An Efficient Index for Massive IOT Data in Cloud Environment

Youzhong Ma[1,2] Jia Rao[3] Weisong Hu[3] Xiaofeng Meng[1]
Xu Han[1] Yu Zhang[1] Yunpeng Chai[1] Chunqiu Liu[1]

[1]School of Information, Renmin University, Beijing, China
[2]Zhengzhou Chenggong University of Finance and Economics, Henan, China
[3]NEC Labs, China

[1]{mayouzhong,xfmeng,hanxumelody,zhangyu1990,ypchai,ae.liushuai}@ruc.edu.cn
[3]{rao_jia,hu_weisong}@nec.cn

ABSTRACT

The Internet of Things (IOT) has been widely applied in many fields, while the IOT data are always large volume, update frequently and inherently multi-dimensional, these characteristics bring big challenges to the traditional DBMSs. The traditional DBMSs have rich functionality and can deal with multi-attributes access efficiently, they can not scale good enough to deal with large volume data and can not support high insert throughput. The cloud-based database systems have good scalability, but they don't support multi-dimensional access natively.In order to deal with the large volume of IOT data, we propose an update and query efficient index framework (UQE-Index) based on key-value store that can support high insert throughput and provide efficient multi-dimensional query simultaneously.We implemented a prototype based on HBase and did comprehensive experiments to test our solution's scalability and efficiency.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Distributed databases; C.2.4 [**Distributed Systems**]: Distributed databases

General Terms

Algotithms,Experimentation,Performance

Keywords

Index, Internet of Things, Cloud

1. INTRODUCTION

With the development of RFID, sensor, GPS and other related techniques, "Internet of Things"(IOT) has been widely applied in many applications successfully and plays an important role in intelligent transportation system (ITS), environment monitoring system etc.

The IOT data is usually in large volume,update frequently and multi-dimensional, these characteristics bring us big challenges for massive IOT data management. For Instance, in the intelligent transportation system of a big city, supposing that there are one million GPS enabled vehicles, the vehicles emit one record every 30 seconds or 60 seconds, each record contains ID,CompanyID, VehicleSimID, GPSTime, GPSLongitude, GPSLatitude and other related attributes, each reord is 100 Bytes, then the total size of the data one day will be $100B * 10^6/min * 60/h * 24/day \approx 144G$. So the database management system driving the IOT applications should be able to deal with so large volume data and can support millions of data insert per minute.

The relational database management systems have been empowered with rich functionality by using K-d tree and R-tree indexes to support efficient multi-dimensional access, but RDBMS can't scale up well to deal with huge volume data and support millions of insert per minute. Key-value stores, such as BigTable[1] can support millions of updates per minute while providing fault tolerance and high availability, but they can't provide rich functionality and don't support multi-dimensional access natively. They can support efficient point and range queries on rowkey, but it has to scan the whole table for the queries on non-rowkeys. Although the MapReduce framework can be used to enhance concurrency and improve the query performance, full scan is still wasteful, especially for the high selective query.

In order to support both high insert throughput and efficient multi-dimensional query, we proposed an update and query efficient index framework (UQE-Index) based on the Key-value store, the Key-value store is used to store the data and to support high inset throughput, while the index is used to support efficient multi-dimensional query. The cost of the index creation and maintenance will be very high if we put the traditional index structures into Key-value store directly, so we consider the data as current data and historical data, and index them at different granularities, such solution can effectively reduce the index update times and decrease the index maintenance cost. We implemented a prototype based on HBase and performed comprehensive

CIKM'12, October 29–November 2, 2012, Maui, HI, USA.

experiments to test the performance of our solution. The main contributions of the paper are as follows:

1. We proposed an update and query efficient index framework based on Key-value store that can support both multi-dimensional query and high inset throughput.
2. We proposed a dynamic data partition strategy that can make sure that the data is evenly distributed into each region in HBase and the data that is close in time and space dimension is usually stored in the same regions.
3. We implemented a prototype based on HBase and performed detailed evaluations using uniform and skewed distribution data set.

The rest of the paper is organized as follows. In section 2 we review the related works. Section 3 describes the index framework and system architecture. Section 4 and 5 describe the data partition strategy and index implementation details. Section 6 gives the multi-dimensional range query algorithm. In section 7 we perform comprehensive experiments on our prototype. Section 8 concludes the paper.

2. RELATED WORK

The index techniques for cloud data management have been well studied in many literatures. Wu et al. [7] are the pioneers who began to do some research about the index for cloud data management. They firstly proposed a two level index framework in the cloud environment. According to the framework, the data is divided into several partitions and these partitions are distributed into different storage nodes, each local index is built on the local data node, and the global index is built by selecting portion of the local index nodes. In order to improve the query efficiency and eliminate the bottleneck of the centralized index paradigm, the computer nodes are organized into overlay networks such as CAN [5]. Based on the above index framework, Wang et al. [6] , Ding et al. [2] and Zhang et al. [8] proposed other different index solutions respectively. Wang et al. [6] built one R-tree to index the local data on each compute node, and organized the compute nodes into a CAN overlay network, the global index was constructed by selecting portion of the local R-tree index nodes to publish into the CAN overlay network. Zhang et al. [8] used the K-d tree for local data, and in the global index level he adopted the centralized index scheme by using R-tree to organize the portion of the local K-d tree nodes. Ding et al. [2] used MX-CIF quad tree as the local index and Chord overlay network as the global index.

Secondary index is another kind of index techniques for cloud data management. IHBase [3] is an open source project that provides transactional and indexing extension for HBase. CCIndex [9] is another kind of secondary index solutions based on Key-value store proposed by Zhou et al., in [9], one secondary index table was built for each indexed column. And in order to reduce the random read, the detailed information of each record was pushed into the secondary index table, so that the random read can be changed into sequential read. The author also proposed some optimization methods to support multi-dimensional query. CCIndex is easily to be implemented, but it has several drawbacks. Firstly it needs much more additional storage space when there are many indexed columns; secondly CCIndex does not support adding or removing index after the table was created.

MD-HBase was proposed by Shohi et al. [4]. In [4] the authors transformed the data from multi-dimensional space into one dimension by using linearization techniques such as Z-ordering and used the z-order value as the rowkey. In order to reduce the false positive scan during the query, the author divided the space into subspaces by using K-d tree and Quad-tree, and then constructed the index layer using the longest common prefix naming scheme.

3. SYSTEM OVERVIEW

3.1 System Architecture

Figure 1 displays the system architecture containing client, index cluster and HBase cluster. The client is mainly responsible for data inserting and sending query requests; HBase cluster acts as two roles, one is used to store the data, the other one is to index the historical data for each region in HBase; Index cluster is mainly responsible for indexing the time intervals, partitioning the space and indexing the subspaces. The data insert procedure is as the follows: when the new data arrives, it will be sent to HBase cluster and index cluster simultaneously, the HBase cluster is used to store the data into regions, and the index cluster is used to index the current data at a coarse granularity (time interval and subspace), such that we can separate the index creation and maintenance from data storage, and the index creation has little burden on the insert performance. After the current time ends, the index cluster will send a request to the HBase cluster to build local index for the data in each region at record level. When dealing with query, the first step is to get the related regions that the desired data resides by filtering through index cluster, and then getting the data from the HBase cluster according to the desired regions.

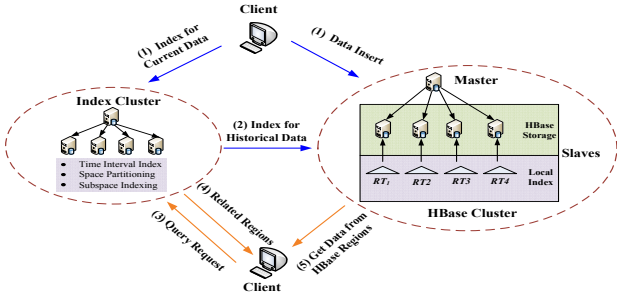


Figure 1: System Architecture

3.2 Index Framework

Figure 2 displays the index framework of UQE-Index, it contains three levels, the first two levels are time interval index and subspace index, they are coarse grained index and responsible for indexing the current data; the third level is local index that is a fine grained index and is used to index the historical data. The details are as the follows: Firstly we divide the time dimension into several time intervals, and these time intervals are indexed using B^+-tree; Secondly, for the data in a specific current time interval,we divide them into several subspaces in space dimension dynamically using

KD Tree or Quad Tree.The data of each subspace is stored into one region in HBase. Thirdly, when the current time interval ends, the following data starts from a new time interval, and the historical data that has been stored in HBase will not change any more, so for the historical data, we can build local index in batch using R-tree or Grid index. According to our UQE-Index approach, for the current data, we just index the time intervals and the subspaces, do not index the data itself, the index update times can be reduced effectively during the data inserting, so UQE-Index solution can support high update rates.

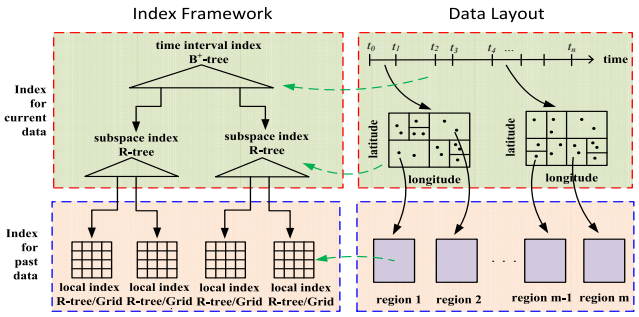


Figure 2: Index Framework

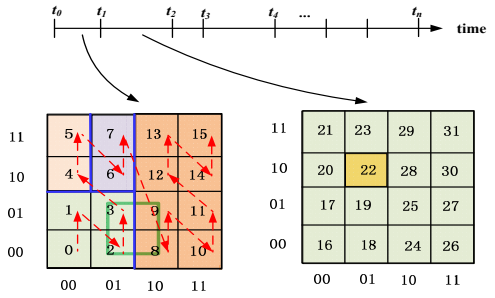


Figure 3: Z-ordering based space partitioning

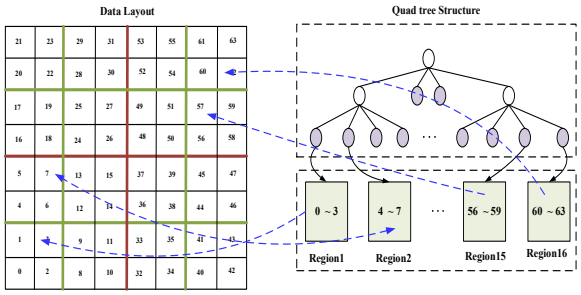


Figure 4: Pre-splitting for the subspaces

4. DATA PARTITIONING

4.1 Z-ordering Based Space Partitioning

In HBase the data is partitioned into several parts according to its rowkey, and in each region the rowkey is sequential. Based on this feature, we need to design a suitable rowkey generation scheme according to the spatio-temporal characteristics, and the rowkey must make sure that the data is generated randomly and distributed evenly in space dimension. So we adopted linearization technique (such as Z-ordering) to translate the multi-dimensional data into one dimension, the z-order value can be used as the rowkey to organize the data. The z-order value is computed as the following: $Z_{t_i} = Z_{loc} + i * N$, Z_{t_i} is the global z-order value in the i_{th} time interval; Z_{loc} is the local z-order value; i displays the i_{th} time interval; N is equal to the square of the z-order rank. For example, as shown in Figure 3, the local z-order code of the grayer region should be 6, and the global z-order code can be computed as $Z_{t_i} = Z_{loc}+1*(4*4) = 6+16 = 22$.

Z-order value can preserve the locality to some extent, but the data that is nearby according to the z-order code is perhaps not adjacent to each other in the original space, so if we query the data directly based on the range of the z-order value, we have to scan many false positive regions. Just as depicted in Figure 3, supposing that the green rectangle is the query range, the possible results are 2,3,8 and 9, while if we directly execute a range query based on the z-order value of the rectangle, we have to scan 2 to 9, actually 4 to 7 are false positives. So in order to eliminate the false positives, we adopt the bucket PR KD Tree to divide the subspaces based on the z-order value, and make sure that the z-order value is continuous in each subspace. For instance, during time interval [t0, t1), the space is divided into four subspaces, and the z-order value range are [0, 3], [4, 5], [6, 7], [8, 15], each subspace corresponds to one region in HBase. If the volume of one subspace exceeds the predefined threshold, the subspace should be divided again, and the corresponding region in HBase has to split.

4.2 Pre-splitting for the Subspaces

When dividing the data in spatial dimension using KD Tree or Quad Tree, if the volume of the data in one subspace exceeds the predefined value, the subspace should be divided again according to the partitioning strategy, at the same time, the region corresponding to the subspace also needs to be split and the data should be transferred accordingly. The dynamic partition strategy can guarantee the uniformity of the data distribution, but during the procedure of dividing the subspace and splitting the region, the related data has to be transferred and distributed again, it will cause additional cost and will affect the insert performance. In order to reduce such cost, we can divide the subspace at predefined level beforehand using the pre-splitting method, the details are described in Figure 4. We can use Quad Tree or KD Tree to divide the data into several subspaces, each subspace corresponds to one region in HBase, so we can pre-create some regions according to the subspaces, so at the beginning, the data will be directly distributed into some specific region based on the range of the rowkey and the region will not split. Until the volume of a subspace is over the threshold value, the region needs to split again.

5. INDEX IMPLEMENTATION DETAILS

5.1 B$^+$-tree Index for Time Intervals

Because the spatial distribution of the IOT data is changing with the time increasing, we divide the data into several time intervals (Time Interval Set) $TIS = \{[t_0, t_1), [t_1, t_2), ..., [t_{i-1}, t_i)\}$, $[t_{i-1}, t_i)$ is close at left and open at right, and these intervals are not overlapped with each other. In order to improve the query performance at the time dimension, we

use B$^+$-tree to index these time intervals. Each leaf node of the B$^+$-tree points to one R-tree, which is used to index the subspaces for the specific time interval. All the data that lies in $[t_{i-1}, t_i)$ are just indexed one time in time dimension.

5.2 R-tree Index for Subspaces

For the current data,we use Quad Tree to divide them into several subspaces, while after the current time interval ends, the data will not change any more, and the subspaces generated from the Quad Tree are not overlapped with each other, so we can use R-tree to index the subspaces. Each subspace is indexed only one time, so the index update times in space dimension is not very frequent. Each subspace corresponds to one region in HBase, when indexing the subspace, we also need to record the abstract information of the region, such as:$< [x_{min}, x_{max}), [y_{min}, y_{max}), [t_{min}, t_{max}), [startKey, endKey), regionIndex >$. $[x_{min}, x_{max})$ represents the longitude range, $[y_{min}, y_{max})$ represents latitude range, $[t_{min}, t_{max})$ represents time range, $[startKey, endKey)$ represents the rowkey range of the region and $regionIndex$ corresponds to the local index file name for the region.

5.3 Local Index for Data inside Region

We can build a local index for each historical region which is used to index every record in order to improve the query performance inside the region. Both R-tree index and grid index can be used as the local index structure. When querying from a region, firstly we can get the desired rowkeys by querying the corresponding local index, then obtain the actual data from HBase using the rowkeys.Because the volume of each region can't exceed the predefined threshold, we can consider the data as three dimensional data that comprised of time stamp, latitude and longitude, and then use R-tree to index the data. R-tree is a high balanced tree, so it just needs several times I/O to finish the query. On the other hand, the data distribution in each subspace is relative even after dividing the space using kd-tree or quad tree, for such kind of data, grid index can provide good query performance. In many cases, one specific query can be finished through two times I/O most, one is to get the grid directory and the other is to get the concrete data.

6. RANGE QUERY PROCESSING

Given a spatio-temporal range query $Q(E_t, E_s)$, E_t and E_s are time range and space range respectively. Firstly we can get the time intervals by querying time interval index for E_t(line:3); for each time interval, we can get the related regions that the query spatial range contains through the R-tree index for E_s(line: 4-6); line 7-19 is responsible for getting the desired data from the candidate regions: if the range of one region is subset of the query range (E_t, E_s), we can scan the region directly(line 8-9); otherwise if the overlap degree between (E_t, E_s) and the range of a region is over the predefined threshold, we also need to scan the whole region and filter the data using the query conditions(line10-15); otherwise we can get the desired rowkeys through the local index for the region, finally get the desired records form HBase using the rowkeys (line 17).

Algorithm 1 Range Query Processing

Input: $Q(E_t, E_s)$
Output: R_Q
1: $R_Q \leftarrow \varnothing$
2: $S_Q \leftarrow \varnothing$ /*Initialize the related region set to empty*/
3: $RT_Q \leftarrow getRtreeFile(B^+\text{-tree}, E_t)$
4: **for each** R-tree file RT in RT_Q **do**
5: $S_Q \cup getRegionsFromRtree(RT, E_s)$
6: **end for**
7: **for each** region R in S_Q **do**
8: **if** $abstract(R) \subseteq (E_t, E_s)$ **then**
9: $R_Q \cup scanRegion(R)$
10: **else if** $overlap(abstract(R), (E_t, E_s)) > \epsilon$ **then**
11: **for each** record r in R **do**
12: **if** $r \in (E_t, E_s)$ **then**
13: $R_Q \cup r$
14: **end if**
15: **end for**
16: **else**
17: $R_Q \cup searchWithRtree(R, E_t, E_s)$
18: **end if**
19: **end for**

7. EXPERIMENT EVALUATION

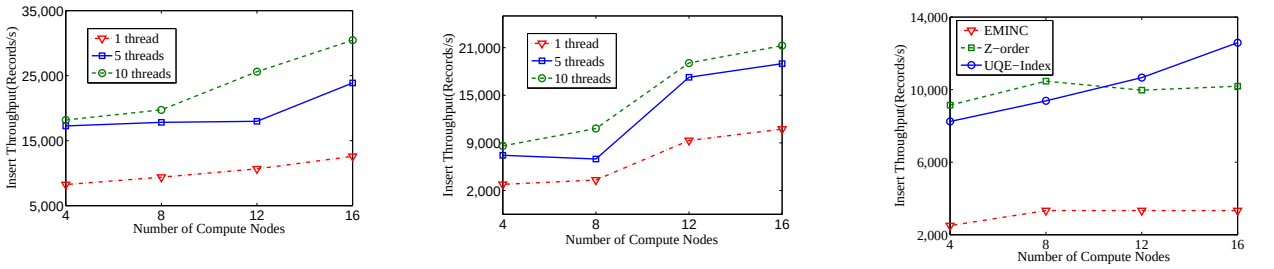
In this section we performance comprehensive evaluations of our prototype which is implemented based on HBase-0.20.6 and Hadoop-0.20.2, we have made some modifications about the internal functions of HBase. There are 16 computer nodes in the cluster which are connected with 1Gbit Ethernet switch, the configuration is: CPU: Q9650 3.00GHz, memory: 4GB, disk:500GB, os: 64bit Ubuntu9.10 server.

We evaluated the performance of our index solution using uniform and skewed distribution data sets, we generate 200 million records for each data set. Each record has nine attributes including ID,CompanyID, VehicleSimID, GPSTime, GPSLongitude, GPSLatitude, ConstantString and so on. For uniform distribution data set,GPSLongitude and GPSLatitude are uniformly distributed in the range [1, 10000], while for skewed distribution data set they are generated following *zipf* like distribution, and the skew factor is set as 0.5. The GPSTime is the system time when the data is generated, ConstantString is used to tune the record size, and other attributes are generated randomly.

7.1 Performance of Insert Throughput

In this section we evaluate the performance of insert. Figure 5 describes the insert throughput of the system for uniform and skewed data, the system can reach 30000 records/s for the uniform data and 20000 records/s for the skewed data. The insert performance for the uniform data is better than that of the skewed data, because we divide the data in space dimension using KD-tree, if the volume of a given subspace exceeds the predefined threshold, the subspace has to be divided into two child subspaces, at the same time, the data in the original subspace also needs to be re-distributed into corresponding child subspace. For the skewed data, the subspaces are divided more frequently, so the cost will be higher and the performance of insert will be decreased.

Figure 5-(c) shows that the performance of UQE-Index is much better than that of EMINC, in EMINC, every record needs to be indexed in KD-Tree, so the index creation cost will bring heavy burden on the insert performance. Because we pre-split the regions in advance, the system will have higher concurrency, the performance of UQE-Index is comparable to that of Z-order.



(a) Uniform Data Set (b) Skewed Data Set (c) Insert Throughput Comparison

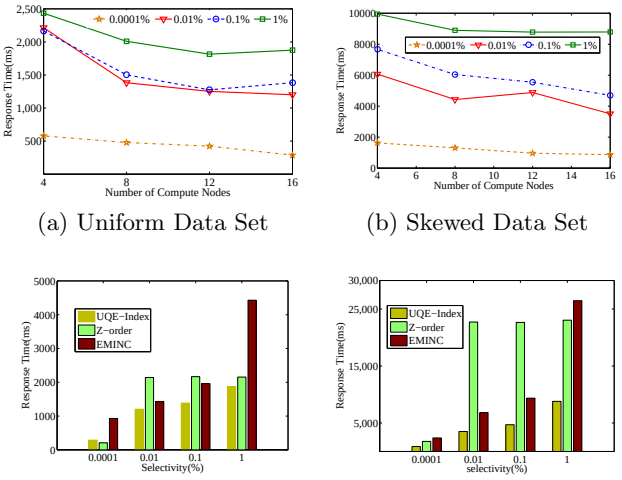
Figure 5: The Performance of Insert Throughput

7.2 Performance of Range Queries

In this section, we evaluate the performance of the range queries of UQE-Index,EMINC[8] and Z-order methods.

Figure 6-(a) and (b) display the scalability of UQE-Index for the range queries with different selectivity. The UQE-Index has good scalability both for uniform and skewed distribution data set, while response time for skewed data set is much higher than that for uniform data set, because more regions need to be scanned for skewed data set.

Figure 6-(c) and (d) compare the performance of range queries among UQE-Index, EMINC and Z-order, UQE-Index has the best performance. For the queries with high selectivity, the performance of Z-order is the worst,because Z-order doesn't have any index structure, it needs to scan many false positive regions. As the selectivity decreases, the response time of UQE-Index and EMINC also increase, but UQE-Index still has the lowest response time.



(a) Uniform Data Set (b) Skewed Data Set

(c) Comparison for Uniform Data Set (d) Comparison for Skewed Data Set

Figure 6: The Performance of Range Queries

8. CONCLUSIONS AND FUTURE WORK

In this paper, We proposed an update and query efficient index framework (UQE-Index) based on the Key-value store to deal with massive IOT data. We also implemented a prototype based on HBase, and comprehensive experiment evaluations have been done to analyze our solution's efficiency and scalability. In the future, we will extend our works to support other query, e.g., KNN query and Skyline query.

9. ACKNOWLEDGMENTS

This work was partially supported by NEC (China) Co., Ltd, the Natural Science Foundation of China (No. 91024032, 61070055), the Research Funds of Renmin University of China (No. 10XNI018), National Science and Technology Major Project of Key Electronic Devices, High-end General-purpose Chips and Fundamental Software Products (No. 2010ZX01042-002-003).

10. REFERENCES

[1] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 2008.

[2] L. Ding, B. Qiao, G. Wang, and C. Chen. An efficient quad-tree based index structure for cloud data management. In *WAIM'11*, pages 238–250, 2011.

[3] Y. Kulbak and D. Washusen. Ihbase. *http://github.com/ykulbak/ihbase*, 2010.

[4] S. Nishimura, S. Das, D. Agrawal, and A. E. Abbadi. Md-hbase: A scalable multi-dimensional data infrastructure for location aware services. In *Mobile Data Management (1)'11*, pages 7–16, 2011.

[5] S. Ratnasamy, P. Francis, M. Handley, R. M. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM'01*, pages 161–172, 2001.

[6] J. Wang, S. Wu, H. Gao, J. Li, and B. C. Ooi. Indexing multi-dimensional data in a cloud system. In *SIGMOD Conference'10*, pages 591–602, 2010.

[7] S. Wu and K.-L. Wu. An indexing framework for efficient retrieval on the cloud. *IEEE Data Eng. Bull.*, pages 75–82, 2009.

[8] X. Zhang, J. Ai, Z. Wang, J. Lu, and X. Meng. An efficient multi-dimensional index for cloud data management. In *CloudDB'09*, pages 17–24, 2009.

[9] Y. Zou, J. Liu, S. Wang, L. Zha, and Z. Xu. Ccindex: A complemental clustering index on distributed ordered tables for multi-dimensional range queries. In *NPC'10*, pages 247–261, 2010.