SCALABLE REAL-TIME OLAP SYSTEMS FOR THE CLOUD

by

Quan Kong

Submitted in partial fulfillment of the
requirements for the degree of
Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
August 2014

Table of Contents

List of Tables . . . . . **v**

List of Figures . . . . . **vi**

Abstract . . . . . **x**

List of Abbreviations Used . . . . . **xi**

Acknowledgements . . . . . **xii**

Chapter 1 Introduction **1**

1.1 Contributions 6

1.2 Structure of the thesis 11

Chapter 2 Background: Data Management on the Cloud **12**

2.1 Data Store Categorization 12
2.1.1 OLAP Stores 13
2.1.2 Key-value Stores 16
2.1.3 Document Stores 17
2.1.4 Extensible Record Stores 18

2.2 OLAP 19
2.2.1 Dimension Hierarchy 21

2.3 Data Structures for OLAP 22
2.3.1 RTree 23
2.3.2 DC Tree 25
2.3.3 PDC Tree 28

Chapter 3 Background: Cloud Computing **31**

3.1 Cloud Computing 31
3.1.1 Characteristics 33
3.1.2 Service models 34

3.2 Communication in the Cloud 37
3.3 Serialization in the Cloud 39
3.4 Coordination in the Cloud 40
3.5 Amazon Web Services 41
3.5.1 Amazon EC2 42
3.6 ZeroMQ 43
3.7 Zookeeper 46
Chapter 4 CR-OLAP 50
4.1 Introduction 50
4.2 PDCR Trees 53
4.3 CR-OLAP: Cloud based Real-time OLAP 56
4.3.1 Concurrent insert and query operations 58
4.3.2 Load balancing 65
4.4 Experimental Evaluation On Amazon EC2 66
4.4.1 Software 66
4.4.2 Hardware/OS 67
4.4.3 Comparison baseline: STREAM-OLAP 67
4.4.4 Test data 68
4.4.5 Test results: impact of the number of workers (m) for fixed database size (N) 69
4.4.6 Test results: impact of growing system size (N & m combined) 72
4.4.7 Test results: impact of multiple query streams 74
4.4.8 Test results: impact of the number of dimensions 74
4.4.9 Test results: impact of query coverages 76
4.4.10 Test results: query time comparison for selected query patterns at different hierarchy levels 78
4.5 Conclusion 79

Chapter 5 vOLAP **81**

5.1 Introduction 81

5.2 *VelocityOLAP* 84
5.2.1 Architecture Overview 84
5.2.2 System Image 86
5.2.3 Data Representation 88

5.3 Algorithms 89
5.3.1 OLAP Insertion Algorithms 89
5.3.2 OLAP Query Algorithms 92

5.4 Load Balancing 95
5.4.1 Migration Protocol 96
5.4.2 Splitting Protocol 97

5.5 Optimizer 98
5.5.1 Optimization Algorithm 99

5.6 Experimental Evaluation 100
5.6.1 High-Velocity Data Ingestion 102
5.6.2 Real-Time Load Balancing 104
5.6.3 Query Performance for TPC-DS Data 105
5.6.4 System Scale-Up 107
5.6.5 Large Scale Experiment 109

Chapter 6 Conclusion **110**

Bibliography **112**

List of Tables

Table 5.1 System Parameters 85

List of Figures

Figure 2.1 Hierarch Schema and Concept Hierarchy for dimension Customer [39] 22

Figure 2.2 A 4-dimensional data warehouse with 3 hierarchy levels for each dimension. The first box for each dimension denotes the name of the dimension. 22

Figure 2.3 Containment and overlapping relationships among MBRs [48] 23

Figure 2.4 A 2-dimensional R-tree example 24

Figure 2.5 The dimensional hierarchies of a sample DC-tree 26

Figure 2.6 A simple DC-tree 27

Figure 3.1 Three cloud computing service models [2] 35

Figure 3.2 Request Reply pattern [3] 43

Figure 3.3 Pub-sub pattern [3] 44

Figure 3.4 Pipeline pattern [3] 45

Figure 3.5 Zookeeper's hierarchical namespace [10] 46

Figure 3.6 Zookeeper's architecture [10] 47

Figure 3.7 Zookeeper components [10] 47

Figure 4.1 A 4-dimensional data warehouse with 3 hierarchy levels for each dimension. The first box for each dimension denotes the name of the dimension. 54

Figure 4.2 Illustration of the compact bit representation of IDs. 56

Figure 4.3 Example of relationships between different hierarchy levels of a given dimension. 56

Figure 4.4 Example of a PDCR tree with 2 dimensions (Store and Date). 57

Figure 4.5 Illustration of a distributed PDCR tree. 58

Figure 4.6 Insertions triggering creation of *new* workers and subtrees. Part 1. (a) Current *hat* configuration. (b) Insertions create overflow at node A and horizontal split. 59

Figure 4.7 Insertions triggering creation of *new* workers and subtrees. Part 2. (a) Same as Figure 4.6b with critical subtrees highlighted. (b) Insertions create overflow at node C and vertical split, triggering the creation of two subtrees in two different workers. 64

Figure 4.8 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. 68

Figure 4.9 Time for 1000 insertions as a function of the number of workers. ($N = 40Mil$, $d = 8$, $1 \leq m \leq 8$) 69

Figure 4.10 Time for 1000 queries as a function of the number of workers. ($N = 40Mil$, $d = 8$, $1 \leq m \leq 8$) 70

Figure 4.11 Speedup for 1000 queries as a function of the number of workers. ($N = 40Mil$, $d = 8$, $1 \leq m \leq 8$) 70

Figure 4.12 Time for 1000 insertions as a function of system size: N & m combined. ($10Mil \leq N \leq 160Mil$, $d = 8$, $1 \leq m \leq 16$) 72

Figure 4.13 Time for 1000 queries as a function of system size: N & m combined. ($10Mil \leq N \leq 160Mil$, $d = 8$, $1 \leq m \leq 16$) 73

Figure 4.14 Time for 1000 OLAP queries as a function of of the number of query streams. X-axis first parameter: number of query streams (clients). X-axis second parameter: total number of queries issued (1,000 queries per query stream). Y-axis: Average time per 1,000 queries in seconds. ($N = 160Mil$, $d = 8$, $m = 16$) . . 74

Figure 4.15 Time for 1000 insertions as a function of the number of dimensions. ($N = 40Mil$, $4 \leq d \leq 8$, $m = 8$) 75

Figure 4.16 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. ($N = 40Mil$, $4 \leq d \leq 8$, $m = 8$) 76

Figure 4.17 Time for 1000 queries (PDCR tree) as a function of query coverages: $10\% - 90\%$. Impact of value "*" for different dimensions. ($N = 40Mil$, $m = 8$, $d = 8$) 77

Figure 4.18 Time for 1000 queries (PDCR tree) as a function of query coverages: $91\% - 99\%$. Impact of value "*" for different dimensions. ($N = 40Mil$, $m = 8$, $d = 8$) 77

Figure 4.19 Time comparison for 1000 queries (Ratio: 1D-index / PDCR tree) for query coverages $10\% - 90\%$. Impact of value "*" for different dimensions. ($N = 40Mil$, $m = 8$, $d = 8$) 78

Figure 4.20 Time comparison for 1000 queries (Ratio: 1D-index / PDCR tree) for query coverages $91\% - 99\%$. Impact of value "*" for different dimensions. ($N = 40Mil$, $m = 8$, $d = 8$) 79

Figure 4.21 Query time comparison for selected query patterns for dimension *Date*. Impact of value "*" for different hierarchy levels of dimension *Date*. ($N = 40Mil$, $m = 8$, $d = 8$). 80

Figure 5.1 System Overview. 85

Figure 5.2 Multi-Threaded Message Handling. 86

Figure 5.3 Insertion on server, 3 possible cases. 90

Figure 5.4 Box expand on server. 91

Figure 5.5 Range query. 92

Figure 5.6 Migration process. 97

Figure 5.7 Split process. 98

Figure 5.8 Data ingestion performance as database size N increases. Zipf data, $p = 16$. 103

Figure 5.9 Average and maximum data size per worker as database size N increases; with and without load balancing. Zipf data, $m = 2$, $p = 16$. 104

Figure 5.10 Performance for various workload mixes and query coverages (TPC-DS, PDC-MBR tree, $N = 400$ million, $p = 16$, $m = 4$). . 106

Figure 5.11 Performance for various workload mixes and query coverages (TPC-DS, PDC-MDS tree, $N = 400$ million, $p = 16$, $m = 4$). . 106

Figure 5.12 Query-heavy workload performance with increasing number of servers m. TPC-DS data, $p = 16$, workload mix = 5% inserts, average of all query coverages (5% ... 95%). 107

Figure 5.13 Performance for various workload mixes with increasing system size. Database size N and number of workers p both increasing. TPC-DS data, PDC-MBR tree, $\frac{N}{p} \approx 25$ million, $m = 4$, average of all query coverages (5% ... 95%). 108

Figure 5.14 Performance for various workload mixes with increasing system size. Database size N and number of workers p both increasing. TPC-DS data, PDC-MDS tree, $\frac{N}{p} \approx 25$ million, $m = 4$, average of all query coverages (5% ... 95%). 109

Abstract

On-line Analytical processing (OLAP) has been an important approach to the analysis of large structured data warehouse systems for many years. OLAP queries, in contrast to On-line Transaction processing (OLTP) queries typically access only a small portion of a data warehouse, may need to aggregate large portions of a data warehouse which often leads to performance bottlenecks. This problem is often compounded in real-time environments where new data may arrive frequently and at high velocity. One approach to addressing these performance challenges is to exploit collections of multi-core servers organized in cloud-based computing platforms.

In this thesis, we explore the design, implementation and evaluation of two new real-time cloud-based OLAP systems. In the first, we introduce *CR-OLAP*, a scalable *C*loud based *R*eal-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. They also highlighted that future scalability was only likely to be achieved in an architecture that supports multiple coordinating servers.

Based on our experience with *CR-OLAP*, we then present *VelocityOLAP* (*vOLAP*), a fully scalable cloud-based system for real-time OLAP on high velocity data. This system supports dimension hierarchies, is highly scalable, exploits both multi-core and multiprocessor parallelism, and guarantees strong serialization for both user and work group sessions. *vOLAP* is also built based on PDC-tree but supports multiple coordinating server processors and real-time dynamic load balancing. An experimental evaluation of our *vOLAP* prototype, using 18 worker instances for a database size of 1.5 billion items, shows that it is able to ingest new data items at a rate of over 600,000 items per second, and can process streams of interspersed inserts and OLAP queries in real-time at a rate of approximately 200,000 queries per second.

List of Abbreviations Used

Amazon EC2	Amazon Elastic Compute Cloud
AMI	Amazon Machine Image
AWS	Amazon Web Services
CR-OLAP	Cloud-based Real-time OLAP
DSS	Decision Support Systems
MBR	Minimum Bounding Rectangle
MDS	Minimum Describing Sets
OLAP	On-line Analytical Processing
OLTP	On-line Transaction Processing
PDC-tree	Parallel DC-tree
PDCR-tree	Parallel DC Range-tree
TPC-DS	Transaction Processing Performance Council
vOLAP	Velocity OLAP

Acknowledgements

Foremost, I would like to express my sincere gratitude to my supervisor Dr. Andrew Rau-Chaplin for the continuous support of my study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better supervisor and mentor for my study.

Besides my supervisor, I would like to thank the rest of my thesis committee: Dr. Peter Bodorik, Dr. Vlado Keselj, and Dr. Malcolm Heywood, for their encouragement, insightful comments, and hard questions.

I would also like to express my gratitude to Dr. Frank Dehne for his productive discussions and feedback during our project meetings. Thanks to Hamidreza Zaboli, David Robillard, Rebecca Zou and John Caskey for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last two years.

Last but not the least, I would like to thank my parents for giving birth to me and supporting me throughout my life.

Chapter 1

Introduction

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 [50, 61].

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 [46] 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.

Queries in on-line *transaction* processing (OLTP) systems which typically access only a small portion of the database (e.g. update a customer record). In contrast, 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 can pose

significant performance issues.

Most of the traditional OLAP research, and most of the commercial systems, follow the *static* data cube approach proposed by Gray et al. [46] 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. [27, 36, 46, 47, 60, 83]). 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 day or week. Hence, latest information can not be included in the decision support process. The static data cube also requires massive amounts of memory space and leads to a duplicate data repository that is separate from the on-line transaction processing (OLTP) system of the organization. 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. [20]). 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. [20, 54, 68, 67]). However, these approaches are not fully real-time. Furthermore, a major obstacle are the significant performance issues that arise in the context of very large scale data warehouses.

In the recent past, we have witnessed dramatic increases in the volume of data literally in every area–business, science and even daily life. According to a report from International Data Corporation in 2011, the overall created and copied data volume in the world was 1.8ZB (1ZB = 10^{21}bytes), which increased by nearly nine times within five years. This figure will double at least every other two years in the near future [43]. Google processes data of hundreds of Petabyte (1PB = 10^{15}bytes), Facebook generates over 10PB of log data per month, and Taobao, a subsidiary of

Alibaba, generate data of tens of Terabyte (1TB = 10^{12}bytes) for online trading per day [26]. The sharply increasing of data size comes with huge challenges on data storage, management and analysis. Traditional OLAP systems struggle to handle the huge volume of big data. One way to address this data challenge is by utilizing cloud based computing platforms.

With the rapid development of processing and storage technologies and the success of the Internet, computing resources have become cheaper and more powerful than before. These technological trends have resulted in the emergence of a new computing model called cloud computing, in which a shared pool of configurable computing resources (e.g., servers, storages and applications) can be provisioned and released easily through network access based on user demands [85, 13]. Cloud computing provides several compelling features that help reduce the infrastructure barrier between an innovative application and its requirements for scalability and performance issues:

* On-demand self-service: Cloud computing uses a pay-as-you-go pricing model. Users can start gaining benefit from cloud computing without investing in the construction of infrastructure. They can just simply rent resources from the cloud according to their requirements and pay for the usage.

* Rapid elasticity: Users can dynamically provision or release computing resources (e.g., servers or storages) depending on the load of their system.

* Easy access: Services hosted in the cloud are generally web-based. Thus, they can be accessed easily through devices (e.g., desktop and laptop computers, cellphones and PDAs) with Internet connections.

However, although cloud based applications have shown considerable potential in solving big data issues, designing and developing cloud-based applications always come with many challenges in terms of algorithms and system design [59]:

* Heterogeneity: A cloud based platform often consists of various of entities. One entity in the system should be able to inter-operate with others despite the differences in hardwares, operating systems, programming languages and data formats.

* Scalability: The cloud-based system should scale up efficiently and automatically by utilizing additional computing resources allocated from the Cloud.

* Concurrency: Cloud-based applications should allow concurrent access to shared resources in the system in order to achieve high performance.

* Migration and load balancing: Allow the movement of tasks and data sets within a system without stopping the system service, and distribute tasks and data sets among available computing resources to avoid system bottlenecks and improve performance.

In this thesis, we explore the design, implementation and evaluation of two new real-time cloud-based OLAP systems.

The basic building block of a cloud-based architecture is the modern multi-core processor. For a real-time OLAP system to be efficient in practice, it must at its heart have an efficient data structure that can answer OLAP queried with hierarchies, while efficiently exploiting the parallelism of multi-core processor architecture.

In 2000, Kriegel et al. [39] published an efficient data structure for processing OLAP for queries with hierarchies on data cubes, the DC-tree. A DC-tree is a sequential tree based index structure specially designed for data warehouses with dimension hierarchies. Even though it does provide an algorithmic solution for real-time OLAP systems and was proved to be efficient for small dataset, the DC-tree has not been used in commercial OLAP systems. A major problem is still performance. For large data warehouses, pre-computed cuboids still outperform real-time data structures even if has the major disadvantage of not allowing real-time updates.

In [38], a parallel DC-tree (PDC-tree) was proposed to support real-time OLAP system on multi-core processors. However, the PDC-tree method could only be used when database can be stored in a single machine. It becomes a problem when the database size is too big that can not be fit in one machine.

Starting with PDC-tree as a basic building block, this thesis explores OLAP architecture at the next scale up, in which many multi-core processors aggregate together in a cloud based architecture are efficiently harnessed.

In the first, we introduce CR-OLAP, a scalable *C*loud based *R*eal-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.

This work has been previously published in the 2013 IEEE International Conference on Big Data (IEEE BigData 2013) [37] and the Journal of Parallel and Distributed Computing (JPDC 2014) [35]. CR-OLAP demonstrated the feasibility of designing a real-time OLAP platform that scales well with increasing number of processors. However, CR-OLAP suffered from the following design weakness: CR-OLAP only supports one single stream of insert and query operations sent to one single master processor holding a master index. CR-OLAP's master processor easily becomes a

performance bottleneck and a single point of failure.

Our second real-time cloud-based OLAP system, *vOLAP* is a fully scalable, cloud-based real-time OLAP system for high velocity data. In the design of *vOLAP* we sought to address the bottleneck posed by the use of a single master in CR-OLAP. Similar to CR-OLAP, *vOLAP* also uses the PDC-tree as a building block but in a different way. In CR-OLAP, one single master processor is capable of receiving and processing one single stream of insert and query operations. The tree data structure in *vOLAP* has also been significantly improved for better handling of query coverage. As outlined in Section 5.1 and discussed in detail in the following section, *vOLAP* allows for an arbitrary number of input streams with interleaved insert and query operations, and these input streams are managed by multiple server processors whose number can be scaled to match the performance requirements. This work has been submitted to the 2014 IEEE International Conference on Big Data for review. *vOLAP* demonstrated the feasibility of designing a fully scalable and load balanced real-time OLAP system without any single node performance bottleneck.

1.1 Contributions

This thesis explored the design, implementation and evaluation of two new real-time cloud-based OLAP systems: *CR-OLAP* and *vOLAP*. We first introduce *CR-OLAP*, a scalable *C*loud based *R*eal-time *OLAP* system. Then we present VelocityOLAP(*vOLAP*): a fully scalable cloud-based system for real-time OLAP on high velocity data.

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 is kept in the processors' main memories [64]. 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. 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 [38], 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 an 16-fold increase in size, including an 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 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 number of dimensions, even simple range search (no dimension hierarchies, no aggregation) can degenerate to linear search (e.g. [39]). 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 second major contribution of this thesis is *vOLAP* which building on the ideas first developed in CR-OLAP.

vOLAP is a fully distributed, cloud-based system with a distributed tree data structure that uses a multi-threaded *PDC-tree* (designed for multi-core processors) as a building block. Data is partitioned into subsets stored in PDC-trees on *worker* nodes of the cloud environment. As is typical for current high performance OLAP systems, *vOLAP* is an in-memory system and supports ingestion of new data but no deletion. Multiple server nodes handle the incoming streams of new data inserts and OLAP queries, and route them to the appropriate workers. A Zookeeper cluster [52] is used for managing global information. A *manager* (background) process monitors the load status of the system and provides instructions to worker nodes for global real-time load balancing. An important property of *vOLAP* is that the load balancing is fully automatic and adjusts dynamically to the data distribution. Note that, due to the high velocity of incoming new data, the data distribution may change significantly

over time. Unlike other distributed OLAP systems, *vOLAP* does *not* use a static data partitioning dimension or a partitioning dimension that needs to be manually set by the systems administrator.

Each user session is attached to one of the server nodes. *vOLAP* guarantees strong serialization of the insert and OLAP query operations within each session. For multiple user sessions that are attached to the same server (e.g. as a work group), *vOLAP* also guarantees strong serialization between those sessions. Note that, since OLAP queries may need to aggregate large portions of the database and thereby overlap with many insert operations currently in progress, serialization is particularly challenging. Between multiple user sessions that are attached to different servers, *vOLAP* provides "best effort" serialization with a configurable bound on "freshness" (3 seconds, in our experiments).

Another important property of *vOLAP* is that it is fully scalable and supports an elastic cloud computing environment. Both server and worker nodes can be added or removed as necessary to adapt to the current workload. All system components are scalable and, unlike other systems, *vOLAP* avoids any single node performance bottleneck.

Experimental evaluation of our prototype system, using 18 workers for a database size of 1.5 billion items, shows that *vOLAP* is able to ingest new data items at a rate of over 600,000 items per second, and *vOLAP* can process streams of interspersed inserts and OLAP queries in real-time at approximately 200,000 queries per second. A distinguishing feature of *vOLAP* is that it exploits dimension hierarchies to improve performance. We have tested *vOLAP* on synthetic hierarchical data as well as TPC-DS test data which includes dimension hierarchies as shown in Fig. 4.8. The above mentioned performance evaluation of *vOLAP* is for data on such dimension hierarchies and includes a wide range of queries ranging from small queries, to queries that need to aggregate several hundred million data items, up to queries that need to aggregate

nearly the entire database.

CR-OLAP and *vOLAP* were designed and implemented in a collaborative setting. This thesis describes primarily the system issues that were the focus of my work. The multi-threaded PDCR trees used here were primarily developed by Hamidreza Zaboli and are described in his PhD thesis. The PDC-MBR and PDC-MDS data structures were primarily by developed David Robillard. My focus was the system design, software framework, migration and splitting protocols, and experimental analysis of performance on the cloud.

1.2 Structure of the thesis

The remainder of this thesis is organized as follows. In Chapter 2 we introduce the data management on the Cloud and some OLAP related concept and show some data structures that would be used in building a real-time OLAP system. In Chapter 3 we review cloud computing related techniques, and in Chapter 4 we present our *CR-OLAP* system for real-time OLAP on cloud architectures and the results of an experimental evaluation of *CR-OLAP* on the Amazon EC2 cloud. In Chapter 5, we show our fully scalable cloud-based system for real-time OLAP on high velocity data, *vOLAP*. Chapter 6 concludes the thesis.

Chapter 2

Background: Data Management on the Cloud

In this chapter, we introduce the concepts and technologies used in data management on the cloud and OLAP systems. We start with an introduction to NoSQL data stores on the Cloud. Then we move to a broad overview of OLAP systems in section 2.2. We also present the definitions of OLAP dimensions and hierarchies that will be used in the remainder of this thesis. In Section 2.3 we introduce several data structures that will be used later in our real-time OLAP systems. We first introduce the R-Tree in Section 2.3.1, then we present the definition and algorithms for DC-Tree in Section 2.3.2. We also describe how to create a parallel DC-Tree for multi-core processors in Section 2.3.3. Finally, we introduce the PDCR-Tree which will be used in our real-time OLAP systems.

2.1 Data Store Categorization

With the growing popularity of the Internet, many applications and services started being delivered to their users over the Internet. And the scale of these applications has been increased rapidly in recent years. As a result, many Internet companies, such as Google, Amazon and Facebook, faced the challenge of processing millions of concurrent user requests. Traditional relational databases could not provide the scalability, availability and performance they required. Hence, many companies and organizations decided to create their own non-relational solutions to satisfy their technical requirements. These non-relational data stores are often called NoSQL data stores [13].

According to a survey conducted by Cattell [23] in 2010, most NoSQL data stores can be classified into the following four groups based on their data model: OLAP stores, key-value stores, document stores and extensible record stores.

2.1.1 OLAP Stores

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 [50, 61]. 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.

In contrast to queries for on-line *transaction* processing (OLTP) [75] 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 historical append-only big dataset to support complex analysis (e.g. calculate the total sales of a certain type of items during a certain time period) which are computationally expensive. Therefore, most of the traditional OLAP research, and most of the commercial systems, follow the *static* data cube approach proposed by Gray et al. [46] 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. [27, 36, 46, 47, 60, 83]). However, the traditional *static* data cube approach has several

disadvantages. The OLAP system can only be updated periodically and in batches, e.g. once every week. Hence, latest information can not be included in the decision support process. The static data cube also requires massive amounts of memory space and leads to a duplicate data repository that is separate from the on-line transaction processing (OLTP) system of the organization. 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. [20]).

Most of the traditional OLAP solutions, such as SAP Business Objects [40] and Oracle OLAP database [70], follow the data cube approach. These traditional OLAP systems are not real-time OLAP systems and are often criticized for their limited scalability with increasing the database size. Microsoft OLAP system [63] is also implemented based on the data cube approach. And it also supports real-time analytical queries. To use real-time OLAP in Microsoft OLAP system, users are required to first create either a real-time dimension or a real-time cube. But real-time dimensions or real-time cubes in Microsoft OLAP system can not support remote partitions. Thus, Microsoft's real-time OLAP system fails to scale dynamically with needs.

In order to satisfy the requirements for querying on large dataset with high throughput and low latency, many cloud-based OLAP systems, such as [22, 25, 49], have been proposed based on the MapReduce [33] framework proposed by Google. By exploiting the parallelism of the MapReduce framework, the scalability of these system has been significantly improved. However, these MapReduce based OLAP systems are still not real-time OLAP systems.

SAP HANA [40] 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 [40]. But the distributed SAP HANA system only has one active master node while the system is working. Thus scalability of the system is limited.

Druid is an open-source, distributed, scalable, in-memory OLAP store designed for real-time exploratory queries on large-scale data sets. By combining a column-oriented storage layout, a distributed, shared-nothing architecture and an advanced indexing structure, Druid is able to compute drill-downs and aggregates over large quantities of multi-dimensional data records with low latency. Druid was originally designed to store and query large quantities of transactional events which are quite similar to the append-only historical data that is stored in OLAP systems. Data records in Druid are all timestamped events with multiple dimensions. Data records are partitioned into a set of segments based on timestamps. By replicating segments and distributing them to computing nodes, Druid is reported to be able to achieve high query throughput and low latency. Even if Druid shares many similarities with other OLAP systems, there still exists several obvious differences. First, all data records in Druid should come with a timestamp while this is not required in other traditional OLAP systems. Second, dimensional hierarchies, which is a key feature supported by most traditional OLAP systems, is not implemented in Druid.

All of the mentioned OLAP systems support typical OLAP queries. SAP HANA is a real-time OLAP system that could be scaled up to a cloud computing environment. However, the distributed SAP HANA system is a centralized system with only one master node. Thus, the master node might become the bottleneck of the whole

system when the frequency of user requests is increased. Druid is a scalable and fault-tolerant real-time OLAP system that is able to answer user queries with low latency. However, Druid can not work as a typical real-time OLAP because of it can only store timestamped transactional log data and does not support dimension hierarchies, which are implemented as a core feature in most traditional OLAP systems.

2.1.2 Key-value Stores

Key-value stores are similar to hashmaps and dictionaries that use a unique key to address data. Values in key-value stores are completely opaque to the system, hence, the only way to retrieve and update stored data is using keys. Also, no secondary keys or indices are used in these systems. Key value stores are useful for simple operations, which are based on key attributes only. Generally, key-value stores support only insert, delete and lookup operations.

Typically, scalability and high availability are the foremost requirements for key-value stores. According to the CAP [18] theorem, a distributed system can only choose two of consistency, availability and partition tolerance. In a cloud-based environment that network partition are inevitable, key-value store systems usually choose availability over consistency.

Amazon's Dynamo [34] is a key-value store used in many large-scale e-commerce applications. It is implemented based on a simple data model where each data record is addressed by a unique key and the value is a binary object. Dynamo provides a put and a get operation. Operations on multiple data records are not supported. It applies consistent hash to distributed data among computing nodes. Unlike traditional relational database systems that provide strong consistency, Dynamo only provides eventual consistency to achieve high performance and availability. Project Voldemort [41] is an open-source implementation of Amazon's Dynamo with substantial contributions from LinkedIn. Voldemort supports simple operations like get, put,

getAll and delete. There is no built-in support for range queries in Voldemort. Similar to Dynamo, Voldemort can be scaled to a large cluster using consistent hash and provides high query throughput. It also provides eventual consistency rather than strong consistency. Other similar key-value data stores include Riak [7], Redis [66], Scalaris [71], Tokyo Cabinet [51], Memcached [42] and Couchbase [19].

All of these key-value stores provide high scalability through key distribution over computing nodes and support simple operations like insert, delete and lookup. To achieve better load balancing, none of these key-value stores have built-in support for the range queries and hierarchical aggregation queries which are a fundamental requirement in OLAP systems.

2.1.3 Document Stores

Comparing to key-value stores that are designed for simple key-value pairs, document stores are designed for storing, retrieving and managing document-oriented information. Document-oriented information is known as semi-structured data. Within a document store, each document has a unique key that represents the document. A document key could be a string, a URI or a path. This unique key can be used to retrieve document from the database. Unlike the key-value stores that only use primary keys to address data records, document stores generally support secondary indexes and multiple types of documents per database. Similar to key-value stores, a unique key of a document can be used to retrieve a document from the database. Besides that, document stores also offer APIs to retrieve documents based on their content. For example, you can execute a query to retrieve all documents that contain a set of words (e.g. "NoSQL", "data" and "store").

Amazon's SimpleDB [72] is a highly available and flexible document store provided by Amazon as part of its web services. It provides simple operations for users to store and query documents via web service requests, thus offloads the work of database

administration. And it could be integrated easily with other AWS services such as Amazon S3 and EC2. Documents in SimpleDB are put into domains. Different domains may be stored on different Amazon nodes. And users can execute queries on one domain with many attributes constraints. SimpleDB supports Select, Delete, GetAttributes and PutAttributes operations on documents. Like most of the NoSQL systems, SimpleDB only supports eventual consistency, not transactional consistency, and it does asynchronous replication. CouchDB [14] is an open source document store implemented in Erlang. It uses JSON to store data and JavaScript as its query language. Documents in CouchDB are grouped into collections which are similar to domains in SimpleDB. CouchDB creates B-tree index for each collection, so the query results in CouchDB can be ordered. MongoDB [28] is another open-source document-oriented data store written in C++. Similar to CouchDB, it provides indexes on collections and document based queries.

Similar to the key-value stores, the document stores provide weaker consistency than traditional databases. But document stores provide richer operations on documents. Users can query a collection of documents based on multiple attribute value constraints. But hierarchical relations cannot be defined among attributes, thus document stores cannot be used to support OLAP queries.

2.1.4 Extensible Record Stores

Extensible record stores are also known as column-oriented stores, column family stores and wide columnar stores. Many existing extensible record stores seems to have been inspired by Google's BigTable [24], which is a distributed storage system for managing structured data that is designed to scale to very large sizes. The basic model used in extensible record stores is rows and columns. By splitting both the rows and columns, a table is split into a set of tablets which are distributed to many servers to achieve high scalability and load balancing. There is no replication of user

data inside BigTable. All user data are stored in the Google File System (GFS) [45] that provides scalable, consistent and fault-tolerant data storage. A master and a Chubby [21] cluster is designed to provide coordination and synchronization between tablet servers.

HBase [44] is an open source implementation of BigTable and it is written in Java. It uses Hadoop distributed file system [74] instead of Google File System to provide scalable, consistent and fault-tolerant data storage. Unlike BigTable, HBase uses Zookeeper [52] to provide coordination and metadata management services. Also, HBase added multiple master support to avoid single node failure.

HyperTable [55] is another open source column-oriented data store similar to BigTable and HBase and it's written in C++. Hypertable supports a number of programming language client interfaces, such as C++, Java, Python, Perl, Ruby and so on. Other similar systems include Facebook's Cassandra [56] and Yahoo's PNUTs [32].

Most of the existing extensible record stores are motivated by BigTable and provide similar functionalities. Generally, they support operations like write or delete values in a table, or look up values from individual rows. But they cannot be easily used as OLAP systems which store multi-dimensional data set with dimension hierarchies.

None of the above mentioned distributed data stores satisfy the requirements for a scalable real-time OLAP system with dimensional hierarchies. In this thesis, we are going to focus on designing a scalable, fault-tolerant and cloud-based real-time OLAP system. Thus, we only introduce OLAP related concepts and technologies in the following section.

2.2 OLAP

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 [50, 61]. 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.

In contrast to queries for on-line *transaction* processing (OLTP) [75] 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. [46] 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. [27, 36, 46, 47, 60, 83]). However, the traditional *static* data cube approach has several disadvantages. The OLAP system can only be updated periodically and in batches, e.g. once every week. Hence, latest information can not be included in the decision support process. The static data cube also requires massive amounts of memory space and leads to a duplicate data repository that is separate from the on-line transaction processing (OLTP) system of the organization. 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. [20]). 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. [20, 54, 68, 67]). However, these approaches are not fully real-time. A major obstacle are significant performance issues with large scale data warehouses.

In the remaining part of this chapter, we are going to explore sequential and parallel data structures that might be used to implement in-memory real-time OLAP systems that gets updated instantaneously as new data arrives and always provide up-to-date query result to support the decision making process.

2.2.1 Dimension Hierarchy

"A data cube consists of several functional attributes, grouped into dimensions, and some dependent attributes, called measures. For dimensions with more than one functional attribute, these attributes are organized into hierarchy schemas." [39] For example, the dimension Date can have functional attributes year, month, and date. Figure 2.1 shows an exmaple for a dimension Date and its functional attributes: Year, Month and Date. All is the root of every concept hierarchy and denotes the union of all values in the concept hierarchy.

Consider a data warehouse with a 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 $H_i = \{H_i^1, H_i^2, ..., H_i^l\}$ where a parent logically summarizes and includes its children. It is common for data warehouses used in practice to have a large set of dimensions each with its own hierarchy. As can be seen in the example drew from the TPC-DS benchmark data set shown in Figure 2.2, some of these dimensions may be very standard (e.g. Date: Year-Month-Day) while others are very application

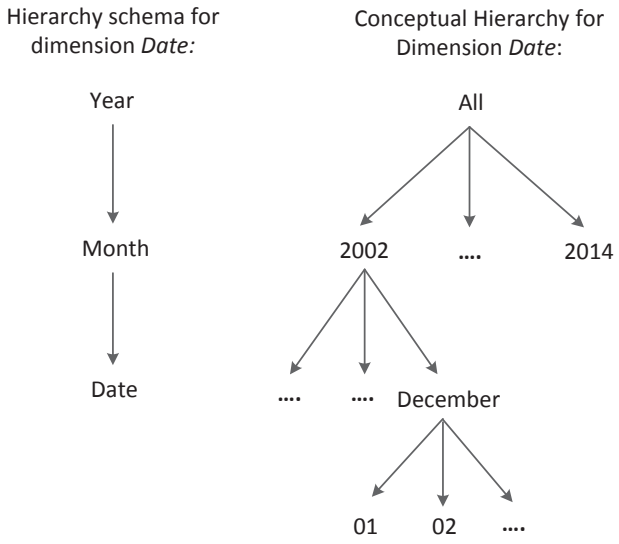

Figure 2.1: Hierarch Schema and Concept Hierarchy for dimension Customer [39]

specific (e.g. Store: Country-State-City).

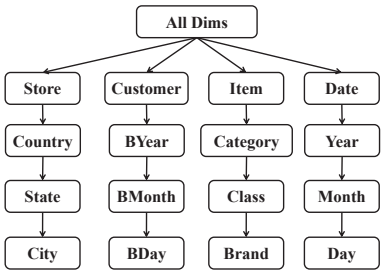

Figure 2.2: A 4-dimensional data warehouse with 3 hierarchy levels for each dimension. The first box for each dimension denotes the name of the dimension.

2.3 Data Structures for OLAP

Typical OLAP systems support data point insert and bulk insert operations, point query, range query and aggregation queries. Generally, OLAP systems don't support delete operations. Data sets stored in OLAP systems are multidimensional data points with dimensional hierarchies. In order to do fast insert and query operations

on multi-dimensional big dataset with dimensional hierarchies, we need to use a data structure that support efficient insertion and queries. In the following section, we explore several multi-dimensional data structures that will be used later in our real-time OLAP systems.

2.3.1 RTree

R-tree [48] is one of the most well-known multidimensional index structures that are widely used in both theoretical and applied contexts. It was proposed by Antonin Guttman in 1984 and was designed for spatial access methods, i.e., for indexing multidimensional information such as geographical coordinates, rectangles or polygons. R-tree has many variants including the R+-tree [73], R*-tree [16], VAMSplit R-tree [82].

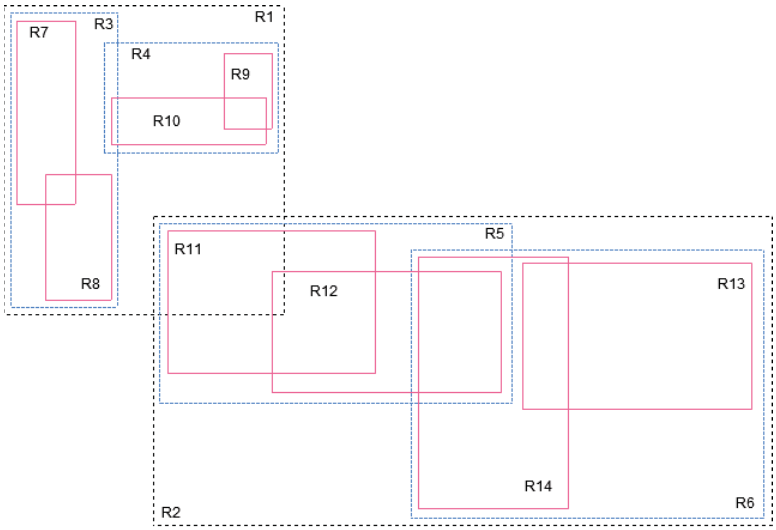

Figure 2.3: Containment and overlapping relationships among MBRs [48]

R-tree is a B-tree [15] like tree data structure and is height balanced. It uses minimum bounding rectangles (MBR) [48] to group the nearby multidimensional data, forming a hierarchical tree structure. Figure 2.3 shows a data distribution example and figure 2.4 shows the corresponding R-tree example.

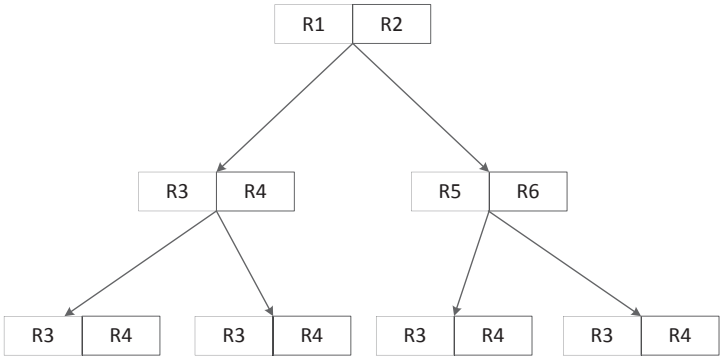

Figure 2.4: A 2-dimensional R-tree example

Data in R-tree is organized in pages and can be stored either in memory or on disk. The maximum and minimum number of data objects contained in each page should be set before creating a new R-tree. Like B-tree, R-tree consists of leaf nodes and directory nodes. Each directory node stores a minimum bounding box covering its children and the links to its children. Leaf nodes store the multidimensional data objects, which are normally represented as data points or polygon data.

The range search in R-tree is quite similar to the searching in B+ tree. It's a top-down procedure and started from the root node and stopped at the leaf node. Since each directory node contains a set of sub-directory nodes, the range search algorithm tests every entry of the directory node to decide if the bounding box of the entry overlaps with the query box. If it overlaps, the corresponding child node will be searched, otherwise, the corresponding sub-tree won't be traversed. The range search algorithm works in a recursive manner and won't stop until a leaf node is reached. Once a leaf node is reached, all data objects contained in the leaf node will be tested. If a data objected in a leaf node is contained in the query box, then this data object will be put into the result set.

2.3.2 DC Tree

The DC-tree [39] introduced by Kriegel et al. in 2000, is a tree based index data structure specifically designed for data warehouses with dimension hierarchies. It's a fully dynamic data structure that could be used to support OLAP queries on data cubes.

A DC-tree defines a concept hierarchy for each dimension, where the concept hierarchy is an additional tree structure for storing all values occurred in a given dimension. Using this concept hierarchy for each dimension, the DC-tree extends the usual R-tree based tree representation for multi-dimensional data by replacing the standard minimum bounding rectangles (MBR) with minimum describing rectangles (MDS).

An MDS is designed to describe a set of hyper-rectangles that contain the data stored in the corresponding subtree. In an MDS with multiple dimensions, each dimension contains a set of values at different levels of the dimension hierarchy. The DC-tree assigns an ID to every attribute value of a data record that is inserted. Thus, an MDS is actually consists of sets of IDs. Like MBRs, MDSs are designed to enable more efficient queries for high dimensional data with dimensional hierarchies.

Consider the following two data records in a data cube with two dimensions Store and Data and one measure:

* ([Canada, ON, Toronto],[2008, 01, 31], [$100])

* ([Canada, ON, Ottawa],[2008, 01, 31], [$200])

Figure 2.5 shows the hierarchies of the two dimension in the data cube. To distinguish data records from each other, an MDS of a single data record has to use the attribute values of the lowest level in the concept hierarchy of each dimension. As the attribute id at the lowest level of a hierarchy are usually represented as a surrogate key in each dimension, data records can distinguish each other based on the

attribute id in the lowest level. For the sample data record, the MDS is ([Toronto, Ottawa], [31]). An MDS that approximates a whole data node or a directory node may use values of higher levels in the concept hierarchies, e.g., ([ON], [2008]). The last element of the above sample data record is the measure value according to the measure attribute of the data cube. The measure value is not part of the MDS, but is related to it and will be stored together with the MDS in each node of the DC-tree. The measure value for an MDS of a datanode or a directory node is the aggregation (e.g. the sum or the average) of measure values of all data records covered by this MDS.

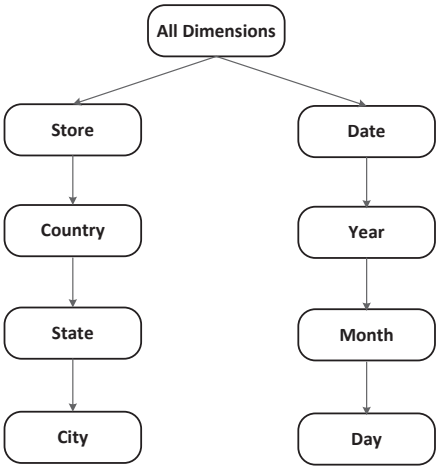

Figure 2.5: The dimensional hierarchies of a sample DC-tree

The DC-tree consists of three different types of nodes: data nodes, normal directory nodes and supernodes. The data nodes of the DC-tree contain minimum describing sets (MDSs) together with pointers to the actual data objects, and the directory nodes contain MDSs together with pointers to sub-MDSs. Supernodes are large directory nodes of variable size. The basic goal of supernodes is to avoid splits in the directory that would result in an inefficient directory structure. Figure 2.6 shows an example of the overall structure of the DC-tree, and Figure 2.5 presents the hierarchies of two

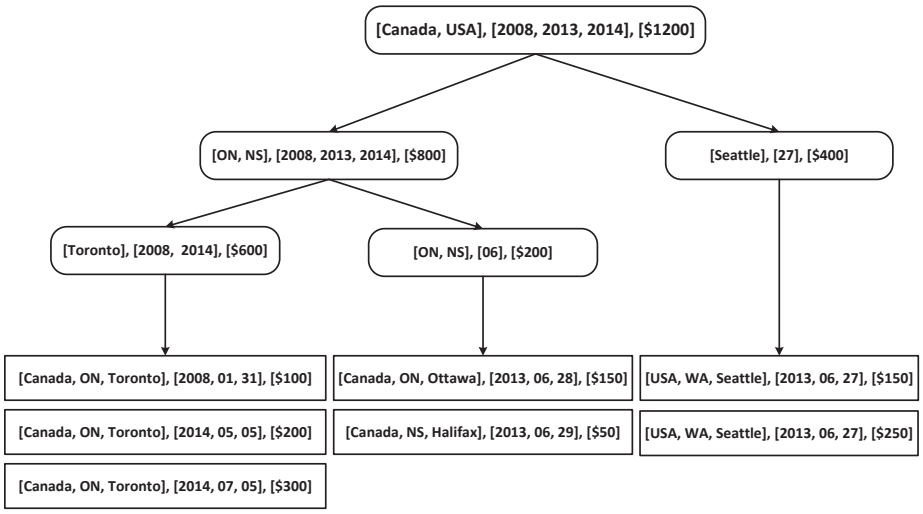

Figure 2.6: A simple DC-tree

dimensions. The DC-tree comes with two operations: Insert and Range Query.

The insertion algorithm takes a new data point P as the input and starts tracing down the tree using the MDS information as guidance. At each directory node, three cases may occur. If P is contained in the MDS of exactly one child, then the algorithm proceeds to that child. If P is contained in the MDS of more than one child, then the algorithm proceeds to the child with the smallest subtree. If P is not contained in the MDS of any child, then P needs to be added to the child whose MDS update leads to minimum overlap between children, in order to maintain efficiency of search queries. Once a leaf node is reached, then P is inserted to the leaf node. When an insert causes a node to exceed its capacity, a split operation will be invoked. The split operation runs through all dimensions to find an appropriate split and then split the exceeded node into two nodes, or ends with creating a supernode if no appropriate split was found.

The Range Query algorithm takes a query Q as the input and reports all data items contained in Q and their aggregate measure value. The Range Query algorithm is a top-down procedure started from the root node using the MDS information as a

guidance and won't stop until reached the leaf nodes. At each directory node, the algorithm runs throung every entry of the directory node. If the overlap between the MDS of the entry and the query MDS is empty, the entry is not relevant for the query and the result remains as it is. Otherwise, we have to further analyze the overlap. If the MDS of the entry is fully contained in the range, then the measure value stored in the son node referenced by the current directory entry, is added to the result. This is the advantage of the algorithm, because the entry and all nodes below this entry do not have to be considered and the algorithm can simply use the measure value computed during the insertions. If the MDS of the entry and the range overlap each other, we cannot use the measure value directly and have to recursively call the range query for the son node.

2.3.3 PDC Tree

Parallel DC-tree (PDC-tree) [38] is a data structure designed for parallel OLAP systems on shared memory multi-core processors. It was designed based on the sequential DC-tree and multi-threaded binary search tree techniques. The PDC-tree handles the case where all of the dimension hierarchies are categorical.

Given a client stream consists of insert and query operations, the main challenge for the parallel DC-tree is the possible interference between parallel insert and query operations. The PDC-tree need to guarantee the query correctness while improving the performance. It's easy for single core processors, but its challenging for shared memory multi-core processors. A straightforward solution could lock the subtree when an insert operation is performed on. However, this method lead to no speedup when increasing the number of processor cores.

With the idea of concept hierarchy structure of data cube and the directory node structure introduced in DC-tree, parallel DC-tree index consists of two more parts for its data structure which are 1) time stamp. 2) link to sibling. The time stamp

stores the most recent time when the node has been modified or created due to node splitting or insertion. The link to sibling is used to maintain a linear chain between the children of each directory node. These two fields are important for the system to perform parallel execution. Also the design acquires the lock to only lock the node which is currently updating.

The algorithms PARALLEL-OLAP-INSERT tracks down the tree by checking the MDS of each directory node to find the right leaf directory node where the new data item should be inserted. Three cases will be considered. When the new item MDS is contained in only one directory node MDS, the algorithm traces down its child. When the new item MDS is included in more than one directory node MDS, the algorithm will choose the smallest subtree to go down. When the new item MDS has overlap with the directory node MDS, choose the node whose MDS has minimal overlap with its neighbors caused by the MDS enlargement if possible insertion happens.

The step will be repeated until the leaf directory node is reached. Then the node will be locked in order to perform insertion. If the node capacity has been exceeded with the insertion, it will call split algorithm to do node splitting. During the process, the MDS and measure will get updated from the bottom nodes until no further update or root node is reached.

The algorithm PARALLEL-OLAP-QUERY takes MDS of the given query range as an input. A stack is used to traverse the tree and detect the modified and new nodes raised by parallel insertion, so that all the new data inserted in parallel with query search can be evaluated as well. Starting from the root, the process pushes one directory node to the stack, and then evaluates its children nodes. The MDS of child node and the MDS of the given query range whoever has lower level hierarchy will be converted the higher hierarchy level so that the two MDS are the same level. Two cases can happen in this step. If the child's MDS is contained in range query MDS, then add the child's measure to the results. If the child's MDS has overlap with query

MDS, then the child will be pushed to the stack.

When searching in a subtree, the direct node of this subtree could be modified with new time stamp caused by parallel insertion. So when the corresponding original node with earlier time stamp is popped up from the stack, we know there are new siblings that need to be evaluated and the subtrees of the updated node should be re-visited. Therefore, the sibling directory nodes are pushed into stack and so is the update directory node. Repeat the steps until the stack is empty.

Chapter 3

Background: Cloud Computing

In this chapter, we introduce the cloud computing related concepts and technologies that are used in this thesis. We start with a broad overview of cloud computing concepts and definitions. We cover the history, characteristics and three primary service models for cloud computing. In Section 3.5, we describe the Amazon Web Services (AWS) [30] platform and its services. In Section 3.2, we introduce software mechanisms of communication in cloud environments and discuss how to do remote node communication in a environment that satisfies the requirements for scalability and load balancing. Finally, we describe the challenges of node coordination in cloud computing environments and illustrate how Zookeeper [52] can be used to coordinate global state the system.

3.1 Cloud Computing

Cloud computing means many different things to different people. In this thesis, we use the following standard definition:

"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." [58]

The cloud computing concept can be dated back to the 1950s, when large-scale mainframe computers [4] became available in research organizations and corporations. A mainframe was normally stored in a server room, and multiple clients/users were

able to access the mainframe via terminal computers, which had no internal processing capacities and can only be used for communications. Due to the big cost of buying and maintaining mainframes, organizations couldn't afford to buy a mainframe for each user. Thus, to make mainframes to be used efficiently, organizations decided to allow multiple users to share access to both storage resources and CPU time of a mainframe. By sharing the time of a mainframe, an organization was able to get better return on its investment of mainframes. The practice of time-sharing formed the initial concept of cloud computing.

In 1966, Douglas Parkhill published a book, *The Challenge of Computer Utility* [62] which illustrated the idea of organizing computation as a public utility. Almost all of the modern characteristics of cloud computing were thoroughly explored from this book. As the cost of computer hardwares became lower, more and more users were able to afford their own computer. Also, with the popularity of network in the 1990s, users started running into a different type of problem: one single computer is not enough to provide the computing resource as required. The industry started thinking about shifting from "splitting up the computing resource" to "combing the computing resources of different computers via network". This shifting formed the basic concept of modern cloud computing.

By installing and configuring softwares across multiple physical nodes, a system would work like a single physical node. Such a system utilized the computing resources of all physical nodes and provided users with computing resources as they needed, we call it a cloud computing system. In cloud computing systems, it was easy to add new computing resources to the "cloud": just need to add a new computer to the physical system and configure so that it become part of the bigger system.

Since 2000, Amazon played a key role in the development of cloud computing technology. Amazon developed their first internal cloud system by modernizing their existing data centers. Having found that the cloud system improved the working

efficiency significantly, Amazon developed a new cloud computing system to provide computing resources, known as Amazon Web Services (AWS) [30], to external customers. Today AWS is one of the most well-known cloud computing platform. Hundreds of thousands of worldwide users are using AWS's computing resources. Some other similar cloud computing platforms include Google App Engine [29] and Microsoft Azure [78].

3.1.1 Characteristics

The National Institute of Standards and Technology has defined the following five essential characteristics for cloud computing [58]:

* **On-demand self-service**. A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provider.

* **Broad network access**. Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, tablets, laptops, and workstations).

* **Resource pooling**. The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand.

* **Rapid elasticity**. Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear unlimited and can be appropriated in any quantity at any time.

* **Measured service**. Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service.

A cloud computing service has several distinct features that are different from traditional distributed computing services or grid computing services. Firstly, it's sold on demand. Users of the service could select the way they want to charge, typically it's charged by the minute or the hour. Secondly, it's elastic, which means a user can request as much or as little resource from the cloud as they want at any given time. It's fully dynamic and really on-demand. Finally, the cloud is managed and maintained by the service provider. To use the cloud service, users only need a computer and the access to connect to the cloud service.

3.1.2 Service models

The National Institute of Standards and Technology also has defined the following three service models for cloud computing [58]:

* **Software as a Service (SaaS)**. The capability provided to the consumer is to use the providers applications running on a cloud infrastructure. The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g., web-based email), or a program interface. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings.

* **Platform as a Service (PaaS)**. The capability provided to the consumer is to

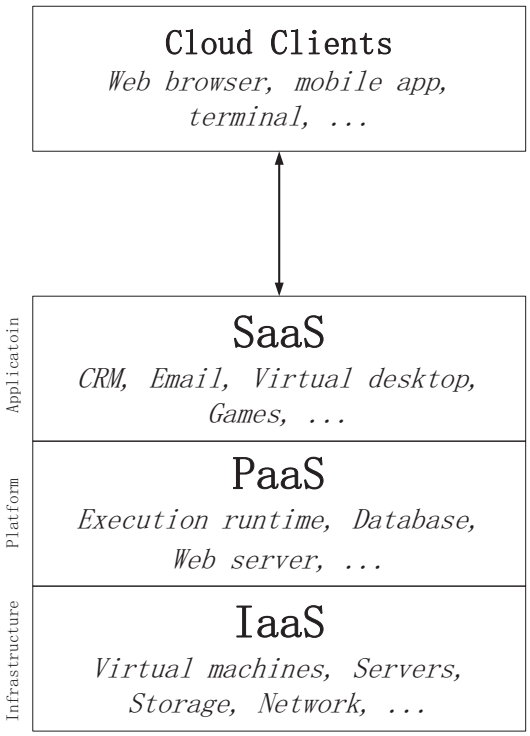

Figure 3.1: Three cloud computing service models [2]

deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment.

* **Infrastructure as a Service (IaaS)**. The capability provided to the consumer is to provision processing, storage, networks and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating

systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls).

Infrastructure as a service (IaaS) is the most basic cloud-service model. Amazon EC2 is a typical example of such a service provider. IaaS providers offer physical or virtual machines for users to use. Users may also get services like file-based storage, firewalls, load balancers, IP addresses and virtual local area networks (VLANs) from IaaS provider. These resources typically are installed in data centers and provided to users based on user demand. To deploy and run applications, cloud users need to install operating-system images and their own software packages on the requested cloud infrastructure. In this IaaS model, cloud users is responsible for maintaining the operating systems and software packages. Cloud providers only charge on the amount of resources user consumed.

Platform as a service (PaaS) is another widely used cloud-service model. In the PaaS service model, the service is provided as a "computing platform", which could be regarded as operating system, database or web server. Application developers can focusing on developing and running their software solutions without buying and managing the hardware and softwares. GoogleApps are examples PaaS.

In the software as a service (SaaS) model, service providers supplies the hardware infrastructure, the software solutions to users. Service users just need a terminal to connect to the service provider when they need to use this service. In this service model, users can get rid of the cost to buy a expensive software, avoid the complicated steps to install a software and maintain the installed software. Service users just use the software when they need it with low cost.

3.2 Communication in the Cloud

When working with large-scale distributed systems, there is always a need for communication between connected components. In a typical distributed database system with master nodes and slave nodes, master node might receive requests from clients and then forward requests to slave nodes. Slave nodes may need to send calculation results back the master node. Both forwarding requests and exchanging calculation results need data exchange among machines. Also, in a load balanced distributed system, data might be migrated from one machine to another machine to balance the workload of the system. Data migration is another example for exchanging data in distributed applications.

Berkeley Sockets (BSD) are the underlying API for all network communication. BSD sockets is the original implementation of the TCP/IP suite and one of the most widely supported components of any operating system today. In BSD sockets, peer-to-peer connections often requires explicit setup, destroy, choice of transport (TCP, UDP), error handling and so on. It's really hard to implement reliable and efficient network services using low-level BSD sockets without deep understanding of I/O multiplexing and buffer management, not to say the scalable data and work distribution patterns required by modern large-scale distributed systems.

Message queues are often used for data exchange in large-scale distributed applications. In computer science, a message queue is defined as a software-engineering component that could be used for interprocess communication or for inter-thread communication within the same process. Basically, a message queue is a service that receives data from the sending process and delivers it to a process that receives the data. Message queues often provide an asynchronous communication protocol which allows the sender and receiver to work independently without interacting with the message queue at the same time. Messages are stored in the message queue once they

are inserted by the sender and won't be removed until the receiver retrieves them. Message queue services often implement advanced features, such as different communication patterns, delivery acknowledgement, security control, high performance and other functions. Using message queues can help us get rid of headache to maintain raw sockets. Moreover, message queues can help significantly improve the scalability of distributed systems [5].

For example, in a distributed system, you may have one server that receives requests from clients to perform some time-consuming calculations. That server forwards the the received request to a second server which performs the actual calculation while the first server waits for a response. Since the first server waits for the computation results before moving to the next request, the whole system is limited by the computation on the second server. If we can increasing the number of servers that perform computation and distributed the computation task to multiple computation servers, the throughput of the system is improved. But the first server still need to wait for each computation server to complete before moving to the next request. A better solution is to send requests to computation servers without waiting for a response. This is where message queues come in. The first server can send requests to the message queue which maintains the list of requests. Computation servers remove messages from the queue, calculate the results and send the results back to the first server. The first server receives calculated results from another queue. If the load of the system is increased, we just need to increase the number of computation servers. The new added computation servers read and remove messages from the same queue.

Many message queues are designed and implemented for a specific operating system or an application, such as [80, 30, 77]. While many other message queues are designed to allow message communication among different operating systems and different applications. These message queue services often come with more high-level features. Examples of these commercial implementation of message queue services include

IBM's WebSphere MQ [31] and Oracle Advanced Queuing [65]. Open source choices of message queue services include ZeroMQ [9], RabbitMQ [79], Apache ActiveMQ [76] and JBoss Messaging [53].

3.3 Serialization in the Cloud

Distributed systems may receive a large number of transactions at the same time. In order to achieve high performance (high throughput and low latency), transactions in distributed systems are generally executed concurrently (they overloap in time). Serializability is the major correctness criterion for concurrent execution of transactions. In the concurrency control of distributed systems, a transaction stream (consisting of insert and query operations) is serializable if its outcome is equal to the outcome if its transactions executed serially, i.e, executed sequentially without time overloap [17, 81].

For example, a distributed system received a transaction stream consisting of insert and query operations: I1, I2, Q1, I3, Q2, I4, I5, Q3. In order to achieve high performance, insert and query transactions may execute in the distributed system at the same time. In this example, Q3 is guaranteed to have all the inserted data prior to Q3 included in the query result if transactions are executed sequentially, thus, the query result of Q3 should include I1, I2, I3, I4, and I5. Hence, the outcome of Q3 should always contain the all data inserted prior to Q3 if the transaction stream is serializaable. Otherwise, if the outcome of Q3 doesn't contain all data inserted prior to Q3, e.g., Q3 only contains I1, I2 and I3, then the transaction stream is not serializable.

Transaction streams that are not serializable are likely to generate erroneous outcomes. A well known example is the distributed banking system. A credit account in the banking system allows multiple users to operate this account at the same time. If the related transaction stream is not serializable, then the total sum of money may

not be preserved. Money paid by this credit account may exceed the maximum credit. Money could disappear, or be generated from nowhere. These erroneous outcomes could be caused by one transaction overwrite the data that has been written by another transaction before it's permanently applied in the database. However, these erroneous outcome do not happen if transaction stream serializability is maintained.

Unlike financial applications, many applications don't require restrict serialiability. For example, data warehouse systems normally store years of historical data. When retrieving the sales history of a specific product, it doesn't matter much if a product was updated a short time ago (e.g., 15 seconds or 2 minutes). This information will eventually be updated in the data warehouse system and included in the query result a short time later. Under such scenarios, relaxing serialization to achieve high performance is a better choice comparing to requiring restrict serialization.

In *vOLAP* which will be introduced in Chapter 5, our system guarantees session serialization instead of serialization. That is, each user session is attached to one of the server nodes. The insert and query transactions within each session is guaranteed to be serializable. But between multiple user sessions, *vOLAP* doesn't guarantee strong serialization. By relaxing the serialization, we achieved higher performance in *vOLAP*.

3.4 Coordination in the Cloud

In distributed systems, consensus is a fundamental problem that involves several processes agreeing on a value in the presence of failure. The problem is often posed in a typical distributed environment where network communication is reliable but processors may fail. In general, the problem can be defined to involve a single coordinator, which sends a binary value to n -1 participants and lets all participants agree on the same value. Several protocols have been proposed to solve the consensus problem in distributed systems. The Paxos [57] protocol is one of the most well-known

consensus protocols. Systems that can be used to solve consensus problems are called coordination systems [12].

Coordination is an essential service in cloud computing. Large-scale distribution applications require different types of coordination services. System configuration sharing is a simple example of coordination. In a distributed system with master nodes and slave nodes, there often exists a list of operational parameters that should be shared among master nodes and slave node. Group membership and leader election are also common coordination services in large-scale distributed systems. Using the distributed system mentioned above as an example, master node always needs to know the number of alive slave nodes in the system and the available resource of each alive node. Master node may use this information to balance the workload of the system while processing client requests. Also, in a distributed system with multiple masters, a leader master may need to be elected to be in charge of the system. In some specific scenarios, the execution order for slave nodes should be guaranteed in a distributed system. In this case, a distributed lock may be required to be implemented as another example of coordination.

There exist many services designed for different coordination needs. For example, in [69], a leader election service is designed especially for dynamic distributed systems. In [11], the authors proposed a configuration system for sharing configuration among machines in a distributed system. More examples of powerful services that implement primitives which could be used to implement simple coordination services are Zookeeper [52] and Chubby [21].

3.5 Amazon Web Services

Amazon Web Services (AWS) is cloud computing platform that is composed of a collection of remote computing services, including compute power, storage, databases,

messaging and other services. Amazon EC2 and Amazon S3 are the two most well-known services. Most of our experiments were done on Amazon EC2.

Amazon has a long history and rich experience of building and using decentralized, large-scale, reliable, efficient IT infrastructures. These infrastructures enabled their development teams to access compute and storage resources on demand, thus helped increase their productivity and agility. In 2006, Amazon decided to launch Amazon Web Services (AWS) so that other organizations and companies could use these web services and benefit from Amazon's rich experience in running and maintaining large-scale distributed IT infrastructure. Today AWS has been deployed in 8 geographical regions all over the world and hundreds of thousands of worldwide customers are using this global cloud computing platform.

Using AWS, you can requisition compute power, storage, and other services in minutes and have the flexibility to choose the development platform or programming model that makes the most sense for the problems you are trying to solve. You pay only for what you use, with no up-front expenses or long-term commitments, making AWS a cost-effective way to deliver applications.

3.5.1 Amazon EC2

Amazon Elastic Compute Cloud (EC2) [1] is a one of the most well-known services of Amazon.com's cloud computing platform, Amazon Web Services (AWS). Amazon EC2 is a typical example of PaaS model and allows cloud service users to rent computers which could be physical or virtual machine in cloud and run applications on these rented computers. A virtual computer rent from Amazon EC2 is called an instance. Amazon allows cloud users to boot an Amazon Machine Image (AMI) to create instances. Users have options to use the images provided in cloud service or create their own image. Users have the control to create desired number of instances using an AMI, and can launch and terminate instances as needed. There are three types of

instances offered by Amazon:

* On-demand: pay by hour without commitment
* Reserved: rent instances with one-time payment receiving discounts on the hourly charge
* Spot: bid-based services (runs the jobs only if the spot price is below the bid specified by bidder)

3.6 ZeroMQ

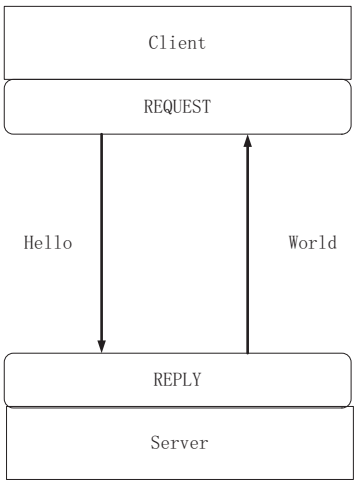

Figure 3.2: Request Reply pattern [3]

As we mentioned in above sections, using and maintaining raw sockets (like BSD sockets) are difficult and cumbersome. There also exists some high level libraries (e.g. Boost asio) that provide standard socket API while hiding the details of maintaining raw sockets. But most existing high-level socket libraries are implemented with the cost of performance and limited two simple communication patterns: bi-directional or multicast. These libraries can not provide scalable data and work distribution

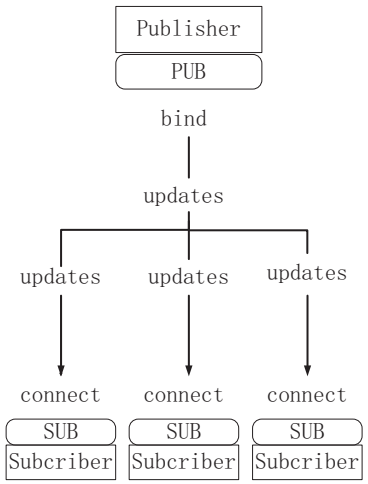

Figure 3.3: Pub-sub pattern [3]

patterns required by modern large-scale distributed systems. This is where ZeroMQ networking library comes in.

ZeroMQ (also seen as MQ, 0MQ, zmq) is a network library that can be used in diverse environments, such as financial services, game development, embedded systems, academic research and aerospace. It was originally conceived as a fast messaging system for stock trading, thus, it was started with the target of high throughput and low latency. ZeroMQ provides simple user interfaces that make sending messages and receiving messages really easy compared to raw socket implementation. For sending a message to a remote machine using ZeroMQ, you can just invoke an asynchronous send call with specified destination, it will queue the message in a separate thread and do the remaining work for you. Your application does not have to wait until the message has been flushed. Messages are guaranteed to arrive the specified destination quickly and safely. ZeroMQ sockets provide a lot of features that make it to be a scalable library. A single ZeroMQ socket can connect to multiple end points and automatically load balance messages over them. A single socket can also collect messages from multiple sources. Some existing messaging systems are implemented with a central

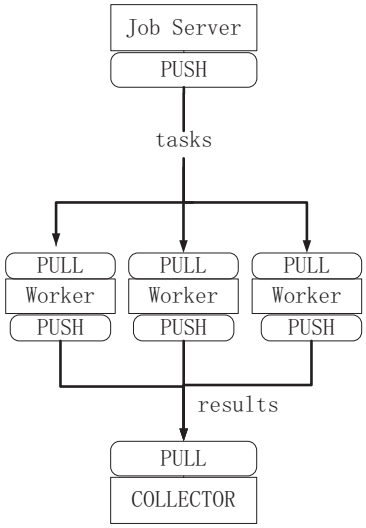

Figure 3.4: Pipeline pattern [3]

broker that is used as the global load balancer. Central broker design always comes with performance bottleneck on the central broker and single point failure problem. While ZeroMQ followed a brokerless messaging model so that there is no single point of failure. Also, this brokerless model makes ZeroMQ become a perfect choice for distributed applications because of its good scalability feature.

ZeroMQ supports 4 different transports: INPROC, IPC, MULTICAST and TCP. TCP transport in ZeroMQ is implemented based on the standard protocol that is familiar to most people, which MULTICAST is encapsulated in UDP. If communication among different machines is required, TCP transport is the best choice. If there is no need to communicate cross the machine boarder, IPC (Inter-Process communication model) or INPROC (In-Process communication model) may be used to lower the latency.

To help distributed data and work among nodes efficiently and easily, ZeroMQ provides 4 different message patterns. There patterns are implemented by pairs of socket with matching types. The following shows the 4 built-in core ZeroMQ patterns:

* **Request-reply**, which connects a set of clients to a set of services. This is

a remote procedure call and task distribution pattern. See figure 3.2 as an example.

* **Pub-sub**, which connects a set of publishers to a set of subscribers. This is a data distribution pattern. See figure 3.3 as an example.

* **Pipeline**, which connects nodes in a fan-out/fan-in pattern that can have multiple steps and loops. This is a parallel task distribution and collection pattern. See figure 3.4 as an example.

3.7 Zookeeper

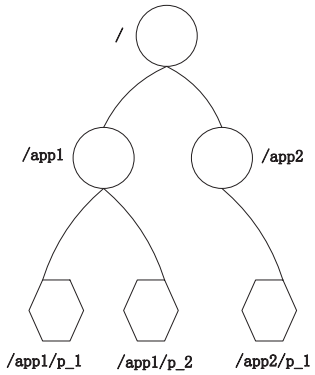

Figure 3.5: Zookeeper's hierarchical namespace [10]

Zookeeper is a distributed, open-source and high-performance coordination service designed for distributed systems. It implemented a simple set of primitives which could be used to build high-level coordination services, such as synchronization, configuration, group membership, leader election and locks. It's easy to program with its simple directory tree like data model and it provides simple APIs for both Java and C.

Zookeeper has implemented a shared hierarchal namespace which is quite similar to a standard file system. Distributed processes can use this shared hierarchal namespace

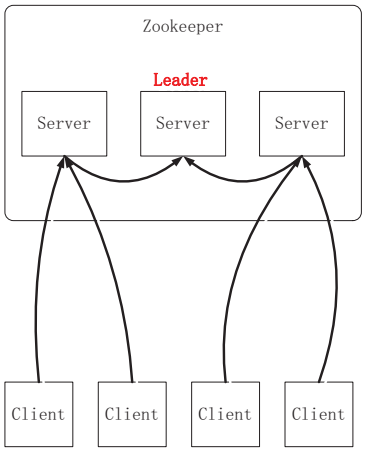

Figure 3.6: Zookeeper's architecture [10]

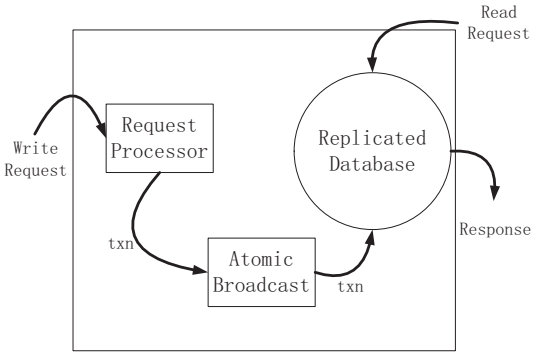

Figure 3.7: Zookeeper components [10]

to coordinate with each other. The namespace is consist of znodes which are similar to files and directories in standard file system. Figure 3.5 shows an example of Zookeeper's hierarchical namespace. Unlike typical file systems which are stored on hard disk, Zookeeper data is kept in memory, which allows Zookeeper to achieve really high performance for its read and write operations.

To guarantee that data maintained in Zookeeper is highly reliable, Zookeeper itself is replicated over a set of servers. The servers that providing Zookeeper service know about each other. Each server maintains an in-memory image of the data maintained in Zookeeper, along with the transaction logs and snapshots of the image which could

be in failure recovery. As long as a majority of the servers are available, the Zookeeper service will be available. Figure 3.6 shows the architecture of Zookeeper. Zookeeper service consists of Zookeeper servers and Zookeeper clients. Zookeeper servers are designed to maintain the data that should be shared among distributed processes while Zookeeper clients are often embedded in distributed processes which use Zookeeper clients to communicate with Zookeeper servers. One of the servers will be elected as the leader server while the rest become followers automatically. Every Zookeeper server service clients. Zookeeper clients connect to one server every time to submit requests. Read requests can be answered by using the local replica of the image. But the requests to change the data in Zookeeper, which are called write requests, are forwarded to the leader server. Leader server is responsible for processing all write requests from clients. Figure 3.7 shows more details of how Zookeeper answer clients' read and write requests. All write requests are logged to disk for failure recovery and also serialized to disk before they are applied to the in-memory data. Zookeeper uses a custom atomic messaging protocol to gurantee that all replicas are also in synchronization.

Zookeeper is designed to be simple, fast and reliable. It provides the following features which are critical for a distributed coordination service:

* Sequential Consistency — Updates from a client will be applied in the order that they were sent

* Atomicity — Updates either succeed or fail. No partial results.

* Reliability — Once an update has been applied, it will persist from that time forward until a client overwrites the update

* Single System Image — A client will see the same view of the service regardless of the server that it connects to.

* Timeliness — The clients view of the system is guaranteed to be up-to-date within a certain time bound.

Chapter 4

CR-OLAP

In this chapter, we introduce *CR-OLAP*, a scalable Cloud based real-time *OLAP* system. We start with a broad overview of our *CR-OLAP* system. In Section 4.2 we introduce the PDCR tree data structure and in Section 4.3 we present our *CR-OLAP* system for real-time OLAP on cloud architectures. Section 4.4 shows the results of an experimental evaluation of *CR-OLAP* on the Amazon EC2 cloud, and Section 4.5 concludes the paper.

4.1 Introduction

We introduce *CR-OLAP*, a scalable *C*loud based *R*eal-time *OLAP* system that utilizes a new distributed index structure for OLAP, refered 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. [39] and our previous PDC tree [38], 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 is kept in the processors' main memories [64]. 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. 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 [38], 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 an 16-fold increase in size, including an 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 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 number of dimensions, even simple range search (no dimension hierarchies, no aggregation) can degenerate to linear search (e.g. [39]). 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").

4.2 PDCR Trees

The *sequential* DC tree introduced in Chapter 2 exploits knowledge about the structure of individual dimension hierarchies both for compact data representation and accelerated OLAP query processing. We also introduced the parallel DC tree for multi-core processors in Chapter 2. In this section, we outline a modified PDC tree, termed PDCR tree, which will become the building block for our *CR-OLAP* system. The PDCR tree extends the PDC tree in the following ways: 1) It supports both categorical and continuous dimension hierarchies. 2) Its memory layout is designed carefully to support the efficient communication of subtrees in a distributed memory system.

Here, we only outline the differences between the PDCR tree and its predecessors,

and we refer to [38, 39] 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 [39] is particularly well suited to handle OLAP queries for both, ordered and unordered dimensions.

Consider a data warehouse with 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 4.1 shows the dimensions and hierarchy levels of each dimension for a 4-dimensional data warehouse.

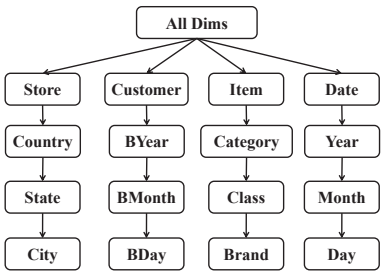

Figure 4.1: A 4-dimensional data warehouse with 3 hierarchy levels for each dimension. The first box for each dimension denotes the name of the dimension.

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 4.3 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 Figure 4.1, 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 4.2 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 4.3 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. [39] and our previous PDC tree [38] store so called "minimum describing set" (MDS) entries at each internal tree node to guide the query process; see [39] for details. The MDS concept was

developed in [39] 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 4.4. 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"*.

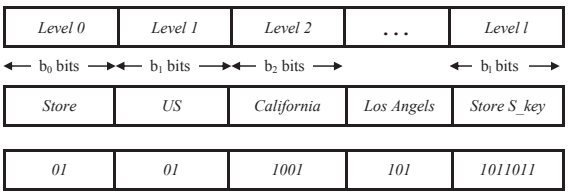

Figure 4.2: Illustration of the compact bit representation of IDs.

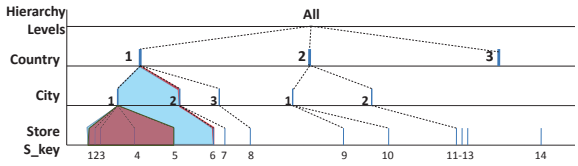

Figure 4.3: Example of relationships between different hierarchy levels of a given dimension.

4.3 CR-OLAP: Cloud based Real-time OLAP

CR-OLAP utilizes a cloud infrastructure consisting of $m + 1$ multi-core processors

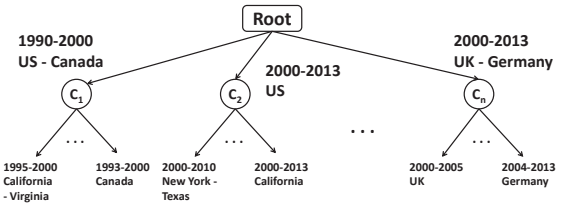

Figure 4.4: Example of a PDCR tree with 2 dimensions (Store and Date).

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*. 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 [39], PDC trees [38] and PDCR trees are leaf oriented. All data is 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 Figure 4.5.

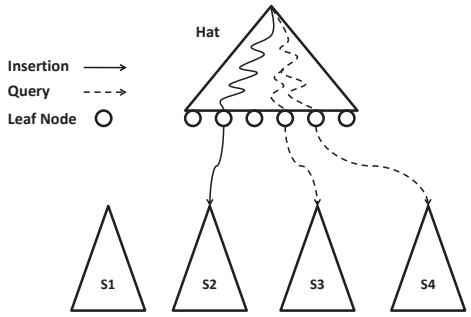

Figure 4.5: Illustration of a distributed PDCR tree.

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 Figure 4.5, both *insert* and *query* operations are executed concurrently.

4.3.1 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 Algorithm 3 and Algorithm 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 Algorithm 1 and Algorithm 2.

Figures 4.6 and 4.7 illustrate how *new* workers and *new* subtrees are added as more data items get inserted. Figures 4.6 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 4.7. 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.

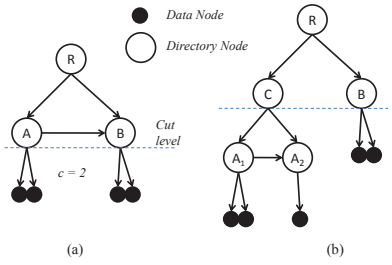

Figure 4.6: Insertions triggering creation of *new* workers and subtrees. Part 1. (a) Current *hat* configuration. (b) Insertions create overflow at node A and horizontal split.

For correct *real time* processing of an input stream of mixed *insert* and *query*

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.

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.
    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.

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.

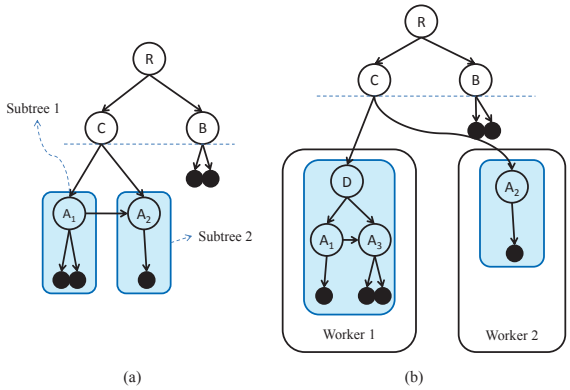

Figure 4.7: Insertions triggering creation of *new* workers and subtrees. Part 2. (a) Same as Figure 4.6b with critical subtrees highlighted. (b) Insertions create overflow at node C and vertical split, triggering the creation of two subtrees in two different workers.

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 [38] 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 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 [38] 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 4.6 and 4.7. 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 [38]. 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. According to the correctness proofs for splitting subtrees in Zaboli's Ph.D thesis [84], correct *real time* processing of mixed *insert* and *query* operations is still maintained.

4.3.2 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 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 it's 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'.

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.

4.4 Experimental Evaluation On Amazon EC2

4.4.1 Software

CR-OLAP was implemented in C++, using the g++ compiler, OpenMP for multi-threading, and ZeroMQ [9] 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 $k = 16$.

4.4.2 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 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 m3.2xlarge 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.

4.4.3 Comparison baseline: STREAM-OLAP

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.

4.4.4 Test data

For our experimental evaluation of *CR-OLAP* and *STREAM-OLAP* we used the standard TPC-DS "Decision Support" benchmark for OLAP systems [8]. 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. Figure 4.8 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.

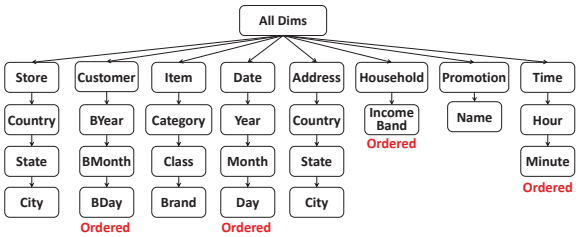

Figure 4.8: The 8 dimensions of the TPC-DS benchmark for the fact table "Store Sales". Boxes below each dimension specify between 1 and 3 hierarchy levels for the respective dimension. Some dimensions are "ordered" and the remaining are not ordered.

4.4.5 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 4.9, 4.10, and 4.11.

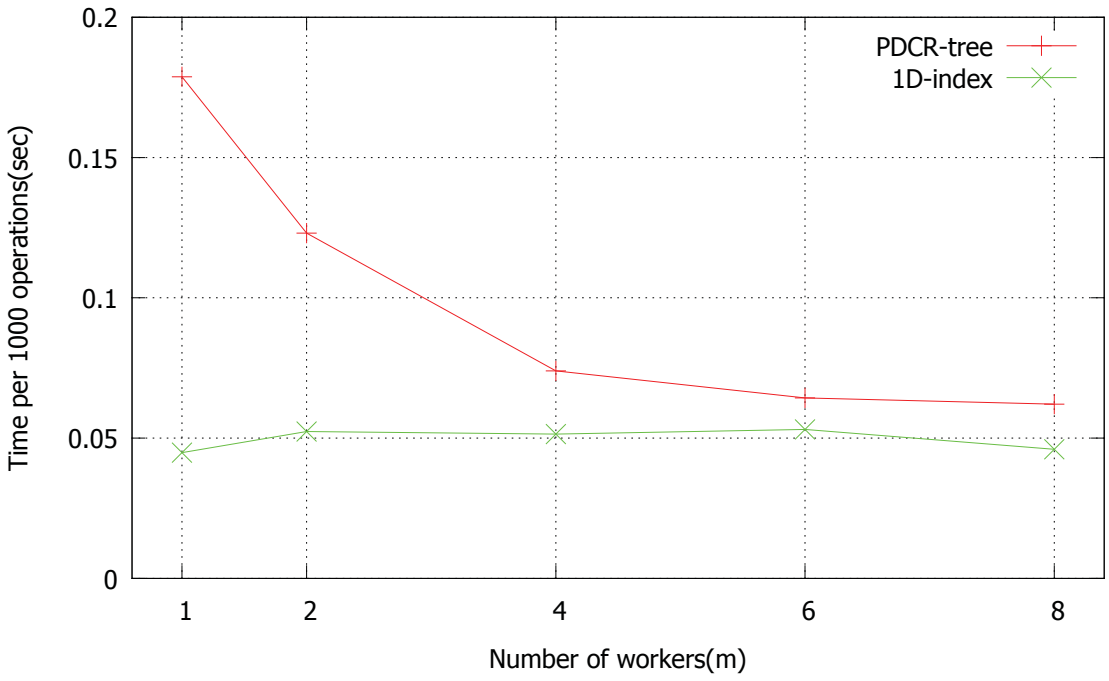

Figure 4.9: Time for 1000 insertions as a function of the number of workers. ($N = 40Mil$, $d = 8$, $1 \leq m \leq 8$)

Figure 4.9 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,

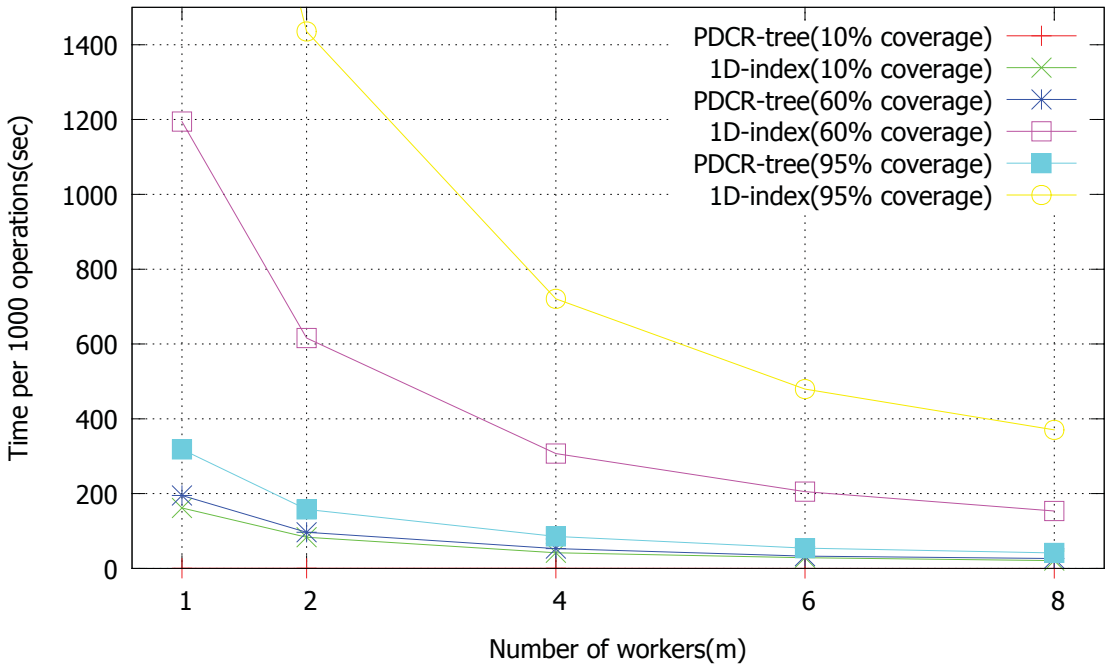

Figure 4.10: Time for 1000 queries as a function of the number of workers. ($N = 40Mil$, $d = 8$, $1 \leq m \leq 8$)

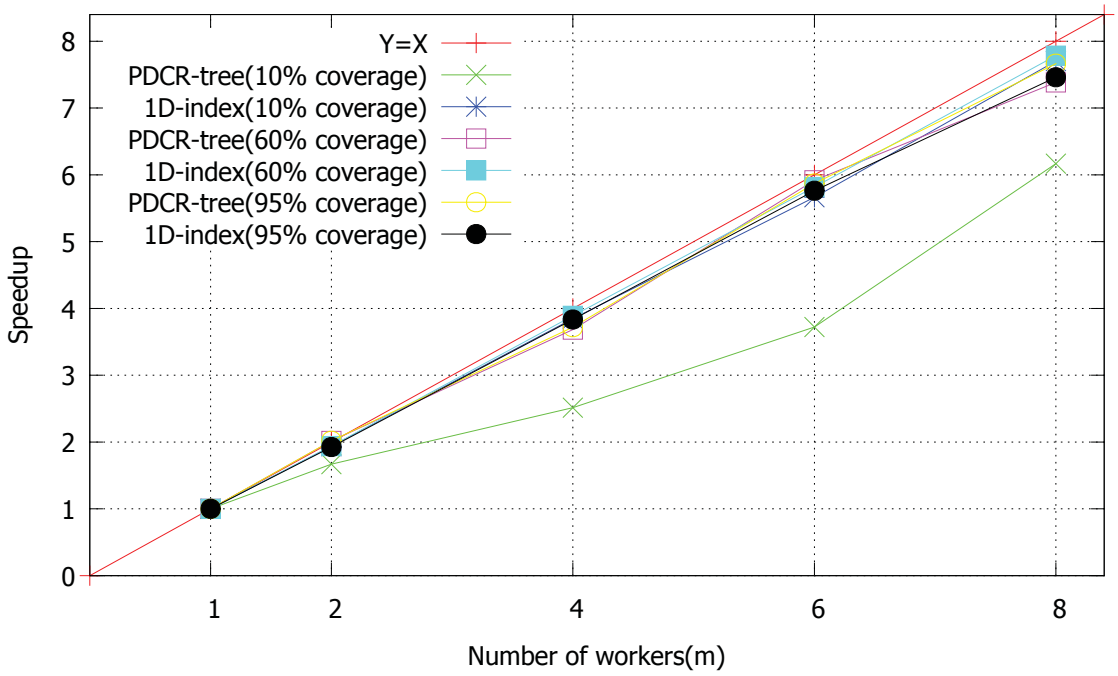

Figure 4.11: Speedup for 1000 queries as a function of the number of workers. ($N = 40Mil$, $d = 8$, $1 \leq m \leq 8$)

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 4.10 shows the time for 1,000 OLAP queries in *CR-OLAP* and *STREAM-OLAP* as a function of the number of workers (m). Figure 4.11 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 4.10 that *CR-OLAP* significantly outperforms *STREAM-OLAP* with respect to query time (in some cases 2,000%). 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%. Figure 4.11 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 Figure 4.11 implies a fixed cost/performance ratio. For example, cutting query response time in half would come at the price of doubling the system cost.

4.4.6 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. In our scale up experiments, as we increase the number N of data items from 10 Mil 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 Figures 4.12 and 4.13, respectively.

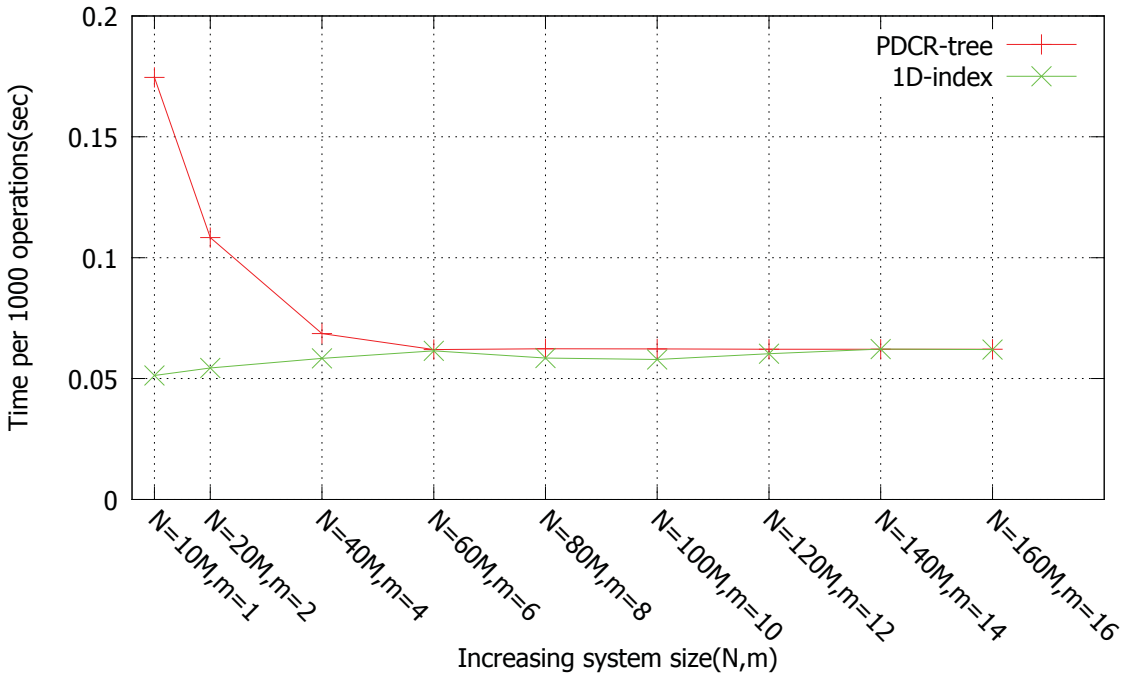

Figure 4.12: Time for 1000 insertions as a function of system size: N & m combined. ($10Mil \leq N \leq 160Mil$, $d = 8$, $1 \leq m \leq 16$)

With growing system size, the time for *insert* operations in *CR-OLAP* (PDCR-tree)

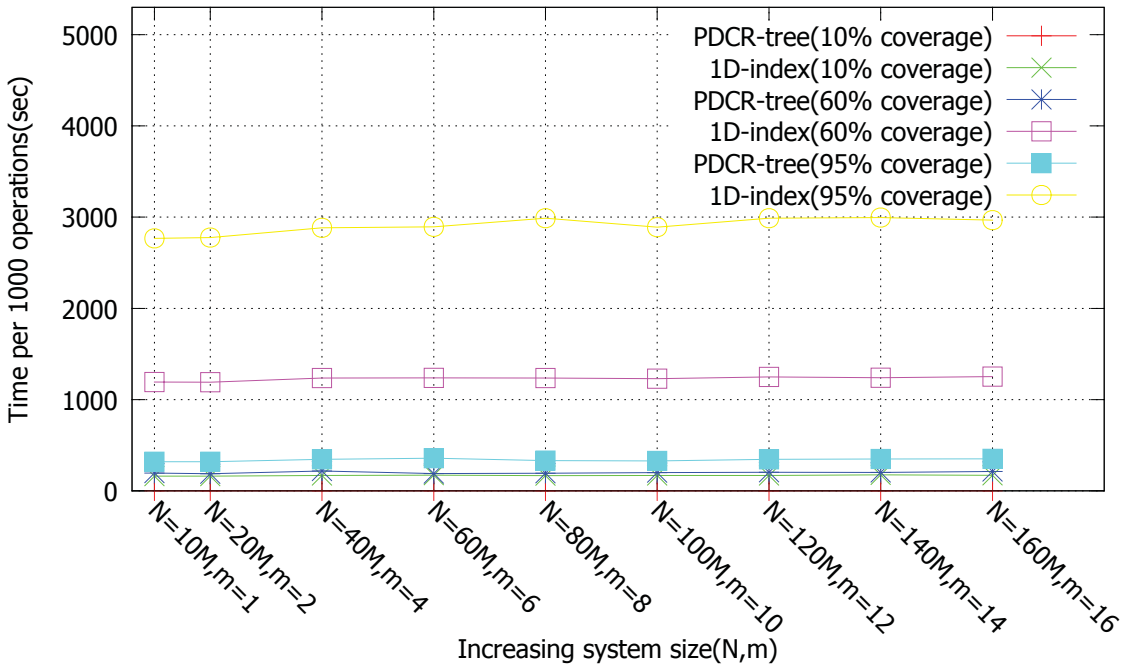

Figure 4.13: Time for 1000 queries as a function of system size: N & m combined. ($10Mil \leq N \leq 160Mil$, $d = 8$, $1 \leq m \leq 16$)

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 1,000%), as shown in Figure 4.13. Also, it is very interesting that 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 4.13 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. It performed an 16-fold increase in database size and number of processors, including an 16-fold increase in the average amount of data aggregated by each OLAP query, without noticeable performance impact for the user.

4.4.7 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 Firgure 4.14, the number of concurrent query streams (clients) has no impact on query performance.

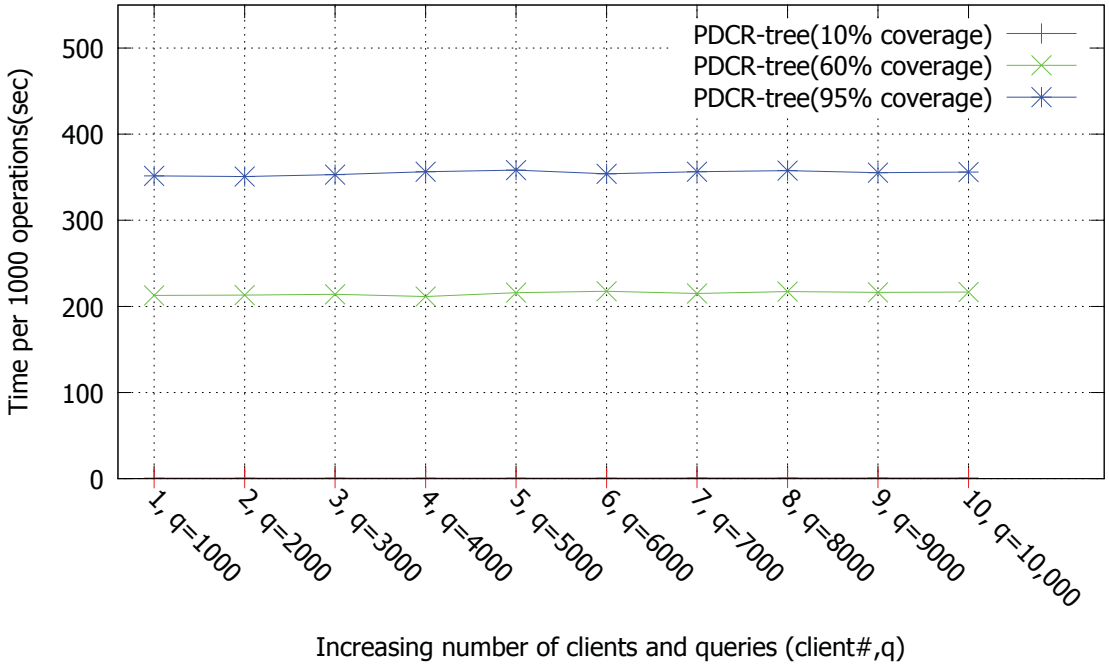

Figure 4.14: Time for 1000 OLAP queries as a function of of the number of query streams. X-axis first parameter: number of query streams (clients). X-axis second parameter: total number of queries issued (1,000 queries per query stream). Y-axis: Average time per 1,000 queries in seconds. ($N = 160Mil$, $d = 8$, $m = 16$)

4.4.8 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 Figures 4.15 and 4.16 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.

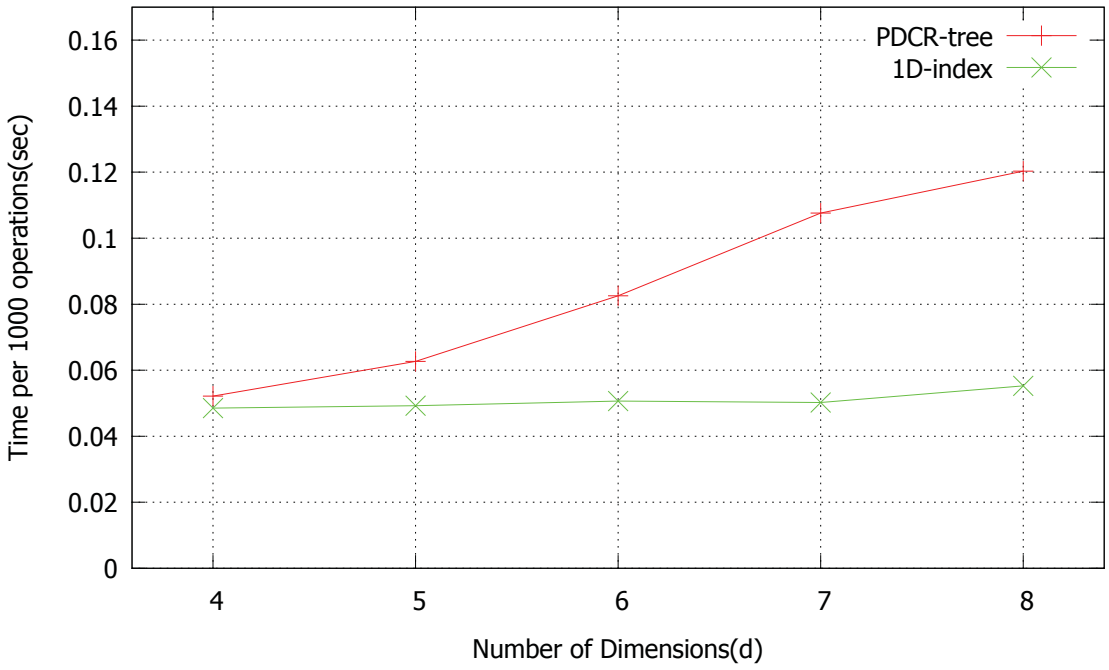

Figure 4.15: Time for 1000 insertions as a function of the number of dimensions. ($N = 40Mil$, $4 \leq d \leq 8$, $m = 8$)

Figure 4.15 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 4.16, 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.

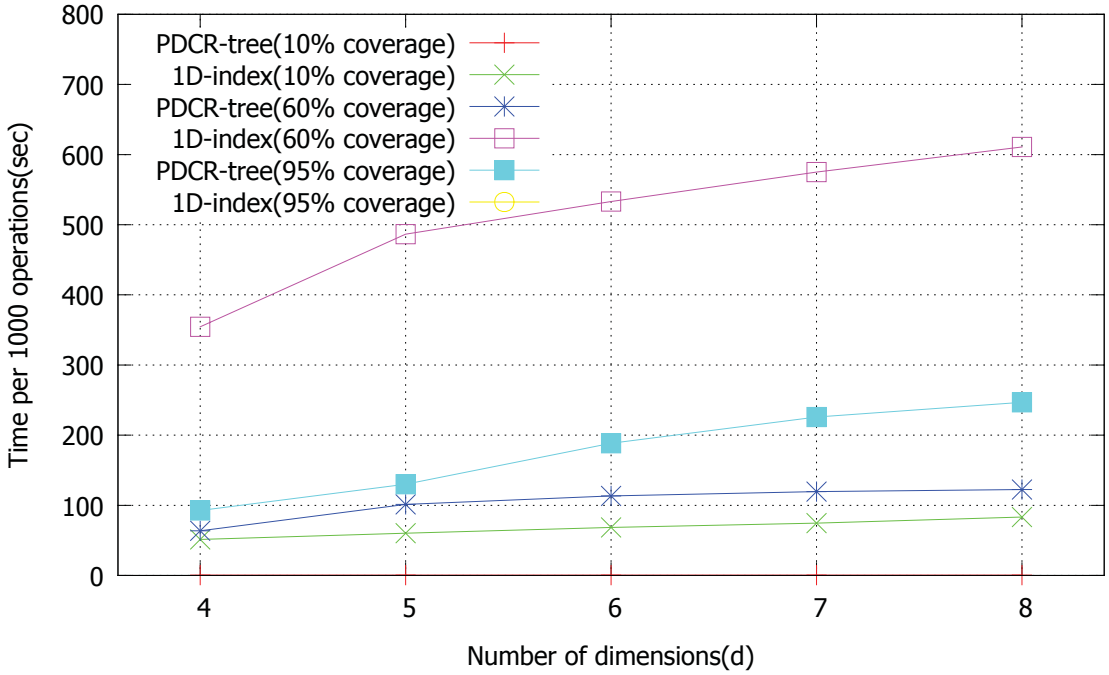

Figure 4.16: Time for 1000 queries as a function of the number of dimensions. The values for "1D-index 95% coverage" are 828.6, 1166.4, 1238.5, 1419.7 and 1457.8, respectively. ($N = 40Mil$, $4 \leq d \leq 8$, $m = 8$)

4.4.9 Test results: impact of query coverages

Figures 4.17, 4.18, 4.19, and 4.20 show the impact of query coverage on query performance in *CR-OLAP* (PDCR-tree) and *STREAM-OLAP* (1D-index).

For fixed database size $N = 40Mil$, 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 4.17 and 4.18 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 4.19, and 4.20 show the performance of *STREAM-OLAP* as compared to *CR-OLAP* (ratio of query times). It shows that *CR-OLAP* consistently outperforms *STREAM-OLAP*

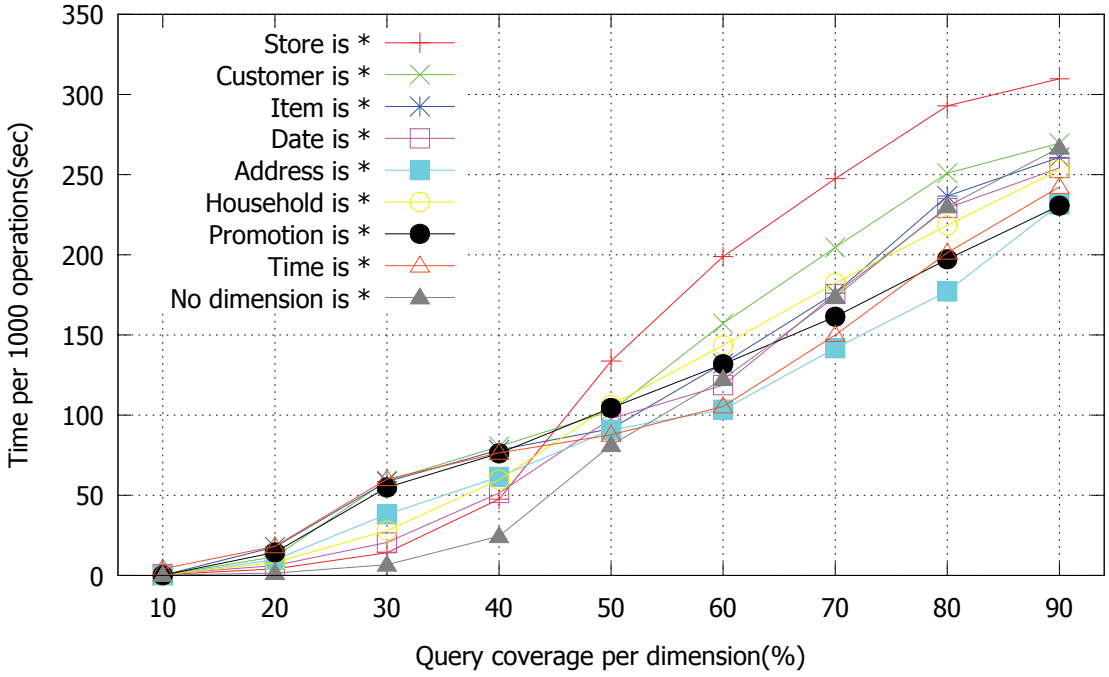

Figure 4.17: Time for 1000 queries (PDCR tree) as a function of query coverages: $10\% - 90\%$. Impact of value "*" for different dimensions. ($N = 40Mil$, $m = 8$, $d = 8$)

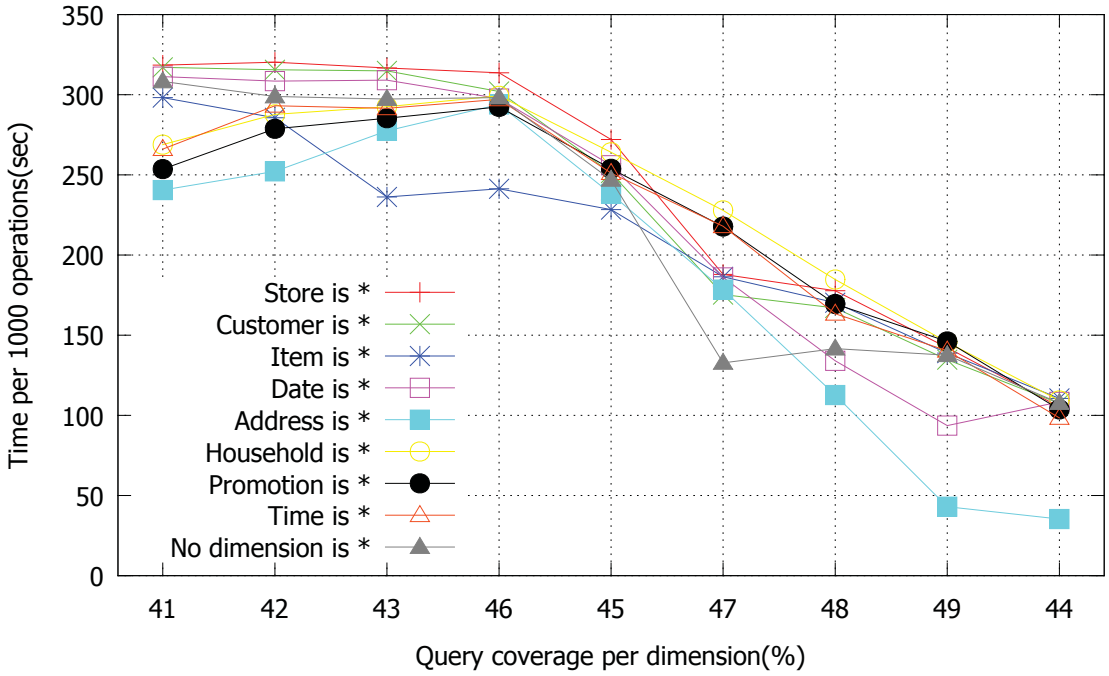

Figure 4.18: Time for 1000 queries (PDCR tree) as a function of query coverages: $91\% - 99\%$. Impact of value "*" for different dimensions. ($N = 40Mil$, $m = 8$, $d = 8$)

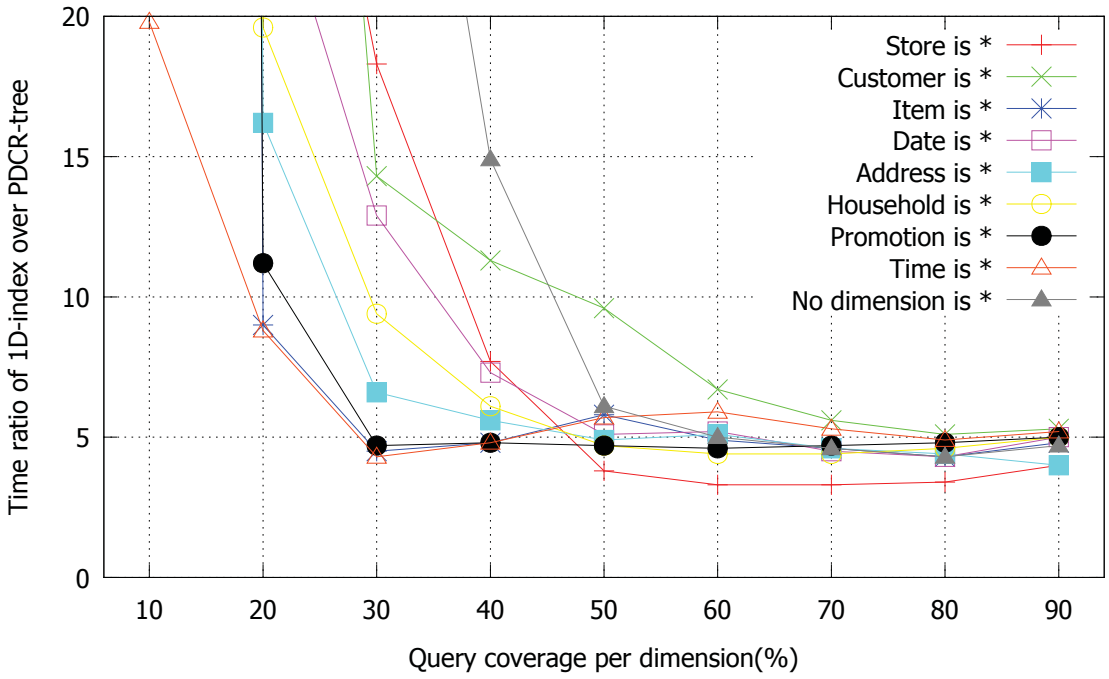

Figure 4.19: Time comparison for 1000 queries (Ratio: 1D-index / PDCR tree) for query coverages $10\% - 90\%$. Impact of value "*" for different dimensions. ($N = 40Mil$, $m = 8$, $d = 8$)

by a factor between 5 and 20.

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

Figure 4.21 shows a query time comparison between *CR-OLAP* (PDCR-tree) and *STREAM-OLAP* (1D-index) for selected query patterns. For fixed database size $N = 40Mil$, 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

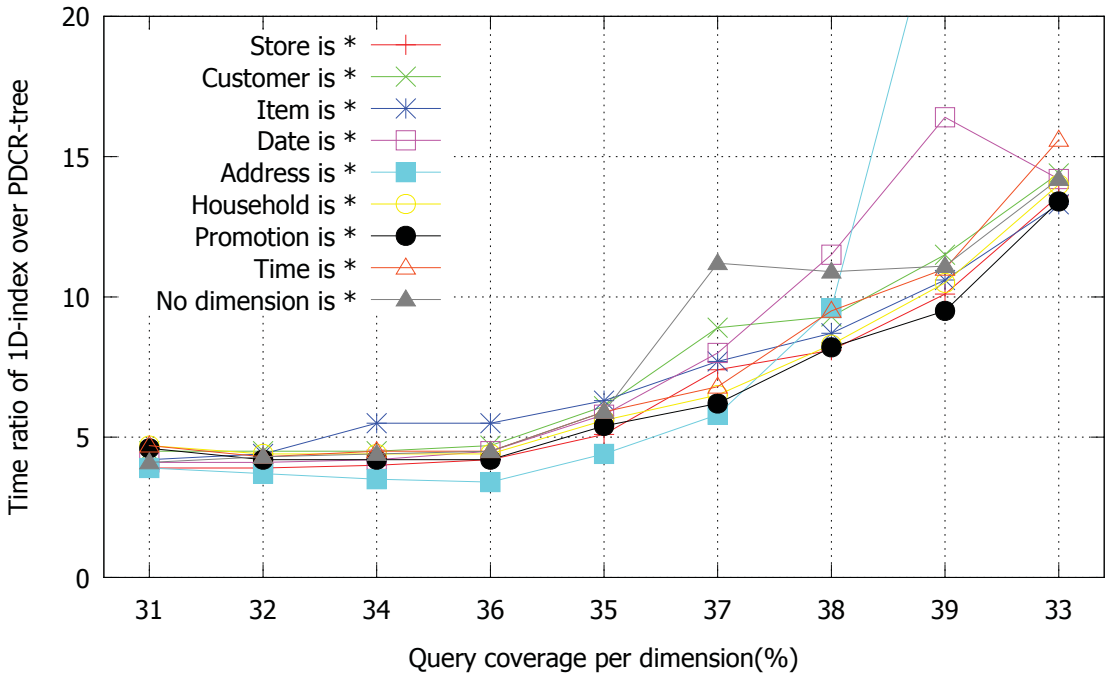

Figure 4.20: Time comparison for 1000 queries (Ratio: 1D-index / PDCR tree) for query coverages $91\% - 99\%$. Impact of value "*" for different dimensions. ($N = 40Mil$, $m = 8$, $d = 8$)

coverages 10%, 60%, and 95%. The test results are summarized in Figure 4.21. 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.

4.5 Conclusion

We introduced *CR-OLAP*, a *C*loud based *R*eal-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 an 16-fold increase in database size and number of processors, including an 16-fold increase in the average amount of data aggregated by each OLAP query, without noticeable

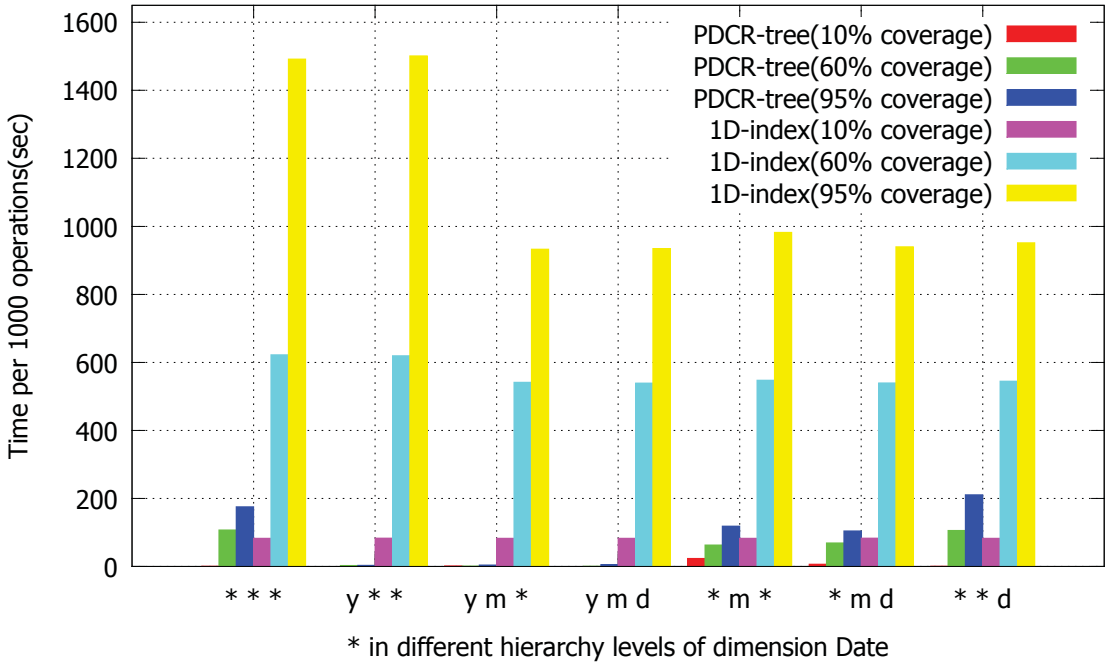

Figure 4.21: Query time comparison for selected query patterns for dimension *Date*. Impact of value "*" for different hierarchy levels of dimension *Date*. ($N = 40Mil$, $m = 8$, $d = 8$).

performance impact for the user.

Chapter 5

vOLAP

In this chapter, we introduce *vOLAP*, a scalable real-time OLAP system for high velocity data in cloud environments. We give a broad overview of our *vOLAP* system in Section 5.1. Section 5.2 describes the design of *vOLAP*, including network architecture, synchronization and data representation. In Section 5.3, we demonstrate the OLAP insert and query algorithms. Section 5.4 shows the load balancing strategies used in *vOLAP*. Section 5.6 presents experimental results that demonstrate the performance of our prototype implementation and concludes with a large scale experiment.

5.1 Introduction

Previously, we introduced CR-OLAP, a scalable cloud based real-time OLAP system based on the distributed PDCR tree. CR-OLAP utilizes scalable cloud infrastructure consisting of multiple commodity processors. Tests conducted on Amazon EC2 demonstrated that CR-OLAP scales well with increasing number of processors, even for complex queries. However, CR-OALP suffered from a design weakness: CR-OLAP is a centralized system with a single master processor. CR-OLAP only supports one single stream of insert and query operations sent to one single master processor holding a master index. CR-OLAP's master processor easily becomes a performance bottleneck and single point of failure.

In the design of *vOLAP* we sought to address the bottleneck posed by the use of a single master in CR-OALP. Similar to CR-OLAP, *vOLAP* also uses the PDC-tree as a building block. In *vOLAP*, the tree data structure has been completely

distributed and the master index has been replaced by multiple distributed indices. This leads to better scaling and significantly better query performance as well as much faster concurrent ingestion of new data (Section 5.6.1, Fig. 5.8 and Section 5.6.3, Fig. 5.12). *vOLAP* allows for an arbitrary number of input streams with interleaved insert and query operations, and these input streams are managed by multiple server processors whose number can be scaled to match the performance requirements. Another major improvement in *vOLAP* over CR-OLAP is a new socket and message queue architecture which allows for much higher message throughput and better load balancing between threads. Packed message support for data ingestion has also been added which dramatically increases throughput when inserting large amounts of data. Rather than storing global information on a single master, *vOLAP* uses a Zookeeper cluster, thereby alleviating another potential bottleneck. In addition, *vOLAP* adds a *manager* process which initiates dynamic, real-time load balancing via movement of data between workers. As a result, the differences in performance between CR-OLAP and *vOLAP* are substantial. Whereas CR-OLAP scaled up to at most 160 million data items and was limited by the capacity of the master node, *vOLAP* scales to billions of data items. The query performance of *vOLAP* is several orders of magnitude better than CR-OLAP, and is more resilient to queries with high coverage.

VelocityOLAP (*vOLAP*), a scalable real-time OLAP system for high velocity data in cloud environments. *vOLAP* is a fully distributed, cloud based, system that uses a multi-threaded *PDC-tree* as a building block. Data is partitioned into subsets stored in PDC-trees on *worker* nodes of the cloud environment. As is typical for current high performance OLAP systems, *vOLAP* is an in-memory system and supports ingestion of new data but no deletion. Multiple server nodes handle the incoming streams of new data inserts and OLAP queries, and route them to the appropriate workers. Zookeeper [52] is used for managing global information. A *manager* background process monitors the load status of the system and provides instructions to worker

nodes for global real-time load balancing. This load balancing is fully automatic and adjusts dynamically to the data distribution, which can change significantly over time due to the high velocity of incoming new data. Unlike other distributed OLAP systems, *vOLAP* does *not* use a special partitioning dimension that needs to be manually configured.

Each user session is attached to one of the server nodes. *vOLAP* guarantees *strong serialization* [17] of the insert and OLAP query operations within each session. For multiple user sessions that are attached to the same server (e.g. as a work group), *vOLAP* also guarantees strong serialization between those sessions. Note that, since OLAP queries can aggregate large portions of the database and thereby overlap many insert operations currently in progress, serialization is particularly challenging. Between multiple user sessions that are attached to different servers, *vOLAP* provides "best effort" serialization. In our experiments, the typical *freshness bounds* observed between servers were under 8 seconds and the worst-case freshness bound was just 15 seconds.

vOLAP is fully scalable and designed for an elastic cloud computing environment. Both, server and worker nodes can be added or removed as necessary to adapt to the current workload. Unlike other systems, no single node acts as a performance bottleneck or failure point for the entire system.

Experimental evaluation of our prototype system, using 18 workers for a database size of 1.5 billion items, shows that *vOLAP* is able to ingest new data items at a rate of over 600,000 items per second, and *vOLAP* can process streams of interspersed inserts and OLAP queries in real-time at approximately 200,000 queries per second. A distinguishing feature of *vOLAP* is that it exploits dimension hierarchies to improve performance. We have tested *vOLAP* on synthetic hierarchical data as well as TPC-DS test data which includes dimension hierarchies as shown in Fig. 4.8. The above mentioned performance evaluation of *vOLAP* is for data on such dimension hierarchies

and includes a wide range of queries ranging from small queries, to queries that need to aggregate several hundred million data items, up to queries that need to aggregate nearly the entire database.

5.2 *VelocityOLAP*

Consider a d-dimensional data warehouse D with N data items and d dimension hierarchies such as the one shown in Fig. 4.8. *vOLAP* processes multiple input streams of interspersed *insert* and OLAP *query* operations on D. Each OLAP query specifies, for each dimension, either a single value, range of values at any level of the respective dimension hierarchy, or a symbol "*" which matches the entire range for that dimension. The OLAP query result/output is the aggregate of the specified items in D. The *coverage* of an OLAP query is the percentage of D that needs to be aggregated.

5.2.1 Architecture Overview

vOLAP is a cloud-based architecture consisting of m *servers* $S_1, \ldots, S_m$ for handling client requests (inserts and queries); p *workers* $W_1, \ldots, W_p$ for storing data and performing operations requested by servers; a *Zookeeper* cluster for maintaining global system state [52]; and a *manager* background process for analysing global state and initiating load balancing operations. Workers and servers are multi-core machines which execute up to k parallel threads. As is typical for current high performance databases, all data is kept in main memory. With increasing database size and/or changing network topology, *vOLAP* reorganises the data to make the best use of currently available resources. Fig. 5.1 illustrates the distributed architecture, and a summary of the system parameters is given in Table 5.1.

Workers are used for storing data and processing OLAP operations. The global data set D is partitioned into data *subsets* $D_1, \ldots, D_n$. Each subset D_i has a *bounding*

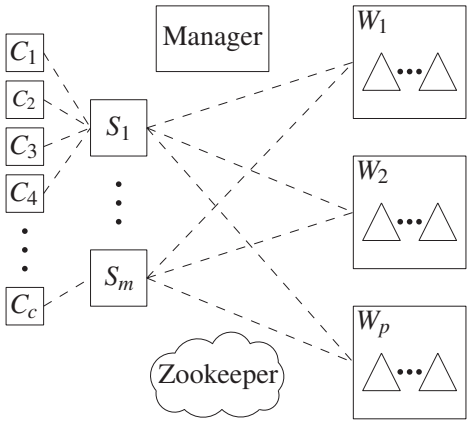

Figure 5.1: System Overview.

Parameter	Description
N	Total number of data items
n	Number of data subsets D_i
p	Number of workers W_j
m	Number of servers S_k
c	Number of clients C_l

Table 5.1: System Parameters

box B_i which is a spatial region containing D_i, represented by either a MBR (one box) or MDS (multiple boxes) [39]. Bounding boxes may overlap, and a single worker can store many subsets.

Servers receive OLAP operations from clients, determine the subsets relevant to each operation, and forward the operations to the worker(s) responsible for those subsets. Once the workers respond, the server reports the relevant results to the originating client.

Servers, workers, and the manager background process communicate using ZeroMQ [9], a high-performance asynchronous messaging library designed for scalable distributed applications. It provides socket-based APIs for inter-process and inter-thread communication. We found that a careful implementation of multi-threaded message handling was critical to achieving high performance. In our implementation,

every working thread has its own local socket to receive messages and a network socket to send messages without locking. A separate network socket receives incoming requests, which are dynamically load-balanced between the local thread sockets to ensure a high degree of parallelism. Fig. 5.2 illustrates the multi-threaded message handling framework.

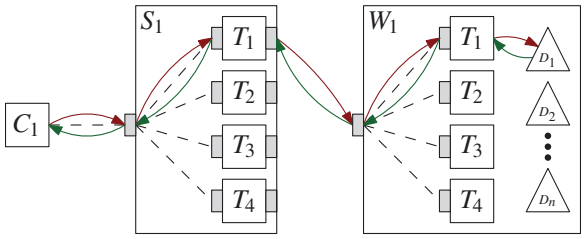

Figure 5.2: Multi-Threaded Message Handling.

5.2.2 System Image

The *global system state* is stored in Zookeeper [52], a fault tolerant distributed coordination service, and includes:

- A list of the current workers.
- A list of the current servers.
- The *global system image*: for each subset D_i: the size $|D_i|$, its bounding box B_i (MBR or MDS), and the address of the worker where it is located.
- Other measures such as the query workload for each subset D_i.

Each server S_k maintains a *local system image* I_k which is used to perform insertions and answer user queries. Given a query, server S_k computes from I_k the address of the target worker(s) that contain data covered by the query. Given an insertion, server S_k uses I_k to determine the subset D_i that the data should be inserted into and the corresponding worker address.

If the local system image is changed by an insert operation, the server updates the *global system image* in Zookeeper. However, the servers do not perform local system image synchronization mutually. Hence, the local system image maintained by each server may be outdated. In our experiments, servers were configured to update Zookeeper (if necessary) every 3 seconds. Servers monitor changes via Zookeeper's *watch* facility, where Zookeeper informs the servers of a change when it occurs, avoiding wasteful polling. Workers update subset statistics in Zookeeper periodically as well. This information is used by the manager background process to plan load balancing operations between workers in real-time.

The use of local system images is central to guaranteeing *strong serialization* of inserts and OLAP query operations within user sessions and between multiple user sessions that are attached to the same server. The global system image and local system image update process provides "best effort" serialization between multiple user sessions that are attached to different servers. In our experiments, the typical *freshness bounds* observed between multiple user sessions on *different* servers were under 8 seconds and the worst-case freshness bound was 15 seconds.

For each OLAP query operation, a server processor computes from its *local system image* I_k the addresses of the target workers that store data items relevant for the given query. Given an insertion, the local image I_k determines the address of the subset S_i that the data should be inserted into. When an insert into S_i leads to a change of its bounding box B_i, then B_i in I_k is updated accordingly. Also, for each insert into S_i, the corresponding $|S_i|$ in I_k is incremented and local changes are communicated to Zookeeper. The detailed geometric algorithms for performing OLAP query operations using system images, as well as the distributed protocol for handling insertion and query operations, are given in Section 5.3.

5.2.3 Data Representation

Each subset D_i is stored in an in-memory data structure designed for one multi-core machine. The data structure provides the following functionality:

- Processing of a stream of OLAP operations on D_i.
- Guaranteed *strong serialization* of OLAP operations on D_i: the result of each query contains exactly all inserts issued prior in the input stream, and none of those issued later.
- An operation SplitQuery(D_i, B_i) returning a hyperplane h that partitions D_i into D_i^1 and D_i^2 with bounding boxes B_i^1 and B_i^2, respectively, such that D_i^1 and D_i^2 are of approximately equal size. Note that this operation does not perform an actual split.
- An operation Split(D_i, B_i, h) returning $(D_i^1, B_i^1, D_i^2, B_i^2)$ where D_i is partitioned into D_i^1 and D_i^2 with bounding boxes B_i^1 and B_i^2, respectively, such that D_i^1 and D_i^2 are spatially separated by hyperplane h. Note that the split operation does not interrupt the query stream and maintains strong serialization of operations on D_i.
- An operation SerializeSubset() which returns a flat binary blob b containing the data in D_i (suitable for network transmission), and a corresponding operation DeserializeSubset(b) which builds the data structure from such a blob.

The SplitQuery, Split, SerializeSubset, and DeserializeSubset operations support load-balancing and data migration. A subset D_i stored on W_s (the *source worker*) can be migrated to another worker W_d (the *destination worker*) if, for example, W_s is running out of memory or W_d is a new worker allocated for spreading the load.

D_i can also be split in two if D_i is too big or the load balancer requires smaller chunks of data. Each worker W_k stores a *mapping table* T_j to handle operations while a split is in progress. If a D_i is split into D_i^1 and D_i^2, then T_j stores an entry with key D_i and value pointing to the two data structures for D_i^1 and D_i^2. The split and migration protocols are discussed in Section 5.4.

vOLAP currently includes three data structures for processing OLAP queries on subsets: a simple array for benchmarking purposes and two variants of the PDC tree [38]. These variants share the same multi-threaded PDC tree implementation but use Minimum Describing Subsets (MDS) and Minimum Bounding Rectangles (MBR) as keys, respectively. The PDC-MDS tree uses a new cache-efficient MDS implementation and improved split heuristics designed to perform better (compared to [38]) as trees get very large, while the PDC-MBR tree uses a simpler MBR bounding box representation for keys that is particularly fast for insert operations. Which PDC tree variant yields the best performance depends on the user's application and workload; we investigate these trade-offs experimentally in Section 5.6.

5.3 Algorithms

This section illustrate the detailed geometric algorithms for performing OLAP insert and query operations using system images as well as the distributed protocols for handling insertions and query operations.

5.3.1 OLAP Insertion Algorithms

When the server receives an insertion, there are three possible cases as shown in Figure 5.3. The server decides which box to insert the new element in, and sends the appropriate messages as described in Algorithm 1. If the insert changes a bounding box B_i, an update message is sent to Zookeeper.

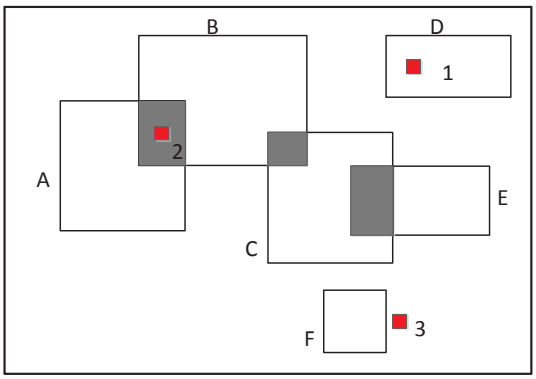

Figure 5.3: Insertion on server, 3 possible cases.

Data: I : insert(P) operation for a point P

Let B = the set of all boxes in the local image that contain P

`/* Find the target box` t `for this insert */`

Let $t = \emptyset$

if $|B| = 0$ **then** no boxes contain P [case 3]

  t = the smallest box in the local image

end

else if $|B| = 1$ **then** exactly one box contains P [case 1]

  $t = B[0]$

end

else multiple boxes contain P [case 2]

  t = the box that causes minimal enlargement when adding P

  Enlarge the size of t in the local image

  Send an asynchronous update of the box size to Zookeeper

end

`/* Send message to destination worker */`

if t *is migrating* **then**

  Send I to the source worker and destination worker of t

end

Algorithm 1: Server-Insert

Data: I : insert(P) operation for a point P

Data: B_i : Box where I should be inserted

Let DS_i be the data structure associated with B_i

if DS_i *has been split* **then**

    Use the routing table to find the correct box B_i'

    Insert I in DS_i'

    if B_i' *is expanded because of the insertion of* P **then**

        Expand the adjacent sub-box of B_i'

    end

end

else

    Insert I in DS_i

end

Algorithm 2: Worker-Insert

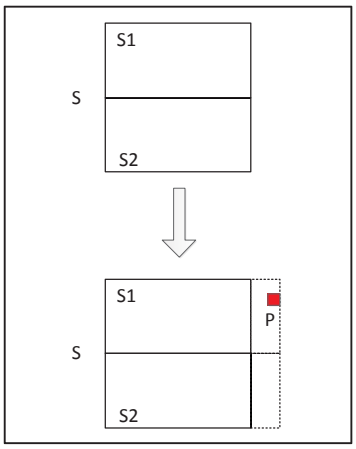

Figure 5.4: Box expand on server.

The messages forwarded by the server during an insert are handled by workers using Algorithm 2. If DS_i can not be found because it was split, the routing table is used to find the correct box B_i' for insertion. Otherwise, insert P in DS_i directly. For

more details see Algorithm 2.

Note the case where the inserted point q leads to a growth of box B_i (to B_i'') on server SP_k. In that case, B_i'' has to be sent to the worker together with q. If DS_i has been split, then not only the sub box containing q but also the adjacent sub-box have to be expanded to reflect B_i''. In Figure 5.4, box S has been split into S_1 and S_2. The inserted point P leads to a growth of box S_1 on worker. In this case, the adjacent sub-box S_2 has to be expanded to reflect S_1.

5.3.2 OLAP Query Algorithms

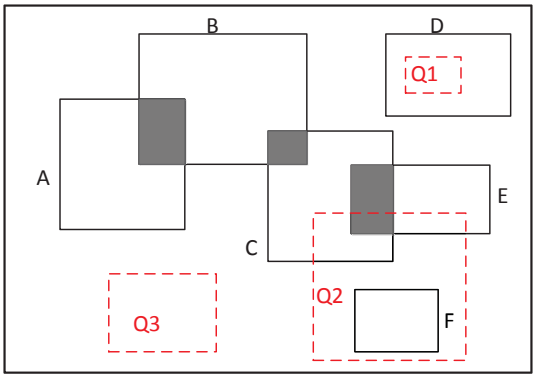

Figure 5.5: Range query.

Data: Query rectangle Q

S searches its local image:

if *find one box intersecting with* Q **then**

    Send query request to the corresponding worker of the intersected box

end

else

    if *find many boxes intersecting with* Q **then**

        Send query request to the corresponding worker of each box

    end

    else

        Return empty result

    end

end

Algorithm 3: Server-Search

Data: Query rectangle Q, Box B_i

if *data box* DS_i *has been split* **then**

Use the routing table to find the descendants of DS_i, DS, that intersected with Q

for ***each*** ds *in* DS **do**

Run query on ds

Save query result

end

Send the query result back to server

end

else

Run query on DS_i

Send query result back to server

end

Algorithm 4: Worker-Search

As described in Algorithm 3, server SP_k receives a range query(Query Rectangle Q) operation. Server SP_k searches its local image to check if any box intersecting with Q. Three cases may occur. If Q is intersected with exactly one box, the range query will be forwarded to the corresponding worker containing the intersected box. If Q is intersected with many boxes, then the range query request will be sent the corresponding worker of each intersected box. If no box intersected with Q, then server SP_k will return empty result to client. Figure 5.5 shows an example of the three possible cases.

Worker WP_k receives a range query(Query Rectangle Q) operation on box B_i. If data box DB_i can not been found because of splitting, use the routing table to find all descendants of DS_i. For every descendant of DS_i intersected with Q, the algorithm run query on the intersected descendant. Otherwise, run query on box B_i directly.

See Algorithm 4 for more details.

5.4 Load Balancing

Effective load balancing is crucial for the scalability of distributed systems. When the workload of the system is unevenly partitioned amongst its resources, some portion goes underutilized while the remainder struggles to pick up the slack. This has a negative impact on throughput, response time, and stability which tends to further compound itself as the system scales up in size. However, the load balancing operations themselves can also incur significant costs due to the overhead of moving potentially large subsets over the network. High performance requires a load balancing scheme which offers a good trade-off between load balancing overhead and effectiveness.

A separate background process called the *manager* initiates load balancing operations in *vOLAP*. The manager periodically analyses the system state stored in Zookeeper and determines appropriate performance-enhancing load balancing operations. The optimization algorithms used in the manager are described in Section 5.5. The manager then initiates these operations, coordinating the necessary actions between workers and servers, and waits for the operation to complete before re-analyzing the system state. For example, the manager identifies subsets that have grown too large and require splitting, or workers that are overloaded or running out of memory. Based on this analysis, it sends messages to workers instructing them to perform splits and/or migrations via the data structure operations discussed above. Note that the manager is a background process that only *initiates* operations which are executed by the workers and servers, and is therefore not a bottleneck for query performance. The manager process can reside anywhere in the system (and even migrate if necessary).

5.4.1 Migration Protocol

A key to achieving high performance in *vOLAP* was the design of migration protocols that allow the system to move subsets transparently from one worker to another, while the system continues to service both inserts and OLAP query requests. In addition to being integral to achieving high performance, dynamic load balancing also permits *vOLAP* to add, remove, or replace, workers dynamically in order to maintain system performance in the face of changing OLAP loads and data sizes.

The *vOLAP* migration protocol is designed to allow the system to continue to service requests normally while subsets are migrating. The main idea is to keep the subset stored on the source worker serving requests while the subset is migrating. Once the destination worker receives the subset from the source worker, the destination takes over handling operations for that subset and the old subset is removed from the source worker. During the migration process, operations are sent to both the source and destination workers in order to maintain the guaranteed strong serialization of OLAP operations. The migration process consists of 8 stages (see Figure 5.6 for an illustraion):

1. `MigratePrepDest`(D_i, W_s): To begin a migration, the manager notifies the destination worker that it will be receiving a new subset D_i. The destination worker begins queuing up any insertions for D_i that may arrive during the migration.

2. `MigratePrepSrv`(D_i, W_s, W_d): Once the destination worker is ready, the manager notifies all servers that subset D_i is about to migrate from W_s to W_d. The servers begin sending any insertions for D_i to both W_s and W_d.

3. `MigrateBegin`(D_i, W_d): Once all servers are aware of the pending migration, the manager instructs the source worker to send subset D_i to W_d.

4. `MigrateData`(D_i, W_d, `data`): The source worker serializes subset D_i into a binary blob `data` and sends it to W_d.

5. `MigrateCompletion`(D_i): When the destination worker receives subset D_i, it adds it to its collection of data structures, applies all pending queued inserts, and notifies the manager that it has received the data.

6. `MigrateDone`(D_i): When the migration is complete, the manager notifies all servers, which cease sending messages for D_i to the source worker.

7. `DeleteTree`(D_i): Now that D_i is no longer considered to reside at the source worker by servers, the manager instructs the source worker to delete it.

8. Finally, the manager updates Zookeeper to reflect the new location of D_i, and the migration is complete.

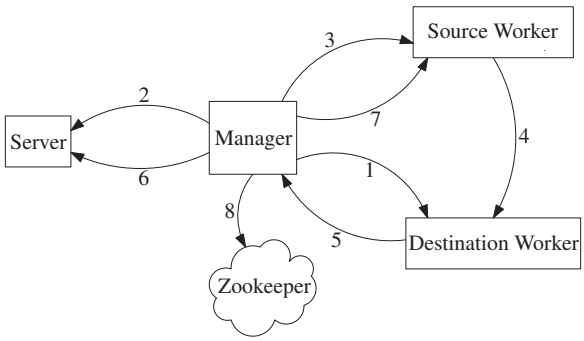

Figure 5.6: Migration process.

5.4.2 Splitting Protocol

In order to minimise the variance in subset size and maintain subsets that are not too large to be effectively migrated it is necessary to be able to split subsets. When a subset becomes too large, *vOLAP* splits the subset into two subsets of approximately equal size.

The splitting process consist of 4 stages each associated with a message delivery as illustrated in Fig. 5.7 and described below:

1. `SplitRequest`(D_i): To initiate a subset split, the manager sends a split request to the worker that hosts the subset. The worker begins the split process and records the

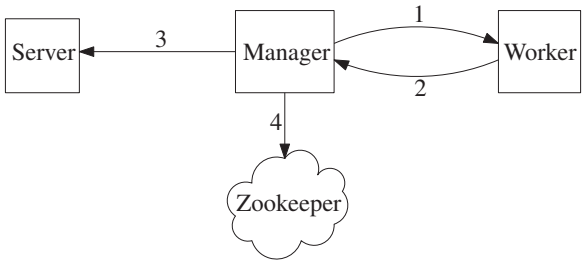

Figure 5.7: Split process.

split in its mapping table.

2. `SplitComplete`(D_i, D'_i, D''_i): Once the subset split is done, the worker notifies the manager that D_i is split into D'_i and D''_i.

3. `SplitUpdate`(D_i): Once the manager receives the `SplitComplete` acknowledgement, the manager notifies all servers that D_i is split into D'_i and D''_i. The servers begin sending insertions and OLAP queries directly to D'_i and D''_i.

4. `UpdateImage`(D_i, D'_i, D''_i): Finally, the manager updates Zookeeper to remove the old set D_i from Zookeeper and add the two new generated sets D'_i and D''_i into Zookeeper.

5.5 Optimizer

The *vOLAP* optimizer is a subcomponent of the manager process. The manager periodically requests a load balancing operation from the optimizer. The optimizer analyses the system state stored in Zookeeper and returns an appropriate load balancing operation. The manager then carries out this operation, coordinating any necessary actions between workers, then waits for its local system image to be updated before requesting another operation. The manager and optimizer may be executed on any machine, though reserving a separate physical machine allows computationally expensive load balancing decisions to be made without affecting system performance. With this configuration, system performance is only affected when load balancing

operations are actually being performed.

Once the optimization infrastructure is in place, there are many system parameters related to system performance that we may want to optimize. For example, a system may optimize: 1. Balance of data size per processor; 2. Balance of query load per processor; 3. Placement of replicated data; 4. Distribution quality of data; Note that there are potentially complex relevance between these different optimization goals.

In *vOLAP* we currently only perform the simplest, but most important of these optimizations. We perform a simple minimization to the variance of data size stored on each worker. Note that this is a critical optimization in a in-memory distributed system because a worker with limited memory size could run out of memory easily if data is distributed unbalanced.

5.5.1 Optimization Algorithm

The primary goal of the optimization algorithm is to minimize the average difference in size between workers without performing unnecessarily expensive operations. The algorithm proceeds in two phases (Migration and splitting optimization) which will result in either a migration, a split, or no operation in the event that all optimization are too costly. As migration is the most direct method for load balancing, the first phase (Migration optimization) of the algorithm is to find a suitable migration operation consisting of a source worker, a destination worker, and a subset belonging to the source to be migrated to the destination. For each possible source, destination, and subset belonging to the source, the algorithm computes a difference, called the *improvement*, in size between the source and destination before and after the hypothetical migration to determine the most beneficial migration. Once this migration is selected, it is then tested to see whether or not the operation is too expensive for the improvement it provides. For the operation to be worth the expense, the improvement must be at least greater than the migration threshold T_m.

If the *improvement* exceeded the migration threshold, the migration is performed and the algorithm terminates. Detailed description of migration optimization is described in Algorithm 5. We give the following notations which will be used in Algorithm 5:

1. $D = D_1, D_2, ..., D_n$: D contains all subsets $D_1, D_2, ..., D_n$ in *vOLAP*.
2. $W = W_1, W_2, ..., W_p$: W contains all workers $W_1, W_2, ..., W_p$ in *vOLAP*.
3. $|D_i|$, $1 \leq i \leq n$: the number of data items in D_i.
4. $|W_j|$, $1 \leq j \leq p$: the number of subsets in W_j.
5. $LOAD(W_j)$, $1 \leq j \leq p$, the number of data items stored in W_j.
6. *AVG*: the average load of each worker in the system.
7. $VAR(W_j)$: the load variance of W_j, $1 \leq j \leq p$.

If it is not within the migration threshold the algorthm enters its second phase and attempts to find a suitable split. By performing a split, the algorithm attempts to create a good migration for the next time the algorithm is called. As such, the process for finding a split is nearly identical to that of finding a migration. To find the best split, we calculate the best migration as previous but assume that all subsets are half their actual size size, that is, their size after the split is performed. By doing so, we find the best migration we can produce by performing a split. We also weigh the improvement against the expense in much the same way, only a heavier split threshold is required to account for the additional cost of the split. As before, if the improvement is within this threshold, we perform a split on the subset specified in out best migration, otherwise we are unable to find a reasonable operation and the algorithm terminates.

5.6 Experimental Evaluation

We evaluated the performance of *vOLAP* with respect to the following parameters: size of the database (N), number of workers (p), number of servers (m), *workload*

Input : $\emptyset$
Output : Source worker W_s, target subset d, destination worker W_d
Let source worker $W_s = \emptyset$;
Let destination worker $W_d = \emptyset$;
Let target subset $d = \emptyset$;
Let maximum improvement $MaxImprovement = 0$;
/* Find the source worker W_s, target subset d and destination worker W_d for migration */

Calculate the average load for each worker in the system:

$$AVG = \frac{\sum_{i=1}^{n} |D_i|}{p}$$

Calculate the load variance of workers:

$$VAR = \sum_{j=1}^{p} (LOAD(W_j) - AVG)^2$$

for *each worker* $W_i (1 \leq i \leq p) \in (W_1, W_2, ...W_p)$ **do**
  for *each subset* $D_k (1 \leq k \leq |W_s|) \in W_i$ **do**
    for *each worker* $W_j (1 \leq j \leq p, j \neq i) \in (W_1, W_2, ...W_p)$ **do**
      $LOAD(W_j) = LOAD(W_j) + |D_k|$;
      $LOAD(W_i) = LOAD(W_i)$ - $|D_k|$;
      Calculate $NEWVAR$, the new load variance of workers

$$NEWVAR = \sum_{j=1}^{p} (LOAD(W_j) - AVG)^2$$

      if *(*VAR - $NEWVAR$*)* $\geq T_m$ *and (*VAR - $NEWVAR$*)* $\geq$ $MaxImprovement$ **then**
        $W_s = W_i$;
        $d = D_k$;
        $W_d = W_j$;
        $MaxImprovement = VAR$ - $NEWVAR$
      end
    end
  end
end

Algorithm 5: Migration Optimization

mix (percentage of inserts in the operation stream; e.g. workload mix 25% is 25% inserts and 75% OLAP queries), and *query coverage* (percentage of the database that needs to be aggregated for an OLAP query). We used two different data sets. "TPC-DS" is a decision support data set from the Transaction Processing Council [8], with $N = 400$ million data items in $d = 8$ dimensions with dimension hierarchies as shown in Fig. 4.8. In order to test even larger data sets with $N \geq 1$ billion data items, we created a synthetic data set "Zipf" with Zipfian distribution, skew = 0.50, and $d = 8$. This dataset has hierarchical, randomly generated IDs, where each dimension hierarchy has 4 levels, each with a Zipfian distribution. The compute nodes used in the experimental evaluation were Intel Core i7-3770 CPUs with 4 cores (8 parallel threads) and 32 GiB of RAM, with Ubuntu 12.04.4 / Linux 3.2.0. We used ZeroMQ 4.0.1 and Zookeeper 3.4.6.

5.6.1 High-Velocity Data Ingestion

Bulk loading is a way to load data into the system in large chunks, and it could be used for loading data items into the system before any query operations started. When loading a great deal of data into the system at once, inserting one data item at a time may be inefficient. Instead, we choose to group insert requests in blocks and send them to workers. Comparing to send many small insertion messages to the same worker, grouping them in a block and sending to the worker use just one message, can significantly reduce the message passing overhead and improve the throughput of the system. And this bulk insert operation has no impact on the serialization of the system and just introduces a very small potential increase in freshness bound.

Fig. 5.8 shows the performance of *vOLAP* for high velocity data ingestion (insert only data stream) up to a database size (N) of 1 billion data items, using Zipf data. We use $p = 16$ workers and either 1 or 2 servers, and show performance with or without real-time load balancing. With 2 servers and real-time load balancing, *vOLAP* is able

to ingest 600,000 data items per second.

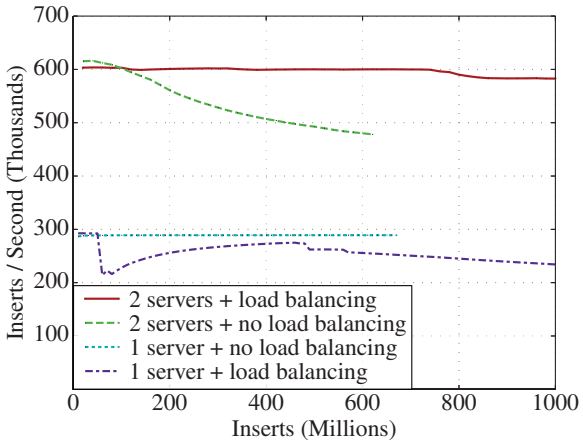

Figure 5.8: Data ingestion performance as database size N increases. Zipf data, $p = 16$.

Fig. 5.8 shows the impact of multiple servers. The second server essentially doubles the data ingestion rate. This shows the importance of being able to scale not only the number of workers but also the number of servers. As discussed in Section 2.1.1, systems such as SAP HANA [40] supports only one central server, thereby creating a performance bottleneck.

The importance of real-time load balancing for an in-memory distributed system is also apparent in Fig. 5.8. Note that the curves for the ingestion rates without load balancing stop at just over 600 million inserts. This is because without load balancing, one of the workers became overloaded. With real-time load balancing, however, the system is capable of managing the entire billion element data set.

Bulk loading is efficient for insert only data streams. This batching technique could also be used for insert and query mixed streams with minor changes. For insert only data streams, a group of messages is sent to the target worker once the group has received enough message or the time threshold is exceeded. When comes to mixed data streams, the grouped insert requests will be blow out as a block once it receives a query message. This is designed to guarantee strong session serialization

while minimize the query latency. If it is a mixed input stream with high insertion percentage, the throughput of the system could be improved a lot. If it is a mixed input stream with high query percentage, the process will be working like normal queries without blocking and grouping messages. Thus, the batching technique should provide some performance enhancement depending on the nature of the data. and it should provide some performance enhancement depending on the nature of the data.

5.6.2 Real-Time Load Balancing

Fig. 5.9 shows more details about the impact of real-time load balancing for the same experiments as in Fig. 5.8. For increasing database size (x-axis) we show the difference (shaded areas) between average and maximum data size on a worker (left y-axis of Fig. 5.9) with or without load balancing. We also show the number of splits and migrations performed for load balancing (right y-axis of Fig. 5.9).

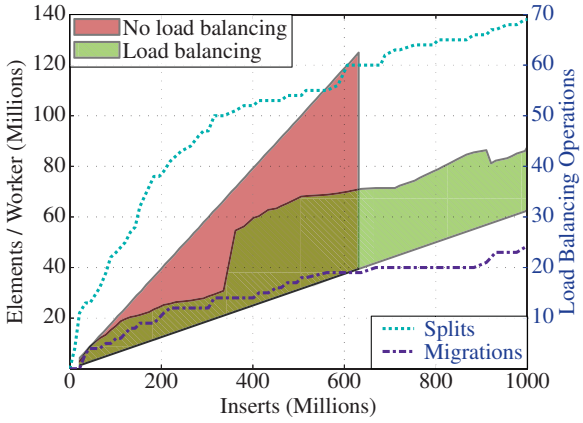

Figure 5.9: Average and maximum data size per worker as database size N increases; with and without load balancing. Zipf data, $m = 2$, $p = 16$.

The impact of load balancing on the maximum worker size is significant, reducing the maximum size by approximately 50% at around $N = 600$ million.

Load balancing can also have an impact on insertion throughput because there is a cost to performing many splits and migrations, even if they result in a more balanced data structure. As can be seen in Fig. 5.8, with one server load balancing

reduces insertion throughput by around 10% due to the cost of performing splits and migrations. However, with two servers, the benefits of load balancing outweighed the cost, and data ingestion performance actually increased with load balancing enabled by 9% on average.

5.6.3 Query Performance for TPC-DS Data

The experiments shown in Fig. 5.10 and Fig. 5.11 use more realistic (but smaller) OLAP data generated by the TPC-DS benchmark. Here, a *vOLAP* instance with $p = 16$ workers and $m = 4$ servers was first bulk-loaded to $N = 400$ million data items. Then, four input streams of interspersed insert and OLAP query operations were sent to the four servers. We measured the performance of *vOLAP* for various workload mixes (percentage of insert operations) and query coverages (percentage of the database aggregated by an OLAP query). Fig. 5.10 and Fig. 5.11 show the performance of *vOLAP* using the PDC-MBR tree and PDC-MDS tree data structures, respectively.

Workload mix has a significant impact on throughput because each insert operation on a mutli-threaded PDC tree may trigger re-balancing operations on a data structure that is concurrently being used to answer queries. Coverage has a significant impact on performance largely because it impacts the number of different PDC trees and workers that must be searched in answering an OLAP query. Fig. 5.10 and Fig. 5.11 show both the impact of workload mix and complexity of OLAP queries (i.e. query coverage).

Note that the choice of the ideal data structure depends on the application and its expected range of workload mix and query coverage. The PDC-MDS tree requires more work for inserts but is better for complex, high coverage OLAP queries because it makes better use of the dimension hierarchies. The PDC-MBR tree performs inserts faster but is slower on high coverage OLAP queries. For workloads with a higher

percentage of inserts and OLAP queries with a smaller coverage, the PDC-MBR tree performs better. For workloads with a lower percentage of inserts and OLAP queries with a larger coverage, the PDC-MDS tree performs better. Overall, with the right choice of data structure, *vOLAP* maintains an average of between 150,000 and 200,000 operations per second for the range of workloads (percentage of inserts; query coverages) shown in Fig. 5.10 and Fig. 5.11.

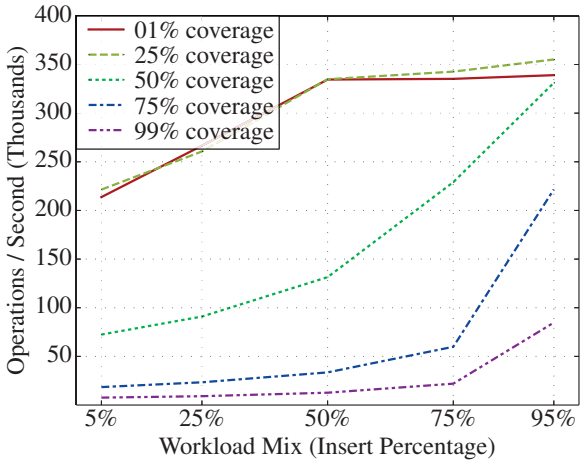

Figure 5.10: Performance for various workload mixes and query coverages (TPC-DS, PDC-MBR tree, $N = 400$ million, $p = 16$, $m = 4$).

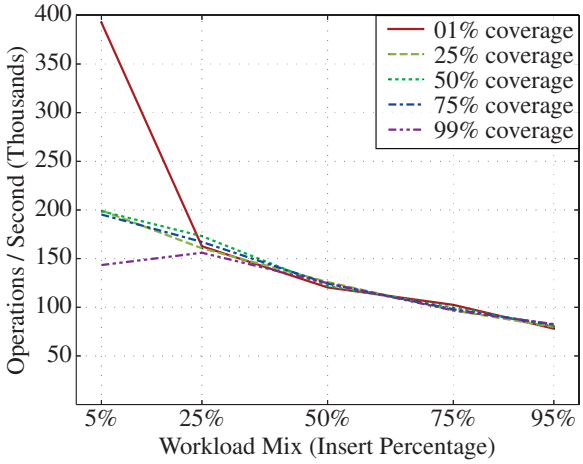

Figure 5.11: Performance for various workload mixes and query coverages (TPC-DS, PDC-MDS tree, $N = 400$ million, $p = 16$, $m = 4$).

The impact of the number of servers on system performance is shown in Fig. 5.12.

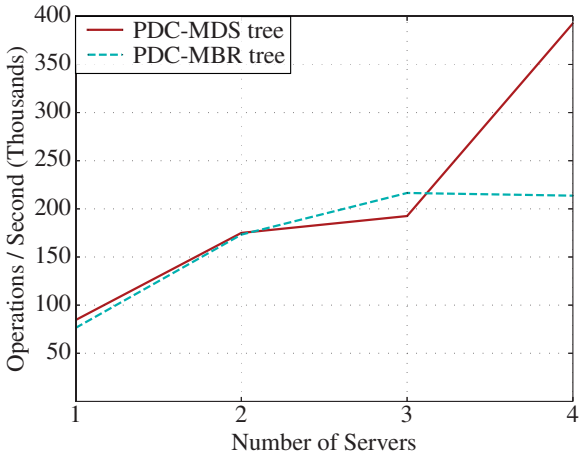

Figure 5.12: Query-heavy workload performance with increasing number of servers m. TPC-DS data, $p = 16$, workload mix = 5% inserts, average of all query coverages (5% ... 95%).

Similar results for Zipf data were observed in Fig. 5.8. For TPC-DS data and a fixed number of $p = 16$ workers, Fig. 5.12 shows the performance of *vOLAP* using the PDC-MBR tree and PDC-MDS tree data structures, respectively, when the number of servers (m) increases from 1 to 4. In both cases, performance increases essentially linear with m, subject to some noise. As discussed earlier, it shows that being able to scale all systems components, including m, is of critical importance. In Fig. 5.12, the increase for the PDC-MBR tree is less than the increase for the PDC-MDS tree because for TPC-DS data with multi-level dimension hierarchies, the PDC-MBR tree is less efficient and puts more load on the worker nodes, making them the limiting factor for performance.

5.6.4 System Scale-Up

Fig. 5.13 and Fig. 5.14 show the performance for various workloads as the system size increases, using TPC-DS data. Here, we increase the number p of workers as the the database size N increases, at a fixed ratio $\frac{N}{p} \approx 25$ million. This is to demonstrate the elastic capabilities of *vOLAP* in a cloud environment. For each system size with

p workers and $N \approx p \times 25$ million data items (up to a maximum of 16 workers and 400 million data items), we tested *vOLAP* with insert/query streams with different combinations of workload mix (5%, 25%, 50%, 75%, 95%) and query coverage (5%, 25%, 50%, 75%, 95%). Each data point in Fig. 5.13 and Fig. 5.14 is the average performance for the five different query coverages. The experiments were repeated for the two different data structures, PDC-MBR tree (Fig. 5.13) and PDC-MDS tree (Fig. 5.14).

The main observation from Fig. 5.13 and Fig. 5.14 is that *vOLAP* scales well in an elastic cloud environment. As the database increases, and processing resources (workers) are added at a fixed ratio, *vOLAP* maintains its performance over the entire range of database sizes. The curves in Fig. 5.13 and Fig. 5.14 are essentially flat, subject to some noise. Compared to Fig. 5.14, there is more noise in Fig. 5.13 with respect to the order of curves for different workload mix. This is again a reflection of the PDC-MDS tree data structure being better adapted to dimension hierarchies which leads to a more predictable behaviour for the TPC-DS data used here.

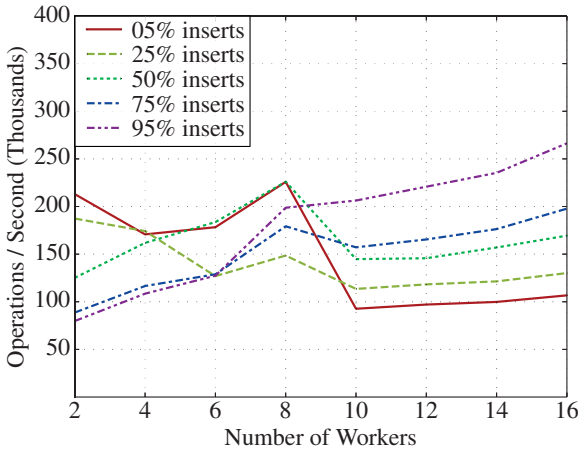

Figure 5.13: Performance for various workload mixes with increasing system size. Database size N and number of workers p both increasing. TPC-DS data, PDC-MBR tree, $\frac{N}{p} \approx 25$ million, $m = 4$, average of all query coverages (5% ... 95%).

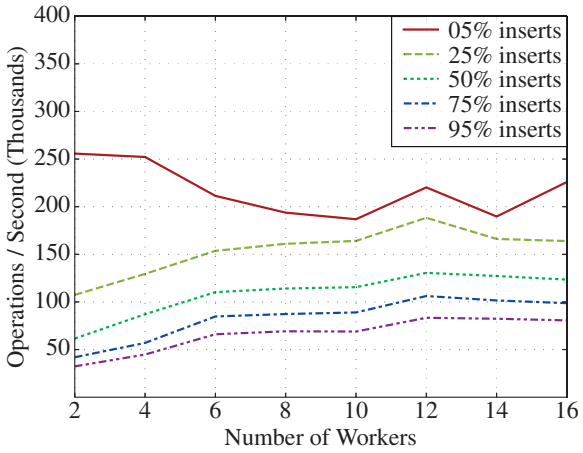

Figure 5.14: Performance for various workload mixes with increasing system size. Database size N and number of workers p both increasing. TPC-DS data, PDC-MDS tree, $\frac{N}{p} \approx 25$ million, $m = 4$, average of all query coverages (5% ... 95%).

5.6.5 Large Scale Experiment

As a final experiment, we tested *vOLAP* on a large database with $N = 1.5$ billion data items from the Zipf data set, using $p = 18$ worker instances and $m = 2$ server instances. At around $N = 1.5$ billion, *vOLAP* still showed an ingest performance of ∼600,000 items per second (essentially the same performance as in Fig. 5.8). We then tested *vOLAP* with insert/query streams with 50% workload mix (equal mix of insert and OLAP query operations) and 50% query coverage (each query aggregates approximately half the database). The measured average performance of *vOLAP* was approximately 200,000 queries per second.

Chapter 6

Conclusion

In this thesis, we introduced two new real-time cloud-based OLAP systems *CR-OLAP* and *vOLAP*. *CR-OLAP* is a *C*loud based *R*eal-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 an 16-fold increase in database size and number of processors, including an 16-fold increase in the average amount of data aggregated by each OLAP query, without noticeable performance impact for the user.

vOLAP is a fully scalable, cloud-based real-time OLAP system for high velocity data. *vOLAP* is also built based on the PDC-tree but supports multiple coordinating servers, query serialization and load balancing. An experimental evaluation of our *vOLAP* prototype, using 18 worker instances for a database size of 1.5 billion items, shows that it is able to ingest new data items at a rate of over 600,000 items per second, and can process streams of interspersed inserts and OLAP queries in real-time at a rate of approximately 200,000 queries per second.

Future work: The next phase of our research includes the study of the following issues:

1. Design and implement a better hat data structure that runs faster and is able to handle aggregation summaries in the hat. The hat used in *vOLAP* currently is a simple array-based implementation. Future research will try more efficient data

structures like PDC-tree and R-tree.

2. Data replication and fault-tolerance. The design of *vOLAP* also anticipates the introduction of data replication, copying part of the global data structures that are in high demand in order to achieve high performance and avoid system bottlenecks. Zookeeper, the migration protocols and the manager provide a basis on what to implement data replication in *vOLAP*, however, due to time limitations, this function was not implemented in the current *vOLAP* prototype. Data replication always comes with better fault-tolerance. In *vOLAP*, Zookeeper plays a key role for fault-tolerance. Zookeeper monitors the status of each processor. Single node failure can be tracked by Zookeeper. The global system image is stored in Zookeeper. Even if all servers died, the system is recoverable. Zookeeper also maintains the location information of each subset. If subsets are fully replicated and distributed, then Zookeeper knows the location of each subset's replicas. Thus, even if a worker dies, a new worker can be allocated to replace the old one by copying all subsets from other workers.

3. Expand optimization algorithms. As mentioned in Section 5.5, we only implemented the balance of data size per processor. Future research will cover the balance of query load per processor, placement of replicated data and distribution quality of data;

4. Batching of query operations. The current system prototype supports bulk insertions. Batching query operations may further improve the system performance.

5. Integrate with full OLAP query optimization engines, such as OLAP4j [6].

Bibliography

[1] Amazon EC2. http://en.wikipedia.org/wiki/Amazon_Elastic_Compute_Cloud.

[2] Cloud computing. http://en.wikipedia.org/wiki/Cloud_computing.

[3] Introduction to ZeroMQ. http://nichol.as/zeromq-an-introduction.

[4] Mainframe Computers. http://en.wikipedia.org/wiki/Mainframe_computer.

[5] Message Queues. http://en.wikipedia.org/wiki/Message_Queue.

[6] OLAP4J: An open Java API for OLAP. http://www.olap4j.org/.

[7] Riak NoSQL Database. http://docs.basho.com/.

[8] Transaction processing performance council, TPC-DS (decision support) benchmark. http://www.tpc.org.

[9] ZeroMQ socket library as a concurrency framework. http://www.zeromq.org/.

[10] ZooKeeper: A Distributed Coordination Service for Distributed Applications. http://zookeeper.apache.org/doc/trunk/zookeeperOver.html.

[11] A. Berkheimer A. Sherman, P. A. Lisiecki and J. Wein. ACMS: The Akamai configuration management system. In *NSDI*, 2005.

[12] Daniel J Abadi. Data management in the cloud: Limitations and opportunities. *IEEE Data Eng. Bull.*, 32(1):3–12, 2009.

[13] Divyakant Agrawal, Sudipto Das, and Amr El Abbadi. Data management in the cloud: challenges and opportunities. *Synthesis Lectures on Data Management*, 4(6):1–138, 2012.

[14] J Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: the definitive guide*. O'Reilly Media, Inc., 2010.

[15] R BAYER and E MCCREIGHT. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1:173–189, 1972.

[16] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. *The R*-tree: an efficient and robust access method for points and rectangles*, volume 19. ACM, 1990.

[17] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[18] Eric A Brewer. Towards robust distributed systems. In *PODC*, page 7, 2000.

[19] Martin C Brown. *Getting Started with Couchbase Server*. O'Reilly Media, Inc., 2012.

[20] R. Bruckner, Beate List, and Josef Schiefer. Striving towards near real-time data integration for data warehouses. *DaWaK*, LNCS 2454:173–182, 2002.

[21] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 335–350, 2006.

[22] Yu Cao, Chun Chen, Fei Guo, Dawei Jiang, Yuting Lin, Beng Chin Ooi, Hoang Tam Vo, Sai Wu, and Quanqing Xu. Es 2: A cloud data storage system for supporting both OLTP and OLAP. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 291–302. IEEE, 2011.

[23] Rick Cattell. Scalable SQL and NoSQL data stores. *ACM SIGMOD Record*, 39(4):12–27, 2011.

[24] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[25] Chun Chen, Gang Chen, Dawei Jiang, Beng Chin Ooi, Hoang Tam Vo, Sai Wu, and Quanqing Xu. Providing scalable database services on the cloud. In *Web Information Systems Engineering–WISE 2010*, pages 1–19. Springer, 2010.

[26] Min Chen, Shiwen Mao, and Yunhao Liu. Big data: A survey. *Mobile Networks and Applications*, pages 1–39, 2014.

[27] Ying Chen, Frank Dehne, Todd Eavis, and Andrew Rau-Chaplin. PnP: sequential, external memory, and parallel iceberg cube computation. *Distributed and Parallel Databases*, 23(2):99–126, January 2008.

[28] Kristina Chodorow. *MongoDB: the definitive guide*. O'Reilly Media, Inc., 2013.

[29] Eugene Ciurana. Google app engine. *Developing with Google App Engine*, pages 1–10, 2009.

[30] Amazon Elastic Compute Cloud. Amazon web services. *Retrieved November*, 9:2011, 2011.

[31] Mark Colan. Service-oriented architecture expands the vision of web services, part 1. *IBM DeveloperWorks, April*, 2004.

[32] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.

[33] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[34] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[35] F. Dehne, Q. Kong, A. Rau-Chaplin, H. Zaboli, and R. Zhou. Scalable Real-Time OLAP On Cloud Architectures.

[36] Frank Dehne, T Eavis, and S Hambrusch. Parallelizing the data cube. *Distributed and Parallel Databases*, 11:181–201, 2002.

[37] Frank Dehne, Quan Kong, Andrew Rau-Chaplin, Hamidreza Zaboli, and Rebecca Zhou. A distributed tree data structure for real-time OLAP on cloud architectures. In *Big Data, 2013 IEEE International Conference on*, pages 499–505. IEEE, 2013.

[38] Frank Dehne and Hamdireza 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)*, pages 588–594, 2012.

[39] M. Ester, J. Kohlhammer, and H.-P. Kriegel. The DC-tree: a fully dynamic index structure for data warehouses. *16th International Conference on Data Engineering (ICDE)*, pages 379–388, 2000.

[40] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. Sap hana database: data management for modern business applications. *ACM Sigmod Record*, 40(4):45–51, 2012.

[41] A Feinberg. Project voldemort: Reliable distributed storage. In *Proceedings of the 10th IEEE International Conference on Data Engineering*, 2011.

[42] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[43] John Gantz and David Reinsel. Extracting value from chaos. *IDC iView*, pages 1–12, 2011.

[44] Lars George. *HBase: the definitive guide*. O'Reilly Media, Inc., 2011.

[45] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

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

[47] Z. Guo-Liang, C. Hong, LI Cui-Ping, W. Shan, and Z. Tao. Parallel Data Cube Computation on Graphic Processing Units. *Chinese Journal of Computers*, 33(10):1788–1798, 2010.

[48] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. *SIGMOD Rec.*, 14(2):47–57, June 1984.

[49] Hyuck Han, Young Choon Lee, Seungmi Choi, Heon Y Yeom, and Albert Y Zomaya. Cloud-aware processing of mapreduce-based OLAP applications. In *Proceedings of the Eleventh Australasian Symposium on Parallel and Distributed Computing-Volume 140*, pages 31–38. Australian Computer Society, Inc., 2013.

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

[51] Mikio Hirabayashi. Tokyo cabinet: a modern implementation of DBM, 2010.

[52] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, volume 8, pages 11–11, 2010.

[53] Javid Jamae and Peter Johnson. *JBoss in action: configuring the JBoss application server*. Manning Publications Co., 2009.

[54] Dong Jin, Tatsuo Tsuji, and Ken Higuchi. An Incremental Maintenance Scheme of Data Cubes and Its Evaluation. *DASFAA*, LNCS 4947:36–48, 2008.

[55] Ankur Khetrapal and Vinay Ganesh. Hbase and hypertable for large scale distributed storage systems. *Dept. of Computer Science, Purdue University*, 2006.

[56] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[57] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[58] Peter Mell and Tim Grance. The NIST definition of cloud computing. *National Institute of Standards and Technology*, 53(6):50, 2009.

[59] Krishna Nadiminti, Marcos Dias De Assunção, and Rajkumar Buyya. Distributed systems and recent innovations: Challenges and benefits. *InfoNet Magazine*, 16(3):1–5, 2006.

[60] Raymond T. Ng, Alan Wagner, and Yu Yin. Iceberg-cube computation with PC clusters. *ACM SIGMOD*, 30(2):25–36, June 2001.

[61] The OLAP Report. http://www.olapreport.com.

[62] Douglas F Parkhill. Challenge of the computer utility. 1966.

[63] Timothy Peterson and Jim Pinkelman. *Microsoft OLAP unleashed*. Sams, 1999.

[64] Hasso Plattner and Alexander Zeier. *In-Memeory Data Management*. Springer Verlag, 2011.

[65] Denis Raphaely, Maitreyee Chaliha, Neerja Bhatt, Charles Hall, and James Wilson. Oracle streams advanced queuing user's guide, 11g release 2 (11.2) e11013-04.

[66] Salvatore Sanfilippo and Pieter Noordhuis. Redis, 2010.

[67] Ricardo Jorge Santos and Jorge Bernardino. Optimizing data warehouse loading procedures for enabling useful-time data warehousing. *IDEAS*, pages 292–299, 2009.

[68] RJ Santos and Jorge Bernardino. Real-time data warehouse loading methodology. *IDEAS*, pages 49–58, 2008.

[69] N. Schiper and S. Toueg. A robust and lightweight stable leader election service for dynamic systems. In *DSN*, 2008.

[70] Michael Schrader, Dan Vlamis, Mike Nader, Chris Claterbos, Dave Collins, Mitch Campbell, and Floyd Conrad. *Oracle Essbase & Oracle OLAP*. McGraw-Hill, Inc., 2009.

[71] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Scalaris: reliable transactional p2p key/value store. In *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, pages 41–48. ACM, 2008.

[72] Edward Sciore. SimpleDB: a simple java-based multiuser syst for teaching database internals. In *ACM SIGCSE Bulletin*, volume 39, pages 561–565. ACM, 2007.

[73] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. The R+ Tree: A dynamic index for multi-dimensional objects. 1987.

[74] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[75] Timothy Sixtus. Method and system for secure online transaction processing, May 11 1999. US Patent 5,903,721.

[76] Bruce Snyder, Dejan Bosnanac, and Rob Davies. *ActiveMQ in action*. Manning, 2011.

[77] Celery Team. Celery-the distributed task queue, 2011.

[78] Christian Vecchiola, Xingchen Chu, and Rajkumar Buyya. Aneka: a software platform for .NET-based cloud computing. *High Speed and Large Scale Scientific Computing*, pages 267–295, 2009.

[79] Alvaro Videla and Jason JW Williams. *RabbitMQ in action*. Manning, 2012.

[80] Steve Vinoski. Advanced message queuing protocol. *IEEE Internet Computing*, 10(6):87–89, 2006.

[81] Gerhard Weikum and Gottfried Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Elsevier, 2001.

[82] David A White and Ramesh Jain. Similarity indexing: Algorithms and performance. In *storage and retrieval for image and video databases (SPIE)*, pages 62–73, 1996.

[83] Jinguo You, Jianqing Xi, Pingjian Zhang, and Hu Chen. A Parallel Algorithm for Closed Cube Computation. *IEEE/ACIS International Conference on Computer and Information Science*, pages 95–99, May 2008.

[84] Hamidreza Zaboli. *PARALLEL OLAP ON MULTI/MANY-CORE AND CLOUD PLATFORMS*. PhD thesis, CARLETON UNIVERSITY, 2013.

[85] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010.