Database Tuning, ITU, Spring 2008

Rasmus Pagh

March 11, 2008

Project – deliverable 2

Deadline: March 20, 23.59 Danish time.

Time plan for deliverable 2

February 26. Description, question/answer session.

March 4 and 11. Office hours for project supervisor (Milan Ružić).

March 20. Hand-in by e-mail (pdf file) to `milan@itu.dk`

Around March 27. Feedback (individually scheduled for each group).

Purpose

The main purpose of this part of the project is to get experience with index tuning. Proper selection of indexes can speed up the queries significantly with only a small overhead in terms of updates, whereas an improper selection can degrade the performance. Hence a clear understanding of when and how to use indexes is important.

Workload generator

Queries and updates will be generated using a workload generator, a Java program, by substituting various values for the parameters in the "generic" queries found below and update statements implemented in the first deliverable. You will be provided with the workload generator containing the names and parameters of methods that you need to implement. You need to execute this program to test your database.

You can download the workload generator from the ITU file system: `H:/milan/DBT-files` or `/import/home/milan/DBT-files`. Practical questions regarding these can be directed to `milan@itu.dk`.

Updates

A large fraction the operations performed on the VXL database will be updates. Some of these, that are relevant for the queries described later, are mentioned below (this will be extended in later parts of the project):

- Road segments and end points are very rarely updated, hence you can think of them as essentially static.
- Customers, vehicles and drivers will be added and deleted occasionally. The *mileage* attribute of vehicles is updated more frequently.

- History entries for parcels will be added very freqently. In this part of the project, you should keep an eye on the cost of maintaining index(es) for the parcel history. On weekends, around 50% of the database transactions add a tuple to the parcel history, while this number is only 20% on weekdays. You want to make indexing decisions that give a good performance in both situations. The percentage of parcel history updates is a parameter to the workload generator. You want to make a single indexing decision that gives a good performance in both situations (compared to the best performance possible).

Queries

The method (of class DatabaseInterface) that you need to implement is specified for each query.

1. Find all 5-way junctions in a given postal code. It can happen that road segments belonging to different postcodes are incident to a node. For a node to be returned it is necessary that it has exactly 5 adjoining road segments from the given postcode, but it is allowed to have additional roads from other postcodes coming into it.
   public ResultSet queryNodes(int post_code)

2. Find the addresses (street number, street name and postal code) of all customers whose name starts with the string s.
   public ResultSet queryCustomers_01(String namePrefix)

3. Find the postal code with the highest number of customers, and list the names of all the customers with that postal code.
   public ResultSet queryCustomers_02()

4. Find the number of customers in the post codes between x and y.
   public int queryCustomers_03(int x, int y)

5. Find the number of parcels that have arrived at storage s on a given date.
   public ResultSet queryParcelLog_01(int s, Date d)

6. Find the first storage to which parcel p was sent. (It is not sufficient to look at the travel plan, since the actual path might have changed.)
   public int queryParcelLog_02(int p)

7. Find all the parcels that were first sent to storage s.
   public ResultSet queryParcelLog_03(int s)

8. Find all the storage locations in the rectangular region with diagonal coordinates $(x1, y1)$ and $(x2, y2)$. Since the data does not provide exact coordinates of storages, the position should be approximated by the positions of the endpoints of the road segment on which the storage lies. We will say that a storage lies in a rectangular region if either of the corresponding nodes lies in that region.
   public ResultSet queryStorages_01(int x1, int y1, int x2, int y2)

9. Find the IDs of all drivers who drive a vehicle of a given type.
   public ResultSet queryDrivers_01(String type)

10. Find the names of all drivers in region *reg* who drive a vehicle with capacity more than *cap* on a given day of the week, according to the fixed schedule.
    public ResultSet queryDrivers_02(String reg, int cap, byte day)

Theoretical investigation — buffered B-trees

We now consider the possible applicability of buffered B-trees in the database. As you may know, a buffered B-tree is not a standard Oracle index type. Imagine that the third party vendor BlastSoft AB is now offering this functionality as a plugin. However, the price is substantial, so it would be nice to know if it would actually be a big advantage to use their software. In particular, we consider the information in the database that deals with arrival and departure of parcels to/from safe storage containers. A common query is to ask for the latest arrival or departure to/from a particular container. A composite index (clustered) on container ID and time allows this to be answered efficiently. (This assumes that you store this information in a single relation — if this is not the case, modify the questions below accordingly.)

Looking through a BlastSoft whitepaper, you learn that they are using blocks of 8192 bytes for their buffered B-tree. According to BlastSoft, the extra time for reading blocks of this size compared to a normal B-tree node access is negligible. Block pointers are 4 bytes. The root block is stored in internal memory. The maximum degree b of a buffered B-tree can be specified by the user. The number of tuples that fit in a buffer depends on your data model (especially the data types you have used) – part of the problem is to determine this.

Assume in the following for simplicity that the degree of every node of a search tree, except possibly the root, is $\frac{3}{4}$ times the maximum degree. Also, when answering the following it is allowed to round, make estimates, ignore negligible terms, etc. For example, you may ignore the fact that a small fraction of elements may be residing in buffers at any time. The aim is not precise calculation, but good estimation (you should *not* ignore constant factors, but may ignore negligible terms).

a) Estimate the size of tuples in the relation, and in particular how many tuples can be buffered in a B-tree node. Since we are aiming at low-degree trees, you may ignore the space for search keys and pointers.

b) What is the depth of the buffered B-tree in terms of b and the number of keys N?

c) What is the (amortized) cost of updating the buffered B-tree when inserting a new tuple? You are allowed to use the fact that rebalancing cost is negligible. Your answer should be stated in terms of the depth of the tree and the number of records in a buffer.

d) What is the depth of a standard B^+-tree index with block size 4096? Express the number of I/Os for a point query and for an insertion in terms of the depth. As above, you may ignore rebalancing cost.

e) Based on the above questions, compare the I/O cost of doing x insertions and y multipoint queries in the two scenarios: 1. Standard B^+-tree index with block size 4096, and 2. Buffered B-tree index with $b = 4$. You may assume that all the records to be found are in a single leaf node (and possibly in buffers).

To be handed in

A pdf file with the following:

- An updated E/R diagram for the VXL database (if it is changed from your earlier submission).
- A description of all the indexes created: attribute(s) of the search key, index type (clustered or unclustered) etc. Include all SQL statements used to create the indexes.
- Two outputs from the test program: For the 20% case and for the 50% case. (Note that the number of queries is not the same in the two cases.)

- Justification: a breif description indicating why these indexes were created. You can discuss other alternatives that you have thought of or tried. You can also include reports on any experiments done (for example, queries to find the statistics, selectivity, etc.) that are relevant. In particular, you should argue that all the indexes you have created are useful.
- Answer to the theoretical investigation.

Tips on indexing in Oracle

A `CREATE INDEX` statement will create an unclustered B-tree index. The primary key of a relation is automatically indexed. If one wants to cluster the primary index, this must be specified when the relation is created, by using the keywords `ORGANIZATION INDEX` immediately after the relation schema.

It is also possible organize a table in a clustered hash index. First, create the hash table (called a "cluster") using

`CREATE CLUSTER <hashtablename>(<key>) HASHKEYS <hashtablesize>`,

and then, when creating the table, specify that is to be stored in the hash table:

`CREATE TABLE <tablename>(<schema>) CLUSTER <hashtablename>(<key>)`.

The size of the hash table is fixed, i.e., the hash table will not grow or shrink. There are several other parameters of the hash table that one can choose. (If you read the documentation for `CREATE CLUSTER`, you will see that it can also be used to specify that several relations can be stored together, *co-clustered*, but for now you should not consider this.)