Using PPIs to improve performance

Tips on how partition elimination can boost your query workload.  

by Paul Sinclair

Partitioned primary indexes (PPIs), introduced in Teradata Database V2R5 and extended in Teradata Database 12.0 to multiple levels—referred to as multilevel partitioned primary indexes (MLPPIs)—provide an opportunity to significantly improve the performance of certain queries and high-volume insert, update and delete operations.

The performance gain depends on the number of partitions and the specific query being measured. In the best case, the query conditions allow every partition but one to be eliminated for each partitioning expression. With thousands of combined partitions, the I/O for such a query can be reduced to less than 1% of the I/O that it takes to run the same query against the table with a non-partitioned primary index (NPPI). Even with only tens or hundreds of partitions, huge improvements can be made in some queries.

Significant performance benefits can be achieved, therefore, if the data demographics and queries in the workload lead to partition elimination. Careful selection of the partitioning expressions for a table or set of tables, including the choice of partitioning columns, number of partitions per level and number of partitioning expressions, is required to be successful. In some cases, modifying the tables can further improve partitioning usefulness.

MLPPI

The following shows a CREATE TABLE statement for a table with an MLPPI:

```sql
CREATE TABLE Sales
    (storeid INTEGER NOT NULL,
     productid INTEGER NOT NULL,
     salesdate DATE FORMAT 'yyyy-mm-dd' NOT NULL,
     totalrevenue DECIMAL(13,2),
     totalsold INTEGER,
     note VARCHAR(256))
UNIQUE PRIMARY INDEX (storeid, productid, salesdate)
PARTITION BY (
    RANGE_N(salesdate BETWEEN DATE '2002-01-01' AND DATE '2008-12-31' EACH INTERVAL '1' YEAR),
    RANGE_N(storeid BETWEEN 1 AND 300 EACH 100),
    RANGE_N(productid BETWEEN 1 AND 400 EACH 100));
```

This table is first partitioned by year based on salesdate. Next, within each year the data will be partitioned by storeid in groups of 100. Finally, within each cluster of years and storeid group, the data will be partitioned by productid in groups of 100. With seven years, three groups of storeids and four groups for productids, this partitioning defines 84 (7*3*4) combined partitions.

Without the partitioning, secondary indexes are often used to provide access to the table for the various dimensions so that performance is improved over a full-table scan. Partitioning avoids the overhead of storing and maintaining such secondary indexes and the query cost of going back to the base table after retrieving the rowids from the index. Partitioning puts rows with similar values in clusters so that access to the table for the various dimensions can be done by eliminating all but the partitions for the dimension values of interest. Only the data blocks associated with the non-eliminated partitions need to be read. This results in fewer data block I/Os needed to retrieve the qualifying rows.

In the alternative approach of rowid retrieval using a secondary index, similar
rows are not clustered; consequently, a data block may contain only one or a few qualifying rows. This creates the need for many more data block I/Os to retrieve the qualifying rows. Also, partitioning supports range-based access, which is often difficult to effectively achieve with secondary indexes.

**Choices**

Some choices of partitioning may have trade-offs. If the partitioning columns are not part of the primary index (PI), then PI access, join and aggregation queries may be degraded, while partition elimination may improve other queries. Reducing the number of partitions or adding a secondary index on the columns of the PI can minimize the negative performance impact to such queries. Note that decreasing the number of partitions reduces the benefits of partitioning, and secondary indexes have their storage and maintenance overhead. The trade-offs must be analyzed for the query workload to determine whether the benefits offset any degradation. Typical queries in the workload can be explained and their performance measured in order to help in this analysis. The following are some tips on partitioning:

1. **Use the DATE data type**, if possible, for a date-based partitioning expression. A date-based partitioning expression is often a good choice for at least one level of partitioning. This will allow the Teradata Database to better recognize partition elimination opportunities.

2. **Keep the partitioning expression simple**. A RANGE_N partitioning expression usually works the best for partition elimination. With multi-level partitioning, while one level usually does RANGE_N date-based partitioning, other levels may use CASE_N partitioning.

3. **Add query conditions on the partitioning columns**, where possible, to improve partition elimination opportunities.

4. If queries join on the PI but the PI doesn’t include the partitioning column, **consider propagating the partitioning column value** to the other table and modifying the query to also join on the partitioning column and the column propagated to the other table.

5. **Make sure the selected partitioning clusters the data** so that a combined partition contains either a large number of rows (resulting in multiple data blocks per AMP) or contains no rows. This is to ensure that when a combined partition is read, a majority of rows read from the data blocks qualify. (The first and last data block may contain rows from other non-qualifying partitions.) Note that an empty partition does not take any space. As a rule of thumb, include at least 10 data blocks per combined partition per AMP. Since, on average, half of the first data block and half of the last data block will be rows for other partitions, then 90% of the rows read will be from the qualifying partition. Simple queries can be run on the data to determine how well the data clusters for a candidate set of partitioning expressions.

6. If you follow the previous tip, the order of partitioning expression shouldn’t matter too much. If all else is constant, **place the partitioning expressions in ascending order** based on the number of partitions they each define. However, you may want to put your date-based partitioning expression first.

7. **Use tools** such as Teradata Database Query Log and Index Wizard to better understand your workload, identify partitioning opportunities and verify the success of your partitioning.

8. **Collect statistics** on the system-derived column PARTITION and the usual recommended indexes and columns. Collecting on the partitioning columns themselves is usually also a good idea, but statistics on PARTITION may be enough for good plans. Check EXPLAINs and measure performance to make sure.

A partitioning expression is good only if queries take advantage of it—in other words, if partition elimination occurs—and all of the following work well together, based on validated trade-off choices: partitioning expressions, specific and overall queries, performance, access method, join strategy, partition elimination, data maintenance, altering the partitioning and backup/restore.

The best choice, if any, of candidate partitioning expressions depends on the mix of anticipated queries and the capability of the Optimizer to detect partition elimination. The extended logical data model can serve as the starting point for making the decision, but testing different scenarios is often still required. Though it may take some experimentation with various choices of partitioning, the potential performance improvements for queries and other operations are usually worth the effort.

---

Paul Sinclair, a software architect and Teradata Certified Master, has been with the Teradata Research and Development Division for more than 19 years.

---

Online

For more on PPI, visit Teradata.com and download the white paper “Single-level and Multilevel Partitioned Primary Indexes.”