Furkan ÖZEU
161044036

1)

Advantages:

B+-trees are widely used. It automatically reorganizes it self with small, local changes across additions and deletions. It is not necessary to reorganize the entire file to maintain performance. Non-leaf nodes are larger, so fan-out is reduced. Thus, B+-Trees typically have lesser depth than corresponding B-Tree. Implementation is easier than B-trees.

Disadvantages:

As many overflow blocks are created, performance degrades as the file grows. The entire file needs to be reorganized periodically. The extra insertion and deletion overhead requires space overhead. It can use more tree nodes than a corresponding B-Tree.

2)

Advantages:

For most queries, hash provides consistently better performance as rows are always evenly distributed across AMP's, providing a consistently balanced workload across parallel volumes. Because the Teradata Database uses B*-tree on fixed-length rowID rather than column values, the B*-tree never has more than two levels, so there is no need to refactor it because the trees never become unbalanced or unordered. This applies to standard operation. When you add nodes or AMP's to your system, the system needs to be reconfigured. B+-trees with variable-length values can become unstable and require a long rearrangement later on. Improved performance due to less I/O's. Reduced storage overhead as there is no primary directory subtable.

2)

Disadvantages!

The hash index is stored in the hash after the hash values, and the size of the hash value is not necessarily the same as the key value before the hash, so the database cannot use the indexed data avoiding any sorting. Hash indexes cannot be queried with partial index keys. for B-Tree-based indices, computation time is often significant to search within each node. The B- Tree family is better if you have any range queries and/or want sorted results on predeterminable columns. The hash index can't be used to avoiding sorting operations on the data

3)

for queries with conditions on several search keys, efficency may not be bad even if only some keys have indexes on them. Therefore, database performance is less improved by adding indexes when there are already many indexes. Each extra index requires additional storage. Indexes on non-primary keys may need to be changed in updates, although an index on the primary key doesn't change. Because updates usually don't change primary key attributes. Each index requires additional CPU time and disk I/O load during insertion and deletion.

4)

In general, it is not possible to have two primary indexes on the same relationship for different keys. Because tuples in a relationship must be stored in different order for the same values to be stored together. We can achieve this by storing the relationship twice and duplicating all values, but for a centralized system this is not efficient

5)

a)

The cost of finding the page number of the leaf page required for an insert is negligible since the non-leaf nodes are in memory. At the leaf level, 1 random disk access is required to read and 1 random disk access to update, along with the cost of writing page. Additions that cause Leaf nodes to split require additional page writting.

To build B^+-tree with n_r entries :
$2 * n_r$ random disk access
$n_r + 2 * (n_r / f)$ page writes.

The other part of the cost comes from the fact that in the worst case each leaf is half full, so the number of splits that occur is twice (n_r/f)

The above formula ignores the cost of writting non-leaf nodes as we assume they are in memory, but in reality they are also written at the end. This cost is closely related to the number of internal nodes $(2 * (n_r/f)/f)$, just above the leaf; we can add more terms to account for higher node levels, but they are much smaller than the number of leaves and can be ignored.

b)

Substituting the values in the above formula and neglecting the cost of writting the page, It takes about (10 million $*$ 20 ms) since each insertion cost 20 ms. $= 2 \times 10^8$ ms

Or formula like that : $2 * n_r$ random access time
2×10^7
disk access take 10 ms
$2 \times 10^7 \times 10 = 2 \times 10^8$ ms

6)

relation $R(A,B,C)$, search keys are $R(A,B)$

child nodes contain actual values
non-leaf nodes contain temporary values
key values stored in ascending order

a)

Worst case time complexity to find records between $10 < A < 50$ using the index for number of records n_1.

In the worst-case scenario, it would have to traverse the entire tree of height h for each record retrieval. So the cost of a single record is $1 * h$.

for n_1 records, worst-case time complexity $(n_1 * h)$ to get all records.
h = total height of tree

b)

$10 < A < 50 \wedge 5 < B < 10$

For two cases, same number of records found. Because the matching tuples between these two conditions are the same for n_1 and n_2 records.

Therefore, same worst case time complexity is $(n_1 * h)$ to find satisfactory records to n_2.

7)

a) To insert to the given materialized view it was necessary to set the trigger on the add to account and depositor.
It assumed that the database system uses instant binding for rule execution.
It is assumed that the current version of a relationship is represented by the relationship name, and the newly added set of tuples is represented by qualifying the relationship name with the new_inserted prefix.

Rules for insertion:

```
define trigger insert-to_branch_cust_by_account
after insert on account
referencing new table as new_inserted for each statement
insert into branch-cust
        select branch_name, customer_name
        from new_inserted, depositor
        where new_inserted.account_number = depositor.account_number
```

```
define trigger insert-branch-cust-by-depositor
after insert on depositor
referencing new table as new_inserted for each statement
insert into branch_cust
        select branch-name, customer_name
        from new_inserted, account
        where account.accountnumber = new_inserted.account_number
```

If the execution binding was delayed, the result of merging new tuples in the account with new tuples from the depositor would have been added by both active rules, which could lead to double values for the corresponding tuples in the branch cust.

7)

b)

```
Create trigger check_delete_trigger after delete on account
referencing old row as orow
for each row
delete from depositor
where depositor.customer_name not in
      Select customername from depositor
      where accountnumber <> orow.account_number
end
```

8)

Characteristics of NoSQL:

* Innovative: NoSQL provides a single way to store, retrieve, and manipulate data. NoSQL supporters are inclusive
* Most NoSQL systems use low-cost commercial processors with seperate RAM and disk.
* Schema-free: NoSQL systems allow you to drag and drop your data into a folder and then query it without creating an entity-relational model
* No Joins: NoSQL systems allow you to extract your data using simple interfaces without joins.
* Runs on many processors: NoSQL systems allow you to store your database on multiple processors and maintain high-speed performance
* Supports linear scability: Add more processors and you'll get a consistent increase in performance

8)

NoSQL	vs SQL
1.) NoSQL are non-relational	SQL are relational
2.) NoSQL databases have dynamic Schemas for unstructured data.	SQL databases use structured query language and have a predefined schema.
3) NoSQL databases are horizontally Scalable	SQL databases are vertically scalable
4) NoSQL databases are document, key-value, graph or wide-column stores	SQL databases are table based
5) NoSQL are better for unstructured data like documents or JSON	SQL databases are better for multi-row transactions.
6) Commodity network	Highly available network