7/15/2015

PROGRESS

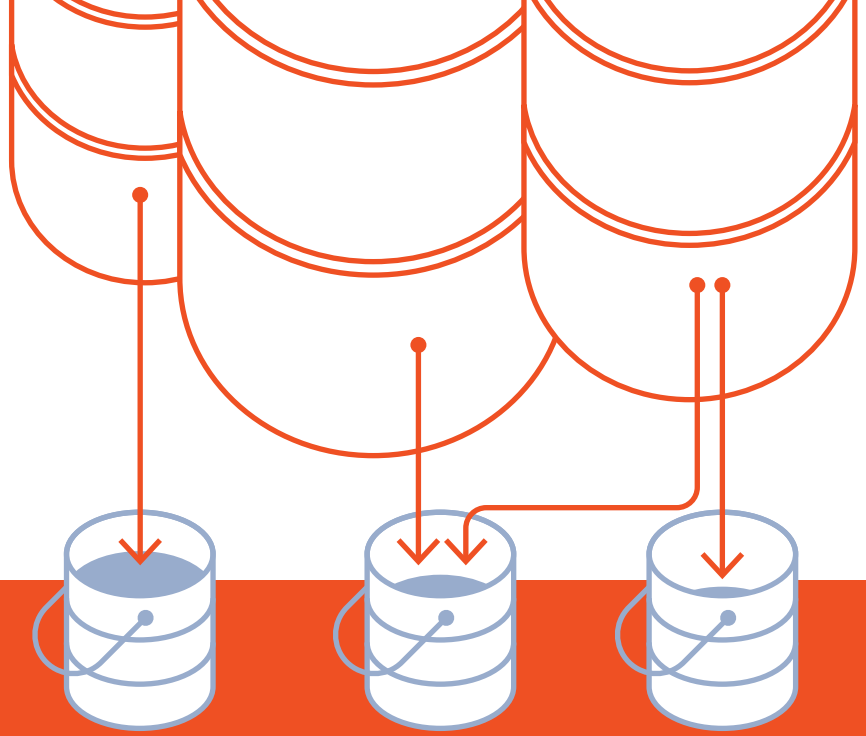


Table Partitioning: Improve Performance through Increased Concurrency

Richard Banville, Dapeng Wu

Table of Contents

1. Introduction 3

2. Table Partitioning Fundamentals 4

3. Table Partitioning Strategies 5

4. Performance Study 7

System Configurations 7

Test Results 9

Results Analysis 10

5. Conclusion 14

1. Introduction

As one of the key features released in Progress® OpenEdge® 11.4, Table Partitioning enables customer data and indexes to be horizontally split into smaller storage units (partitions), while maintaining logical integrity of the data. The benefits of Table Partitioning include higher data availability, ease of management and improved performance. Availability and management benefits are due to the ability to perform maintenance operations on an individual partition, rather than the entire table or index. Improved performance can be less obvious to some OpenEdge developers and database administrators, and is dependent on the partitioning strategy and data access patterns of the deployment.

The purpose of this white paper is to discuss partitioning strategies to achieve higher performance through increased concurrency inside the database engine. The materials used in this paper are based on the performance testing work performed by Richard Banville, which he reported at the 2015 North America PUG Challenge. The process and testing was reviewed and repeated by Dapeng Wu.

The points discussed in this paper assume basic knowledge of OpenEdge database internals.

2. Table Partitioning Fundamentals

Progress OpenEdge supports three types of partitioning:

1. List Partitioning: Data is split into partitions according to discrete list values. For example, table "Order" can be partitioned on the "Region" column into "Western Region," "Northern Region" and "Southern Region"

2. Range Partitioning: Data is split into partitions based on certain range values. For example, table "Order" can be partitioned on the "Date" column into "1/1/2011–12/31/2011" and "1/1/2012–12/31/2012."

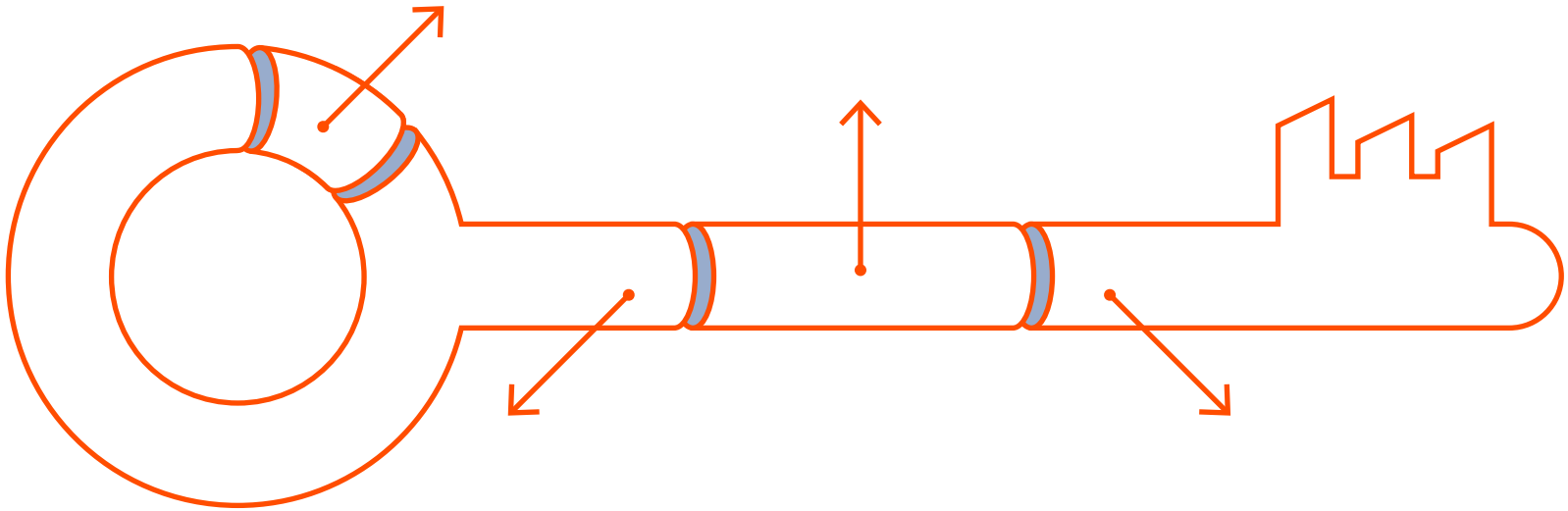
3. Sub-Partitioning. This is a combination of List Partitioning and Range Partitioning. For example, table "Order" can be partitioned on the "Region" and "Date" columns, into partitions "Western Region, 1/1/2011–12/31/2011," "Western Region, 1/1/2012–12/31/2012" and "Northern Region, 1/1/2011 – 12/31/2011."
Up to 15 levels of sub-partitioning can be constructed to meet different requirements.

3. Table Partitioning Strategies

For the database administrator, the first and probably the biggest challenge to implementing Table Partitioning is to identify proper partition key(s). Partition keys are the table columns used to determine in which partition a row should be stored. Choosing the right partitioning keys is important for two reasons:

1. It can improve database performance and manageability.

2. A wrong choice can lead to disappointing results. Changing partitioning keys will require the whole table to be recreated and indexes rebuilt, which can be very expensive, depending on the data size.



The following are guidelines and recommendations to consider when choosing partitioning keys:

1. Selecting a partitioning key and storage layout that will result in performance wins via table partitioning requires understanding the access patterns of the applications using the database tables. For instance, for an application inserting stock trades into a large table for many different trade symbols, and reports are frequently done based on queries for a particular stock symbol, partitioning by stock symbol and spreading the partitions across many storage areas could result in significant performance benefits. However, if 90 percent of the queries and trades are for a single symbol, it is likely there will not be a significant performance benefit with this partition strategy. So, it is important that to analyze the access patterns as much as possible. For DBAs without a lot visibility into the access patterns of the application, the proposed partition strategy should be discussed with those who do, for example, the developers of the application using the database.

2. Look for "well-known" values. The columns should appear the most in queries or maintenance operations. For example, "cust-name" may be used most often when querying the "Customer" table; "order-date" in the "Order" table may be used to generate sales reports; or "order-num" in the "Order" table may be used to spread out data evenly.

3. Look for "static" values. The values of the key should be known at creation time or change infrequently. This will avoid potential data movement due to partition change. Using the "Order" table as an example, "order-date" is obviously a better candidate than "ship-date," because the latter may not be known at the time the record is created.

4. Sub-partitioning offers versatility to suit different customer needs. Although up to 15 levels of columns are supported by OpenEdge Table Partitioning, a partitioning key that is too deep may reduce the flexibility, complicate the manageability, and slow down performance. It also has the potential to exhaust partition numbers much quicker.

5. To improve performance, choosing a partitioning key so that data will be distributed evenly is a critical success factor. Another factor is to avoid or at least spread out the "hot" partitions. In other words, work load on each partition should be balanced.

6. As a best practice, preliminary performance tests should be constructed on a smaller set of data to examine the performance impacts from different partitioning strategies. The right strategy then can be applied on a large scale.

4. Performance Study

A performance test will be examined to demonstrate the performance impact of different partitioning strategies on a database system. The test is configured to measure performance for record READ, WRITE and DELETE operations on the "Order" table, then results are compared when the table partitioning is enabled and disabled. Two partitioning strategies are used when the table is partitioned:

1. Range partitioning using "order-date" as partitioning key

2. Sub-partitioning using "region" and "order-date" as partitioning keys

For each partitioning strategy, a non-Table-Partitioning test will be used as the baseline for comparison. The performance changes are calculated with differences between partitioned and non-partitioned configurations. Various numbers of concurrent users are included in the test to show the performance changes along with work load.

SYSTEM CONFIGURATIONS

Physical Characteristics:

- **Type II Areas**
  - Data and index separated
  - 8 Kb block size with cluster sizes of 512 (data) and 64 (index)
  - All partitions in separate areas
  - Areas of proportional fixed sizes with matching database extents

- **Data**
  - Average record size 257, all same RPB (32)
  - 50,000 records to 10,000,000 per run (base on # users)
  - 3 Global indexes and 2 local indexes

- **Recovery**
  - 8 KB block with 128 MB cluster size

Server Parameters:

- **Buffer pool:** -B 50000 -lruskips 250
- **Lock table:** -L 100000 -lkwtmo 3600
- **Transaction:** -TXERetryLimit 1000
- **BI:** -bibufs 4000 -bwdelay 20
- **Latching:** -spin 50000 -napmax 10
- **Page writers:** 1 BIW 3 APWs

Testing Performed:

- **Scale users**
  - 1, 2, 5, 10, 25, 50, 100, 200
  - Avoid application side conflicts
  - Monitor internal resource conflicts

- **Operations executed:** Basic Create, Read, Delete

- **Vary transaction scope:** 10, 100, 500 records per transaction

- **Vary partitioning scheme**
  - No partitioning
  - Range partitioning on {order-date}
  - Sub-partitioning on {region(9), order-date}

Other Test Information:

- **Machine Stats**
  - 16 sparcv9 processor operating at 3600 MH
  - Memory size: 32768 Megabytes

- **Dbanalys performed before and after each activity**

- **Database recreated with same .st file for each run**

- **Variation across runs: ±1%**

TEST RESULTS

Two partitioning strategies have been used to measure performance changes between a partitioned and a non-partitioned configuration. The first strategy is to use sub-partitioning on the "Region" and "Order-date" columns; the second is to use range partitioning on the "Order-date" column. In both strategies, WRITE, READ and DELETE operations are performed on the "Order" table, and the time to complete each operation is used to measure performance of this operation. Then, the time values are compared to the ones from the non-partitioned configuration to show the performance difference. Different numbers of users are used for each configuration and tested to show the performance impact on system scalability.

Figure 1 is the comparison of partitioning by sub-partitioning "region" and "order-date" versus the non-partitioning baseline. As shown in the chart, the performance differences (the y-axis) of all three operations stay flat when user number (the x-axis) is less than 10. The performance of CREATE and DELETE starts to improve dramatically when the user number is greater than 25. The performance of "READ" operations shows little to no difference.

Another partitioning strategy using range partitioning shows different results, as illustrated in Figure 2 below:

Sub-partitioning on region AND order-date

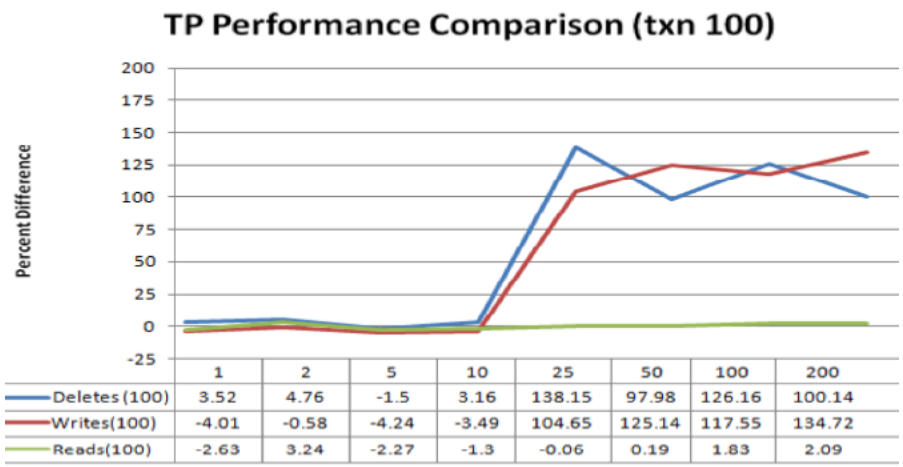


Figure 1: Performance comparison using "region" and "order-date" as the partitioning key.

Range partitioning on order-date only

Figure 2: Performance comparison using "order-date" as the partitioning key.

Figure 2 is the comparison of partitioning by range partitioning "order-date" versus the non-partitioning baseline. As shown in the chart, there is barely any improvement in performance from all three operations. And in most cases, the partitioned version is even slightly slower than the non-partitioned one.

From the comparison of these two partitioning strategies, sub-partitioning on the "Region" and "Order-date" columns proves the better choice, with as much as 138-percent performance gains instead of minor performance loss from the range partitioning on the "Order-date" column only.

RESULTS ANALYSIS

What is causing the big difference in performance when using different partitioning strategies? Load balancing, a technique to distribute workload and increase concurrency, contributes the most to the performance gains in the sub-partitioning scheme. As shown in Figure 3 below, when "Order" table is partitioned by "order-date," all newly created records will usually go to the same partition because their "order-date" values are close to each other. This makes partition A7 the only "hot spot" in the table, with no difference from the non-partitioning configuration.

Increasing Concurrency With Table Partitioning

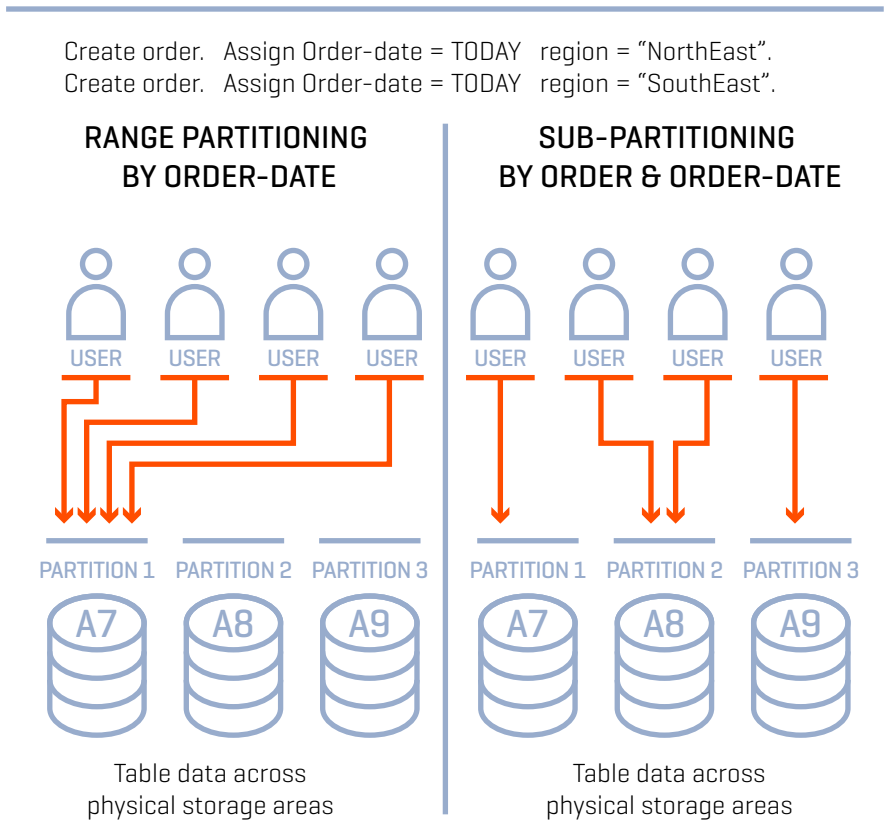


Figure 3: Increase concurrency with Table Partitioning.

When the table is partitioned by "region" and "order-date," even though the "order-date" values in two records may still be close to each other, they may be created into two different partitions, because they belong to different "regions." In this configuration, data access may happen to all three partitions: A7, A8, and A9. This makes the workload more balanced and less "hot."

Similar behavior can be observed for "DELETE" operations, but very little difference can be seen for "READ" operations using these two partitioning strategies. This is because the "READ" query will go through all the records in the table, all partitions will be accessed evenly, no matter how they are partitioned. Different results may be observed for queries with different brackets, because some query types may benefit more from one partitioning strategy than others. It's also interesting to note slightly slower performance for "READ" operations when compared to non-partitioning configuration. (I'll cover this in more detail later.)

Statistics on these operations can reveal more details about the internals of the system. Figure 4 is the promon data for "WRITE" operations using sub-partitioning strategy.

Analysis of the 100 user "write" example (117% runtime performance improvement):

Start	Base	Partition	Delta	Waits Delta
DB Buf I Lock	24,965,752	24,219,272	746,480	48,485,873
DB Buf S Lock	49,845,176	43,137,912	6,707,264	3,669,065
Find index entry	100	5,000,100	-5000000	0
BUF Latch	325,197,441	230,610,801	94,586,640	2,488,178
MTX Latch	31,877,745	31,280,314	597,431	-433,931
TXQ	60,339,744	60,930,902	-591,158	-71,178
Latch timeouts	3,205,140	1,088,120		2,177,020
Resource waits	64,643,103	6,940,906		57,702,197
Extends	44,224	48,224		

Figure 5: Statistics on WRITE operations, using "region" and "order-date" as the partitioning key.

In this case, buffer access from both records and indexes is significantly less, compared to the non-partitioned table. More dramatic change comes from the buffer waits in partitioned table, which is 21 times less. Further investigation also shows this sub-partitioning strategy makes much better usage of the buffer pool management algorithm, for example, with more concurrent buffer access and less paging. Statistics on buffer latch and total latch timeout also support the analysis that performance is improved due to better load balancing and increased concurrency.

It's also interesting to note that the partitioned version has more latch requests on MTX and TXQ latches, which are indicators for bottleneck in the crash recovery system, a very low-level component inside the database engine. Increased requests on these latches that this bottleneck on performance has been pushed much lower in the system, and also explains the higher throughput of the whole system.

Similar statistics can be observed for "DELETE" operations using the sub-partitioning strategy, as show in Figure 5 below.

Analysis for the 100 user "delete" example (126% runtime performance improvement):

Start	Base	Partition	Delta	Waits Delta
DB Buf I Lock	49,764,712	39,622,400	10,142,312	28,235,912
DB Buf S Lock	135,589,472	172,297,568	-36,708,096	45,324,198
BHT Latch	308,628,367	261,774,931	46,843,436	471,857
BUF Latch	527,202,766	516,930,247	10,272,519	2,995,964
Latch timeouts	5,200,069	1,849,659		3,350,410
Resource waits	120,187,904	46,963,709		73,224,195

Figure 5: Statistics on DELETE operations, using "region" and "order-date" as the partitioning key.

Analysis for the 100 user "read" example (1.8% runtime performance improvement):

Start	Base	Partition	Delta	Waits Delta
Index operations	25,000,100	25,000,200	100	0
DB Buf S Lock	26,059,284	26,044,498	14,786	6,816
BHT Latch	25,872,927	29,698,636	-3,825,709	29,532
BUF Latch	81,084,554	81,295,318	-210,764	-3,519
Latch timeouts	57,193	31,327		25,866
Resource waits	7,172	356		6,816

Figure 6: Statistics on READ operations, using "region" and "order-date" as the partitioning key.

Figure 6 is the promon data for "READ" operations, which shows very little difference between the partitioned and non-partitioned tables. One thing to notice is more requests on BHT and BUF latches for the partitioned case, which indicates some bottleneck in this area that prevents "READ" operations from fully taking advantage of load balancing and concurrency.

By contrast, Figures 7 and 8 are statistics from the partitioning strategy that uses range partitioning on "order-date" only. For the "WRITE" operations, buffer requests are significantly higher for the partitioned table, due to contentions in the buffer pool. "READ" operations show identical behavior as in Figure 6, with little change from different partitioning strategies, due to similar access patterns in the buffer pool.

Analysis for the 100 user "write" example (0.38% runtime performance loss):

Start	Base	Partition	Delta	Waits Delta
DB Buf I Lock	24,965,752	24,953,548	-11,156	347,384
DB Buf S Lock	49,845,176	55,152,964	-5,284,724	-16,267
DB Buf X Lock	41,941,720	42,181,992	-240,272	-324,844
Find index entry	100	5,000,100	-5000000	0
BUF Latch	325,197,441	333,313,379	-11,323,669	48,054
MTX Latch	31,931,319	31,944,574	-13,255	-4,037
Latch timeouts	3,166,505	3,132,653		33,852
Resource waits	61,308,496			6,865
Extends	3,392	10,880	-7488	

Figure 7: Statistics on WRITE operations, using "order-date" as the partitioning key.

Analysis for the 100 user "delete" example (126% runtime performance improvement):

Start	Base	Partition	Delta	Waits Delta
Index operations	25,000,100	25,000,100	0	0
DB Buf S Lock	26,782,308	26,503,194	279,144	372
BHT Latch	25,863,634	26,933,298	-1,069,664	1,428
BUF Latch	82,393,823	82,089,754	304,069	-790
Latch timeouts	59,747	58,588		1,159
Resource waits	3,893	3,521		372

Figure 8: Statistics on READ operations, using "order-date" as the partitioning key.

5. Conclusion

Table Partitioning can greatly enhance the manageability and availability of a database. In addition, different partitioning strategies can affect query performance and operational performance dramatically. Choosing the right strategy can help improve performance for operations such as “WRITE” and “DELETE,” but this test case has demonstrated very limited impact on “READ” operations. Conversely, a poorly chosen strategy may adversely affect performance, especially on “WRITE” and “DELETE” operations.

The key to achieve best performance improvement lies in balancing the load and increasing concurrency. As a best practice, different partitioning schemes should be tested for any performance impact prior to implementing Table Partitioning.

About Progress

Progress (NASDAQ: PRGS) is a global software company that simplifies the development, deployment and management of business applications on-premise or in the cloud, on any platform or device, to any data source, with enhanced performance, minimal IT complexity and low total cost of ownership. Progress can be reached at www.progress.com or 1-781-280-4000.

ADDITIONAL RESOURCES

Progress Corporate Blog

Follow Progress on Twitter, Facebook, LinkedIn and Google+

Progress and OpenEdge are trademarks or registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and other countries.
Any other marks contained herein may be trademarks of their respective owners. Specifications subject to change without notice.
© 2015 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.

Rev 08/15 | 150811-0079