A Distributed Tree Data Structure For Real-Time OLAP On Cloud Architectures

F. Dehne¹, Q. Kong², A. Rau-Chaplin², H. Zaboli¹, R. Zhou¹
¹ School of Computer Science, Carleton University, Ottawa, Canada
² Faculty of Computer Science, Dalhousie University, Halifax, Canada
E-mail: frank@dehne.net, qkong@cs.dal.ca, arc@cs.dal.ca
zaboli@graduate.org, xiaoyunzhou@cmail.carleton.ca

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 Cloud based Real-time OLAP system based on a new distributed index structure for OLAP, the distributed PDCR tree, that utilizes a cloud infrastructure consisting of \((m + 1)\) multi-core processors. With increasing database size, CR-OLAP dynamically increases \(m\) 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, on an Amazon EC2 cloud instance with eight processors, for a TPC-DS OLAP query stream on a data warehouse with 80 million tuples where every OLAP query aggregates more than 50% of the database, CR-OLAP achieved a query latency of 0.3 seconds which can be considered a real time response.

I. 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, real-time OLAP.

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. 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 [1], [2]. 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, 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 [3] 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. [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 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. [9]). 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. [9], [10], [11], [12]). However, these approaches are not fully real-time. A major obstacle are significant performance issues with large scale data warehouses.

B. 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 [13] 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 cloud environment.

We introduce CR-OLAP, a Cloud based Real-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. [14] and our previous PDC tree [13], exploits knowledge
about the structure of individual dimension hierarchies both for
compacted 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 repre-
sented as an aggregate range query that specifies for each di-

tension either a single value or range of values at any level of
the respective dimension hierarchy, or a symbol “$\ast$” 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 is kept in
the processors’ main memories [15]. 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 do usually not support data deletion.) 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 [13], adapted to the
cloud environment. 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. 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, on an Amazon EC2 cloud instance with eight
workers and a stream of OLAP queries on a data warehouse
with 80 million tuples, where each query has more than 50%
coverage, CR-OLAP achieved a query latency of 0.3 seconds,
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 num-
ber of dimensions, even simple range search (no dimension
hierarchies, no aggregation) can degenerate to linear search
(e.g. [16]). In our experiments, we observed that increasing
number of dimensions does not significantly impact the per-
formance 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 imple-
mented 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 II we review related work. In Section III we introduce
the PDCR tree data structure and in Section IV we present our
CR-OLAP system for real-time OLAP on cloud architectures.
Section V shows the results of an experimental evaluation of
CR-OLAP on the Amazon EC2 cloud, and Section VI
concludes the paper.

II. RELATED WORK

In addition to the related work discussed in the introdus-
tion, 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 [18]. Related projects most similar to our work are Hive[19] and HadoopDB[20]. However, these systems are not designed for real-time (OLTP style) operation. Instead, they use batch processing similar to [9], [10], [11], [12]. The situation is similar for BigTable[21], BigQuery[22], and Dremel[23]. In fact, Dremel[23] 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 [24] 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. For peer-to-peer networks, related work includes distributed methods for querying concept hierarchies such as [25], [26], [27], [28]. However, none of these methods provides real-time OLAP functionality. There are various publications on distributed B-trees for cloud platforms such as [29]. However, these methods only support 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 [30], [31], [32]. However, these methods do not support dimension hierarchies which are essential for OLAP queries.

III. PDCR TREES

Consider a data warehouse with the fact table F and a set of d dimensions \{D_1, D_2, ..., 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 \(H_i = \{H_{i1}, H_{i2}, ..., H_{il}\}\) where a parent logically summarizes and includes its children. Figure 1 shows the dimensions and hierarchy levels of each dimension for a 4-dimensional data warehouse.

![Diagram of a 4-dimensional data warehouse with 3 hierarchy levels for each dimension. The first box for each dimension denotes the name of the dimension.](image)

The sequential DC tree introduced by Kriegel et al. [14] and our previous PDC tree [13] store so-called “minimum describing set” (MDS) entries at each internal tree node to guide the query process; see [14] for details. The MDS concept was developed in [14] 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 Figure 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”.

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 IV requires that a subtree 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. In the following we outline some details of the encoding used for the 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 Figure 1, an ID may represent an entity at the 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 Figure 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. Figure 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. [14] and our previous PDC tree [13] store so-called “minimum describing set” (MDS) entries at each internal tree node to guide the query process; see [14] for details. The MDS concept was developed in [14] 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 Figure 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”.

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 IV requires that a subtree 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. In the following we outline some details of the encoding used for the 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 Figure 1, an ID may represent an entity at the 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 Figure 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. Figure 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. [14] and our previous PDC tree [13] store so-called “minimum describing set” (MDS) entries at each internal tree node to guide the query process; see [14] for details. The MDS concept was developed in [14] 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 Figure 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”.
IV. 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 \textit{master}, and the remaining \( m \) processors are called \textit{workers}. The master receives from the users the input stream of OLAP \textit{insert} and \textit{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 PDCR 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 \textit{hat} and we refer to the remaining trees rooted at the leaves of the \textit{hat} as the \textit{subtrees} \( s_1, \ldots, s_n \). Level \( h \) is referred to as the \textit{cut level}. The \textit{hat} will be stored at the master and the \textit{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 \textit{hat} (PDCR tree). Note that, DC trees [14], PDC trees [13] and PDCR trees are leaf oriented. All data is stored in leaves called \textit{data nodes}. Internal nodes are called \textit{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 \textit{hat} (PDCR tree) grows. When directory nodes of the \textit{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 Figure 5.

For a typical database size, the \textit{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 \textit{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 Figure 5, both \textit{insert} and \textit{query} operations are executed concurrently.

Concurrent insert and query operations

Each \textit{query} operation in the input stream is handed to the master which traverses the \textit{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 \textit{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 Algorithm 3 and Algorithm 4 in the Appendix.

For each \textit{insert} operation in the input stream, the master will search the \textit{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 Algorithm 1 and Algorithm 2 in the Appendix. Figures 6 and 7 illustrate how \textit{new} workers and \textit{new} subtrees are added as more data items get inserted. Figures 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 [13] we presented a method to ensure correct query processing for a single PDC tree on a multi-core processor, where multiple insert and query operations are processed concurrently. The PDC tree maintains for each data or 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 [13] for more details. The PDCR tree presented in this paper inherits this mechanism for each of its subtrees. In fact, the above mentioned sibling links are shown as horizontal links in Figures 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 [13]. 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 show that correct real time processing of mixed insert and query operations is still maintained.

**Theorem 1.** Consider the situation depicted in Figure 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 labelled “a” and “c” are not required for correct real time query processing (as defined above).

**Proof outline.** A full case analysis is lengthy and omitted due to space limitations. Here we only present the most interesting and critical case. 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 are in $stack(Q)$ then Q does not contain any data covered by B. Otherwise, Q will follow the sibling link labelled “b” to find B’ and remaining data from the node split of B.

**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 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 >> 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 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 if 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 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 $w'$.

V. EXPERIMENTAL EVALUATION ON AMAZON EC2

Software

CR-OLAP was implemented in C++, using the g++ compiler, OpenMP for multi-threading, and ZeroMQ [33] 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 $k = 16$.

Hardware/OS

CR-OLAP was executed on the Amazon EC2 cloud. For the master processor we used an Amazon EC2 m2.4xlarge instance: “High-Memory Quadraple 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 m3.2xlarge 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.

Comparison baseline: STREAM-OLAP

As outlined in Section II, there is no comparison system for CR-OLAP that provides 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 unsortd, 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.

Test data

For our experimental evaluation of CR-OLAP and STREAM-OLAP we used the standard TPC-DS “Decision Support” benchmark for OLAP systems [34]. We selected “Store Sales”, the largest fact table available in TPC-DS. Figure 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 highest to 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.

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

We tested how the time of 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 $d=8$ dimensions) from the TPC-DS benchmark into CR-OLAP and STREAM-OLAP, and then we executed 1,000 (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 1,000 insert or query operations is a sufficiently stable measure. The results of our experiments are shown in Figures 10, 11, and 12.

Figure 10 shows the time for 1,000 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. Figure 11 shows the time for 1,000 OLAP queries in CR-OLAP and STREAM-OLAP as a function of the number of workers ($m$). Figure 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 Figure 11 that CR-OLAP significantly outperforms STREAM-OLAP with respect to query time. 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 $N^{1-\frac{d}{2}}$. However, even in this worst case scenario, CR-OLAP outperforms STREAM-OLAP. Figure 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.

Test results: impact of growing system size ($N$ & $m$ combined)

In an elastic cloud environment, CR-OLAP and STREAM-OLAP increase the number of workers ($m$) as the database size ($N$) increases. The impact on the performance of insert and query operations is shown in Figures 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, as shown in Figure 14. Also, it is very interesting that the for both systems, the query performance remains essentially unchanged with increasing database size and 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. Figure 14 indicates that the overhead incurred by CR-OLAP’s load balancing mechanism (which grows with increasing $m$) is balanced out by the performance gained through more parallelism. CR-OLAP appears to scale up without affecting the performance of individual queries.

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 Figures 15 and 16 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. Figure 15 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 Figure 16, 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.

Test results: impact of query coverages

Figures 17, 18, 19, and 20 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. Figures 17 and 18 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.) Figures 19, and 20 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.

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

Figure 21 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 Figure 21. The main observation is that CR-OLAP consistently outperforms
STREAM-OLAP even for complex and very broad queries that one would expect could be easier solved through data streaming than through tree search.

VI. 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. CR-OLAP has the potential to enable OLAP systems with real-time OLAP query processing for large databases.

ACKNOWLEDGMENT

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

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

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

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

Fig. 13. Time for 1000 insertions as a function of system size: \(N \& m\) combined. \(10Ml \leq N \leq 80Ml, d = 8, 1 \leq m \leq 8\)

Fig. 14. Time for 1000 queries as a function of system size: \(N \& m\) combined. \(10Ml \leq N \leq 80Ml, d = 8, 1 \leq m \leq 8\)

Fig. 15. Time for 1000 insertions as a function of the number of dimensions. \(N = 40Ml, 4 \leq d \leq 8, m = 8\)

Fig. 16. 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. \(N = 40Ml, 4 \leq d \leq 8, m = 8\)

Fig. 17. Time for 1000 queries (PDCR tree) as a function of query coverages: 10% – 90%. Impact of value “*” for different dimensions. \(N = 40Ml, m = 8, d = 8\)

Fig. 18. Time for 1000 queries (PDCR tree) as a function of query coverages: 91% – 99%. Impact of value “**” for different dimensions. \(N = 40Ml, m = 8, d = 8\)

Fig. 19. Time comparison for 1000 queries (Ratio: 1D-index / PDCR tree) for query coverages 10% – 90%. Impact of value “*” for different dimensions. \(N = 40Ml, m = 8, d = 8\)

Fig. 20. Time comparison for 1000 queries (Ratio: 1D-index / PDCR tree) for query coverages 91% – 99%. Impact of value “**” for different dimensions. \(N = 40Ml, m = 8, d = 8\)

Fig. 21. Query time comparison for selected query patterns for dimension Date. Impact of value “**” for different hierarchy levels of dimension Date. \(N = 40Ml, m = 8, d = 8\)
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
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.

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.
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.
  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
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.

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.
  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
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.