Partitioned B-trees – a user's guide

Goetz Graefe
Microsoft Corporation
Redmond, WA 98052-6399

Abstract: A recent article introduced partitioned B-trees, in which partitions are defined not in the catalogs but by distinct values in an artificial leading key column. As there usually is only a single value in this column, there usually is only a single partition, and queries and updates perform just like in traditional B-tree indexes. By temporarily permitting multiple values, at the expense of reduced query performance, interesting database usage scenarios become possible, in particular for bulk insert (database load). The present paper guides database administrators to exploiting partitioned B-trees even if they are not implemented by their DBMS vendor.

1 Introduction

The essence of partitioned B-tree indexes [G 03] is to maintain partitions within a single B-tree, by means of an artificial leading key column, and to reorganize and optimize such a B-tree online using, effectively, the merge step well known from external merge sort. This key column should be an integer of 2 or 4 bytes. By default, the same single value appears in all records in a B-tree, and the techniques proposed here rely on temporarily permitting and exploiting multiple alternative values. If a table or view in a relational database has multiple indexes, each index has its own artificial leading key column. The values in these columns are not coordinated or propagated among the indexes. In other words, each artificial leading key column is internal to a single B-tree, such that each B-tree can be reorganized and optimized independently of all others.

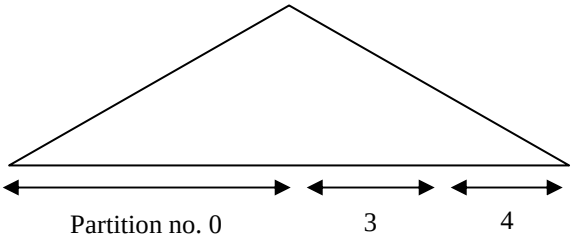

Figure 1. B-tree with partitions

The leading artificial key column effectively defines partitions within a single B-tree. Each existing distinct value implicitly defines a partition, and partitions appear and vanish automatically as the first or last record with a specific value are inserted and deleted. Partitioned B-trees differ from traditional horizontal partitioning using a separate B-tree for each partition in an important way. Most advantages of partitioned B-trees depend on partitions (or distinct values in the leading artificial key column) being created and removed very dynamically. In a traditional implementation of partitioning, each creation or removal of a partition is a change of the table's schema and catalog entries, which re-

quires locks on the table's schema or catalog entries and thus excludes concurrent or long-running user accesses to the table, as well as forcing recompilation of cached query and update plans. If partitions are created and removed as easily as inserting and deleting rows, smooth continuous operation is relatively easy to achieve.

Queries must probe each partition; while multiple partitions exist, query performance is reduced. Suitable algorithms for searching indexes with multiple partitions have been described elsewhere [LJB 95] and are not considered here in detail. Database management systems vary in their ability to generate and execute such plans; experimentation is required to ensure acceptable query plans are chosen when probing multiple partitions. For sorted scans, the ideal plan merges multiple ordered scans, one per partition. If indexes usually imply statistics and histograms, selectivity estimation may be improved by separate statistics on the user's column only, i.e., without the artificial leading key column.

While partitioned B-trees are very useful for sorting, index creation, and bulk insertion [G 03], the present paper focuses on bulk insertion. Adding a large amount of data to a large, fully indexed data warehouse so far has created a dilemma between dropping and rebuilding all indexes or updating all indexes one record at a time, implying random insertions, poor performance, a large log volume, and a large incremental backup. Partitioned B-trees resolve this dilemma in most cases. Multiple indexes may exist and are maintained correctly, without the need for expensive random insertions into each B-tree. Note that partitioned B-trees offer these advantages without special new data structures, which means that B-trees as implemented and available today can be adapted and used as partitioned B-trees. The present paper provides guidance on how to do so.

2 Comparison of bulk insertion strategies

As an example, assume a table with a clustered index and no other indexes. Assume also that this table already contains 100M rows on 1M pages, and that another 1M rows must be added. This 1% could represent one eight-hour shift within a month, one day within a quarter, or one week within a two-year period – thus, this is a typical scenario in a data warehouse. Finally, assume that the clustered index is not sorted on a time attribute, i.e., integrating the new data into the existing clustered index will require a lot of key searches and random I/Os. If the operation modifies (reads and writes) 1M pages, a database server performing 1,000 I/Os per second will require 2,000 seconds or about ½ hour of dedicated server time. Note that this strategy will require either a large number of key locks or it will lock the entire table for the entire time, completely preventing concurrent updates and queries.

If the database server employs one of the optimized strategies that pre-sort the change set before applying it to an index, we may presume that I/O operations can often move multiple pages at once and will thus be 2-4 times faster than random I/O. Thus, the time to apply the bulk insert is reduced to about ¼ hour. Again, this strategy is likely to lock the entire table for practically the entire time.

In fact, it might be faster to drop the pre-existing index, add the new data, and then build an entirely new index. Adding 1M rows or about 10K pages to a heap is very fast, requiring about 10 seconds. Presuming index creation employs an external merge sort with a single merge step, index creation will require I/O for about 4M pages. Presuming that

sorting uses large I/Os with bandwidth 4 times higher than random I/Os, this strategy will require also about ¼ hour. Even if the update plan and index construction do not lock the entire table and clustered index, concurrent queries and updates will perform extremely poorly due to the missing index.

Now presume that the clustered index has an artificial leading key column, and that the value in this column for all pre-existing rows is 0. The fastest way to insert 1M rows is to insert all of them with a new value for this column, say 1. If so, the pre-existing rows and the new rows will be in separate partitions within the clustered index. All new rows can be appended to the pre-existing B-trees, which permits not only packing new records very densely but also permits optimizations for the insertion logic (no need for a root-to-leaf search for each record), for write I/O (large writes), and for logging (log entire pages rather than single records). If 1M new rows require 10K new pages, this insertion can complete in 10 seconds or less.

The insert set ought to be sorted because appending to a B-tree is faster than inserting into a B-tree. For truly large inserts, the sort operation might require external sorting and thus I/O of its own. In this case, it might make sense to append multiple smaller partitions, each one requiring only an in-memory sort. The reorganization that merges the newly appended partitions into the pre-existing main partition is equally efficient for any small number of new partitions.

Once all records have been inserted into the database, the new partitions ought to be merged into the pre-existing main partition. The important observations are that (1) the pre-existing keys and pages do not need to be locked and are available for concurrent queries and updates at all times, (2) the index is never dropped and concurrent operations proceed with normal performance, (3) the new data are available for queries and updates immediately after the insertion is complete, and (4) partitions can be merged in many small transactions, each transaction merging only a small range of keys from the new partition into the main partition. Because these transactions are small and fast, they are unlikely to interfere very much with concurrent transaction processing.

3 Example SQL commands

The following code shows some SQL commands in order to illustrate the suggestions above. The SQL code below includes a query as it should be used in all application queries – better yet, the table should be queried through a view that adds this predicate to all queries and it should be updated to a view that enforces the default value for the artificial leading key column in all insertions.

```
Create table tbl (a varchar(20), b float, c datetime, …) -- initial, empty table
Alter table tbl add column x int not null check (value >= 0) default 0
Insert into tbl (x, a, b, c) select 0, … from … -- initial 100M rows, partition 0
Create clustered index clu on tbl (x, a) -- index with artificial leading key column
Insert into tbl (x, a, b, c) select 1, … from … -- add 1M rows, partition 1
Insert into tbl (x, a, b, c) select 2, … from … -- add another 1M rows
…
Insert into tbl (x, a, b, c) select 5, … from … -- add the 5th 1M rows
Select … from tbl where x in (0, 1, 2, 3, 4, 5) and … -- application query or view
```

```
While (exists (select * from tbl where x in (1, 2, 3, 4, 5)) -- reorganization
Begin -- reorganize about 100 rows at a time
    If (exists (select * from tbl where x = 1))
        Update tbl set x = 0 where x in (1, 2, 3, 4, 5) and a <= (select max (a) from
            (select top 20 a from tbl where x = 1 order by a) as t)
    …
    If (exists (select * from tbl where x = 5))
        Update tbl set x = 0 where x in (1, 2, 3, 4, 5) and a <= (select max (a) from
            (select top 20 a from tbl where x = 5 order by a) as t)
End
```

The set of existing partitions (or values in the artificial leading key column) could be administered using a small auxiliary table and a referential constraint, which permits leaving the view definition in place as partitions appear and vanish.

If there is more than one index, e.g., non-clustered indexes, there ought to be a separate artificial key column for each of them. When inserting data, appropriate new values are assigned to all of them, ensuring fast append logic for all B-tree indexes.

4 Summary and conclusions

In summary, an artificial leading key column in each B-tree index enables partitioned B-trees, which permit not only fast bulk insertion but also fast memory-adaptive external merge sort as well as improved index creation [G 03]. If not implemented by the database management system vendor, artificial leading key columns can be declared explicitly by database administrators adapting the sample code provided above, which can also be adapted for bulk deletion from tables with one or multiple indexes.

The important value for bulk insertion is that the actual insertion step becomes very fast, and that pre-existing indexes are available for queries immediately after the insertion command completes. While multiple partitions exist, query performance is reduced, and online reorganization might require more log space than traditional import methods. These disadvantages must be judged against the fast insertion times; in many practical cases, partitioned B-trees based on artificial leading key columns and online reorganization can be more attractive than any of the alternatives. If this brief paper helps database administrators save valuable time developing import strategies using partitioned B-trees, it has achieved its purpose.

References

[G 03] Goetz Graefe: Sorting and Indexing with Partitioned B-trees. Conf. Innovative Data Systems Research, Asilomar, CA, Jan. 2003.

[LJB 95] Harry Leslie, Rohit Jain, Dave Birdsall, Hedieh Yaghmai: Efficient Search of Multi-Dimensional B-Trees. VLDB Conf. 1995: 710-719.