The Design and Implementation of Seeded Trees: An Efficient Method for Spatial Joins

Ming−Ling Lo and Chinya V. Ravishankar, 1996, IEEE.

Presented by Arwa Maiss

The Need for Dynamically Constructed Index Structure

Existing methods for spatial joins require pre−existing spatial indices or other pre−computation, but such approaches are inefficient and limited in generality.

- They are mostly designed for spatial *selections*, and are not always efficient for joins.
- Operand data sets of spatial joins may not all have pre−computed indices:
  - It may not be cost−effective to maintain index structures for all data sets
  - They may be dynamically generated by other selection or join operations (*Derived Data Set*)
- For spatial queries that allow multiple execution paths, it may be beneficial to execute other operations before spatial joins
  Query: Find all buildings built on government−owned land.

Needed: A spatial data structure that is:

- *Inexpensive* to build dynamically at join time (Spatial indices are built incrementally and not optimized for all–at–once construction)

- Takes advantage of the information available at join time to support efficient spatial join processing (Spatial indices are designed to minimize the cost of spatial selection)

    - Fig.1

The Seeded Tree

We assume a structure in which the R–tree is the main type of spatial index.

We want to join

- A derived data set D_s
- An ordinary data set D_R Has an R–Tree index T_R

Upon encountering such a join operation, the join algorithm:

1– Constructs a seeded tree for one of the participating data sets

2– Proceeds to compute the join result using some standard tree–matching algorithm (TM).

Seeded Tree Basics

A seeded tree consist of *seed levels* and *grown levels*

The tree nodes at the seed levels are called the *seed levels* and those at the grown levels are called *grown nodes*

The Construction of the seeded tree consist of:
1–Seeding phase
2–Growing phase
3–Clean up phase

1– Seeding Phase

The seed levels of the seeded tress T_s are set up by copying over the top K levels of the R–tree T_R == *seeding tree*

The pointer fields of the level K–1 nodes are set to NULL
We call each node at level K–1 a *slot*, and the level K–1 *slot level*

The seed levels have the following properties: (Fig. 3)

1– the seed nodes at the slot level have null pointers but non–null bounding boxes

2– During tree construction, the bounding boxes in the seed nodes are used only to guide data insertion. (modified in *clean up phase*)

3– Node splitting at the grown levels never propagates upwards into the seed levels

Seed levels guide the growth of the tree (Fig. 4)

Three different strategies for *copying information* from the bounding box fields of seeding tree nodes:

C1: copy the Minimum Bounding Boxes

C2: copy the center points of the Minimum Bounding Boxes

C3: At the slot level, copy the center points of the Minimum Bounding Boxes. At other levels, the bounding box field contains the true minimum bounding box of its children

C2 and C3 almost always outperform C1

2– Growing Phase

In this phase, data objects from D_s are inserted into the seeded tree:

1– Traverse the tree from the root to the slot level, at each level choosing a suitable node to traverse from the next level till the lot level is reached.
2– If first child for slot node, a new grown node is created. Otherwise the data is inserted to an R–tree rooted at the slot node.

Node splitting does not propagate up to the seed levels.

The structure of the seed levels remains unchanged during the whole growing phase. (Fig. 2)

The bounding box fields of the traversed seed nodes are not always updated after each data insertion

3– Clean–up Phase

– The bounding box fields of the seed node are adjusted to be the true minimum bounding boxes of their children

– Slots containing no data objects are deleted and relevant data structures made consistent

Tree Construction Issues

1– Tree Construction Using Linked Lists

Traditional tree–like indices have been optimized for incremental updates rather than all–at–once construction.

Cost == buffer misses due to relative size of Tree and Buffer

- Avoid most random disk access due to buffer misses by forming intermediate linked lists under the slot. (Fig. 5)
- If a linked list page is full, we write all linked lists longer than a predefined constant (a batch) to disk.
- Construct the grown sub–trees one by one instead of all together. Grown sub–tree size is relatively smaller.

2– Packing Grown Sub–tree Roots

The large size of a seeded tree consumes more disk space and incurs more random disk I/O to a page in the tree during the matching phase.

Solution
Pack multiple grown sub–tree roots into one physical node. (Fig.7)

Experiments show that the seeded tree built with the packing algorithm almost never exceed the size of an R–tree built using the same data set, and provide stable performance regardless of the number of slots used.

3– Buffer Management During Seeded Tree Construction

4–

Write−purge is the action of writing the content of a buffer page to disk and freeing the buffer page by marking it unused.

Write−reflect is the action of writing the contents of a buffer page to disk and marking it clean. (It remains a used page)

Instead of delaying writing dirty pages to disk until space is needed, we write–reflect all dirty pages in linked lists and grown sub–trees at an early time, no random writes will be incurred during space reclamation.

If the write–reflects are done in large chunks, most pages will be written to disk in sequential I/O.

This will increase the total number of pages written to disk but the benefits of performing I/O sequentially will outweigh the penalties. (Table 1)

Theoretical Analysis

The number of sees levels (K) copied from seeding tree to the seeded tree is crucial to the performance of the seeded tree algorithm

Too Small   Big grown sub–trees

Too Large   The seed levels will occupy too much buffer space,
Too many grown sub–trees

R1 the size of the seed levels should

- Be smaller than the buffer size
- Occupy a small fraction of the buffer

R2 the average grown sub–tree size should

- Be smaller than the available buffer space to avoid random disk access during grown sub–tree construction
- Occupy a small fraction of the available buffer for good performance during actual join

$$\frac{B-\sqrt{B^2-4k}}{2} < n_1 < \frac{B}{E}$$

$$\frac{B-\sqrt{B^2-4k}}{2} < S < \frac{Bf_1}{E}$$

B: buffer size
K: seed level
N_1: number of nodes at level 1
E: >1 a tunable constant
S: number of slots

F_1: average fanout at level 1

Discussion and conclusions

- Seeded tree method is not just efficient but also stable
- New buffer management eliminates all random writes from the tree matching phase.
- It represents as much as 40% improvement in I/O costs over the old strategy.
- It runs 3 to 5 times faster R–Tree Join algorithm
- It uses a seeded tree and a pre–computed R–tree, but there might be situations where there in no pre–computed index for either input data set (*sampling techniques*)
- A seeded tree can be retained after join and used as an ordinary index