2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

Parallel Real-Time OLAP on Multi-Core Processors

Frank Dehne
School of Computer Science
Carleton University
Ottawa, Canada
Email: frank@dehne.net

Hamidreza Zaboli
School of Computer Science
Carleton University
Ottawa, Canada
Email: hzaboli@connect.carleton.ca

***Abstract*—One of the most powerful and prominent technologies for knowledge discovery in *Decision Support* systems is *On-line Analytical Processing* (OLAP). Most of the traditional OLAP research, and most of the commercial systems, follow the *static* data cube approach proposed by Gray etal. 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 are 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 processors, providing the opportunity for real-time OLAP on large scale data warehouses. Our system allows for multiple *insert* and multiple *query* operations (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 sizes of results, different system loads, etc.), using the TPC-DS "Decision Support" benchmark data set. The tests demonstrate that our parallel system achieves a significant speedup in transaction response time and a significant increase in transaction throughput. Since hardware performance improvements are currently 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 are *real-time* and efficient/feasible for large databases.

***Keywords*-Parallel Computing; Real-Time OLAP; Multi-Core Processors;**

I. 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, *On-line 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.

A. 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 On-line 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 [1], [2]. The processing logic associated with this form of analysis is encapsulated in what is known as the OLAP server. By exploiting multi-dimensional 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 [3]. 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

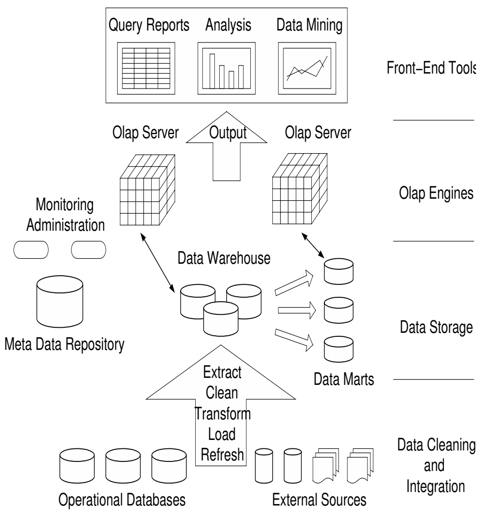


Figure 1. Three-tiered OLAP model.

cell in the cube structure corresponds to an aggregate value along one or more measure attributes (e.g. Total Sales).

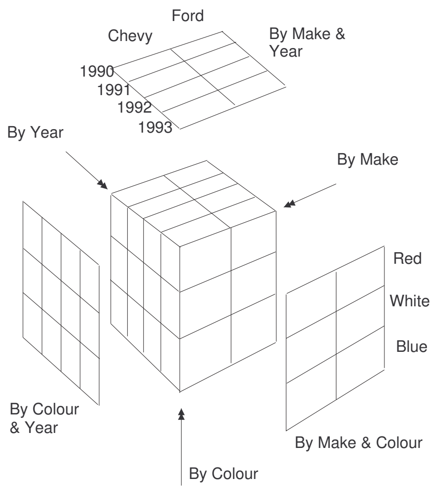


Figure 2. A three dimensional data cube for automobile sales data.

B. Contributions

Most of the traditional OLAP research, and most of the commercial systems, follow the *static* data cube approach proposed by Gray etal. [3] 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. [4], [5], [3], [6], [7], [8]).

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 can not 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 on-line 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 the decision support process (e.g. [9]). 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. [9], [10], [11], [12]). However, these approaches are not fully real-time. One major problem are 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 processors. Our system allows for multiple *insert* and multiple *query* operations (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 sizes of results, different system loads, etc.), using the TPC-DS "Decision Support" benchmark data set. The tests demonstrate that our *parallel* real-time OLAP system achieves a significant speedup in transaction response time and a significant increase in transaction throughput on contemporary *Sandy Bridge* multi-core processors. 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 are *real-time* and efficient/feasible for large databases.

The remainder of this paper is organized as follows. In Section II we present our new parallel algorithm for real-time OLAP on multi-core processors, and in Section III we analyze the performance of our method and show the speedup and improved system throughput that can be achieved through multi-core parallelization. Section IV concludes our paper.

II. Algorithms Overview: Parallel 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 databases which have been studied since the 90's, e.g. in [13], [14], and more recently in [15], [16]. Concurrent operations in spatial databases have recently been studied e.g. in [17]. However, the general DB index structures presented are not efficient for the large number of dimensions typically required for OLAP. 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 etal. [18], which is a *sequential* tree based index structure specifically designed for data warehouses with dimension hierarchies. Even though it was published more than 10 years ago, and despite the fact that it does provide an 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. The main contribution of this paper is the design of a *parallel* DC-tree for multi-core architectures. We demonstrate that the performance of our *parallel* DC-tree method scales as the number of processor cores increases. Therefore, our new *parallel* DC-tree method has the potential to enable OLAP systems that are *real-time* and efficient for large databases.

In the remainder of this section, we will outline our *parallel* DC-tree method, based on the *sequential* DC-tree [18] which extends the classical X-tree [19] and R-tree [20] data structures for multi-dimensional data indexing. A data cube 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 defines a partial ordering and *concept hierarchy* for each dimension, where the concept hierarchy is an additional tree structure storing for a given dimension all values that occur in the DC-tree at each given time. Using this partial ordering and concept hierarchy for each dimension, the DC tree extends the usual R-tree [20] based tree representation for multi-dimensional data 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 comes with two operations: *Insert* (Section 4.1 in [18]) and *Range Query* (Section 4.4 in [18]). When an insert causes a node to exceed its capacity, this is handled by operations *Split* and *Hierrachy Split* (Section 4.2 and 4.3 in [18], respectively).

Our *parallel* DC-tree method consists of two parts: (1) An extension of the DC-tree data structure and (2) new algorithms *PARALLEL_OLAP_INSERT* and *PARALLEL_OLAP_QUERY* to replace the *Insert* and *Range Query* operations in [18]. (We also updated the Split and Hierarchy Split operations to ensure that they correctly maintain our extended data structure.) 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. 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 *time stamp* 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 allow transactions to recover after they have detected that they were working on invalid or incomplete data.

MDS	**TS**	M	Child C_1:	...	Child C_n:	**Link**
			Link$_1$		Link$_n$	**to**
			MDS$_1$		MDS$_n$	**Sib-**
			TS$_1$		TS$_n$	**ling**

Figure 3. Extended structure of a directory node D for the Parallel DC-Tree (MDS = minimum describing set, TS = time stamp, M = measure). Bold/Color: Added fields "TS" and "Link to Sibling".

Our solution is outlined in Algorithm 1 and Algorithm 2. As illustrated in Figure 3, we add two fields to each directory node of the DC-tree. A time stamp (TS) field indicates the most recent time at which the node has been modified (or created). A "Link to Sibling" field is used to create and maintain a linear chain between the children of each directory node. Our *parallel* DC-tree operations *PARALLEL_OLAP_INSERT* and *PARALLEL_OLAP_QUERY* use these fileds to efficiently and correctly handle parallelism. We will now discuss each operation in more detail.

Our *PARALLEL_OLAP_INSERT* method shown in Algorithm 1 takes a new data item N and starts tracing down the tree using the MDS information as guidance (Steps 2-4). At each directory node, three cases may occur. If N is contained in the MDS of exactly one child, then the algorithm proceeds to that child (Step 2). If N is contained in the MDS of more than one child, then the algorithm proceeds to the child with the smallest subtree (Step 3). If N is not contained in the MDS of any child, then N

needs to be added to the child whose MDS update leads to minimum overlap between children, in order to maintain efficiency of search queries (Step 4). Algorithm 1 performs this operation without locking by first creating a copy of the directory node, performing all of the above operations on the copy, and finally 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) field added to the modified directory node. After Steps 1 to 4 of Algorithm 1 are completed, a leaf node has been found where the new data item N is 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 the entire 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 nodes capacity is exceeded by the new entry. The potential interference between parallel transactions caused by these locks should not result in significant wait times. The main effect is that queries proceed upwards in parallel but in lock-step. To which degree this impacts the scalability of our method (i.e. the speedup obtained with increasing number of cores), will be measured in our experimental evaluation in Section III.

```
  Algorithm 1: PARALLEL_OLAP_INSERT
Input: New data item N.
Local Variable: Directory node D.
  (1) Set D=root.
REPEAT
  (2) IF N 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) IF N is contained in the MDS of more than one of the
      entries of D THEN set D equal to the the root of the
      child subtree with minimum number of data nodes.
  (4) IF N is not contained in any MDS of a child of D
      THEN
      (4.1) Make a copy D' of D.
      (4.2) For each child entry of D':
              Add the new data item N and update its
              MDS if necessary. Calculate the (possible)
              MDS enlargement and the overlap caused
              by the (possible) MDS enlargements.
      (4.3) Set D = the child which causes minimal overlap.
UNTIL D is a leaf directory 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 call Split(D) [18].
  (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.
— End of Algorithm —
```

Our *PARALLEL_OLAP_QUERY* method shown in Algorithm 2 takes a query range R (hyper-rectangle on a cuboid/aggregate) and reports all data items contained in R and their aggregate measure value (e.g. total dollar value of sales). The query process is guided by a stack S which controls the tree traversal as well as the error recovery from a detected parallel query interference (e.g. 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). 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. 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 one 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 the stack S and a comparison of the time stamp of D' and the current time stamp of D detects a difference, indicating a parallel update. In order for the search query to recover and report the correct result, the list of siblings maintained by the "Link to Sibling" pointers is traversed and added to the stack S, thereby making sure that the subtrees are re-visited and the newly inserted item is found.

```
  Algorithm 2: PARALLEL_OLAP_QUERY
Input: R (MDS of the given query range).
Local Variables: Directory node D. Stack S.
Output: Result.
  (1) Set D=root. 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 /* interfering parallel update */
```

```
        (4.1) Using the "Link to Sibling" field in directory
              nodes, traverse 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) IF MDS of C is contained in R THEN
                      add C to Result.
              (4.3.3) IF MDS of C overlaps R but is not
                      contained in R THEN push C into stack
                      S.
UNTIL stack S is empty.
— End of Algorithm —
```

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 which could potentially affect the scalability of our method. To which degree this does actually happen will be measured in our experimental evaluation in Section III.

III. Experimental Evaluation

The main goal of our paper is the design of a *parallel* real-time OLAP system for multi-core processors which scales and provides increasing speedup as the number of processor cores increases. Since, to the best of our knowledge, there are no other published *parallel* real-time OLAP systems for multi-core processors, we can not provide comparison data. The focus of our experimental evaluation will be on how our method scales as the number of available processor cores increases.

An important question for the experimental study is the choice of input data. For a real-time OLAP system, the input consists of a stream of *OLAP_INSERT* and *OLAP_QUERY* transactions. One possibility would be to use random insert and random query operations. However, in our discussions with the data cube team at IBM/COGNOS Canada it became clear that such a test would not provide a realistic measure for the performance of our system. The IBM/COGNOS team recommended the *TPC-DS "Decision Support" Benchmark* by the *Transaction Processing Performance Council* [21] as a realistic test data set. This benchmark is available to TPC members and we obtained access through IBM/COGNOS. The TPC-DS benchmark provides transactions which 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.) As indicated on the TPC-DS website, "although the underlying business model of TPC-DS 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".

The hardware platform for our experiments was a *Sandy Bridge* multi-core processor (8 hardware cores, 16 cores with hardware supported hyperthreading) with 16 GB memory. Our system was implemented in C++ with OpenMP, and compiled/executed on Linux kernel 2.6.38 using g++ 4.5.2.

Figures 4(a&b) show the average transaction response time and throughput when 400,000 *PARALLEL_OLAP_INSERT* operations (with TPC-DS data) are executed on 1, 2, 4, 8, and 16 processor cores, respectively. Figures 4(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 speedup for Figures 4(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 a very encouraging result for a fully running system on real-world data. Figures 4(a&b) show a somewhat smaller speedup for the first 400,000 transactions into an empty database. This is expected because for smaller data sets, the tree data structure is smaller and the likelihood of interference between transactions is therefore larger. In general, we observed that the speedup with increasing number of processor cores is better for larger data warehouses. Therefore, all remaining experiments in this section are on a modest size initial data warehouse with 400,000 pre-loaded data items. For larger data warehouses, the speedup results would be further improved.

The most important results of our experimental evaluation are shown in Figures 5, 6 and 7. Here we show the performance (transaction response time and transaction throughput) for mixed sets of *PARALLEL_OLAP_INSERT* and *PARALLEL_OLAP_QUERY* operations (using TPC-DS benchmark data) executed on 1, 2, 4, 8, and 16 processor cores, respectively. Please note that the single thread version 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 (I OLAP insertions and Q OLAP queries). Since the performance of a (sequential or parallel) OLAP query is strongly influenced by the size of the query result, we provide three different graphs, Figure 5, Figure 6 and Figure 7 for queries that report as output 1%, 5% and 25% of the entire database, respectively. In each case, we show response time and throughput for 1, 2, 4, 8, and 16 processor cores. 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 in all cases shown we achieve a close to linear speedup, as demonstrated by the straight (linear) curves in the log-log scale. Considering that the cores of the

Sandy Bridge processor share resources (e.g. memory bus), this is a very encouraging result for a fully running system on real-world benchmark data. Since hardware performance improvements are nowadays achieved by increasing the number of processor cores, our new *parallel* DC-tree method appears to have the potential to enable OLAP systems that are *real-time* and efficient/feasible for large databases.

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 need to search subtrees and may need to report large size results. However, in the mix of parallel insert and query transactions on the same database, the speedup observed was best for the hardest case of "only queries", which is where speedup is most needed in practice.

Finally, Figure 8 shows that speedup is also maintained for "low load" scenarios with only 16 parallel queries. Figures 8(a & b) show average response time and throughput for a set of 16 insertions, and Figures 8(c & d) show average response time and throughput for three sets of 16 queries with results of size 1%, 4% and 25% of the database.

IV. Conclusions

In this paper, we have presented a *parallel* real-time OLAP method for multi-core processors and demonstrated on real-life TPC-DS "Decision Support" benchmark data that our system scales and provides increasing speedup as the number of processor cores increases. Since hardware performance improvements are currently 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 are *real-time* and efficient/feasible for large databases.

V. Achnowledgements

This research project was financially supported by the IBM Center for Advanced Studies Canada. We also acknowledge the help of H.-P. Kriegel and his group [18] who provided us with code segments for sequential DC tree operations.

References

[1] J. Han and M. Kamber, *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2000.

[2] "The OLAP Report," http://www.olapreport.com.

[3] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh, "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals," *Data Min. Know. Disc.*, vol. 1, pp. 29–53, 1997.

[4] Y. Chen, F. Dehne, T. Eavis, and A. Rau-Chaplin, "PnP: sequential, external memory, and parallel iceberg cube computation," *Distributed and Parallel Databases*, vol. 23, no. 2, pp. 99–126, Jan. 2008. [Online]. Available: http://www.springerlink.com/index/10.1007/s10619-007-7023-y

[5] F. Dehne, T. Eavis, and S. Hambrusch, "Parallelizing the data cube," *Distributed and Parallel Databases*, vol. 11, pp. 181–201, 2002. [Online]. Available: http://www.springerlink.com/index/BGN4YJUMUBPELXK0.pdf

[6] Z. Guo-Liang, C. Hong, L. Cui-Ping, W. Shan, and Z. Tao, "Parallel Data Cube Computation on Graphic Processing Units," *Chines Journal of Computers*, vol. 33, no. 10, pp. 1788–1798, 2010. [Online]. Available: http://cjc.ict.ac.cn/eng/qwjse/view.asp?id=3197

[7] R. T. Ng, A. Wagner, and Y. Yin, "Iceberg-cube computation with PC clusters," *ACM SIGMOD*, vol. 30, no. 2, pp. 25–36, Jun. 2001. [Online]. Available: http://portal.acm.org/citation.cfm?doid=376284.375666

[8] J. You, J. Xi, P. Zhang, and H. Chen, "A Parallel Algorithm for Closed Cube Computation," *IEEE/ACIS International Conference on Computer and Information Science*, pp. 95–99, May 2008. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4529804

[9] R. Bruckner, B. List, and J. Schiefer, "Striving towards near real-time data integration for data warehouses," *DaWaK*, vol. LNCS 2454, pp. 173–182, 2002. [Online]. Available: http://www.springerlink.com/index/G5T567NVR9AA96XQ.pdf

[10] D. Jin, T. Tsuji, and K. Higuchi, "An Incremental Maintenance Scheme of Data Cubes and Its Evaluation," *DASFAA*, vol. LNCS 4947, pp. 36–48, 2008. [Online]. Available: http://joi.jlc.jst.go.jp/JST.JSTAGE/ipsjtrans/2.36?from=CrossRef

[11] R. Santos and J. Bernardino, "Real-time data warehouse loading methodology," *IDEAS*, pp. 49–58, 2008. [Online]. Available: http://portal.acm.org/citation.cfm?id=1451949

[12] R. J. Santos and J. Bernardino, "Optimizing data warehouse loading procedures for enabling useful-time data warehousing," *IDEAS*, pp. 292–299, 2009. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1620432.1620464

[13] D. Banks, "High-Concurrency Locking in R-Trees," *VLDB*, pp. 1–12, 1995.

[14] K. Chakrabarti, "Efficient Concurrency Control in Multidimensional Access Methods," *ACM SIGMOD*, pp. 25–36, 1999.

[15] J. R. Haritsa, S. Member, and S. Seshadri, "Real-Time Index Concurrency Control," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, no. 3, pp. 429–447, 2000.

[16] M. L. Lee, W. Hsu, C. S. Jensen, B. Cui, and K. L. Teo, "Supporting Frequent Updates in R-Trees : A Bottom-Up Approach," *VLDB*, 2003.

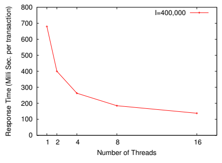
[17] Jing Dai, "Efficient Concurrent Operations in Spatial Databases," *PhD Thesis, Virginia Polytechnic*, 2009.

[18] M. Ester, J. Kohlhammer, and H.-P. Kriegel, “The DC-tree: a fully dynamic index structure for data warehouses,” *16th International Conference on Data Engineering (ICDE)*, pp. 379–388, 2000. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=839438

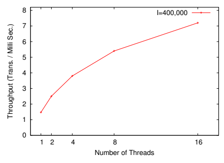
[19] S. Berchtold, D. A. Keim, and H.-P. Kriegel, “The X-tree: An index structure for high-dimensional data,” *VLDB*, pp. 28–39, Oct. 1996.

[20] A. Guttman, “R-trees: a dynamic index structure for spatial searching,” *ACM SIGMOD*, pp. 47–57, 1984. [Online]. Available: http://portal.acm.org/citation.cfm?id=602266

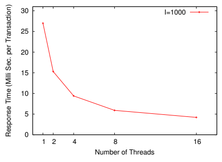
[21] Transaction Processing Performance Council, “TPC-DS (Decision Support) Benchmark,” *http://www.tpc.org/tpcds/tpcds.asp*.



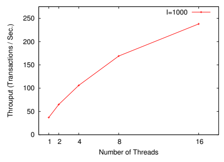
(a)



(b)

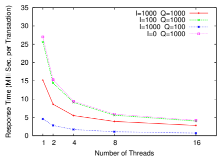


(c)

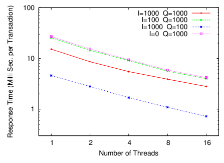


(d)

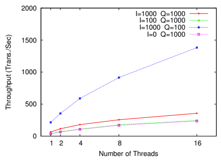
Figure 4. Parallel OLAP insertion performance as a function of the number of parallel threads. (a) & (b) Average transaction response time & throughput for 400,000 OLAP insertions to build an initial database. (c) & (d) Average transaction response time & throughput for a subsequent 1,000 OLAP insertions into the built database.



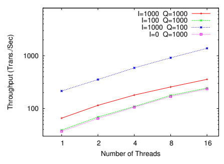
(a)



(b)

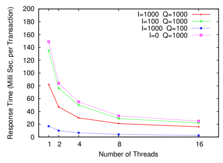


(c)

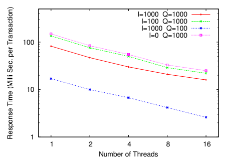


(d)

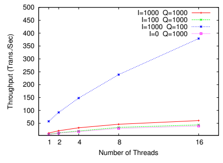
Figure 5. Parallel OLAP transaction performance as a function of the number of parallel threads. Mixed input of I OLAP insertions and Q OLAP queries. Queries return 1% of database. (a) Average response time, linear scale. (b) Average response time, log-log scale. (c) Throughput, linear scale. (d) Throughput, log-log scale.



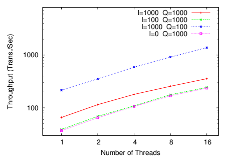
(a)



(b)

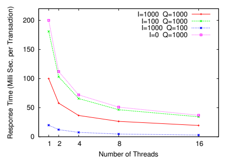


(c)

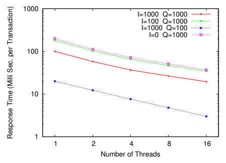


(d)

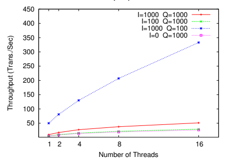
Figure 6. Parallel OLAP transaction performance as a function of the number of parallel threads. Mixed input of I OLAP insertions and Q OLAP queries. Queries return 5% of database. (a) Average response time, linear scale. (b) Average response time, log-log scale. (c) Throughput, linear scale. (d) Throughput, log-log scale.



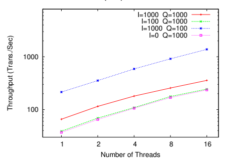
(a)



(b)

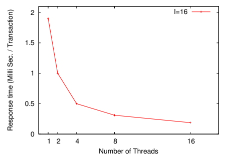


(c)

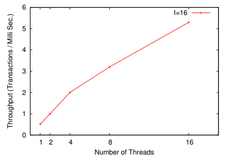


(d)

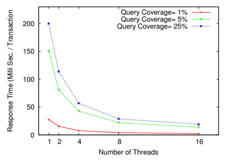
Figure 7. Parallel OLAP transaction performance as a function of the number of parallel threads. Mixed input of I OLAP insertions and Q OLAP queries. Queries return 25% of database. (a) Average response time, linear scale. (b) Average response time, log-log scale. (c) Throughput, linear scale. (d) Throughput, log-log scale.



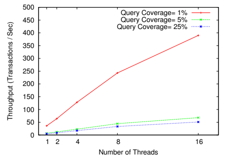
(a)



(b)



(c)



(d)

Figure 8. Parallel OLAP transaction performance for low load scenarios of only 16 parallel queries. (a) & (b) Average response time & throughput for OLAP insertions. (c) & (d) Average response time & throughput for OLAP queries with different coverage (percentage of database returned by query).