The PostgreSQL Btree Index

INFO-H417: Lab Session 5

2023-2024

PostgreSQL allows the creation of indexes, which are special database objects used to speed up data access. This exercise session will focus on the *Btree* index. **Read the blog *Indexes in PostgreSQL — 4 (Btree)*[1] to answer the following questions about Btree indexes in PostgreSQL.** The blog uses a demo database that can be found here[2].

1. What data is stored in the elements of the leaf nodes?
   The leaf nodes store pairs of keys (the data/value to be indexed) and TIDs (references to table rows).

2. What data is stored in the elements of the internal and root nodes?
   The elements of the internal or root nodes contain references to a child page, as well as the minimum key in that page.

3. What is required for a type to be indexed using a Btree index? That is: if we define a new type and want to be able to create a Btree index on it, what operators also need to be defined for this type? Why is this necessary?
   The type needs to have the operators *less than*, *less than or equal*, *equal*, *greater than* and *greater than or equal*. These operators are required to be able to sort the values of the given type, and this sorting is required for the creation of the Btree index.

4. Does the Btree store the TIDs of all the rows of the indexed column (like a dense index) or only a subset of the rows (like a sparse index)?
   Btree indexes store the TIDs of all rows.

5. What properties of the Btree allows us to answer inequality ($<, <=, >, >=$) or range searches (*between*) efficiently?
   The two important properties are: (1) the keys stored in a Btree are stored and (2) the nodes contain pointers to the previous and next node in the same level of the

[1] https://postgrespro.com/blog/pgsql/4161516
[2] https://postgrespro.com/docs/postgrespro/9.6/demodb-bookings

tree. We can then for example answer an inequality search by first looking for the boundary value using the equality condition and then iterate along the leaf nodes in the correct direction to return all the searched rows.

6. Let's assume that an index contains 3 levels, each containing respectively 1 (root), 10 and 1000 nodes/pages. On average, how many index pages will be fetched when answering a query containing a search by equality? How many will that be when the query contains a search by inequality?
   Assuming that the indexed column does not contain many duplicates, a search by equality will on average fetch 3 index pages. A search by inequality on the other hand will on average fetch 502 pages, assuming that the compared value is random between the min and max, and that the values in the column are more or less evenly distributed.

7. How can a Btree index be used to answer a query of the form: `SELECT ... FROM ... ORDER BY col`? Could we use the same index for a query of the form: `SELECT ... FROM ... ORDER BY col DESC`?
   A Btree index stores the keys in a sorted order (ASC by default). To return rows sorted by a column indexed using a Btree, we can thus simply scan the index and return the rows in the order returned by this index scan. Btree indexes can be scanned backwards aswell. We can thus use the same index for both queries.

8. What is an *index only scan* and when is it used? Give an example of a query where an index only scan would be used.
   An index only scan, as the name suggests, only scans the index and does not fetch the pages of the table itself. This is used when the index already contains all the information required by the query. For example, the query `SELECT ticket_no FROM tickets;` will use an index only scan if there is a Btree index on the column `ticket_no`.

9. Let's assume we have a table `T(A, B)` and we create two multi-column indexes on it.
   ```
   CREATE INDEX idx_ab ON T(A, B);
   CREATE INDEX idx_ba ON T(B, A);
   ```
   Which of the two indexes will be used to answer the following queries, and why?
   ```
   SELECT A, B FROM T WHERE A = 'a';
   SELECT A, B FROM T WHERE A = 'a' AND B = 'b';
   SELECT A, B FROM T WHERE B = 'b';
   ```
   The first query will be answered using the index `idx_ab`, and the third one using the index `idx_ba`. The second query can be answered by both indexes. This comes from the fact that multi-column indexes sort their keys first by their first element and

secondly by their second element. This means that the elements in the intermediate nodes do not contain enough information to search a key using only its second element.

10. How are `NULL` values handled by Btree indexes?
    All `NULL` values (and their respective TIDs) are stored together either at the start or at the end of the leaf nodes. This allows the creation of Btree indexes on columns that contain `NULL` values, even though a `NULL` value is not sortable. For some queries working with `NULL` values, this index can thus still be used.