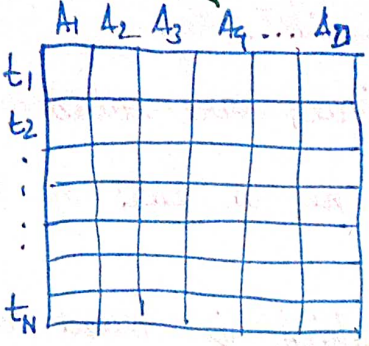


table r

The table/relation schema is specified by

$$r(A_1, A_2, \ldots, A_D)$$

A database schema may be specified by —

$$r_1(A_1, A_2, \ldots, A_{D_1})$$
$$r_2(A_1, A_2, \ldots, A_{D_2})$$
$$\vdots$$

Foreign Key:

r_1

A_2	A_5	A_7	A_8	A_9

r_2

A_1	A_2	A_3	...	A_D

If A_2 is a key for r_1 & r_2 is a weak entity set, then A_2 is a foreign key referencing r_2 from r_1

E.g.:

Courses

Cid	Cname

Student

Roll	Cid

Cid is a foreign key referencing Student table from Courses table

Conversion from ER-Diagram to Table

- Have a table for
  1. Each entity → { Normal : All attributes
     Weak : All attributes, plus attributes of all referencing relations entity (primary key) }
  2. Each relation. (key of both participating entities)

Example:

RELATIONAL ALGEBRA

"Queries" are fired on table (or a set of tables). The answers may also be represented by a table.

result-table = Query(table1, table2, ...);

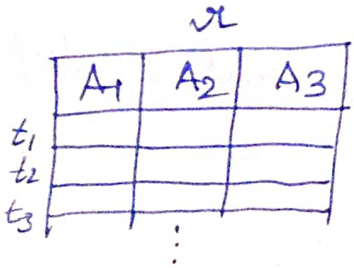
↑ can be specified by some Data Query Language.

Data Query Languages
- Procedural {Relational Algebra}
- Descriptive {Relational Calculus}

We start with Relational Algebra.

RELATIONAL ALGERA

Table ≡ Relation



, ~~r~~ r(A1, A2, A3); t1, t2, t3 ∈ r

This is a relation as, $r(A_1, A_2, A_3) \subseteq A_1 \times A_2 \times A_3$

Now, we consider a relational algebra expression.

$$r_4 = (r_1 \ op_1 \ r_2 \ op_2 \ r_3)$$

In relational algebra, there are six primitive operators.

UNITARY OPERATORS

• "SELECT" operator

$$result = \sigma_{\theta}(r)$$

("clause")

Where θ is a clause, like
- $(A_1 > 30) \ \& \ (A_2 == 0)$
- $(A_3 == 0)$

then, we have result as the set of tuples from r satisfying θ.

$$result = \{ t \in r \mid t \text{ satisfies } \theta \}$$

• "PROJECT" operator

$$result = \Pi_{\{A_{i_1}, \ldots, A_{i_k}\}}(r)$$

result will just be the table with only the columns of the attributes $A_{i_1}, \ldots, A_{i_k}$. Mathematically, removal of duplicates are necessary.

In SQL, implementation does lead to duplicates.

- "RENAME" operator

$\rho_{R_1}(R_2)$; renames R_2 as R_1

Example: student (Name, Roll, Hall, course).

Find the Name & Roll of students from RK, taking the DBMS course.

$$res = \Pi_{\{Name, Roll\}}\left(\sigma_{Hall == 'RK' \ \& \ course == 'DBMS'}(student)\right)$$

BINARY OPERATORS

- "SET UNION" operator

$R_3 = R_1 \cup R_2$ { The attributes in R_1 and R_2 must be the same }

Mathematically, R_3 is without duplicates.

- To save computational power, SQL implements union by allowing duplicates.

- "SET DIFFERENCE" operator

$R_3 = R_1 - R_2$ { Again, the attributes in R_1 & R_2 must be the same }

- "CARTESIAN PRODUCT" operator

$R_3 = R_1 \times R_2$

if $R_1(A_1, A_2, A_3)$ and $R_2(A_4, A_5)$; $R_3 = (A_1, A_2, A_3, A_4, A_5)$

if $R_1(A_1, A_2, A_3)$ and $R_2(A_4, A_5, A_3)$; $R_3 = (R_1.A_1, R_1.A_2, R_1.A_3, R_2.A_4, R_2.A_5, R_2.A_3)$

DERIVED OPERATORS

"NATURAL JOIN"

$$\mathcal{R}_1 \bowtie \mathcal{R}_2 = \Pi_{A_1 \cup A_2}\left(\sigma_{\mathcal{R}_1[A_1 \cap A_2] = \mathcal{R}_2[A_1 \cap A_2]}\left(\mathcal{R}_1 \times \mathcal{R}_2\right)\right)$$

$\mathcal{R}_1$

	A_1	A_2	A_3
t_{11}	a_{11}	a_{21}	a_{31}
t_{12}	a_{12}	a_{22}	a_{32}
t_{13}	a_{13}	a_{23}	a_{33}

$\mathcal{R}_2$

	A_2	A_3	B_1
t_{21}	a_{21}	a_{31}	b_{11}
$\bowtie t_{22}$	a_{22}	a_{32}	b_{12}
t_{23}	a_{21}	a_{31}	b_{13}

$\Downarrow$

$\mathcal{R}_1 \bowtie \mathcal{R}_2$

A_1	A_2	A_3	A_4	
a_{11}	a_{21}	a_{31}	b_{11}	$t_{11} \bowtie t_{21}$
a_{11}	a_{21}	a_{31}	b_{13}	$t_{11} \bowtie t_{23}$
a_{12}	a_{22}	a_{32}	b_{12}	$t_{12} \bowtie t_{22}$

Example:

"DIVISION"

$\mathcal{R}_1 \div \mathcal{R}_2$ ⟶ Take joinable tuples, but take only the attributes in $\mathcal{R}_1$ but not in $\mathcal{R}_2$.

$$\mathcal{R}_1 \div \mathcal{R}_2 = \Pi_{A_1 - A_2}\left(\sigma_{\mathcal{R}_1[A_1 \cap A_2] = \mathcal{R}_2[A_1 \cap A_2]}\left(\mathcal{R}_1 \times \mathcal{R}_2\right)\right)$$

"INTERSECTION" $\mathcal{R}_1 \cap \mathcal{R}_2$

EXTENDED RELATIONAL ALGEBRA

• "OUTER JOIN":

$r_1 \overline{\bowtie} r_2$ (left)

$r_1 \overline{\bowtie} r_2$ (right)

$r_1 \overline{\bowtie} r_2$ (full)

- For $r_1 \overline{\bowtie} r_2$, include $r_1 \bowtie r_2$ and include unjoined tuples in r_1 and append "Null" to attributes from r_2
- Similarly for $r_1 \overline{\bowtie} r_2$ and for $r_1 \overline{\bowtie} r_2$, we do both $r_1 \overline{\bowtie} r_2$ & $r_1 \overline{\bowtie} r_2$.

$$r_1 \overline{\bowtie} r_2 = (r_1 \overline{\bowtie} r_2) \cup (r_1 \overline{\bowtie} r_2).$$

• "AGGREGATION OPERATORS":

$${}_{A_2}\mathcal{G}_{r_1}(A_1)$$

$\mathcal{G}$ – count unique
– average
– max
– min
– sum.

gives the $\mathcal{G}$-aggregation of attribute A_1 grouped according to A_2.

DATABASE CONSISTENCY

Written down in terms of Functional Dependency.

Course_registration

Roll	Name	Cid	Cname

Things which are obvious:

→ If roll is same ⇒ name has to be same

→ If Cid is same ⇒ Cname has to be same

We write these as:

Roll → Name

Cid → Cname

Let $r(R)$ be a table where R is a set of attributes,

"$\alpha \rightarrow \beta$" , $\alpha, \beta \subseteq R$

means

$$[t_1[\alpha] = t_2[\alpha]] \Rightarrow [t_1[\beta] = t_2[\beta]]$$

F: { roll → name, cid → Cname }

F^+ { ... Set of implied dependency ... (e.g. roll, cid → Cname) }

ARMSTRONG'S AXIOMS:

1. Reflexivity:
   $\alpha \rightarrow \beta$ if $\beta \subseteq \alpha$
2. Augmentation:
   if $\alpha \rightarrow \beta$ then $\alpha\gamma \rightarrow \beta\gamma$
3. Transitivity:
   if $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$ then $\alpha \rightarrow \gamma$

} All that is needed,

Let $r(R)$ be a table where R is a set of attributes,

"$\alpha \rightarrow \beta$" , $\alpha, \beta \subseteq R$

means

$$[t_1[\alpha] = t_2[\alpha]] \Rightarrow [t_1[\beta] = t_2[\beta]]$$

F: $\{$ roll → name, cid → Cname $\}$

F^+ $\{$ ... Set of implied dependency ... (e.g. roll, cid → Cname) $\}$

ARMSTRONG'S AXIOMS:

1. Reflexivity:
   $\alpha \rightarrow \beta$ if $\beta \subseteq \alpha$
2. Augmentation:
   if $\alpha \rightarrow \beta$ then $\alpha\gamma \rightarrow \beta\gamma$
3. Transitivity:
   if $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$ then $\alpha \rightarrow \gamma$

} All that is needed,

* Union Rule:

if $\alpha \to \beta$ and $\alpha \to \gamma \Rightarrow \alpha \to \beta\gamma$

Proof:

$\alpha \to \beta$ and $\alpha \to \gamma$

$\Rightarrow \alpha, \alpha \to \beta.\alpha$ and $\beta, \alpha \to \beta, \gamma$

$\Rightarrow \alpha \to \alpha\beta$ and $\alpha\beta \to \beta\gamma$

$\Rightarrow \alpha \to \beta\gamma$.

* Pseudotransitivity:

if $\alpha \to \beta$ and $\beta\gamma \to \delta$ then $\alpha\gamma \to \delta$

Proof:

$\alpha \to \beta$ and $\beta\gamma \to \delta$

$\Rightarrow \alpha\gamma \to \beta\gamma$ and $\beta\gamma \to \delta$

$\Rightarrow \alpha\gamma \to \delta$

* Decomposition:

if $\alpha \to \beta\gamma$ then $\alpha \to \beta$ (and $\alpha \to \gamma$)

Proof:

$\alpha \to \beta\gamma$ and $\beta\gamma \to \beta$

$\Rightarrow \alpha \to \beta$.

- If $A \to R$ where R is the set of all attributes, then A is a candidate key.

CLOSURE OF AN ATTRIBUTE SET α

α^+ is the set of attributes in R such that $\alpha \rightarrow \alpha^+$, with respect to F^+, the closure of a functional definition, then α^+ is the closure of the attribute set α.

- If $\alpha^+ = R$, then α is a candidate key.

- If there exists F' (simpler than F) such that

$$F'^+ = F$$

then checking database consistency of F' suffices. This F' is called a 'canonical cover'.

F' is a canonical cover of F, if there does not exist F'' such that

$$F'' \subseteq F' \text{ and } F''^+ = F'^+$$

whereas $F'^+ = F^+$.

Example:

$$F = \left\{ \begin{array}{l} A \rightarrow B \\ B \rightarrow C \\ A \rightarrow BC \end{array} \right\} \Longrightarrow \text{then } F' = \left\{ \begin{array}{l} A \rightarrow B \\ B \rightarrow C \end{array} \right\}$$

Roll	Name	Cid	Cname
101	AB	CS1	PDS
101	AB	CS2	Algo
102	AG	CS1	PDS

Example : Roll → Name ∧ Cid → Cname

Extraneous Attribute :

Consider three functional dependencies :

FD1 : $\alpha \rightarrow \beta$
FD2 : $\beta \rightarrow \gamma$
FD3 : $\alpha \rightarrow \beta\gamma$

FD3 is implied from FD1 & FD2, we can drop γ from FD3 and it becomes FD1. Here γ is an extraneous attribute.

Obtaining the Canonical Cover F^c of F

~~1. Apply Union Rule (apparently to get closure)~~
~~2. Check for extraneous attributes on LHS & RHS remove all the~~

1. Apply Union Rule and add only the unions to F^c
2. Check for extraneous attributes on LHS & RHS, and remove this.

REDUNDANCIES IN TABLES

Faculty

	Name	Dept	Salary	Dept_addr	Ph_no
1	AB	CS	1000	Nalanda	123
2	AG	PHY	1000	Takshila	124
3	PM	PHY	1000	Takshila	125
4	NG	CS	1500	Nalanda	126
5	PM	CS	1000	Nalanda	125

Anomalies:

- **Update Anomaly:**
  - Update at one location may cause inconsistency.
- **Insertion Anomaly:**
  - Insertions causing inconsistency.
- **Deletion Anomaly:**
  - Deletion of one row may cause unnecessary loss in information.

* The following decomposition may be helpful.

Faculty

Name	Dept	Salary	Ph_no

Department

Dept	Addr

(Should be Lossless)

* "Lossy join" —— Cannot recover from the join
* "Lossless join" —— Possible to recover original table after naturally joining the tables.

Lossless Decomposition:

If R is split into R_1 & R_2, then it forms a lossless join, if —

- $R_1 \cap R_2 \neq \phi$
- Either $R_1 \cap R_2 \longrightarrow R_1$
  or $R_1 \cap R_2 \longrightarrow R_2$

[which is basically

$$\{R_1 \cap R_2 \to R_1 \,,\, R_1 \cap R_2 \to R_2\} \cap F^+ \neq \phi$$]

Lossless decomposition makes sense only under a set of functional ~~decomposition~~ dependencies.

Dependency Preservation

$R = R_1, R_2, \ldots, R_n$ be a split and F be the set of functional dependencies.

$$F_i := \{ [\alpha \to \beta] \in F \mid \alpha, \beta \subseteq R_i \}$$

This decomposition is Dependency Preserving if

$$(F_1 \cup F_2 \cup F_3 \ldots F_n)^+ = F^+$$

or $$(F_1 \cup F_2 \cup F_3 \ldots F_n)^c = F^c \quad \ldots\ldots (?)$$

A Decomposition must be

- Dependency Preserving
- Lossless join.

Algorithm to check if $(F_1 \cup F_2 \cdots F_n)^+ = F^+$:

for $\alpha \to \beta \in F - (F_1 \cup F_2 \cup \cdots \cup F_n)$

$\alpha_0 := \alpha$

for i in 1 to n

$\alpha_i :=$ closure of α_{i-1} in F_i

~~$\alpha^+ :=$ closure of α in F~~

if $\beta \notin \alpha_n$

return NOT_DP

return DP

SCHEMA REFINEMENT

We do a set of decomposition until we reach a "normal form"

Normal Forms:

"In a nutshell, these are non-redundant form"

• First Normal Format (1-NF):

A table is in 1-NF if it does not contain any multivalued or composite attribute.

[Just split up into another column if needed]

• Boyce Codd Normal Form (BCNF)

A relation $R(R)$ is in BCNF if for every FD $\alpha \to \beta$ in F^+ the following holds true.

$\alpha \to \beta$ is trivial or α is a superkey of R.

Example:

Faculty

Name	Dept	Salary	Ph

Dept

dept	address

~~Name → Dept~~

Name → Phone

Name → Salary

dept → address

NOT IN BCNF

Conversion to BCNF:

- Split R into R_1, R_2 for $\alpha \to \beta$ which are violating (non-trivial & α is not superkey) make $R_1(\alpha\beta)$ and $R_2(R-(\beta-\alpha))$

Further split R_1 and R_2 if needed for other violating dependencies.

* BCNF decomposition is always a Lossless join but may not be dependency preserving.

* BCNF can still have "transitive redundancy"

Transitive Redundancy:

In a set of FD F if

$\alpha \rightarrow \beta \in F^+$

$\alpha \rightarrow \gamma \in F^+$

$\beta \rightarrow \gamma \notin F^+$

then F consists transitive redundancy.

Third Normal Form (3NF):

A table $r(R)$ is in 3NF under a set of dependencies F, if $\forall \alpha \rightarrow \beta \in F^+$,

$\alpha \rightarrow \beta$ is trivial or α is a part of a candidate key.

Example:

Faculty (Name, dept, salary, address, ph) has candidate key (Name, dept), then under

$F = \{$

Name → ~~Address~~ Phone

Dept → ~~Address~~

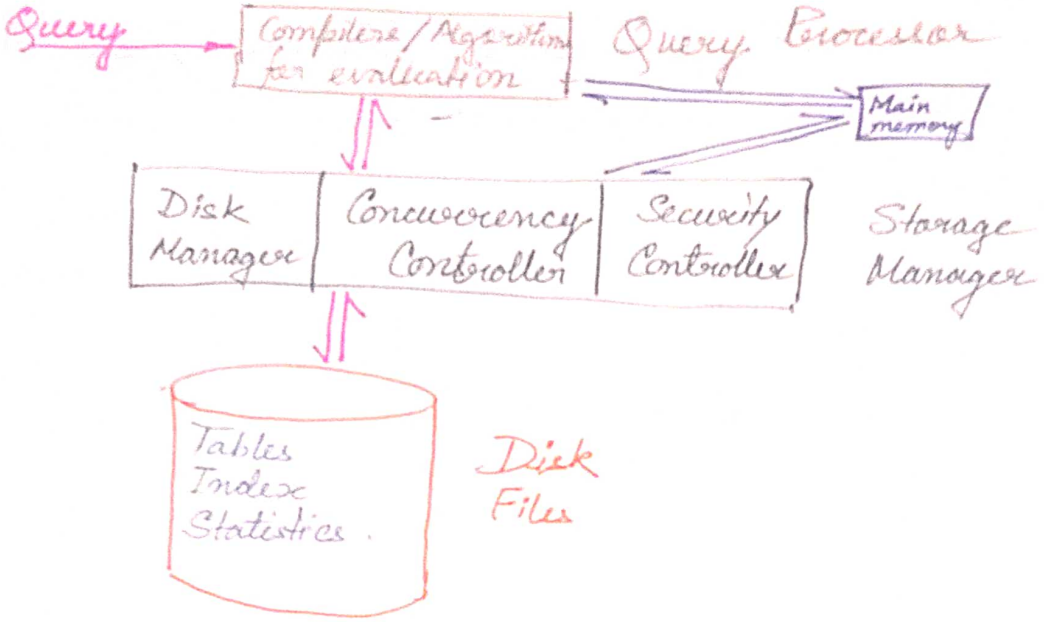
$\}$

is not in BCNF but is in 3CNF.

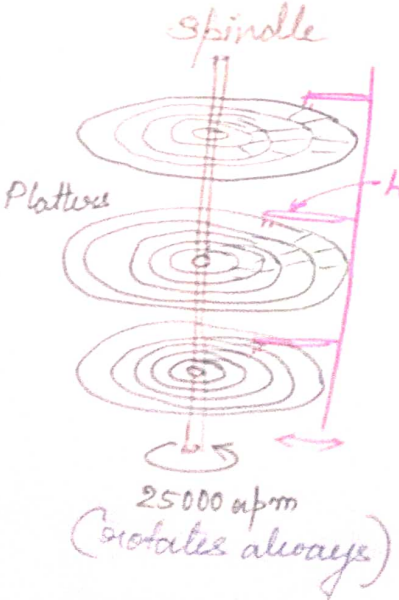
3. CNF decomposition

Take union $\alpha \rightarrow \beta$ and make tables for each L.H.S.

PHYSICAL DATABASE DESIGN



Disk



(Controlled by disk controller)

Seek-time → Time required for head to move to right track (~4 ms)

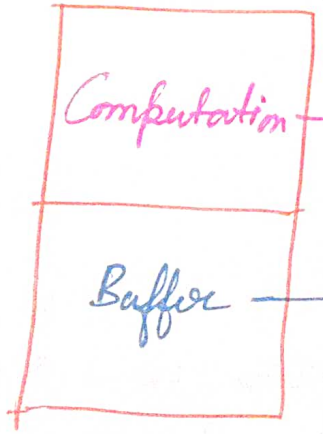
Rotation Latency → Time required for the disk to rotate & the correct sector to arrive (~2 ms)

Disk read/write time is so bad, that any query algorithm is evaluated on the bases of disk read/write.

DBMS (not OS) does the following:

- Disk Controller Algorithm
- Memory Management
- Parallel Read/Write.

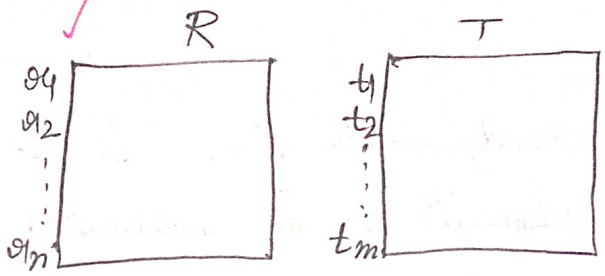
Main Memory



Computation — CPU is currently using.

Buffer — Extra data from Disk (like Cache).

- DBMS does not follow LRU scheme, because of:
  - Consistency
    - * Pinned Block
  - Optimality



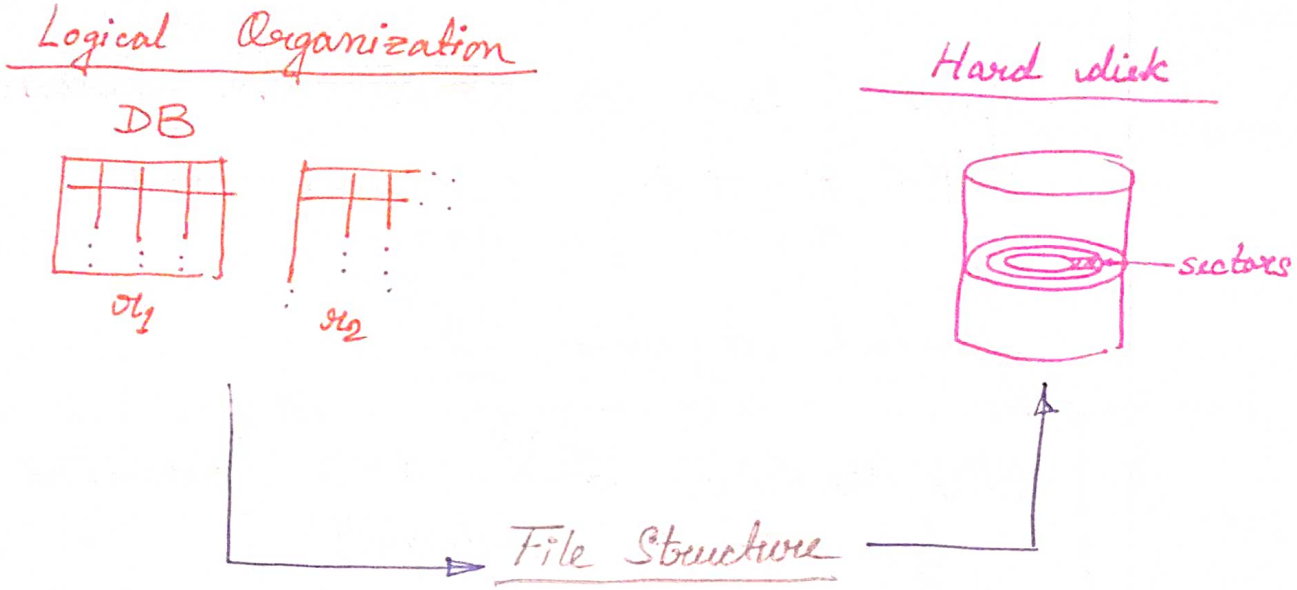
$R \bowtie T$

algo:

```
for (r from r1 to rn)
    for (t from t1 to tm)
        if r joins t
            output r, t
```

LRU is not optimal here.

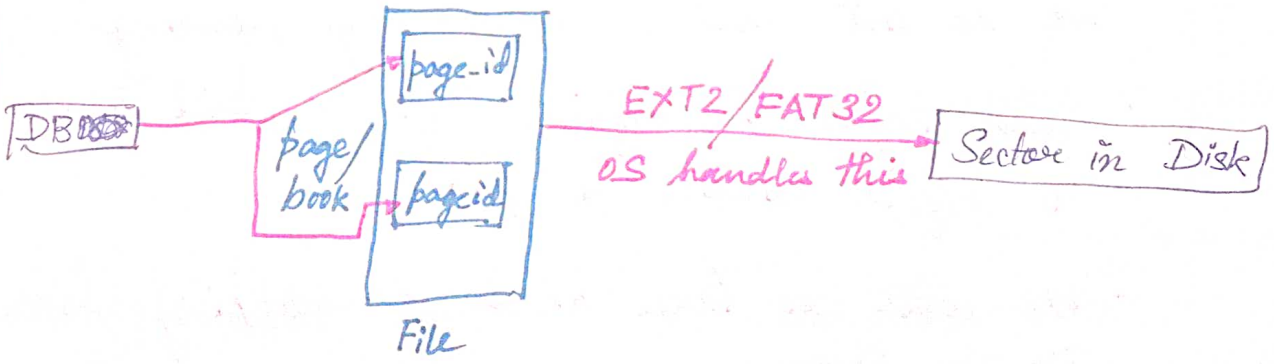
FILE STRUCTURE



- MongoDB has column oriented file structure (efficient for narrow tables
- We will study tuple oriented file structure

Tuple Oriented File Structure

- Each file has multiple pages/books.



- page id (pid) is mapped to location at disk.
- pid is unique across all files.
- Mapping from DB to ~~file~~ pages is DBMS dependent

Multi Table Clustered File

Files contain multiple tables.

- Good if tables are small. join operations might be easy.

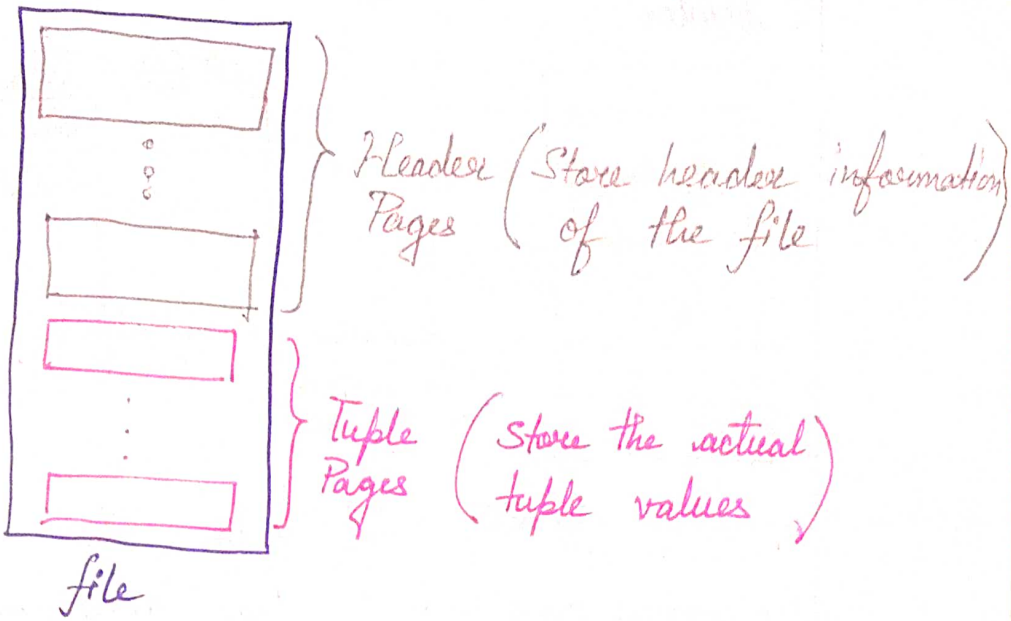
Single table in single file

- disk/Hardware ~~disk~~ page — 4KB (correctness guaranted by hardware)
- Memory page — 4KB (no guarantee)
- File page — 512 B to 16 KB (no guarantee)

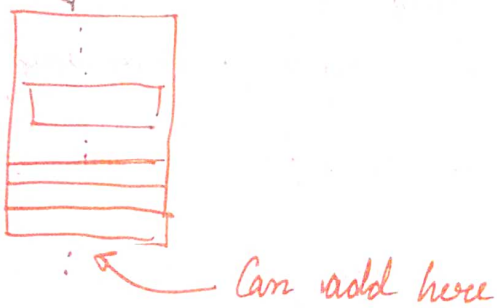
[SQLite page size — 512B
MySQL page size — 16 KB
Postgres page size — 8KB]

- Due to the nature of relational ~~db~~ algebra, we do not need to follow any ordering.
- Each tuple identified by a pid and its offset within a pid.
- We will see three kinds of mapping from DB to pages
  - Heap file structure (most popular)
  - Sorted file structure
  - Block file structure.

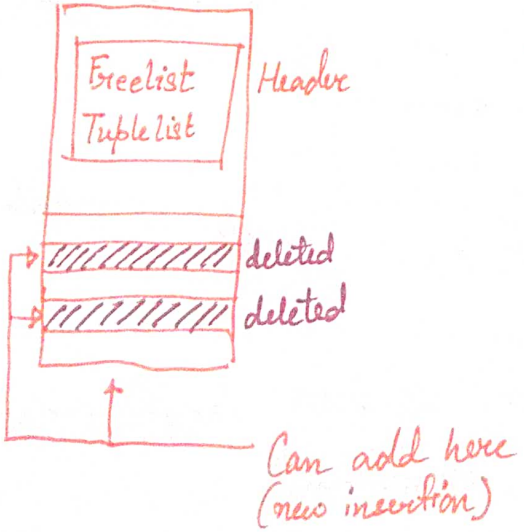
Heap File Structure [Assumption: One page file at least one tuple]



° Adding Tuples



° Deletion of Tuples

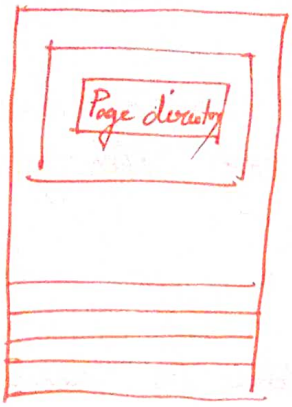


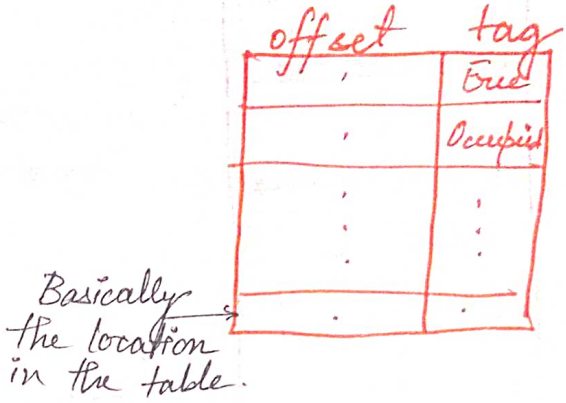
Freelist: Linked list containing the location of all free space~~list~~ (including unallocated space)

Tuplelist: Linked list containing the location of all occupied space

- Updates are costly, hence this is not done! (o_O, o_O)

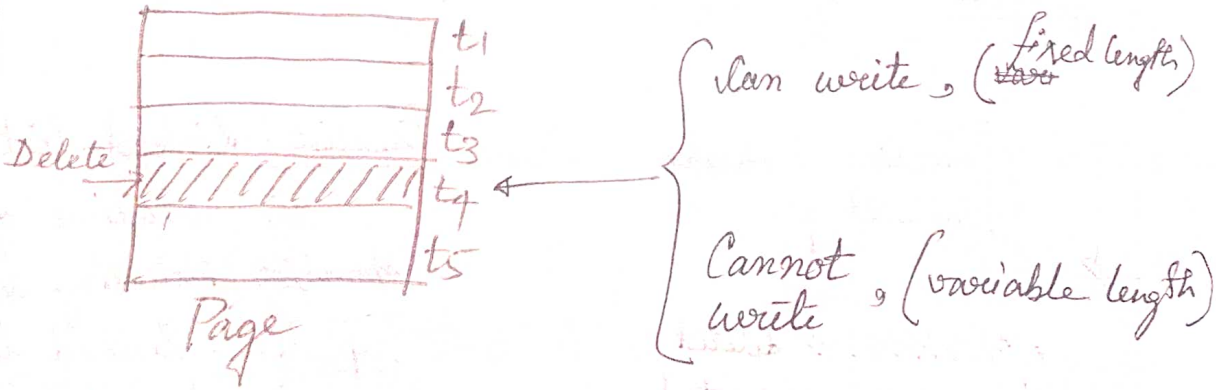
- Approach 2: Keep a "Page Directory" in the header





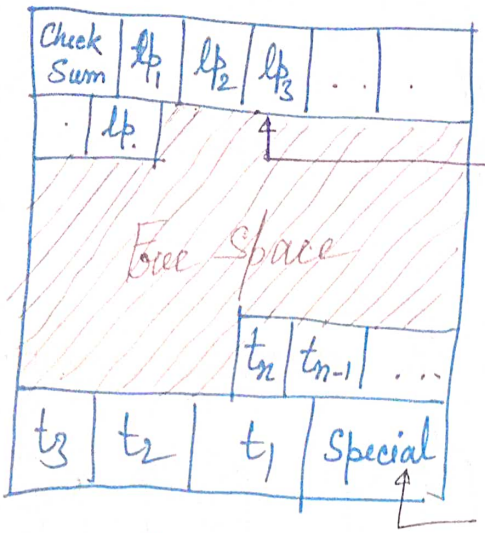
- Turns out that for large files, the memory overhead of Page directory is very less
- This "Page Directory" is small and can be kept in buffer (in memory).
- This, is what is done! ☺

Structure within the pages



- One structure for variable length records is "Slotted Page Structure"

Slotted Page Structure



line pointer
(tuple_number, offset, length)

Used for recovery of a page

- Within a page, each tuple is given a number
- When the tuples move around, only the offset changes in the corresponding line pointers

Can try on postgres

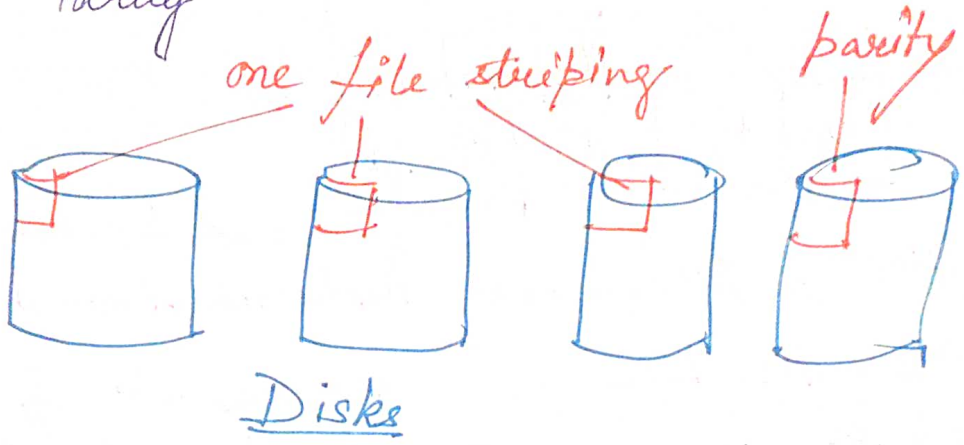
```
> CREATE EXTENSION page_inspect.
> SELECT get_raw_page ('pg_class', 0)
```

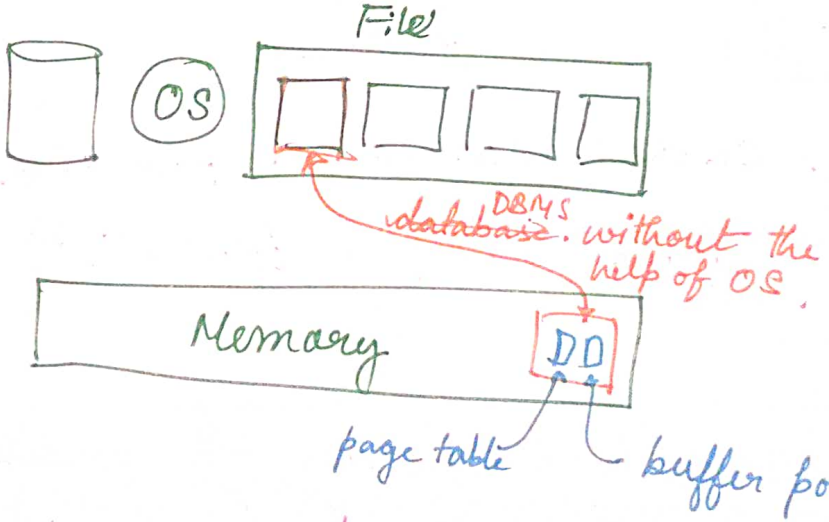
Redundant Array of Independent Disks (RAID)

Advantages of multiple disks
- Parallel Read/Write
- Reliability (by redundancy)

RAID5

- Striping
  - bit level
  - page level
- Parity





In postgres you have a command called CREATE BUFFERPOOL name size

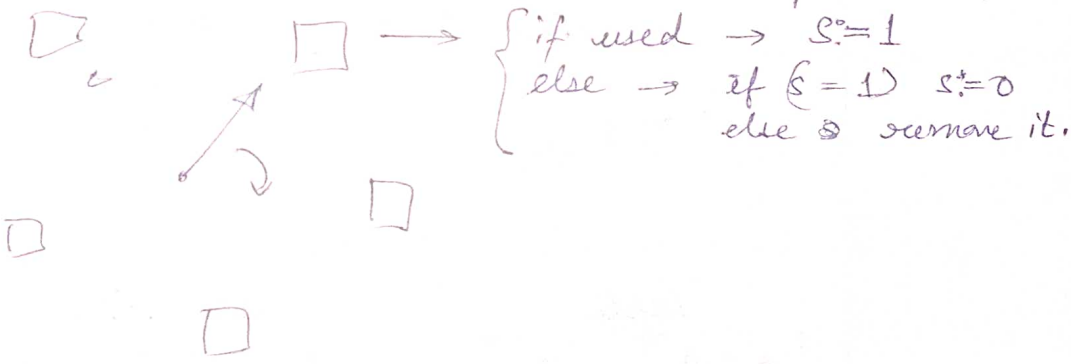
There are things which are needed to be kept in mind (for multithread access or multiple query queries)

- Consistency
- Efficiency.

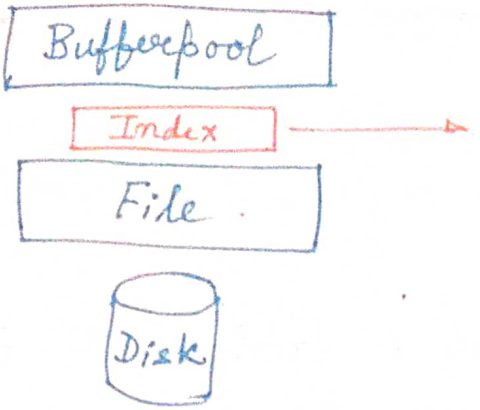
We use something called latch (shorter than locks) which says that you cannot change buffer ~~there~~ without making it durable.

Strategies used in Cache

- LRU (not always optimal)
- Toss immediate (similar to MRU).
- Clock sweep LRU. (approximate version of LRU, easier to implement)



- LRU-K (maintain timestamp of last K accesses, estimate time of next access).



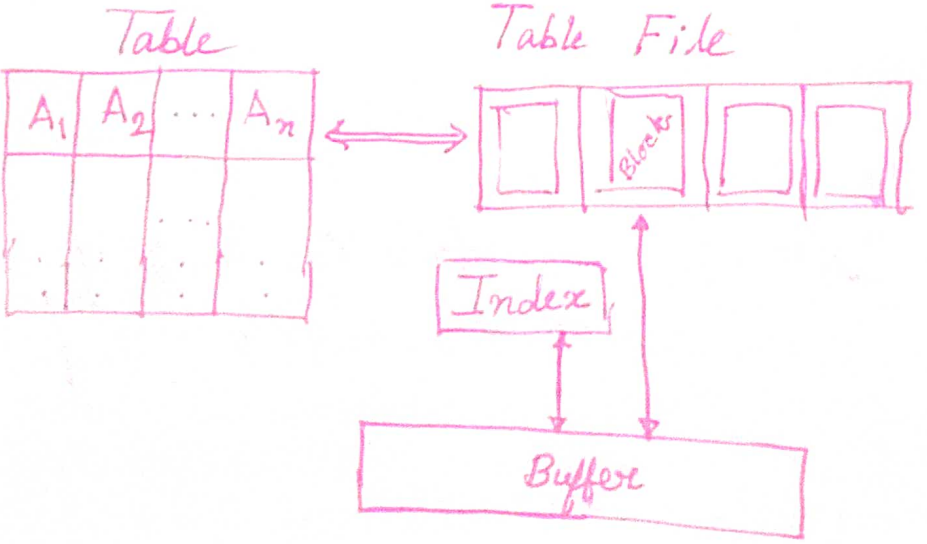
/* OK, sir now moved to slides 😐 */

Index file

search-key	pointer

- If search-key is sorted → ordered index
- search-keys uniformly distributed into buckets using hash function → hash index

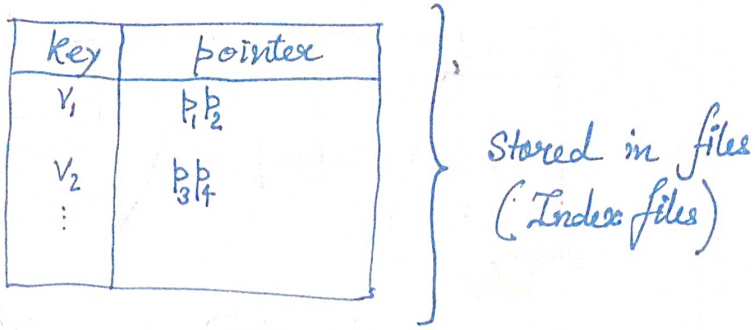
/* More slide stuff */



Structure of Index

Structure of index :

(key, pointer) pairs.



Command in postgres.

In this episode of 'You have a Command in Postgres, we learn

```
CREATE INDEX I1 USING A1 OF R1
```

*Note:

- Buffer management ~~is~~ of Index files is same as Table files
- Index files ~~needs~~ need to be maintained.
- Index files are usually smaller than table files

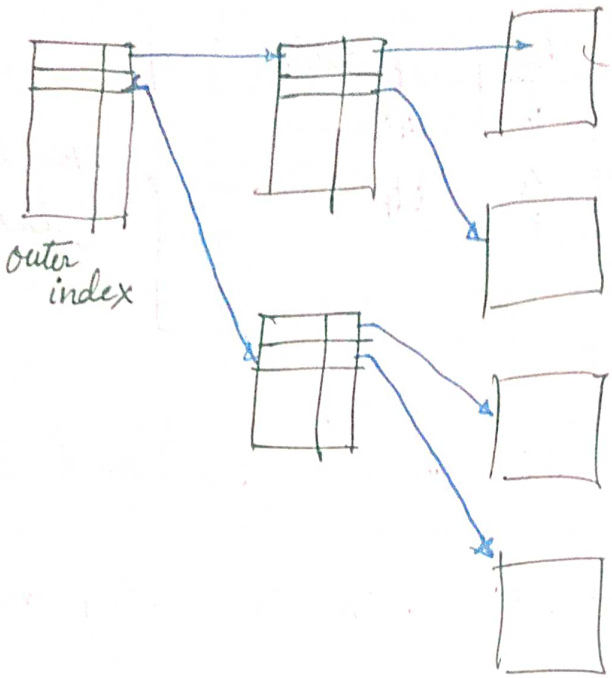
Index Evaluation Metrics :

- Access type supported
- Access. time

/*
and other things I am too slow too copy ☺
*/

Key insight: What if we keep an index of the index file?

We get a multilevel index!



We get a tree structure (yay)

"BST sounds good, but it needs to be stored in memory".

Use B^+ Trees!

B^+ tree:

- Spanning more
- Height less
- Each node is the size of a block (disk block)
- Each node is an array of key-pointer pairs.

properties:

- Balance: Each path from root to leaf are of the same length (self balancing)
- Fix n, the order of the tree.
  - A node has between n & $\lceil \frac{n}{2} \rceil$ pointers
  - ~~A~~ children.
  - A node has between $(n-1)$ and $\lceil \frac{(n-1)}{2} \rceil$ values.
  - n is decided on the basis of disk block size & the key-pointer pair size.

$n \rightarrow$ order, $R \rightarrow$ number of records

Search: ~~log~~ $O(\log_{\frac{n}{2}} R)$

Delete: $O(\log_{\frac{n}{2}} R)$

Insert: $O(\log_{\frac{n}{2}} R)$

Number of block I/O (reads/writes).

Values are sus.

Observations: (Rather design properties).

~~"Einstein's children all less than~~

"Einstein's child are all less than Einstein"

- Leaf nodes point directly to records (or a "bucket of pointer").
- All values in a node are sorted.
- pointers pointing to a child from a (pointer, key) pair, all ~~this and~~ ~~value~~ keys in the children are less than ~~equal to~~ the original key. ~~It is~~ Values to "to the right" are greater or equal to the original key. (non-leaf).

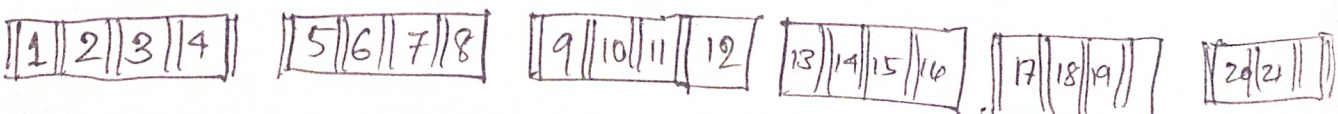
"Let us build a tree ☺"

Example: $n = 4$, entries $= \{1, 2, \ldots, 21\}$

"build bottom up".

/* much confusion in class */

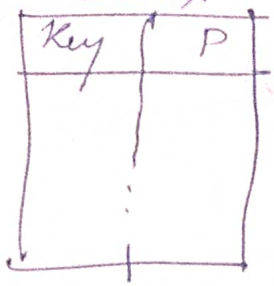
Fed up; not writing anything, go read ~~slides~~ some other source!



Data Access

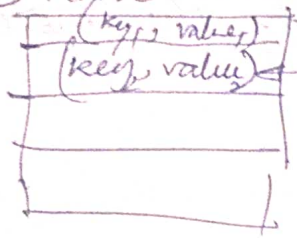
⊛ Key — fast → Record location in disk.

⊛ Index



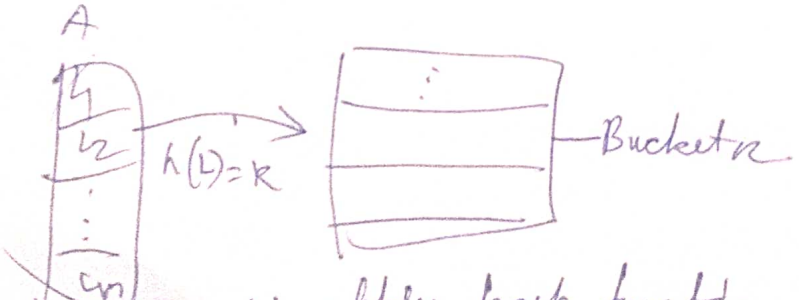
ordered

⊛ Hash Table



Buckets (typically size of a disk block

- The value itself can be a set of records (if tuples are small enough). Then its called a hash file
- But more commonly, there are pointers pointing to disk blocks containing the data (called hash index).



(i) apply hash function
(ii) probe into bucket.

Currently accepted best hash function: XXHash3 (given by Facebook)

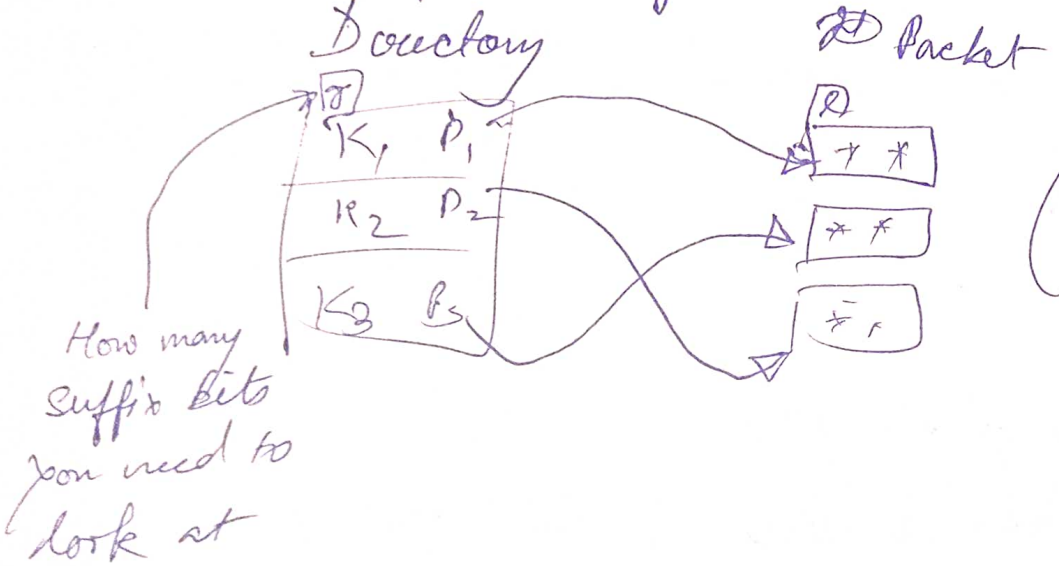
Google → FarmHash.

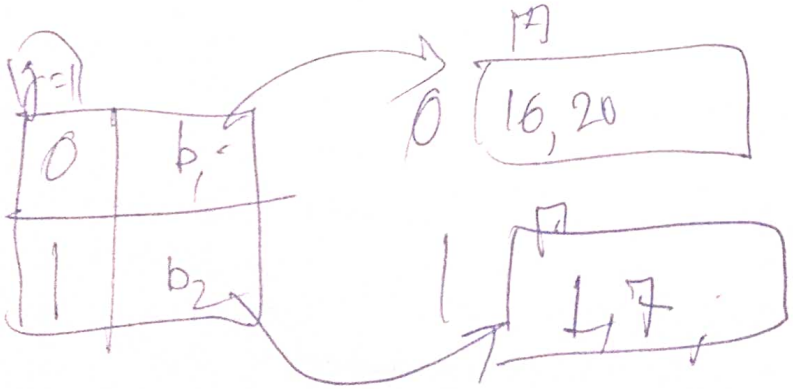
Dynamic Hashing:

Overflow Bucket

Dynamic Hashing:

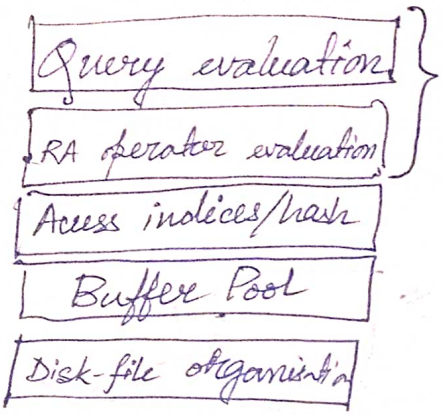
Extensible Hashing:





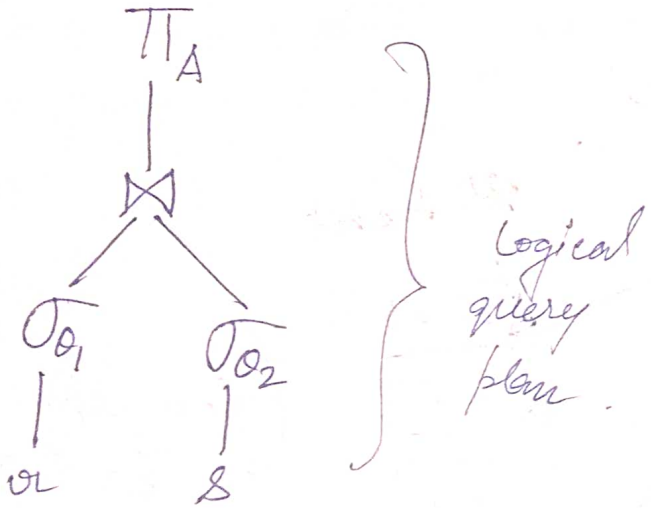
when overflow occurs, double g.

DBMS



Parse tree:

$$\Pi_A(\sigma_{\theta_1}(\mathcal{R}) \bowtie \sigma_{\theta_2}(\mathcal{S}))$$



Cost: Number of disk I/O's.

Optimizer tries to find the least cost tree (NP-Hard)

Physical Execution Plan

(Annoted parse tree containing algorithms & indices)

~~Disk Co~~

Disk Cost

- (# Seek) * (avg seek time)
- (# Blocks read) * (avg read time)
- (# Blocks written) * (avg. ~~block~~ write time).

Evaluation of operators

σ_θ

θ ⟶ match selection

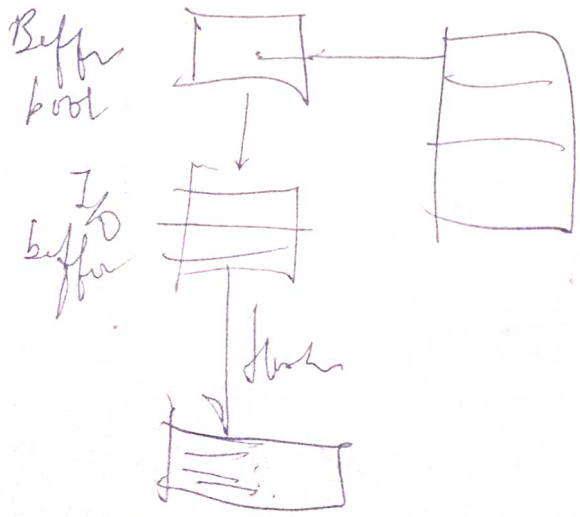
$(A = 10)$

θ ⟶ Range selection

$A > 10$

θ ⟶ complecated selection $\theta_1 \wedge \theta_2 \wedge \theta_3$.

Match selection



~~Range~~ Range query

(easy using B+ tree)

Conjunction -

$\sigma_{\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_n}(r)$

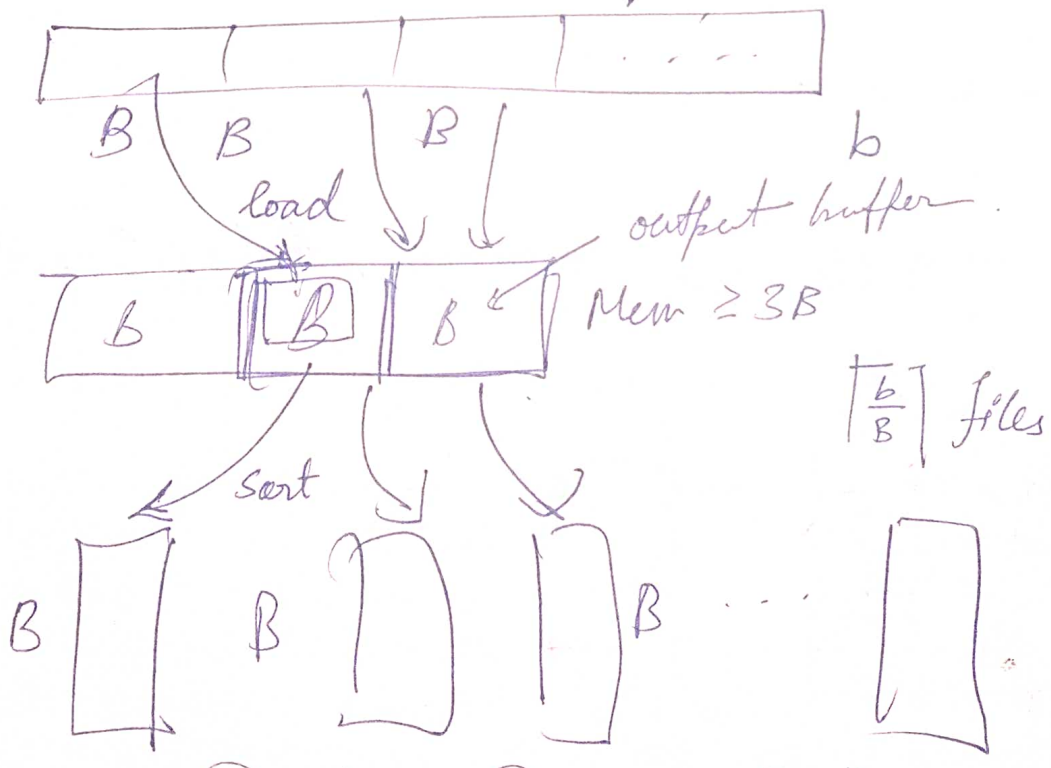
check for the smallest θ_i's, take ~~tuples~~ pointers to buffer, and only satisfied ~~pairs, then~~ see other conditions in buffer.

Sorting

Memory — Megabytes

Files — Terabytes.

Hack: Chop up the memory into runs.



$\{b(t_T + t_S)\} \log_2 b$

Join Operation

(*) Stupid nested loop join
— as name implies.

(*) Block Nested Loop joint.
— just keep loop on blocks.
— indexed file in inner loop.

(*) Merge - join
— merge on sorted attributes
(if indexed, you move down the index, instead of the attribute value).
~~index~~

(*) Hash - join:

[Sorts cannot be inplace, or anything.
→ other queries may be reading the same file]

DBMS
20/08/23

Query SQL

↓

Query Rewrite RA

↓

Query Transformation RA

↓

Query Plan ——— Cost Estimate → Statistical hard

- Computationally hard.
- Optimization Algorithm
- Best plan. (NP Complete)

Prefer $\sigma_{\theta}(\sigma_{\theta_2}(E))$ rather than $\sigma_{\theta_1 \wedge \theta_2}(E)$

Query Optimization

1. Generate Equivalent Algebra Expressions.
2. Annotate with execution plan
3. Select the best plan

3.1 Use some Heuristic Rules

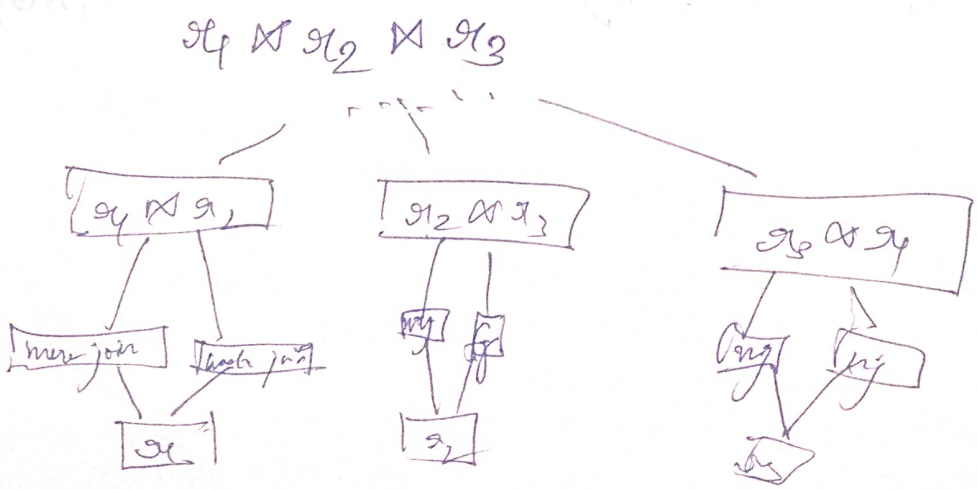
- move down σ / Π
- sequentialize conjunctive predicates.

"Selectivity of a predicate for a table is $\frac{|\text{output table}|}{|\text{original table}|}$"

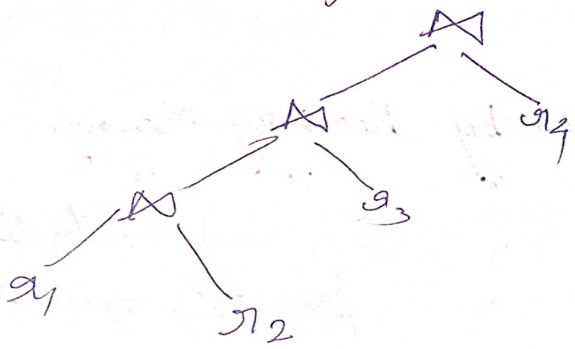
- replace σ-$\times$ by $\bowtie_\theta$

3.2 Cost based optimization.
(Combination of search & heuristic. Join operations are explored, σ & Π are not explored).

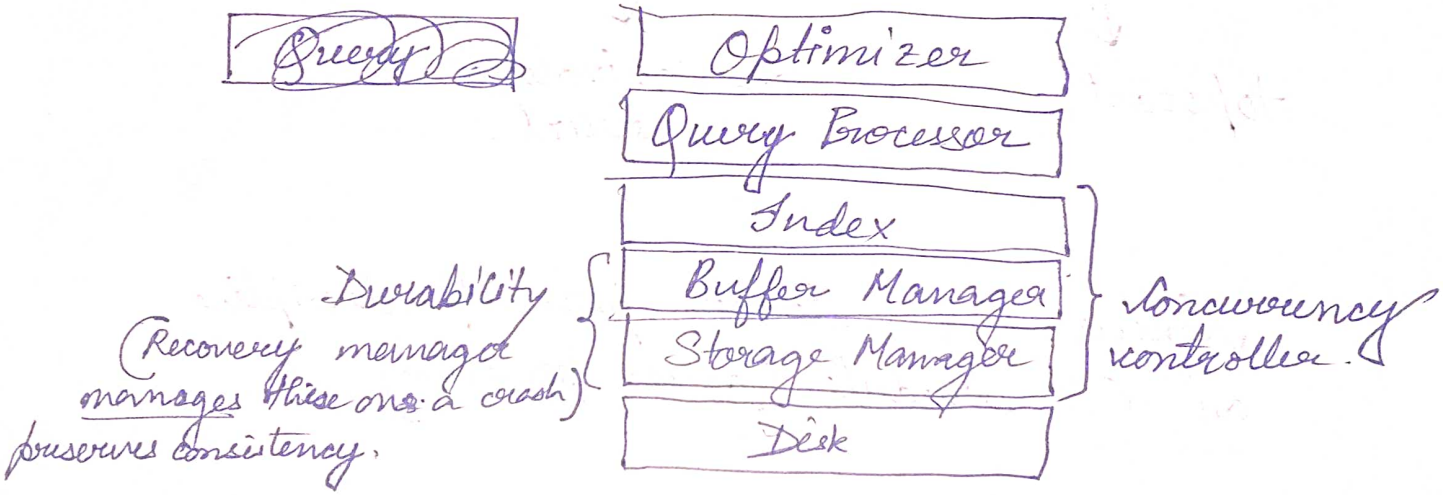
Selinger style optimization (bottom up):



Produces left-diff. join tables



Interesting join orders — Some orders already have something sorted.



Transaction Concept

- Set of instructions that must be executed as a unit
- ACID properties.

Atomicity ensured by Recovery Manag.
Durability " " " "
Isolation " " Concurrency Control
Consistency " " ~~Error~~ programmer.

Precedence graph is dag ≡ Conflict Serializable

↓ ↓

toposort ⟶ equivalent serial.

Concurrency Controller puts up discipline on OS schedule (called protocols)

TRANSACTIONS

Transfer 1000 Rs from account A to B

Transaction

START
Read(A)
A = A - 1000
Read(B)
B = B + 1000
Write(A)
Write(B)
COMMIT

- Transactions start with START
- Transactions end with COMMIT / ABORT

Schedule
Conflict equivalance } Much Concepts

Dependency Graph

T_1 precedes T_2 in a conflicting pair of operations.

"Pessimistic protocols are popular & easy to implement"

↓

Popular: Locking Protocol

Locking Protocol

lock - X(A) - unlock (exclusive)

lock - S(A) - unlock (shared)

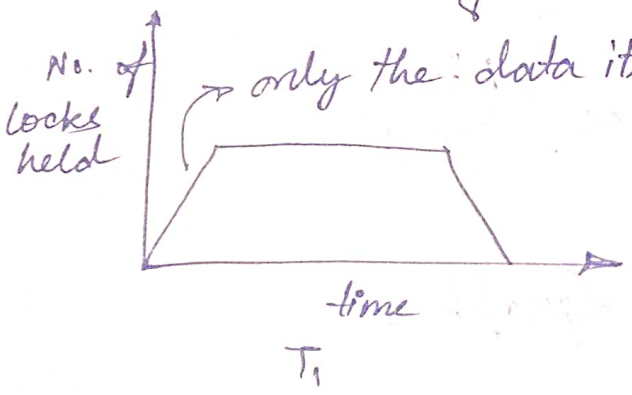
T_2 \ T_1	S	X
S	✓	x
X	x	x

"There is a 'lock manager' which maintains a 'lock table'"

↓ maintains some kind of priority queue.

~~Transactions~~

Two Phase Locking (2PL)



T_1

Read(A)
Read(B)
A = A + 1000
B = B - 1000
Write(A)
Write(B)

lock-X(A)
lock-X(B)
Read(A)
Read(B)
A = A + 1000
B = B - 1000
Write(A)
Write(B)
Unlock(B)
Unlock R(A)

"Any protocol which is Two-phase locking are conflict serializable".

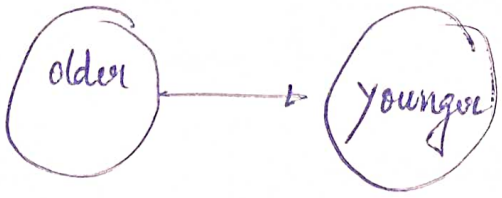
- may lead to cascaded & rollback & deadlocks
- serial → sorted according to unlocking time.

Timestamp Based Concurrency Control

time-stamp order = serializability order.

(Read) Write, Timestamp (Q) → Time stamp of youngest transaction that has written (read), it successfully.

[There can never be cycles in the precedence graph].



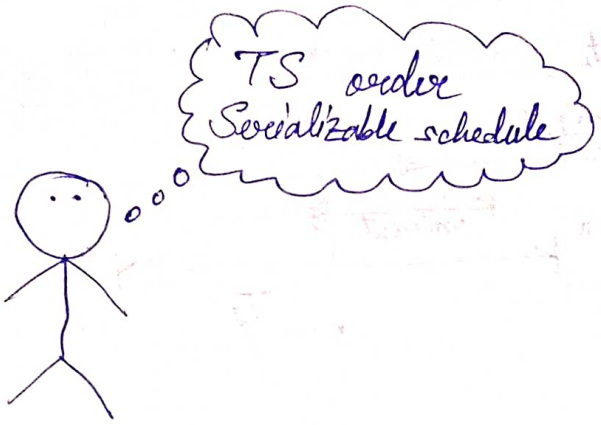
CONCURRENCY CONTROL.

SQL
Normalization } Concepts.

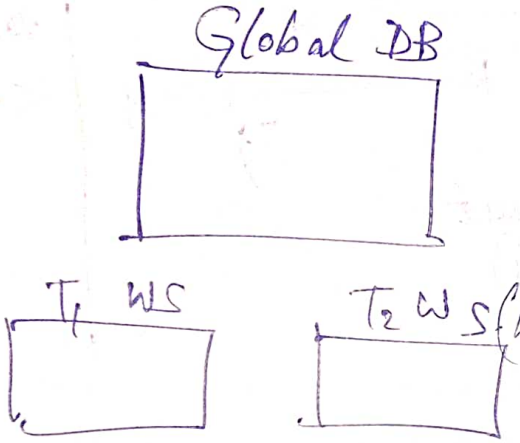
Disk Management.
Buffer Mangement
Disk Controllere.
File Organization.
Indexing → Def/ B+ tree
(including dynamic hashing)
↓
extensible

heavy
- Query Processing → Join, select mergesort.
- Query Optimization
  - Basic notions of cost based optimisation
  - Parse tree
  - which tree is better
  - ~~which tree has better~~
  - to how calculate cost

heavy
- Transaction
  - ACID Properties
  - Concurrency
  - Serializability, Recoverable, Cascaded roll back in schedule.
  - → 2PL
  - → Timestamp
  - → Validation
  - Recovery → Log based,



Concurrency Controller



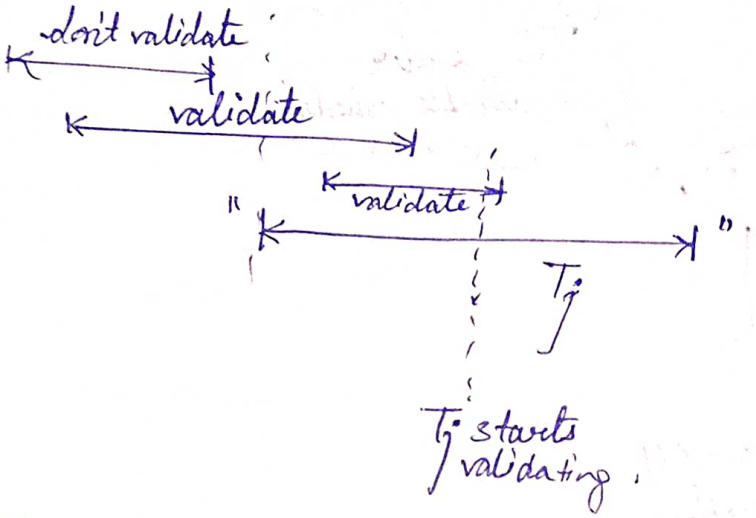
"If validation fails, rollback (i.e. give a new timestamp)"

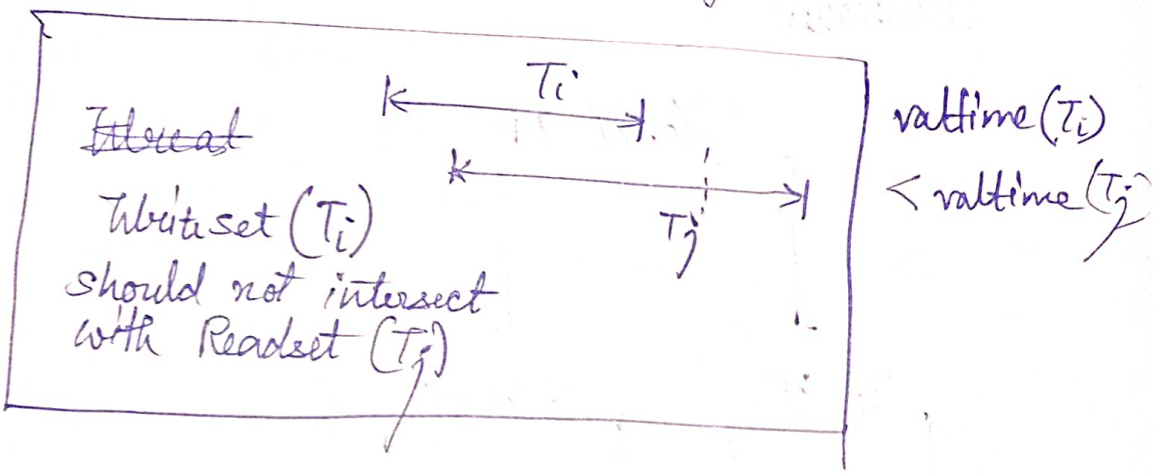
"Every Transaction has three phases".

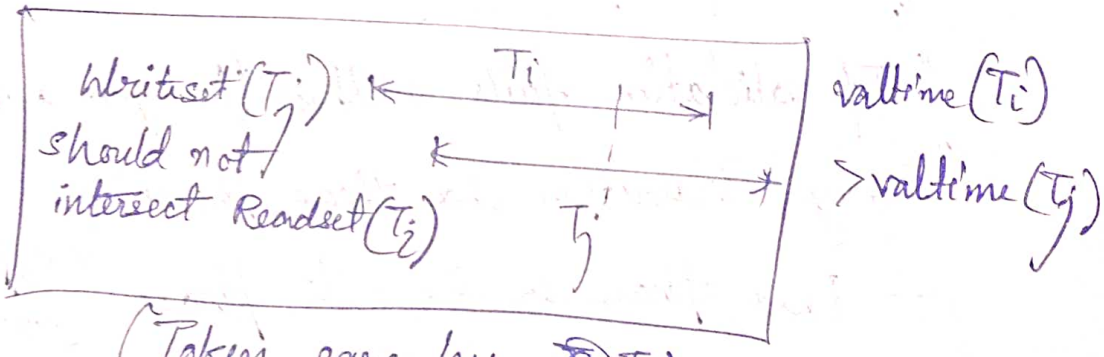
- Read ~~phase~~ & Execute phase (Copy to local Workspace).
- Validation phase (Atomic)
- Write phase (Atomic) on entire WS.

} entirely atomic sometimes.

"Now validation timestamp is THE timestamp instead of start timestamp. Thus, ~~the~~ longer transactions no longer starve, at the cost of the extra workspace!"



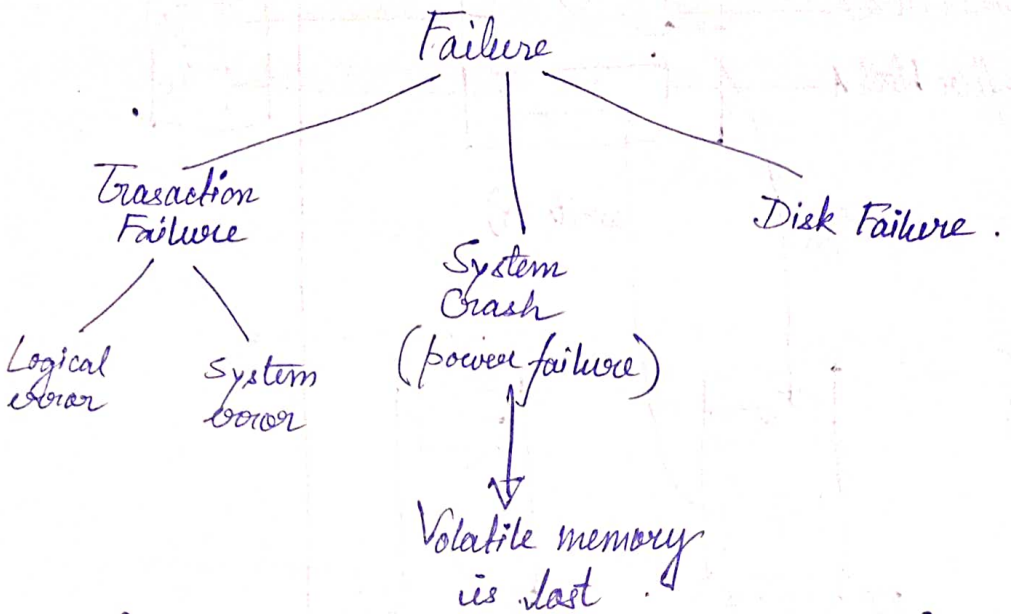


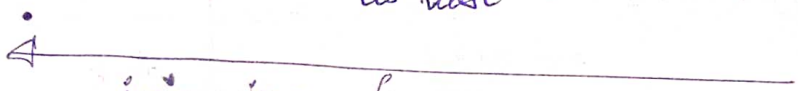


(Taken care by T_i)

"Snapshot → any transaction trying to write to a data item, creates a version."

Recovery System





increasing frequency.

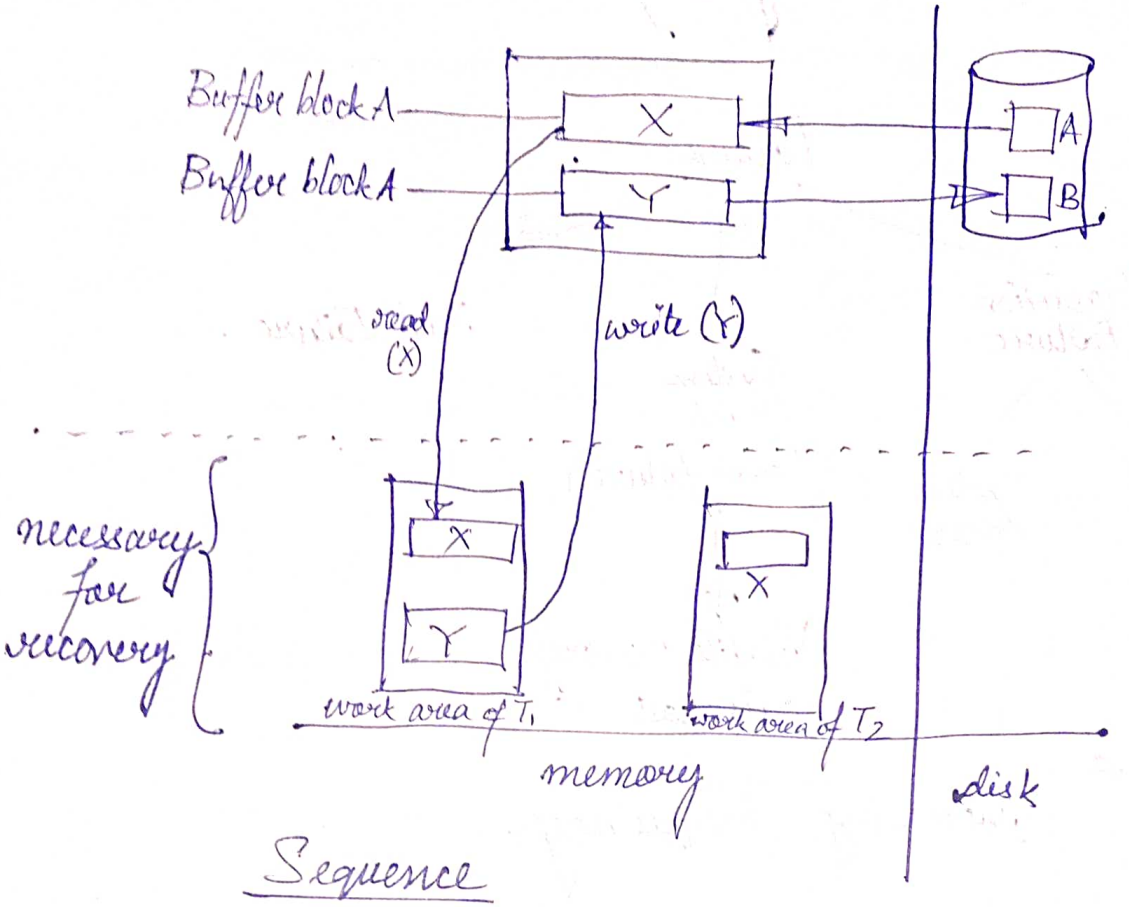
and consistency.

Recovery System: (Atomicity & Durability)

Recovery Algorithm → What to do in normalcy
→ What to do on crash.

Stable Storage: → Without change, data stays here till infinity, without failure.

→ Implemented by multiple disks, possibly seperated geographically.



Sequence

- First perform input
- Then read/write
- Then output.

Log based recovery.

- Before writing, write to log ~~enbe~~ file about what I want to write.
- The log file must be written to stable storage, and only then is it allowed to actually perform the write

Log Records

- $< T_i, \text{start} >$: Transaction T_i start.
- $< T_i, X, V_1, V_2 >$: Transaction T_i wants to write X from old value V_1 to new value V_2
- $< T_i, \text{commit} >$: T_i finishes the last statement

Changes: Local Workspace → Global Workspace → Disk

Two approaches

- Immediate database modification
- ~~Deff~~ Deferred database modification.

New definition of Commit: All previous log records have been written.

idempotent
- undo(T_i): → change value to old value
  → add compensatory record that undo has been done
- redo(T_i): → ~~xxx~~ keep to same new value
  → compensatory record of redo

undo(T_i) → ~~redo~~ undo($L_{i_{k-1}}$), undo($L_{i_{k-2}}$) ... undo(L_{i_0})

redo(T_i) → redo(L_{i_0}), redo(L_{i_1}), ..., redo($L_{i_{k-1}}$)

→ When we want to rollback a transaction.

When crash occurs

- Look at the log records that were written just before crash
- undo transactions, having a start but no ~~end~~ commit.
- redo transactions having ~~but~~ both a start & commit.

The compensatory log records are used when system crashes while recovering.

Checkpoints

- Hold all active transaction
- Output all log records from main memory to stable storage.
- Output all modified buffers to the disk
- Write a log record <checkpoint L> onto stable storage where L is a list of all active transaction
- All updates are stopped while doing checkpointing.