SPEEDING UP BULK-LOADING OF QUADTREES

GÍSLI R. HJALTASON
HANAN SAMET
YORAM J. SUSSMANN

COMPUTER SCIENCE DEPARTMENT AND
CENTER FOR AUTOMATION RESEARCH AND
INSTITUTE FOR ADVANCED COMPUTER STUDIES
UNIVERSITY OF MARYLAND

COLLEGE PARK, MARYLAND 20742-3411 USA

SPATIAL INDEX CONSTRUCTION

1. Typically a database system allows a user to designate the attributes for which an index is to be built
2. Query optimizers also have the ability to create indexes on un-indexed data or temporary results
   - Index construction can not take too much time as otherwise the operation could be executed more efficiently without the index
   - In other words, an index is not very useful if the execution time of the operation without the index is faster than the total time to execute it when the time to build the index is included
   - Indexes are used even though it takes long to build them when the indexed data is queried many times
     a. time to build the index is amortized over the number of queries before having to build a new index (on account of updates)
     b. assume database is relatively static

DYNAMIC DATABASES

- Often neglected issue in design of spatial databases
- Factors in choosing an index:
  1. speed to perform queries
  2. amount of storage required
- Emphasis on retrieval efficiency may lead to a wrong choice of an index when the operations are not limited to retrieval (e.g., creation of new data)

SPATIAL JOIN

- Problem: given river and road relations, find the locations where a river and a road meet (i.e., locations of bridges and tunnels)
- Solution: compute a spatial join of the two relations where the join condition results in extracting all tuples whose spatial attribute have at least one point in common
- Spatial join operation has both a relational and spatial component
  1. we don’t just want the names of the object pairs that satisfy the join condition (e.g., the names of the rivers and roads that intersect)
  2. we also want their actual locations so that they can serve as input to subsequent spatial operations (e.g., a cascaded spatial join as would be common in a spatial spreadsheet)
  3. implies need to construct a map for the output as well which means that we need a spatial index
  4. implies that the time to build the index plays an important role in the overall performance of the index in addition to the time needed to perform the spatial join itself whose output is not always required to be spatial
  5. most traditional studies on the efficiency of the spatial join only focus on the relational component of the output while very few include a spatial component in the output

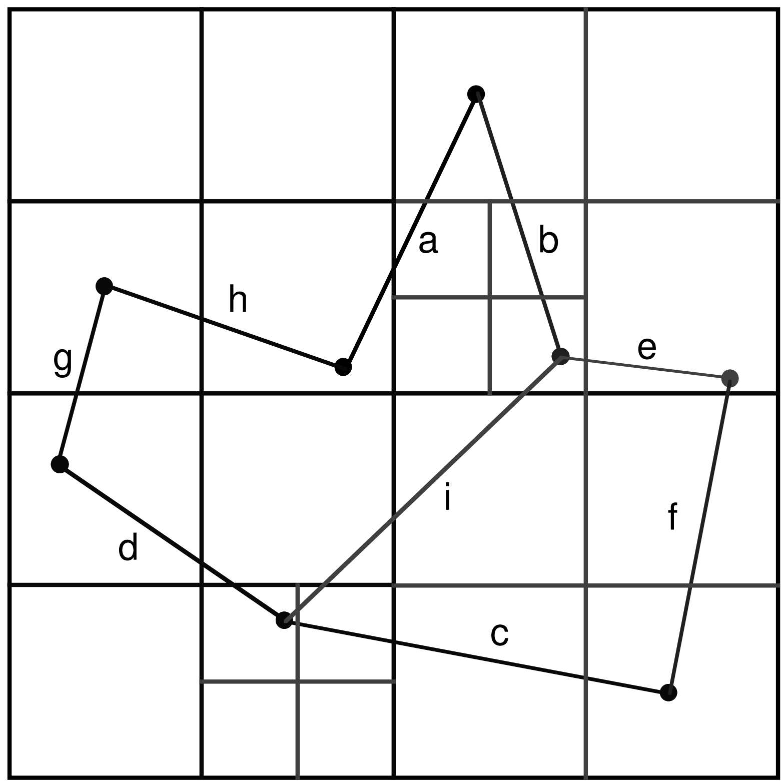
OVERVIEW

1. Techniques for speeding up construction of spatial indexes
2. Focus on PMR quadtree
3. Bulk-loading is the process of building a disk-based spatial index for an entire set of objects without any intervening queries
4. Strategy: fill up memory with as much of the quadtree as possible before writing some of its nodes to disk

PMR QUADTREE

- Edge-based
- Avoids having to split many times when two vertices or lines are very close as in PM1 quadtree
- Probabilistic splitting and merging rules
- Uses a splitting threshold value — say *N*

Ex: $N = 2$



DECOMPOSITION RULE:

Split a block *once* if upon insertion the number of segments intersecting a block exceeds *N*

Merge a block with its siblings if the total number of line segments intersecting them is less than *N*

- Merges can be performed more than once
- Does not guarantee that each block will contain at most *N* line segments
- Splitting threshold is not the same as bucket capacity
- Shape depends on order of insertion

QUADTREE IMPLEMENTATION

- Makes use of the Morton Block Index (MBI)
- MBI is a linear quadtree which is a collection of location codes for the leaf nodes of the quadtree
- Each location code is a pair of numbers
  1. size of the side of a block (or log of the size)
  2. bit-interleaved value of lower-left corner of block termed its *Morton code*
     - provide a mapping from *d* dimensions to one dimension
     - when points are sorted on the basis of their Morton codes, the result is known as Z-order
- MBI uses a B-tree to organize the locational codes (known as *Morton block values*) employing a lexicographic sorting order on the Morton code and the block side length
- A quadtree leaf node with *k* objects is represented *k* times in the B-tree, once for each object

BUILDING AN MBI

- Loading an MBI with a large number of objects simultaneously (i.e., bulk loading) was slow due to the cost of node splits
  1. when splitting a quadtree node, references to the objects must be deleted from the B-tree, and then reinserted with Morton block value identifiers of the newly created quadtree nodes
  2. deletions from the B-tree may cause merging of B-tree nodes, and subsequent reinsertions of the objects with their new Morton values will cause splitting of the same nodes thereby causing much disk activity
- Our approaches:
  1. attack problem on B-tree level
     - based on dramatically increasing the amount of buffering done by the B-tree (termed *B-tree buffering*)
  2. attack problem on quadtree level
     - based on reducing the number of accesses to the B-tree as much as possible by storing parts of the PMR quadtree in main memory (termed *quadtree buffering*)

B-TREE BUFFERING

1. Experiments with a buffer and an LRU (least recently used) node replacement policy
   - a node locking mechanism insures that nodes on the path from the root to the current node are not replaced
2. Existing implementation only buffered the B-tree nodes on the path from the root to the current node
   - found to be adequate for most applications
   - another policy (e.g., LRU) did not seem to be much benefit as most testing involved dynamic updates of isolated objects and other query types
   - problem was that for such use, even with LRU, most B-tree nodes were found to have been replaced by the time they are needed again as the quadtree blocks were accessed in a random manner
3. Try to increase the likelihood that the nodes where the insertions are taking place are already in the buffer
   - sort the objects prior to insertion in Z-order based on their centroid
     a. sorting is a small price to pay in comparison to cost of building a spatial index
     b. commonly used technique for static databases (e.g., Hilbert Packed R-trees)
   - tends to localize insertions within the B-tree nodes with the highest Morton block values

QUADTREE BUFFERING

- Key: build a pointer-based quadtree in memory thereby bypassing the costly updates of the B-tree of the Morton Block Index (MBI)
- Once entire quadtree has been built, output into the Morton Block Index (MBI) which resides on disk
- If memory is limited, then can't build the whole quadtree in memory
- Once exhaust available memory, invoke a node flushing algorithm that attempts to free up memory by writing parts of the memory resident pointer-based quadtree to disk
- Flushing algorithm uses a set of heuristics to decide what parts of the pointer-based quadtree to flush to disk
  1. goal: flush nodes of the tree that will not be needed later on for insertions
  2. impossible to satisfy for arbitrary insertion patterns
  3. presorting the data set in Z-order makes it possible to approach our goal
     - such an ordering tends localize the insertion activity
     - thus if a node has not been inserted into for a while, then it will probably not be inserted into again
- Flushing a quadtree node *n* to disk:
  1. insert the leaves in the subtree rooted at *n* into the Morton Block Index (MBI)
  2. free all nodes (and q-objects) in the subtree, except *n*, which is retained to indicate that the subtree is on disk

QUADTREE BUFFERING: FLUSHING NODES

- Choosing which part of the memory resident pointer-based quadtree to flush to disk is complicated by two conflicting goals:
  1. choosing a reasonably large part of the tree so as to avoid flushing too often
  2. not choosing too large a part of the tree so as not to flush to disk nodes whose region intersects many of the objects that are later inserted
- Node flushing algorithm makes use of following statistics, maintained for each node *n*, to decide which nodes in the pointer-based quadtree to flush
  1. time stamp for the last insertion into *n*
  2. total number of objects inserted into subtree rooted at *n* (regardless of flushing)
  3. number of q-objects currently present in subtree rooted at *n* (not counting q-objects in leaf nodes that have already been flushed)
  4. number of nodes in the subtree rooted at *n* excluding *n* (not counting nodes that already have been flushed)
- For each invocation of the flushing algorithm, the goal is to free a certain percentage *Q* of the nodes and q-objects in the pointer-based quadtree
  1. *Q* is termed *flushing quotient*
  2. experiments show $0.3 \leq Q \leq 0.6$ give good results

QUADTREE BUFFERING: FLUSHING ALGORITHM

- Node flushing algorithm is applied recursively starting at the root, say *n*, of the pointer-based quadtree
- Variables N_q and N_n indicate how many q-objects and nodes, respectively, are yet to be freed from the pointer-based quadtree by flushing parts of it to disk
- Algorithm:
  1. if *n* is a leaf node, then it is flushed
  2. if *n* is a non-leaf node, then
     - if number of q-objects and nodes in the subtree rooted at *n* is less than N_q and N_n, respectively, then flush *n*
     - otherwise, consider the unflushed child nodes of *n* in order of their last time of insertion and apply the flushing algorithm recursively to child nodes whose subtrees contain an adequate number of objects:
       a. the total number of objects that have been inserted in the child node is at least $1/2^d$ times the total number of objects inserted into *n*, OR
       b. the number of q-objects present in the subtree of the child node is at least $1/2^d$ times the number of q-objects present in the subtree rooted at *n*

QUADTREE BUFFERING: FLUSHING CRITERIA

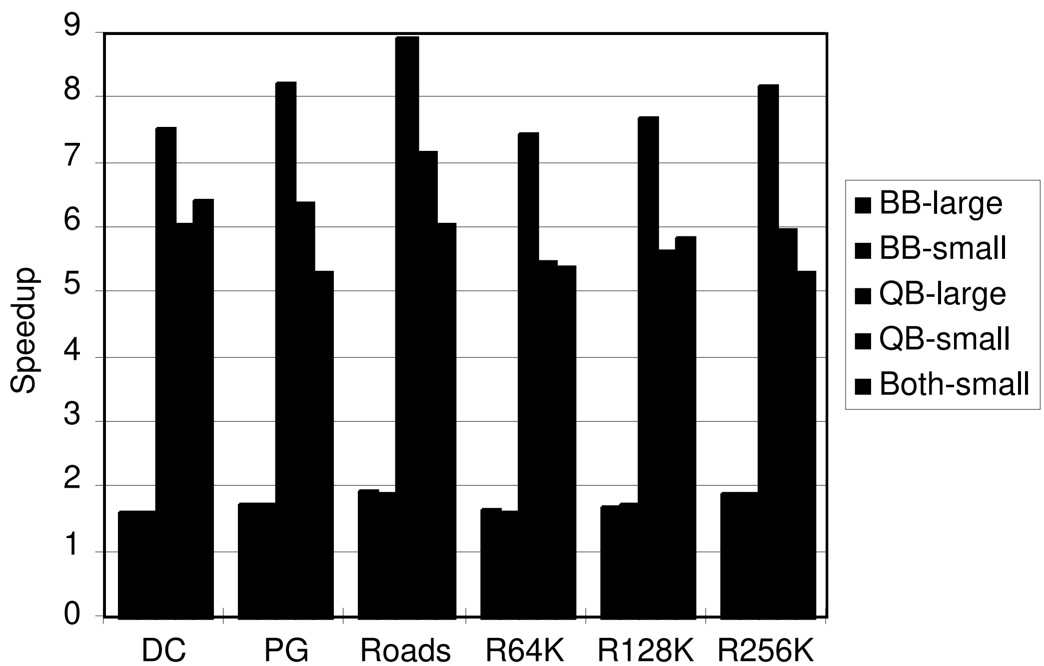
- Basing the decision process on both the total number of inserted objects and the q-objects present in the pointer-based tree results in a more stable overall performance
- Alternatives:
  1. if only use number of q-objects in subtree, then subtrees would be kept in memory too long
     - if all objects have already been inserted into child *n* of the root but number of q-objects in *n* is less than $1/2^d$ of the number of q-objects in the whole memory-based tree, then *n* stays in memory until almost all of the objects have been inserted
  2. if only use total number of inserted objects, then nothing may get flushed
     - if one of child nodes *n* of the root contains most of the inserted objects so that none of the other child nodes of the root satisfy the flushing criteria, and if *n* has already been flushed, then no node can be flushed (at least until more insertions take place)
     - ignoring nodes that have been flushed already is similar to just taking the q-objects into account

EXPERIMENTAL ENVIRONMENT

- TIGER/Line files
  1. Washington DC: 19,185 lines
  2. Prince George's County, MD: 59,551 lines
  3. roads in Washington DC metro area: 200,482 lines
  4. randomly generated data sets with 64K, 128K, and 256K lines
     - use random infinite lines
     - clipped to embedding space
     - subdivided at intersection points
- PMR quadtrees with splitting threshold of 8
- $2^{15} \times 2^{15}$ embedding space
- SUN SPARCstation 5 Model 70 rated at 60 SPECint92 and 47 SPECfp92 with 32MB memory

EXPERIMENT: SPEEDUP FOR SORTED INPUT

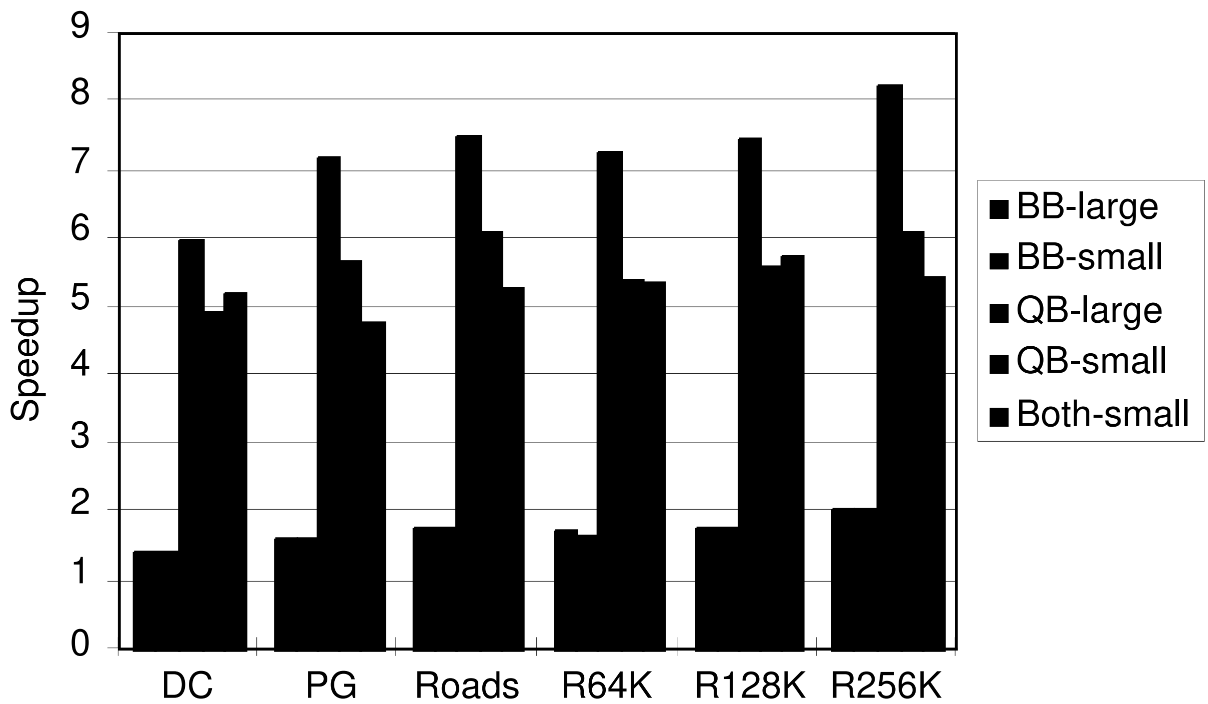
- Assume data is already sorted in Z-order and cost for sorting is not included
- Use B-tree buffering and/or quadtree buffering, with a large (so entire tree for all but largest data sets fits in memory) and small buffer size
  1. small B-tree buffer is 100 nodes occupying 400K
  2. small quadtree buffer is 100K
- Large buffer enables benchmark for maximum speedup achievable with buffering
- Compared with existing MBI method



- Conclusions:
  1. B-tree buffering provides modest speedup
  2. quadtree buffering provides dramatic speedup, almost one order of magnitude (factor of 7)
  3. not worth using both B-tree and quadtree buffering as the overhead of B-tree buffering may cancel some of benefit of quadtree buffering

EXPERIMENT: SPEEDUP FOR UN-SORTED INPUT

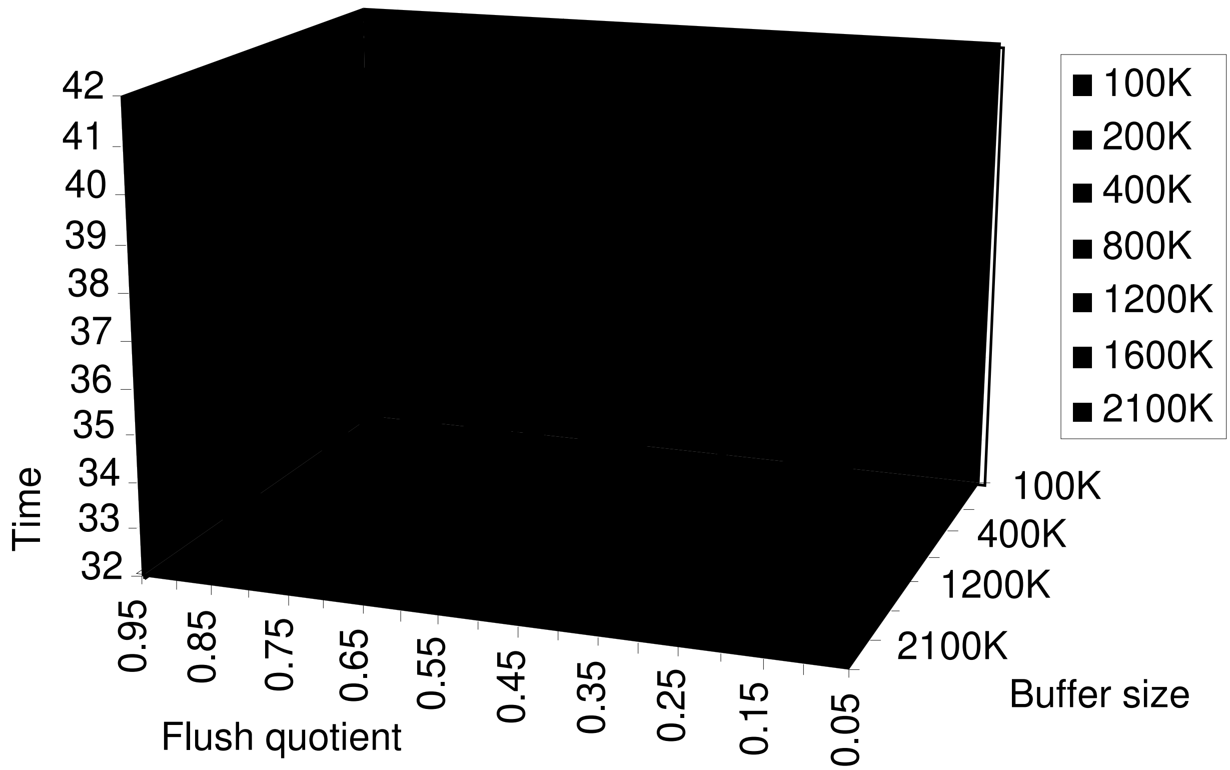
- Include cost of sorting the input before building the quadtree with buffering
- Use B-tree buffering and/or quadtree buffering, with a large (so entire tree for all but largest data sets fits in memory) and small buffer size
  1. small B-tree buffer is 100 nodes occupying 400K
  2. small quadtree buffer is 100K
- Large buffer enables benchmark for maximum speedup achievable with buffering
- Compared with existing MBI method



- Conclusion: same relative performance as when sorting time is not taken into account although speedup is off by less than 10%

EXPERIMENT: VARYING FLUSHING QUOTIENT

- Evaluate quadtree buffering using DC data set
- Vary flushing quotient from 0.05 to 0.95 and buffer size from 100K to 2100K
- Only plot up to 42 seconds (highest time: 70 sec.)



- Conclusions:
  1. no single value of the flushing quotient appears to be consistently the best for all buffer sizes
  2. execution time fluctuates somewhat for different values of the flushing quotient
  3. nevertheless, for values between .3 and .6, difference was only 4% for buffer size of 100K and 200K and 7% for other buffer sizes
  4. similar results with less fluctuation for other maps

EXPERIMENT: SPATIAL JOIN

- Ex: find intersections of roads (200,482) and rivers (37,495)
- Spatial join can be performed without any pre-existing spatial indexes, with only one index, or with two indexes
- Spatial join can be performed by building additional spatial indexes
- Results:
  1. using no buffering to build one new index prior to performing a join is slower than performing the join without indexes
  2. using B-tree buffering to build one new index prior to performing a join provides very small improvement over performing the join without indexes
  3. if an index exists for roads but not rivers, it is 4.6 times faster to build an index on rivers (71 seconds) using quadtree buffering and perform join with both data sets indexed (45 seconds more) than perform it with only the index on rivers (116 seconds vs. 533 seconds)
  4. if an index exists for rivers but not roads, the speedup was over 4 times (542 seconds vs. 2212 seconds) as building the roads index takes 497 seconds using quadtree buffering to which we add the cost of the join when both data sets are sorted which is 45 seconds
  5. if no index exists, then it is at least an order of magnitude faster to build new indexes using quadtree buffering (71+497 seconds) and then perform join (45 more seconds) than to perform a nested loop join (over 6000 seconds even when all data in memory)

CONCLUSIONS

- Quadtree buffering offers almost an order of magnitude speedup for building PMR quadtrees
- B-tree buffering provided a modest speedup, suggesting that insertions in a linear quadtree are highly CPU intensive
  1. bit manipulation on Morton Block values takes time
  2. avoided this cost by storing the values in the pointer-based structure when evaluating quadtree buffering
- Future work:
  1. investigate whether dynamic insertions and queries may be sped up through buffering
  2. exploit the faster building of spatial indexes by constructing a query processor that builds temporary spatial indexes when responding to queries