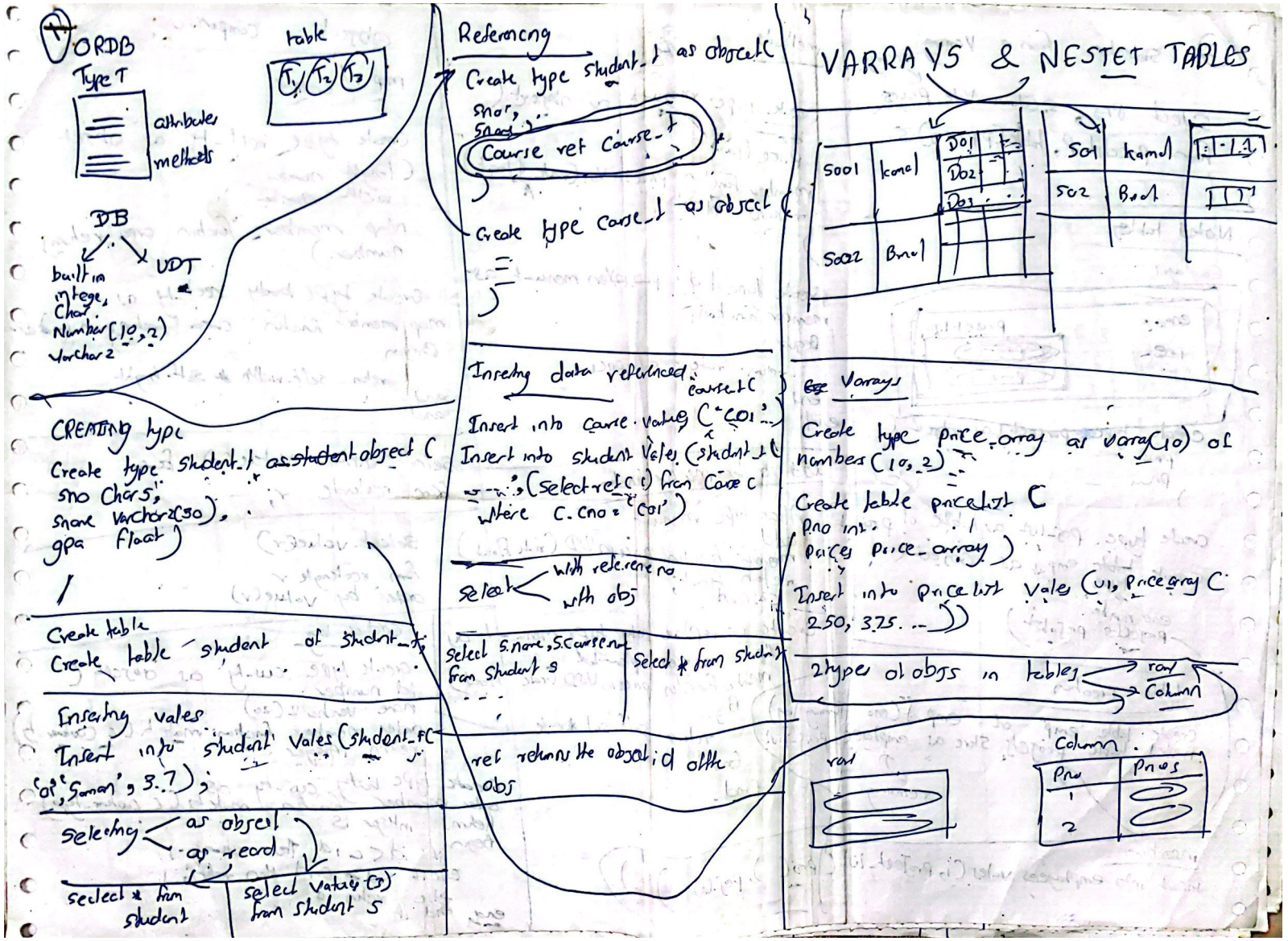
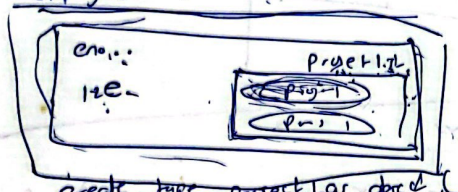


② Selection from a varray renaming.

Select pno, S.column_value Prices
from pricelist P, table(p.prices) S;

Nested tables

Employee



Create type project_tar object (
pno —
pname —
)

Create type proj_list as table of proj_t
Create table emp_t as object (
type
eno number,
projects proj_list)

table creation

Create table 'emp' of emp_t (eno
nested table projects store as employee_proj_list
renaming
Primary key)

insertion

Insert into employees values (1, project_list (proj(..), proj(..) ..))

methods

Create type menu_t as object (
bar
price float,
member function price_in_yen (rate float)
returns float
)

Create type body price_yen menu_t as
member function
Begin
return rate * self.price
end
end

add another member function

Alter type menu_t
add
member function price_in_USD (rate float)
return float;
cascade.

Create or replace body type menu_t as
//price in yen implement
member function price_in_USD (rate float)
Begin
return self.sal * rate
End
End.

object comparison:

map / order

Create type rect_ty as object
(length number,
width number,
Map member function area return
number.)

Create type body rect_ty as
map member function area return number
Begin
return self.width * self.height;
end
end

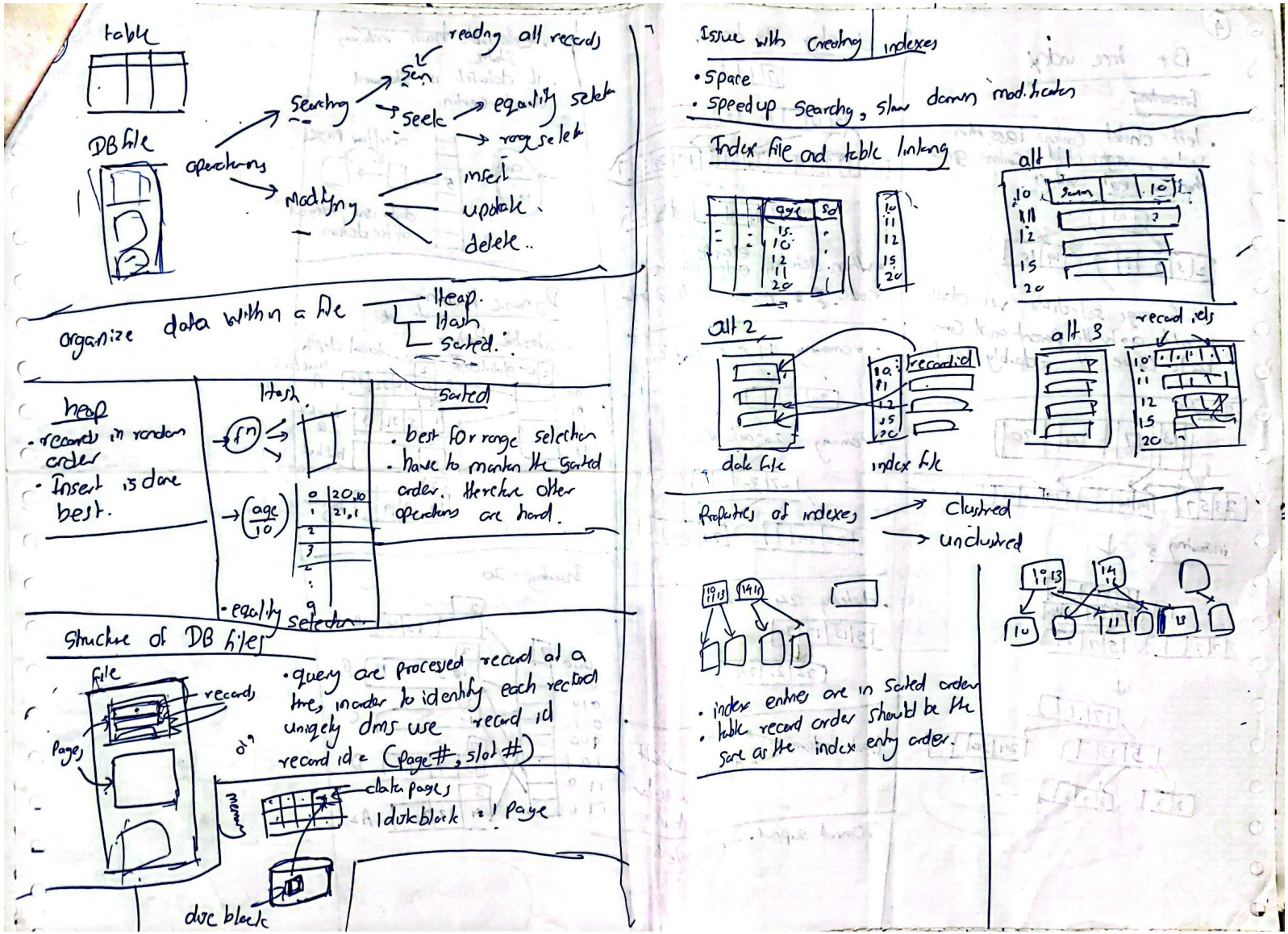
Select distinct value(r) from
from rectangle r;

Select value(r)
from rectangle r
order by value(r)

order by

Create type cus_ty as object (
id number
name varchar2(20)
order member function match(c cust_ty)
return integer

Create type body cus_ty as
order member function match(c cust_ty)
Return integer is
Begin
if id < c.id then return -1
else if id > c.id then return 1
else return 0;
End if
end

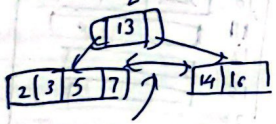


(4)

B+ tree index

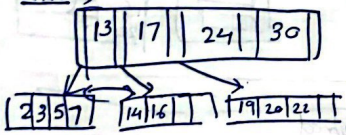
Inserting

- left child contains less than value, right child contains greater than or eql values

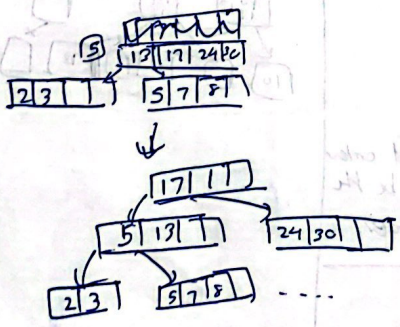


- In range selection we don't need to go to the parent and come back cause of a doubly link list.

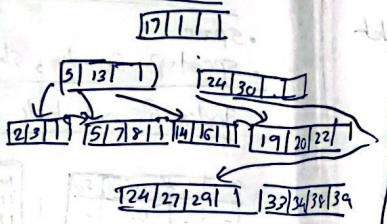
Inserting



inserting 8 ↓

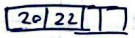


deleting B+ tree

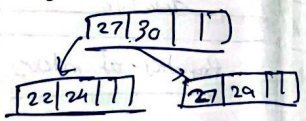


- When deleting element order of the node $\geq \lceil \frac{n}{2} \rceil$ no of elements $\geq$ order

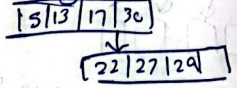
- removing 19 is no problem



- removing 20 after that



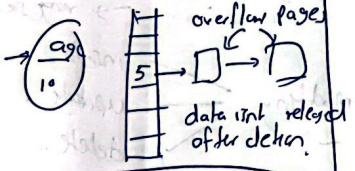
- deleting 24 after that



Hash indexes → Static
→ dynamic → Ext
→ linear hashing

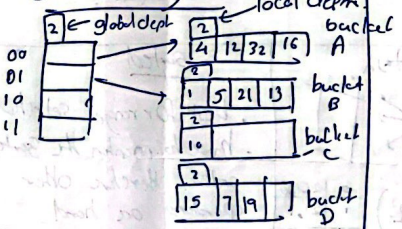
- Cannot support range selection

- ~~Extendable~~ hash indexes Static
- If data is skewed Grand Cerlen

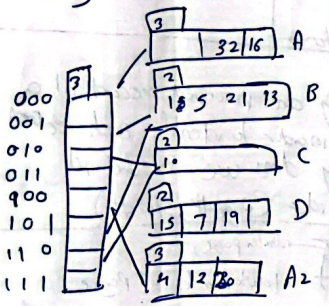


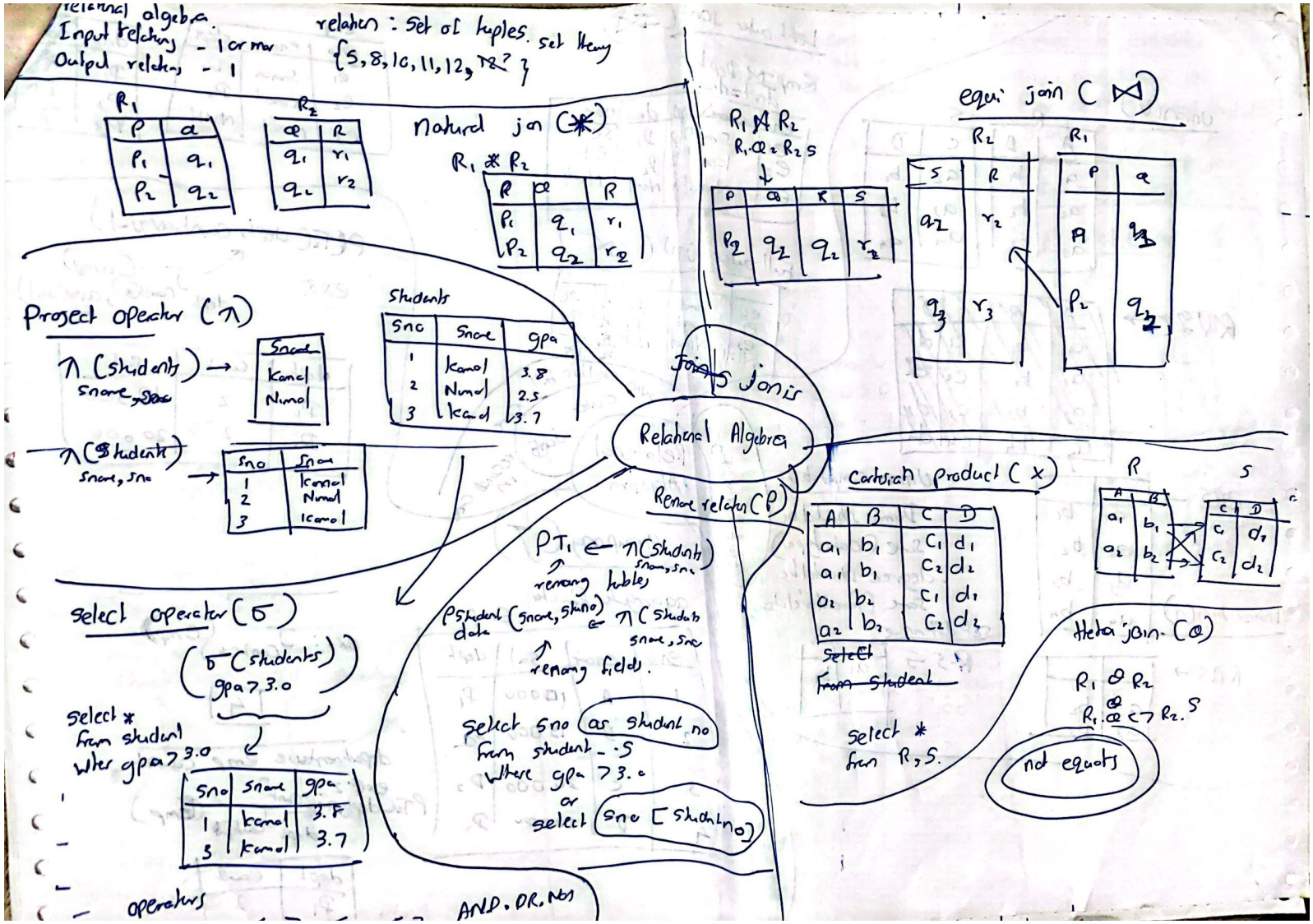
Dynamic hashing

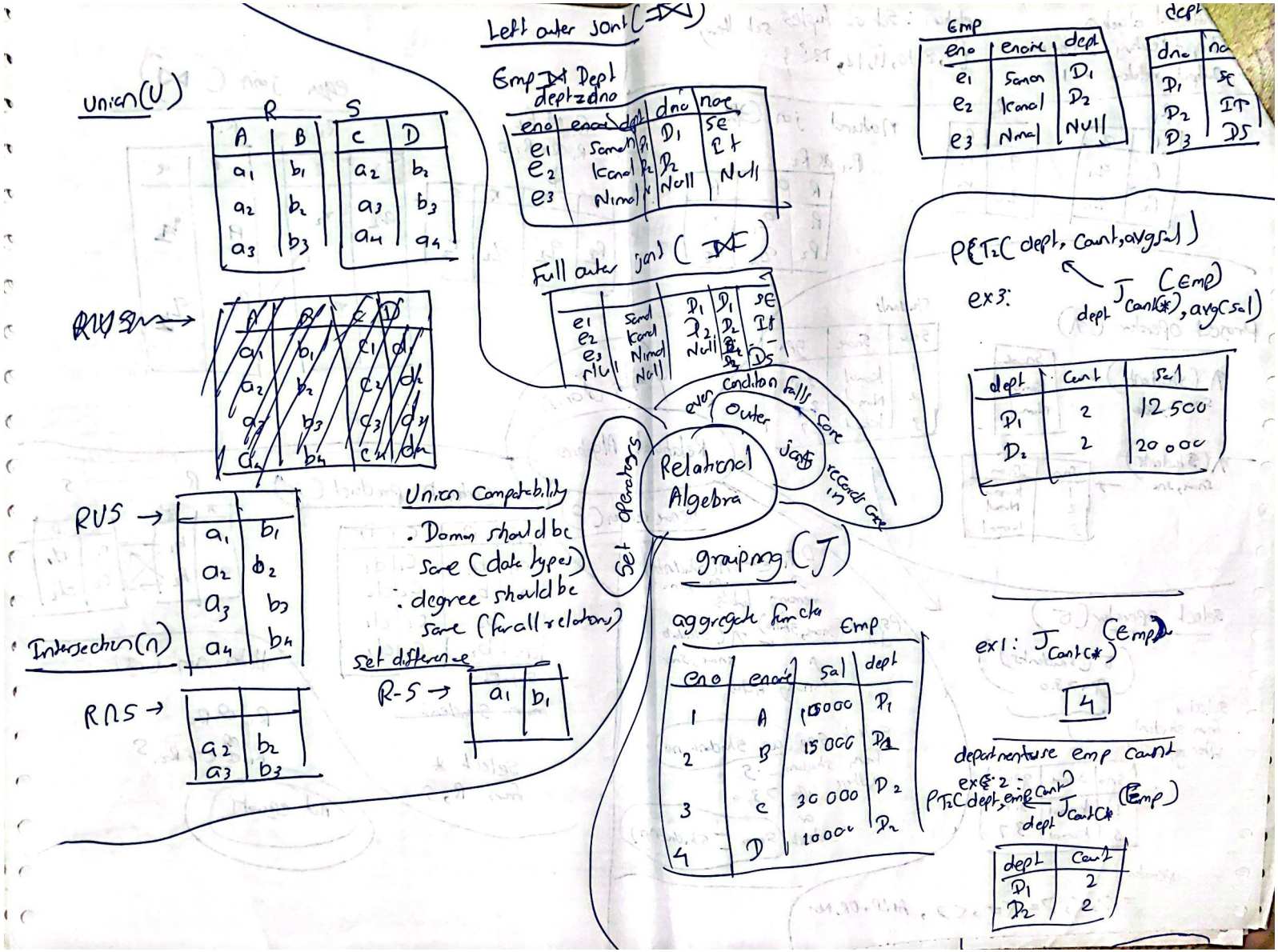
Extensible hashing



Inserting 20







Query Processing

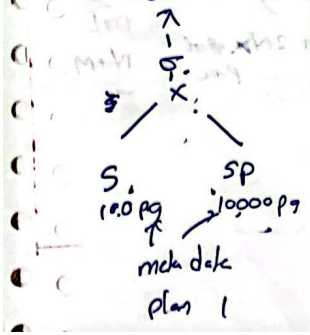
1. Parsing and translation
2. Optimization
3. Evaluation.

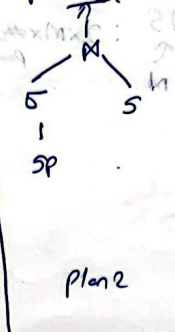
Select S.name
from S, SP
where S.sno = SP.sno AND
SP.pno = 'P2'

$\Pi_{sname}\left(\sigma_{S.sno = SP.sno \text{ AND } SP.pno = 'P_2'}(S \times SP)\right)$

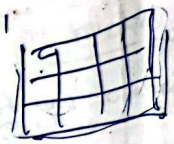
$\Pi_{sname}\left(S \bowtie \sigma_{pno='P_2'}(SP)\right)$

To find the best execution plan, we check cost: IO operations.

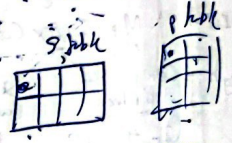




Plan 1



for each S record all tuple in page matched and sent back to HD.

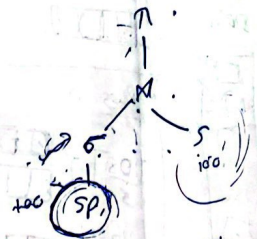


# of reads $= 100 \times 10000$

# of writes back to HD, since we cannot keep all data in memory. 100,000

# of reads to check the condition = 100,000

Plan 2



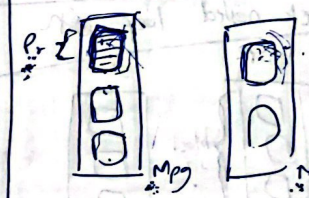
# reads to check condition in SP table = 10,000
# of reads to check join from P table = 100
Total = 10,000 + 100 = 10,100.

Algorithms to join tables

- Simple nested loop join
- Page oriented Nested loop join
- Block nested loop join
- Index nested loop join
- Sort merge join

Simple nested loop join

R ⋈ S



$Cost_{SNLP} = M + (MP_r \times N)$

since we take a page of outer table at a time — for each rec, inner table is scanned

- for each tuple in outer table
the inner table is scanned.

(8)

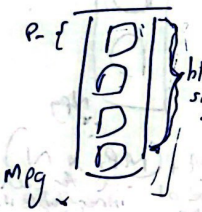
Page oriented nested loop join

- Unlike simple nested loop join in here, 1 page of tuples will be compared to the inner tbl to join Out one go

Selected

Cost $_{PNLJ}$ = M + M x N.

Block nested loop join

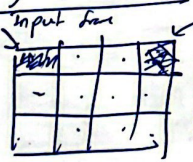


Cost $_{BNLJ}$ = M + N x (#of blocks in R)

= M + N $\left(\left\lceil \frac{M}{b-2} \right\rceil\right)$

$\lceil 10.2 \rceil = 11$

Finding # of blocks



Index nested loop join

- Create index on inner table B_+ or hash.

Cost $_{INLJ}$ = M + (Index Cost)(M x P_L)

Heigh = Cost B_+ tree (2-4) IO

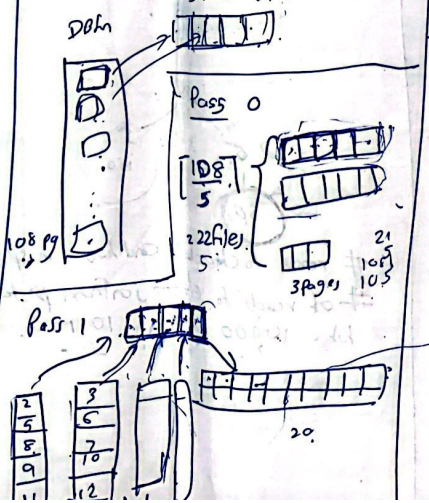
Hash index 1.2

Sorting algorithms

External merge sort

- group by
- order by
- distinct

DBMS Sort records



Pass 0

$\left\lceil \frac{108}{5} \right\rceil$ = 22 files — 5 pages each file, 21 x 5 + 3

Pass 1

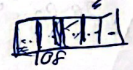
$\left\lceil \frac{22}{4} \right\rceil$ = 6 files — 20 pages each file, 20 x 5 + 8

Pass 2

$\left\lceil \frac{6}{4} \right\rceil$ = 2 files — 80 pages each file

Pass 3

$\left\lceil \frac{2}{3} \right\rceil$ = 1



80 x 1 + 28

Cost : no of reads and writes.

: 2 x # of pass x # of pages

: 2 x 4 x 108.

Sort merge join algorithm

Cost $_{SMJ}$: Sorting Cost + Merge Cost

: Sorting Cost (R) + Sorting Cost (S) + Merge Cost

BNLS : 2 x M x #of pass + 2 N x #of pass + N + M

M N

...sical DB design & DB tuning

- Index selection & creation
- Tuning the (conceptual) schema
  - alternative normalized schema
  - Denormalization
  - Vertical partitioning
- Query rewrite.

When selecting an index

- What relations to index
- What are the search fields
- Single attributes or multi attribute
- type? clustered / unclustered
  B+ or hashed
- Impact on update.

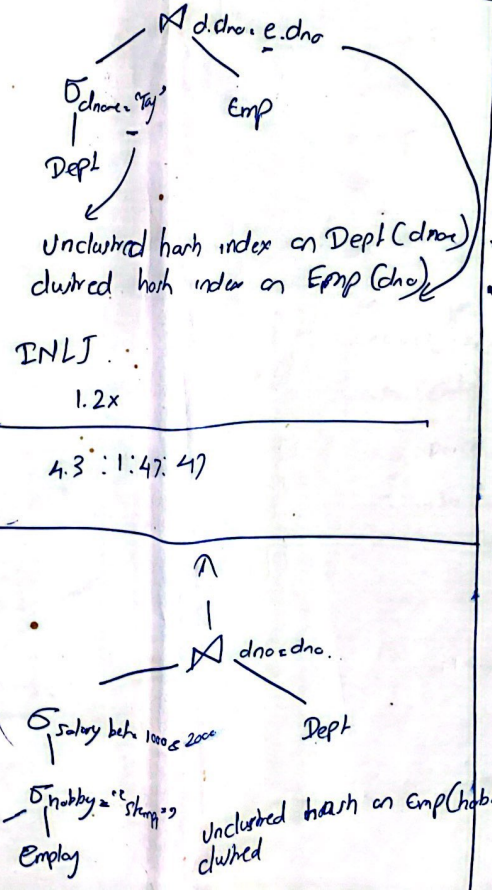
- hash indexes — equality selection
  B+ tree — range selection.

Most selective one is taken first in a multi attribute index.

---

clustered v unclustered.

```
Select e.ename, d.mgr
From Empl e, Dept d
Where d.dnoe = 'Toy' and
      e.dno = d.dno
```



---

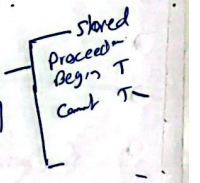
heuristic rules

- Avoid cartesian product
- Bring selection down
- left d plan

Tuning Conceptual schema

- denormalization
- Vertical partitioning.
- views

- DB programming level
- front end coding level



connection object
Set properties
Comit Comit false)
• set Commit point.

- Isolation level
- Serializable (S2PL + index locks)

- Repeated Read (S2PL)
  Phantoms possible

Transaction and Concurrency Control

Transaction: Sequence of reads & writes that belong to one unit of work.

benefits from the database side / properties

A - Atomicity
C - Consistency
I - Isolation &
D - Durability

Atomicity

all actions of the transaction happen or nothing happens.
(Executed as a single unit of work)

Consistency

Maintain integrity Correct dataset in the DB
(Compile constraints)
- key
- table
- referential
- Domain
- assertions

(consistent DB instances)

Isolation

Even though actions of several transactions are interleaved, the net effect is identical to executing all transactions in some serial order.

Durability

After transaction is committed, the data should be persistent.
(Transaction services crashes & system failures)

---

schedule actions

read write Commit abort

Why ∥ parallel transaction important

T_1	T_2
Calculating interest	Withdrawing money

Parallel

not parallel

Serial schedule: Schedule does not interleave the actions of multiple transactions

T_1	T_2

or

T_1	T_2

- no conflicts
- always, ends with correct schedules.

equivalent schedule: Same effects from the schedules despite execution order.
Different state effect of executing T_1 & T_2

serializable schedule: exact

Things are happening parallelly but equivalent to some serial schedule.

Anomalies in interleaved transactions

(1) W(A), R(A) (dirty reads i.e. r.w.)
(2) R(A), W(A) (Unrepeatable reads)
(3) W(A), W(A) Overwriting uncommitted data

Lock based Concurrency protocol

Strict 2 phase locking (Strict 2PL)

- transactions must obtain a shared lock before reading and exclusive lock before modifying
- All locks for a given transaction is released after committing (complete)
- If a transaction has an exclusive lock on an object, other transactions cannot have a shared or exclusive lock on that

How 2PL solves WR.

T_1	T_2
W(A)	
	R(A)

T_1	T_2
X(A) W(A)	
	R(A) wait

Cannot take shared lock to read. ∴ it wait.

RW solving

T_1	T_2
R(A)	
	W(A)

Shared lock already given

T_1	T_2
S(A) R(A)	
	X(A) W(A) wait

Cannot give exclusive lock since it violates 3rd rule

T_2 waits.

therefore S2PL only allows serializable schedules.

- Cascading aborts
- Unrecoverable aborts.

Cascading

A = 50

T_1	T_2
W(A) 100	
abort	R(A) - 10

How S2PL solves cascading

T_1	T_2
X(A) W(A)	
abort	R(A) wait

Unrecoverable

T_1	T_2
W(A)	
	R(A)
	W(B)
abort	Commit

Cannot recover since already commit

overhead of lock based cc

- Maintaining & handling of locks
- Handling deadlocks
- Handling phantoms.

Deadlock

- Prevention ✓
- Avoidance (deadlock detection)

- Cycle of wait.

Prevention methods

- wait & die
- wound & wait

Wait & die

T_1	T_2
S(A)	
R(A)	
	S(B)
	R(B)
X(B) wait	
	X(A) wait

- Priorities are assigned based on time

T_1 is high priority ∴ it waits

But when T_2 is asking for X(A), it is already locked and priority since it is low, it is aborted.

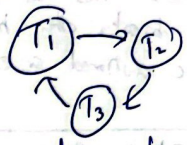
Wound & wait

* T_1	T_2
S(A)	
R(A)	
	S(B)
waiting	R(B)
X(B)	
	X(A)

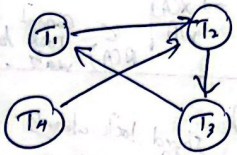
- High priority doesn't wait for low priority
- Wounds the low priority and cancels it and takes resource

Detection

- Create a wait for graph



- Circular wait.



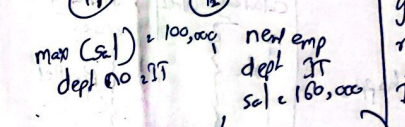
Phantom Problem

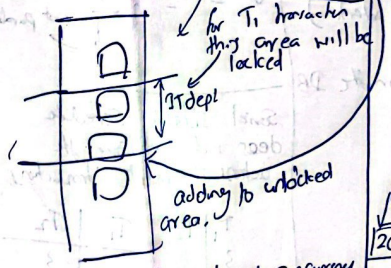
DATE : 10-06-2021
5:13:00

static database: only allows inserts and updates

dynamic : database is growing insert and delete.

Emp(eno, ename, sal, dept)





Phantom Problem: lock based concurrency protocols can only lock existing data in the database but parallely running other transactions can manipulate data into other areas conflicting with

Solution. Predicate locking.

Index locking

- Create an index on the selection criteria C dynamically in the above case
- lock the index entries along with

Index locking

Problem? It locks all nodes from root to leaf.

- Since in S2PL locks are released after completing transactions

Simple tree locking algorithm.

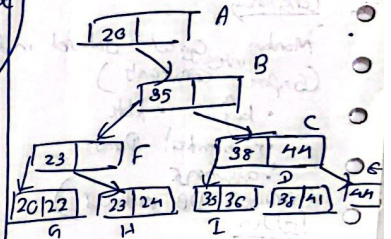
Search: Start from root & go down. go to child and release parent.

Safe

Insert: Node is not full

Safe

Delete: Node is not half empty



- Delete 38

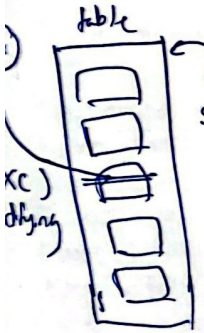
X(A)
X(B) not safe
X(C) rel A, B
X(D) rel C

Tried as

X(B)
X(B) rel (A)
X(C) not safe
X(E) - rel C

Multiple granularity lock scheme

locks more efficient



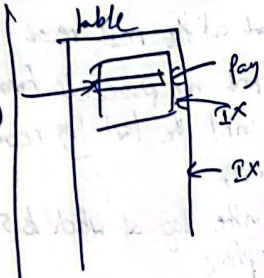
Solution: multiple granularity levels

db
↓
table
↓
Page
↓
record.

S, X, intension shared (IS)
• intension exclusive (IX)

• This gives the intension that someone is modifying inside.

assume that I want to have a shared lock on page lvl.
Then I will have to have intension shared locks on all the above levels.



How does this solve?
Before releasing a lock, the lock manager wants to search, so many objects.

Crash recovery

The way of guaranteeing atomicity and durability.

Why it is needed.?

• before Commit to the commiting part might write to disk
• Commited transactions might not be written to disk.

Why dbms writing before commiting part?

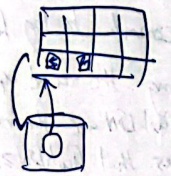
Why not writing to disk after commited?

↓

When load is high, server might say done without doing

• Writes middle of the disk in the transaction

Steal



• Can we write pages to disk of uncommitted transaction
• Data goes to disk from uncommited transactions.

(Commited data already gone to disk
• undo

Force

• at commiting part things are written to disk.
• Redo

	No steal	steal
Force	Trivial	
No force		Desired

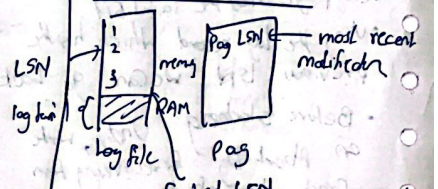
Logging

• To decide what to undo and redo.
• log Carries
XID, pageID, offset, length, old data
new data.
• Sequential writes.

Write ahead logging protocol

• Must force log records for an update before corresponding data page goes to disk (Guarantees atomicity)
• Must write all records for transaction before Commit (Guarantees durability)

Log Sequence Number



• Before a page is written: Page LSN ≤ flushedLSN

Log records

has: • XID
• type

Possible log record types
• update • Abort • undo
• Commit • End

transaction table

TID	status	last LSN

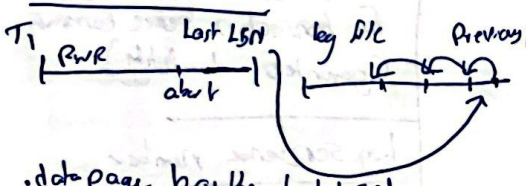
Dirty page table — oldest modification to the given data pr

Page #	record LSN

Checkpoints

- Creates checkpoints to reduce the time taken to recover in the event of a system crash.

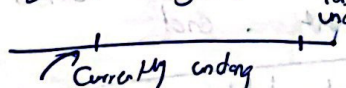
Simple transaction Abort



- data page has the last LSN
- Using the log record which has the previous LSN we can go back.
- Before starting Undo, Write an Abort log for recovering from crash during aborting.

To perform undo, it must have a lock on data.

- CLR entry is again written back to the log just incase a failure happens during the aborting process.



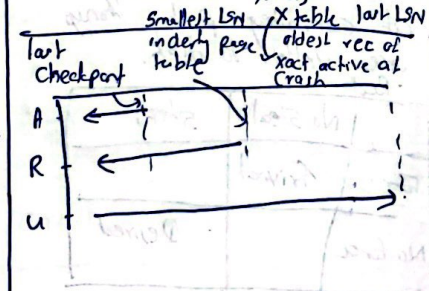
- at the end of undo "end" is written.

Commit

- Write commit to log file after logging the record upto Xact's last LSN are flushed.
- guarantees that flushLSN ≥ lastLSN

~~ARIES~~ ARIES algorithm

- there are 3 phases in the algorithm
  - Analysis to figure out what Xact's have failed since the last commited checkpoint.
  - Redo all actions
  - Undo failed xacts



= 245

Analysis phase

- reconstruct the state at checkpoint via end_checkpoint record.
- Scan log forward from checkpoint.
  - End: remove these Xacts from the Xact table.

Analysis start at the most recent begin_checkpoint and proceeds forward in the log until the last log record. It determines;

- point in the log at which to start the redo phase
- the dirty pages in the buffer at the time of crash
- Transactions that were active at the time of crash which needs to be undone

Redo phase: Redoes all changes to any page that might have been dirty at the time of crash.

Undo: Undoes all changes of all xacts that were active at the time of crash.

Hadoop

drawbacks of RDBMS
- difficult to scal VRT big data
- Difficult to maintain & configure.
- Diversifications available system complicates selection

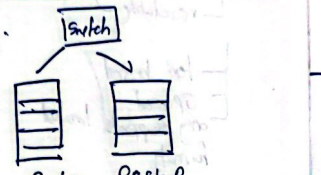
NoSQL advantages
- Elastic Scaling
- Less administration
- Better economics
- Flexible data models

NoSQL disadvantages
- Hard to access without explicit knowledge of data models.
- limited access control
- Limited indexing

Distributed file systems.
- designed for Clustred computing
- Analysis runs on files.
- Scalable analysis.
- Analysis is parallelized

Hadoop Why?
- Redundant, fault tolerance data Storage with HDFS.
- Parallel processing framework map reduce
- Job Coordination - Yarn.

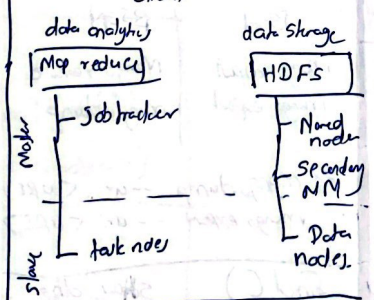
- horizontal Scaling
- Processing in nodes
- Scale on demand.



intra rack speed > inter rack speed

Master nodes & Slave nodes
- higher level
  - Named nodes
  - Secondary NN
  - Job tracker
- data node
- task nodes

client



Name node
- arbitrator of repository for all HDFS meta data
- Manages file system namespace
- mapping of blocks to data node
- regulates file access by client.

Data nodes
- Manages Storage
- Store block of files
- responsible for serving R/W req of clients.
- Performs block creation & duplication from NN.

HDFS characteristics
- Write once read only model.
- Content in a file cannot be modified but appending new data

HDFS operations
- Create files
- Delete files
- Rename files
- Append content to the end
- Modify attributes.

Partitioning
- Files are divided into blocks and duplicated in atleast 3 nodes

HDFS write
- client consults NN.
- Named node replies with location
- client writes to data node
  - DN replicates
  - cycle repeat of next block.
- acknowledgement.
- client send a request to NN to close file

HDFS read
- determine location of data from NN
- Contacts DN to retrieve data.

HDFS robustness
- Data disk failure
- Heartbeats & replication
- Cluster rebalancing
  - Balancer.
  - moves data to another DN if freespace falls under threshold
- Data integrity
  - checksum for each block file
- Meta data failure
  - Replication of NN.
- Snapshot
  - Store a copy of working system

FS Image - complete file system namespace including mapping of blocks to files and FS.

Edit log - Store modification of FS meta data.

Map reducer

YARN - Yet another resource negotiator.
- Allocates resources to applications effectively.
- Allows to run multiple applications simultaneously.

Job tracker
- Resource manager: resource management
- Application master: Job scheduling, monitoring

- Resource manager runs on master nodes.
- Node manager runs on slave nodes.

Hadoop implementation	Hadoop YARN
Job tracker (Master) • resource management • Job lifecycle management. • fault tolerance	Resource manager (Master) • resource management • scheduling
Task tracker (Per node) • launch task • report status to	Application manager (Per app) • Job lifecycle management
	Node manager. (Per node) • launch containers

YARN Container.

- Container: Collection of resources (RAM, CPU, disks)
- where the YARN app runs.
- An app might run on one or many containers.
- Supervised by the NM & scheduled by RM.

General workflow.

- Submits application to RM
- RM asks NM to create an application master which registers with the RM.
- AM determines how many resources are needed.
- RM accepts requests and queue it.

Map reduce

- manages large scale computations, tolerant to hw failures.
- Manages automatic parallelization & distribution.
- Coordination of tasks implemented in map() and reduce() & copes with unexpected failure.

MongoDB

- JSON
  - user friendly
  - readable
  - text based
  - spaced
  - only supports limited formats
- BSON
  - speed
  - space
  - Flexibility

Exporting data

JSON	BSON
mongo import	Mongo restore.
mongo export	mongodump

ex mongodump --uri <URI>
mongo export --uri <URI>

find()

Show dbs.
Show Collections.

ex db.zips.find({"state": "NY"})

Insert()

BSON

db.zip.insert({ ... })

inserting multiple

db.inspection.insert([{}, {}]);
- insert([{}], {ordered: false})

Update()

updateOne(), updateMany()

db.zips.updateMany({"city": "Hudson"}, {"$inc": {"pop": 10}})

$inc - increment
$set - set
$push - add elements to an array feild.

{$push: [{}, {}]}

Delete()

deleteOne() deleteMany()
Collection.drop

Comparison operators

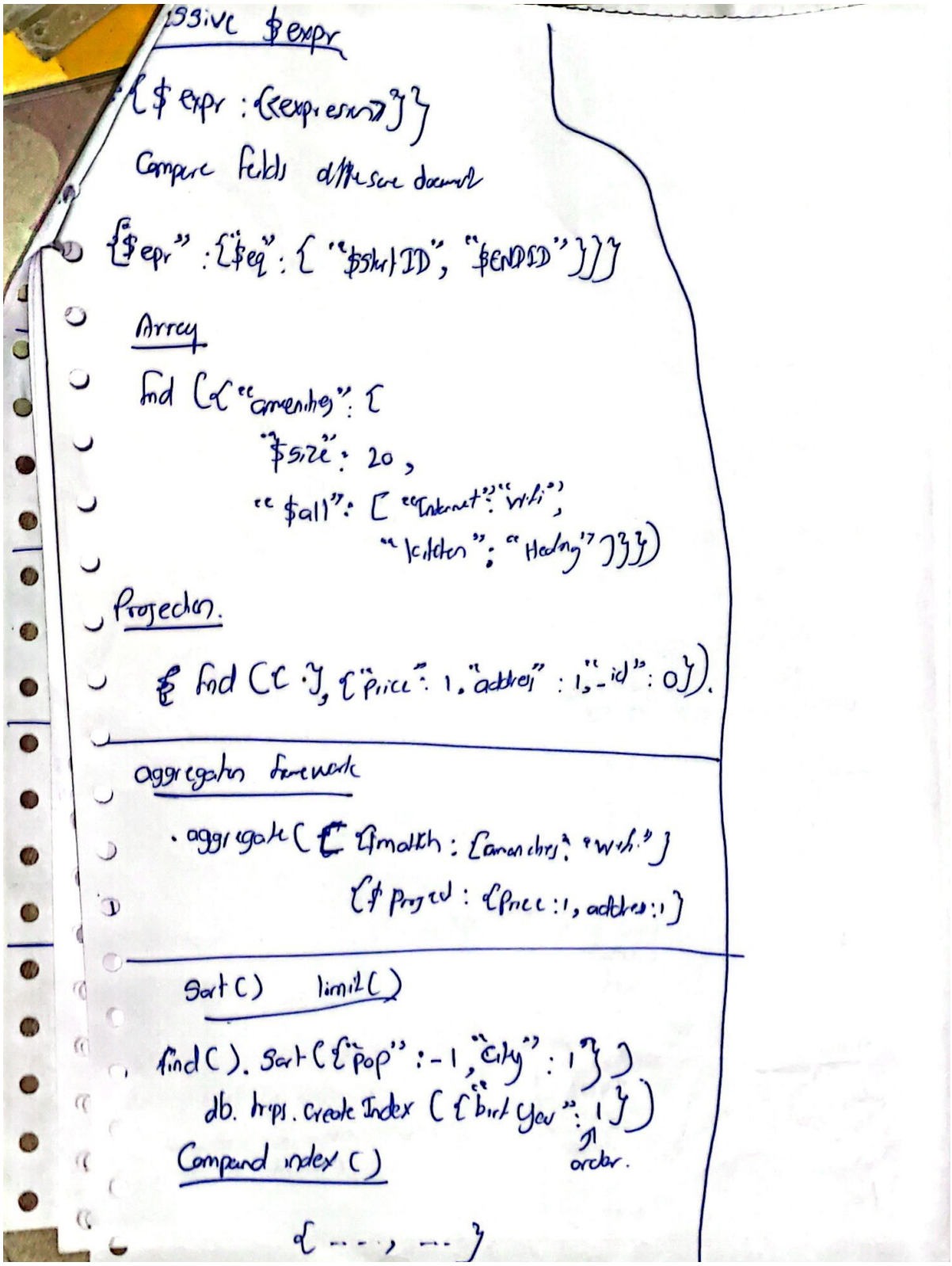
$eq, $ne, $gt, $lt, $gte

{"feild": {"operator": <value>}}

inside find() or delete()...

Logic $and $or $nor

{<operator>: [{statement}, {}]}



ssive $expr

{$expr : {<expression>}}

Compare fields d/fferent documents

{"$expr" : {"$eq" : ["$startID", "$endID"]}}

Array

find ({"amenities": {
"$size": 20,
"$all": ["Internet", "Wifi",
"Kitchen": "Heating"}}})

Projection.

find ({}, {"price": 1, "address": 1, "_id": 0}).

aggregation framework

.aggregate ([{$match: {amenities: "Wifi"}}
{$project: {price: 1, address: 1}}

sort() limit()

find().sort({"pop": -1, "city": 1})
db.trips.createIndex ({"birth year": 1})
order.

Compound index()

{..., ...}