Database Optimization

June 9

2009

A brief overview of database optimization techniques for the database developer. Database optimization techniques include RDBMS query execution strategies, cost estimation, join performance, the proper application of indexing, formulating intelligent queries in the context of a single-server RDBMS environment, and illustration of automated optimization tools.

Kyle Lee
Ayoka, L.L.C.

Table of Contents

Overview 3
Types of Indexes 4
Crafting a Clustered Index 4
Unclustered Indexes 5
SQL Server Automated Index Recommendations 6
Cost Estimation for SELECT Operations 8
Linear Search 8
Binary Search 8
Primary Index / Hash 9
Cost Estimation of JOIN Operations 9
Nested Loop 9
Single Loop (using an index) 10
Sort-Merge Join 10

Introduction

Database management systems are pervasive in the modern world. The notion of a persistent, redundant, and highly distributable library of information has become the single most important concept in our information technology repertoire. In fact, virtually every human being in the Western world interacts with a database management system of some kind on a daily basis—often without using a personal computer at any time throughout the day.

With millions of data transactions taking place every second, it comes as little surprise that database optimization is an area of key research for academic institutions and corporate research and development departments. From a software company's perspective, the relational database most often serves as the core of data-driven software applications, and lack of database optimization in such a key area can incur significant costs to both the client and the company.

The purpose of this paper is to discuss basic database optimization using mathematical cost estimation for different types of queries, a review of join performance, and the effects of various physical access structures on specific query examples. The intended audience should be familiar with SQL and basic relational database concepts – typically an experienced database developer. Specific examples will be given in the context of MS SQL Server 2005, but the concepts they illustrate will be general enough to apply to any SQL-supporting relational DBMS (Database Management System).

After reviewing the paper, the reader will hopefully have a better understanding of how RDBMSs formulate execution strategies for complex queries and be able to use this knowledge to retrieve information at a lower cost.

Indexes

Overview

A **database index** is a physical access structure for a database table that functions much as the name would suggest: it is a sorted file that informs the database of where records are physically located on the disk. To get the idea of what an index does, consider a textbook. In order to find a particular section, the reader can either start reading the book and keep reading until he finds what he is seeking, or, alternatively, he can consult the table of contents and go directly to the desired section. A database index functions much like a textbook index does.

Adding appropriate indexes to large tables is the single most important part of database optimization, as we will see when we discuss some examples of cost estimation. Creating a single index for a large table with no existing indexes can reduce a query's execution time by an order of magnitude. As an example, consider the following scenario. Say we have a database table called EMPLOYEE with 100,000 records. Assume that we wish to perform the following simple query on the table and that no indexes exist on the table:

```
SELECT FirstName, LastName FROM EMPLOYEE WHERE EmpID = 12345;
```

In order to find the employee record with the appropriate EmpID, the database must potentially scan through all 100,000 employee records to return the correct result. This type of scan is referred to as a **full table scan.**

Luckily, a database developer can create an index on the EmpID column to prevent such scans from occurring. Additionally, if this field has a UNIQUE constraint, the DBMS will internally compile the index as a hash table, with each employee ID hashing to the desired record's physical disk address. Scanning thus becomes completely unnecessary, and record location is performed in constant time. After the database developer adds this index, the DBMS can immediately locate the employee record with EmpID 12345—a potential reduction of 100,000 operations.

Types of Indexes

Indexes fall into one of two categories: clustered and unclustered. The primary distinction between the two categories is that an unclustered index does not affect that actual ordering of the records on disk and clustered indexing does. Because clustered indexing affects the physical ordering of the records, there can be at most one clustered index for each table. The same restriction does not apply to unclustered indexes, so we can create as many as disk space allows (although this is not necessarily a good idea, as we'll see in a moment).

Crafting a Clustered Index

An SQL database developer should be aware that the clustered index is the most important index for any table, so it should be crafted with care. As a general guideline, every table should have at least a clustered index. In some cases, clustered indexes are created automatically. In SQL Server, for instance, defining a primary key for a table will automatically create a clustered index for that table. However, once a clustered index has been created for a primary key, it is no longer editable. If the database developer wishes to configure the columns that appear in this database index when dealing with primary keys, it might be a better idea to remove the primary key and create an unclustered index on the former primary key column with the unique constraint enabled. This allows the database developer to create a more custom-tailored clustered index.

Indeed, creating a proper clustered index is important, as not all clustered indexes are created equal. Take, for example, a phone book. If the listings are ordered by city, then by last name, then by first name, it is not difficult to find out how many phone numbers are listed for a particular city. If, however, the listings are ordered by last name, then first name, then city, we would have to go through the whole phone book and count every listing with the desired city—not a fun task for anyone.

The same idea applies to databases. Order the columns in your clustered index based on your business logic. Group them in some way that makes sense for your software application. If your queries are most frequently searching for all customers who have a particular type of vehicle, set the vehicle column as

the first column in your clustered index and the customer ID as the second. Customers are both logically and physically (on disk) grouped by their vehicle type. Carefully crafting these database index strategies will improve your database optimization techniques.

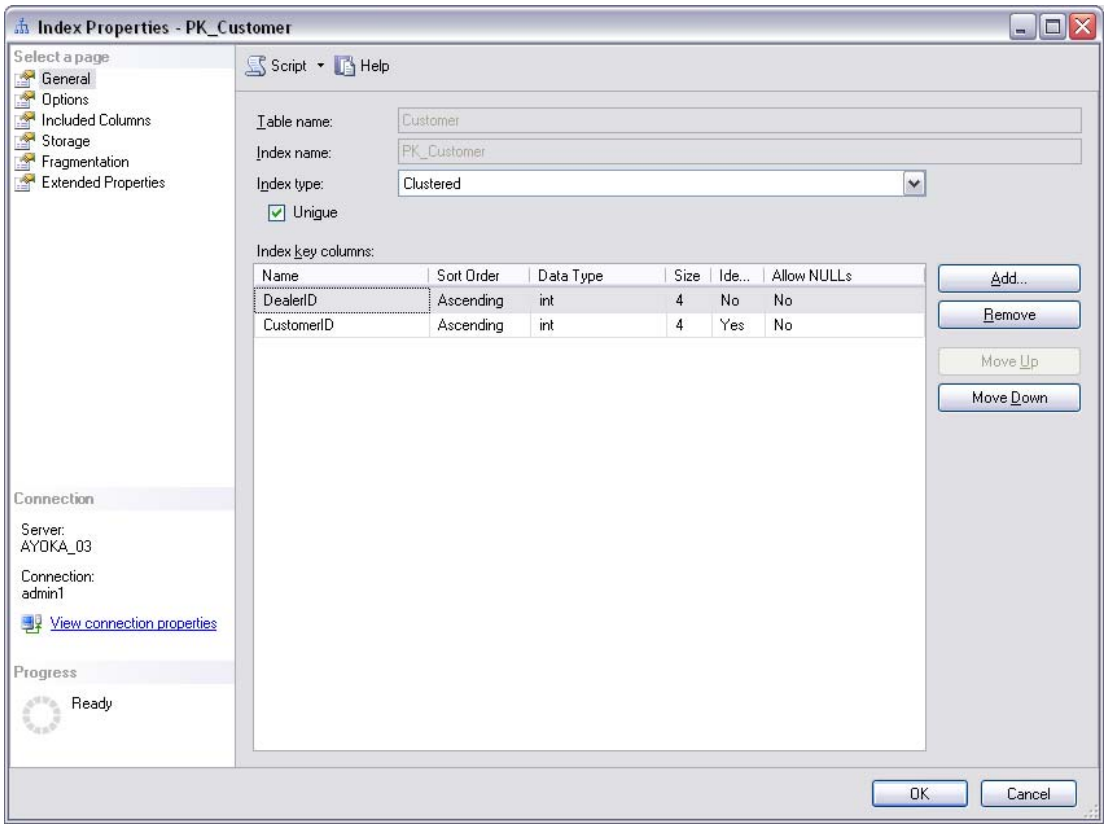

Figure 1

Figure 1 shows an existing clustered index for a Customer table in SQL Server 2005. Since our database keeps track of customer records for many different car dealerships, the customers are grouped by the dealership and then by their customer ID for optimal selection performance.

Unclustered Indexes

From the perspective of the database developer, an unclustered database index does not seem at its surface to be very different from a clustered index. As stated above, the real difference is that a clustered index will reorder the records on disk, whereas an unclustered index will not. Indeed, the same grouping ideas apply just as much to unclustered indexes as they do to clustered indexes, so we will not go over them again.

Unlike the clustered index, there is no theoretical limit on the number of unclustered indexes that can exist for a specific database table. There is, however, two important caveats. The first is that indexes must be stored on disk, and generally, the larger the table, the larger the database index. Disk space consumption can be of concern in many production environments, so the creation of indexes must be balanced against resource scarcity.

The second—and more important—caveat is that indexes must be updated whenever the tables are modified, either through column updates, insertion, or deletion. For large indexes (i.e., for large tables) these database index updates are an expensive and time consuming operation.

The database developer must then enter into a sort of balancing act for adequate database optimization: indexes speed up access time enormously, but they slow down data modification. If your software application rarely performs UPDATE, INSERT, and DELETE operations, creating indexes is not going to incur significant data modification performance penalties beyond their initial creation, but for applications that perform frequent updates, it might be a better idea to carefully tailor the indexes and create as few of them as possible. The database developer can use the reporting utilities in the DBMS to determine what queries are performed the most frequently and their average CPU times to help make this determination.

SQL Server Automated Index Recommendations

SQL Server 2005 and SQL Server 2008 come with a utility called the Database Engine Tuning Advisor. In SQL Server Management Studio, the database developer can highlight the query to optimize and then click the button labeled “Database Engine Tuning Advisor.” A window should appear similar to the following:

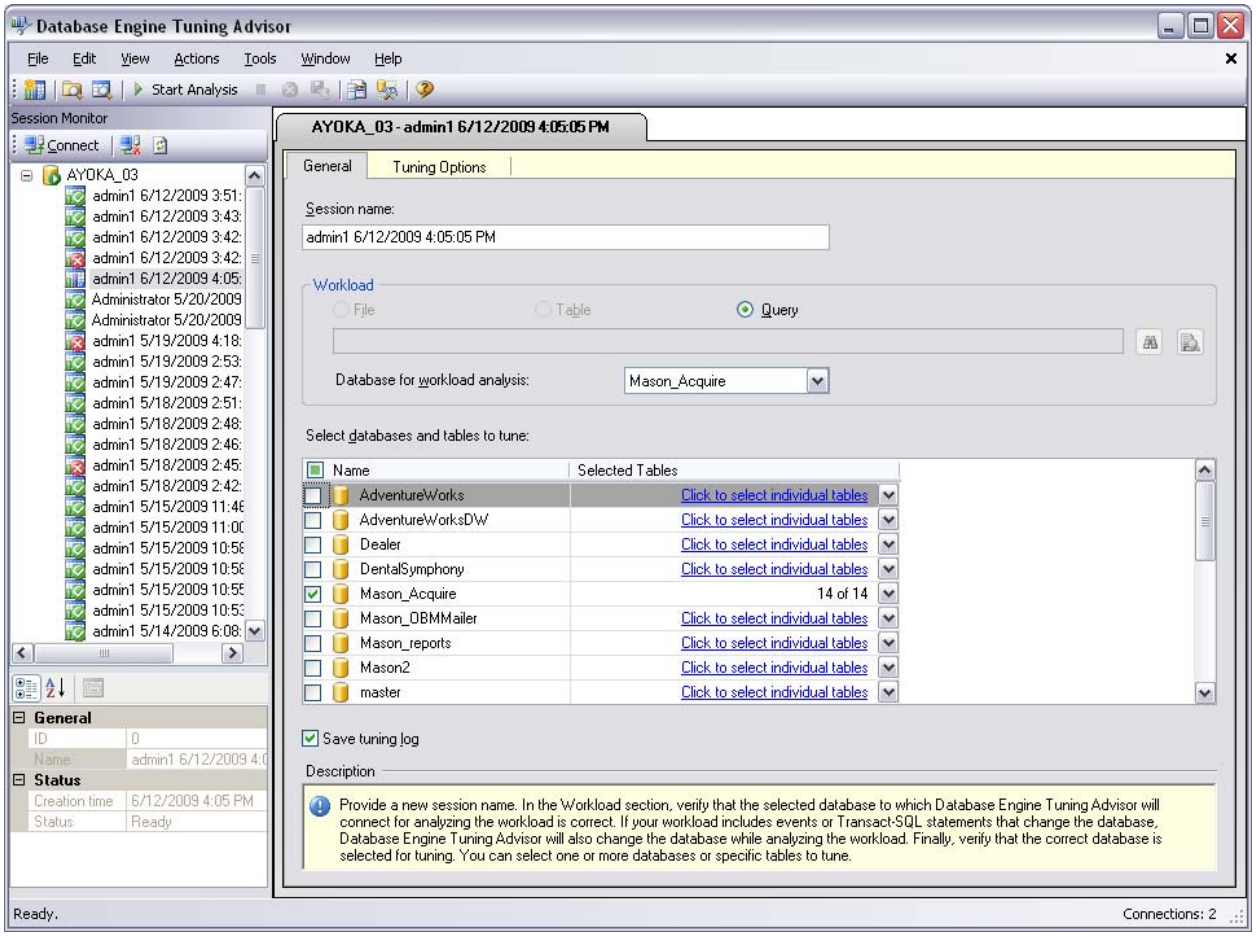

Figure 2

The Tuning Options tab allows the database developer to configure whether or not they want the advisor to replace existing indexes or only consider adding new ones. The database developer should evaluate existing indexes to determine whether or not existing indexes should be dropped.

Once the configuration options are set, clicking the "Start Analysis" button will begin the tuning. Once analysis is complete, a window with recommendations will be displayed along with the estimated performance improvement for the query being optimized. These recommendations include:

1) The creation of indexes.
2) The creation of statistics.

Recommendations may be applied automatically via the Actions > Apply Recommendations menu item.

Cost Estimation

Cost estimation is the process of applying a meaningful and consistent measure of execution cost to a particular query. Various metrics can be used for this purpose, but the most common and most relevant

metric is the number of block accesses required by the query. Since disk I/O is such an expensive operation in terms of time consumption, our goal is to minimize the number of block accesses as much as possible while not sacrificing functionality.

Cost Estimation for SELECT Operations

For a given SELECT query, the DBMS has a number of possible execution strategies. Here are a few that we will discuss (the list is not complete):

1) Linear search (brute force)
2) Binary search
3) Using a primary database index / hash key to retrieve a single record

Assumptions:

Field	Value
Query	SELECT FROM EMPLOYEE WHERE EmpID=125
Number of EMPLOYEE records (r)	100,000
Number of disk blocks (b)	10,000
Blocking factor (bfr) (records per block)	10

Linear Search

The DBMS must use a linear search when no database index exists on the selection condition (e.g., the EmpID). This is precisely the type of operation that we want to prevent with proper indexing.

Given our assumptions, the cost of a linear search for this query would be:

- $C = \frac{b}{2}$ on average if the record exists
- $C = b$ if the record does not exist

So, for the query above, $C = \frac{b}{2} = 5{,}000\ block\ accesses$.

Binary Search

The DBMS might employ a binary search on an index with non-unique entries. Say, for instance, that the database developer has a nonclustered index that groups by city, then first name, then last name, and they need to retrieve all customers who live in a certain city. The DBMS can use a binary search to find the first matching city record and then retrieve all subsequent city records by traversing down the database index row by row.

The cost of a performing a binary search is exactly the same as performing a binary search anywhere else, namely, $C = \log_2 b$.

For the query above, the database developer has $C = \log_2 b = \log_2 10{,}000 = 14\ block\ accesses$.

Primary Index / Hash

If the column is unique (like a primary key, for example) then the database index can be implemented as a hash table. Such an index on the EmpID column would allow the database developer to hash directly to the correct employee record in constant time.

In general, static and linear hashes have a cost of $C = 1$.

For SELECT queries with composite selection conditions, the cost must be evaluated based on the database index structure which exists for each column involved in the join condition—which goes into more detail than this paper is designed to cover. Nevertheless, even for our very simple example query, we can mathematically verify an enormous performance improvement by using a good database index.

Cost Estimation of JOIN Operations

Estimating the cost of join operations is a little more complicated than doing so for SELECT operations because we have to consider the type of indexing available for each table that participates in the join. We must also factor in the cost of writing the result file, but since the size of the result file remains constant despite altering the execution strategy, we can ignore it for the basis of comparison.

Assume that in addition to the EMPLOYEE table, we also have a DEPT table with a DeptNum field indicating the department's number.

Other assumptions:

Field	Value
Number of EMPLOYEE records (r_e)	100,000
Number of EMPLOYEE disk blocks (b_e)	10,000
Blocking factor (bfr) (records per block)	5
Number of DEPARTMENT records (r_d)	25
Number of DEPARTMENT disk blocks (b_d)	5

The query we wish to evaluate is:

1) SELECT * FROM EMPLOYEE AS E, DEPARTMENT AS D WHERE E.DeptNum = D.DeptNum;

We will compare the following execution strategies for this query:

1) Nested-loop join (brute force)
2) Single loop join (using an index)
3) Sort-merge join

Nested Loop

The nested loop strategy may be effective for database optimization when no indexes exist on either table for the join column (in this case, DeptNum). The database developer has two options here. We can either choose to use EMPLOYEE or DEPARTMENT in the outer loop, and the choice we make is very

important. In general, the database developer will want to choose the table with the smaller number of records for the outer loop. This can drastically reduce the number of loop iterations. The DBMS will make this determination automatically, but it is helpful to be aware of what's going on behind the scenes when writing queries.

Using EMPLOYEE as the outer loop:

$$C = b_e + (b_e * b_d) = 10{,}000 + (10{,}000 * 5) = 60{,}000\ block\ accesses.$$

Using DEPARTMENT as the outer loop:

$$C = b_d + (b_d * b_e) = 5 + (5 * 10{,}000) = 50{,}005\ block\ accesses.$$

Using DEPARTMENT as the outer loop cut the number of block accesses down by about 10,000, which is a substantial improvement.

Single Loop (using an index)

If a primary index exists on one of the tables for the join column, then the DBMS can use this index and perform the join with a single loop. So for instance, if a hash-type index exists for the DeptNum column for the DEPARTMENT table, the database developer can hash it rather than perform searching, so an inner loop is unnecessary. Still, the choice of which table we want to loop with is important. Assume that an index exists for DeptNum for both the EMPLOYEE and DEPARTMENT tables.

Note: h is the number of block accesses required to retrieve a record given its hash value.

Using EMPLOYEE as the outer loop:

$$C = b_e + (|DEPARTMENT| * h) = 10{,}000 + 25(1) = 10{,}025\ block\ accesses.$$

Using DEPARTMENT as the outer loop:

$$C = b_d + (|EMPLOYEE| * h) = 5 + 100{,}000(1) = 100{,}005\ block\ accesses.$$

Sort-Merge Join

With this strategy, the DBMS will use a traditional merge on the two sorted files. If the two files are already sorted on the join column, then the cost is simply

$$C = b_e + b_d = 10{,}000 + 5 = 10{,}005\ block\ accesses.$$

If the files are unsorted, the cost of sorting must be factored into the equation. We approximate the sorting as $C_{sort} = blog_2 b$ for a file with b disk blocks.

$$C = b_e + b_d + C_{sort} b_d + C_{sort} b_e = 10{,}000 + 5 + 140{,}000 + 10 = 150{,}015\ block\ accesses.$$

Conclusion

After reading this paper, the database developer will hopefully have a better understanding of basic database optimization techniques and how the DBMS formulates execution strategies for different types of queries, even though the provided examples are very limited in scope. The database developer should also understand the importance of creating well-tuned indexes and what criteria go into selecting the columns for indexing.

References

The cost estimation examples provided in this document were modified from examples provided in Fundamentals of Database Systems 5th edition by Doctors Ramez Elmasri of the University of Texas at Arlington and Shamkant Navathe of the Georgia Institute of Technology.