Improving Performance with Bulk-Inserts in Oracle R-Trees

Ning An, Ravi Kanth V Kothuri, Siva Ravada

Spatial Technologies, Oracle Corporation, One Oracle Drive, Nashua, New Hampshire 03062, USA
Email: {*ning.an, ravi.kothuri, siva.ravada*}*@oracle.com*

Abstract

Spatial indexes play a major role in fast access to spatial and location data. Most commercial applications insert new data in bulk: in batches or arrays. In this paper, we propose a novel bulk insertion technique for R-Trees that is fast and does not compromise on the quality of the resulting index. We present our experiences with incorporating the proposed bulk insertion strategies into Oracle 10i. Experiments with real datasets show that our bulk insertion strategy improves performance of insert operations by 50%-90%.

1 Introduction

Spatial databases are widely used in multiple sectors such as census, cadastral management, environmental and urban planning, telecommunications, and CAD/CAM. Recent advances in wireless technologies have increased the scope of location-based applications thereby augmenting the spatial database market. Given the wide-range of applications and the proliferating market segment for spatial databases, most commercial database vendors, like Oracle and IBM, provide the infrastructure and support for storage and retrieval of spatial data. Oracle Spatial supports fast searching of spatial data using R-trees and Quadtrees [RSSB99]. These indexes combine and extend the salient performance features of existing spatial indexing techniques [Gut84, BKSS90, GLL98]. These indexes provide fast searching of spatial data and have been heavily optimized based on the type of data (points, non-points) and type of queries (within-distance, nearest-neighbor, etc.). In addition to fast searching capabilities, most commercial spatial applications also require fast insertion into spatial tables/indexes.

In this paper, we address the problem of inserting new data *in bulk* into indexed spatial tables. We first describe a framework for enabling bulk insertion in Oracle R-Trees. We then examine existing strategies for bulk insertion, most of which perform localized insertion and do not perform any global reorganization. We then describe Oracle's approach for performing bulk insertion in spatial indexes. The features of this approach are: (1) batched insertion into subtrees resulting in fast insertion times, and (2) fast reorganization of subtrees whenever there is overlap (not present in existing techniques) to ensure good quality of index. In addition to presenting this novel bulk insertion technique, we also present experiments to evaluate this proposed bulk-insertion technique in a commercial database server like Oracle. In these experiments on real datasets, we observe that the new bulk insertion strategy improves insertion-performance by 50%-90% without compromising the quality of the index. In some cases, the quality of the index (and thereby query performance) actually improves due to the better clustering behavior of our bulk-insertion strategy.

The rest of the paper is organized as follows. In Section 2, we describe the framework for enabling bulk-inserts in spatial indexes. In Section 3, we describe our new bulk insertion strategy. In Section 4, we describe our experiments with this bulk insertion strategy in Oracle10i database. In the final section, we summarize these results with pointers to future research.

2 Framework for Bulk Insertions

Bulk insertion in spatial indexes can be performed in two ways. In the first approach, bulk inserts into spatial indexes can be performed when the user issues an

```
insert into <user-table> (sub-query)
```

Any data resulting from the "sub-query" are grouped into 64K-memory-size batches and inserted into the spatial indexes.

In addition to the above array-insert interface through sub-queries in an insert SQL statement, Oracle Spatial provides the following *defered-indexing* model for performing bulk insertions:

1. User alters the index to be *deferred*.
2. Subsequent inserts into the index are stored in a separate "deferred" table associated with the index.

**Proceedings of the 29th VLDB Conference,
Berlin, Germany, 2003**

3. User alters the index to *synchronize*. An optional batch_size is also specified by the user. This operation retrieves all inserts in the "deferred" table and inserts them in batches of specified size into the index. Optionally, a quad-code could be computed for the centers of the MBRs of these geometry data and can be used to order the data being fetched and inserted in batches into the spatial index.

Note that the batch_size for bulk insertion in the deferred indexing model is controlled by the user. In contrast, for the SQL array insert it depends on internal memory chunk-size parameters set for the database.

3 Bulk Insertion in Oracle R-trees

In this section, we examine different approaches for performing batched inserts in R-Trees. We then describe our technique implemented in Oracle 10i.

3.1 Related Work

Other than trivially invoking regular insertion one-by-one (OBO) to perform an R-Tree bulk insertion, existing R-Tree bulk insertion techniques can be classified into merging-based and buffering-based.

Merging-based Techniques: In [CCR99], Choubey et al. proposed a technique, called Generalized R-Tree Bulk-Insertion strategy (GBI), an improved version of their work in [CCR98]. They first employed a conventional clustering algorithm to cluster the input data into a set of clusters and outliers. For each cluster, an R-Tree was built by bulk loading, and merged into the target R-Tree. They then use OBO method to insert the outliers into the target R-Tree.

Working along the same line, Kamel et al. [KKK96] first sorted the input MBR items based on the Hilbert values of MBRs' centers. Then they packed the sorted MBRs into blocks that is identical to nodes of the target R-Tree. At last, they would simply merge these blocks one by one using the standard sub Rtree insertion on the target R-Tree.

Merging-based techniques utilized the potential spatial proximity within the input data, and insert the proximate data into the target R-Tree together under the umbrellas of sub R-Trees. However, the constructed subtrees and the target tree could have overlapping nodes resulting in poor quality of index. This could lead to poor performance of subsequent queries.

Buffering-based Techniques: Arge et al. [AH$^+$99] took a different approach on implementing bulk insertions in R-Trees. Using an external memory paradigm [Arg96], they grouped the input MBR items that share the same descending path along the target R-Tree, and kept them together in buffers. Once a buffer is full, they descend its MBR items together along the target R-Tree to the next (lower) level buffers. This process would repeat until the data reach the leaf level, and were inserted to the target R-Tree properly. Their approach is asymptotically optimal in terms of I/O cost, and the query performance of their updated R-Tree matches the OBO updated R-Tree [AH$^+$99]. Its large auxiliary structure, however, requires extra space cost, and might not be feasible in the commercial database environment.

Our technique extends buffering-based techniques by not materializing the auxiliary structures and pushing the data right to the leaves. Besides, our technique merges subtrees whenever they overlap which is not present in any of the previous techniques

3.2 Bulk Insertion in Oracle R-Trees (BIOR)

In this section, we discuss our technique of Bulk Insertion in Oracle R-Tree, referred to as BIOR.

```
Bulk-insert(n, b, E):
  returns array of R-tree entries

  1. If (b is null and E   is null)
     return R-tree entry for node n.
  2. If n  is    a   leaf  node,
     child_entries_list
     := Union of (entries in n, b
        and E)
  3. Otherwise do the following:
    3.1 Consider n' =  n  union  b.
    3.2 Partition entries in n' into
        <c1, bc1, ec1>, <c2, bc2,
        ec2> ... such that
          ci is not null, bci and eci
          could be null, bci=bcj
          (ci=cj) iff i=j, and
          Union of eci=E(disjoint
          subsets).
          The partition that
          minimizes the total overlaps
          across all partitions is
          chosen using the Choose
          Subtree algorithm (of
          R*-Tree).
    3.3 child_entries_list
        := Union of (Bulk-insert(ci,
           bci, eci)) for all i
  4. return
    rtree_cluster(child_entries_list);
```

Figure 1: BIOR Algorithm at A Node

Whenever new data items are inserted into the child subtrees of a node, the child subtrees could expand and overlap. Such overlaps could be avoided if the subtrees are reorganized so as to minimize overlaps. To minimize the effort in re-organization, new data items should be inserted into sets of potentially-overlapping subtrees. This could be achieved by treating the potentially-overlapping roots of the subtrees as one big "cluster node". As this process is continued downward to the leaves, the "cluster node" can become too large to fit in memory. To limit the size of this "cluster node", we restrict the size of the potentially overlapping subtree set to 2, i.e., at most two child entries will

be paired for inserting a set of items. We refer to such pairs of entries as "buddy" entries and corresponding nodes as buddy nodes. In Figure 1, we describe the algorithm for inserting the data items array E into a node n and its buddy b (b can be null, e.g. at root). This algorithm returns a set of entries to be incorporated in the parent of node n.

Using a stack-based implementation and expanding the children in a depth-first fashion, the above recursive algorithm can be implemented with limited memory (proportional to the height of the resulting tree). The clustering algorithm for clustering the child entries could be the same as the one employed at index creation time. Note that each node in the tree is accessed/updated at most once. This algorithm combines the salient features of existing bulk insertion algorithms and at the same time improves index quality by a "regulated" merge of the most overlapping pairs of subtrees.

4 Empirical Study

We have implemented our bulk insertion technique in Oracle 10i. To evaluate this novel technique, we conduct various experiments with real datasets and different queries. Due to space limitation, we will only present the result of the US Blockgroup (USBG) dataset which consists of about 230K polygon geometries.

In the following study, we keep 100K of USBG as the base table, and build an Oracle R-Tree on it. The remaining 130K USBG data serve as the pool of incoming data, various portions of which will be inserted to the existing R-Tree.

We use a workload of 1,000 window queries that follows the distribution of the target dataset and reflects the aspect ratio of actual data. To eliminate the caching effects in Oracle System, we choose the number of R-Tree nodes being accessed as the main metric in our study. Our testing system is a SUN OS 5.8 running on a SUN Ultra-60 Workstation.

4.1 Batch Size Study

In this subsection, we study the impact of the batch size on BIOR technique. Here, we randomly take 10K USBG data from the incoming data pool, which is 10% of the number of geometries in the base table. Figure 2 shows the insertion performance and the query performance for various batch sizes. As the batch size increases initially, the insertion performance improves because more incoming data can be grouped together and descend the R-Tree together. When the batch size goes beyond 800, the insertion performance starts to flatten revealing that it approaches the saturation point where the number of nodes must be accessed for a particular set of incoming data. In general, batch size 3200 delivers both good insertion and query performance, and it will be used for the following studies.

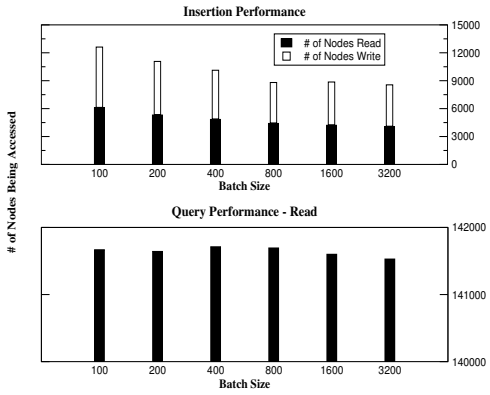

Figure 2: **The insertion performance improves as the batch size increases from 100 to 800, and starts to flatten beyond 800. The query performances are marginally different across the board while the batch size 3200 has the best result.**

4.2 Incoming Dataset Size Study

In this subsection, we examine how the size of incoming dataset affects BIOR technique, and compare it with OBO technique. For this study, we randomly take 10,000, 20,000, 40,000, 80,000, and 100,000 data from the incoming data pool as our incoming datasets which in turn are 10%, 20%, 40%, 80%, 100% of number of geometries in our base table respectively.

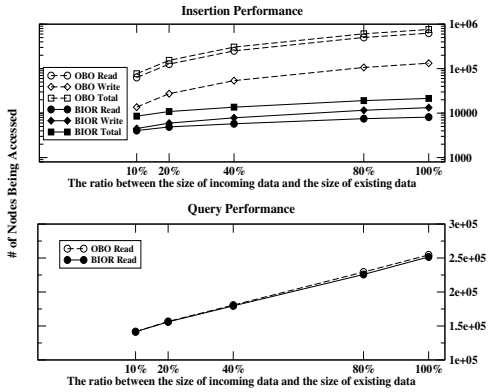

Figure 3: **As the size of incoming dataset doubles, the insertion cost of OBO almost doubles while the one of BIOR only grows about 20%-30%.**

From Figure 3, we have the following observations:

- In OBO technique, the number of reads is much larger than number of writes during the insertion. It reveals that OBO requires more work in locating the right place for the incoming data than actually putting them in the right place. On the other hand, in BIOR, the number of writes is consistently larger than the number of reads because BIOR mostly locates the right place for a group of incoming data instead of individual one, and does more work in putting the incoming data in the right place than locating the right place for them.

- As the size of incoming dataset doubles, the insertion cost of OBO almost doubles while the one of BIOR only grows about 20-30%.
- As to the query performance, BIOR has a slight edge over OBO technique. It becomes more noticeable with the increase of incoming data size.

We can conclude from these observations that the advantage of employing BIOR over OBO becomes more apparent as the size of incoming dataset increases.

4.3 Clustering Impact

Incoming datasets often have different clustering characteristics. To study the impact of these different clustering characteristics on the performance of BIOR, We take a set of 100,000 geometries as our base table, and the following three sets of incoming geometries: 1) *Clustering Outside*: 10,000 geometries that are clustered in one area, and intersect with no geometry in the base table; 2) *Clustering Inside*: 10,000 geometries that are clustered in another area, and intersect with some geometries in the base table; and 3) *Random*: 10,000 geometries that are randomly chosen and hence are not clustered in any area. Figure 4 compares BIOR with OBO on these three different kind of incoming data.

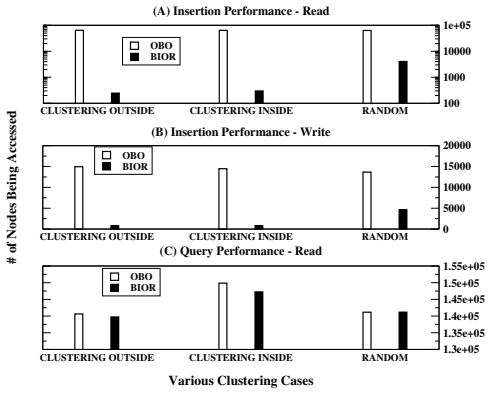

Figure 4: **For the clustering datasets, BIOR has superior insertion performance, and slightly better query performance compared to the the ones of OBO. For random data set, BIOR has much better insertion performance than OBO while their query performances are almost same.**

For the clustering datasets, where incoming data are spatially close to each other, BIOR effectively combines the traversal/descent of the target R-Tree. As a result, BIOR is about 90% faster than OBO for the clustering dataset as shown in Figure 4. Even for the random datasets, where the grouping affect of BIOR is not as dominant, BIOR is still 20% faster. In terms of query performance, BIOR is comparable to OBO, and is even slightly better than the latter one in the clustering datasets.

5 Conclusions

Bulk insertion in spatial indexes is a common operation in many commerical applications. Inserting data one by one (OBO) into the spatial index (1) could be slow, and (2) could deteriorate the quality of the index thereby adversely affecting subsequent query performance. In this paper, we proposed a novel bulk insertion strategy for Oracle R-Trees that combines multiple inserts and reduces the number of tree traversals. This strategy also improves the quality of the constructed R-tree index by reorganizing overlapping subtrees. The experiments using real datasets show an improvement in insertion performance by 50%-90% in comparison to an OBO insertion approach. Query performance also improves. Future work can compare the performance of our bulk insertion strategy with other proposed bulk insertion techniques.

References

[AH+99] L. Arge, K. Hinrichs, et al. Efficient Bulk Operations on Dynamic R-trees. In *Workshop on Algorithm Engineering and Experimentation (ALENEX)*, pages 328–348, 1999.

[Arg96] L. Arge. *Efficient External-Memory Data Structures and Applications*. BRICS Dissertation Series, DS-96-3, University of Aarhus, 1996.

[BKSS90] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R* tree: An efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 322–331, 1990.

[CCR98] L. Chen, R. Choubey, and E. Rundensteiner. STLT: Bulk Insertion into R-trees. In *Proceedings of ACM International Workshop on Advances in Geographic Information Systems*, pages 161–162, 1998.

[CCR99] R. Choubey, L. Chen, and E. Rundensteiner. GBI: A Generalized R-Tree Bulk-Insertion Strategy. In *Symposium on Large Spatial Databases*, pages 91–108, 1999.

[GLL98] Y. J. Garcia, S. T. Leutenegger, and M. A. Lopez. A greedy algorithm for bulk loading R-trees. In *Proc. of ACM GIS*, 1998.

[Gut84] A. Guttman. R-trees: A dynamic index structure for spatial searching. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 47–57, 1984.

[KKK96] I. Kamel, M. Khalil, and V. Kouramajian. Bulk Insertion In Dymanic R-trees. In *Proceedings of 4th International Symposium on Spatial Data Handling*, pages 3B.31–3B.42, 1996.

[RSSB99] K. V. Ravi Kanth, Siva Ravada, J. Sharma, and J. Banerjee. Indexing medium-dimensionality data in oracle. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 1999.