CHAPTER 24

Advanced Application Development

Exercises

24.6 Find out all performance information your favorite database system provides. Look for at least the following: what queries are currently executing or executed recently, what resources each of them consumed (CPU and I/O), what fraction of page requests resulted in buffer misses (for each query, if available), and what locks have a high degree of contention. You may also be able to get information about CPU and I/O utilization from the operating system.

Answer:

- Postgresql: The *EXPLAIN* command lets us see what query plan the system creates for any query. The numbers that are currently quoted by *EXPLAIN* are:
  a. Estimated start-up cost
  b. Estimated total cost
  c. Estimated number of rows output by this plan node
  d. Estimated average width of rows
- SQL: There is a Microsoft tool called *SQL Profiler*. The data is logged, and then the performance can be monitored. The following performance counters can be logged.
  a. Memory
  b. Physical Disk
  c. Process
  d. Processor
  e. SQLServer:Access Methods
  f. SQLServer:Buffer Manager
  g. SQLServer:Cache Manager

h. SQLServer:Databases
i. SQLServer:General Statistics
j. SQLServer:Latches
k. SQLServer:Locks
l. SQLServer:Memory Manager
m. SQLServer:SQL Statistics
n. SQLServer:SQL Settable

24.7 a. What are the three broad levels at which a database system can be tuned to improve performance?

b. Give two examples of how tuning can be done for each of the levels.

Answer:

a. We refer to performance tuning of a database system as the modification of some system components in order to improve transaction response times, or overall transaction throughput. Database systems can be tuned at various levels to enhance performance. viz.

i. Schema and transaction design
ii. Buffer manager and transaction manager
iii. Access and storage structures
iv. Hardware - disks, CPU, busses etc.

b. We describe some examples for performance tuning of some of the major components of the database system.

i. **Tuning the schema**
In this chapter we have seen two examples of schema tuning, viz. vertical partition of a relation (or conversely - join of two relations), and denormalization (or conversely - normalization). These examples reflect the general scenario, and ideas therein can be applied to tune other schemas.

ii. **Tuning the transactions**
One approach used to speed-up query execution is to improve the its plan. Suppose that we need the natural join of two relations - say *account* and *depositor* from our sample bank database. A *sort–merge-join*(Section 12.5.4) on the attribute *account-number* may be quicker than a simple nested-loop join on the relations.

Other ways of tuning transactions are - breaking up long update transactions and combining related sets of queries into a single query. Generic examples for these approaches are given in this chapter.

For client-server systems, wherein the query has to be transmitted from client to server, the query transmission time itself may form a large fraction of the total query cost. Using *stored procedures* can significantly reduce the queries response time.

iii. **Tuning the buffer manager**
The buffer manager can be made to increase or decrease the number of pages in the buffer according to changing page-fault rates. However, it must be noted that a larger number of pages may mean higher costs for latch management and maintenance of other data-structures like free-lists and page map tables.

iv. **Tuning the transaction manager**
The transaction schedule affects system performance. A query that computes statistics for customers at each branch of the bank will need to scan the relations *account* and *depositor*. During these scans, no updates to any customer's balance will be allowed. Thus, the response time for the update transactions is high. Large queries are best executed when there are few updates, such as at night.

Checkpointing also incurs some cost. If recovery time is not critical, it is preferable to examine a long log (during recovery) rather than spend a lot of (checkpointing) time during normal operation. Hence it may be worthwhile to tune the checkpointing interval according to the expected rate of crashes and the required recovery time.

v. **Tuning the access and storage structures**
A query's response time can be improved by creating an appropriate index on the relation. For example, consider a query in which a depositor enquires about her balance in a particular account. This query would result in the scan of the relation *account* if it has is no index on *account-number*. Similar indexing considerations also apply to computing joins. i.e an index on *account-number* in the *account* relation saves scanning *account* when a natural join of *account* is taken with *depositor*.

In contrast, performance of update transactions may suffer due to indexing. Let us assume that frequent updates to the balance are required. Also suppose that there is an index on *balance* (presumably for range queries) in *account*. Now, for each update to the value of the balance, the index too will have to be updated. In addition, concurrent updates to the index structure will require additional locking overheads. Note that the response time for each update would not be more if there were no index on *balance*.

The type of index chosen also affects performance. For a range query, an order preserving index (like B-trees) is better than a hashed index.

Clustering of data affects the response time for some queries. For example, assume that the tuples of the *account* relation are clustered on *branch-name*. Then the average execution time for a query that finds the total balance amount deposited at a partic-

ular branch can be improved. Even more benefit accrues from having a clustered index on *branch-name*.

If the database system has more than one disk, *declustering* of data will enable parallel access. Suppose that we have five disks and that in a hypothetical situation where each customer has five accounts and each account has a lot of historical information that needs to be accessed. Storing one account per customer per disk will enable parallel access to all accounts of a particular customer. Thus, the speed of a scan on *depositor* will increase about five-fold.

vi. **Tuning the hardware**
The hardware for the database system typically consists of disks, the processor, and the interconnecting architecture (busses etc.). Each of these components may be a bottleneck and by increasing the number of disks or their block-sizes, or using a faster processor, or by improving the bus architecture, one may obtain an improvement in system performance.

24.8 When carrying out performance tuning, should you try to tune your hardware (by adding disks or memory) first, or should you try to tune your transactions (by adding indices or materialized views) first. Explain your answer.

Answer: The 3 levels of tuning - hardware, database system parameters, schema and transactions - interact with one another; so we must consider them together when tuning a system. For example, tuning at the transaction level may result in the hardware bottleneck changing from the disk system to the CPU, or vice versa.

24.9 Suppose that your application has transactions that each access and update a single tuple in a very large relation stored in a B^+-tree file organization. Assume that all internal nodes of the B^+-tree are in memory, but only a very small fraction of the leaf pages can fit in memory. Explain how to calculate the minimum number of disks required to support a workload of 1000 transactions per second. Also calculate the required number of disks, using values for disk parameters given in Section 10.2.

Answer: Given that all internal nodes of the B^+-tree are in memory, and only a very small fraction of the leaf pages can fit in memory. We can deduce that each I/O transaction that access and update a single tuple requires just 1 I/O operation. The disk with the parameters given in the chapter would support a little under 100 random-access I/O operations of 4 kilbytes each per second. So, number of disks needed to support a workload of 1000 transactions is 1000/100 = 10 disks.

The disk parameters given in Section 10.2 are almost the same as the values in the current chapter. So, the number of disks required will be around 10 in this case also.

24.10 What is the motivation for splitting a long transaction into a series of small ones? What problems could arise as a result, and how can these problems be averted?

Answer: Long update transactions cause a lot of log information to be written, and hence extend the checkpointing interval and also the recovery time after a crash. A transaction that performs many updates may even cause the system log to overflow before the transaction commits.

To avoid these problems with a long update transaction it may be advisable to break it up into smaller transactions. This can be seen as a *group* transaction being split into many small *mini-batch* transactions. The same effect is obtained by executing both the group transaction and the mini-batch transactions, which are scheduled in the order that their operations appear in the group transaction.

However, executing the mini-batch transactions in place of the group transaction has some costs, such as extra effort when recovering from system failures. Also, even if the group transaction satisfies the *isolation* requirement, the mini-batch may not. Thus the transaction manager can release the locks held by the mini-batch only when the last transaction in the mini-batch completes execution.

24.11 Suppose the price of memory falls by half, and the speed of disk access (number of accesses per second) doubles, while all other factors remain the same. What would be the effect of this change on the 5 minute and 1 minute rule?

Answer: There will be no effect of these changes on the 5 minute or the 1 minute rule. The value of n, i.e. the frequency of page access at the break-even point, is proportional to the product of memory price and speed of disk access, other factors remaining constant. So when memory price falls by half and access speed doubles, n remains the same.

24.12 List at least 4 features of the TPC benchmarks that help make them realistic and dependable measures.

Answer: Some features that make the TPC benchmarks realistic and dependable are -

a. Ensuring full support for ACID properties of transactions,

b. Calculating the throughput by observing the *end-to-end* performance,

c. Making sizes of relations proportional to the expected rate of transaction arrival, and

d. Measuring the dollar cost per unit of throughput.

24.13 Why was the TPC-D benchmark replaced by the TPC-H and TPC-R benchmarks?

Answer: Various TPC-D queries can be significantly speeded up by using materialized views and other redundant information, but the overheads of using them should be properly accounted. Hence TPC-R and TPC-H

were introduced as refinements of TPC-D, both of which use same schema and workload. TPC-R models periodic reporting queries, and the database running it is permited to use materialized views. TPC-H, on the other hand, models ad hoc querying, and prohibits materialized views and other redundant information.

24.14 Explain what application characteristics would help you decide which of TPCC, TPC-H, or TPC-R best models the application.

Answer: Depending on the application characterictics, different benchmarks are used to model it.

The TPCC benchmark is widely used for transaction processing. It is appropriate for applications which concentrate on the main activities in an order-entry environment, such as entering and delivering orders, recording payments, checking status of orders, and monitoring levels of stock.

The TPCH (H represents *adhoc*) benchmark models the applications which prohibit materialized views and other redundant information, and permits indices only on primary and foriegn keys. This benchmark models ad-hoc querying where the queries are not known beforehand.

The TPCR (R represents for *reporting*) models the applications which has queries, inserts, updates, and deletes. The application is permitted to use materialized views and other redundant information.