Post-optimization and Incremental Refinement of R-trees*

Yván J. García R. Mario A. López Scott T. Leutenegger

Mathematics and Computer Science Department

University of Denver, Denver, CO 80208-0189

{ygarcia,mlopez,leut}@cs.du.edu

Abstract

We develop an R-tree node restructuring algorithm that performs post-optimization of existing R-trees and improves current dynamic insertion algorithms. On realistic data and relative to state-of-the-art R-trees, our post optimization technique improves point query performance by 36% – 43% on average (and up to a factor of 2 in some cases), while only incurring an optimization cost (measured in disk I/Os) equal to STR loading. When used to modify existing dynamic insertion algorithms, our technique results in trees that require between 25% and 45% fewer disk accesses on average (up to 70% in some cases), relative to R*-trees and Hilbert R-trees, respectively, at only an additional 10% insertion cost.

1 Introduction

R-trees [5] are a common indexing technique for multidimensional data and are widely used in spatial and multidimensional databases. A significant amount of research has focused on the efficient construction of good quality R-trees [1, 2, 3, 4, 6, 7, 9, 13, 14]. This research has considered both static and dynamic data. Dynamic data can change via insertions and/or deletions. Static data, on the other hand, does not change. Once a static data set is loaded, only query operations are allowed.

We distinguish between "packed" and dynamic R-trees. The former result from bulk loading a static set of data. The latter, which are produced by repeated updates to the tree, correspond to dynamic data. Since static data is known a priori, packing algorithms [4, 6, 9, 13] load the tree, filling nodes to near 100% capacity. Packed R-trees have been shown to require lower loading time and result in improved query performance, relative to dynamic R-trees. Popular variants of dynamic R-trees include [2, 5, 7, 14]. Current dynamic R-trees typically suffer from three disadvantages: (a) node utilization is lower than packed trees, (b) node splitting heuristics usually ignore rectangle boundaries by using rectangle centers only, and (c) splitting decisions are often localized and ignore the global structure of the tree.

We propose a method that uses incremental node restructurings to address the above disadvantages. Node restructuring can also be used to improve the efficiency of packed trees. Specifically, the main goals of this work are: (a) to improve existing R-trees through post-optimization regardless of how poorly structured the tree currently is; and (b) to develop a dynamic R-tree framework that results in query performance comparable or better than current packing algorithms.

Through extensive experimentation we show a reduction in the number of query disk accesses by factors of up to 2.13 and 1.71 for post-optimization and dynamic construction respectively. On average, our method increases insertion cost by 10% or less compared to existing dynamic algorithms. However, node restructuring may occur during idle periods of time, thus reducing the effective cost even further.

The only similar work we know of is that of Gavrila [4], but his method achieves significant improvements only for highly skewed high dimensional data. Our method, on the other hand, uses a totally different approach that results in significant performance improvements for low dimensional data. Further experimentation is needed to asses the effectiveness of our approach for high dimensions.

In Section 2 we discuss background information on R-trees. In Section 3 we present our new optimization framework. Section 4 describes our experimental methodology. Section 5 contains experimental results, and Section 6 concludes.

2 R-trees and Loading Algorithms

In this section we describe background material. First, we assume the reader is familiar with the basic R-tree structure [5] and two of the best performing dynamic variants: R*-trees [2] and Hilbert R-trees [7] (dynamic Hilbert). GUT denotes R-trees built using Guttman's insertion algorithm with quadratic split [5].

*This work has been partially supported by the National Science Foundation under grant IRI-9610240.

```
restructure( node n )
{
    if n is not the root then
        restructure( parent(n) )
    mark n and all its siblings as clean
    foreach node s that is a clean
            overlapping sibling of n
        if |s| + |n| ≤ M then
            Let n = s ∪ n     { merge n, s }
            remove entry for s in parent(n)
            break (out of the for loop)
        else
            (n1, n2) = split( n ∪ s )
            Let n = n1
            mark n2 as dirty
            make the entry for parent(s)
                point to n2
}
```

Figure 1: Basic node restructuring algorithm.

Second, we assume the reader is familiar with packing algorithms, where the entire data set is bulk-loaded in order to produce a better R-tree faster. We consider both the Hilbert [6] (packed Hilbert) and STR [9] packing algorithms. As a stress-test for our post-optimization algorithm, we also consider a packing that generally provides poorly structured trees. This algorithm, which we call BULK, packs entries in the leaf nodes in random order.

Finally, the reader should be familiar with past work on performance prediction of R-trees [6, 8, 12].

3 The Post-Optimize and MIX Algorithms

In this section we describe our post-optimization and dynamic algorithms. Both use a node restructuring step which is described first.

3.1 Node Restructuring

Node restructuring is used to improve the global structure of the tree. It is based on the following principles:

(a) Increase node utilization by merging selected nodes.

(b) Redistribute node entries among siblings so as to reduce the expected query cost.

(c) Siblings for redistribution of entries are those whose MBRs overlap the MBR of the node being restructured.

(d) Before redistributing among siblings, improve sibling selectivity by first restructuring the parent node.

These principles are combined to produce the node restructuring algorithm of Figure 1. We also investigated the effect of the following split algorithms for entry redistribution:

R-star. Node splitting used for the R*-tree [2].

Axis-Sweep. The rectangle centers are sorted on each axis and a partitioning at each point in the sorted list is considered. We choose the partitioning over all axes that minimizes the weighted area and perimeter function of [6, 12].

Hilb-Sweep. Similar to axis-sweep, but using Hilbert values of rectangle centers to guide the sweep.

Pseudo-Optimal. This is the split algorithm proposed in [3]. In Section 5, we show that this method produces the best trees among the split functions considered.

3.2 Post-Optimization

Our post-optimization algorithm consists of a sequence of node restructurings. As discussed in Section 5, empirical evidence suggests that n restructurings are sufficient in order to improve the performance of a tree of n nodes. We consider the following ways of selecting the next node to restructure:

Weight Driven. A priority is assigned to each node. We consider the following priorities: (1) area of the node's MBR, (2) node cardinality (number of entries in the node) and, (3) random. Experimental evidence suggests that a good heuristic is to select each node for restructuring exactly once.

Query driven. Node restructuring is guided by a a sequence of rectangles chosen at random from a candidate set. One rectangle at a time is used to traverse the tree as when inserting the rectangle. The selected leaf node is then restructured. We consider different candidate sets: (1) MBRs of leaves from the original tree, (2) A sample of 1% of the data entries in the tree, and (3) random points in the unit square.

3.3 Dynamic MIX

The MIX algorithm is used for building dynamic R-trees. After an insertion, a restructuring is performed with probability p on a node chosen as described in Section 3.2. As discussed in Section 5, we found that a value of $p = 0.04$ achieves a good balance between overhead and tree quality. This value increases insertion cost by at most 10%. Furthermore, restructurings invoked during idle periods reduce the effective overhead cost. For our tests, we use Guttman's insertion with quadratic split.

4 Experimental Methodology

4.1 Metrics

All experimental results are obtained by repeatedly querying a tree built using a given algorithm. We report the number of disk accesses for a "warmed" buffer of 50 pages since the number of nodes accessed by a query can be a misleading metric [8]. Each data point in our tables and graphs is obtained by querying the tree 10,000 times and taking the mean number of buffer misses. Queries are square regions whose side length is either 0 or 0.02 and whose lower left corner is uniformly distributed within the unit square. Queries are truncated to fit into the unit square.

4.2 Data

We consider several synthetic and real data sets (available from www.cs.du.edu/users/leut/MultiDimData.html):

CFD. A set of 52,510 points corresponding to solutions of a 2-dimensional system of equations modeling air flows over and around aerospace vehicles [11].

Tiger. A set of 53,145 line segments for the Long Beach county coming from the TIGER system of the U.S. Bureau of Census. This set has been extensively used in the past.

VLSI. A set of 97,069 rectangles taken from a CIF file provided by Bell Labs for the design of a chip [10].

Unif.0. a synthetic set of 50,000 points uniformly distributed over the unit square.

Unif.1. A synthetic set of 50,000 squares with uniformly distributed lower left corners. All squares have area 1/50,000. Squares are truncated to fit into the unit square.

Cluster. A synthetic set of 50,000 rectangles intended to resemble clustered data arising from actual applications [3].

5 Experimental Results

For the remainder of this section we define the *disk access ratio* to be the ratio of the average number of buffer misses (disk accesses) for the non-optimized tree divided by the average number of buffer misses for the optimized tree.

The buffer (50 pages) holds about 5% to 10% of the tree. For static data, the minimum node utilization during a split is the total number of entries to be redistributed minus the maximum node utilization. This allows the maximum freedom of choice. We choose a minimum utilization of 40% for dynamic data. We assume the number of restructured nodes during the optimization is equal to the number of nodes in the original tree, and the objective function value for perimeter weighting is 0.2. We justify these choices in Section 5.5.

5.1 Improvement for Static Data

We first build the tree for the given data set using a given loading method and then post-optimize the tree by applying a series of restructurings. Table 1 presents the disk access ratios for post-optimization of trees built using the dynamic-Hilbert, packed-Hilbert, and STR loading algorithms. Post-optimization can decrease the number of query disk accesses by a up to a factor of 2. When averaged over the realistic, i.e. non-uniform, data sets (cfd, cluster, tiger, and vlsi) post-optimization of packed-Hilbert and STR trees reduced the number of disk accesses by 43% and 36%, respectively. We exclude uniform sets from the averages because it is easy to provide good packing for uniform data, thus diminishing the need for sophisticated packing or restructuring algorithms. As shown in section 5.4, the cost for the post-optimization is about the same as the initial loading using STR and twice that of Hilbert.

Note that post-optimization does not improve STR for the uniform data sets. This is because STR creates a packing with no leaf level intersections and hence the node restructurings do nothing. We did try a different optimization algorithm in which each node restructure reparations with the nearest sibling only. This optimization did improve STR for uniform data, but did not help significantly for other data sets.

Similar tests assuming region queries showed similar trends but the magnitude of the improvement is reduced.

data	dynamic Hilbert	packed Hilbert	STR
unif.0	1.29	1.24	1.00
unif.1	1.31	1.23	1.00
cfd	1.36	1.21	1.13
cluster	1.53	1.36	1.11
tiger	1.53	1.38	1.08
vlsi	2.08	1.78	2.13
avg.	1.63	1.43	1.36

Table 1: Disk access ratios for post-optimization (point queries).

We next consider the effectiveness of post-optimization when applied to trees of different quality. In Figure 2 we consider the Tiger data set and plot the number of query disk accesses versus the number of nodes restructured for five different initial trees: STR, static-Hilbert, dynamic-Hilbert, Guttman's quadratic, and BULK. As the number of nodes restructured increases the performances of the trees converge regardless of the initial tree structure.

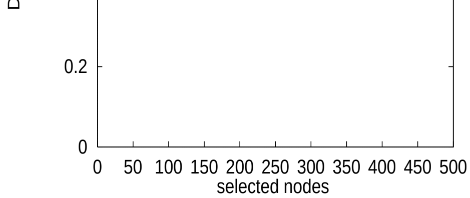

Figure 2: Equivalence of various post-optimized trees (point queries).

5.2 Improvement for Dynamic Data

We compare our method with both dynamic and static R-tree algorithms. For the Hilbert R-tree we assume 2-3 splitting is used as suggested in [7]. For the R*-tree method, as suggested in [2], we assume 30% of node entries are forced to be reinserted upon insertion into a full node. We assume the minimum node utilization during a split is 40%, and the objective function value for perimeter weighting is 0.2. Furthermore, we assume the probability that an insertion triggers a node restructuring is 0.04. We justify these choices for MIX in Section 5.5. In Section 5.4 we show that the restructurings require less than 10% additional I/O cost relative to the normal dynamic insertion algorithms.

data	R*	dynamic Hilbert	packed Hilbert	STR	GUT	post-opt GUT
unif.0	1.00	1.18	1.18	0.89	11.43	1.00
unif.1	1.01	1.10	1.09	0.82	3.56	0.92
cluster	1.18	1.37	1.32	1.08	3.19	0.89
tiger	1.19	1.40	1.38	1.04	1.45	0.89
vlsi	1.38	1.71	1.58	1.92	1.93	0.86
avg.	1.25	1.45	1.43	1.35	2.19	0.88

Table 2: Disk access ratios for MIX (point queries).

Table 2 summarizes disk access ratios, where the denominator is the average number of disk accesses for MIX. Comparing with other dynamic algorithms we find that for non-uniform data on average MIX requires (25%,45%,119%) fewer disk accesses than the number of accesses performed by the (R^*, Hilbert, Guttman) algorithms. For some data sets the improvement is significantly higher. Thus, MIX significantly outperforms the current state of the art. MIX is even superior to the static packing algorithms (static-Hilbert and STR) for realistic data. Finally, we compare MIX to post-optimization of a tree built using Guttman's quadratic split. We see that despite MIX doing almost twice the number of node restructurings, the post-optimization algorithm performs best. Note that if we apply post-optimization to the MIX tree, it too converges to the performance of the post-optimized Guttman tree.

5.3 Occasional Re-packing

We now present results for trees built for data that does not change often. We consider trees built initially by Hilbert packing with only 90% of the input data and then the remaining 10% inserted one record at a time by using Guttman's quadratic method. Note that due to the tree convergence effect, any other dynamic insertion method would have led to similar results. Consider two alternatives for "fixing" the structural damage to the tree due to insertions: (a) Re-apply the initial packing algorithm (Hilbert). (b) Post-optimize the tree using a series of n restructurings where n is the number of nodes in the tree before post-optimization.

In Table 3 we present the disk access ratios (Hilbert / post-optimize) of the two resulting trees for each of the data sets. Post-optimization is a better alternative by a factor of 3.85 (2.67) in improvement of point (region) queries.

5.4 Cost of Algorithm

We now show that the I/O cost of post-optimization is within a factor of 2 of STR, and the I/O cost of MIX is less than an additional 10% relative to dynamic Hilbert.

data	point query	.02 x .02 query
unif.0	1.286	1.090
unif.1	1.255	1.099
cluster	1.496	1.211
tiger	1.441	1.170
vlsi	8.615	5.629
avg.	3.851	2.670

Table 3: Disk access ratios for re-packing (point queries).

data	avg. number of nodes touched	missed
unif.0	5.29	0.50
unif.1	12.91	2.41
cfd	5.64	0.53
cluster	10.62	1.74
tiger	12.05	2.22
vlsi	18.72	3.85
avg.	11.76	2.09

Table 4: STR restructuring time

data	avg. number of nodes touched	missed
unif.0	20.29	5.45
unif.1	23.06	6.27
cfd	17.82	3.66
cluster	13.30	2.64
tiger	13.90	2.67
vlsi	20.01	4.66
avg.	16.26	3.41

Table 5: BULK restructuring time

In Tables 4 and 5 we present overhead costs for restructuring one node when applying post-optimization to STR, and BULK. Note that the worse the initial tree is the bigger the overhead incurred by post-optimization, i.e., the overhead for BULK is greater than for STR. The results for BULK are included to show the worst case costs we observed for any tree, as BULK is probably a more poorly structured tree than one would see in practice. The tables present metrics for the restructuring of one node. The first column is the number of nodes touched during the restructuring, the second column the number of buffer misses (i.e., disk accesses). The time for I/O dominates CPU time so we use use column 2 as our primary comparison metric. For the STR tree the number of disk access per restructured node is about 2 on average. Thus, for a tree of n nodes, if the post-optimization process restructures n nodes, the total I/O cost is about twice the number of nodes in the tree. This is comparable to the original STR packing algorithm.

We now consider the overhead incurred by MIX. We assume that 4% of the insertions trigger a node restructuring. In this case, the 4 restructurings require about 8-10 I/Os relative to 100 leaf level insertion I/Os. Thus, the additional I/O cost is less than 10%.

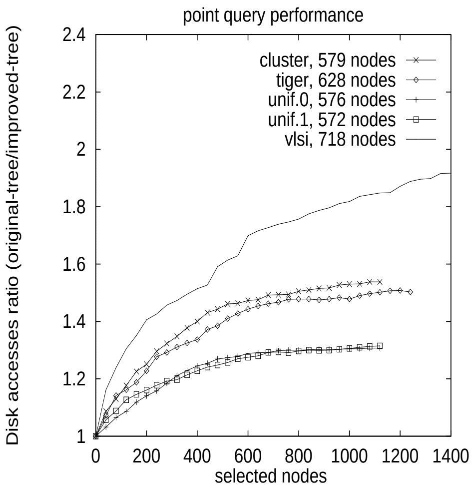

Figure 3: Dynamic Hilbert disk access ratio vs. number of restructured nodes.

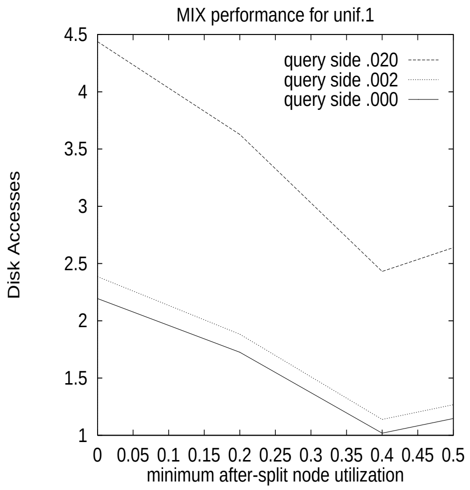

Figure 4: Disk accesses as function of node utilization.

5.5 Parameter Choices

We now justify our choice of parameters. First consider the number of nodes restructured during post-optimization. Figure 3 plots disk access ratio versus number of node restructurings, assuming the initial tree is built by dynamic Hilbert. We consider six data sets and vary the number of restructurings from 0 to twice the number of nodes in the tree. As a rule of thumb, we find that the "knee" of the curve occurs at about the half way point, i.e., about the number of nodes in the tree, thus justifying our choice.

The next parameter we consider is the minimal occupancy per node allowed by the pseudo optimal split during a node restructuring in MIX. In Figure 4 we plot the number of disk accesses versus the minimum allowed node utilization for MIX using the unif.1 data set. The best performance is obtained for a utilization of 40%. Similar tests with other data sets showed 40% to be best also.

Next consider the perimeter weighting parameter used in the objective function of the pseudo optimal split. In Figure 5 we plot the number of disk access versus the margin weight for the unif.1 data set. For this data set and others we experimentally found the best parameter to be around 0.2.

Another parameter is the probability of triggering a restructuring after an insertion in MIX. In Figure 6 we plot the number of disk accesses versus the percentage of restructurings for the MIX algorithm. A value of 4% provided the best trade-off between query performance and insertion cost.

We now show why we use random for picking the next node to restructure. In Table 6 we present the disk access ratio for post-optimization averaged over all data sets for the different methods of choosing the node to be split for both static-Hilbert and STR trees. We chose random since it is comparable and the most efficient to implement.

	Hilbert packed		STR	
query	point	.02x.02	point	.02x.02
area	1.387	1.243	1.279	1.187
card.	1.402	1.245	1.268	1.185
random	1.397	1.244	1.275	1.185
leaf MBR	1.381	1.239	1.240	1.166
data subset	1.349	1.217	1.246	1.166
random points	1.311	1.206	1.251	1.170

Table 6: Average improvement by node selection.

Finally, in Table 7 we present the disk access ratio for various split algorithms. As can be seen from the table, pseudo-split results in the best performance.

6 Conclusions

We presented an R-tree optimization technique that executes a sequence of nodes restructures. This technique can be used to post-optimize existing R-trees or to improve insertion algorithms by intermixing insertions and restructures.

We have shown that when applied to existing R-trees, produced by Hilbert or STR packing, the technique can cut the number of query disk access by up to a factor of 2. When averaged over the realistic data sets considered we show an improvement of 43% and 36%, relative to Hilbert and STR R-trees, respectively. These savings come at a cost roughly equal to STR packing.

We have also shown that when a small number of restructuring are mixed in with insertions (the MIX algorithm), resultant query performance improves by up to 70% and, on average, (25%, 45%) relative to the current best dynamic R-tree algorithms (R^*, Hilbert). This improvement increases

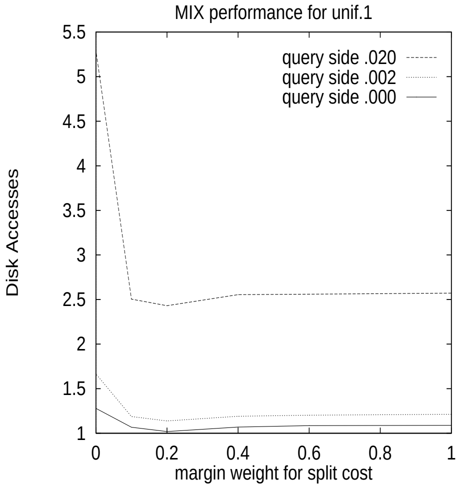

Figure 5: Disk accesses as function of perimeter weight

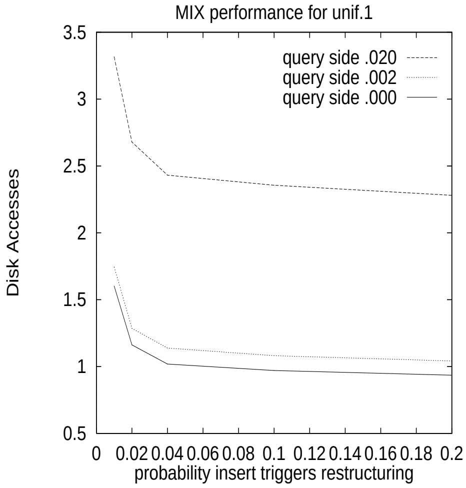

Figure 6: Disk accesses as function of /restructuring probability.

	Hilbert packed		STR	
query	point	.02x.02	point	.02x.02
pseudo optimal	1.397	1.244	1.275	1.185
axis-sweep	1.097	1.086	1.125	1.091
hilb-sweep	1.004	0.999	0.671	0.738
R-star	1.101	1.081	1.085	1.073

Table 7: Average improvement by splitting policy.

insertion cost by no more than 10%.

Finally, our restructuring technique can be used during periods of low utilization to continually refine existing R-trees at no cost other than unused machine capacity.

References

[1] B. Becker, P. Franciosa, S. Gschwind, T. Ohler, G. Thiemt, and P. Widmayer. Enclosing many boxes by an optimal pair of boxes. In *Proc. 9th Annual Symp. on Theoretical Aspects of Computer Science (STACS 92)*, p. 475–486. Springer LNCS #577, 1992.

[2] N. Beckmann, H.P. Kriegel, R. Schneider, and B. Seeger. The $R^\star$-tree: an efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD*, p. 323–331, May 1990.

[3] Y. Garcia, M.A. Lopez, and S.T. Leutenegger. On optimal node splitting for R-trees. In *Proc. of VLDB 98*, New York, NY, August 1998.

[4] D. Gavrila. R-tree index optimization. Tech. Report CS-TR-3292, Univ. of Maryland, College Park, June 1994.

[5] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. ACM SIGMOD*, p. 47–57, Boston, Mass, June 1984.

[6] I. Kamel and C. Faloutsos. On packing R-trees. In *Proc. of 2nd Int. Conf. on Information and Knowledge Management (CIKM)*, November 1993.

[7] I. Kamel and C. Faloutsos. Hilbert R-tree: an improved R-tree using fractals. In *Proc. of VLDB Conference*, p. 500–509, September 1994.

[8] S.T. Leutenegger and M.A. Lopez. The effect of buffering on the performance of R-trees. In *Proc. of the 15th International Conference on Data Engineering (ICDE 98)*, p. 164–171, 1998.

[9] S.T. Leutenegger, M.A. Lopez, and J.M. Edgington. Str: A simple and efficient algorithm for R-tree packing. In *Proc. of the 14th International Conference on Data Engineering (ICDE 97)*, p. 497–506, 1997.

[10] M.A. Lopez, R. Janardan, and S. Sahni. Efficient net extraction for restricted orientation designs. *IEEE Trans on Computer-Aided Design*, 15(9), p. 1151–1159, 1992.

[11] D.J. Mavriplis. An advancing front delaunay triangulation algorithm designed for robustness. *Journal of Computational Physics*, 117, p. 90–101, 1995.

[12] B-U. Pagel, H-W. Six, H. Toben, and P. Widmayer. Towards an analysis of range query performance in spatial data structures. In *Proc. ACM PODS*, p. 214–221, May 1993.

[13] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed R-trees. In *Proc. ACM SIGMOD*, May 1985.

[14] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+ tree: A dynamic index for multidimensional objects. In *Proc. of 13th International Conference on Very Large Databases (VLDB-87)*, p. 507–518, Sept. 1987.