ISSN No. 0976-5697

Volume 3, No. 3, May-June 2012

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

Trends in Hash Partition for Effective Object Data Access in Oracle Object-Oriented Database System

Clarence J M Tauro*
Center for Research
Christ University
Bangalore, India
clarence.tauro@res.christuniversity.in

Guruprasanna S
Department of Computer Science
Christ University
Bangalore, India
guruprsn@gmail.com

Girish S R
Department of Computer Science
Christ University
Bangalore, India
malur.girish@gmail.com

Joby Thomas
Department of Computer Science
Christ University
Bangalore, India
joby.thomas8@yahoo.com

Abstract: — Object-oriented database system is a superior model than relational database system, but as the database expands, even simple objects become complicated to handle. Although the more deep hierarchy B tree structure implies that we have good reuse of code but it has awful effects on memory and performance. In this paper we studied performance of B tree over retrieval of objects, effect on time and data sizes. Also we studied Hash partition provides different criteria to split the database and retrieval of the heavy objects using hash partition method in oracle object oriented database this is important for query performance and this paper emphasizes on performance of B-tree vs. Hash portioning.

Keywords: Oracle, OODB, OODBMS, Hash, B-tree

I. INTRODUCTION

In an object-oriented database management system (OODBMS) the information is represented in the form of objects as used in object-oriented programming. All entities of interest to an application can be defined as objects[2]. Objects having the same structural and behavioral properties are grouped together to form an object class. Object classes are interrelated with each other through various association types [3].

The performance of Object-Oriented Database depends on the access method implemented in the data model. B-tree is an indexing technique supporting query processing in Object-Oriented Databases which is effective and efficient for multimedia databases. This is a new access method which supports range queries on Object-Oriented Databases [4, 5]. B-tree supports inheritance and aggregation hierarchies. This has the structure of Dynamic Interpolation B-tree. Dynamic Interpolation B-tree consists of hashing and B-tree. Both hashing and B-tree are dynamic. By studying the performance of both hashing and B-tree it can be conclude which one has better performance over Oracle Object-Oriented Database [6].

II. PERFORMANCE STUDIES OF B TREE

A. *Effect of Time:*

To study the search performance of the indexes with the passage of time and updates summarizes the results, showing that the TPR-tree degrades considerably faster than the B-trees due to continuous enlargements of the MBR (minimum bounding regions) [2].

B. *Effect of Data Sizes:*

In the average number of I/O operations and the CPU time per range query for each index. The both B-tree variants scale very well maintain consistent performance, while the TPR-tree degrades linearly with the increase of the dataset size. When the dataset reaches 1M objects, the B-trees are nearly five times better than the TPR-tree [2]. This behavior may be explained as follows.

In the B-trees, every object has a linear order, which is determined by the space domain as the dataset grows the range query cost of the B-trees increases mainly due to the increase of the number of objects inside the range. However, the structure of the TPR tree is affected more by the dataset size. When the number of objects increases, the MBRs in the TPR (Time-Parameterized R-tree) tree have higher probabilities of overlapping. The B-tree(H-curve) achieves better performance than the B-tree(Z-curve) because the Hilbert curve generates a better distance-preserving mapping than the Piano curve, and hence yields fewer search intervals on the B-tree, i.e., less disk access. [2]

III. HASH PARTITIONING

Hash partitioning maps data to partitions based on a hashing algorithm that Oracle applies to the partitioning key that you identify. The hashing algorithm evenly distributes rows among partitions, giving partitions approximately the same size.

Hash partitioning is the ideal method for distributing data evenly across devices. Hash partitioning is also an easy-to-use alternative to range partitioning, especially when the data to be partitioned is not historical or has no obvious partitioning key. Note: You cannot change the hashing algorithms used by partitioning.

Figure 1: Hash partitioning

A. Composite List-Hash Partitioning:

Composite list-hash partitioning enables hash sub partitioning of a list-partitioned object; for example, to enable partition-wise joins.

B. Composite Range-Hash Partitioning

Composite range-hash partitioning partitions data using the range method, and within each partition, sub partitions it using the hash method. Composite range-hash partitioning provides the improved manageability of range partitioning and the data placement, striping, and parallelism advantages of hash partitioning as shown in Figure 2

Figure 2: Composite Partitioning - Range Hash

C. Global Hash Partitioned Indexes:

Global hash partitioned indexes improve performance by spreading out contention when the index is monotonically growing. In other words, most of the index insertions occur only on the right edge of an index [5, 6].

D. Hash Cluster Tables:

Clustered tables give you the ability to physically 'pre-join' object data together. You use clusters to store related object data from many tables on the same object database block. Clusters can help read intensive operations that always join object data together or access related sets of object data. They will reduce the number of blocks that Oracle must cache; instead of keeping 10 blocks for 10 employees in the same department, they will be put in one block and therefore would increase the efficiency of your buffer cache. Oracle will take the key value for a row, hash it using either an internal function or one you supply, and use that to figure out where the data should be on disk. One side effect of using a hashing algorithm to locate data however, is that you cannot range scan a table in a hash cluster without adding a conventional index to the table. [5, 6]

```
select * from emp where deptno between 10 and 20
```

Figure 3: SQL Query

a. In a hash cluster, the query in Figure 3 would result in a full table scan unless you had an index on the DEPTNO column. Only exact equality searches may be made on the hash key without using an index that supports range scans. In a perfect world, with little to no collisions in the hashing algorithm, a hash cluster will mean we can go straight from a query to the data with one I/O. In the real world, there will most likely be collisions and row chaining periodically, meaning we'll need more than one I/O to retrieve some of the data. Like a hash table in a programming language, hash tables in the database have a fixed 'size'. When you create the table, you must determine the number of hash keys your table will have, forever. That does not limit the amount of rows you can put in there. When you create a hash cluster, you will use the same CREATE CLUSTER statement you used to create the index cluster with different options. We'll just be adding a HASHKEYs option to it to specify the size of the hash table. Oracle will take your HASHKEYS values and round it up to the nearest prime number, the number of hash keys will always be a prime. Oracle will then compute a value based on the SIZE parameter multiplied by the modified HASHKEYS value. It will then allocate at least that much space in bytes for the cluster. This is a big difference from the index cluster above, which dynamically allocates space, as it needs it. A hash cluster pre-allocates enough space to hold HASHKEYS. So for example, if you set you're SIZE to 1,500 bytes and you have a 4 KB block size, Oracle will expect to store 2 keys per block. If you plan on having 1,000 HASHKEYs, Oracle will allocate 500 blocks.

Below is the result of a small utility stored procedure [6] to see what sort of space hash clusters takes. If we issue a CREATE CLUSTER statement, the storage it allocated in Figure 4:

```
tkyte@TKYTE816> create cluster hash_cluster
2 ( hash_key number )
3 hashkeys 1000
4 size 8192
5 /
Cluster created.
tkyte@TKYTE816> exec show_space(
'HASH_CLUSTER', user, 'CLUSTER' )
FreeBlocks.................0
TotalBlocks................1016
TotalBytes.................8323072
UnusedBlocks...............6
UnusedBytes................49152
Last Used ExtFileId.........5
Last Used ExtBlockId........889
Last UsedBlock.............2
PL/SQL procedure successfully completed.
```

Figure 4: Result of the Procedure

In the result given in Figure 4, the total number of blocks allocated to the table is 1,016. Six of these blocks are unused (free). One block goes to table overhead, to manage the extents. Therefore, there are 1,009 blocks under the high water mark of this object, and these are used by the cluster. 1,009 just happen to be the next largest prime over 1,000 and since my block size is 8 KB we can see that Oracle did in fact allocate (8192 * 1009) blocks. This figure is a little higher than this, due to the way extents are rounded and/or by using locally managed table spaces with uniformly-sized extents. This point out one of the issues with hash clusters you need to be aware of. Normally, if we create an empty table, the number of blocks under the high water mark for

that table is 0. If we full scan it, it reaches the high water mark and stops. With a hash cluster, the tables will start out big and will take longer to create as Oracle must initialize each block, an action that normally takes place as data is added to the table. They have the potential to have data in their first block and their last block, with nothing in between. Full scanning a virtually empty hash cluster will take as long as full scanning a full hash cluster. This is not necessarily a bad thing; you built the hash cluster to have very fast access to the data by a hash key lookup. You did not build it to full scan it frequently. [5]

a. The hash cluster did significantly less I/O (query column). This is what we had anticipated. The query simply took the random numbers, performed the hash on them, and went to the block. The hash cluster has to do at least one I/O to get the data. The conventional table with an index had to perform index scans followed by a table access by row ID to get the same answer. The indexed table has to do at least two I/Os to get the data. [6]
b. The hash cluster query took significantly more CPU. This too could be anticipated. The act of performing a hash is very CPU-intensive. The act of performing an index lookup is I/O intensive.
c. The hash cluster query had a better elapsed time. This may vary. The elapsed time for the hash cluster query was very close to the CPU time and system used.

IV. CONCLUSION

In this paper, we presented importance of OODBMS and the two methods of indexing method one is B tree where we did performance study on the effect of Time Elapsed on Range

Query Performance as time passes and effect of Data Sizes on Range Query Performance is relatively independent of the number of moving objects. Two the overview of Hashing partition and different types, we studied stored procedure to see what sort of space hash clusters take and we concluded the hash cluster query had a better elapsed time, significantly more CPU.

V. REFERENCES

[1]. Ubaid, M.; Atique, N.; Begum, S.; , "A pattern for the effective use of object oriented databases," Information and Communication Technologies, 2009. ICICT '09. International Conference on , vol., no., pp.229-234, 15-16 Aug. 2009.

[2]. Christian S. Jensen, Dan Lin, Beng Chin Ooi "Query and Update Efficient B -Tree Based Indexing of Moving Objects" [online] (2004). http://www.vldb.org/conf/2004/RS20P3.PDF (Accessed: 3 April 2012).

[3]. Silberschatz. Korth &, Sudarshan, Database System Concepts. 5th ed., Foxit Software Company, 2004, pp. 361–365.

[4]. J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[5]. Jianzhong Li; Wenjun Sun; Yingshu Li; , "Parallel join algorithms based on parallel B+-trees," Cooperative Database Systems for Advanced Applications, 2001. CODAS 2001. The Proceedings of the Third International Symposium on , vol., no., pp.178-185, 2001.

[6]. Thomas Kyte, "Expert One-on-One Oracle", Apress Publication, 2003, ISBN-10: 1590592433. pp. 170-235

[7]. Thomas Kyte,"Expert Oracle", Apress Publication 2005, pp. 231-260