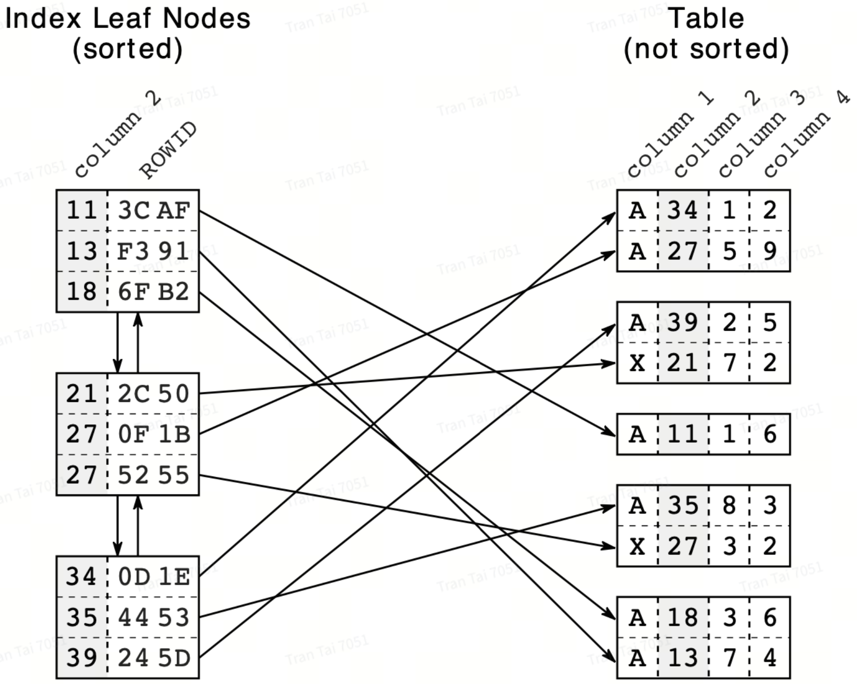
Indexes in database

1. Quick Intro

- Index is a distinct structure in db that built using CREATE INDEX statement.
- It requires **disk space** and **holds a copy of the indexed table data.** It creates a new data structure that refers to the table.
- **Challenge:** process insert, delete and update stmts immediately and keep the order without moving large amounts of data.
  - Database combines 2 DS (search tree + linkedlist) to explain database's performance characteristics.

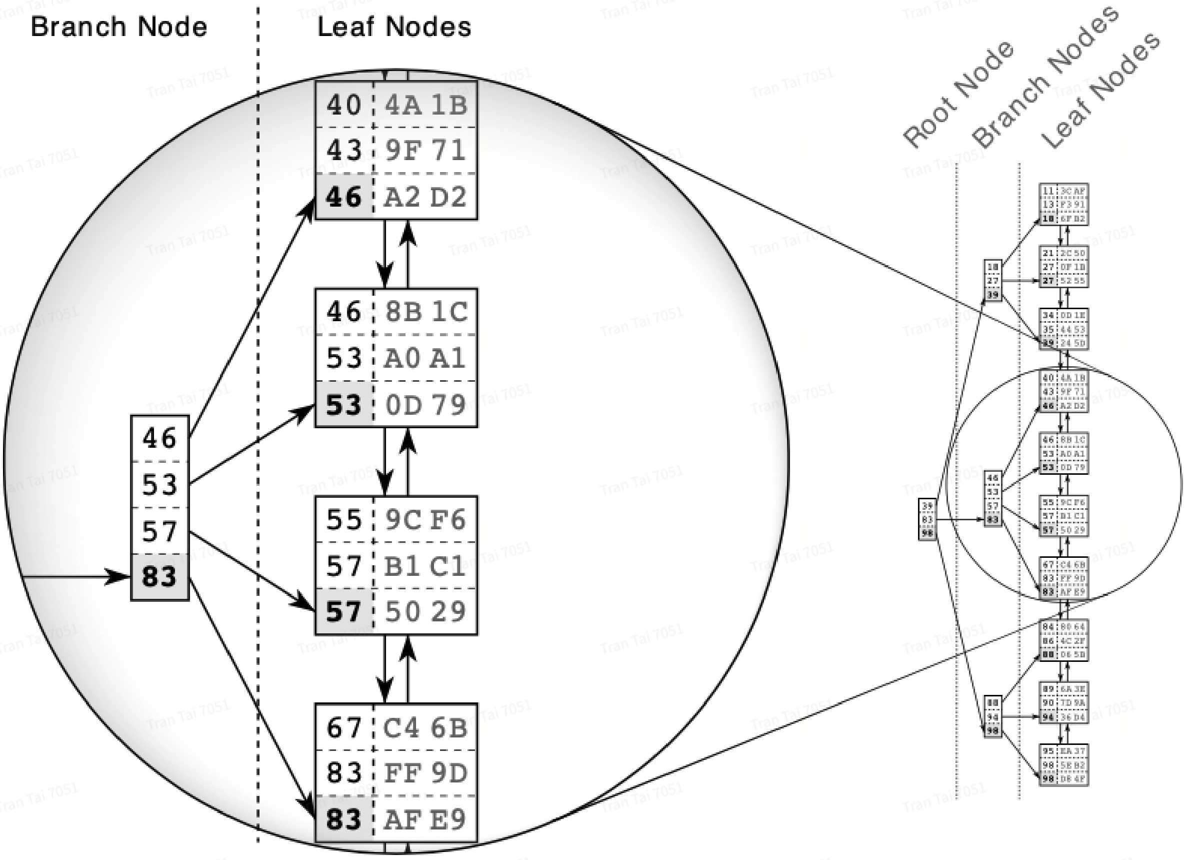
Index Leaf Nodes

- Databases used Double LinkedList to connect **index leaf nodes**.
  - Each leaf node is stored in a database block or page-smallest storage unit.
  - Database uses space in each block to extent and store as many index entries.
  - Index order is maintained on **2 different levels**: index entries within each leaf nodes, the leaf nodes among others.



Search Tree (B-Tree)

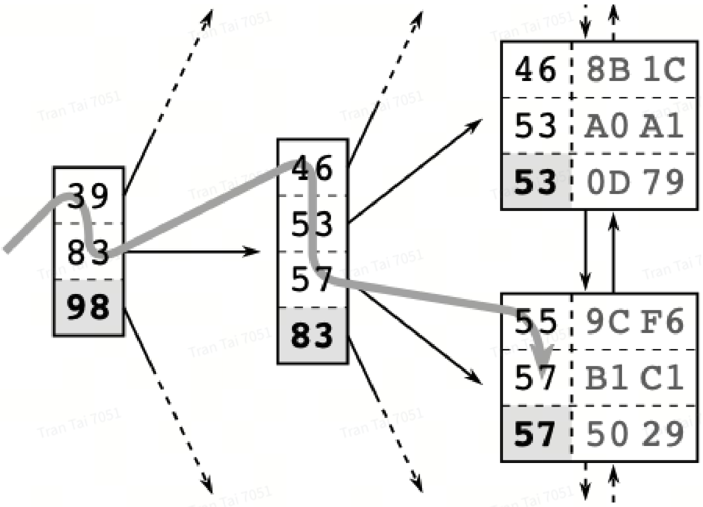
- Aka Balanced Search Tree because the depth of each leaf is similar to the others.
- Structure:
  - Last layer is the index leaf nodes.
  - Optimized structure for **searching for a leaf node quickly** .
  - From the last layer, multiple branch nodes will be built up until all keys fit into a single node (root node).



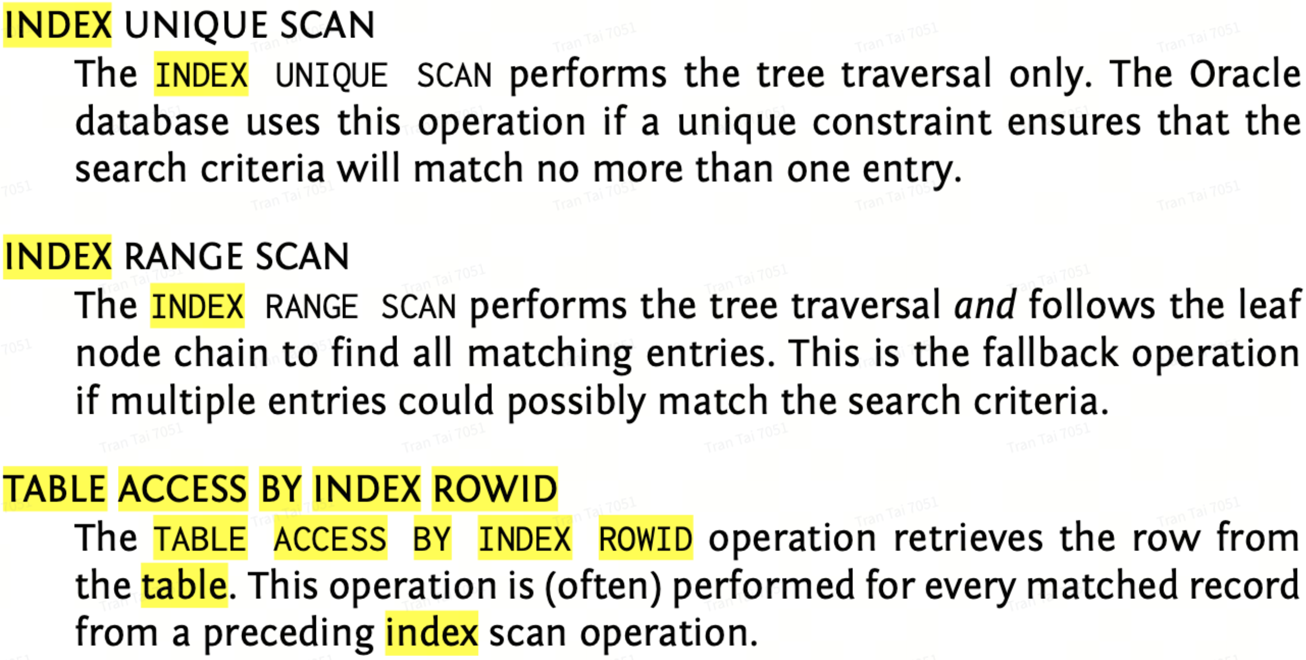
- **Tree Traversal** is 1st power of indexing because:
  - Tree balance which allows accessing all elements with same no of steps.
  - Those steps is growth with `O(log(no of leaf nodes))`
  - **Comound index?**

Slow Indexes

- Index lookup still not work as expected sometimes due to:
  - **Leaf node chain**
    - When traverse to the right leaf nodes, db must read the next left node to see if there are anymore matching entries



        - **Accessing table**
            - A single leaf node will contain many records **scattered from different table blocks**.
            - Need additional table access for each hit.
    - Index looks up consists of 3 steps:
        - Tree Traversal.
        - Leaf node chain.
        - Fetching table data.
    - The 2 later steps can cause slow index looks up.
    - Basic index lookup



INDEX UNIQUE SCAN

The INDEX UNIQUE SCAN performs the tree traversal only. The Oracle database uses this operation if a unique constraint ensures that the search criteria will match no more than one entry.

INDEX RANGE SCAN

The INDEX RANGE SCAN performs the tree traversal *and* follows the leaf node chain to find all matching entries. This is the fallback operation if multiple entries could possibly match the search criteria.

TABLE ACCESS BY INDEX ROWID

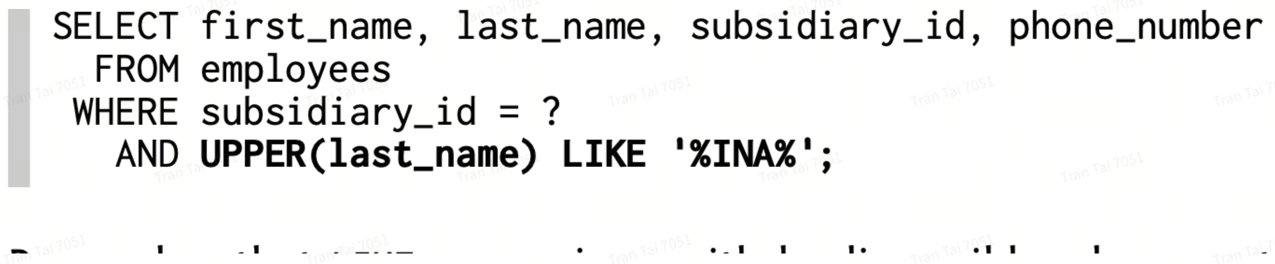
The TABLE ACCESS BY INDEX ROWID operation retrieves the row from the table. This operation is (often) performed for every matched record from a preceding index scan operation.

2. Clustering Data

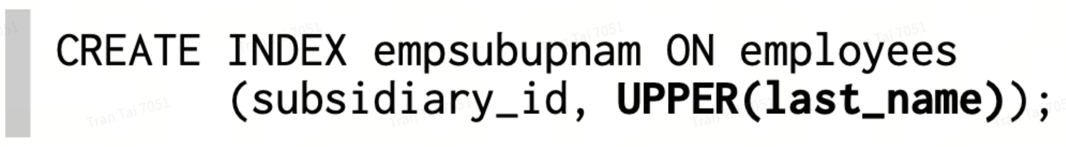
- Aka **2nd power of indexing.**
- Clustering data means storing consecutive data closely tgt so that accessing them requires less I/O operation.
- **Index Clustering Factor**: measure probability that 2 succeeding index entries refer to the same table block for optimizer to consider when calculating the value of `TABLE ACCESS BY INDEX ROWID` op
- Some technique:

Index Filter Predicates [Clustered index]

  - You can add columns to an index so that they are automatically stored in a **well defined order.**
  - The key is **not to improve range scan perf** but to group consecutively accessed data tgt.
  - **Index Clustering Factor**: the probability that two succeeding index entries refer to the same table block.
  - Consider the query:



```
SELECT first_name, last_name, subsidiary_id, phone_number
  FROM employees
 WHERE subsidiary_id = ?
   AND UPPER(last_name) LIKE '%INA%';
```



```
CREATE INDEX empsubupnam ON employees
       (subsidiary_id, UPPER(last_name));
```

  - subsidiary_id: **access predicate;**
  - UPPER(last_name): **index filter predicate** that helps to group the data with last_name in similar pattern in some order.

  The index above will help speed up for INDEX RANGE SCAN as the LIKE filter also applied.

  - So why don't we create an index for all columns in the where clause?
    - Bigger size for the index space.

    - Need to take care of the column order to determine what's conditions to used as access predicate.

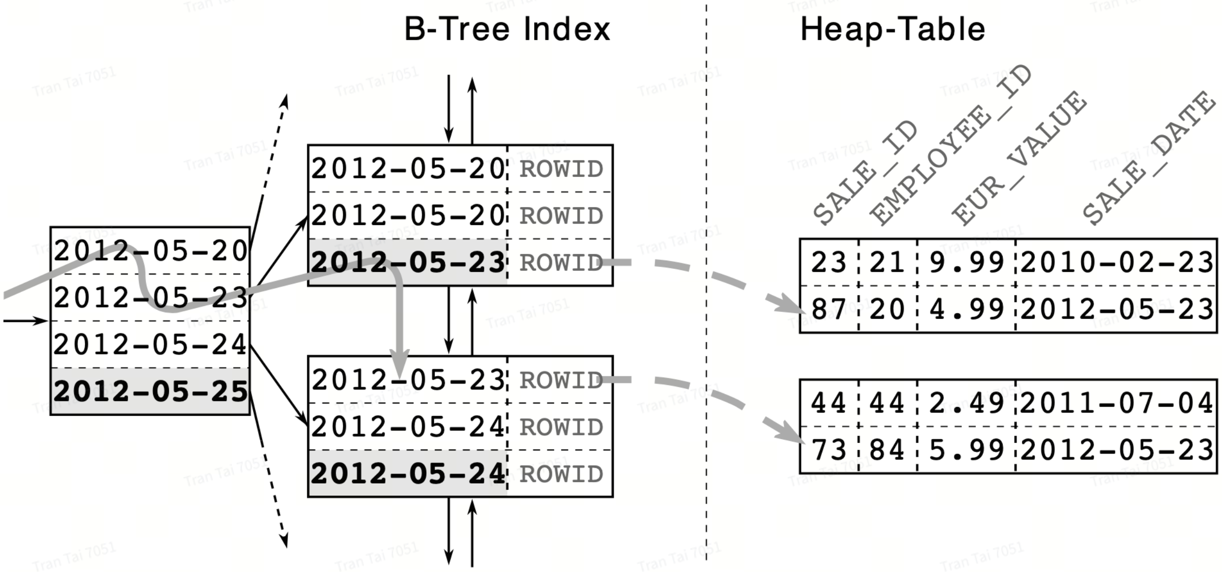
Index-Only Scan

- One of the most powerful tuning methods that **avoid scanning the table**. The table scanning could be avoid completely if the database can find selected columns in the index.
- A index covers the entire query is called **covering index**. So The more columns you query, the more columns you have to add to the indexed to support an index-only scan.
- Most databases impose some limitation on the number of columns per index and total size of an index entry.
- The performance advantage depends on:
    - **Index Clustering Factor:** if the respective rows already distributed in few blocks -> the advantage will not be that enormous.
    - **No of access rows:** if tree traversal needs to fetch a few blocks, the improvement will not be significant.
- This index only scan will increase maintainance effort for **update** stmt.

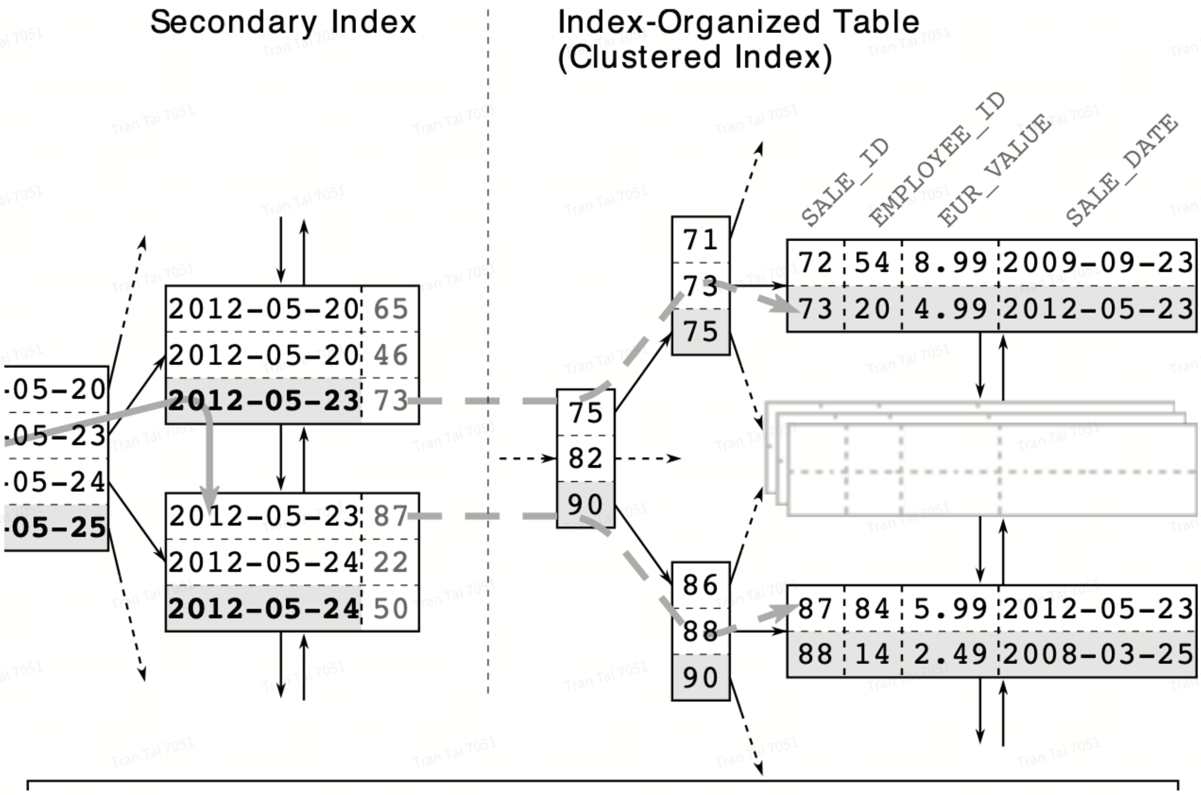
Index-Organized Table

- B-Tree with heap table

Figure 5.1. Index-Based Access on a Heap Table



- Index-Organized Table is a B-Tree without a heap table.
    - Save spaces for heap.
    - Access every clustered index is an **index-only scan**.



- The original data is stored in the Index-Organized Table.
- When a new index is built (secondary index), it refers to the logical key from the cluster-organized table instead of rowID as the old index.

3. Sorting And Grouping

- The 3rd power of indexing.

Indexing Order by

- If the index order **corresponds to the order by claus**e, the db will omit explicit sort op.

```
SELECT sale_date, product_id, quantity
  FROM sales
 WHERE sale_date = TRUNC(sysdate) - INTERVAL '1' DAY
 ORDER BY sale_date, product_id;
```

- If the index using is (sale_date, product_id), no SORT ORDER BY operation is needed.
- This operation also works if we just sort by **product_id**.

- When **extending scanned index range**, the optimization, might not help get rid of explciit SORT ORDER BY.

```
SELECT sale_date, product_id, quantity
  FROM sales
 WHERE sale_date >= TRUNC(sysdate) - INTERVAL '1' DAY
 ORDER BY product_id;
```

- If db uses explicit sort op which is not our expectation:
  - (1) execution plan with explicit sort op has better cost.
  - (2) index order in scanned index range does not correspond to the ORDER BY clause.
  - Can distinguished between two cases by adjusting the query using full index definition.
- Using EXPLAIN show CREATE TEMP FILE -> bad sign.

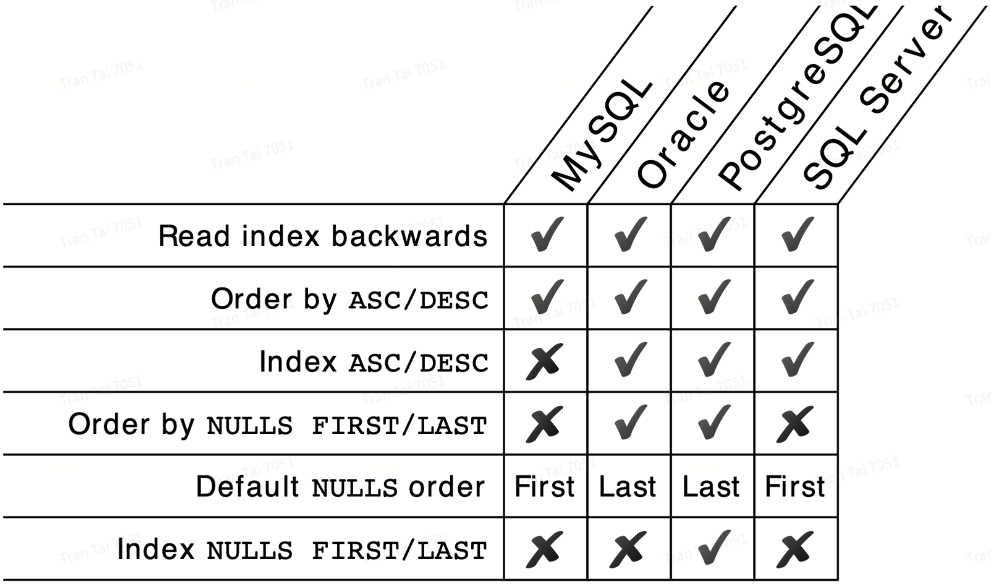
Indexing ASC/DESC and NULLs FIRST/LAST

- Database can read the indexes in both directions (ASC, DESC). For DESC, it find the last matching entry, then follows the back pointer in the doubleLinkedList node. This explains why our list node is a double linked list.
- ASC/DESC modifiers can be used in the index declarations that optimize (avoid explicit SORT) for the query involves SORT by multiple columns in different ASC/DESC order.

```
CREATE INDEX sales_dt_pr
    ON sales (sale_date ASC, product_id DESC);
```

- For NULLs, the standard is that all NULLS must appear tgt aft sorting but not specify it will be appeared before/aft other entries.

Figure 6.5. Database/Feature Matrix



	MySQL	Oracle	PostgreSQL	SQL Server
Read index backwards	✓	✓	✓	✓
Order by `ASC/DESC`	✓	✓	✓	✓
Index `ASC/DESC`	✗	✓	✓	✓
Order by `NULLS FIRST/LAST`	✗	✓	✓	✗
Default `NULLS` order	First	Last	Last	First
Index `NULLS FIRST/LAST`	✗	✗	✓	✗

Indexing Group by

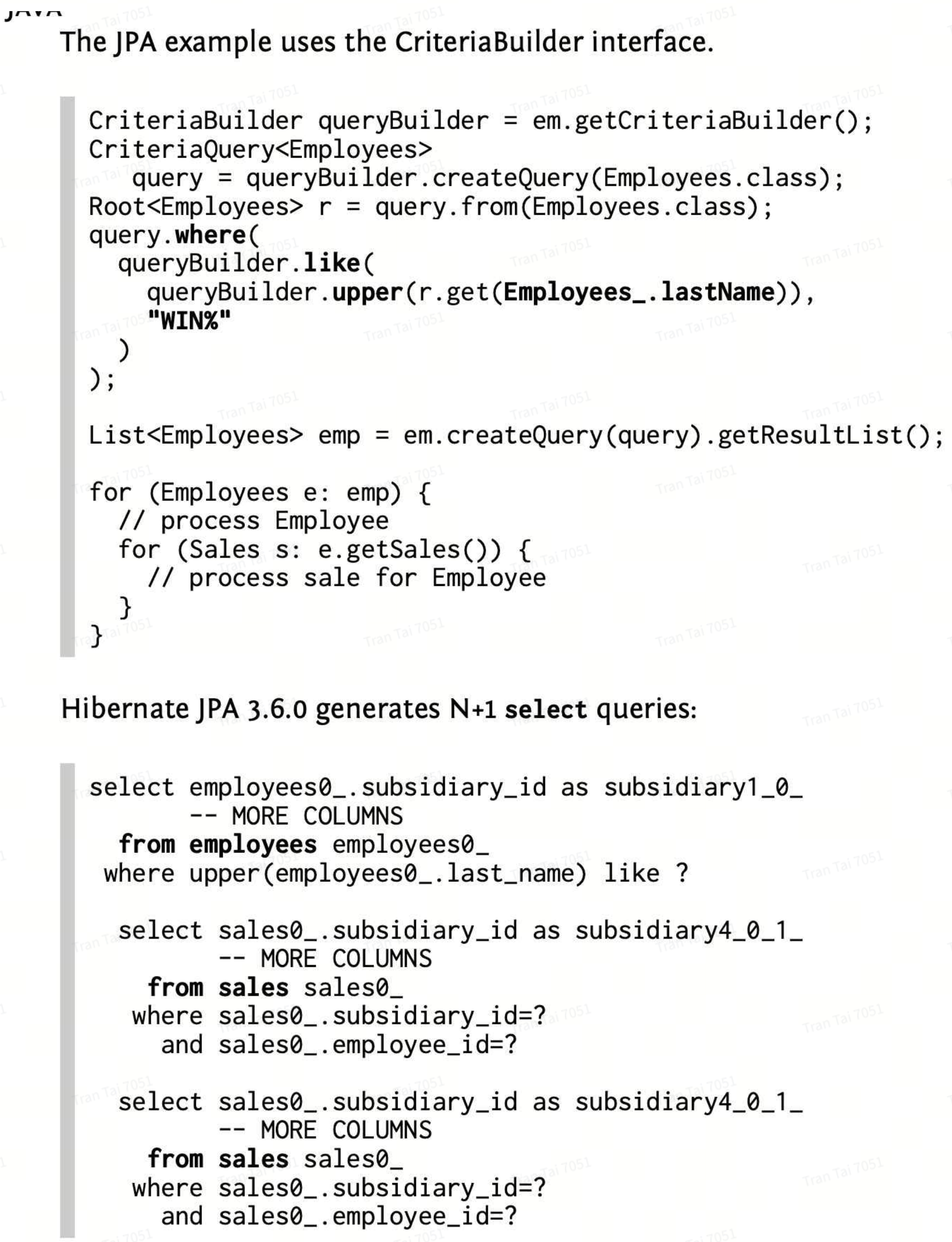
- There are 2 types of group by algorithms:
  - Hash algorithm: aggregate records in a temp hash table. **Only need to buffer the aggregated result. Need less memory**
  - Sort algorithm: sort the input by grouping key. **Materialized the complete input set**.

4. Join

- Transforms data from normalized model to denormalized form. Koining is sensitive to disk seek latencies because it combines scattered data fragments.
- Proper indexing will help to reduce the response time.
- All joins algorithm processes **only two tables at a time**. SQL query with more tables required multiple steps to build **intermediate result** set by joining 2 tables, then joining result with the next table and so forth.
  - Join order is not affect the final result but will affect the performance.
  - It is optimizer responsibility to determine the best join order strategy and select the best one. The more tables to join, the more execution plan to evaluate.
  - Intermediate result does not mean the database has to materialize it. Database can use **pipelining** to reduce memory usage. Each row from intermediate result is **pipelined to the next join operation**.
- Difference joins algorithms:

Nested Loop

- Work like a 2 nested loop: Each loop fetch the result from one table.
- **Nested select** can be used to implement the nested loop join but this will introduce the **N + 1 SELECT problem**. This is notoriously well-known issues of ORM tool. e.g. For JPA to join



JAVA

The JPA example uses the CriteriaBuilder interface.

```
CriteriaBuilder queryBuilder = em.getCriteriaBuilder();
CriteriaQuery<Employees>
   query = queryBuilder.createQuery(Employees.class);
Root<Employees> r = query.from(Employees.class);
query.where(
  queryBuilder.like(
    queryBuilder.upper(r.get(Employees_.lastName)),
    "WIN%"
  )
);

List<Employees> emp = em.createQuery(query).getResultList();

for (Employees e: emp) {
  // process Employee
  for (Sales s: e.getSales()) {
    // process sale for Employee
  }
}
```

Hibernate JPA 3.6.0 generates N+1 **select** queries:

```
select employees0_.subsidiary_id as subsidiary1_0_
       -- MORE COLUMNS
  from employees employees0_
 where upper(employees0_.last_name) like ?

  select sales0_.subsidiary_id as subsidiary4_0_1_
         -- MORE COLUMNS
    from sales sales0_
   where sales0_.subsidiary_id=?
     and sales0_.employee_id=?

  select sales0_.subsidiary_id as subsidiary4_0_1_
         -- MORE COLUMNS
    from sales sales0_
   where sales0_.subsidiary_id=?
     and sales0_.employee_id=?
```

- Multiple queries will introduce **network latency (each query separately sent to the db introduce this kind of latency) on top of disk latency**.
- Internally, **the database implements the nested loop join exactly the way nested selects do** but it is better than the nested select because it get rid of the network latency. To speed up those select queries, can create some extra indexes for the lookup.

```
CREATE INDEX emp_up_name ON employees (UPPER(last_name));
CREATE INDEX sales_emp ON sales (subsidiary_id, employee_id);
```

- ORM tools can offer some way for creating join. **Eager fetching mode** can be configured at the property level in entity mappings such that we will join 1 table when accessing the data from another table. Only do this when we always need some records to go along with another records from another table.
- This join algo will deliver good performance if the driving query returns a small result set.

Hash Join

- Aim for weak spot of nested loops join about many B-tree traversals when executing inner query.
- It will load the candidate records from one side of the join into hash table that can be probed quickly for each row from other side of the join.
- Indexing strategy: done on the independent **where predicate**. **Index on join predicate does not improve hash join performance**
- Another approach to optimize hash join performance is to minimize the hash table size when loading in memory.
    - Only select needed columns.
    - Adding relevant filter.
- Warning: MySQL does not support hash joins.

Sort Join

- Combine two sorted lists. **Both sides of the join must already be sorted by the join predicates**.
- It needs the same index as hash join - **index for independent conditions** to read all candidate records in one shot.
- Sorting both sides is very expensive, so hash join is often preferred.
- This helps to do both left/right join at the same time - full outer join.
- The strength appears when the input is already sorted.

- Enabling SQL Logging is critical when using ORM tool for viewing the raw query send for debugging, especially in performance.

5. Side questions

- Cardinality in database.
- MySQL special indexes and pros.

6. References

- SQL Performance Explained Book.
- https://medium.com/@hemangdtu/indexing-in-databases-unlocking-the-power-of-data-retrieval-362008983e46

7. Next sharings:

- Message Queues Pro/Con + Kafka Case Study.