ADB Lecture 3

• Introduction to Query Processing

→ Consider a Select query with Search Key ⟨age, sal⟩

* The way we Process the query will depend on the underlying File Organization.

- **Heap File**
  → Can be an unclustered index (B+ tree or hash)
- **Sorted File**
  → Can be a clustered index (B+ Tree)

* The query may involve one of many operations

Scan	Equality Search	Range Selection	Insert	Delete
• Fetch all records	• Field = ... • Fetch the block then locate record	• Field >, ... (or $\geq, <, \leq$)	• Fetch the Page • do the operation the write it	

→ The Key to good Physical design is accurately describing the expected workload
→ workload is a mix of queries and updates
(search query) (action query)

→ For each query in the workload, need to identify

→ For each update

• Choice of Indexes

① → Which relations should have indexes? • Freq. of Change
② → What fields should be the search key?
③ → How many indexes for the relation?
④ → What's the type of each? • Clustered? • At most 1
• Hash / Tree?

• A Possible approach
  • Given how DBMS evaluates queries and creates query evaluation plans.
    1. Consider most important queries
    2. Consider the best evaluation plans for them given the current indexes
    3. If a better plan is possible with an additional index, then create it.
       → Should as well consider the impact on updates before creating an index.

• Recall Indexes → Faster Queries
→ Slower updates (must propagate)
→ Disk space
} Trade-offs

⇒ Index Selection Guidelines

1. For index keys, consider attributes in WHERE
   • Hash Index ← Exact Match Condition
   • Tree Index ← Range Queries
     B+ • Many changes
     ISAM • Otherwise
   * In general, clustered index is helpful for range queries and exact match if there are duplicates

2. Use multi-attribute search keys if WHERE clause contains several conditions (composite)

- In case range queries are expected, order the attributes based on them.

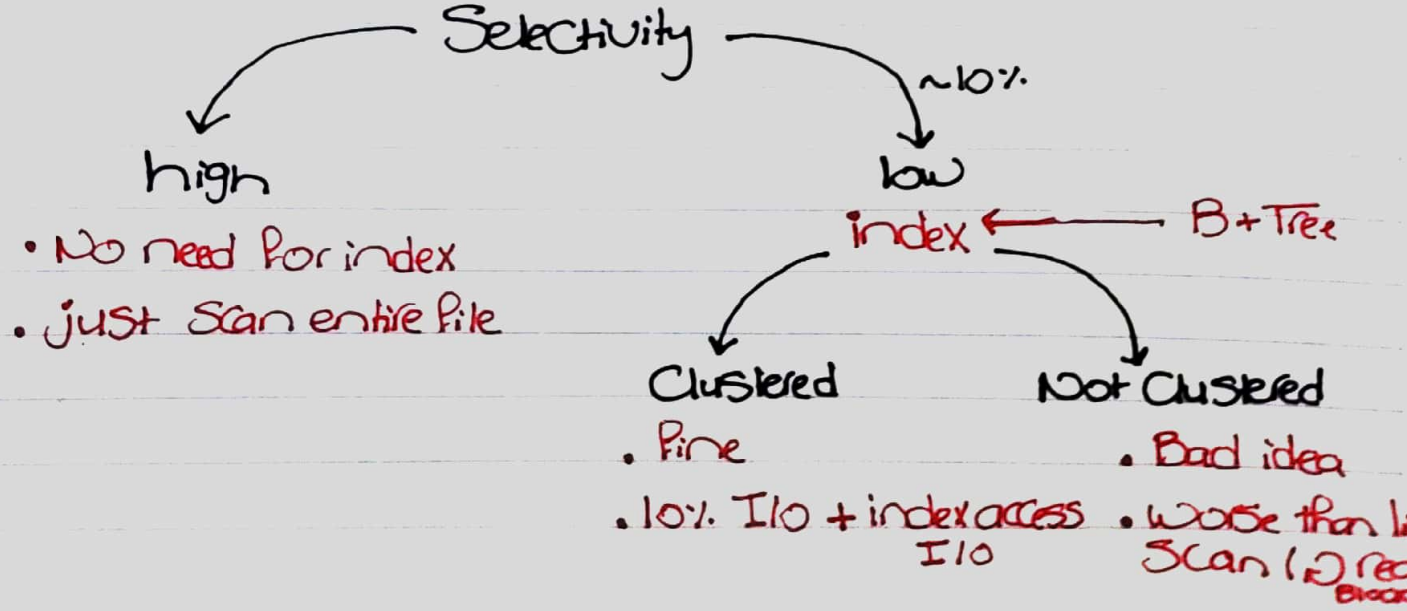
3. Choose indexes that benefit as many queries

- e.g., we know we can have at most one clustered index per relation
  → Choose it based on frequent queries that would benefit the most from clustering.

- Should carefully decide when it comes to clustered index (expensive updates; although cheaper than sorted files)

- Examples

① SELECT E.dno
FROM EMP E
WHERE E.age>40 #Range Query



② SELECT E.dno, Count(*)
FROM EMP E
WHERE E.age > 10
Group by E.dno

• Suppose an index is to be used. (e.g., also equality)

Selectivity

high (WHERE is not highly selective)
→ Clustering index on E.age not a good idea. (too much E.dno Sorting afterwards)

low (WHERE is highly selectivity)
⇒ is okay in this case.

→ Clustering on E.dno is better
- group by is easy due to sort
- age > 10 is just a Filter

⇒ no need.

③
SELECT E.dno
From EMP E
Where E.hobby = 'stamps'

- A secondary index on E.hobby can help (duplicates are possible)

④
Select E.dno
From EmP E
Where E.eid = 50

- eid is unique
- By Guideline 1, hash is better this time.

(5)

Select *
From EMPE
Where Sal > 10

Consider a Composite index

<age, Sal> <age> <Sal, age> <Sal>

- Not helpful (Sorted by age then by Sal)
- No
- Both work
- Sal must be Prefix

Visually

- all occurences in file sorted by age then Sal.

- Consider the following modifications

age = 30 and Sal = 4000	20 < age < 30 and 300 < Sal < 400	age = 30 and 300 < Sal < 400
→ <age, Sal>	→ <age, Sal> or <Sal, age>	→ <age, Sal>
• better than <age> or <Sal> (Can binary Search twice)	, intuitive as range queries benefit much from sorting	• (1) definitly holds
, better than <Sal, age> if age = ... is more Selective. (1)		

* Index Only Plans

→ Some queries can be answered just using the Index (no tuples retrieved from relation).

→ in this case, Clustering is not important (Connection to relation)

Examples

```
Select E.dno, Count(*)
From Emp E
Group by E.dno
```

- Key is E.dno
  - → e.g., count pointers out of each E.dno value
- Secondary index (non-key)

```
Select E.dno, MIN(E.Sal)
From Emp E
Group by E.dno
```

- Key is <E.dno, E.Sal>
  - → For each group of dnos, Salary sorted by default.

```
Select avg(E.Sal)
From Emp E
Where E.age = 25 and
E.Sal between 3000 and 5000
```

- Key is <E.age, E.Sal>
  - → Find <25, x> then keep reading. (3000 or more if not.)

* External Sort

- Needed to create clustered indexes (sort w.r.t key)
  - → 1st step in bulk loading B+ tree (instead of repeated insertions, can sort first)
- Eliminating duplicates
- Needed in sort-merge join algorithm

Problem to solve
- → Sort 1GB of data with 1MB of RAM while minimizing disk access.

- Two-way sort
  - → uses 3 buffers (2 input, one output) • Site = block
  - → Suppose file has 2^k pages

Only one input buffer used.

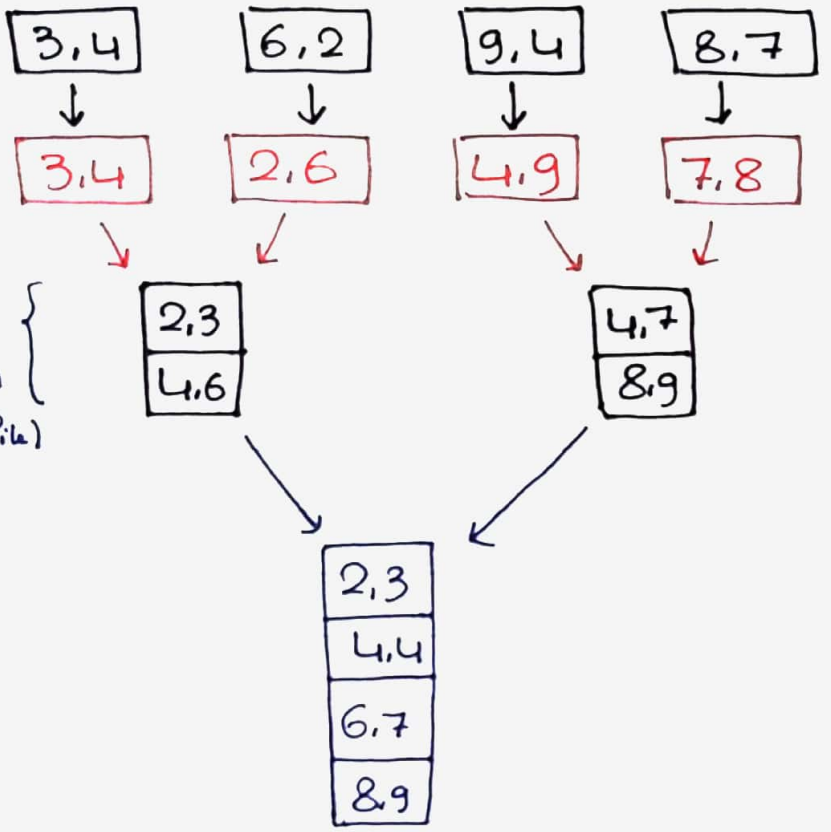
- Pass 0
- read all 2^k pages & sort each

- Pass 1
- read 2 pages and sort-merge then into output buffer
- Clearly need to do that 2^{k-1} times

- Keep going (Pass 2, 3, 4, ..., K) = Sorted File

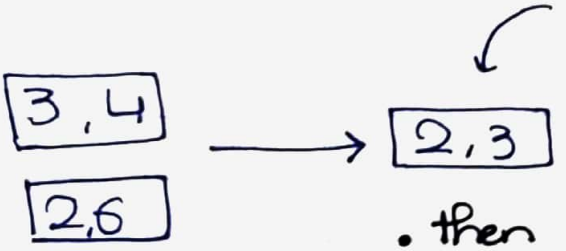
#blockSize=2

e.g.



* Consider if [8,7] was empty (not a Problem).

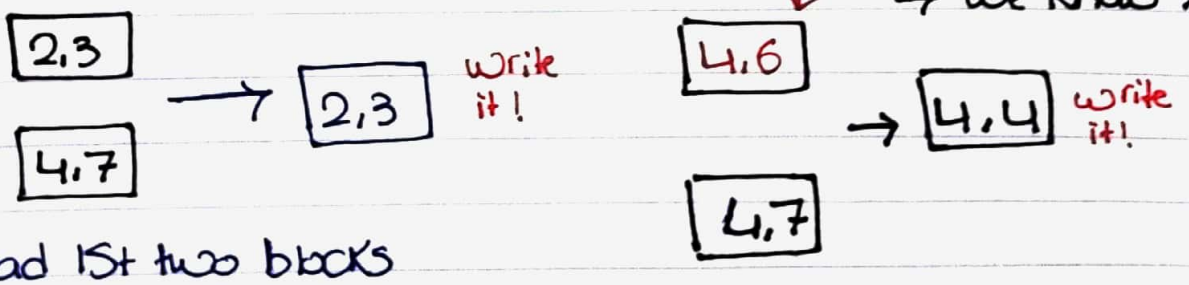
→ Zoom in [3,4] [2,6] loaded into the two input buffers



- then 4,6 remains, write it next.

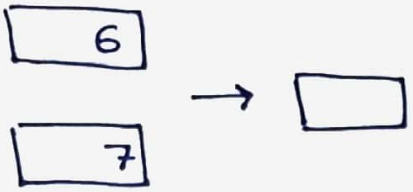
two reads and two writes

→ Zoom in ① [2,3] [4,6] and ② [4,7] [8,9]



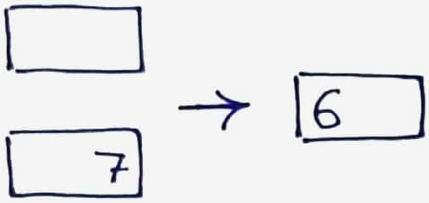
- read 1st two blocks of each

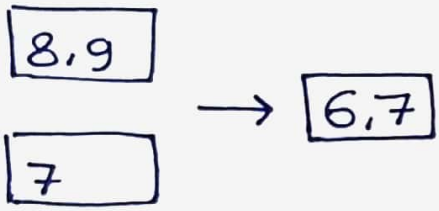
Now



- Since $x \geq 6$ and $x \geq 7$ for ①, ② resp., Put 6 in output

- load 8,9 (final block)





write it ... then write 8,9

→ Consider another subproblem (to further clarify)

- next block in each of the following runs

① 7,11 ② 23,24

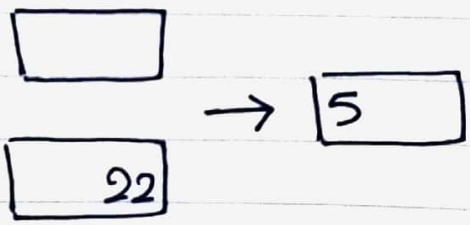
- buffers state was

From ① 1,5

From ② 2,22

→ 1,2 →

- now it is



* which run should we load from the next block?

① $x \geq 5$ ② $x \geq 22$

Choose ① (now $x \geq 11$ for the next decision) ← ①

- The no. of Passes is clearly $\lceil \log_2 N \rceil + 1$ $(2^K \to K)$ Pass
  → in each, we read and write N blocks

- hence, # IOs $= 2N(\lceil \log_2 N \rceil + 1)$

⟹ Suppose we have B buffer Pages in main memory (B-1 as input and 1 as output)

Pass 0

- not the same as 2-way
  → load the N blocks B by B and Sort each internally //in Place
  - result is hence $\lceil N/B \rceil$ Sorted runs.

Pass 1, 2, ...

- like 2-way
- Need the output Page (hence B-1)
  → merge each B-1 runs (Sort-merge)

⟹ Initially have $\lceil N/B \rceil$ runs and tree branching factor is B-1, thus

$$\#IO = 2N\left(1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil\right)$$

- 1: Pass 0
- $\lceil \log_{B-1} \lceil N/B \rceil \rceil$: Pass 1, 2, ...
- 2N: read & write N blocks each Pass.

- Example) Consider B=5 and N=108
  ↳ 4-way merge

Pass 0 ⟶ Pass 1 ⟶ Pass 2 ⟶ Pass 3

- Produce $\lceil 108/5 \rceil = 22$ Sorted runs
- $\lceil 22/4 \rceil = 6$ Sorted runs
- $\lceil 6/4 \rceil = 2$ Sorted runs
- Converge (2-way merge)

- Works Surprisingly Well

→ $N = 10^9$ and $B = 257$ means we need $1 + \lceil \log_{256} 7 \rceil = 4$ Passes (excluding Pass 0)

→ although high B helps ↓ #IO a lot, it also increases CPU time

- Next Slide (algorithm modifications) was Probably skipped, has a lengthy discussion in the book and is Per se Confusing. (B-b input buffers, b output..)

→ Consider when an input block becomes empty
  - need to suspend execution until its loaded again (then find min. over all blocks...)
  - Use an additional B shadow blocks that Prefetch from disk (B-1 and output)

  → once input buffer becomes empty switch to a full shadow block and Prefetch in the mean time.

  * CPU hence, is now never idle

#Double Buffering

- Query Evaluation

→ Needs to be optimized ↗ what are the possible query evaluation Plans?

→ Two main issues

↳ How is the Cost of each estimated?

* Ideally ← want best Plan!
* Practically ← avoid worst Plans!

- Use Search algo. to find Plan of best Cost.

- In the following, we'll ignore the final write by the operation (as if its output will be used by a parent operation in the query tree; stays in memory instead of being written into disk)

* Algorithms for Joins

```
Select *
From Reserve R, Sailors S
Where R.sid = S.sid
```

} Need to Compute $R \bowtie_C S$

- Since RxS is large, its inefficient to do that followed by selection to implement the join.
- Want to optimize # IOs (excluding final write)
- Algorithms to consider
  - → Nested Loops joins
    - Simple and block versions
  - → Sort Merge join
  - → Hash join

1. Simple Nested Loop Join

```
For each tuple r in R
    For each tuple s where r==s
        add <r,s> to result
```

- let M, P_R be the no. of blocks for R and the no. of tuples in each, respectively.
- N, P_S for S.

- then $\#IOs = M + (P_r \cdot M)N$
  - M: R is scanned once (outer loop)
  - $(P_r \cdot M)$: For each tuple in R
  - N: Scan S (look for a match)

→ Simple refinement (Page-at-time)

```
For each Page m in R                    (Page block)
  For each Page n in S
    add any rEm, sEn, r=s to result }memory
```

$\#IOs = M + M.N$
- M: R is scanned once (outer loop)
- N: S is scanned for each block in R

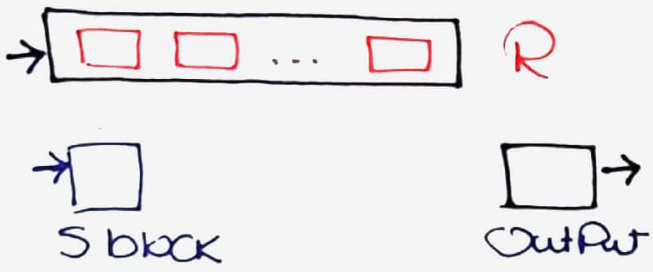
- Clearly, making the outer loop the smaller relation results in less I/Os
- This uses 3 buffers (one to read R block another to read S block and third to store the output).

2. Block Nested loop Join

→ Suppose we have enough buffer pages
- to load entire smaller relation R
- Still have 2 extra buffer pages

- In this case, once we load in the smaller relation R (M IOs) we only need to read S block by block and write the output (2 extra buffers)

$$\# IO = M + N$$



- Same for loop but R is all in memory

- What if the smaller relation R doesn't fit in memory?
  - → Have B blocks
  - → load R Part by Part in B-2 blocks
  - → 2 Blocks left for S and Output
  - Part of
  - each time R is loaded, find all matching pairs for each block in S.

Clearly, R will be loaded over $\lceil \frac{M}{B-2} \rceil$ chunks

$$\# IO = M + \lceil \frac{M}{B-2} \rceil . N$$

The general nested loop join for blocks is here

For each chunk of B-2 Pages of R
  For each Page of S
    - add <r,s> to result for all matches of the chunk and S block

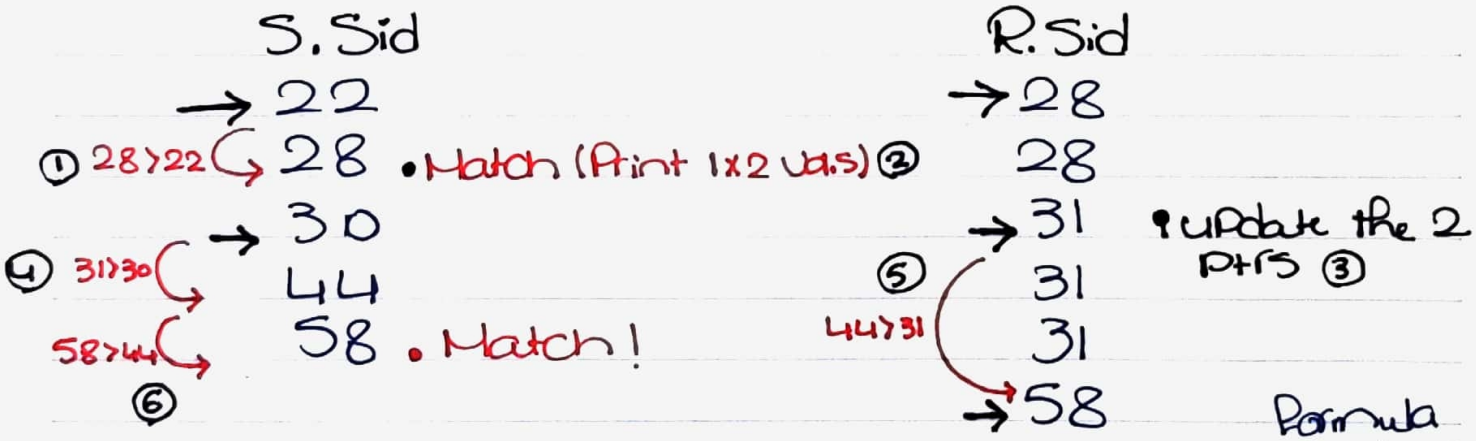
3. Sort-merge Join

→ Sort each of R and S on the join Col. (R.Sid and S.sid respectively) # external sort

racing {
- → Keep scanning R until you find a tuple ≥ the current tuple in S
- → Switch to S and do the same

if S>r / if r>S
}

- Whenever r=S, output the qualifying tuples (tuples having the same value are next to one another)
  → Start racing again until r=S one more or both r and S are at the end of table.



- Cost:

Formula we derived earlier (2NT...)

$$\#IOs = O(M \log M) + O(N \log N) + (M+N)$$

Sort R | Sort S | Traversal

- An extreme corner case would be that all of Sid in both relations have the same value, in this case, if the buffer pool is limited we may need to read all S for each block of R → MN instead of M+N
- If at least R or S has distinct values then M+N is guaranteed.

- Sort-Merge or block Nested (M=500, N=1000)

B =	35	100	300
#IOs (block n.)	16500	6000	2500
#IOs (sort m.)	7500	7500	7500

- More buffers make block nested better
- results are by the Previously shown formulae

3. Hash Joins

Partition Phase:
- Partition each of R, S into K Partitions on the disk using the same hash function h_1. (each Partion can span multiple blocks)

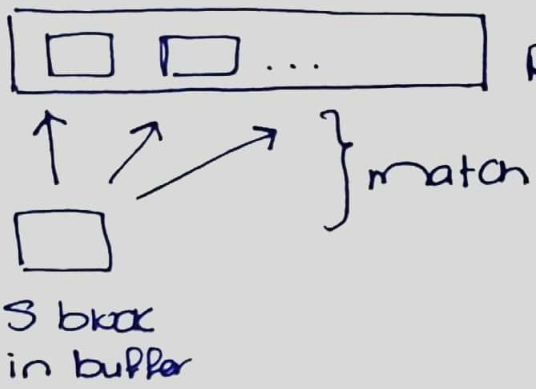
Probing Phase:
→ Assume we can hold an entire Partition of R in memory (have enough buffers)
- then load R, Partion by Partion
→ for each load the corresponding Partion of S (block by block is okay here)
- Check for matches!

* if $x=y$ then $h(x)=h(y)$ so once a Partion of R or a block of S is loaded, we never have to do that again in the future.

$\#IOs = 3(M+N)$

- $2(M+N)$ in Partition Phase
- $M+N$ in Probing Phase

→ To reduce matching effort by CPU
→ We can use an in-memory hash table h_2
- hash entire R Partition and S block from corresponding Partition by h_2
- Use it to match (much easier now)



normal case
- Better:
- hash R Partition into mini-Partitions
- hash S using same hash
- Compare with corresponding mini-Partition

* Compared to the Prev. 2 algos

$3(1000+500) = 4500$ IOs

Hash Join	vs.	Sort-Merge
$3(M+N)$ (also memory assumption)		$3(M+N)$ if $B > \sqrt{N}$ (has Proof)
. highly Parallelizable (hashing)		. less sensitive to data skew & gives sorted result.

* We implicitly assumed in hash join that each relation was ideally uniformly hashed (no data skew)
So its just as if we resorted the file (no extra space)

* Algorithms For Selection

Select *
From R
where R.name=...

- No index, unsorted
  → Scan entire relation
  → expensive/inefficient if low selectivity
- No index, sorted
  → binay search to find 1st tuple then read all matches
- B+ Tree or hash index
  → Consider cost of accessing data entries (leaves of B+) and matching
- Complex Conditions (day<... and bid=... and...)
  → use one index then filter (e.g. on one of the keys)
  → use two indexes (on day and bid), intersect then apply 3rd condition (filter)

* Algorithms For Projection

→ Removing fields is easy
→ Removing duplicates (DISTINCT specified) isn't

- Sort-based
  → Sort to eliminate duplicates
  → remove unwanted attributes before sort
  - more standard than hash ← memory assump.
- hash-based
  → hash-join (relation with itself to find duplicates)
- index-only scan
  - attributes ⊆ index attr. ✓

* Algorithms for Set operations

- Intersection, Cross Products are a special case of Joins
- For union, Except: Sort and hash options
  Very similar to Joins and Projection.

* Algorithms for Aggregate operations

- Sum., avg., min (Similar to Prev. approaches)

- Grouping
  → Sort or hash on group by attribute (or both!)
  → index only Plan can be possible

- No Grouping
  → generally, need to scan it all in this case
  → if index has all attrs in select/WHERE then index only Plan is okay