Parallel Real-Time OLAP on Multi-Core Processors

Frank Dehne, School of Computer Science, Carleton University, Ottawa, Canada

Hamidreza Zaboli, School of Computer Science, Carleton University, Ottawa, Canada

ABSTRACT

One of the most powerful and prominent technologies for knowledge discovery in decision support systems is online analytical processing (OLAP). Most of the traditional OLAP research, and most of the commercial systems, follow the static data cube approach proposed by Gray et.al. and materialize all or a subset of the cuboids of the data cube in order to ensure adequate query performance. Practitioners have called for some time for a real-time OLAP approach where the OLAP system gets updated instantaneously as new data arrives and always provides an up-to-date data warehouse for the decision support process. However, a major problem for real-time OLAP is the significant performance issues with large scale data warehouses. The aim of our research is to address these problems through the use of efficient parallel computing methods. In this paper, we present a parallel real-time OLAP system for multi-core processors. To our knowledge, this is the first real-time OLAP system that has been parallelized and optimized for contemporary multi-core architectures. Our system allows for multiple insert and multiple query transactions to be executed in parallel and in real-time. We evaluated our method for a multitude of scenarios (different ratios of insert and query transactions, query transactions with different amounts of data aggregation, different database sizes, etc.), using the TPCDS "Decision Support" benchmark data set. As multi-core test platforms, we used an Intel Sandy Bridge processor with 4 cores (8 hardware supported threads) and an Intel Xeon Westmere processor with 20 cores (40 hardware supported threads). The tests demonstrate that, with increasing number of processor cores, our parallel system achieves close to linear speedup in transaction response time and transaction throughput. On the 20 core architecture we achieved, for a 100 GB database, a better than 0.25 second query response time for real-time OLAP queries that aggregate 25% of the database. Since hardware performance improvements are currently, and in the foreseeable future, achieved not by faster processors but by increasing the number of processor cores, our new parallel real-time OLAP method has the potential to enable OLAP systems that operate in real-time on large databases.

Keywords: *Index Structures, Multi-Core OLAP, Parallel, Processors, Real-Time*

DOI: 10.4018/ijdwm.2015010102

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

1. INTRODUCTION

This paper reports on the results of an IBM funded research project to investigate the use of multi-core processors for high performance, real-time, online analytical processing (OLAP). Such OLAP systems are at the heart of many business analytics applications. The ever growing data warehouses built by corporate and institutional users have lead to significant performance bottlenecks which motivated this research project.

1.1. Background

Decision Support Systems (DSS) are designed to empower the user with the ability to make effective decisions regarding both the current and future state of an organization. To do so, the DSS must not only encapsulate static information, but it must also allow for the extraction of patterns and trends that would not be immediately obvious. Users must be able to visualize the relationships between such things as customers, vendors, products, inventory, geography, and sales. Moreover, they must understand these relationships in a chronological context since it is the time element that ultimately gives meaning to the observations that are formed. One of the most powerful and prominent technologies for knowledge discovery in DSS environments is online analytical processing (OLAP).

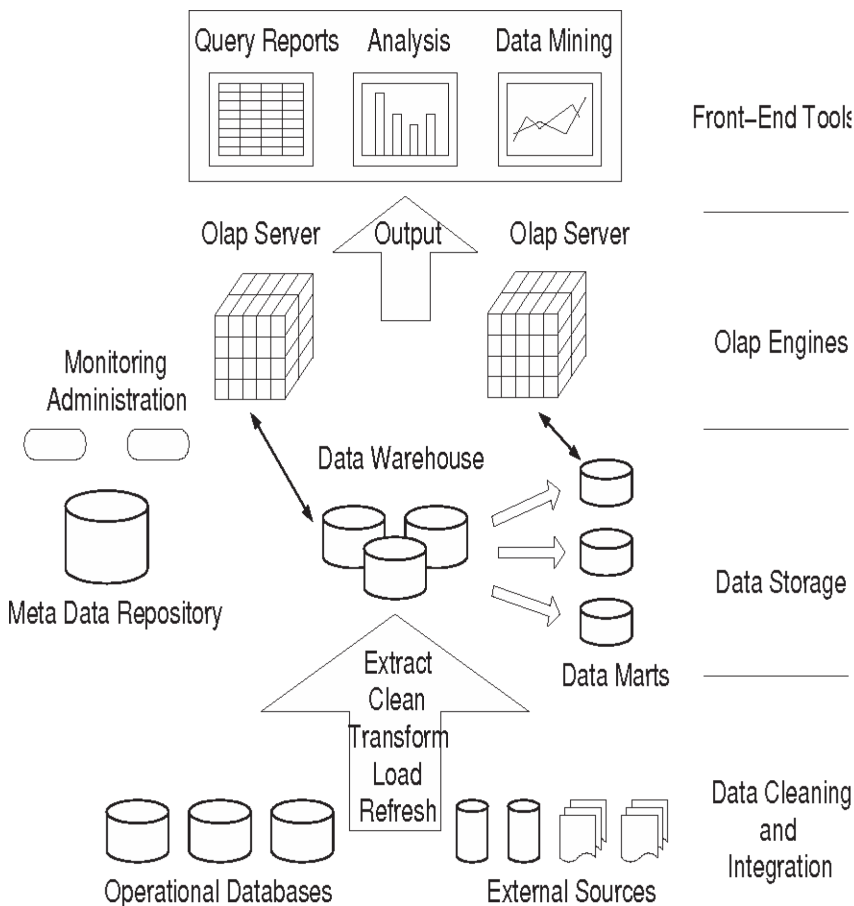
OLAP is the foundation for a wide range of essential business applications, including sales and marketing analysis, planning, budgeting, and performance measurement (Han, 2000 & The OLAP Report). The processing logic associated with this form of analysis is encapsulated in what is known as the OLAP server. By exploiting multidimensional views of the underlying data warehouse, the OLAP server allows users to “drill down” or “roll up” on hierarchies, “slice and dice” particular attributes, or perform various statistical operations such as ranking and forecasting. Figure 1 illustrates the basic model where the OLAP server represents the interface between the data warehouse proper and the reporting and display applications available to end users.

To support this functionality, OLAP relies heavily upon a classical data model known as the data cube (Gray, 1997). Conceptually, the data cube allows users to view organizational data from different perspectives and at a variety of summarization levels. It consists of the base cuboid, the finest granularity view containing the full complement of *d* dimensions (or attributes), surrounded by a collection of 2^{d-1} sub-cubes/cuboids that represent the aggregation of the base cuboid along one or more dimensions. Figure 2 illustrates a small four-dimensional data cube that might be associated with the automotive industry. In addition to the base cuboid, one can see a number of various planes and points that represent aggregation at coarser granularity. Note that each cell in the cube structure corresponds to an aggregate value along one or more measure attributes (e.g. total sales).

Most of the traditional OLAP research, and most of the commercial systems, follow the static data cube approach proposed by Gray (1997) and materialize all or a subset of the cuboids of the data cube in order to ensure adequate query performance. Building the data cube can be a massive computational task, and significant research has been published on sequential and parallel data cube construction methods (e.g. (Chen, 2008 & Dehne, 2002 & Gray, 1997 & GuoLiang, 2010 & Ng, 2001 & You, 2008)). However, the traditional static data cube approach has several disadvantages. The OLAP system can only be updated periodically and in batches, e.g. once every week. Hence, latest information cannot be included in the decision support process. The static data cube also requires massive amounts of memory space and leads to a duplicate data repository that is separate from the online transaction processing (OLTP) system of the organization. Several practitioners have therefore called for some time for an integrated OLAP/OLTP approach with a real-time OLAP system that gets updated instantaneously as new data arrives and always provides an up-to-date data warehouse for

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 1. Three-tiered OLAP model



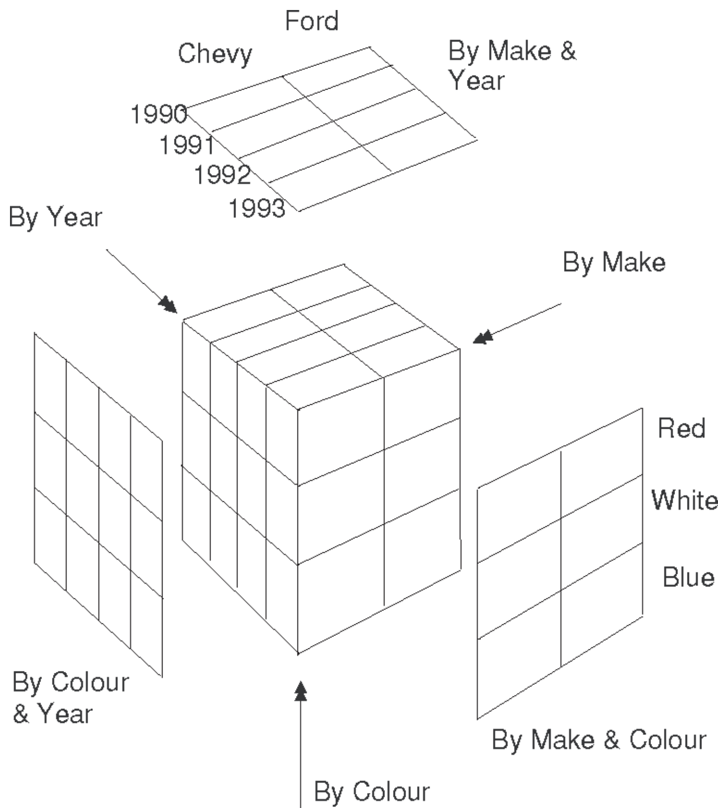
the decision support process (e.g. (Bruckner, 2002)). Some recent publications have tried to address this problem by providing "quasi real-time" incremental maintenance schemes and loading procedures for static data cubes (e.g. (Bruckner, 2002 & Jin, 2008 & Santos, 2008 & Santos, 2009)). However, these approaches are not fully real-time. A major problem is significant performance issues with large scale data warehouses. The aim of our research is to address these performance problems through the use of efficient parallel multi-core computing methods.

1.2. Contributions

In this paper, we present a parallel real-time OLAP system for multi-core processors. To our knowledge, this is the first real-time OLAP system that has been parallelized and optimized for contemporary multi-core processors. Our system is an in-memory data management system for OLAP (Plattner, 2012) that allows for multiple insert and multiple query operations to be executed in parallel and in real-time. It is based on a new parallel data structure termed PDC-tree. The basic mechanism is outlined in Figure 3. In order to process an input stream of OLAP insert and OLAP query transactions in real-time, our PDC-tree data structure allows for

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 2. A three dimensional data cube for automobile sales data



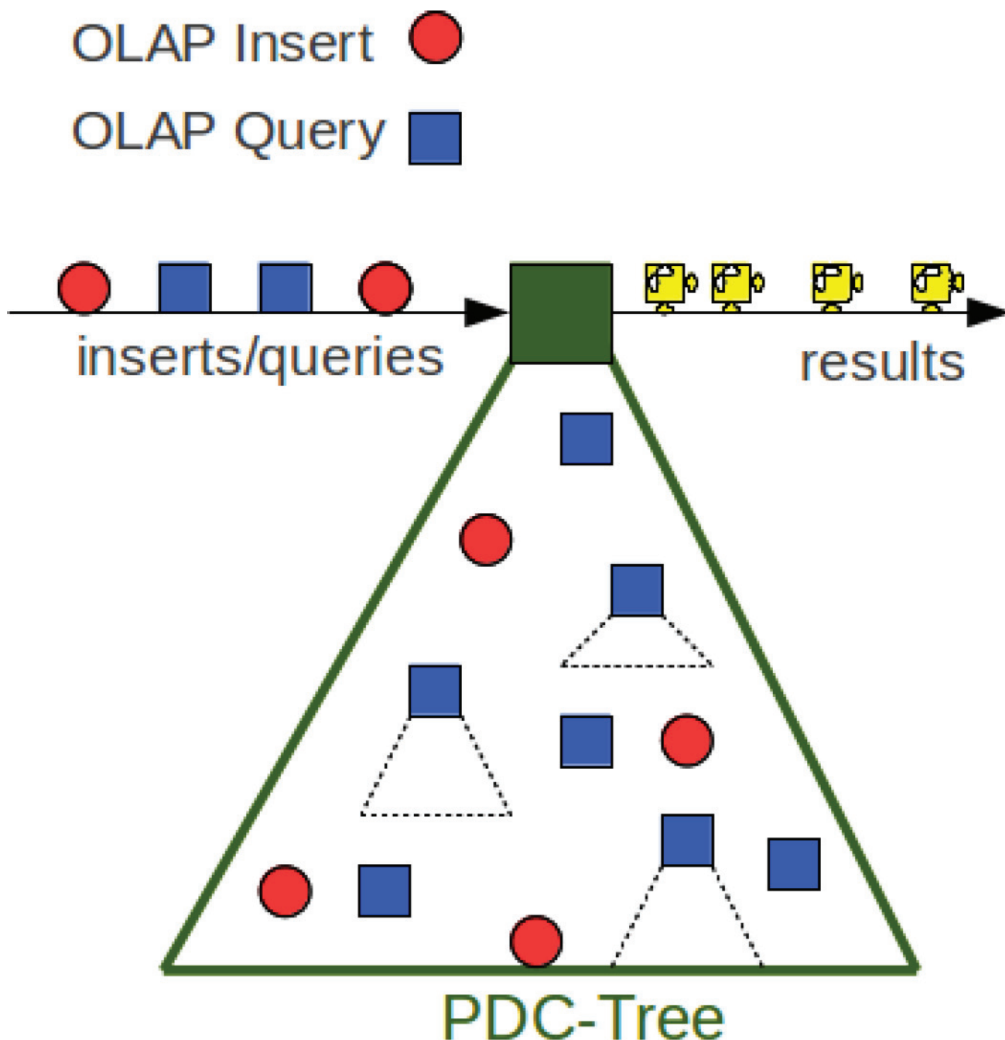
the concurrent execution of these transactions through multiple threads executed in parallel on a multi-core processor. Here we employ two types of parallelism: executing multiple OLAP transactions in parallel and further speeding up individual OLAP transactions by assigning each of them multiple parallel execution threads.

The main challenge is the possible interference between parallel insert and query transactions, as well as between parallel inserts operating on the same portion of the tree data structure. For example, each OLAP query has to include all data from prior OLAP insert transactions, including those recent insert transactions that are not yet completed and are being executed in parallel with current OLAP query transactions. A straightforward solution would e.g. lock subtrees on which an insert is being performed. This would however lead to significant wait times for other queries and result in a method where the performance does not scale with increasing number of processor cores. Our main contribution is the design of a minimal locking scheme which allows concurrent insert and query transactions to move "freely" and which detects and recovers transactions working on invalid or incomplete data.

We evaluated our method for a multitude of scenarios (different ratios of insert and query transactions, query transactions with different amounts of data aggregation, different database sizes, etc.), using the TPCDS "Decision Support" benchmark data set. As multi-core test platforms, we used an Intel Sandy Bridge processor with 4 cores (8 hardware supported threads) and an Intel Xeon Westmere processor with 20 cores (40 hardware supported threads). The tests demonstrate that our parallel real-time

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 3. A PDC-tree: In order to process an input stream of OLAP insert and OLAP query transactions in real-time, we provide speedup through concurrent execution of transactions on a multi-core processor



OLAP system scales with increasing number of processor cores and achieves a close to linear speedup in transaction response time and throughput on contemporary multi-core processors. On the 20 core architecture we achieved a better than 0.25 second query response time for real-time OLAP queries that aggregate 25% of a 100 GB database. Since, for the foreseeable future, hardware performance improvements are achieved not by faster processors but by increasing the number of processor cores, our new parallel real-time OLAP method has the potential to enable OLAP systems that operate in real-time on large databases.

The remainder of this paper is organized as follows. In Section "Review: Multi-Dimensional Index Structures for Real-Time OLAP on Multi-core Processors", we review related previous results and in Section "A Parallel DC-Tree (PDC-Tree) Data Structure for Parallel

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Real-Time OLAP", we present our parallel algorithms for real-time OLAP on multi-core processors. In Section "Experimental Evaluation", we analyze the performance of our method on two contemporary multi-core platforms and Section "Conclusions" concludes our paper.

2. REVIEW: MULTI-DIMENSIONAL INDEX STRUCTURES FOR REAL-TIME OLAP ON MULTI-CORE PROCESSORS

Building a parallel real-time OLAP data warehouse is related but considerably more complex than concurrent updates and searches in general OLTP databases which have been studied since the 90's for example in (Banks, 1995 & Chakrabarti, 1999) and more recently in (Haritsa, 2000 & Lee, 2003). One major difference is that OLAP queries may need to aggregate large portions of the database whereas OLTP transactions are more local in nature. Concurrent operations in spatial databases have recently been studied e.g. in Dai (2009). Whereas such databases can process range queries over large portions of the DB, spatial database index structures are generally not efficient for the large number of dimensions typically required for OLAP. It is not unusual for OLAP systems to process data with 10, 15 or more dimensions. Another important difference are the elaborate dimension hierarchies which are typical for OLAP systems. To our knowledge, the only published fully dynamic data structure for OLAP queries on data cubes is the DC-tree introduced by Kriegel et.al. (Ester, 2000), which is a sequential tree based index structure specifically designed for data warehouses with dimension hierarchies. An extension of the DC-tree to handle special properties of the time dimension was recently published in (Ahmed, 2010). The DC-tree (Ester, 2000) extends the X-tree (Berchtold, 1996) and R-tree (Guttman, 1984) data structures for multidimensional data indexing.

An OLAP database consists of several functional attributes, grouped into dimensions, and some dependent attributes, called measures. For dimensions with more than one functional attribute, these attributes are organized into hierarchy schemas. For example, the dimension customer can have functional attributes region, nation, customer ID. A DC-tree builds a partial ordering and concept hierarchy for each dimension. A concept hierarchy is a tree structure storing for a given dimension all values that occur in the DC-tree at a given time. Using the partial ordering defined by the concept hierarchy for each dimension, the DC tree extends the X-tree (Berchtold, 1996) and R-tree (Guttman, 1984) by replacing the standard minimum bounding rectangles (MBR) assigned to directory nodes by minimum describing rectangles (MDS). An MDS contains for each dimension a set of values at different levels of the dimension hierarchy, and describes a set of hyper rectangles which together contain the data stored in the respective subtree. The rationale for these minimum bounding rectangles is that they enable more efficient queries for the high dimensional data and multiple levels of granularity that are typical for OLAP. The DC-Tree includes two operations: Insert (Section 4.1 in Ester (2000)) and Range Query(Section 4.4 in Ester (2000)). When an insert causes a DC-tree node to exceed its capacity, this is handled by operations Split and Hierarchy Split(Section 4.2 and 4.3 in Ester (2000), respectively).

Even though the DC-tree was first published more than 10 years ago, and despite the fact that it does provide an elegant algorithmic solution for real-time OLAP systems, the DC-tree data structure has not found its way into commercial OLAP systems. A major problem is performance. For large data warehouses, pre-computed cuboids still outperform real-time data structures but of course with the major disadvantage of not allowing real-time updates, as discussed earlier. The main contribution of this paper is the design of a parallel DC-Tree (termed PDC-tree) for multi-core architectures. We demonstrate that the performance of our parallel DC-Tree method scales with increasing number of processor cores, thereby providing

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

the opportunity for real-time OLAP on large databases.

3. A PARALLEL DC-TREE (PDC-TREE) DATA STRUCTURE FOR PARALLEL REAL-TIME OLAP

Our parallel DC-tree method (termed PDC-tree) consists of two parts: (1) An extension of the DC-tree data structure to help exploit parallel processing and (2) new algorithms PARALLEL OLAP INSERT and PARALLEL OLAP QUERY to replace the Insert and Range Query operations in Ester (2000). The main challenge for our parallel DC-tree method is the possible interference between parallel insert and query operations, as well as between parallel insert operations operating on the same portion of the tree data structure. A straightforward solution would e.g. lock subtrees on which an insert is being performed. This would however lead to significant wait times for other queries and result in a method where the performance does not scale with increasing number of processor cores. Our solution consists of three parts: (1) A minimal locking scheme where insert operations only lock the node they are currently updating instead of the entire subtree. This can however result in concurrent other transactions working on invalid or incomplete data. (2) A timestamp mechanism added to the DC-tree data structure which allows for concurrent transactions to detect when they are working on invalid or incomplete data. (3) A set of horizontal sibling links added to the DC-tree structure which allows transactions to recover after they have detected that they were working on invalid or incomplete data.

Our method includes features that are similar to previously presented parallelizations of B-trees and R-trees. Kornacker and Banks (1995) presented the first parallel R-tree by adding two features to regular R-trees: a LSN (logical sequence number) and rightward links for each node. Their methods were improved in (Song, 2004) and (K., 1998) by introducing a new directory node structure and applying node copying strategies at the time of split. However, these methods for B-trees and R-trees are considerably simpler than our parallelization of DC-trees presented here. The DC-tree data structure is much more complicated than R-trees and B-trees because it needs to deal efficiently with very high dimensional data, intricate dimension hierarchies and data aggregation, all of which are not considered for R-trees or B-trees and their parallelizations. For example, the variety of node types in DC-trees is much more involved, and the directory node split strategies for DC-trees are much more complicated. As discussed in the remainder of this paper, this leads to many new problems and our solution includes many novel features that are needed for an efficient and scalable parallelization of DC-trees for parallel real-time OLAP.

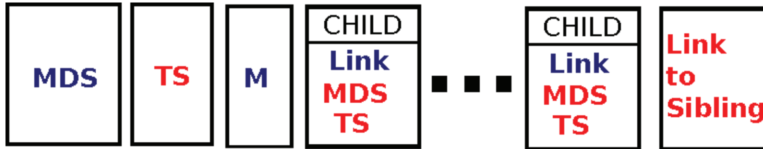
3.1. Extension of the DC-Tree Data Structure

We first describe our extension of the DC-tree data structure to handle multiple parallel transactions (insert and query operations). There are three types of nodes in a DC-tree: data nodes which are leaves of the tree, directory nodes which are internal index nodes and super nodes which are directory nodes with unlimited capacity. The directory nodes are the backbone of the DC-tree index. Next, we describe our extension of these directory nodes.

An illustration of an extended directory node is shown in Figure 4. The blue entries show the original DC-Tree entries as presented in Ester (2000): MDS is the minimum describing set for the subtree rooted at the given directory node, M is the aggregate measure of all data in the subtree, and a link provides a reference to each child node. The MDS is used for the routing of OLAP queries and the M entry is used for data aggregation in OLAP queries. The red entries in Figure 4 are those that we are adding for the parallel DC-Tree. These new entries include: (1) a time stamp TS which records the time of the most recent modification of the directory node, (2) a link to the right sibling of the given directory node, and (3) MDS and TS

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 4. Extended structure of a directory node for the Parallel DC-Tree. DC-Tree entries (blue): MDS (minimum describing set), M (measure), link to each child node. Additional entries for the parallel DC-Tree (red): TS (time stamp), link to sibling, MDS and TS for each child node link.



entries for each child link, recording the MDS and TS values for those children. The TS entry is for concurrent transactions to detect when they are working on invalid or incomplete data. The link to the right sibling will allow transactions to recover after they have detected that they were working on invalid or incomplete data. The MDS and TS entries for each child link will allow to further reduce the number of locks. In the following Sections "Algorithm PARALLEL OLAP INSERT" and "Algorithm PARALLEL OLAP QUERY" we will outline our method in more detail by presenting new parallel DC-Tree operations PARALLEL OLAP INSERT and PARALLELOLAPQUERY, respectively.

3.2. Algorithm PARALLEL OLAP INSERT

An outline of our PARALLEL OLAP INSERT method is shown in Algorithm 1. In the following, we will discuss this algorithm in more detail. For a new data item N, the algorithm starts tracing down the tree using the MDS information (Steps 24). At each directory node, three cases may occur. If Nis contained in the MDS of exactly one child, then the algorithm proceeds to that child (Step 2). If Nis contained in the MDS of more than one child (overlap), then the algorithm proceeds to the child with the smallest subtree (Step 3) in order to balance subtree sizes. If Nis not contained in the MDS of any child, then N needs to be added to the child whose MDS update leads to a minimum overlap between children in order to maintain the efficiency of search queries (Step 4). Algorithm 1 performs this operation without any node locking by fisiz creating a copy of the respective directory node, performing all of the above operations on the copy, and ficreat inserting the new directory node with a single link update. Note, however, that search queries passing through this node during the update may not become aware of the update and may therefore miss the newly inserted data item N. As discussed later, Algorithm 2 for search queries will detect and correct this with the help of the time stamp (TS) and right sibling entries added to the modified directory node. After Steps 1 to 4 of Algorithm 1 are completed, a leaf directory node has been found where the new data item N can be inserted. The remaining Steps 5 to 11 will trace the path back to the root and update the MDS entries of all directory nodes on the path. Note that, during this process at most two nodes are locked at any point in time: the current node and its parent. This is necessary to correctly perform the split operation in Step 8 which is required when a directory node's capacity is exceeded because of the new entry. In this case, a split operation has to be performed where directory node D is split into two directory nodes D and D'. For the split, we applied the sequential method in Ester (2000), Sections 4.2 and 4.3. As discussed at the end of this section, we also experimented with parallelizing the split operation itself. After D is split into two nodes D and D', node D' becomes the right sibling of D and we need to update the right sibling links accordingly. Furthermore, the time stamps of D and D' will be set such that D' receives the old time stamp of D (before the split) and D receives a new time stamp representing the current update. As discussed in the following Section "Algorithm

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Algorithm 1. PARALLEL OLAP INSERT

```
INPUT: N (new data item).
 1: Set D=root.
REPEAT
 2: IFN is contained in the MDS of only one of the children
    of D THEN set D equal to the directory node for that child.
 3: IFN is contained in the MDS of more than one of the
    entries of D THEN set D equal to the root of the child
    subtree with minimum number of data nodes.
 4: IFN is not contained in any MDS of a child of D THEN
    4.1: Make a copy D′ of D. Note that this also copies the
         MDS and TS values of the children because of our
         extended directory node structure.
    4.2: For each child C of D′: Add the new data item N to C
         and calculate the MDS enlargement and overlap caused.
    4.3: Set D = the child which causes minimal overlap.
UNTILD is a leaf node.
 5: Acquire a LOCK for D.
REPEAT
 6: Insert data item N into D and update the measure, MDS,
    and time stamp (TS) of D.
 7: Acquire a LOCK for the parent of D.
 8: IF capacity of D is exceeded THEN
    8.1: Split D into two directory nodes D and D′ as
         outlined in Ester (2000), Sections
        4.2 and 4.3.
    8.2: Make D′ the right sibling of D and update the right
         sibling links accordingly.
    8.3: Set the time stamp TS of D′ equal to the old TS
         value for D and assign D a new time stamp TS
         representing the current update.
 9: Update the Measure and MDS fields for the parent of D.
10: Release the LOCK for D.
11: Set D = parent of D.
UNTIL no further update required OR D=root.
```

PARALLEL OLAP QUERY”, this will be important for concurrent OLAP queries.

At any point in time, our PDC-tree data structure will be performing multiple concurrent PARALLEL OLAP INSERT operations executed by different threads of the multi-core processor. Any two such PARALLEL OLAP INSERT operations will not interfere unless they are attempting to perform a node split on the same directory node at the same time. This is our minimal node locking condition. Otherwise, there are no locks. In particular, at any point in time there can be multiple concurrent node split operations happening in the PDC-tree.

While the PARALLEL OLAP QUERY operations presented in the following section are also individually parallelized by applying multiple threads to each OLAP query operation, each individual PARALLEL OLAP INSERT operation as outlined above is single threaded.

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

We experimented with parallelizing the node split operation (Step 8) in Algorithm 1 which is the most time consuming part. As outlined in Ester (2000), Sections 4.2 and 4.3, a node split for a directory node D considers all "seed" pairs of entries in D, computes for each pair of seeds a node split by assigning all remaining entries to the closest seed element, and then chooses the best split of D with minimum overlap and volume. Clearly, one can also perform this operation in parallel using multiple threads. We experimented with this and it turns out that parallelizing the split operations is usually not beneficial for current multi-core processors. The reason is that a benefit would only occur if there are free compute resources available for additional threads. However, for a typical OLAP workload, such as the TPCDS benchmark used in our experiments (Section "Experimental Evaluations"), the number of concurrent transactions on the PDC-tree data structure usually exceeds the number of hardware threads available on current multi-core processors. In fact, parallelizing the split operations created additional context switching overheads in our experiments and we therefore switched the node split back to the sequential code. However, for future multi-core processors with many more processor cores, switching on the internal OLAP query parallelization described above could bring further performance benefits.

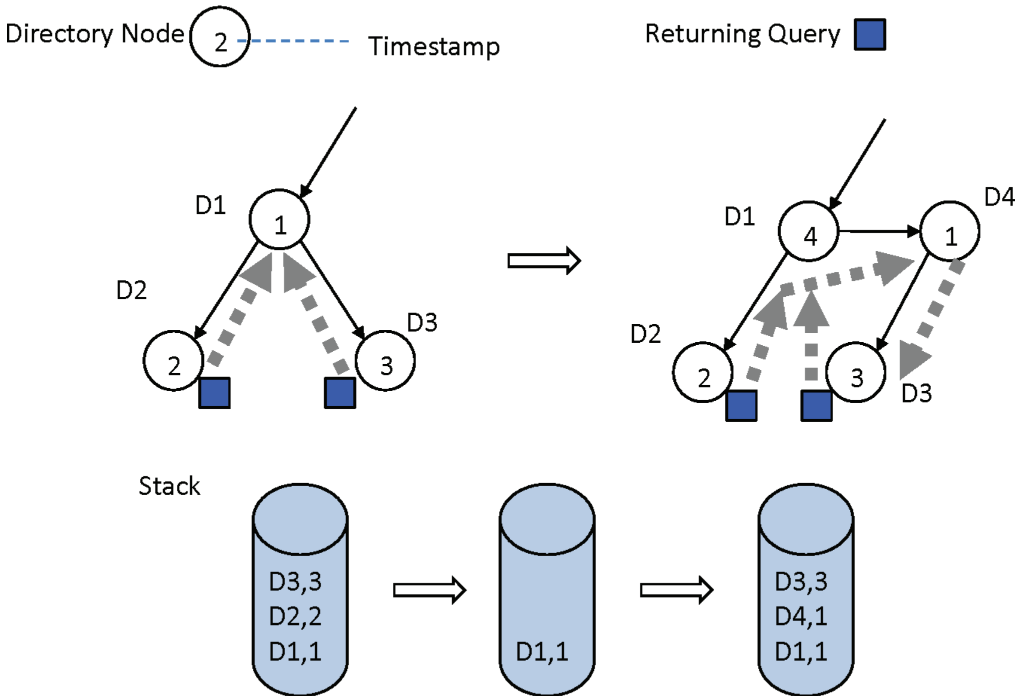
3.3. Algorithm PARALLEL OLAP QUERY

Our PARALLEL OLAP QUERY method shown in Algorithm 2 answers OLAP aggregate range queries. For a query range R (hyper rectangle on a cuboid/aggregate), it reports the aggregate measure value of all data items contained in R(e.g. total value of sales). In addition to the extended directory nodes outlined in Section "Extension of the DC-Tree Data Structure", we add a stack S to each query in order to ensure proper execution. Stack S controls the tree traversal as well as the error recovery from a detected interference with a parallel insert. The query process starts at the root and proceeds downwards. At each directory node, all children are evaluated for possible overlap with the query range R (Step 4.3). For those dimensions where the child MDS and query are at different levels of the dimension hierarchy, the one with lower level needs to be converted to the higher level (Step 4.3.1). If a child MDS fully contains R, then the entire subtree is part of the result (Step 4.3.2) and the query does not need to search inside the subtree. If a child MDS overlaps R, then that child is pushed into the stack S for further examination (Step 4.3.3). This leads to a branching off into multiple subtrees for those directory nodes where multiple children overlap R. The stack mechanism ensures that these subtrees are traversed in depth first order. For parallel transactions, the problem arising is that while one subtree of a directory node is being searched, the directory node itself could be modified by a parallel insert operation (e.g. a directory node split). This problem is addressed in the IF statement at the beginning of Step 4 together with Steps 4.1 & 4.2. Assume that the search branches off into a subtree of node D and that, during that time, node D is modified by a parallel insert. When the search returns to node D, its "old" version D' is on top of stack Sand a comparison of the time stamp of D' and the current time stamp of D detects a difference, indicating a parallel update. Figure 5 shows an illustration. Directory node D_1 is pushed into stack Sand then directory nodes D_2 and D_3. In the meantime, a concurrent insert causes a directory node split for D_1 that updates its time stamp, creates a new sibling node D_4 with the old time stamp of node D1, and redistributes the children. Node D_3 is now a child of node D_4. The concurrent search query completes its task and pops nodes D_2 and D_3 from the stack. However, when the search query wants to pop node D_1 it detects that node D_1 has been split.

In order for the search query to recover and report the correct result, the list of siblings maintained by the "Link to Sibling" pointers shown in Figure 4 should be traversed and added to stack S, thereby making sure that the subtrees are revisited and the newly inserted item is found. Note that, when a directory node

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 5. Usage of stack and timestamp mechanism to detect updates on already counted nodes in an OLAP query



is split, its children are redistributed between the original and the newly added directory node. Therefore, we need to be careful when revisiting subtrees to not "double count" items by adding them twice to the aggregate measure value R. For each item in a revisited subtree, the time stamps recorded and contents of the stack S are sufficient to determine whether this item has already been counted or not.

As outlined in Section "Contributions", our system employs two types of parallelism: executing multiple OLAP operations in parallel and further speeding up individual OLAP operations by assigning each of them multiple parallel execution threads. For a directory node D, if multiple children of D have an MDS that overlaps the query range R, then these multiple subtrees need to be traversed. If this happens close to the root node then the sizes of these subtrees can be substantial. Therefore, we parallelize the search of the subtrees as shown in Step 4.3.3 of Algorithm 2. More precisely, for each child C of D that overlaps R but is not contained in R we create a new thread that executes Algorithm 2 with input parameters Rand C.

We note that our PARALLEL OLAP QUERY method creates no locks whatsoever and therefore creates no slowdown between parallel transactions. However, it can create additional work during the recovery phase of interfering parallel transactions. This could potentially affect the scalability of our method. To which degree this does actually happen will be determined in our experimental evaluation in the following Section "Experimental Evaluation".

4. EXPERIMENTAL EVALUATION

For our experimental evaluation, we used two multi-core hardware platforms shown in Table 1:

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Algorithm 2. PARALLEL OLAP QUERY

```
INPUT: R (MDS of the given query range), D (root node).
OUTPUT: Result (aggregate measure of all data items contained in R).
Local Variable: Stack S.
 1: Push D into stack S.
REPEAT
 2: Pop top item D' from stack S.
 3: Set D to the tree node corresponding to D'.
 4: IF the time stamp (TS) of D' is smaller (earlier) than
    the time stamp (TS) of D THEN
4.1: Using the "Link to Sibling" field in directory nodes, tra-
verse the list of siblings of D. Push all siblings with time
stamp (TS) larger (later) than the parent of D into stack S.
    4.2: Push D into stack S.
  ELSE
    4.3: FOR each child C of D DO
       4.3.1: For each dimension of C where C and R are at
              different level in the dimension hierarchy,
              convert the lower level entry to the higher level.
       4.3.2: IFMDS of C is contained in R THEN add C to Result.
       4.3.3: IFMDS of C overlaps R but is not contained in R
              THEN create a new thread that executes
              Algorithm 2 with input parameters R and C.
UNTIL all threads are completed and stack S is empty.
```

- An Intel Sandy Bridge processor with 4 cores (8 hardware threads via hyper-threading) and 16 GB memory;
- A dual socket Intel Westmere EX with 20 cores (40 hardware threads via hyper-threading) and 256 GB memory.

The main goal of our experimental evaluation is to determine how our DC-Tree parallelization scales when we increase the number of processor cores. As discussed in Section "A Parallel DC-Tree (PDC-Tree) Data Structure for Parallel Real-Time OLAP", our PDC-tree algorithms use a minimal locking scheme which allows parallel transactions to execute independently without concern about interference. Should such interference occur, our algorithm detects this and corrects the result during a recovery step. How much overhead is created during those recovery steps depends on the actual data and queries. Clearly, using random data and random queries does not provide a

Table 1. The two hardware platforms used in our experiments

Processor	Number of Cores	Memory Size
Intel Sandy Bridge	4 Cores	16 GB Memory
	8 Hardware Threads (Hyper-threading)	
Intel Xeon Westmere EX (2 Sockets)	20 Cores	256 GB Memory
	40 Hardware Threads (Hyper-threading)	

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

correct estimate because one would expect it to evenly distribute the parallel threads over the PDC-tree which would be a best case scenario for the amount of interference between concurrent transactions. What is needed is a realistic set of data and queries that provides real life scenarios. After discussions with members of the data analytics team at IBM/Cognos Canada, we selected the TPCDS "Decision Support" benchmark by the Transaction Processing Performance Council (TPCDS). The TPCDS benchmark provides transactions that model the decision support system of a retail product supplier. It includes OLAP queries and data insertions. (Decision support systems are based on historic corporate data and usually do not include data deletions.) The TPC benchmark and its variants are the standard and most widely used benchmarks for OLAP. As stated on the TPC website, "although the underlying business model of TPCDS is a retail product supplier, the database schema, data population, queries, data maintenance model and implementation rules have been designed to be broadly representative of modern decision support systems" (TPCDS).

Our PDC-tree algorithm was implemented in C++ with OpenMP, and compiled/executed on Linux kernel 2.6.38 using g++ 4.5.2. For each experiment, we first built an initial PDC-tree with TPCDS data and then tested our system with a stream of TDCDS insert and query transactions as shown in Figure 3. Another important parameter influencing query time (sequential or parallel) is the amount of data that an OLAP query needs to aggregate. Clearly, a query that computes an aggregate over 5% of the database has a much smaller workload than a query that computes an aggregate over 50% of the database. We refer to this parameter as query coverage, and we evaluate in our experiments how our PDC-tree method performs for different query coverages.

4.1. Experiments on Intel Sandy Bridge (4 Cores)

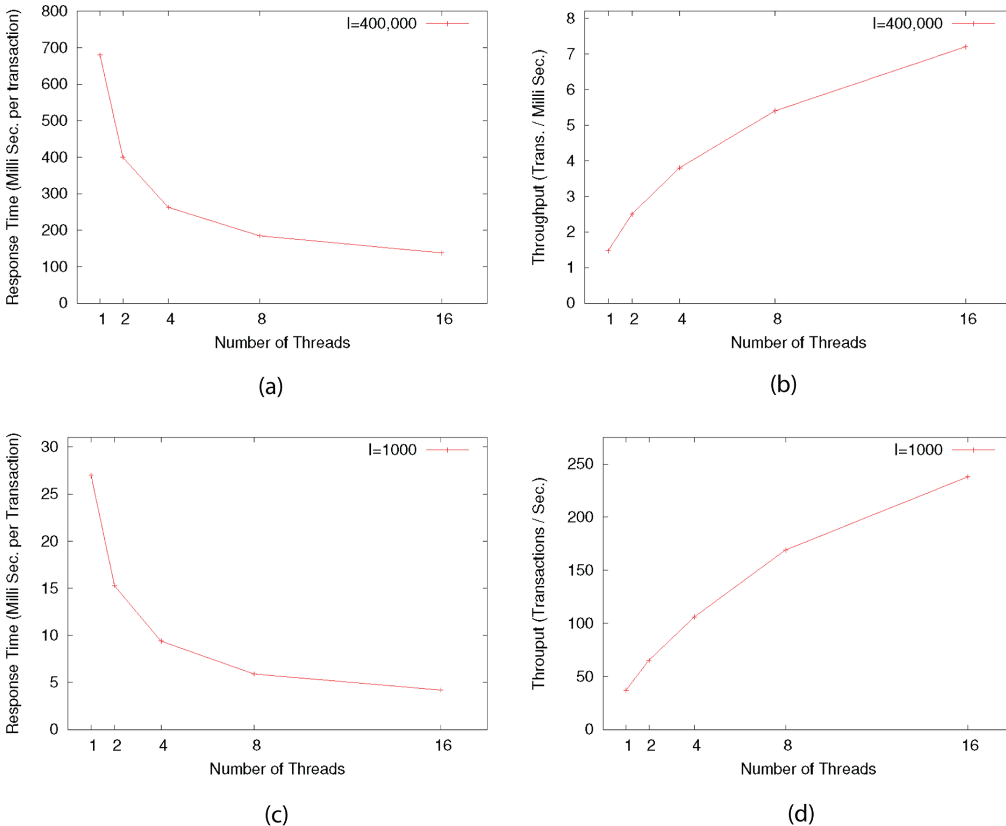
Figures 6 (a & b) show the performance of the initial PDC-tree building phase. Here we show the average transaction response time and throughput when 400,000 initial PARALLEL OLAP INSERT operations (with TPCDS data) are executed on 1, 2, 4, 8, and 16 threads, respectively. Figures 6 (c & d) show the average transaction response time and throughput for a subsequent set of 1,000 PARALLEL OLAP INSERT operations into an OLAP database with 400,000 loaded items. The performance in Figures 6(a & b) is lower because the fib) inserts into an initially small tree data structure create a relatively large number of expensive directory node splits and have a very high probability of query interference.

As shown in Figures 6(c & d), for a database with 400,000 loaded items, performance improves significantly. In general, we observed that performance is better for larger data warehouses. The speedup shown in Figures 6(c & d) is approx. 40% of the maximum theoretically possible linear speedup. Considering that the cores of the Sandy Bridge processor share resources (e.g. memory bus), this is an encouraging result for a fully running system on "real life" benchmark data. The Sandy Bridge platform is clearly too small a platform with insufficient memory for a sufficiently large database. For larger data warehouses on a larger platform, as discussed in Section "Experiments on Intel Xeon Westmere EX (20 Cores)" below, the performance results are considerably better.

Figures 7, 8 and 9 show the performance (transaction response time and transaction throughput) for a stream of PARALLEL OLAP INSERT and PARALLEL OLAP QUERY transactions (using TPCDS benchmark data) executed on 1, 2, 4, 8, and 16 threads, respectively. Note that the single thread code on one processor core is the sequential code only with all parallelization code (and possibly resulting overhead) removed. The different curves correspond to different ratios between the number of insertions and queries (IOLAP insertions and QOLAP queries). Since the performance of a (sequential or parallel) OLAP query is strongly influenced by the query coverage, we provide three different graphs, Figure 7, Figure 8 and Figure 9 for queries that aggregate 1%, 5% and

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 6. Parallel OLAP insertion performance of the PDC-tree as a function of the number of parallel threads for Intel Sandy Bridge. (a) & (b) Average transaction response time & throughput for 400,000 OLAP insertions to build an initial database. (c) & (d) Average transaction response time & throughput for a subsequent 1,000 OLAP insertions into the built database.

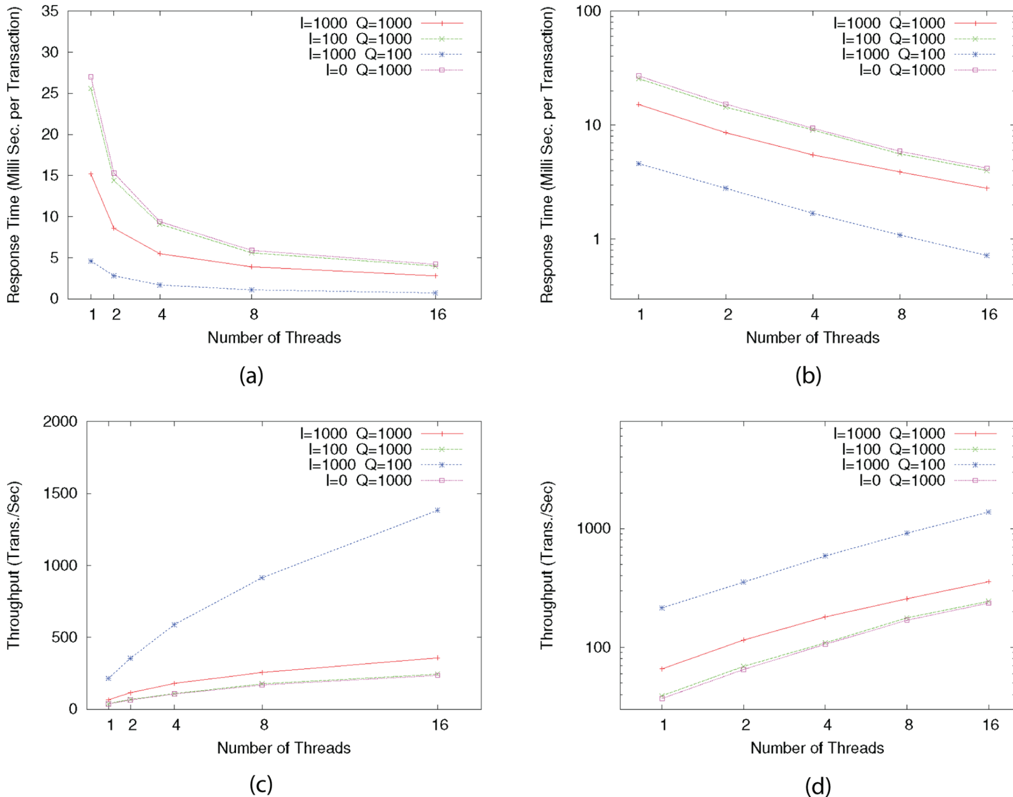


25% of the entire database, respectively. In each case, we show response time and throughput for 1, 2, 4, 8, and 16 threads. In order to better show the speedup achieved, we show each set of curves in linear scale as well as in log-log scale. Our main observation is that we achieve about 40% of optimal linear speedup. Considering that the cores of the Sandy Bridge processor share resources (e.g. memory bus), this is an encouraging result for a fully running system on such a small platform. We also observe that query transactions are considerably slower than insert transactions. This is typical (also for sequential transactions) since an insert corresponds to just one path down and back up the tree whereas query transactions may need to search multiple subtrees and may need to aggregate a sizeable portion of the database. In the mix of parallel insert and query transactions on the same database, the speedup observed was best for the hardest case of queries only, which is where speedup is most needed in practice.

In addition to the speedup observed, we also compared the runtimes of our PDC-tree implementation with those of a current multithreaded database system. We chose MySQL (MySQL), a well-known open source database system which has been optimized for multi-core parallelism. On the same Intel Sandy Bridge architecture, we ran our PDC-tree implementation against MySQL with its multithreading option set to “maximum” parallelism, i.e. its fastest setting.

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 7. Parallel OLAP transaction performance of the PDC-tree as a function of the number of parallel threads for Intel Sandy Bridge. Mixed input of I OLAP insertions and Q OLAP queries. Queries aggregate 1% of database. (a) Average response time, linear scale. (b) Average response time, log-log scale. (c) Throughput, linear scale. (d) Throughput, log-log scale.

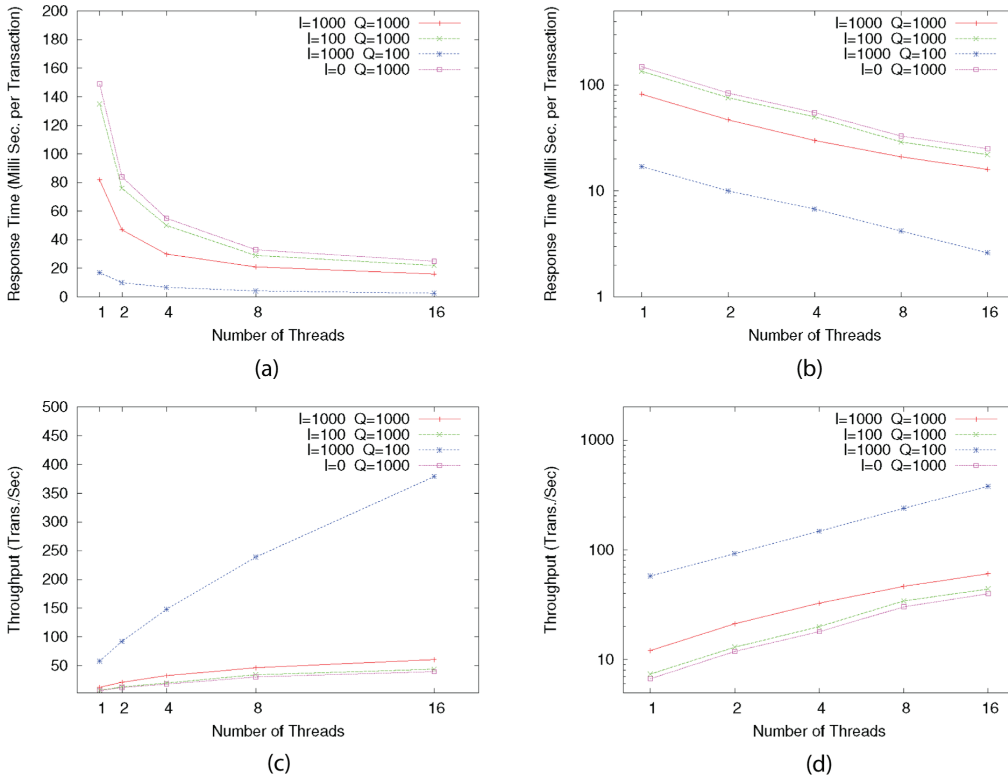


For both, the PDC-tree and MySQL, we first executed the same sequence of 400,000 data insertions from the TPCDS data set and then, for query coverages 1%, 25%, 75% and 100%, the same sequences of queries. Table 2 summarizes the results. For 1% query coverage, multithreaded MySQL performs essentially at the same speed as the PDC-tree. Such queries aggregate only a very small portion of the database and are easy to answer directly in standard database systems like MySQL. However, as we increase query coverage, the queries become much more time consuming for MySQL and the PDC-tree starts outperforming MySQL by a wide margin. For the extreme case of 100% query coverage, the PDC-tree can pick up the aggregated result at nodes very close to the root and performs extremely fast, two orders of magnitude faster than MySQL. For 25% and 75% query coverage, our PDC-tree implementation outperforms multithreaded MySQL by approximately a factor five.

We also observe that in Table 2, for 25% and 75% query coverage, the single thread performance of our PDC-tree implementation approximately matches the performance of multithreaded MySQL. If we compare that with e.g. Figure 9a, this means that the baseline for the speedup observed for our PDC-tree Figure 9a (i.e. it's single thread performance) is similar to the performance of multithreaded MySQL. We are highlighting this because

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 8. Parallel OLAP transaction performance of the PDC-tree as a function of the number of parallel threads for Intel Sandy Bridge. Mixed input of I OLAP insertions and Q OLAP queries. Queries aggregate 5% of database. (a) Average response time, linear scale. (b) Average response time, log-log scale. (c) Throughput, linear scale. (d) Throughput, log-log scale.



speedup is easy to achieve with respect to a slow sequential performance but in our case, the PDC-tree achieves a good speedup relative to a highly optimized and widely used public domain system.

4.2. Experiments on Intel Xeon Westmere EX (20 Cores)

As discussed in the previous section, the Intel Sandy Bridge platform is very limited in terms of memory (i.e. database size) and number of cores available. In order to test our PDC-tree on an architecture that would be more typical for commercial in-memory OLAP systems, IBM provided us access to one of their commercial servers, a 20 core Xeon Westmere EX with 256 GB memory. This allowed us to use a more realistic database size and also test the speedup achieved by our PDC-tree method for a larger number of processor cores. The latter is critical since achieving good speedup is typically easier for a smaller number of processors and becomes harder as the number of processor cores increases. We performed the following experiments:

1. Measuring the performance of 14 million OLAP inserts into a PDC-tree with 1 million data items (PDC-tree loading);
2. Measuring the performance of a mixed stream of OLAP insert and query transactions on a PDC-tree with 1 million data items;

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 9. Parallel OLAP transaction performance of the PDC-tree as a function of the number of parallel threads for Intel Sandy Bridge. Mixed input of I OLAP insertions and Q OLAP queries. Queries aggregate 25% of database. (a) Average response time, linear scale. (b) Average response time, log-log scale. (c) Throughput, linear scale. (d) Throughput, log-log scale.

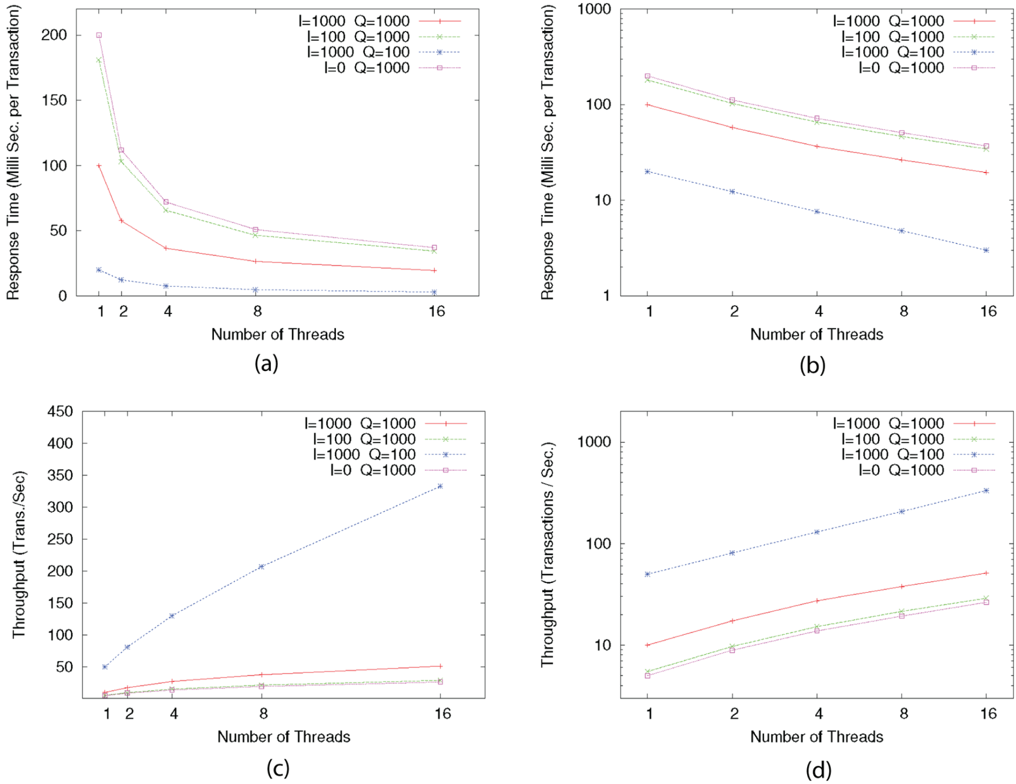
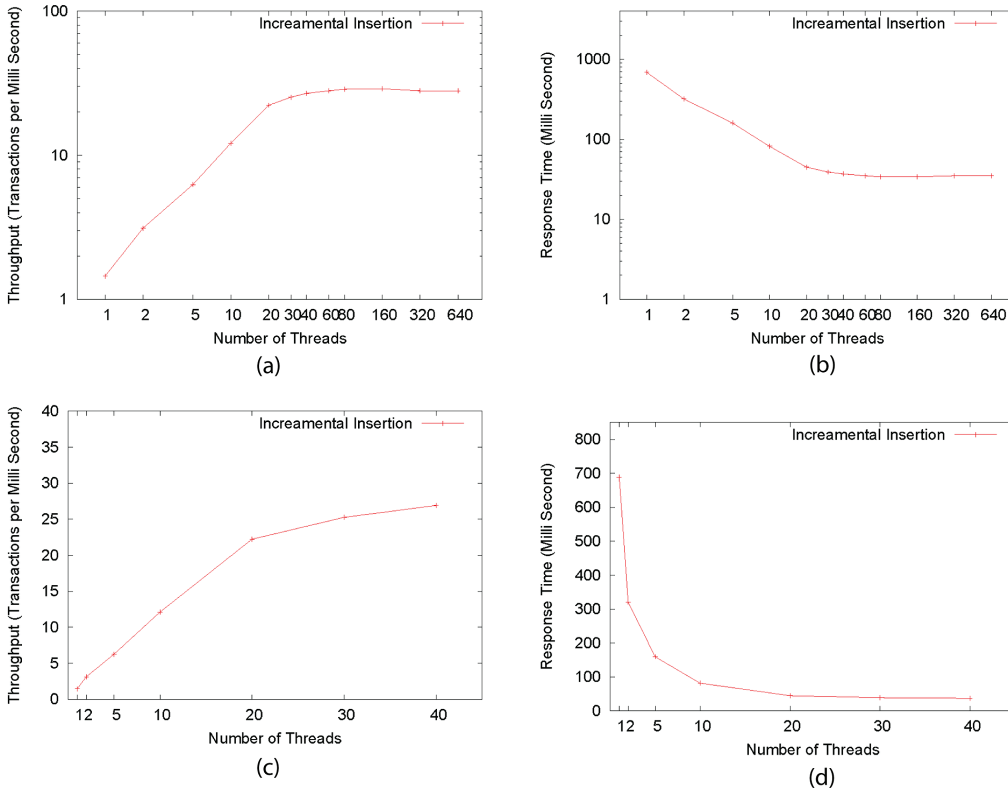


Table 2. Runtime comparison between PDC-tree and multithreaded MySQL. Times shown are the total runtimes for the same sequences of 1,000 OLAP queries, with varied query coverages, on the same databases with 400,000 items. All data and queries are from the TPCDS benchmark.

PDC-Tree				
# Threads:	**1% Coverage**	**25% Coverage**	**75% Coverage**	**100% Coverage**
1	26.894 sec	197.745 sec	274.416 sec	9.136 sec
2	15.395 sec	112.229 sec	156.912 sec	6.824 sec
4	9.433 sec	72.014 sec	100.114 sec	5.433 sec
8	5.942 sec	51.634 sec	72.265 sec	4.961 sec
16	4.220 sec	37.892 sec	53.091 sec	4.788 sec
MySQL				
# Threads:	1% Coverage	25% Coverage	75% Coverage	100% Coverage
“maximum”	5 sec	161 sec	245 sec	415 sec

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 10. PDC-tree insertions only performance as a function of the number of parallel threads for Intel Xeon Westmere EX. Stream of 14 million insertions into an initial PDC-tree with 1 million data items. (a) Throughput in log-log scale. (b) Average transaction response time in loglog scale. (c) Throughput in linear scale. (d) Average transaction response time in linear scale.



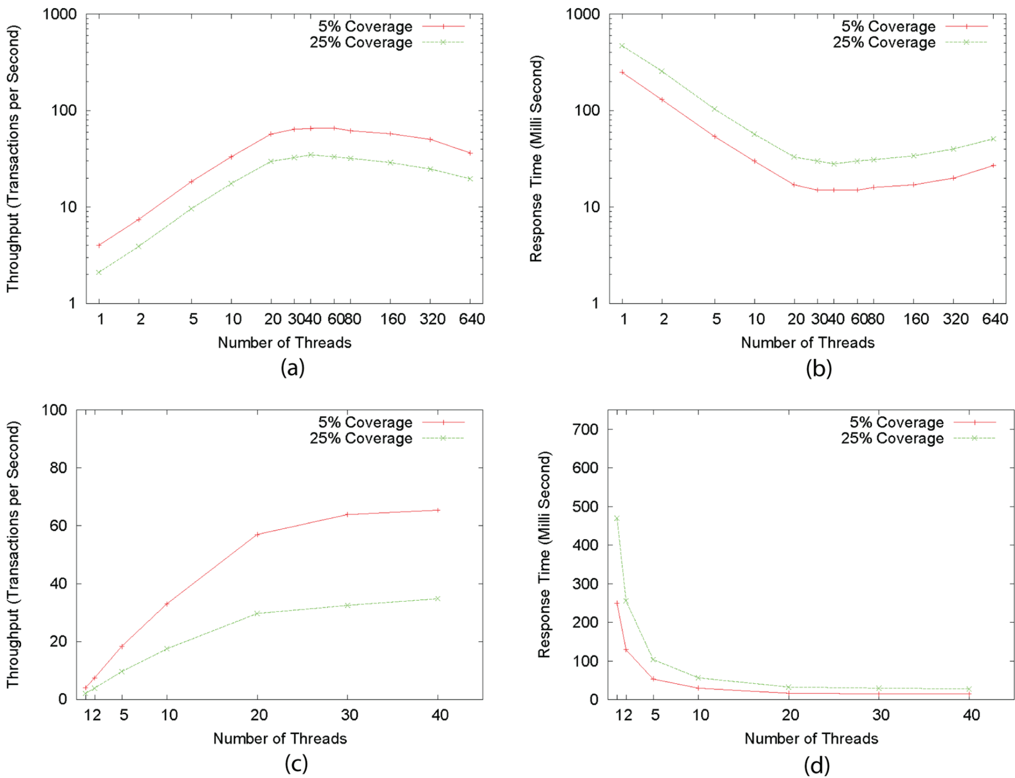
3. Measuring the performance of a mixed stream of OLAP insert and query transactions on a PDC-tree with 10 million data items (100 GB).

The first experiment tests the performance of our PDC-tree implementation during the initial loading of a database. We built an initial PDC-tree with 1 million TPCDS data items and then executed a stream of 14 million TPCDS data insertions. The measured throughput and transaction response times for the 14 million insertions are shown in Figure 10. The speedup shown for up to 20 threads (on 20 processor cores) is very close to optimal linear speedup. As outlined in Section "A Parallel DC-Tree (PDC-Tree) Data Structure for Parallel Real-Time OLAP", at any point in time, a PARALLEL OLAP INSERT operation only locks a most two PDC-tree directory nodes (e.g. for a node split). When the PDC-tree is large, there is close to no interference and parallel PARALLEL OLAP INSERT threads can operate independently on the tree data structure. This explains the very close to optimal speedup. As shown in Figure 10, when the number of threads is between 20 and 40, the additional speedup decreases significantly because these additional threads are supported by hyper-threading rather than actual cores.

The second experiment tests the performance of our PDC-tree implementation for a mixed stream of OLAP insert and query operations on a PDC-tree for a smaller size

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 11. Parallel OLAP insertion and query performance of the PDC-tree as a function of the number of parallel threads for two query coverages, 5% & 25%, on a PDC-tree with 1 million data items for Intel Xeon Westmere EX. Number of queries and insertions are 1000 and 100, respectively. (a) Throughput in log-log scale. (b) Average transaction response time in log-log scale. (c) Throughput in linear scale. (d) Average transaction response time in linear scale.

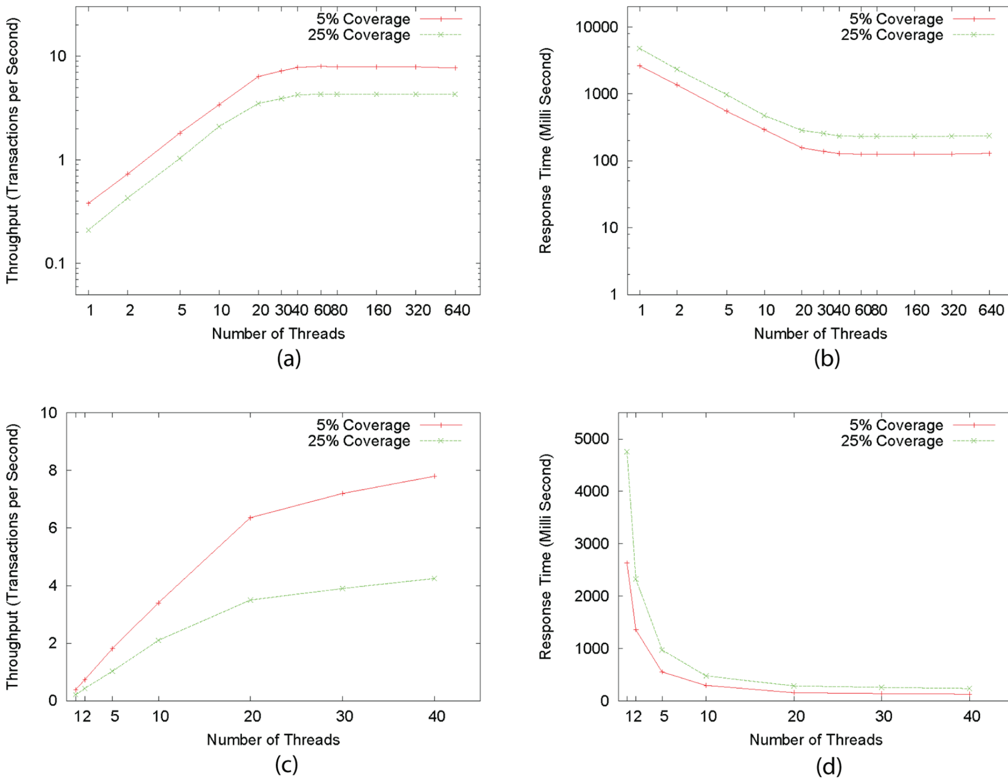


database (but larger than the database on the Intel Sandy Bridge). We first built an initial PDC-tree by inserting 1 million rows of the TPCDS data set and then executed a stream of TPCDS transactions that consists of a mix of 1,000 query and 100 insert operations. Figure 11 shows throughput and response time of our PDC-tree implementation as we increase the number of threads. The red and green curves show the performance for OLAP queries with 5% and 25% query coverage, respectively. As shown in Figure 11, performance increases with close to optimal linear speedup for the range of 1 to 20 threads (on 20 processor cores). For the range of 20 to 40 threads, we obtain another 18% performance increase through hyper-threading which is within the typical 15%30% interval reported by Intel (D., 2002). Performance peaks at around 40 threads. For more than 40 threads, performance stays flbu (no more gains via latency hiding compared to overhead from context switching between threads).

The third experiment tests the performance of our PDC-tree implementation for a mixed stream of OLAP insert and query transactions on a PDC-tree for a larger size database. We first built an initial PDC-tree by inserting 10 million rows (100 GB) of the TPCDS data set and then executed a stream of TPCDS transactions that consists of a mix of 10,000 query and 1,000 insertion operations. Figure 12 shows throughput and response time of our PDC-tree

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 12. Parallel OLAP insertion and query performance of the PDC-tree as a function of the number of parallel threads for two query coverages, 5% & 25%, on a PDC-tree with 10 million data items for Intel Xeon Westmere EX. Number of queries and insertions are 10,000 and 1000, respectively. (a) Throughput in log-log scale. (b) Average transaction response time in log-log scale. (c) Throughput in linear scale. (d) Average transaction response time in linear scale.



implementation as we increase the number of threads. The red and green curves show the performance for OLAP queries with 5% and 25% query coverage, respectively. Note that, the response times in Figure 12 are approximately ten times the response times in Figure 11 because on the PDC-tree with 10 million data items, OLAP queries with 5% and 25% query coverage aggregate ten times as much data as OLAP queries with 5% and 25% query coverage on a PDC-tree with 10 million data items. As discussed earlier, query times are significantly larger than insertion times and dominate the average transaction performance. As shown in Figure 12, performance on the larger size database increases with close to optimal linear speedup for the range of 1 to 20 threads (on 20 processor cores). For the range of 20 to 40 threads, we obtain again a smaller performance increase through hyper-threading that is within the typical range reported by Intel (D., 2002). Performance peaks again at around 40 threads. For more than 40 threads, performance stays flla. The speedup obtained is further elucidated in Table 3 (same data is in Figure 12) which shows for our mixed stream of 10,000 queries and 1,000 insertions the average transaction response times and speedup for 5% and 25% query coverage. In both cases, we measure an optimal linear speedup of 20 for a machine with 20 processor cores. We also observe that our multi-core parallelization enables us to achieve

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Table 3. Parallel OLAP insertion and query performance of the PDC-tree as a function of the number of parallel threads for two query coverages, 5% & 25%, on a PDC-tree with 10 million data items for Intel Xeon Westmere EX. Number of queries and insertions are 10,000 and 1000, respectively. Same data as in Figure 12. Average transaction response times and associated speedup.

Thread #	Response Time (sec) for 5% Query Coverage	Speedup for 5% Query Coverage	Response Time (sec) for 25% Query Coverage	Speedup for 25% Query Coverage
1	2.47	1.00	4.57	1.00
2	1.27	1.93	2.35	1.94
5	0.51	4.77	0.94	4.82
10	0.27	9.02	0.50	9.09
20	0.14	16.80	0.28	16.25
40	0.12	20.06	0.22	19.90
80	0.12	20.48	0.22	20.05
160	0.12	20.63	0.22	20.07
320	0.12	20.39	0.22	19.95
640	0.12	20.40	0.22	19.92

a better than 0.25 second response time for real-time OLAP queries that aggregate 25% of a 100 GB database.

5. CONCLUSION

In this paper, we presented a parallel real-time OLAP system for multi-core processors. We evaluated our method on an Intel Sandy Bridge processor with 4 cores and an Intel Xeon Westmere processor with 20 cores, using the TPCDS "Decision Support" benchmark data set. The tests demonstrate that, with increasing number of processor cores, our parallel system achieves close to optimal linear speedup in transaction response time and transaction throughput. On the 20 core architecture we achieved, for a 100 GB database, a better than 0.25 second query response time for real-time OLAP queries that aggregate 25% of the database. Since hardware performance improvements are currently, and in the foreseeable future, achieved not by faster processors but by increasing the number of processor cores, our new parallel-real-time OLAP method has the potential to enable OLAP systems that operate in real-time on large databases.

ACKNOWLEDGMENT

Research partially funded by the IBM Center for Advanced Studies Canada. We'd like to thank in particular Stephan Jou, Mikhail Genkin and Robin Grosset for their support and helpful discussions. We also acknowledge the help of H.P. Kriegel and his group at Univ. Munich who provided us with code segments for sequential DC-Tree operations (Ester, 2000).

REFERENCES

Ahmed, U., Tchounikine, A., Miquel, M., & Servigne, S. Realtime temporal data warehouse cubing. In: Proceedings of the 21st international conference on Database and expert systems applications: Part II (2010)

Banks, D. (1995). *HighConcurrency Locking in RTrees* (pp. 1–12). VLDB.

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Berchtold, S., Keim, D. A., & Kriegel, H. P. (1996). *The Xtree: An index structure for highdimensional data* (pp. 28–39). VLDB.

Bruckner, R., List, B., & Schiefer, J. (2002). Striving towards near real-time data integration for data warehouses. *DaWaK*, *LNCS2454*, 173–182.

Chakrabarti, K. (1999). *Efficient Concurrency Control in Multidimensional Access Methods* (pp. 25–36). ACM SIGMOD.

Chen, Y., Dehne, F., Eavis, T., & Rau-Chaplin, A. (2008). RauChaplin, A.: PnP: sequential, external memory, and parallel iceberg cube computation. *Distributed and Parallel Databases*, *23*(2), 99–126. doi:10.1007/s10619-007-7023-y

D. L. (2002). H. G., H. D., A. K., J. A., M. M., U. Deborah, T. Marr, F.B.: Hyperthreading technology architecture and microarchitecture. *Intel Technology Journal*, *6*, 4–15.

Dai, J. Efficient Concurrent Operations in Spatial Databases. PhD Thesis, Virginia Polytechnic (2009)

Dehne, F., Eavis, T., & Hambrusch, S. (2002). Parallelizing the data cube. *Distributed and Parallel Databases*, *11*, 181–201.

Ester, M., Kohlhammer, J., & Kriegel, H. P. The DC-Tree: a fully dynamic index structure for data warehouses. *16th International Conference on Data Engineering (ICDE)* pp. 379–388 (2000) doi:10.1109/ICDE.2000.839438

Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., & Venkatrao, M. et al. (1997). Data Cube: A Relational Aggregation Operator Generalizing GroupBy, CrossTab, and SubTotals. *Data Mining and Knowledge Discovery*, *1*(1), 29–53. doi:10.1023/A:1009726021843

Guttman, A. (1984). *Rtrees: a dynamic index structure for spatial searching* (pp. 47–57). ACM SIGMOD.

Han, J., & Kamber, M. (2000). *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers.

Haritsa, J. R., Member, S., & Seshadri, S. (2000). RealTime Index Concurrency Control. *IEEE Transactions on Knowledge and Data Engineering*, *12*(3), 429–447. doi:10.1109/69.846294

Jin, D., Tsuji, T., & Higuchi, K. (2008). An Incremental Maintenance Scheme of Data Cubes and Its Evaluation. *DASFAA*, *LNCS4947*, 36–48.

Kornacker, M., & Banks, D. Highconcurrency locking in rtrees. VLDB '95 Proceedings of the 21th International Conference on Very Large Data Bases (1995)

K.V. R.K., D., S., A.K., S.: Improved concurrency control techniques for multidimensional index structures. Parallel Processing Symposium, 1998. IPPS/SPDP 1998. (1998)

Lee, M. L., Hsu, W., Jensen, C. S., Cui, B., & Teo, K. L. (2003). *Supporting Frequent Updates in RTrees: A BottomUp Approach*. VLDB.

Mysql database management system, http://www.mysql.com

Ng, R. T., Wagner, A., & Yin, Y. (2001). Icebergcube computation with PC clusters. *ACM SIGMOD*, *30*(2), 25–36. doi:10.1145/376284.375666

Plattner, H., & Zeier, A. (2012). *InMemory Data Management An Inflection Point for Enterprise Applications*. Springer.

Santos, R., & Bernardino, J. (2008). *Realtime data warehouse loading methodology* (pp. 49–58). IDEAS.

Santos, R. J., & Bernardino, J. (2009). *Optimizing data warehouse loading procedures for enabling usefultime data warehousing* (pp. 292–299). IDEAS.

Song, S. I., Kim, Y. H., & Yoo, J. S. (2004). An enhanced concurrency control scheme for multidimensional index structures. IEEE Transactions on Knowledge and Data Engineering Archive, 16(1).

The OLAP Report. Http://www.olapreport.com

TPCDS. Transaction Processing Performance Council: (Decision Support) Benchmark. http://www.tpc.org/tpcds/tpcds.asp

You, J., Xi, J., Zhang, P., & Chen, H. A Parallel Algorithm for Closed Cube Computation. *IEEE/ACIS International Conference on Computer and Information Science* pp. 95–99 (2008) doi:10.1109/ICIS.2008.63

Zhou, G.-L., Chen, H., Li, C.-P., Wang, S., & Zheng, T.GuoLiang. (2010). Z., Hong, C., CuiPing, L., Shan, W., Tao, Z.: Parallel Data Cube Computation on Graphic Processing Units. *Chines Journal of Computers*, *33*(10), 1788–1798. doi:10.3724/SP.J.1016.2010.01788

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Frank Dehne received a MCS (Dipl. Inform.) from RWTH Aachen University, Germany and a PhD from the University of Würzburg, Germany. He is currently Chancellor's Professor of Computer Science at Carleton University in Ottawa, Canada. His research program is focused on parallel algorithms engineering for current parallel architectures (multi-core, cluster, cloud, GPU) with the goal of improving the performance of large-scale data science systems, in particular for computational biology and big data analytics. Dr. Dehne is a Fellow of the IBM Centre for Advanced Studies Canada, member and former vice-chair of the IEEE Technical Committee on Parallel Processing, member of the ACM Symposium on Parallel Algorithms & Architectures Steering Committee, and co-founder of the Algorithms and Data Structures Symposium (WADS).

Hamidreza Zaboli received his M.Sc. from Amirkabir University of Technology (Tehran Polytechnic). He is currently a PhD candidate in the School of Computer Science at Carleton University. From 2010 to 2013, he had been a PhD fellow student at IBM Centre for Advanced Studies (CAS) Canada. His current research interest is in the application of parallel methods to enable real-time OLAP on multi-core and cloud platforms.

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.