Partitioning Event Set/Loss Datasets

Approach - Solution A

Selection within available partition schemes on a given database platform should consider the intent, data type and query needs against the tables to provide the most applicable option. Observation of the event set /loss footprint tables can assist in inferring the base application or reporting needs. To complete this exercise in a live environment though, some review of application logic, explain plans and benchmarking are required for design acceptance. In most cases basic row distribution can provide impact however, even with a inefficient design. These short-term gains based in data division, ordered rowsets or tuned sql can provide false positives to design acceptance.

With respect to to the event_set and event_loss_footprint tables, the overall column count is low. Considering a logical rowlength of 16 and 24 bytes respectively and that due to hash or bitwise storage methods that this could physically be ¼ of that byte length on disk (or less). As most current server architectures include extensive RAM capacity one must also consider what resources are available for buffer cache in addition to the totality of table pages. With these details and an arbitrary threshold of 100GB we can discern that these tables would have to be > 4 billion rows before partitioning begins to have relevance.

Event_loss_footprint could be partitioned using one of the following schemes: range partition on event_id or range partition on t031_id. I would not partition on the primary key: event_loss_footprint.id as this is a sequential row identifier most likely that will extend as data growth happens. Introduction of the necessity for partition maintenance is normally a non-starter in my personal designs but there are applications where this makes sense, typically where archiving is assumed and frequent. In this case event_id is a nice bounded range (0-10k on t00001 and 5-20k on t00001) and doesn't create a large number of partitions.

Event_set could be partitioned using the following schemes: range partition on event_id, range_partition on t010_id or range on event_day. I selected to partition on event_day as I've found that blindly using known timeseries fields can simply provide the row distribution sought with partitioning in cases of limited decisions. Partitioning on the PK of event_id is likely the better scenario in a larger environment though as the performance gain through partition-aware joins on event_id between the two tables would be substantial.

Indexing - Solution C

Utilizing SQL Server 2014 meant we have the typical clustered, non-clustered index options but also the new clustered columnstore index type. This type is most interesting from a performance standpoint as it has a lot of relevance to large datasets or data warehousing. The data density and lowered seek time could mean radical performance gains but table data refreshes could become less simple. In the end I feel that unless the report needs push for some filtered index or a unconventional column concatenation (unlikely with four columns) in a non-partitioned index than typical clustered, non-clustered index types will suffice. In most implementations the clustered index can provide some benefit but its page-change overhead (updates,inserts,deletes) must be weighed against the maintenance and overall performance gains.

Monitoring Design Decisions - Solution F

Explain plans provide views into sql execution from the optimizer and/or sql parser that enable a developer or dba to review the likely resource cost of the execution. Cost can be in terms of a number of resources but typically its i/o, cpu and memory. Its important to use the cost metrics outside of glaring sql issues and my approach is to “trust but verify” statistics and plans provided by any rdbms’ optimizer. By review and actual execution with timed statistics its possible to determine that in some cases a full-table scan can indeed be faster than a hash join due to invocation, host and environmental circumstances. In the end plan generation and comprehension is as important to the db engine as a dba and successful support of data platforms often require preproduction and live system review of executed sql. Ongoing tuning is often a progressive need as well as data skew occurs and this can be another impetus to logic review.