

420-TP-018-001

Database Indexing Strategies

**Technical paper - Not intended for formal review
or Government approval.**

August 1997

Prepared Under Contract NAS5-60000

RESPONSIBLE ENGINEER

David Marquette /s/	8/11/97
<hr/>	
David Marquette, Sybase Consultant	Date
EOSDIS Core System Project	

SUBMITTED BY

J. A. Feldman /s/	8/11/97
<hr/>	
J. A. Feldman, Data Engineering	Date
EOSDIS Core System Project	

Hughes Information Technology Systems
Upper Marlboro, Maryland

This page intentionally left blank.

Contents

Abstract

Why Index?

Choosing Indexes

Types of Indexes

Clustered Indexes

Nonclustered Indexes

Composite Indexes

Advantages of Composite Indexes.....	11
Good choices for composite indexes are:	11
Disadvantages of Composite Indexes	11
Poor choices are:	12

Other Indexing Guidelines

Performance Price for Updates	14
Key Size and Index Size	14
User Perceptions and Covered Queries.....	14

Examples

Examining a Single Query 17

Examining Two Queries with Different Indexing Requirements 17

Abstract

A major aspect of database tuning is reducing contention for system resources. As the number of users increases, applications contend for resources such as the data and procedure caches, spinlocks on system resources, and the CPU or CPUs. The probability of lock contention on data pages also increases. Indexing is a major means of tuning the databases in the ECS system. This primer identifies different indexing strategies as well as providing guidelines on how to choose the most appropriate index for a particular design.

Much of the information in this document has been extracted from “*Indexing for Performance*”, *Sybase SQL Server Performance and Tuning Guide*. The full guide can be found at <http://www.sybase.com>

This page intentionally left blank.

Why Index?

A primary goal of improving performance with indexes is avoiding table scans. In a table scan, every page of a table must be read from disk. Since the optimizer cannot know if you have one row, or several, that match the search arguments, it can't stop when it finds a matching row; it must read every row of the table.

If a query is searching for a unique value in a table that has 600 data pages, this requires 600 disk reads. If an index points to the data value, the query could be satisfied with two or three reads, a performance improvement of 200 percent to 300 percent. On a system with a 12-ms. disk, this is a difference of several seconds compared to less than a second. Even if this response time is acceptable for the query in question, heavy disk I/O by one query has a negative impact on overall throughput.

Table scans may occur:

- When there is no index on the search arguments for a query
- When there is an index, but the optimizer determines that the index is not useful
- When there are no search arguments

The underlying problems relating to indexing and poor performance are:

- No indexes are assigned to a table, so table scans are always used to retrieve data.
- [↑] An existing index is not selective enough for a particular query, so it is not used by the optimizer.
- The index does not support a critical range query, so table scans are required.
- Too many indexes are assigned to a table, so data modifications are slow.
- The index key is too large, so using the index generates high I/O.

This page intentionally left blank.

Choosing Indexes

Questions to ask when working with index selection are:

- What indexes are associated currently with a given table?
- What are the most important processes that make use of the table?
- What is the overall ratio of select operations to data modifications performed on the table?
- Has a clustered index been assigned to the table?
- Can the clustered index be replaced by a nonclustered index?
- Do any of the indexes cover one or more of the critical queries?
- Is a composite index required to enforce the uniqueness of a compound primary key?
- What indexes can be defined as unique?
- What are the major sorting requirements?
- Do the indexes support your joins, including those referenced by triggers and referential integrity constraints?
- Does indexing affect update types (direct vs. deferred)?
- What indexes are needed for cursor positioning?
- If dirty reads are used, are there unique indexes to support the scan?
- Should IDENTITY columns be added to tables and indexes to generate unique indexes? (Unique indexes are required for updatable cursors and dirty reads.)

This page intentionally left blank.

Types of Indexes

Three types of indexes are discussed: clustered (where the table data is physically stored in the order of the keys on the index.), non-clustered (where the storage order of data in the table is not related to index keys), and composite (where the index is comprised of multiple fields).

Index keys need to be differentiated from logical keys. Logical keys are part of the database design, defining the relationships between tables: primary keys, foreign keys, and common keys. When you optimize your queries by creating indexes, these ***logical keys may or may not be used as the physical keys for creating indexes***. You can create indexes on columns that are not logical keys, and you may have logical keys that are not used as index keys.

Tables that are read-only or read-mostly can be heavily indexed, as long as your database has enough space available. If there is little update activity, and high select activity, you should provide indexes for all of your frequent queries. Be sure to test the performance benefits of index covering.

This page intentionally left blank.

Clustered Indexes

These are general guidelines for clustered indexes:

- ↑ Most tables should have clustered indexes or use partitions. Without a clustered index or partitioning, all inserts and some updates go to the last page. In a high-transaction environment, the locking on the last page severely limits throughput.
- ↑ If your environment requires a lot of inserts, the clustered index key should not be placed on a monotonically increasing value such as an IDENTITY column. Choose a key that places inserts on "random" pages to minimize lock contention while remaining useful in many queries. Often, the primary key does not meet this guideline.
- ↑ Clustered indexes provide very good performance when the key matches the search argument in range queries, such as: where colvalue >= 5 and colvalue < 10
- ↑ Other good candidates for clustered index keys are columns used in order by and group by clauses and in joins.
- Columns that are not frequently changed

If there are multiple candidates, choose the most commonly needed physical order as a first choice. As a second choice, look for range queries. During performance testing, check for "hot spots," places where data modification activity encounters blocking due to locks on data or index pages.

This page intentionally left blank.

Nonclustered Indexes

When choosing columns for nonclustered indexes, consider all the uses that were not satisfied by your clustered index choice. In addition, look at columns that can provide performance gains through index covering.

The one exception is noncovered range queries, which work well with clustered indexes, but may or may not be supported by nonclustered indexes, depending on the size of the range.

Consider composite indexes to cover critical queries and support less frequent queries:

- The most critical queries should be able to perform point queries and matching scans.
- ↑ Other queries should be able to perform nonmatching scans using the index, avoiding table scans.

When you consider nonclustered indexes, you must weigh the improvement in retrieval time against the increase in data modification time.

In addition, you need to consider these questions:

- How much space will the indexes use?
- How volatile is the candidate column?
- How selective are the index keys? Would a scan be better?
- Is there a lot of duplication?

Because of overhead, add nonclustered indexes only when your testing shows that they are helpful. Candidates include:

- Columns used for aggregates
- Columns used for joins, *order by*, *group by*

This page intentionally left blank.

Composite Indexes

If your needs analysis shows that more than one column would make a good candidate for a clustered index key, you may be able to provide clustered-like access with a composite index that covers a particular query or set of queries. These include:

- Range queries
- Vector (grouped) aggregates, if both the grouped and grouping columns are included
- Queries that return a high number of duplicates
- Queries that include *order by*
- Queries that table scan, but use a small subset of the columns on the table

Choose the right ordering of the composite index so that most queries form a prefix subset.

Advantages of Composite Indexes

Composite indexes have these advantages:

- A dense composite index provides many opportunities for index covering.
- ↑ A composite index with qualifications on each of the keys will probably return fewer records than a query on any single attribute.
- A composite index is a good way to enforce uniqueness of multiple attributes.

Good choices for composite indexes are:

- Lookup tables
- Columns frequently accessed together

Disadvantages of Composite Indexes

The disadvantages of composite indexes are:

- ↑ Composite indexes tend to have large entries. This means fewer index entries per index page and more index pages to read.
- ↑ An update to any attribute of a composite index causes the index to be modified. The columns you choose should not be those that are updated often.

Poor choices are:

- Indexes that are too wide because of long keys
- ↑ Composite indexes where only the second or third portion, or an even later portion, is used in the where clause

Other Indexing Guidelines

Here are some other considerations for choosing indexes:

- ↑ If an index key is unique, be sure to define the index as unique. Then, the optimizer knows immediately that only one row will be returned for a search argument or a join on the key.
- ↑ If your database design uses referential integrity (the *references* or *foreign key...references* keywords in the create table statement), the referenced columns must have a unique index. However, SQL Server does not automatically create an index on the referencing column. If your application updates and deletes primary keys, you may want to create an index on the referencing column so that these lookups do not perform a table scan.
- If your applications use cursors, see "Index Use and Requirements for Cursors" .
- ↑ If you are creating an index on a table where there will be a lot of insert activity, use fillfactor to temporarily:
 - Minimize page splits
 - Improve concurrency and minimize deadlocking
- ↑ If you are creating an index on a read-only table, use a fillfactor of 100 to make the table or index as compact as possible.
- ↑ Keep the size of the key as small as possible. Your index trees remain flatter, accelerating tree traversals. More rows fit on a page, speeding up leaf-level scans of nonclustered index pages. Also, more data fits on the distribution page for your index, increasing the accuracy of index statistics.
- Use small datatypes whenever it fits your design.
 1. Numerics compare faster than strings internally.
 2. Variable-length character and binary types require more row overhead than fixed-length types, so if there is little difference between the average length of a column and the defined length, use fixed length. Character and binary types that accept null values are by definition variable length.
 3. Whenever possible, use fixed-length, non-null types for short columns that will be used as index keys.

- ↑ Keep datatypes of the join columns in different tables compatible. If SQL Server has to convert a datatype on one side of a join, it may not use an index for that table. See *"Datatype Mismatches and Joins"* for more information.

Performance Price for Updates

Also, remember that with each insert, all nonclustered indexes have to be updated, so there is a performance price to pay. The leaf level has one entry per row, so you will have to change that row with every insert.

All nonclustered indexes need to be updated:

- For each insert into the table.
- For each delete from the table.
- ↑ For any update to the table that changes any part of an index's key, or that deletes a row from one page and inserts it on another page.
- ↑ For almost every update to the clustered index key. Usually, such an update means that the row moves to a different page.
- For every data page split.

Key Size and Index Size

Small index entries yield small indexes, producing less index I/O to execute queries. Longer keys produce fewer entries per page, so an index requires more pages at each level, and in some cases, additional index levels.

The disadvantage of having a covering, nonclustered index, particularly if the index entry is wide, is that there is a larger index to traverse, and updates are more costly.

User Perceptions and Covered Queries

Covered queries can provide excellent response time for specific queries, while sometimes confusing users by providing much slower response time for very similar-looking queries. With the composite nonclustered index on `au_lname`, `au_fname`, `au_id`, this query runs very fast:

```
select au_id
  from authors
 where au_fname = "Eliot"
       and au_lname = "Wilk"
```

This covered point query needs to perform only three reads to find the value on the leaf level row in the nonclustered index of a 5000-row table.

Users might understand why this similar-looking query (using the same index) does not perform quite as well:

```
select au_fname, au_lname
      from authors
     where au_id = "A1714224678"
```

However, this query does not include the leading column of the index, so it has to scan the entire leaf level of the index, about 95 reads. Adding a column to the select list, which may seem like a minor change to users, makes the performance even worse:

```
select au_fname, au_lname, phone
      from authors
     where au_id = "A1714224678"
```

This query performs a table scan, reading 222 pages. In this case, the performance is noticeably worse. But the optimizer has no way of knowing the number of duplicates in the third column (au_id) of a nonclustered index: it could match a single row, or it could match one-half of the rows in a table. A composite index can be used only when it covers the query or when the first column appears in the where clause.

Adding an unindexed column to a query that includes the leading column of the composite index adds only a single page read to this query, when it must read the data page to find the phone number:

```
select au_id, phone
      from authors
     where au_fname = "Eliot"
        and au_lname = "Wilk"
```

This page intentionally left blank.

Examples

Examining a Single Query

Assume that you need to improve performance of the following query:

```
select title
from titles
where price between $20.00 and $30.00
```

You know the size of the table in rows and pages, the number of rows per page, and the number of rows that the query returns:

- 1,000,000 rows (books)
- 190,000 are priced between \$20 and \$30
- 10 rows per page; pages 75 percent full; approximately 140,000 pages

With no index, the query would scan all 140,000 pages.

With a clustered index on price, the query would find the first \$20 book and begin reading sequentially until it gets to the last \$30 book. With 10 data rows per page, and 190,000 matching rows, the query would read 19,000 pages plus 3 or 4 index pages.

With a nonclustered index on price and random distribution of price values, using the index to find the rows for this query would require 190,000 logical page reads plus about 19 percent of the leaf level of index, adding about 1500 pages. Since a table scan requires only 140,000 pages, the nonclustered index would probably not be used.

Another choice is a nonclustered index on price, title . The query can perform a matching index scan, finding the first page with a price of \$20 via index pointers, and then scanning forward on the leaf level until it finds a price more than \$30. This index requires about 35,700 leaf pages, so to scan the matching leaf pages requires about 6,800 reads.

For this query, the nonclustered index on price, title is best.

Examining Two Queries with Different Indexing Requirements

This query also needs to run against the same table:

```
select price
from titles
where title = "Looking at Leeks"
```

You know that there are very few duplicate titles, so this query returns only one or two rows. Here are four possible indexing strategies, identified by the numbers used in subsequent discussion:

1. Nonclustered index on titles(title) ; clustered index on titles(price)
2. Clustered index on titles(title) ; nonclustered index on titles(price)
3. Nonclustered index on titles(title, price)
4. Nonclustered index on titles(price, title)

Table 6-6 shows some estimates of index sizes and I/O for the range query on price and the point query on title . The estimates for the numbers of index and data pages were generated using a fillfactor of 75 percent with sp_estspace :

sp_estspace titles, 1000000, 75

The values were rounded for easier comparison.

Comparing index strategies for two queries

	Index Choices	Index Pages	Range Query on price	Point Query on title
1	Nonclustered on title; Clustered on price	36,800 650	Clustered index, about 26,600 pages (140,000 .19) With 16K I/O: 3,125 I/Os	Nonclustered index, 6 I/Os
2	Clustered on title; Nonclustered on price	3,770 6,076	Table scan, 140,000 pages With 16K I/O: 17,500 I/Os	Clustered index, 6 I/Os
3	Nonclustered on title , price	36,835	Nonmatching index scan, about 35,700 pages With 16K I/O: 4,500 I/Os	Nonclustered index, 5 I/Os
4	Nonclustered on price , title	36,835	Matching index scan, about 6,800 pages (35,700 .19) With 16K I/O: 850 I/Os	Nonmatching index scan, about 35,700 pages With 16K I/O: 4,500 I/Os

Examining these figures shows that:

- For the range query on price , indexing choice 4 is best, and choices 1 and 3 are acceptable with 16K I/O.
- For the point query on titles , indexing choices 1, 2, and 3 are excellent.

The best indexing strategy for the combination of these two queries is to use two indexes:

- The nonclustered index in price, title , for range queries on price
- The clustered index on title , since it requires much less space

Other information could help determine which indexing strategy to use to support multiple queries:

- What is the frequency of each query? How many times per day or per hour is the query run?
- What are the response time requirements? Is one of them especially time critical?
- What are the response time requirements for updates? Does creating more than one index slow updates?
- Is the range of values typical? Is a wider or narrower range of prices, such as \$20 to \$50 often used? How do these ranges affect index choice?
- Is there a large data cache? Are these queries critical enough to provide a 35,000-page cache for the nonclustered composite indexes in index choice 3 or 4? Binding this index to its own cache would provide very fast performance.

This page intentionally left blank.