J. Parallel Distrib. Comput. 79–80 (2015) 31–41

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Scalable real-time OLAP on cloud architectures ☆

F. Dehne [a,*], Q. Kong [b], A. Rau-Chaplin [b], H. Zaboli [a], R. Zhou [a]

[a] School of Computer Science, Carleton University, Ottawa, Canada
[b] Faculty of Computer Science, Dalhousie University, Halifax, Canada

HIGHLIGHTS

- Collaboration with the IBM on alleviating performance bottlenecks for OLAP queries.
- OLAP queries may aggregate large portions of the database, creating bottlenecks.
- We study the use of parallel computing on scalable clouds to accelerate queries.
- Our system, CR-OLAP, is based on a new scalable distributed index structure.
- CR-OLAP uses dynamic cloud elasticity to improve performance.

ARTICLE INFO

Article history:
Received 14 November 2013
Received in revised form
7 May 2014
Accepted 18 August 2014
Available online 6 September 2014

Keywords:
Real-time OLAP
Scalability
Cloud architecture
TPC-DS benchmark

ABSTRACT

In contrast to queries for on-line *transaction* processing (OLTP) systems that typically access only a small portion of a database, OLAP queries may need to aggregate large portions of a database which often leads to performance issues. In this paper we introduce *CR-OLAP*, a scalable Cloud based Real-time *OLAP* system based on a new distributed index structure for OLAP, the *distributed PDCR tree*. *CR-OLAP* utilizes a scalable cloud infrastructure consisting of multiple commodity servers (processors). That is, with increasing database size, *CR-OLAP* dynamically increases the number of processors to maintain performance. Our distributed PDCR tree data structure supports multiple dimension hierarchies and efficient query processing on the elaborate dimension hierarchies which are so central to OLAP systems. It is particularly efficient for complex OLAP queries that need to aggregate large portions of the data warehouse, such as "report the total sales in all stores located in California and New York during the months February–May of all years". We evaluated *CR-OLAP* on the Amazon EC2 cloud, using the TPC-DS benchmark data set. The tests demonstrate that *CR-OLAP* scales well with increasing number of processors, even for complex queries. For example, for an Amazon EC2 cloud instance with 16 processors, a data warehouse with 160 million tuples, and a TPC-DS OLAP query stream where each query aggregates between 60% and 95% of the database, *CR-OLAP* achieved a query latency of below 0.3 s which can be considered a *real time* response.

1. Introduction

On-line analytical processing (OLAP) systems are at the heart of many *business analytics* applications. This paper reports on the results of a research project (supported by the IBM Centre For Advanced Studies Canada) to investigate the use of *cloud computing* for high performance, scalable *real-time* OLAP.

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 states of an organization. DSS allow users to study relationships in a chronological context between things such as customers, vendors, products, inventory, geography, and sales. 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 [18,27]. By exploiting multi-dimensional views of the underlying data warehouse, the OLAP server allows users to "drill down" or "roll up" on dimension hierarchies, "slice and dice" particular attributes,

☆ A preliminary (short) version appeared in the conference proceeding of *IEEE Big Data 2013*.

* Corresponding author.
E-mail addresses: frank@dehne.net (F. Dehne), qkong@cs.dal.ca (Q. Kong), arc@cs.dal.ca (A. Rau-Chaplin), zaboli@graduate.org (H. Zaboli), xiaoyunzhou@cmail.carleton.ca (R. Zhou).

http://dx.doi.org/10.1016/j.jpdc.2014.08.006
0743-7315/

or perform various statistical operations such as ranking and forecasting. To support this functionality, OLAP relies heavily upon a classical data model known as the data cube [15] which 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.

In contrast to queries for on-line *transaction* processing (OLTP) systems which typically access only a small portion of the database (e.g. update a customer record), OLAP queries may need to aggregate large portions of the database (e.g. calculate the total sales of a certain type of items during a certain time period) which may lead to performance issues. Therefore, most of the traditional OLAP research, and most of the commercial systems, follow the *static* data cube approach proposed by Gray et al. [15] 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. [6,8,15,16,22,33]). 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 on-line transaction processing (OLTP) system of the organization. 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. [4]). Some recent publications have begun to address this problem by providing "quasi real-time" incremental maintenance schemes and loading procedures for static data cubes (e.g. [4,19,24,25]). However, these approaches are not fully real-time. A major obstacle is significant performance issues with large scale data warehouses.

1.2. Contributions

The aim of our research project is to help address the above mentioned performance issues for *real-time* OLAP systems through the use of efficient *parallel* computing methods. In a recent paper [9] we presented the first *parallel* real-time OLAP system designed to be executed on a *multi-core* processor. We documented significant performance increases with increasing number of processor cores. Our system won the 2012 *IBM Canada Research Impact of the Year Award* and an IBM sponsored patent application has been submitted. In this paper, we report on the next phase of our project: to scale up our real-time OLAP system to utilize a collection of $(m + 1)$ multi-core processors in a scalable *cloud* environment.

We introduce *CR-OLAP*, a scalable *C*loud based *R*eal-time *OLAP* system that utilizes a new distributed index structure for OLAP, referred to as a *distributed PDCR tree*. This data structure is not just another distributed R-tree, but rather a multi-dimensional data structure designed specifically to support efficient OLAP query processing on the elaborate dimension hierarchies that are central to OLAP systems. The *distributed PDCR tree*, based on the sequential DC tree introduced by Kriegel et al. [12] and our previous PDC tree [9], exploits knowledge about the structure of individual dimension hierarchies both for compact data representation and accelerated query processing. The following is a brief overview of the properties of our system.

Consider a d-dimensional data warehouse with d dimension hierarchies. *CR-OLAP* supports an input stream consisting of *insert* and *query* operations. Each OLAP query can be represented as an aggregate range query that specifies for each dimension either a single value or range of values at any level of the respective dimension hierarchy, or a symbol "*" indicating the entire range for that dimension. *CR-OLAP* utilizes a cloud infrastructure consisting of $m+1$ multi-core processors where each processor executes up to k parallel threads. As typical for current high performance databases, all data are kept in the processors' main memories [23]. With increasing database size, *CR-OLAP* will increase m by dynamically allocating additional processors within the cloud environment and re-arranging the distributed PDCR tree. This will ensure that both, the available memory and processing capability will scale with the database size. One of the $m+1$ multi-core processors is referred to as the *master*, and the remaining m processors are called *workers*. The master receives from the users the input stream of OLAP *insert* and *query* operations, and reports the results back to the users (in the form of references to memory locations where the workers have deposited the query results). In order to ensure high throughput and low latency even for compute intensive OLAP queries that may need to aggregate large portions of the entire database, *CR-OLAP* utilizes several levels of parallelism: distributed processing of multiple query and insert operations among multiple workers, and parallel processing of multiple concurrent query and insert operations within each worker. For correct query operation, *CR-OLAP* ensures that the result for each OLAP query includes all data inserted prior but no data inserted after the query was issued within the input stream.

CR-OLAP is supported by a new distributed index structure for OLAP termed *distributed PDCR tree* which supports distributed OLAP query processing, including fast real-time data aggregation, real-time querying of multiple dimension hierarchies, and real-time data insertion. Note that, since OLAP is about the analysis of historical data collections, OLAP systems usually do not support data deletion. Our system does however support *bulk insert* operations of large groups of data items.

The distributed index structure consists of a collection of PDCR trees whereby the master stores one PDCR tree (called *hat*) and each worker stores multiple PDCR trees (called *subtrees*). Each individual PDCR tree supports multi-core parallelism and executes multiple concurrent *insert* and *query* operations at any point in time. PDCR trees are a non-trivial modification of the authors' previously presented PDC trees [9], adapted to the cloud environment and designed to scale. For example, PDCR trees are array based so that they can easily be compressed and transferred between processors via message passing. When the database grows and new workers are added, sub-trees are split off and sent to the new worker.

We evaluated *CR-OLAP* on the Amazon EC2 cloud 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 *CR-OLAP* scales well with increasing number of workers. For example, for fixed data warehouse size (10,000,000 data items), when increasing the number of workers from 1 to 8, the average query throughput and latency improves by a factor 7.5. When increasing the data warehouse size from 10,000,000 data items to 160,000,000 data items while, at the same time, letting *CR-OLAP* increase the number of workers used from 1 to 16, respectively, we observed that query performance remained essentially unchanged. That is, the system performed a 16-fold increase in size, including a 16-fold increase in the average amount of data aggregated by each OLAP query, without noticeable performance impact for the user.

A particular strength of *CR-OLAP* is to efficiently answer queries with large query *coverage*, i.e. the portion of the database that needs to be aggregated for an OLAP query. For example, for an

Amazon EC2 cloud instance with 16 processors, a data warehouse with 160 million tuples, and a TPC-DS OLAP query stream where each query aggregates between 60% and 95% of the database, *CR-OLAP* achieved a query latency of below 0.3 s which can be considered a *real time* response. *CR-OLAP* also handles well increasing dimensionality of the data warehouse. For tree data structures this is a critical issue as it is known e.g. for R-trees that, with increasing number of dimensions, even simple range search (no dimension hierarchies, no aggregation) can degenerate to linear search (e.g. [13]). In our experiments, we observed that increasing number of dimensions does not significantly impact the performance of *CR-OLAP*. Another possible disadvantage of tree data structures is that they are potentially less cache efficient than in-memory linear search which can make optimum use of streaming data between memory and processor caches. To establish a comparison baseline for *CR-OLAP*, we implemented *STREAM-OLAP* which partitions the database between multiple cloud processors based on one chosen dimension and uses parallel memory to cache streaming on the cloud processors to answer OLAP queries. We observed that the performance of *CR-OLAP* is similar to *STREAM-OLAP* for simple OLAP queries with small query coverage but that *CR-OLAP* vastly outperforms *STREAM-OLAP* for more complex queries that utilize different dimension hierarchies and have a larger query coverage (e.g. "report the total sales in all stores located in California and New York during the months February–May of all years").

The remainder of this paper is organized as follows. In Section 2 we review related work. In Section 3 we introduce the PDCR tree data structure and in Section 4 we present our *CR-OLAP* system for real-time OLAP on cloud architectures. Section 5 shows the results of an experimental evaluation of *CR-OLAP* on the Amazon EC2 cloud, and Section 6 concludes the paper.

2. Related work

In addition to the related work discussed in the introduction, there are many efforts to store and query large data sets in cloud environments. Hadoop [17] and its file system, HDFS, are popular examples of such systems which are typically built on MapReduce [7]. Related projects most similar to our work are Hive [28] and HadoopDB [1]. However, these systems are *not* designed for *real-time* (OLTP style) operation. Instead, they use batch processing similar to [4,19,24,25]. The situation is similar for BigTable [5], BigQuery [3], and Dremel [21]. In fact, Dremel [21] uses a columnar data representation scheme and is designed to provide data warehousing and querying support for read-only data. To overcome the batch processing in Hadoop based systems, Storm [30] introduced a distributed computing model that processes in-flight Twitter data. However, Storm assumes small, compact Twitter style data packets that can quickly migrate between different computing resources. This is not possible for large data warehouses.

SAP HANA is a real time in-memory database system that also supports OLAP queries. In a cloud computing environment, a basic HANA instance is for a multi-core processor single compute node. A scale out version of HANA can be executed on multiple compute nodes, using a distributed file system (GPFS) that provides a single shared data view to all compute nodes. Large tables can be partitioned using various partitioning criteria and complete tables or parts thereof can then be assigned to different nodes. The execution engine schedules queries over the different compute nodes and attempts to execute them on the node that holds the data [14].

For peer-to-peer networks, related work includes distributed methods for querying concept hierarchies such as [11,2,10,26]. However, none of these methods provide *real-time* OLAP functionality. There are various publications on distributed B-trees for cloud platforms such as [32]. However, these method only supports

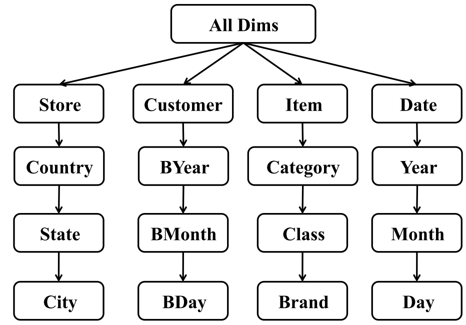


Fig. 1. A 4-dimensional data warehouse with 3 hierarchy levels for each dimension. The first box for each dimension denotes the name of the dimension.

1-dimensional indices which are insufficient for OLAP queries. There have been efforts to build distributed multi-dimensional indices on *Cloud* platforms based on R-trees or related multi-dimensional tree structures, such as [31,35,20]. However, these methods do not support dimension hierarchies which are essential for OLAP queries.

3. PDCR trees

Consider a data warehouse with fact table F and a set of d dimensions $\{D_1, D_2, \ldots, D_d\}$ where each dimension D_i, $1 \leq i \leq d$ has a hierarchy H_i including hierarchical attributes corresponding to the levels of the hierarchy. The hierarchical attributes in the hierarchy of dimension i are organized as an ordered set H_i of parent–child relationships in the hierarchy levels $H_i = \{H_{i1}, H_{i2}, \ldots, H_{il}\}$ where a parent logically summarizes and includes its children. Fig. 1 shows the dimensions and hierarchy levels of each dimension for a 4-dimensional data warehouse.

The *sequential* DC tree introduced by Kriegel et al. [12] exploits knowledge about the structure of individual dimension hierarchies both for compact data representation and accelerated OLAP query processing. In our previous work [9], we introduced the PDC tree, a *parallel* DC tree for a single multi-core processor. In this section, we outline a modified PDC tree, termed PDCR tree, which will become the building block for our *CR-OLAP* system. Here, we only outline the differences between the PDCR tree and its predecessors, and we refer to [9,12] for more details. We note that, our PDCR tree data structure is not just another distributed R-tree, but rather a multi-dimensional data structure designed specifically to support efficient OLAP query processing on the elaborate dimension hierarchies that are central to OLAP systems. Also note that, DC tree [12] is particularly well suited to handle OLAP queries for both, ordered and unordered dimensions.

For a cloud architecture with multiple processors, each processor will store one or more PDCR trees. Our *CR-OLAP* system outlined in the following Section 4 requires that a sub-tree of a PDCR tree can be split off and transferred to another processor. This required us to (a) devise an array based tree implementation that can be packed into a message to be sent between processors and (b) a careful encoding of data values, using compact IDs related to the different dimension hierarchy levels. For our array based PDCR tree implementation, a PDC tree is represented in a single array where all tree links are represented by integer references to array locations. Allocation of new tree nodes was re-implemented as an append operation at the end of the array, and all tree operations were re-implemented to use integer references instead of memory pointers. In the following we outline some details of the encoding of data values used for our PDCR tree.

IDs for each dimension represent available entities in the dimension. Each dimension has a hierarchy of entities with l levels. In the example of Fig. 1, an ID may represent an entity at the

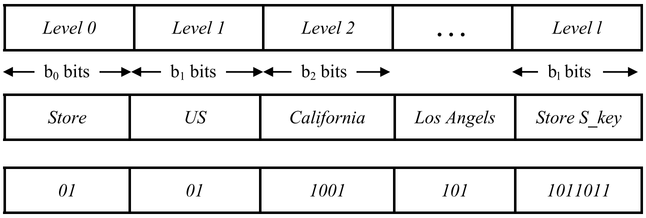


Fig. 2. Illustration of the compact bit representation of IDs.

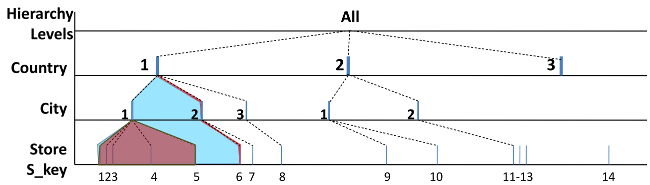


Fig. 3. Example of relationships between different hierarchy levels of a given dimension.

Country level for the Store dimension, e.g. US or Canada. Similarly, another ID may represent an entity at the City level, e.g. Chicago or Toronto. It is important to note that an ID may summarize many IDs at lower hierarchy levels. To build an ID for a dimension with l levels, we assign b_j bits to the hierarchy level j, $0 \leq j \leq l-1$. Different entities at each hierarchy level are assigned numerical values starting with "1". By concatenating the numerical values of the levels, a numerical value is created. We reserve the value zero to represent "All" or "*". The example in Fig. 2 shows an example of an entity at the lowest hierarchy level of dimension *Store*. An ID for the state *California* will have a value of zero for its descendant levels *City* and *Store S_key*. As a result, containment of IDs between different hierarchy levels can be tested via fast bit operations. Fig. 3 illustrates IDs and their coverages in the *Store* dimension with respect to different hierarchy levels. As illustrated, each entity in level j (*Country*) is a country specified by a numerical value and covers cities that are represented using numerical values in level $j+1$. Note that IDs used for cities will have specific values at the city level, while the ID of a country will have a value of zero at the city level and a specific value only at the country level.

The *sequential* DC tree introduced by Kriegel et al. [12] and our previous PDC tree [9] store the so called "minimum describing set" (MDS) entries at each internal tree node to guide the query process; see [12] for details. The MDS concept was developed in [12] to better represent unordered dimensions with dimension hierarchies. Experiments with our *CR-OLAP* system showed that in a larger cloud computing environment with multiple tree data structures, the number of MDS entries becomes very large and unevenly distributed between the different trees, leading to performance bottlenecks. On the other hand, the bit representation of IDs outlined above gives us the opportunity to convert unordered dimensions into ordered dimensions, and then use traditional ranges instead of the MDS entries. An example is shown in Fig. 4. The ranges lead to a much more compact tree storage and alleviated the above mentioned bottleneck. It is important to note that, this internal ordering imposed on dimensions is invisible to the user. OLAP queries can still include unordered aggregate values on any dimension such as "*Total sales in the US and Canada*" or "*Total sales in California and New York*".

4. CR-OLAP: cloud based real-time OLAP

CR-OLAP utilizes a cloud infrastructure consisting of $m+1$ multi-core processors where each processor executes up to k parallel threads. One of the $m+1$ multi-core processors is referred to as the *master*, and the remaining m processors are called *workers*.

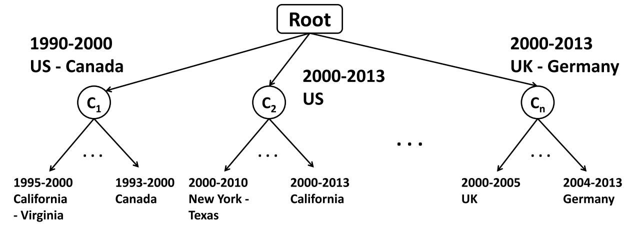


Fig. 4. Example of a PDCR tree with 2 dimensions (Store and Date).

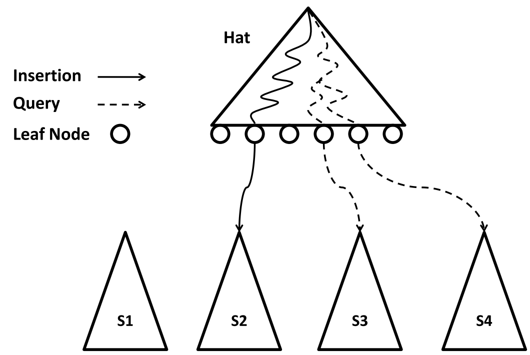


Fig. 5. Illustration of a distributed PDCR tree.

The master receives from the users the input stream of OLAP *insert* and *query* operations, and reports the results back to the users (in the form of references to memory locations where the workers have deposited the query results). In order to ensure high throughput and low latency even for compute intensive OLAP queries that may need to aggregate large portions of the entire database, *CR-OLAP* utilizes several levels of parallelism: distributed processing of multiple query and insert operations among multiple workers, and parallel processing of multiple concurrent query and insert operations within each worker. With increasing database size, *CR-OLAP* will increase m by dynamically allocating additional processors within the cloud environment and re-arranging the distributed PDCR tree. This will ensure that both, the available memory and processing capability will scale with the database size.

We start by outlining the structure of a *distributed PDC tree and PDCR tree* on $m+1$ multi-core processors in a cloud environment. Consider a single PDCR tree T storing the entire database. For a tunable *depth* parameter h, we refer to the top h levels of T as the *hat* and we refer to the remaining trees rooted at the leaves of the *hat* as the *subtrees* $s_1, \ldots, s_n$. Level h is referred to as the *cut level*. The *hat* will be stored at the master and the *subtrees* $s_1, \ldots, s_n$ will be stored at the m workers. We assume $n \geq m$ and that each worker stores one or more subtrees.

CR-OLAP starts with an empty database and one master processor (i.e. $m = 0$) storing an empty *hat* (PDCR tree). Note that, DC trees [12], PDC trees [9] and PDCR trees are leaf oriented. All data are stored in leafs called *data nodes*. Internal nodes are called *directory nodes* and contain arrays with routing information and aggregate values. Directory nodes have a high capacity and fan-out of typically 10–20. As insert operations are sent to *CR-OLAP*, the size and height of the *hat* (PDCR tree) grows. When directory nodes of the *hat* reach height h, their children become roots at subtrees stored at new worker nodes that are allocated through the cloud environment. An illustration of such a distributed PDCR tree is shown in Fig. 5.

For a typical database size, the *hat* will usually contain only directory nodes and all data will be stored in the subtrees $s_1, \ldots, s_n$. After the initial set of data insertions, all leaf nodes in the *hat* will usually be directory nodes of height h, and the roots of subtrees in workers will typically be directory nodes as well. As illustrated in Fig. 5, both *insert* and *query* operations are executed concurrently.

4.1. Concurrent insert and query operations

Each *query* operation in the input stream is handed to the master which traverses the *hat*. Note that, at each directory node the query can generate multiple parallel threads, depending on how many child nodes have a non empty intersection with the query. Eventually, each query will access a subset of the *hat*'s leaves, and then the query will be transferred to the workers storing the subtrees rooted at those leaves. Each of those workers will then in parallel execute the query on the respective subtrees, possibly generating more parallel threads within each subtree. For more details see Algorithms 3 and 4.

For each *insert* operation in the input stream, the master will search the *hat*, arriving at one of the leaf nodes, and then forward the insert operation to the worker storing the subtree rooted at that leaf. For more details see Algorithms 1 and 2.

Algorithm 1: Hat_Insertion

Input: D (new data item).
Output: void
Initialization:
Set $ptr = root$

Repeat:
Determine the child node C of ptr that causes minimal MBR/MDS enlargement for the distributed PDCR/PDC tree if D is inserted under C. Resolve ties by minimal overlap, then by minimal number of data nodes.
Set $ptr = C$.
Acquire a LOCK for C.
Update MBR/MDS and TS of C.
Release the LOCK for C.
Until: *ptr* is a *leaf* node.
if *ptr is the parent of Data Nodes* **then**
  Acquire a LOCK for ptr.
  Insert D under ptr.
  Release the LOCK for C.
  if *capacity of ptr is exceeded* **then**
    Call Horizontal Split for ptr.
    if *capacity of the parent of ptr is exceeded* **then**
      Call Vertical Split for the parent of ptr.
      if *depth of ptr is greater than h* **then**
        Create a new subtree with the parent of ptr as its root, ptr and its sibling node as the children of the root.
        Choose the next available worker and update the list of subtrees in the master.
        Send the new subtree and its data nodes to the chosen worker.
      end
    end
  end
end
if *ptr is the parent of a subtree* **then**
  Find the worker that contains the subtree from the list of subtrees.
  Send the insertion transaction to the worker.
end
End of Algorithm.

Figs. 6 and 7 illustrate how *new* workers and *new* subtrees are added as more data items get inserted. Fig. 6 illustrates insertions creating an overflow at node A, resulting in a horizontal split at A into A_1 and A_2 plus a new parent node C. Capacity overflow at C then triggers a vertical split illustrated in 7. This creates two subtrees in two different workers. As outlined in more details in the *CR-OLAP* "migration strategies" outlined below, new workers are requested from the cloud environment when either new subtrees are created or when subtree sizes exceed the memory of their host workers. Workers usually store multiple subtrees. However, *CR-OLAP* randomly shuffles subtrees among workers. This ensures that *query* operations accessing a contiguous range of leaf nodes in the *hat* create a distributed workload among workers.

For correct *real time* processing of an input stream of mixed *insert* and *query* operations, *CR-OLAP* needs to ensure that the result for each OLAP query includes all data inserted prior but no data inserted after the query was issued within the input stream. We will now discuss how this is achieved in a distributed cloud based system where we have a collection of subtrees in different workers, each of which is processing multiple concurrent *insert* and *query* threads. In our previous work [9] we presented a method to ensure correct query processing for a single PDC tree on a mutli-core processor, where multiple *insert* and *query* operations are processed concurrently. The PDC tree maintains for each data or

Algorithm 2: Subtree_Insertion

Input: D (new data item).
Output: void
Initialization:
Set $ptr = root$

Repeat:
Determine the child node C of ptr that causes minimal MBR/MDS enlargement for the distributed PDCR/PDC tree if D is inserted under C. Resolve ties by minimal overlap, then by minimal number of data nodes.
Set $ptr = C$.
Acquire a LOCK for C.
Update MBR/MDS and TS of C.
Release the LOCK for C.
Until: *ptr* is a *leaf* node.
Acquire a LOCK for *ptr*.
Insert D under *ptr*.
Release the LOCK for C.
if *capacity of ptr is exceeded* **then**
  Call Horizontal Split for *ptr*.
  if *capacity of the parent of ptr is exceeded* **then**
    Call Vertical Split for the parent of *ptr*.
  end
end
End of Algorithm.

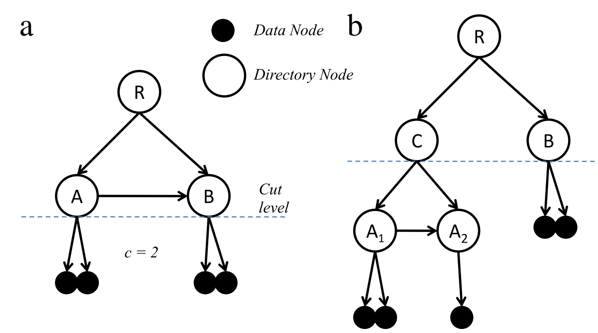


Fig. 6. Insertions triggering creation of *new* workers and subtrees. Part 1. (a) Current *hat* configuration. (b) Insertions create overflow at node *A* and horizontal split.

Algorithm 3: Hat_Query

Input: Q (OLAP query).
Output: A result set or an aggregate value
Initialization:
Set $ptr = root$
Push *ptr* into a local stack *S* for query *Q*.

Repeat:
Pop the top item ptr' from stack *S*.
if *TS(time stamp) of* ptr' *is smaller (earlier) than the TS of ptr* **then**
  Using the *sibling links*, traverse the sibling nodes of *ptr* until a node with TS equal to the TS of *ptr* is met. Push the visited nodes including *ptr* into the stack (starting from the rightmost node) for reprocessing.
end
for *each child C of ptr* **do**
  if *MBR/MDS of C is fully contained in MBR/MDS of Q* **then**
    Add *C* and its measure value to the result set.
  end
  else
    if *MBR/MDS of C overlaps MBR/MDS of Q* **then**
      if *C is the root of a sub-tree* **then**
        Send the query *Q* to the worker that contains the subtree.
      end
      else
        Push *C* into the stack *S*.
      end
    end
  end
end
Until: stack *S* is empty.
if *the query Q is dispatched to a subtree* **then**
  Wait for the partial results of the dispatched queries from workers.
  Create the final result of the collected partial results.
  Send the final result back to the client.
end
End of Algorithm.

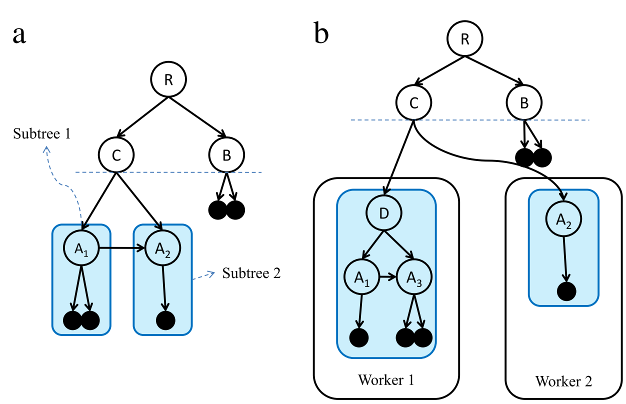


Fig. 7. Insertions triggering creation of *new* workers and subtrees. Part 2. (a) Same as Fig. 6(b) with critical subtrees highlighted. (b) Insertions create overflow at node *C* and vertical split, triggering the creation of two subtrees in two different workers.

directory item a time stamp indicating its most recent update, plus it maintains for all nodes of the same height a left-to-right linked list of all siblings. Furthermore, each *query* thread maintains a stack of ancestors of the current node under consideration, together with the time stamps of those items. We refer to [9] for more details. The PDCR tree presented in this paper inherits this mechanism

Algorithm 4: Subtree_Query

Input: Q (OLAP query).
Output: A result set or an aggregate value
Initialization:
Set $ptr = root$
Push *ptr* into a local stack *S* for query *Q*.

Repeat:
Pop the top item ptr' from stack *S*.
if *TS(time stamp) of* ptr' *is smaller (earlier) than the TS of ptr* **then**
  Using the *sibling links*, traverse the sibling nodes of *ptr* until a node with TS equal to the TS of *ptr* is met. Push the visited nodes including *ptr* into the stack starting from the rightmost node for reprocessing.
end
for *each child C of ptr* **do**
  if *MBR/MDS of C is fully contained in MBR/MDS of Q* **then**
    Add *C* and its measure value to the result set.
  end
  else
    if *MBR/MDS of C overlaps MBR/MDS of Q* **then**
      Push *C* into the stack *S*.
    end
  end
end
Until: stack *S* is empty.
Send the result back to the master or client depending on whether *Q* is an aggregation query or a data report query.
End of Algorithm.

for each of its subtrees. In fact, the above mentioned *sibling links* are shown as horizontal links in Figs. 6 and 7. With the PDCR tree being a collection of subtrees, if we were to maintain sibling links between subtrees to build linked list of siblings across all subtrees, then we would ensure correct query operation in the same way as for the PDC tree [9]. However, since different subtrees of a PDCR tree typically reside on different workers, a PDCR tree only maintains sibling links inside subtrees but it does *not* maintain sibling links between different subtrees. The following lemma shows that correct *real time* processing of mixed *insert* and *query* operations is still maintained.

Theorem 1. *Consider the situation depicted in Fig. 8 where the split of node B created a new node D and subtree rooted at D that is stored separately from the subtrees rooted at A and B. Then, the sibling links labeled "a" and "c" are not required for* correct *real time query processing (as defined above).*

Proof. Assume a thread for a query *Q* that is returning from searching the subtree below *B* only to discover that *B* has been modified. Let B_{stack} be the old value of *B* that is stored in the stack $stack(Q)$ associated with *Q*. If neither *B* nor any ancestor of *B* is in $stack(Q)$, then *Q* does not contain any data covered by *B*. Otherwise, *Q* will follow the sibling link labeled "b" to find B' and remaining data from the node split off *B*.

4.2. Load balancing

CR-OLAP is executed on a cloud platform with $(m + 1)$ processors (m workers and one master). As discussed earlier, *CR-OLAP* uses the cloud's elasticity to increase m as the number of data items increases. We now discuss in more detail *CR-OLAP*'s mechanisms for worker allocation and load balancing in the cloud. The *insert* operations discussed above create independent subtrees for each

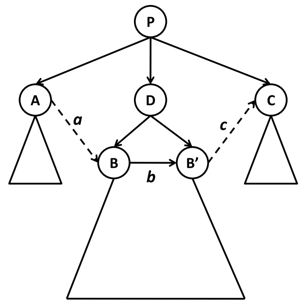


Fig. 8. Illustration for Theorem 1.

height h leaf of the *hat*. Since internal (directory) nodes have a high degree (typically 10–20), a relatively small height of the *hat* typically leads to thousands of height h leaves and associated subtrees $s_1, \ldots, s_n$. The master processor keeps track of the subtree locations and allocation of new workers, and it makes sure that a relatively high n/m ratio is maintained.

As indicated above, *CR-OLAP* shuffles these $n \gg m$ subtrees among the m workers. This ensures that threads of query operations are evenly distributed over the workers. Furthermore, *CR-OLAP* performs load balancing among the workers to ensure both, balanced workload and memory utilization. The master processor keeps track of the current sizes and the number of active threads for all subtrees. For each worker, its memory utilization and workload are the total number of threads of its subtrees and the total size of its subtrees, respectively.

If a worker w has a memory utilization above a certain threshold (e.g. 75% of its total memory), then the master processor determines the worker w' with the lowest memory utilization and checks whether it is possible to store an additional subtree from w while staying well below its memory threshold (e.g. 50% of its total memory). If that is not possible, a new worker w' is allocated within the cloud environment. Then, a subtree from w is compressed and sent from w to w' via message passing. As discussed earlier, PDCR trees are implemented in an array format and using only array indices as pointers. This enables fast compression and decompression of subtrees and greatly facilitates subtree migration between workers. Similarly, if a worker w has a workload utilization that is a certain percentage above the average workload of the m workers and is close to the maximum workload threshold for a single worker, then the master processor determines a worker w' with the lowest workload and well below its maximum workload threshold. If that is not possible, a new worker w' is allocated within the cloud environment. Then, the master processor initiates the migration of one or more subtrees from w (and possibly other workers) to w'.

Note that, in case workers are under-utilized due to shrinking workload, the above process can easily be reversed to decrease the number of workers. However, since the emphasis of our study is on growing system size for large scale OLAP, this was not implemented in our prototype system.

5. Experimental evaluation on Amazon EC2

5.1. Software

CR-OLAP was implemented in C++, using the g++ compiler, OpenMP for multi-threading, and ZeroMQ [34] for message passing between processors. Instead of the usual MPI message passing library we chose ZeroMQ because it better supports cloud elasticity and the addition of new processors during runtime. *CR-OLAP* has various tunable parameters. For our experiments we set the depth h of the *hat* to $h = 3$, the directory node capacity c to $c = 10$ for the *hat* and $c = 15$ for the subtrees, and the number k of threads per worker to $k = 16$.

5.2. Hardware/OS

CR-OLAP was executed on the Amazon EC2 cloud. For the master processor we used an Amazon EC2 m2.4 × large instance: "High-Memory Quadruple Extra Large" with 8 virtual cores (64-bit architecture, 3.25 ECUs per core) rated at 26 compute units and with 68.4 GiB memory. For the worker processors we used Amazon EC2 m3.2 × large instances: "M3 Double Extra Large" with 8 virtual cores (64-bit architecture, 3.25 ECUs per core) rated at 26 compute units and with 30 GiB memory. The OS image used was the standard Amazon CentOS (Linux) AMI.

5.3. Comparison baseline: STREAM-OLAP

As outlined in Section 2, there is no comparison system for *CR-OLAP* that provides scalable cloud based OLAP with full *real time* capability and support for dimension hierarchies. To establish a comparison baseline for *CR-OLAP*, we therefore designed and implemented a *STREAM-OLAP* method which partitions the database between multiple cloud processors based on one chosen dimension and uses parallel memory to cache streaming on the cloud processors to answer OLAP queries. More precisely, *STREAM-OLAP* builds a 1-dimensional index on one ordered dimension d_{stream} and partitions the data into approx. $100 \times m$ arrays. The arrays are randomly shuffled between the m workers. The master processor maintains the 1-dimensional index. Each array represents a segment of the d_{stream} dimension and is accessed via the 1-dimensional index. The arrays themselves are unsorted, and *insert* operations simply append the new item to the respective array. For *query* operations, the master determines via the 1-dimensional index which arrays are relevant. The workers then search those arrays via linear search, using memory to cache streaming.

The comparison between *CR-OLAP* (using PDCR trees) and *STREAM-OLAP* (using a 1-dimensional index and memory to cache streaming) is designed to examine the tradeoff between a sophisticated data structure which needs fewer data accesses but is less cache efficient and a brute force method which accesses much more data but optimizes cache performance.

5.4. Test data

For our experimental evaluation of *CR-OLAP* and *STREAM-OLAP* we used the standard TPC-DS "Decision Support" benchmark for OLAP systems [29]. We selected "Store Sales", the largest fact table available in TPC-DS. For the remainder, the database size N refers to the number of data items from "Store Sales" that were inserted into the database. Fig. 9 shows the fact table's 8 dimensions, and the respective 8 dimension hierarchies below each dimension. The first box for each dimension denotes the dimension name while the boxes below denote hierarchy levels from the highest to the lowest. Dimensions *Store*, *Item*, *Address*, and *Promotion* are unordered dimensions, while dimensions *Customer*, *Date*, *Household* and *Time* are ordered. TPC-DS provides a stream of *insert* and *query* operations on "Store Sales" which was used as input for *CR-OLAP* and *STREAM-OLAP*. For experiments where we were interested in the impact of query *coverage* (the portion of the database that needs to be aggregated for an OLAP query), we selected sub-sequences of TPC-DS queries with the chosen coverages.

5.5. Test results: impact of the number of workers (m) for fixed database size (N)

We tested how the time of the *insert* and *query* operations for *CR-OLAP* and *STREAM-OLAP* changes for fixed database size (N) as we increase the number of workers (m). Using a variable number of workers $1 \leq m \leq 8$, we first inserted 40 million items (with

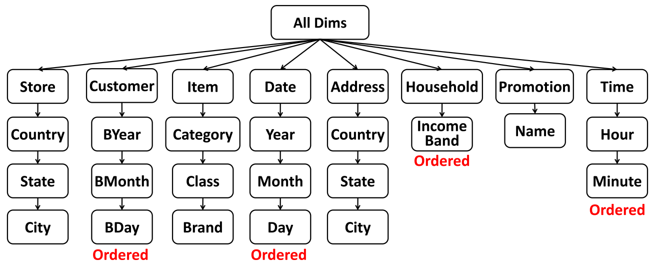


Fig. 9. The 8 dimensions of the TPC-DS benchmark for the fact table "Store Sales". Boxes below each dimension specify between 1 and 3 hierarchy levels for the respective dimension. Some dimensions are "ordered" and the remaining are not ordered.

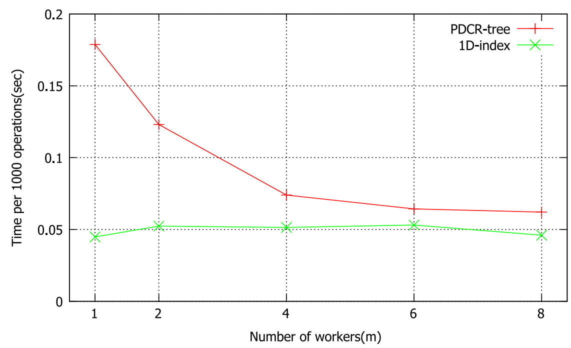


Fig. 10. Time for 1000 insertions as a function of the number of workers (N = 40 Mil, $d = 8$, $1 \leq m \leq 8$).

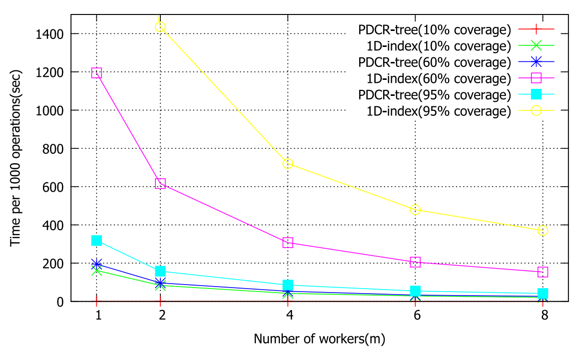


Fig. 11. Time for 1000 queries as a function of the number of workers (N = 40 Mil, $d = 8$, $1 \leq m \leq 8$).

$d = 8$ dimensions) from the TPC-DS benchmark into *CR-OLAP* and *STREAM-OLAP*, and then we executed 1000 (*insert* or *query*) operations on *CR-OLAP* and *STREAM-OLAP*. Since workers are virtual processors in the Amazon EC2 cloud, there is always some performance fluctuation because of the virtualization. We found that the total (or average) of 1000 *insert* or *query* operations is a sufficiently stable measure. The results of our experiments are shown in Figs. 10–12.

Fig. 10 shows the time for 1000 insertions in *CR-OLAP* (PDCR-tree) and *STREAM-OLAP* (1D-index) as a function of the number of workers (m). As expected, insertion times in *STREAM-OLAP* are lower than in *CR-OLAP* because *STREAM-OLAP* simply appends the new item in the respective array while *CR-OLAP* has to perform tree insertions with possible directory node splits and other overheads. However, *STREAM-OLAP* shows no speedup with increasing number of workers (because only one worker performs the array append operation) whereas *CR-OLAP* shows a significant speedup (because the distributed PDCR tree makes use of the multiple workers). It is important to note that insertion times are not visible to the users because they do not create any user response. What is important to the user are the response times for OLAP queries.

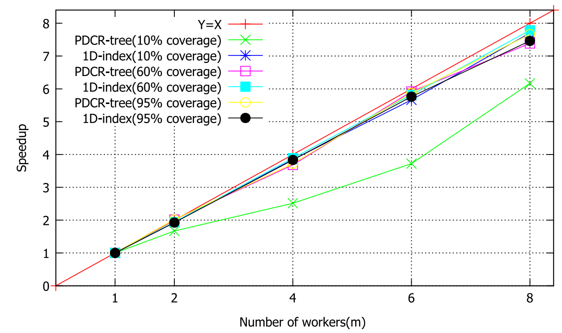


Fig. 12. Speedup for 1000 queries as a function of the number of workers (N = 40 Mil, $d = 8$, $1 \leq m \leq 8$).

Fig. 11 shows the time for 1000 OLAP queries in *CR-OLAP* and *STREAM-OLAP* as a function of the number of workers (m). Fig. 12 shows the speedup measured for the same data. We selected OLAP queries with 10%, 60% and 95% *query coverage*, which refers to the percentage of the entire range of values for *each* dimension that is covered by a given OLAP query. The selected OLAP queries therefore aggregate a small, medium and large portion of the database, resulting in very different workloads. We observe in Fig. 11 that *CR-OLAP* significantly outperforms *STREAM-OLAP* with respect to query time (in some cases 2000%). The difference in performance is particularly pronounced for queries with small or large coverages. For the former, the tree data structure shows close to logarithmic performance and for the latter, the tree can compose the result by adding the aggregate values stored at a few roots of large subtrees. The worst case scenario for *CR-OLAP* are queries with medium coverage around 60% where the tree performance is proportional to $N^{1-\frac{1}{d}}$. However, even in this worst case scenario, *CR-OLAP* outperforms *STREAM-OLAP* by between 300% and 500%. Fig. 12 indicates that both systems show a close to linear speedup with increasing number of workers, however for *CR-OLAP* that speedup occurs for much smaller absolute query times.

In a pay-as-you-go cloud environment, relating query response time to cloud computing *cost* may also be of interest. In that context, the close to linear speedup observed in Fig. 12 implies a fixed cost/performance ratio. For example, cutting the query response time in half would come at a price of doubling the system cost.

5.6. Test results: impact of growing system size (N and m combined)

In an elastic cloud environment, *CR-OLAP* and *STREAM-OLAP* increase the number of workers (m) as the database size (N) increases. In our scale up experiments, as we increase the number N of data items from 10 to 160 Mil, *CR-OLAP* and *STREAM-OLAP* increase the number m of workers from 1 to 16. That is, for each 10 Mil inserted items, *CR-OLAP* and *STREAM-OLAP* add one additional worker to the system. The impact on the performance of *insert* and *query* operations is shown in Figs. 13 and 14, respectively.

With growing system size, the time for *insert* operations in *CR-OLAP* (PDCR-tree) approaches the time for *STREAM-OLAP* (1D-index). More importantly however, the time for *query* operations in *CR-OLAP* again outperforms the time for *STREAM-OLAP* by a significant margin (in some cases more than 1000%), as shown in Fig. 14. Also, it is very interesting that for both systems, the query performance remains essentially unchanged with increasing database size and the number of workers. This is obvious for *STREAM-OLAP* where the size of arrays to be searched simply remains constant but it is an important observation for *CR-OLAP*. Fig. 14 indicates that the overhead incurred by *CR-OLAP*'s load balancing mechanism (which grows with increasing m) is balanced

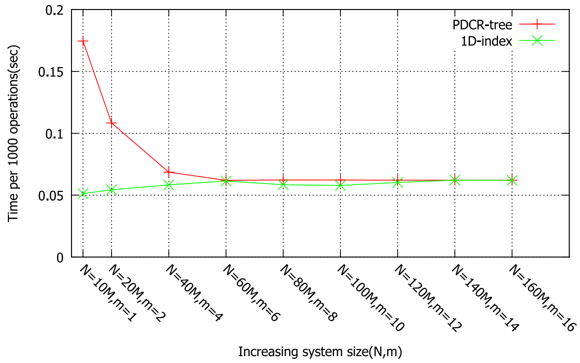


Fig. 13. Time for 1000 insertions as a function of system size: N and m combined (10 Mil $\leq N \leq$ 160 Mil, $d = 8$, $1 \leq m \leq 16$).

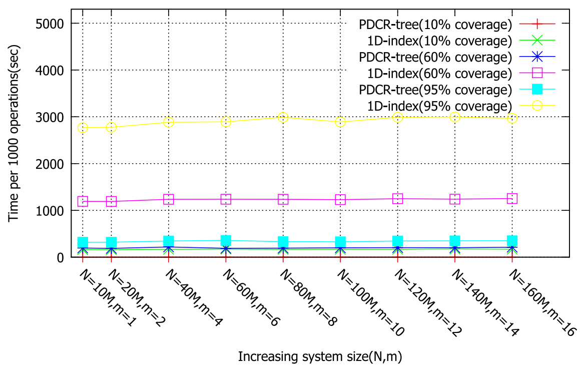


Fig. 14. Time for 1000 queries as a function of system size: N and m combined (10 Mil $\leq N \leq$ 160 Mil, $d = 8$, $1 \leq m \leq 16$).

out by the performance gained through more parallelism. *CR-OLAP* appears to scale up without affecting the performance of individual queries. It performed a 16-fold increase in the database size and the number of processors, including a 16-fold increase in the average amount of data aggregated by each OLAP query, without a noticeable performance impact for the user.

5.7. Test results: impact of multiple query streams

We evaluated the impact of the number of query streams on the performance of *CR-OLAP*. In all other experiments, we use one single stream of OLAP queries to measure performance. Here, we use multiple client processes, issuing multiple concurrent streams of OLAP queries that are fed into our *CR-OLAP* system. As shown in Fig. 15, the number of concurrent query streams (clients) has no impact on query performance.

5.8. Test results: impact of the number of dimensions

It is well known that tree based search methods can become problematic when the number of dimensions in the database increases. In Figs. 16 and 17 we show the impact of increasing d on the performance of *insert* and *query* operations in *CR-OLAP* (PDCR-tree) and *STREAM-OLAP* (1D-index) for fixed database size $N = 40$ million and $m = 8$ workers.

Fig. 16 shows some increase in *insert* time for *CR-OLAP* because the PDCR tree insertion inherits from the PDC tree a directory node split operation with an optimization phase that is highly sensitive to the number of dimensions. However, the result of the tree optimization is improved query performance in higher dimensions. As shown in Fig. 17, the more important time for OLAP *query* operations grows only slowly as the number of dimensions increases. This is obvious for the array search in *STREAM-OLAP* but for the tree search in *CR-OLAP* this is an important observation.

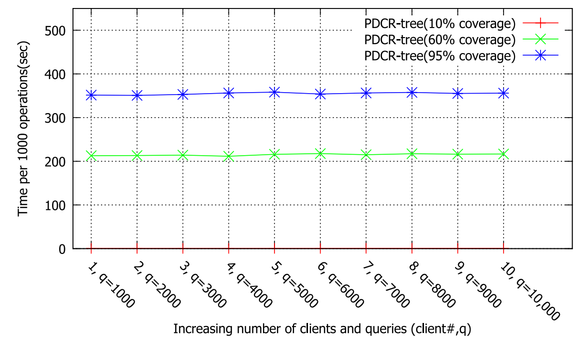


Fig. 15. Time for 1000 OLAP queries as a function of the number of query streams. X-axis first parameter: the number of query streams (clients). X-axis second parameter: the total number of queries issued (1000 queries per query stream). Y-axis: Average time per 1000 queries in seconds (N = 160 Mil, $d = 8$, $m = 16$).

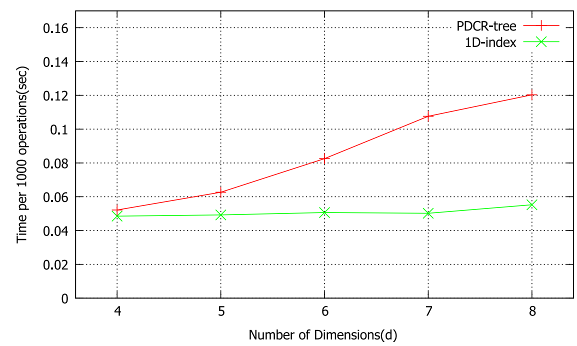


Fig. 16. Time for 1000 insertions as a function of the number of dimensions (N = 40 Mil, $4 \leq d \leq 8$, $m = 8$).

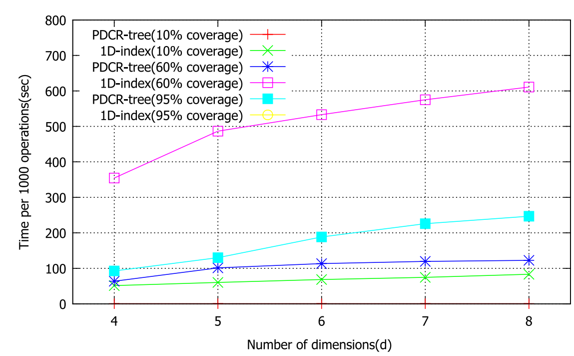


Fig. 17. Time for 1000 queries as a function of the number of dimensions. The values for "1D-index 95% coverage" are 828.6, 1166.4, 1238.5, 1419.7 and 1457.8, respectively (N = 40 Mil, $4 \leq d \leq 8$, $m = 8$).

5.9. Test results: impact of query coverages

Figs. 18–21 show the impact of query coverage on query performance in *CR-OLAP* (PDCR-tree) and *STREAM-OLAP* (1D-index).

For fixed database size N = 40 Mil, number of workers $m = 8$, and number of dimensions $d = 8$, we vary the query coverage and observe the query times. In addition we observe the impact of a "*" in one of the query dimensions. Figs. 18 and 19 show that the "*" values do not have a significant impact for *CR-OLAP*. As discussed earlier, *CR-OLAP* is most efficient for small and very large query coverage, with maximum query time somewhere in the mid range. (In this case, the maximum point is shifted away from the typical 60% because of the "*" values.) Figs. 20 and 21 show the performance of *STREAM-OLAP* as compared to *CR-OLAP* (ratio of query times). It shows that *CR-OLAP* consistently outperforms *STREAM-OLAP* by a factor between 5 and 20.

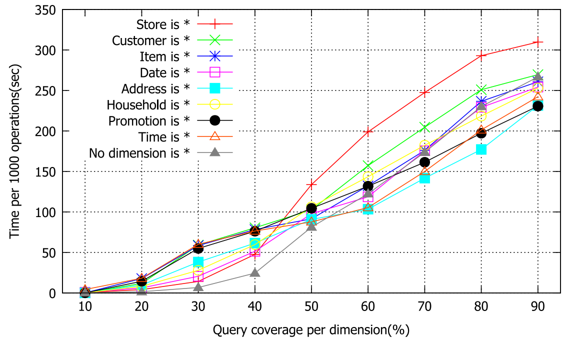


Fig. 18. Time for 1000 queries (PDCR tree) as a function of query coverages: 10%–90%. Impact of value "*" for different dimensions ($N = 40$ Mil, $m = 8$, $d = 8$).

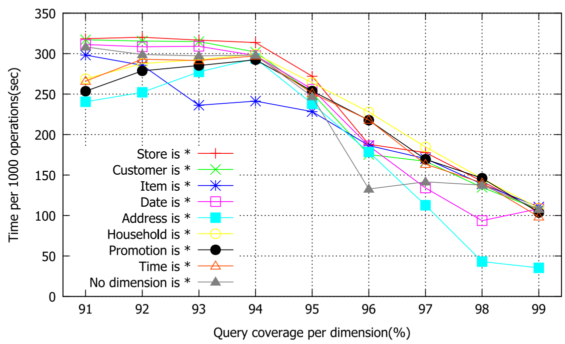


Fig. 19. Time for 1000 queries (PDCR tree) as a function of query coverages: 91%–99%. Impact of value "*" for different dimensions ($N = 40$ Mil, $m = 8$, $d = 8$).

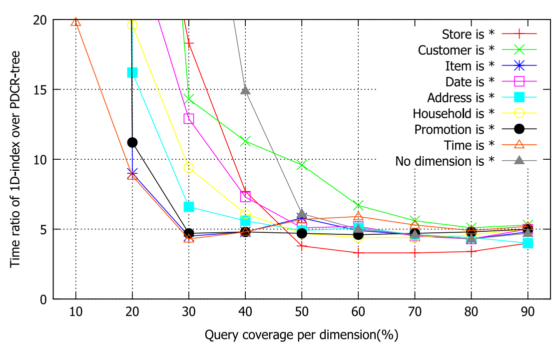


Fig. 20. Time comparison for 1000 queries (ratio: 1D-index/PDCR tree) for query coverages 10%–90%. Impact of value "*" for different dimensions ($N = 40$ Mil, $m = 8$, $d = 8$).

5.10. Test results: query time comparison for selected query patterns at different hierarchy levels

Fig. 22 shows a query time comparison between *CR-OLAP* (PDCR-tree) and *STREAM-OLAP* (1D-index) for selected query patterns. For fixed database size $N = 40$ Mil, number of workers $m = 8$ and $d = 8$ dimensions, we test for dimension *Date* the impact of value "*" for different hierarchy levels. *CR-OLAP* is designed for OLAP queries such as "*total sales in the stores located in California and New York during February–May of all years*" which act at different levels of multiple dimension hierarchies. For this test, we created 7 combinations of "*" and set values for hierarchy levels *Year*, *Month*, and *Day*: *-*-*, year-*-*, year-month-*, year-month-day, *-month-*, *-month-day, and *-*-day. We then selected for each combination queries with coverages 10%, 60%, and 95%. The test results are summarized in Fig. 22. The main observation is that *CR-OLAP* consistently outperforms *STREAM-OLAP* even for complex and very broad queries that one would expect could be easily solved through data streaming than through tree search.

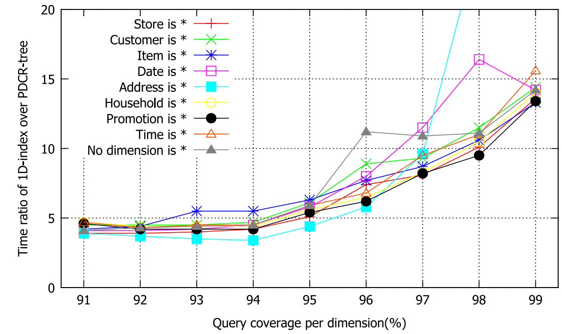


Fig. 21. Time comparison for 1000 queries (ratio: 1D-index/PDCR tree) for query coverages 91%–99%. Impact of value "*" for different dimensions ($N = 40$ Mil, $m = 8$, $d = 8$).

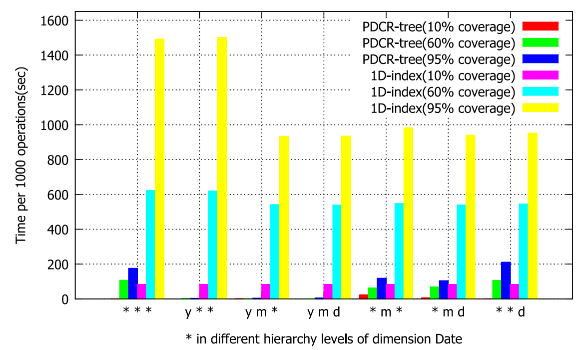


Fig. 22. Query time comparison for selected query patterns for dimension *Date*. Impact of value "*" for different hierarchy levels of dimension. *Date* ($N = 40$ Mil, $m = 8$, $d = 8$).

6. Conclusion

We introduced *CR-OLAP*, a Cloud based Real-time *OLAP* system based on a *distributed PDCR tree*, a new parallel and distributed index structure for OLAP, and evaluated *CR-OLAP* on the Amazon EC2 cloud for a multitude of scenarios. The tests demonstrate that *CR-OLAP* scales well with increasing database size and increasing number of cloud processors. In our experiments, *CR-OLAP* performed a 16-fold increase in database size and the number of processors, including a 16-fold increase in the average amount of data aggregated by each OLAP query, without noticeable performance impact for the user.

Future work: the next phase of our research collaboration with IBM includes the study of how to add data replication and fault tolerance to our system. We also plan to study how the master processor could be replaced by a decentralized system.

Acknowledgments

The authors would like to acknowledge financial support from the IBM Centre for Advanced Studies Canada and the Natural Science and Engineering Research Council of Canada. We thank the research staff at the IBM Centre for Advanced Studies Canada, and in particular Stephan Jou, for their support and helpful discussions.

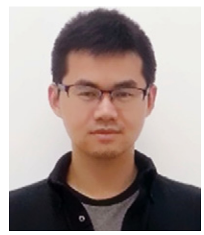
References

[1] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, A. Rasin, HadoopDB: an architectural hybrid of mapReduce and DBMS technologies for analytical workloads, Proc. VLDB Endow. 2 (1) (2009) 922–933.

[2] A. Asiki, D. Tsoumakos, N. Koziris, Distributing and searching concept hierarchies: an adaptive DHT-based system, Cluster Comput. 13 (3) (2010) 257–276.

[3] Bigquery. URL: http://developers.google.com/bigquery/.
[4] R. Bruckner, B. List, J. Schiefer, Striving towards near real-time data integration for data warehouses, in: DaWaK LNCS, vol. 2454, 2002, pp. 173–182.
[5] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, R.E. Gruber, Bigtable: a distributed storage system for structured data, ACM Trans. Comput. Syst. 26 (2) (2008) 4:1–4:26.
[6] Y. Chen, F. Dehne, T. Eavis, A. Rau-Chaplin, PnP: sequential, external memory, and parallel iceberg cube computation, Distrib. Parallel Databases 23 (2) (2008) 99–126.
[7] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.
[8] F. Dehne, T. Eavis, S. Hambrusch, Parallelizing the data cube, Distrib. Parallel Databases 11 (2002) 181–201.
[9] F. Dehne, H. Zaboli, Parallel real-time olap on multi-core processors, in: Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, ccgrid 2012, 2012, pp. 588–594.
[10] K. Doka, D. Tsoumakos, N. Koziris, Brown Dwarf: a fully-distributed, fault-tolerant data warehousing system, J. Parallel Distrib. Comput. 71 (11) (2011) 1434–1446.
[11] K. Doka, D. Tsoumakos, N. Koziris, Online querying of d-dimensional hierarchies, J. Parallel Distrib. Comput. 71 (3) (2011) 424–437.
[12] M. Ester, J. Kohlhammer, H.-P. Kriegel, The DC-tree: a fully dynamic index structure for data warehouses, in: Proceedings of the 16th International Conference on Data Engineering, ICDE, 2000, pp. 379–388.
[13] M. Ester, J. Kohlhammer, H.-P. Kriegel, The DC-tree: a fully dynamic index structure for data warehouses, in: 16th International Conference on Data Engineering, ICDE, 2000, pp. 379–388.
[14] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, J. Dees, The SAP HANA database—an architecture overview, IEEE Data Eng. Bull. 35 (1) (2012) 28–33.
[15] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, H. Pirahesh, Data cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-totals, Data Min. Knowl. Discov. 1 (1997) 29–53.
[16] Z. Guo-Liang, C. Hong, L. Cui-Ping, W. Shan, Z. Tao, Parallel data cube computation on graphic processing units, Chinese J. Comput. 33 (10) (2010) 1788–1798.
[17] Hadoop. URL: http://hadoop.apache.org/.
[18] J. Han, M. Kamber, Data Mining: Concepts and Techniques, Morgan Kaufmann Publishers, 2000.
[19] D. Jin, T. Tsuji, K. Higuchi, An incremental maintenance scheme of data cubes and its evaluation, in: DASFAA LNCS, vol. 4947, 2008, pp. 36–48.
[20] M.C. Kurt, G. Agrawal, A fault-tolerant environment for large-scale query processing, in: High Performance Computing, HiPC, 2012 19th International Conference on, 2012, pp. 1–10.
[21] S. Melnik, A. Gubarev, J.J. Long, G. Romer, S. Shivakumar, M. Tolton, T. Vassilakis, Dremel: interactive analysis of web-scale datasets, Proc. VLDB Endow. 3 (1–2) (2010) 330–339.
[22] R.T. Ng, A. Wagner, Y. Yin, Iceberg-cube computation with PC clusters, ACM SIGMOD 30 (2) (2001) 25–36.
[23] H. Plattner, A. Zeier, In-Memeory Data Management, Springer Verlag, 2011.
[24] R. Santos, J. Bernardino, Real-time data warehouse loading methodology, in: IDEAS, 2008, pp. 49–58.
[25] R.J. Santos, J. Bernardino, Optimizing data warehouse loading procedures for enabling useful-time data warehousing, in: IDEAS, 2009, pp. 292–299.
[26] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, Y. Kotidis, Dwarf: shrinking the PetaCube, in: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, 2002, pp. 464–475.
[27] The OLAP Report. http://www.olapreport.com.
[28] A. Thusoo, J.S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, R. Murthy, Hive: a warehousing solution over a map-reduce framework, Proc. VLDB Endow. 2 (2) (2009) 1626–1629.
[29] Transaction processing performance council, TPC-DS (decision support) benchmark. URL: http://www.tpc.org.
[30] Twitter storm. URL: http://storm-project.net/.
[31] J. Wang, S. Wu, H. Gao, J. Li, B.C. Ooi, Indexing multi-dimensional data in a cloud system, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, 2010, pp. 591–602.
[32] S. Wu, D. Jiang, B.C. Ooi, K.-L. Wu, Efficient B-tree based indexing for cloud data processing, Proc. VLDB Endow. 3 (1–2) (2010) 1207–1218.
[33] J. You, J. Xi, P. Zhang, H. Chen, A parallel algorithm for closed cube computation, in: IEEE/ACIS International Conference on Computer and Information Science, 2008, pp. 95–99.
[34] Zeromq socket library as a concurrency framework. URL: http://www.zeromq.org/.
[35] X. Zhang, J. Ai, Z. Wang, J. Lu, X. Meng, An efficient multi-dimensional index for cloud data management, in: Proceedings of the First International Workshop on Cloud Data Management, 2009, pp. 17–24.

F. Dehne received an M.C.S. (Dipl. Inform.) from RWTH Aachen University, Germany and a Ph.D. from the University of Würzburg, Germany. He is currently Chancellor's Professor of Computer Science at Carleton University in Ottawa, Canada. His research interests include the design and implementation of efficient parallel algorithms, the interrelationship between the theoretical analysis of parallel algorithms and the performance observed on current parallel architectures, and the use of efficient parallel algorithms for large-scale data analytics and computational biology/bioinformatics. Dr. Dehne is serving or has served on the Editorial Boards of IEEE Transaction on Computers, Information Processing Letters, Journal of Bioinformatics Research and Applications, and International Journal of Data Warehousing and Mining. He is a Fellow of the IBM Centre for Advanced Studies Canada, a member and the former vice-chair of the IEEE Technical Committee on Parallel Processing, a member of the ACM Symposium on Parallel Algorithms & Architectures Steering Committee, and a co-founder of the Algorithms and Data Structures Symposium (WADS).

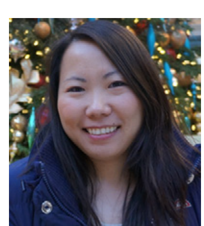


Q. Kong is a Master student in the Faculty of Computer Science at Dalhousie University. His main area is real-time parallel in-memory analytics on the cloud. He received his B.Sc. in Software Engineering from Harbin Institute of Technology, China.

A. Rau-Chaplin received a M.C.S. and Ph.D. from Carleton University in Ottawa in 1990 and 1993, respectively. From 1993 to 1994 he was a Postdoctoral Fellow at DIMACS—a National Science Foundation center run by Princeton University, Rutgers, and AT&T Bell labs. In 1994 he joined the Technical University of Nova Scotia, in 1998 be became an Associate Professor in the Faculty of Computer Science at Dalhousie University where he is currently a Professor. He is interested in the application of parallel computing to a wide range of problem domains including data mining, image processing, solid modeling, geographic information systems (GIS), computational geometry/CAD, and AI. His parallel computing research program tends to be almost evenly divided between theoretical studies, where the focus is on developing asymptotically optimal or near optimal algorithms in a given complexity model, and implementation work which is used to guide and "ground" the theoretical work.

Much of his recent research has concentrated on coarse grained and BSP style parallel processing. Together with Frank Dehne and Andreas Fabri he introduced the CGM model which aims at delivering practical parallel algorithms whose performance analysis matches very closely with the actually observed running times on commercial parallel machines. This model has been very successful in practice. Since 1993, an increasing number of researchers and graduate students have used his CGM model for parallel algorithm design in various application areas, and Algorithmica dedicated a special issue to the CGM model.

H. Zaboli received his M.Sc. degree from Amirkabir University of Technology (Tehran Polytechnic) in 2007. He has graduated from Carleton University with a Ph.D. degree in 2014. From 2010 to 2013, he had been a Ph.D. fellow student at IBM Centre for Advanced Studies (CAS) Canada. His current research interest is in the application of parallel and distributed methods for real-time OLAP on multi-core and cloud platforms.



R. Zhou received a master's degree in Computer Science from Carleton University, Ottawa, Canada, in 2013. Her primary research interests include data modeling, data warehousing, and business intelligence. Currently, she works as a Senior Analyst for Enterprise Data Warehouse group at the Department of Citizenship and Immigration Canada.