Columnstore Indices and Batch Processing

SQL Server 2012 has a new method of storing nonclustered indices. In addition to regular row storage, SQL Server 2012 can store index data column by column, in what’s called a columnstore index. Columnstore indices can speed up data warehousing queries by a large factor, from 10 to even 100 times.

A columnstore index is just another nonclustered index on a table. Query Optimizer considers using it during the query optimization phase just as it does any other index. All you have to do to take advantage of this feature is to create a columnstore index on a table.

A columnstore index is often compressed even further than any data compression type can compress the row storage—including Page and Unicode compression. When a query references a single column that is a part of a columnstore index, then SQL Server fetches only that column from disk; it doesn’t fetch entire rows as with row storage. This also reduces disk IO and memory cache consumption. Columnstore indices use their own compression algorithm; you cannot use Row or Page compression on a columnstore index.

On the other hand, SQL Server has to return rows. Therefore, rows must be reconstructed when you execute a query. This row reconstruction takes some time and uses some CPU and memory resources. Very selective queries that touch only a few rows might not benefit from columnstore indices.

Columnstore indexes accelerate data warehouse queries, not OLTP workloads. Because of the row reconstruction issues, tables containing a columnstore index become read only. If you want to update a table with a columnstore index, you must first drop the columnstore index. If you use table partitioning, you can switch a partition to a different table without a columnstore index, update the data there, create a columnstore index on that table (which has a smaller subset of the data), and then switch the new table data back to a partition of the original table. You will learn how to implement table partitioning with columnstore indices in Lesson 3 of this chapter.

There are three new catalog views you can use to gather information about columnstore indices:

* sys.column\_store\_index\_stats
* sys.column\_store\_segments
* sys.column\_store\_dictionaries.

The columnstore index is divided into units called *segments*. Segments are stored as large objects (LOB), and consist of multiple pages. Segments are the unit of transfer from disk to memory. Each segment has metadata that stores the minimum and maximum value of each column for that segment. This enables early segment elimination in the storage engine. SQL Server loads only those segments requested by a query into memory.

SQL Server 2012 includes another important improvement for query processing. In batch mode processing, SQL Server processes data in batches rather than processing one row at a time. In SQL Server 2012, a batch represents roughly 1000 rows of data. Each column within a batch is stored as a vector in a separate memory area, meaning that batch mode processing is vector-based. Batch mode processing interrupts a processor with metadata only once per batch rather than once per row, as in row mode processing, which lowers the CPU burden substantially.

You can find out whether SQL Server used batch mode processing by analyzing the query execution plan. There are two new operator properties in the Actual Execution Plan: EstimatedExecutionMode and ActualExecutionMode. Batch mode processing is available for a limited list of operators only:

* Filter
* Project
* Scan
* Local hash (partial) aggregation
* Hash inner join
* (Batch) hash table build.

Batch mode processing is particularly useful for data warehousing queries, combined with bitmap filtered hash join in a star join pattern.

Columnstore indices have quite a few limitations:

* Columnstore indices can be nonclustered only.
* You can have only one columnstore index per table.
* If your table is partitioned, the columnstore index must be partition-aligned.
* Columnstore indices are not allowed on indexed views.
* A columnstore index can’t be a filtered index.
* There are additional data type limitations for columnstore indices.

You should use a columnstore index on your fact tables, putting all columns of a fact table in a columnstore index. In addition to fact tables, very large dimensions could benefit from columnstore indices as well. Do not use columnstore indices for small dimensions. Other best practices for columnstore indices include:

* Use columnstore indices for:
  + Read-mostly workloads
  + Updates that append new data
  + Workflows that permit partitioning or index drop/rebuild
  + Queries often scan and aggregate lots of data.
* Don’t use columnstore indices when:
  + You update the data frequently
  + Partition switching or rebuilding index doesn’t fit your workflow
  + Your workload includes mostly small lookup queries.