TÉCNICO LISBOA

Índices

Slides e Soluções do Laboratório 11

Default Indexes

1. Let's populate the **`account`** table by running the **`\i index_data.sql`** in the psql shell.
2. To obtain information about the indexes of the account table, run the command **`\d account`**. The system seems to have created an index for the table's primary key by default. What kind?
   - A Btree index
3. Enable automatic execution time reporting in the psql shell with the command **`\timing`**.
4. Now when you run the following query:

   ```
   SELECT * FROM account WHERE account_number='A-012345';
   ```

   You should see first the results followed by the time it took the system produce them
5. Now lets delete the primary key (along with the index that was created by default):

   ```
   ALTER TABLE account DROP CONSTRAINT account_pkey;
   ```

Default Indexes

6. Repeat step 4 and take note of the time. How do you explain this result?

   - Removing the primary key also removed the Btree index. This means that PSQL now has to scan the whole heap (i.e. table file) whereas before it just had to navigate the tree structure of the index file to find any specific account. Thus, we should observe a massive increase in run time from 4 to 6.

7. So, let's add back the primary key:

   ```
   ALTER TABLE account ADD PRIMARY KEY(account_number);
   ```

   Notice it took some time to execute this command. Why?

   - To recreate the index, PSQL has to scan the whole heap for the table, sort the index column and organize it in the Btree structure

8. Repeat step 4 and note the general trend in terms of execution time.

   - The execution time should be similar to the original 4.

Index Creation

9. Let's run \i index_data.sql again to populate the account table again and clear any changes.

10. Run the query and note its execution time

```
SELECT account_number FROM account WHERE balance=1000;
```

11. Run the query and note its execution time

```
SELECT MAX(balance) FROM ACCOUNT;
```

12. Now let's create an index for the column "balance" used in both queries with the command:

```
CREATE INDEX balance_idx ON account(balance);
```

Is this a primary index or a secondary index?

- It is a secondary index, as the data in the table is not ordered by the index

Index Creation

13. To obtain information about the indexes of the account table, run the command **`\d account`**. The system created an index for the balance column. What kind?

    - A Btree index, which is the default type of index in PSQL

14. Now repeat the queries from step 10 and 11 and note their new execution time. For both queries, how do you explain the time difference?

    - The run time of both queries is substantially reduced with the index. This is because without the index, PSQL has to scan the heap in search of all values that match the selection (10) or do a full scan to find the maximum (11). With the index, it can quickly iterate through the Btree to find the desired values.

15. Now delete the index created previously in step 12

```
DROP INDEX balance_idx;
```

TÉCNICO LISBOA

Index Creation

16. Let's explicitly create a HASH index for the column balance, specifying the type of index like so:

```
CREATE INDEX balance_idx ON account USING HASH(balance);
```

17. Now repeat the queries from step 10 and 11 and note their new execution time. For both queries, how do you explain the time difference?

    - Query 10 should run slightly faster with the Hash index than with the previous Btree index, since a Hash index is more efficient for equality comparisons (lookup on the Hash table is near-instantaneous whereas lookup on the Btree requires iterating through a few nodes.

    - Query 11 runs significantly slower with the Hash index than with the Btree index. Indeed it runs as slow as without any index, because the Hash index cannot be used for range comparisons (the Hash table is unordered), and finding the maximum is a range comparison.

18. You can now delete the index created in step 16:

```
DROP INDEX balance_idx;
```

TÉCNICO LISBOA

Execution Plans

19. Let's run `\i index_data.sql` again to populate the account table again and clear any changes.

20. Get the execution plan for the queries of step 10 and 11 with the commands:

```
EXPLAIN SELECT account_number FROM account WHERE balance=1000;

EXPLAIN SELECT MAX(balance) FROM ACCOUNT;
```

    What access method is used in each query? Justify.

- In both, the access method is a sequential scan of the heap, as the EXPLAIN indicates “Seq Scan” for both queries, with a filter condition for the first one, and with an aggregation for the second.

Execution Plans

21. Now create a B+TREE index on the balance attribute and check the access plan again:

```
CREATE INDEX balance_idx ON account(balance);

EXPLAIN SELECT account_number FROM account WHERE balance=1000;

EXPLAIN SELECT MAX(balance) FROM ACCOUNT;
```

What difference do you see in the access method in each query?

- In the first query, there are two paired access methods:
  - First a Bitmap Index Scan is used to scan the index file and build a bitmap of the rows that meet the filter condition.
  - Then a Bitmap Heap Scan is used to scan the heap using the bitmap as index and rechecking the filter condition. This is PSQL's default behavior for filters on index columns (to double-check the heap).
- In the second query the access method is an Index Only Scan, which is done Backwards, as the maximum is the always the last value in the Btree.

TÉCNICO LISBOA

Execution Plans

22. Create a HASH index for the balance column, compare the access plan with step 20

```
DROP INDEX balance_idx;

CREATE INDEX balance_idx ON account USING HASH(balance);

EXPLAIN SELECT account_number FROM account WHERE balance=1000;

EXPLAIN SELECT MAX(balance) FROM ACCOUNT;
```

How do you explain that the hash index is never used for the second query?

- In the first query, the access methods remain the same, as PSQL's default behavior for filters on index columns is the same regardless of whether the index is Hash or Btree.
- In the second query the access method is a Sequential Scan (the same as no index) because the Hash index is not ordered and thus is useless for range searches, which finding the maximum corresponds to.

TÉCNICO LISBOA

Query Optimisation

22. Consider the following table:

```
CREATE TABLE employee (
  eid INTEGER PRIMARY KEY,
  ename VARCHAR(40) NOT NULL,
  address VARCHAR(255) NOT NULL,
  salary NUMERIC(12,4) NOT NULL,
  bdate DATE NOT NULL);
```

Think about the indexes you could/should create to improve the efficiency of frequently executed queries. For this exercise, let's suppose that the following queries are quite common:

a. What is the identifier, name, and address of employees aged within a certain range?

- Only age is a filter in this query (WHERE clause), the other columns are just to be returned (SELECT clause) so an index is only needed on bdate (from which age is derived). As the desired filter is a range filter, the index should be Btree:
  - Btree(bdate)

Query Optimisation

22. Consider the following table:

```
CREATE TABLE employee (
  eid INTEGER PRIMARY KEY,
  ename VARCHAR(40) NOT NULL,
  address VARCHAR(255) NOT NULL,
  salary NUMERIC(12,4) NOT NULL,
  bdate DATE NOT NULL);
```

Think about the indexes you could/should create to improve the efficiency of frequently executed queries. For this exercise, let's suppose that the following queries are quite common:

b. What is the identifier and address of employees with a given name?

- Only name is a filter in this query (WHERE clause), the other columns are just to be returned (SELECT clause) so an index is only needed on name. As the desired filter is an equality filter (as is the typical case for string comparisons), the optimal index is Hash:
  - Hash(name)

Query Optimisation

22. Consider the following table:

```
CREATE TABLE employee (
  eid INTEGER PRIMARY KEY,
  ename VARCHAR(40) NOT NULL,
  address VARCHAR(255) NOT NULL,
  salary NUMERIC(12,4) NOT NULL,
  bdate DATE NOT NULL);
```

Think about the indexes you could/should create to improve the efficiency of frequently executed queries. For this exercise, let's suppose that the following queries are quite common:

c. What is the maximum salary for employees?

- As we’ve seen in the previous section, finding the maximum of a column is a range query, which benefits substantially from a Btree index:
  - Btree(salary)

TÉCNICO LISBOA

Query Optimisation

22. Consider the following table:

    Think about the indexes you could/should create to improve the efficiency of frequently executed queries. For this exercise, let's suppose that the following queries are quite common:

    d. What is the average salary of employees by age?

- “By age” is a hidden range filter, as we typically consider age with a granularity of years yet bdate has a granularity of days. This means that for any given age, we want all people with bdate >= CURRENT_DATE - age AND bdate < CURRENT_DATE - age + 1. As such, we need a Btree index on bdate. No index is needed on salary, since we need to scan all salaries for the specified age in order to compute the average.
    - Btree(bdate)

- If we wanted the maximum salary, and had a very large database (with many people sharing the same bdate) there might be some gain in performance by having a composite index on (bdate,salary) but in “normal” circumstances, we wouldn’t define a composite index even for finding the maximum, as salary values are likely to be randomly distributed across the portion of the btree on bdate that corresponds to any given age.