Supporting High Updates Disk-based Index in Road Network

Liangxu Liu [1, 2] Weimin Li [3] Yongming Guo [1] Jiajin Le [1]

[1] School of Computer Science and Technology, Donghua University. Shanghai, China

[2] School of Electric and Information, Ningbo University of Technology, Ningbo, China.

[3] School of Computer Engineering and Science , Shanghai University , Shanghai, China

luransh@gmail.com wmli@shu.edu.cn 1_mail@dhu.edu.cn lejiajin@dhu.edu.cn

Abstract

With the development of GPS and wireless techniques, more and more applications require maintaining the current position of moving objects on road network environment. It is the key stone that how to built an efficient index to supporting very high rates of spatial-index updates. Previous works in this domain assume that system holds enough main memory to buffer operations. This specialty enable these approaches don't work as main memories is unavailable. To buffer the operations without main memory, we propose a new R-tree-based indexing technique, called disk-based R*-tree (DBR*tree for short), which buffers update operations in disk page as well as grouping them to reduce disk I/O. DBR*-tree employs R*-tree to index the edges on the network (for example the road), not moving object, which eliminates the expensive costs caused by frequent changes in R*-tree structure while indexing moving objects by R*-tree. Finally, we present performance analysis and experimental results. And both of them show the proposed technique holds high performance on update and query operations.*

1. Introduction

Along with the development of GPS, communication, and wireless techniques, more and more applications are required to maintain the positions of a huge number of moving objects on the network, for example, in the context of intelligent transport system that maintains the location of a huge number of vehicles. In this case, the maximum objects throughput that can be tracked with a given accuracy is determined the efficiency of the updates. Therefore, how to support high update in index structure while maintaining query performance is a more important topic in spatiotemporal database. To improve the update efficiency, many methods have been proposed. The first approach in TPR-tree [1], TPR*-tree [2], B^x-tree [3], B_r^x-tree [4] is aimed to reduce the costs caused by frequent changes in R-tree structure. The second approach in RUM [5], LUGrid [6], LGU [7], R^R-tree[8] is to try to avoid performing deletion or insertion operation on disk, and buffer them in main memory or disk to process them in bulk. The third approach in IMORS [9], AU [10] is aiming to avoid the changes in R-tree structure by maintaining R-tree structure static.

The above approaches have their defeats: the first approach reduces the costs caused by frequent changes in R-tree structure to some extent, and it is focusing on reduce the costs of each update operation, its costs must be more than 2 (one for reading from the disk and the other for writing to the disk). The second approach hold efficient performance by processing operation in bulk, but they must be require the availability of enough main memories for buffering operation, and most of them couldn't work on few main memory environment. The third approach avoids the costs caused by efficient changes in index structure (R-tree), but its performance isn't enough efficient to be accepted by the pubic.

This paper explores a new index framework, called Disk Buffer R*-tree (DBR*-tree, for short), which supports high updates of moving objects on the network. Firstly, DBR*-tree employs R*-tree to index the edges (such as the road) on the network to obtain index structure static and support network-based query efficiently. Secondly, R*-tree holds a disk-based buffer (called R-Buffer) that buffering all updates receiving from the client, and each sub-entry of internal node holds a disk-based buffer (called I-Buffer) to buffer the entries maybe belong to this sub-node. In contrast to previous research on I/O efficient update of R-tree, DBR*-tree can process the operation in bulk while main memory is unavailable.

2. RELATED WORK

The first approach to indexing moving objects is reducing concurrency overheads during node splitting to index moving objects more efficiently. Such as, TPR-Tree [1] employs an MBR, which denotes its extent at reference time 0, and a VBR to reduce highly frequent alteration of R-Tree structure. TPR*-Tree [2] improves the TPR-Tree further by employing a new set of insertion and deletion algorithms that aim at minimizing node access cost. But both of them would lead the enlargement of dead overlaps in non-leaf node of R-tree. To avoid the costs caused by R-tree node splitting, B^x-tree [3], B_r^x-tree [4] employs B+-tree, not R-tree, to manage moving objects. B^x-tree adopts a transformation-based approach to index moving objects. Each objects position, modeled as linear function of time in 2D, is subjected to a transformation that uses a space-filling curve to map such function to point in 1D, and these resulting points are indexing by B^+-tree. Moreover, B^x-tree is relatively easy to

978-0-7695-3395-7/08 $25.00 © 2008 IEEE
DOI 10.1109/ICEBE.2008.39

integrate into an existing DBMS. B_r^x-tree, being the extent of B^x-tree, eliminates being sensitive to data skew.

The second approach is aiming to reduce the costs of index traversals in the R-tree. The so-called bottom-up approach aims to avoid the expensive tree traversals involved in deletion operation by offering direct leaf node access [11]. Another method (called group update in this paper) is maintaining deletion information until the pages that they reside are accessed and buffering insertion entries and then processing them in bulk to share same path I/Os. For example, through maintaining a main-memory structure, RUM-Tree [5] avoids immediate deletion of obsolete entries in R-Tree. LUGrid [6] is based on Grid File where grid cells have different size, which minimizes the I/O costs for updates by adopting Group Updates: (1) Group Insertion. Objects which belong to a same disk page are grouped together and are flush to disk in one time by memory grid. (2) Group Deletion. The deletion process is delayed until the pages where old entries reside are retrieved into memory by miss-deletion table. LGU [7] introduces two key additional structures to group similar updates in R-Tree. All incoming insertion entries are performed in a group batch fashion (Group Insertion) by a disk-based insertion buffer (I-Buffer) for each internal node, and Deletions are performed based on the main- memory lookup table. R^R-tree [8] employs two R-trees: a disk-based tree and an operation-buffer tree in main memory. All operations are buffered into operation buffer tree yet to be performed on the disk-based tree.

The third approach, which is used in the network environment, aims to maintain R-tree structure static to low update costs. It employs R-tree to index the edges, not moving objects. IMORS [9] employs index the edges to maximizing static part, and keep index structure static. J. Chen presents a new technique [10], called Adaptive Unit (AU for short), which group objects with similar movement pattern (position and velocity) to reduce update costs, obviously, it is efficient as most objects moves on fixed motion model. Moreover, both IMORS and AU performance weren't accepted by the public.

3. Problem Setting

Previous research on I/O efficient update of R-tree are mainly designed for Euclidean distance, they are some difficult to index moving objects on the network. Firstly, because the closer in Euclidean distance doesn't mean the closer in road network, the distance between two moving objects is computed by the length of shortest path between two objects unlike the straight line length. For example, suppose that a kNN (k=2) query Q as Figure 1. Objects *B*, *C* are the results of this query if the Euclidean distance is used. However, the correct results are objects *A*, *C*. Moreover, if R-tree is used to index moving objects directly, expensive cost caused by frequent changes of R-tree structure is unavoidable. (TPR-tree and TPR*-tree only reduce its costs to some extent, But the cost is that R-tree holds more and more dead overlap area in non-leaf node along with update processing). Based on the above reasons, this paper employs R*-tree to index the edge.

With the enlargement of the client functionality, it is feasible to assume that each update includes the last update position. Based on this, each update operation could divide into two operations: deleting old entry and inserting new entry. That is to say, if objID is object ID, (xp_{new}, yp_{new}) denotes the new position, and (xp_{old}, yp_{old}) is the old position, each update operation would insert two entries (<objID, xp_{new}, yp_{new}, ins> <objID, xp_{old}, yp_{old}, del>, where ins (or del) denotes that this entry is insertion (or deletion)) into R*-tree. Next, because the object position is 1 dimension on the network, that is to say, suppose that (x_st, y_st), (x_en, y_en) is two terminals of the edge that object locates, (x, y) is one object's position, we could use x or y to express this object position. To avoid the exception caused by the condition of x_st (or y_st) is equal to x_en (or y_en), the chosen method is that x (or y) is used to express the object position if |x_en-x_st| is more (or less) than |y_en-y_st|.

4. GTR-Tree Framework

This section explores the data structure, update and query processing of Disk Buffering R*-tree (DBR*-tree, for short), a new algorithm that employs group update technique based on disk.

4.1 Data Structure

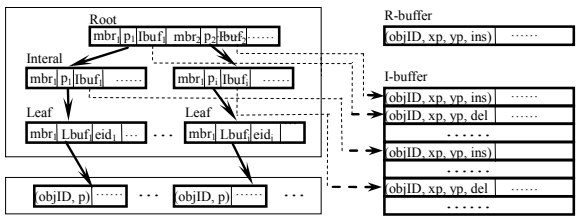

Figure 1: Data Structure of DBR*-tree

Besides employing R*-tree to index the edges on the network, DBR*-tree use two disk-based buffers (R-Buffer and I-Buffer) to improve the update throughout (Figure 1):

- R-buffer: A disk-based buffer which used to buffer all updates operations (insertion and deletion operations) that were submitted by the clients.
- I-buffer: A disk-based buffer that attached to each sub-entry of internal node, insertion and deletion operations were pushed from root to leaf in group through this I-buffer.

Then, sub-entry of internal node is expressed as 3-tuples: (mbr, p, Ibuf), where p is the pointer pointing to sub-node, and IBuf is page ID where I-Buffer locates. And leaf node is 3-tuples: (mbr, Lbuf, eid), where mbr is used to store two points of the edge, Lbuf is page ID where store moving objects that belong to this edge, and eid is edge ID. All buffer (R-Buffer, I-Buffer, and L-Buffer) are allocated dynamically; that is to say, if page amount of buffer is determined by actual amount of moving objects that buffer in it (the buffer is null if the amount is zero).

There are four noticeable points that need to describe in advance in DBR*-tree. The first is that because there are overlapping areas existing in R*-tree, one entries maybe flush into several sub-node, but invalid entries would remove by the following sub-node. The second is how to process that moving object locates at the intersection of several edges. In this case, DBR*-tree inserts the entry into each edge of intersection. The third is the entries in R-buffer and I-buffer is ordered by objID and first-in-later (the new entry is located at the front of existing entries as they holds same objID), and if I-buffer holds $obj_{(i, del)}$, $obj_{(i, ins)}$ with same (xp, yp), both of them are removed. The last is the different representation of position in leaf node. Each entry is expressed as <objID, p> pair, where p is x or y coordinate of the location.

```
Procedure Update (update_entry)
1 obj(i, ins),obj(i, del) are created and inserted into entryIns
2 entryDisk=ReadFromRBuffer();
3 entryRet=CombineEntry(entryIns, entryRet);
4 if entryRet exceeds max size of R-buffer
5    root_ptr. FlushBuffer(entryRet);
6 else
7    WriteToRBuffer (entryRet);
```

Algorithm 1: the Pseudo Code of Update Processing

4.2 Update Processing

This section discusses the updating operation in DBR*-tree (as Algorithm 1). Once the system receives an update, insertion entry $obj_{(i, ins)}$ and deletion entry $obj_{(i, del)}$ are created to form the array entryIns; the entries of R-buffer are read into the array entryDisk; and then combining entryIns and entryDisk to conduct the array entryRet (it is complemented by CombineEntry). If entryRet size exceeds max size of R-buffer, all entries in entryRet flush into the corresponding entries I-buffer of root node (it is complemented by FlushBuffer), otherwise entryRet is written back R-buffer.

```
Procedure CombineEntry (entryIns, entryDisk)
1 for each entry e_ins in entryIns do
2   for each entry  e_disk in entryDisk do
3       if e_ins.objID>e_disk.objID
4           break;
5   if the next entry of e_disk is opposite operation of e_ins
6      remove the next entry from entryDisk
7      continue;
8   insert e_ins into the position before e_disk;
9 return entryDisk;
```

Algorithm 2: the Pseudo Code of CombineEntry

CombineEntry complements that combine entryIns and entryDisk. It is called by Update and FlushBuffer. Algorithm 2 is the pseudo code of it. For each entry e_ins in the array entryIns, search the inserting position i in entryDisk that e_ins inserts, if opposite entry exists in entryDisk (The opposite entry is defined as the entry with the same (xpos, ypos) and objID, but with del flag value). If such opposite entry is found, the opposite entry is removed from entryDisk and incoming entry is ignored. Otherwise, e_ins inserts into position i of entryDisk. Finally, return entryDisk.

```
Procedure FlushEntry (entryBuf)
1 for each sub-node (mbr_i, p_i, Ibuf_i) in current node do
2   entrySubbuf = ObtainSubbuffer (entryBuf, mbr_i);
3   entryDisk = ReadFromIBuffer (Ibuf_i);
4   entryRet = CombineEntry (entrySubbuf, entryDisk);
5   if size(entryRet) exceed max size of I-Buffer
6      p_i ->FlushEntry(entryRet);
7   else
8      WriteToIBuffer(Ibuf_i);
```

Algorithm 3: the Pseudo Code of FlushEntry of interal node

```
Procedure FlushEntry (entryBuf)
1 for each sub-node (mbr_i, Lbuf_i, eid_i) in current node do
2   entrySubbuf = ObtainSubbuffer (entryBuf, mbr_i);
3   entryDisk = ReadFromLBuffer (Lbuf_i);
4   entryRet = CombineLeafEntry (entrySubbuf, entryDisk);
5   WriteToLBuffer(Lbuf_i);
```

Algorithm 4: the Pseudo Code of FlushEntry of leaf node

FlushEntry complements flushing entryRet into corresponding sub-node of current node. According that whether current node is leaf or not, it complements different function. As current node is internal node, FlushEntry complements the function as Algorithm 3. For each sub-node (mbr_i, p_i, $Ibuf_i$) of current node, entrySubbuf, which includes all entries that inside mbr_i, is obtained (ObtainSubbuf complement it), entryDisk is read from I-Buffer, and then they are combined to form entryRet (CombineEntry complements it). If the size of entryRet exceeds maximum size of I-Buffer, entryRet is flushed into sub-node (FlushEntry is called recursively), otherwise entryRet is written back I-Buffer. As current node is leaf node, FlushEntry complements the function as Algorithm 4. For each sub-node (mbr_i, $Lbuf_i$, eid_i) of current node, entrySubbuf, which includes all entries that locates at he edge, not inside mbr_i, is obtained (ObtainSubbuf complement it too), entryDisk is read from L-Buffer, and then they are combined to form entryRet (CombineLeafEntry complements it). Finally, entryRet is written back I-Buffer. ObtainSubBuffer complements the function obtaining sub-array entrySubbuf, which entries belong to current sub-node, from entryBuf. If current node is internal node, belonging to current sub-node means that entry is inside mbr_i , otherwise it means that entry locates at eid_i.

CombineLeafBuffer complements combining in leaf node. There are three exceptions. The first is only the first entry need to process if more than one entry with same objID exists in entryIns; the second is that we need process deletion and insertion entries with different method. If it is deletion entry, corresponding entry in entryDisk is removed; otherwise, if entryDisk holds corresponding entry, this corresponding entry will be replaced with current entry, else current entry inserts into entryDisk. The last is the entryIns entry is 4-tuple: (objID, xp, yp, isDel), but entryDisk entry is 2-tuple: (objID, p), not 4-tuple. The transformation needs to be done according to the property of current edge.

```
Procedure RTree::RangeQuery(qr)
1  entryBuf = ReadFromRBuffer();
2  entryRet= ObtainRangeEntry(entryBuf, qr)
3  ret=root_ptr->RangeQuery(entryRet, qr)
4  return ret;

Procedure RDirNode::RangeQuery(entryBuf, qr)
1  ret=NULL;
2  for each sub-node (mbr_i, p_i, Ibuf_i) of current node
3      if qr and mbr_i aren't overlap
4          continue;
5      entrySubbuf=ObtainSubBuffer(entryBuf, mbr_i)
6      entryDisk = ReadFromIBuffer(Ibuf_i);
7      entryDisk= ObtainRangeEntry(entryDisk, qr);
8      entryRet=CombineEntry(entrySubbuf, entryDisk);
9      ret+=p_i ->RangeQuery(entryRet, qr);
10 return ret;

Procedure RDataNode::RangeQuery(entryBuf, qr)
1  ret=NULL;
2  for each sub-node (mbr_i, Lbuf_i, eid) of current node
3      if qr and mbr_i aren't overlap
4          continue;
5      entrySubbuf=ObtainSubBuffer(entryBuf, mbr_i)
6      entryDisk = ReadFromLBuffer(Lbuf_i);
7      entryDisk= ObtainRangeEntry(entryDisk, qr);
8      entryRet=CombineLeafEntry(entrySubbuf,
entryDisk);
9      ret+=entryRet;
10 return ret;
```

Algorithm 5: Pseudo Code of RangeQuery

4.3 Query Processing

In query processing, we focus on the processing of range querying as it is one of the most important types of spatial queries. Since some current entries maybe exist in I-Buffer or R-Buffer, we must check them to delete obsolete entry and obtain current position of some entries that don't exist in L-Buffer. Algorithm 6 shows its pseudo code of range query. Given query range qr, the algorithms first checks R-Buffer to find entries inside qr to form entryBuf, and then check R-Tree to find results (RangeQuery of the root complements it). If root node is internal (or leaf), RDirNode:: RangeQuery (or RDirNode:: RangeQuery) is called. Their processing is similar to FlushEntry.

RDirNode:: RangeQuery complements range query in a internal node. The pseudo code is in Algorithm 5. For each sub-node (mbr_i, p_i, $Ibuf_i$), do following processions:

- If qr and mbr_i aren't overlap, skip this sub-node;
- Obtains entrySubbuf, which entries are inside mbr_i, from entryBuf;
- Reads entryDisk from I-Buffer and then removes the entries that aren't inside qr;
- Combines entrySubbuf and entryDisk to form entryRet (called CombineLeafEntry);
- Process range query in sub-node p_i, and add the results into ret;

Finally, returns ret.

RDataNode:: RangeQuery complements range query in a leaf node. The pseudo code is in Algorithm 6. For each sub-node (mbr_i, $Lbuf_i$, eid_i), do following processions:

- If qr and mbr_i aren't overlap, skip this sub-node;
- Obtains entrySubbuf, which entries are inside eid_i, from entryBuf;
- Reads entryDisk from L-Buffer and then removes the entries that aren't inside qr;
- Combines entrySubbuf and entryDisk to form entryRet (called CombineLeafEntry), and add the results into ret;

Finally, returns ret.

Table 1. Experimental Parameters.

Description	Values
amount of edges	7035
amount of objects	10K, 20K, …, **1M**
main memory(page)	**0**, 10, 20, …, 100
disk page size	**1K**, 2K, 4K
object moving speed	10, **50**, 250

5. Experimental Evaluation

In this section, we describe the experimental performance results of DBR*-tree, disk-based R^R-tree, and TPR*-tree. Disk-based R^R-tree is DBR*-tree without I-Buffer, which is similar as R^R-tree [8] with disk-based buffer, so in this paper, DBR*-tree without I-Buffer is called Disk-based R^R-tree (DR^R-tree, for short). Several parameters are examined to figure out their influences on the performance, which are the most important indexing methods for moving objects. All experimental data are created on a 2.4GHz P4 machine with 512M main memory. Owing to this lack of real spatiotemporal data, our performance study was based upon synthetic datasets created using a network-based data generator [12] and the real-world road networks of Oldenbourg (Figure 2). The main parameters used are shown in Table 1 and the default in bold. If no extra specification, the parameters are set to the default, update cost is the average number of four million times update, query cost is average of one thousand times range query with range of 1% of all area.

5.1 Effect of Buffer Size

In this new data structure, two kinds of disk-based buffers, I-Buffer and R-Buffer, is used to buffer all operations. At previous work, the buffer is based on main memory, and its costs neglect always, so its amount is the more, the better. However, it need more access cost in DBR*-tree if the buffer is based on disk. Next, we describe what the best amount of buffer page is. Figure 3 plots update costs of DBR*-tree and DR^R-tree with different buffer page amount (for simplification, the page amount in R-Buffer and I-Buffer is same). We can find that the update costs in DBR*-tree and DR^R-tree don't reduce along with the increasing of amount page of the buffer, but increase. Figure 4 shows the query costs of DBR*-tree and DRR-tree with different buffer page amount. Its result is similar as update costs. This reason is the more page amount could buffer more objects, but its costs increases proportionately. Therefore, in our experimental, the page amount of the buffer is always set to 1.

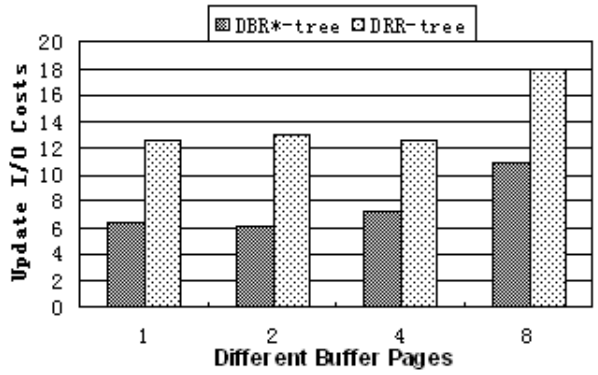

Figure 3: Update Costs with Different Buffer Size

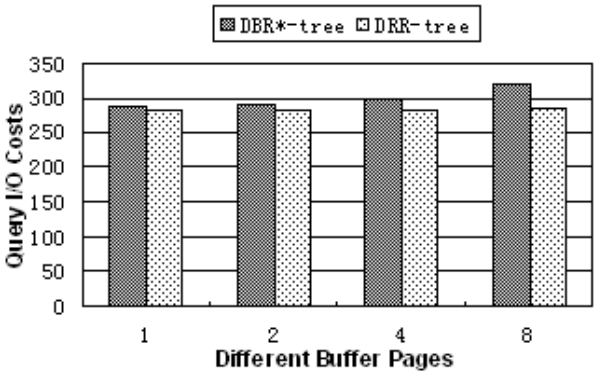

Figure 4: Query Costs with Different Buffer Size

5.2 Effect of Object Size

In this section, we compare update and query I/O costs in DBR*-tree, DR^R-tree, TPR*-tree with different objects amount (from 100K to 1M). Figure 5 shows the comparison graph of three algorithms in update I/O costs. With the increasing of objects amount, we can find that update costs in TPR*-tree increase quickly, this is because the larger objects amount is, the higher the tree is. However, DBR*-tree and DR^R-tree show comparable smooth along with the increase of objects amount. Moreover, DBR*-tree not only greatly outperformed TPR*tree in Update I/O cost, but also is more than DR^R-tree (about two times). Figure 6 plots experimental results of query costs with different objects amount (from 100K to 1M) in TPR*tree, DBR*-tree, and DR^R-tree. We can found that the query costs become larger quickly along with increase of objects because tree is higher. But DBR*-tree and DR^R-tree show good performance in query costs, the costs always keep under 500. The costs in DBR*-tree is a few larger than that of DR^R-tree, the reason is that I-Buffer must to be access in DBR*-tree. And comparing to its improvement in update performance, it is a trivial thing as the frequent of update is more than query. Therefore, in this case, the total performance of DBR*-tree outperforms DR^R-tree and TPR*-tree.

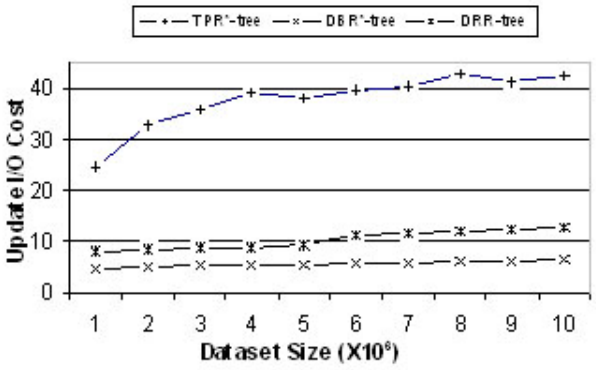

Figure 5: Update Costs with different Object Size

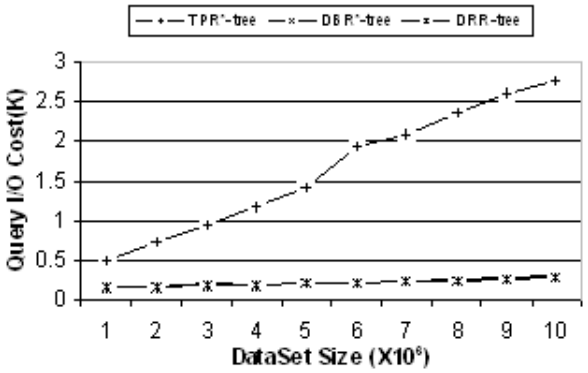

Figure 6:Query Costs with different Object Size

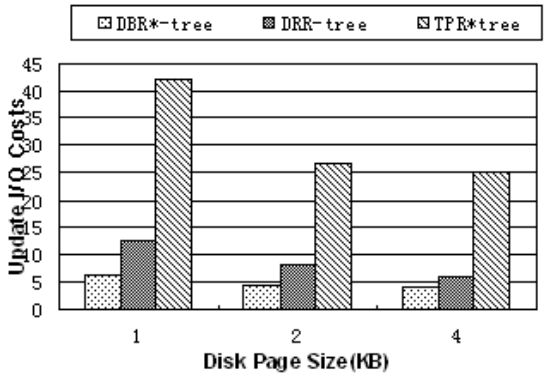

Figure 7: Update Costs with different Buffer Size

5.3 Effects of Different Page Size

Figure 7 plots update costs of three algorithms in different page size (1K, 2K, and 4K). The update costs reduce with the increasing of page size, but TPR*-tree reduces less than the others. The reason is that its costs still effects by R-tree structure's changing. As to the costs of query, as the reason describing in the above, it always fluctuates slightly. Figure 8 shows the maximum

values in DBR*-tree and DR^R-tree. We found that in most time DBR*-tree and DR^R-tree greatly outperforms TPR*-tree because it has no cost in R-tree structure changing. And DBR*-tree is slightly weaker than DR^R-tree in most time, the reason is DBR*-tree need access I-Buffer as internal node is accessing. Obviously, in this case, the total performance in DBR*-tree prior to the others.

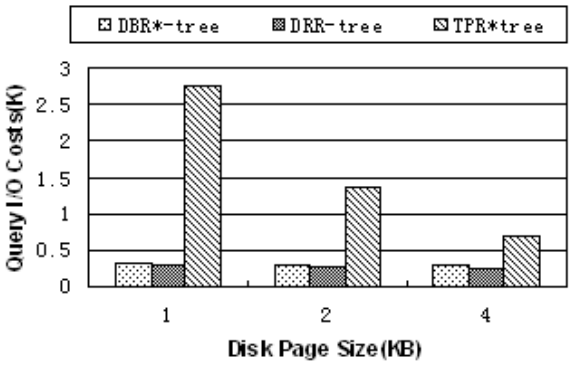

Figure 8: Query Costs with different Buffer Size

6. Conclusion

Aimed to indexing moving objects on the network efficiently as main memory is unavailable, this paper presents a novel index structures called Disk Buffer R*-tree, which holds more efficient performance in indexing the objects moving the network while no main memory is available. The main idea of DBR*-tree has three factors. Firstly, R-Buffer, a disk-based buffer that buffer all coming updates firstly, is employed, and each sub-entry of internal node holds a disk-based buffer (I-Buffer) that buffers all entries that maybe locates at its sub-tree; secondly, R-tree is used to index the edges enable R-tree structure stable and supporting the query based on network further; at final, the position of objects is expressed as 1D (x or y coordinate). The experimental results show DBR*-tree outperforms other approaches while no main memory is available.

7. REFERENCES

[1] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the Positions of Continuously Moving Objects. In Proc. ACM SIGMOD. 2000.

[2] Y. Tao, D. Papadias, J. Sun. 2003. The TPR*-Tree: An Optimized Spatiotemporal Access Method for Predictive Queries. In Proceedings of the 29th international conference on Very large data (Berlin, Germany, September 09 – 12, 2003). VLDB'03. VLDB Endowment, Salt Lake City, USA, 2003, 790 - 801.

[3] C. S. Jensen, D. Lin, and B. C. Ooi. 2004. Query and Updale Efficient B^+-Tree Based Indexing of Moving Objects. In Proceedings of the 30th international conference on Very large data (Toronto, Canada, August 29 - 30, 2004). VLDB'04. VLDB Endowment, Salt Lake City, UT, 2004, 768 - 779.

[4] Jensen, C.S., Tielsytye, D., Tradilauskas, N. Robust B+-Tree-Based Indexing of Moving Objects. In Proc. 7th International Mobile Data Management (MDM 2006) (Atlanta, USA, April 3 – 8, 2006) IEEE Computer Society, Washington, DC, 2006, 12.

[5] X. Xiong and W. G. Aref. 2006. R-trees with update memos.In proceeding of the 22nd International Conference on Data Engineering (Atlanta, USA, April 3 – 8, 2006). ICDE'06. IEEE Computer Society, Washington, DC, 2006, 22.

[6] X. Xiong, M. F. Mokbel, and W. G. Aref. 2006. LUGrid: update-tolerant grid-based indexing for moving objects. In procedding of the 7th International Conference on Mobile Data Management (Nara, Japan, May 9-13, 2006). MDM'06. IEEE Computer Society, Washington, DC, 2006, 13.

[7] B. Lin and J. Su. 2005. Handling frequent updates of moving objects. In proceeding of the 2005 ACM CIKM International Conference on Information and Knowledge Management (Bremen, Germany, October 31 - November 5, 2005.). ACM Press, New York, NY, 2005, 493-500.

[8] L. Biveinis, S. Saltenis, C. S. Jensen. 2007. Main-Memory Operation Buffering for Efficient R-Tree Update. In Proceedings of the 33th international conference on Very large data (Vienna, Austria, September 23-27, 2007). VLDB'07. VLDB Endowment, Salt Lake City, USA, 2007, 608–619.

[9] K. S. Kim, S. Kim, T. Kim, and K. Li. 2003. Fast indexing and updating method for moving objects on road networks. In Proceeding of the Fourth International Conference on Web Information Systems Engineering Workshops (Roma, Italy, December 13, 2003). 34 – 42.

[10] J. Chen, X. Meng, Y. Guo, X. Zhen. 2006. Update-efficient Indexing of Moving Objects in Road Networks. In Proceedings of the Third Workshop on Spatio-Temporal Database Management in conjunction with VLDB'06 (Seoul, Korea, September 11, 2006). VLDB-STDBM'06

[11] M.-L. Lee, W.Hsu, C. S. Jensen, B. Cui, and K. L. Teo. 2003. Supporting frequent updates in R-trees: a bottom-up approach. In Proceedings of the 29th international conference on Very large data (Berlin, Germany, September 09 – 12, 2003). VLDB'03. VLDB Endowment, Salt Lake City, USA, 2003, 608–619.

[12] Brinkhoff T. A framework for generating network based moving objects. Geoinformatica. 2(6):153-180. 2002.