Performance Tuning SQL Server Joins

文章出处：http://www.sql-server-performance.com/tuning_joins.asp

One of the best ways to boost JOIN performance is to limit how many rows need to be JOINed. This is especially beneficial for the outer table in a JOIN. Only return absolutely only those rows needed to be JOINed, and no more. [6.5, 7.0, 2000, 2005] *Updated 7-25-2005*

*****

If you perform regular **joins between two or more tables** in your queries, performance will be optimized if each of the joined columns have their own indexes. This includes adding indexes to the columns in each table used to join the tables. Generally speaking, a clustered key is better than a non-clustered key for optimum JOIN performance. [6.5, 7.0, 2000, 2005] *Updated 7-25-2005*

*****

If you have two or more tables that are frequently joined together, then the columns used for the joins on all tables should have an appropriate index. If the columns used for the joins are not naturally compact, then considering adding surrogate keys to the tables that are compact in order to reduce the size of the keys, thus decreasing read I/O during the join process, increasing overall performance. [6.5, 7.0, 2000, 2005] *Updated 7-25-2005*

*****

JOIN performance has a lot to do with how many rows you can stuff in a data page. For example, let's say you want to JOIN two tables. Most likely, one of these two tables will be smaller than the other, and SQL Server will most likely select the smaller of the two tables to be the inner table of the JOIN. When this happens, SQL Server tries to put the relevant contents of this table into the buffer cache for faster performance. If there is not enough room to put all the relevant data into cache, then SQL Server will have to use additional resources in order to get data into and out of the cache as the JOIN is performed.

If all of the data can be cached, the performance of the JOIN will be faster than if it is not. This comes back to the original statement, that the number of rows in a table can affect JOIN performance. In other words, if a table has no wasted space, it is much more likely to get all of the relevant inner table data into cache, boosting speed. The moral to this story is to try to get as much data stuffed into a data page as possible. This can be done through the use of a high fillfactor, rebuilding indexes often to get rid of empty space, and to optimize datatypes and widths when creating columns in tables. [6.5, 7.0, 2000, 2005] *Updated 7-25-2005*

*****

Keep in mind that when you create foreign keys, an index is not automatically created at the same time. If you ever plan to join a table to the table with the foreign key, using the foreign key as the linking column, then you should consider adding an index to the foreign key column. An index on a foreign key column can substantially boost the performance of many joins. [6.5, 7.0, 2000, 2005] *Updated 7-25-2005*

*****

Avoid joining tables based on columns with few unique values. If columns used for joining aren't mostly unique, then the SQL Server optimizer may not be able to use an existing index in order to speed up the join. Ideally, for best performance, joins should be done on columns that have unique indexes. [6.5, 7.0, 2000, 2005] *Updated 7-25-2005*

*****

For best join performance, the **indexes on the columns being joined should ideally be numeric data types**, not CHAR or VARCHAR, or other non-numeric data types. The overhead is lower and join performance is faster. [6.5, 7.0, 2000, 2005] *Updated 7-25-2005*

*****

For maximum performance when joining two or more tables, the **indexes on the columns to be joined should have the same data type, and ideally, the same width**.

This also means that you shouldn't mix non-Unicode and Unicode datatypes when using SQL Server 7.0 or later. (e.g. VARCHAR and NVARCHAR). If SQL Server has to implicitly convert the data types to perform the join, this not only slows the joining process, but it also could mean that SQL Server may not use available indexes, performing a table scan instead. [6.5, 7.0, 2000, 2005] *Updated 7-25-2005*

*****

When you create joins using Transact-SQL, you can choose between two different types of syntax: either ANSI or Microsoft. ANSI refers to the ANSI standard for writing joins, and Microsoft refers to the old Microsoft style of writing joins. For example:

ANSI JOIN Syntax

```
SELECT fname, lname, department
FROM names INNER JOIN departments ON names.employeeid =
departments.employeeid
```

Former Microsoft JOIN Syntax

```
SELECT fname, lname, department
FROM names, departments
WHERE names.employeeid = departments.employeeid
```

If written correctly, either format will produce identical results. But that is a big if. The older Microsoft join syntax lends itself to mistakes because the syntax is a little less obvious. On the other hand, the ANSI syntax is very explicit and there is little chance you can make a mistake.

For example, I ran across a slow-performing query from an ERP program. After reviewing the code, which used the Microsoft JOIN syntax, I noticed that instead of creating a LEFT JOIN, the developer had accidentally created a CROSS JOIN instead. In this particular example, less than 10,000 rows should have resulted from the LEFT JOIN, but because a CROSS JOIN was used, over 11 million rows were returned instead. Then the developer used a SELECT DISTINCT to get rid of all the unnecessary rows created by the CROSS JOIN. As you can guess, this made for a very lengthy query. Unfortunately, all I could do was notify the vendor's support department about it, and they fixed their code.

The moral of this story is that you probably should be using the ANSI syntax, not the old Microsoft syntax. Besides reducing the odds of making silly mistakes, this code is more portable between database, and eventually, I imagine Microsoft will eventually stop supporting the old format, making the ANSI syntax the only option. [6.5, 7.0, 2000] *Updated 11-1-2005*

*****

If you have to regularly **join four or more tables** to get the recordset you need, consider denormalizing the tables so that the number of joined tables is reduced. Often, by adding one or two columns from one table to another, the number of joins can be reduced, boosting performance. [6.5, 7.0, 2000] *Updated 11-1-2005*

*****

If your join is slow, and currently includes hints, remove the hints to see if the optimizer can do a better job on the join optimization than you can. This is especially important if your application has been upgraded from version 6.5 to 7.0, or from 7.0 to 2000. [6.5, 7.0, 2000] *Updated 11-1-2005*

*****

One of the best ways to boost JOIN performance is to ensure that the JOINed tables include an appropriate WHERE clause to minimize the number of rows that need to be JOINed.

For example, I have seen many developers perform a simple JOIN on two tables, which is not all that unusual. The problem is that each table may contain over a million rows each. Instead of just JOINing the tables, appropriate restrictive clauses needed to be added to the WHERE clause of each table in order to reduce the total number of rows to be JOINed. This simple step can really boost the performance of a JOIN of two large tables. *Updated 11-1-2005*

*****

In the SELECT statement that creates your JOIN, **don't use an * (asterisk) to return all of the columns in both tables**. This is bad form for a couple of reasons. First, you should only return those columns you need, as the less data you return, the faster your query will run. It would be rare that you would need all of the columns in all of the tables you have joined. Second, you will be returning two of

each column used in your JOIN condition, which ends up returning way more data that you need, and hurting performance.

Take a look at these two queries:

```
USE Northwind
SELECT *
FROM Orders
INNER JOIN [Order Details]
ON Orders.OrderID = [Order Details].OrderID
```

and

```
USE Northwind
SELECT Orders.OrderID, Orders.OrderDate,
     [Order Details].UnitPrice, [Order Details].Quantity,
     [Order Details].Discount
FROM Orders
INNER JOIN [Order Details]
ON Orders.OrderID = [Order Details].OrderID
```

Both of these queries perform essentially the same function. The problem with the first one is that it returns not only too many columns (they aren't all needed by the application), but the OrderID column is returned twice, which doesn't provide any useful benefits. Both of these problems contribute to unnecessary server overhead, hurting performance. The moral of this story is never to use the * in your joins. [6.5, 7.0, 2000] *Updated 7-24-2006*

*****

While high index selectivity is generally an important factor that the Query Optimizer uses to determine whether or not to use an index, there is one special case where indexes with low selectivity can be useful speeding up SQL Server. This is in the case of indexes on foreign keys. **Whether an index on a foreign key has either high or low selectivity, an index on a foreign key can be used by the Query Optimizer to perform a merge join on the tables in question**. A merge join occurs when a row from each table is taken and then they are compared to see if they match the specified join criteria. If the tables being joined have the appropriate indexes (no matter the selectivity), a merge join can be performed, which is often much faster than a join to a table with a foreign key that does not have an index. [7.0, 2000] *Updated 7-24-2006*

*****

For very large joins, **consider placing the tables to be joined in separate physical files in the same filegroup**. This allows SQL Server to spawn a separate thread for each file being accessed, boosting performance. [6.5, 7.0, 2000] *Updated 7-24-2006*

*****

Don't use CROSS JOINS, unless this is the only way to accomplish your goal. What some inexperienced developers do is to join two tables using a CROSS JOIN, and then they use either the DISTINCT or the GROUP BY clauses to "clean up" the mess they have created. This, as you might imagine, can be a huge waste of SQL Server resources. [6.5, 7.0, 2000] *Updated 7-24-2006*

*****

If you have the choice of using a JOIN or a subquery to perform the same task, generally the JOIN (often an OUTER JOIN) is faster. But this is not always the case. For example, if the returned data is going to be small, or if there are no indexes on the joined columns, then a subquery may indeed be faster.

The only way to really know for sure is to try both methods and then look at their query plans. If this operation is run often, you should seriously consider writing the code both ways, and then select the most efficient code. [6.5, 7.0, 2000] *Updated 8-21-2006*

*****

We have a query that contains two subselects containing an aggregate function (SUM, Count, etc.) in the SELECT part. The query was performing sluggishly. We were able to isolate the problem down to the aggregate function in the subselect.

To rectify the problem, we reorganized the query so that there was still an aggregate function in the SELECT part, but replaced the subselects with a series of JOINS. The query executed much faster.

So, if this holds true — **developers, as a rule, should use JOINS in lieu of subselects when the subselect contains aggregate functions**. [7.0, 2000] *Tip provided by Silverscape Technologies, Inc (www.silverscape.net) Updated 8-21-2006*

*****

If you have a query with many joins, **one alternative to de-normalizing a table to boost performance is to use an Indexed View to pre-join the tables**. An Indexed View, which is only available from SQL Server 2000 Enterprise Edition, allows you to create a view that is actually a physical object that has its own clustered index. Whenever a base table of the Indexed View is updated, the Indexed View is also updated. As you can imagine, this can potentially reduce INSERT, UPDATE, and DELETE performance on the base tables. You will have to perform tests, comparing the pros and cons of performance in order to determine whether or not using an Indexed View to avoid joins in query is worth the extra performance cost caused by using them. [2000] *Updated 8-21-2006*

*****

If you have a query that uses a **LEFT OUTER JOIN**, check it carefully to be sure that is the type of join you really want to use. As you may know, a LEFT OUTER JOIN is used to create a result set that includes *all* of the rows from the left table specified in the clause, not just the ones in which the joined columns match. In addition, when a row in the left table has no matching rows in the right table, the result set row contains NULL values for all the selected columns coming from the right table. If this is what you want, then use this type of join.

The problem is that in the real world, a LEFT OUTER JOIN is rarely needed, and many developers use them by mistake. While you may end up with the data you want, you may also end up with more than the data you want, which contributes to unnecessary overhead and poor performance. Because of this, always closely examine why you are using a LEFT OUTER JOIN in your queries, and only use them if they are exactly what you need. Otherwise, use a JOIN that is more appropriate to your needs. [6.5, 7.0, 2000] *Updated 8-21-2006*

*****

If you are having difficulty tuning the performance of a poorly performing query that has one or more JOINs, check to see if they query plan created by the query optimizer is using a hash join. When the query optimizer is asked to join two tables that don't have appropriate indexes, it will often perform a hash join.

A hash join is resource intensive (especially CPU and I/O) and can slow the performance of your join. If the query in question is run often,

you should consider adding appropriate indexes. For example, if you are joining column1 in table1 to column5 in table2, then column1 in table1 and column5 in table2 need to have indexes.

Once indexes are added to the appropriate columns used in the joins in your query, the query optimizer will most likely be able to use these indexes, performing a nested-loop join instead of a hash join, and performance will improve. [7.0, 2000] *Updated 8-21-2006*