LUCAS HENRIQUE MOREIRA SILVA

Advisor: Joubert de Castro Lima

COMPUTING DATA CUBES OVER GPU CLUSTERS

Ouro Preto

December 2018

Federal University of Ouro Preto
Institute of Exact Sciences and Biology
Undergraduate Program in Computer Science

COMPUTING DATA CUBES OVER GPU CLUSTERS

Monograph presented to the Undergraduate Program in Computer Science of the Federal University of Ouro Preto in partial fulfillment of the requirements for the degree of Bachelor in Computer Science.

LUCAS HENRIQUE MOREIRA SILVA

Ouro Preto
December 2018

S381c Silva, Lucas Henrique.
Computing data cubes over GPU clusters [manuscrito] / Lucas Henrique Silva. - 2018.

48f.: il.: color; grafs; tabs.

Orientador: Prof. Dr. Joubert Lima.

Monografia (Graduação). Universidade Federal de Ouro Preto. Instituto de Ciências Exatas e Biológicas. Departamento de Computação.

1. Tecnologia OLAP. 2. Redes de computadores. I. Lima, Joubert. II. Universidade Federal de Ouro Preto. III. Titulo.

CDU: 004.72

Catalogação: ficha.sisbin@ufop.edu.br

FEDERAL UNIVERSITY OF OURO PRETO

CERTIFICATE OF APPROVAL

Computing Data Cubes Over GPU Clusters

LUCAS HENRIQUE MOREIRA SILVA

Monograph defended and approved by the examination board composed by:

Ouro Preto, December 2018

Resumo

O cubo de dados é um operador relacional fundamental para sistemas de suporte à tomada de decisão, dessa forma útil para a análise de Big Data. O problema apresentando nesse trabalho é: como reduzir os tempos de resposta de consultas multidimensionais complexas? Tal problema se torna ainda mais agravado se atualizações recorrentes nos dados de entrada acontecem e se existe um grande volume de dados de alta dimensionalidade a ser analisado. A hipótese deste trabalho é que uso de clusters de dispositivos CPU-GPU acelerará consultas em cubos de dados holísticos de alta dimensão que são constantemente atualizados. A solução alternativa proposta neste trabalho, chamada de JCL-GPU-Cubing, particiona a base de dados em múltiplas representações de cubos parciais sem introduzir redundância de dados. Tais cubos parciais são usados para executar consultas em CPU ou CPU-GPU de maneira eficiente. As avaliações experimentais preliminares demonstraram que a versão baseada em clusters de CPU escala bem quando ambos os dados de entrada e o tamanho do cluster aumentam.

Palavras-chave: GPU, OLAP, data cube, distributed computing, parallel computing, big data

Abstract

The data cube is a fundamental relational operator for decision support systems, thus very important for analytics. Unfortunately, a full data cube with all of its tuples has exponential complexity in terms of runtime and memory consumption as the dimensions increase linearly, so algorithms to reduce query response times continue under development. The problem stated in this work is: how can we reduce complex multidimensional queries response times from high dimensional data cubes? The problem is aggravated if recurrent updates occur and if there is a huge volume of high dimensional data to be managed. The hypothesis of this work is that clusters of CPU-GPU devices can speedup queries from high dimensional holistic data cubes that are updated constantly. The alternative solution presented in this work, named JCL-GPU-Cubing, partitions the base relation into multiple independent sub-cubes. These multiple sub-cubes represent a partial data cube to reduce the exponentiality and they are used to perform queries in CPU or in CPU-GPU computer architectures efficiently. The experimental evaluations using complex multidimensional queries demonstrated that the CPU cluster version scaled well when the base relation increased and the CPU-GPU version outperformed the CPU only version in certain scenarios.

keywords: GPU, OLAP, data cube, distributed computing, parallel computing, big data

I dedicate this work to my Father, Nereu, and Mother, Marilene, who taught me that one should not brag about his accomplishments, but let them shine through one's hard work

Acknowledgments

None of the accomplishments of this work would be possible without my advisor, Dr. Joubert C. Lima. I must thank him for the opportunity he gave me to learn from him and our peers throughout those years in his Lab. His assistance and guidance are immeasurable to my success not only with this work, but as professional.

Second, but not less important I thank my parents and my brother for their unconditional support and patience, without it I would never be able to reach anything. I thank all my family as well, who made a study in a good university possible, especially my aunts Eunice and Gilvane. Also, I thank all the friends I made it during this time, who helped me and shared happiness and hard times.

Most importantly, I would like to thank God for His Grace and Provision through this journey, who comforted my family despite the distance and gave me strength to carry on no matter what challenge I faced.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter we introduce and detail the the data cube operator. We present the goals we aim to achieve and the hypothesis of the work. The results are briefly presented, since discussions are detailed in Chapter 3.

Recent investigations in Big Data and Internet of Things (IoT) pointed out that the data cube operator and the OLAP technology are being redesigned for new challenges (Cuzzocrea et al. (2013); Cuzzocrea (2015a); Cuzzocrea et al. (2016)), so volume, data type variety and update speed continue to be hard problems and if they appeared together in a business domain we are faced with an open problem in data cube algorithms literature. In this work, we do not address an alternative computational solution for big data cubes, since data type variety and volume are not investigated, but the utilization of clusters of GPUs to perform complex multidimensional queries can be considered the research frontier in database.

Since the seminal paper of Gray et al. (1997), efficient data cube algorithms are reducing storage and runtime impacts to compute full or partial data cubes. There are cube solutions for different data types, including traditional or alphanumeric data cubes like this work (Sismanis et al. (2002); Xin et al. (2007); Lima and Hirata (2011)), spatial data cubes (Bimonte et al. (2006); Moreno et al. (2009); Bimonte et al. (2011)) , text data cubes (Lin et al. (2008); Zhang et al. (2009); Souza et al. (2017)), graph or network data cubes (Zhao et al. (2011); Wang et al. (2014); Benatallah et al. (2016)), RFID or stream data cubes (Gonzalez et al. (2006); Liu et al. (2011)) and image data cubes (Jin et al. (2010)). Besides many approaches for different data types, promising High Performance Computer (HPC) architectures that take the advantage of Graphic Processing Units (GPU), CPU clusters or cloud environments, demonstrated that the data cube problem is also hard to be efficiently partitioned and distributed (Wang et al. (2010); Kaczmarski (2011); Moreira and Lima (2012); Zhang et al. (2014); Cuzzocrea (2015b)).

The main goal of a data cube operator is to organize data into multiple hierarchies of dimensions and measures. Dimensions are composed of attributes, so they can have multiples attribute hierarchies. The *time* dimension can be composed of *year*, *month* and *day* attributes, for instance. These attributes can build the hierarchies h_1={year, day}, h_2={month, day},

h_3={year, month, day}, and many more. Besides alphanumeric dimensions, there are *text* dimensions with topic hierarchies, *spatial* dimensions with *resolution* hierarchies and many other hierarchy types in a data cube operator.

Hierarchization is a decision making active, where the information is classified into hierarchy levels, varying from detailed information to high abstraction levels of a decision making process. Drilling down an hierarchy level indicates that the analyst wants more detailed information. Roll-up is the opposite direction. There are other data cube operations, such as slice, dice and drill-through (Han et al. (2011)). In summary, this relational operator enables different alternatives to navigate through multiple hierarchy levels, simplifying business domain decisions.

Measures are how a data cube evaluate the attribute combinations. We can suppose a combination of attribute values in a tuple t = (*year*=2016; *month*=December; *studentName*=Robert; *grade*=6.9; *coefficient*=C). The attributes *grade* and *coefficient* are measures. *Grade*, for example, can represent an average of all grades in all courses of a school. The *coefficient* 'C' can also represent a collection of courses evaluations. Measures can be a numerical value with a statistical function SUM, AVG or an inverted index to text data cubes or the distance using roadways from a location L to a river or a dam in a spatial data cube.

A data cube has base tuples and aggregate tuples. Suppose an input relation R with three dimensions (A, B, and C) and the tuple, $t_1 = (a_1, b_1, c_1, m)$, where a_1, b_1, and c_1 are the attribute values for each dimensional attribute and m is a numerical value representing a measure value of t_1. In this example, each dimension has a unique attribute, so there is only one possible hierarchy. Given R, a full data cube has eight tuples representing all possible aggregations: t_1, $t_2 = (a_1, b_1, *, m)$, $t_3 = (a_1, *, c_1, m)$, $t_4 = (*, b_1, c_1, m)$, $t_5 = (a_1, *, *, m)$, $t_6 = (*, b_1, *, m)$, $t_7 = (*, *, c_1, m)$, and $t_8 = (*, *, *, m)$, where the asterisk (*) denotes a wildcard representing any value that a dimensional attribute can assume on the data cube. Generally speaking, a data cube computed from the relation R with three dimensions (A, B, and C), three attribute values (a_1, b_1, c_1) and cardinality $C_A = C_B = C_C = 1$, can have 8 or $(C_A + 1) \times (C_B + 1) \times (C_C + 1)$ tuples. Cardinality indicates the number of unique values that each dimension attribute (Ex. A, B, or C) can assume. In this example, t_1 is a base tuple and $t_2, t_3, t_4, t_5, t_6, t_7, t_8$ are aggregate tuples. The cardinalities C_A, C_B, C_C are from the three attributes of dimensions A, B and C, respectively.

If we consider relation $ABCD$ instead of relation ABC, and $C_A = C_B = C_C = C_D = 2$, there can be 16 $ABCD$ base tuples and 81 aggregate tuples that must be calculated in a full data cube. As we can see, the data cube operator has exponential complexity in terms of runtime and memory consumption as the dimensions increase linearly. High cardinality and huge volume of tuples in R turn the problem harder, so the Big Data requirements are demanding redesigns of data cube algorithms for indexing, querying, and updating.

The HPC literature in data cube is starting to adopt GPU cards to speedup query responses with low cost, since most recent cards bypassed 1k cores at 1GHz each, the memory capacity is increasing rapidly and current off-the-shelf PC motherboards can handle up to four cards. Unfortunately, some improvements achieved are restricted to data cubes with low dimensionality and some of them completely stored in GPU memory (Lauer et al. (2010); Wittmer et al. (2011); Wang and Zhou (2012)), which is not suitable for high number of tuples.

Other approaches partially store data cubes in GPU Riha et al. (2011); Malik et al. (2012), so CPU working-memory (RAM, for instance) is adopted in conjunction, but both CPU and GPU data structures are not designed for constant updates and there is always the overhead of multiple CPU-to-GPU data transfers. The approach of Wang et al. (2018) provides a GPU cluster system, based on Hadoop and MapReduce to accelarete OLAP workloads, but presents no hibryd solution for it's algorithms to run in GPU and CPU. Another limitation in the GPU data cube literature is that most of the solutions implement equality operators and few range operators in their queries, like between, greater than, less than, some, similar and others. This limitation occurred because range query operators increase the number of tuples substantially, as detailed by Ho et al. (1997); Silva et al. (2013, 2015). Finally, no related work innovates to support complex holistic measures, like MODE, RANK, and others, but they represent the biggest computational challenge, since they impose computational costs to be both calculated and, mainly, updated.

1.1 Goal

Given the limitations described above, this work presents the first data cube approach to index and query multidimensional data over clusters of multicore-CPUs and multiple GPU cards, named JCL-GPU-Cubing. It is designed for high dimensional data and it supports recurrent updates. Holistic measures, like MODE, RANK, VARIANCE, TOP-K and many more are also supported. The alternative solution JCL-GPU-Cubing is classified as RAM-only, therefore no external memory is considered. Instead, several private main memories are used to achieve high storage capacity.

The specific goals are:

1. The design and implementation of a CPU and a CPU-GPU version to index, update and query data cubes in multicore-CPU clusters. The CPU version is used as a baseline version and both versions adopted a third-party middleware for the distribution of tasks and data, their scheduling, their communication using several network protocols and so forth. We considered JCL (Almeida et al. (2018)) as the underling solution due to its simplicity in terms of development and deployment. It scales well, has a distributed

version of the Map Java Interface, which is very familiar to Java community, and finally it runs over small platforms, including Android devices (de Resende et al. (2017));

2. The experiments and evaluations of the CPU and GPU versions using traditional alphanumeric databases. Large high dimensional datasets and update tests are considered;

3. The design and implementation of a CPU-GPU version;

4. The comparative tests with a benchmark to corroborate with the scalability results obtained from CPU versus CPU-GPU comparisons.

1.2 Hypothesis

This work introduces a new HPC data cube approach with index, query and update algorithms, where the query algorithm uses GPU devices as accelerators, therefore it must be not only faster than its CPU version, but also faster than the benchmark. These results are unique in the data cube topic, thus in database computer science area, proving that the utilization of clusters of GPUs can scale OLAP technology. Big Data involves huge volume of data, a variety of data types and recurrent updates in data, so this approach reduces a bit more the gap of an OLAP tool to Big Data.

1.3 Work Organization

The rest of this work is organized as follows: Chapter 2 discusses the related work about GPU as accelerators in data cube literature; Chapter 3 details the architecture and design of JCL-GPU-Cubing solution; Chapter 4 presents the experimental results and evaluations; Chapter 5 concludes the work.

Chapter 2

Related Work

In this chapter, we describe the literature about data cubes over a single device with one or few GPUs. In all the works, the CPU memory system is unique and shared among the cores, so there is what is called a private memory system. A cluster is a group of devices that follows the private CPU memory system explained before, providing transparencies for users, like a distributed and shared memory system abstraction.

The related work was evaluated according to the following requirements obtained from the literature Cuzzocrea et al. (2016); Wittmer et al. (2011); Lauer et al. (2010); Wang and Zhou (2012); Wittmer et al. (2011); Malik et al. (2012); Riha et al. (2011); Silva et al. (2015):

1. Single or multiple GPU support (**SoMG**): is important since even off-the-shelf PC motherboards support up to four cards nowadays;

2. Multicore or cluster based deployment (**MCD**): is highlighted since distributed alternatives are quite uncommon in the literature, but distributed computing becomes the last resort when private memory systems reach their limit;

3. Large base relations support (**LBR**): is part of the Big Data concept, thus very important today, but very often the existing approaches limit the relation size to the available GPU memory, so only few gigabytes of data can be processed;

4. CPU-GPU hybrid support (**CGHS**): useful for large relations, enabling, for instance, data cube indexing in CPU and query in both CPU-GPU;

5. Support holistic measures (**HM**): imposes update and indexing challenges in data cube literature, so how to support this measure type is fundamental;

6. Support high dimensionality (**HD**): it is a hard problem since the data cube operator grows exponentially in space and time when dimensions increase linearly. Today this kind of high dimensional data is common in biological domains, but also in many social network and entertainment scenarios like films, music, news and so forth;

7. Update support (measures, dimensions and hierarchies) (**US**): is another key requirement of Big Data concept, but not implemented by most of existing OLAP products and data cube prototypes. Recurrent updates are very common, for instance, in the stock-market domain, for stream processing demands and many other niches;

8. Complex data types support (spatial, text, stream, graph and so forth) (**CDT**): represents a Big Data requirement, but unfortunately not attended until now by products or research prototypes. There are only specific solutions for specific data types, thus no integration is presented;

9. Range query operators support (between, some, greater, similar, contains, etc.) (**RQO**): enables filters not only on measures, but also on dimensions, becoming primordial in modern multidimensional analysis and Business Intelligence.

At the end of this section there is a comparative table, summarizing the benefits and drawbacks of each work according to the previously defined requirements.

The work of Lauer et al. (2010) introduced a data structure for the data cube stored in the GPU global memory, where the attributes from the tuples are allocated contiguously in that memory, but separated by each dimension. An array of measures associated to each tuple is also stored in GPU memory. A set of indexes can be calculated from the query in a way that each thread from the GPU can operate over independent sets of tuples, avoiding memory conflicts and thread serialization. To enable the use of multiple GPUs, the algorithm needs an extra preprocessing step in which it must partition the query data equally among all cards. On the CPU, each card has it's own "host thread" that waits for the result, submitting it to another thread that aggregates the final result in a parallel reduction operation. The authors evaluated their approach using just the SUM measure during the aggregation, hence it's not clear the support to holistic measures. The experimental evaluation using multiple GPUs on a single machine obtained linear query results. Compared to CPU only versions, the response time for some queries achieved up to 42 times faster using the GPU version. The approach adopts the number of tuples in a query result to indicate if the query should be processed in CPU or in GPU, but they did not present any hybrid solution that switches from GPU to CPU and vice-versa. The number of dimensions was constrained to seven and the number of tuples to tens of millions.

The approaches presented in Riha et al. (2011) and Malik et al. (2012) implemented a hybrid strategy for the query processing. A scheduler decides if a query will run only in CPU mode or if any GPU processing will be necessary. The experiments pointed out that there is a point where the cost of the query overwhelms the cost of data transfer between CPU and GPU, thus queries that demand fewer aggregations or lower processing should be executed in CPU to avoid the overhead, similar to the suggestions of Lauer et al. (2010). To perform the scheduling it is used the hardware details and a complexity estimation for the query to create a performance model. For queries processed in GPU the relation should be completely stored on its global memory and for such it is used a storage oriented by columns.

The work Kaczmarski (2011) presents a comparison between CPU and GPU data cube algorithms. The GPU alternative transfers all the data from CPU to the GPU global memory and then performs the cube creation. Similar to other works, this data transfer is the bottleneck of the whole solution, but even with a single data transfer the aggregation phase outperformed the CPU version, being 50% faster than it. The experiments used a low dimensional base relation with 5 dimensions and multiple GPUs per machine are considered to support large relations, but no cluster deployment is proposed. For multiple GPUs and many CPU cores, the approach must scan the input data once to create a parallel execution plan in which each GPU card accesses an independent intersection of the data cube. A parallel reduction phase aggregates the final results of a multidimensional query, similar to many hybrid CPU-GPU approaches, including the JCL-GPU-Cubing approach. The experimental evaluations used just the SUM measure, thus no holistic measure was investigated.

The approach Kaczmarski and Rudny (2011) presented a compact representation of a data cube and an algorithm based on the primitives of parallel scan and parallel reduction to perform queries on the GPU. The base relation is entirely stored in GPU memory and when the data cube is sparse the approach introduces data compression. The representation in GPU of the data cube is not very prone for updates, so the update of dimensions, measures, hierarchy levels or a simple new attribute value would imply the re-indexing of the data cube from scratch, which is impracticable in many online or real time domains, like financial trading, stream processing, social networks, logistic and so forth.

The work of Wang and Zhou (2012) used a linearization function responsible to map each tuple of the data cube to a position P of the GPU memory, where such function has the property of being reversible, that is, with P it is possible to retrieve the attribute values of each dimension. This work considers that the data is transferred to the GPU global memory when the system receives a query. Precisely, the query is interpreted in the CPU and then transferred to the GPU, which performs the filtering, data cube creation and aggregations. The design for storing tuples was not suitable for a dimensional or tuple increase since it demands even larger vectors to store the GPU memory positions. This work investigated range multidimensional queries, precisely the query operators "greater than" and "less than" applied into dimensional attributes.

The authors in Sitaridi and Ross (2012) reinforced memory accesses conflicts and thus synchronization issues in OLAP over GPU literature. To avoid memory conflicts, it is presented an algorithm that allocates specific regions of GPU memory for each thread and then reorder the query results according to those regions. If it is detected that a conflict may occur, regions are duplicated among the threads, enabling private operations without conflicts, but introducing consistency problems. The implemented operators (JOIN and GROUP BY, specifically) achieved a speedup of only 1.2 times when compared with a baseline version. When the memory conflicts are not significant, the memory footprint and processing overhead degraded the performance. The base relation must fit entirely in the GPU global memory, an important constraint. Besides that, the data type supported is limited to 4-byte integers.

The works Zhang et al. (2012) and Zhang et al. (2014) investigated spatial-temporal aggregations using GPU processors, presenting a case study about taxi rides. The generated data cube had 10 dimensions, including pick-up latitude and longitude (spatial) and pick-up time (temporal) . The authors presented algorithms and data structures to store those data types efficiently on both GPU and CPU memories, using, for example, a 4-byte representation instead of 66-byte for date-time dimensions. The parallel implementation for GPU achieved a speedup of up to 13x if compared to the CPU version.

In Riha et al. (2013), there is a query optimization solver for hybrid memory systems that adopts information beyond the query cost estimation and data availability, i.e., it uses the current workload of each processor and the memory architecture. Experiments demonstrated an accurate performance model to estimate both the processing time and the minimum response time of a query. The GPU data cube structure stores all the data in a one-dimensional array in the global memory, performing the filtering and aggregations over this array. The GPU approach can process multiple queries in parallel since its threads manipulate private memory blocks, thus avoiding thread serialization. The best hybrid CPU-GPU solution achieved a speedup of only 1.7 times while dealing with several queries in parallel if compared with the CPU only version. The base relation must fit in GPU memory and the approach runs in a single GPU card, so large relations are not suitable.

Another CPU-GPU query scheduler is proposed by Breß et al. (2013). The authors identified a limitation in query solvers based on query response times since a scenario where one of the processors outperforms the other for all queries in terms of processing time will saturate the best processor when the others end up unused. Such scenario could degrade the overall performance, but these solutions still report that the processors allocation is optimal. In order to attenuate the previously described limitations it was implemented several heuristics focused on optimizing the processor scheduling in terms of workload distribution to maximize the throughput across all processors. The first phase for the heuristic is determine the best device for each query operator accounting for response time only, then it must calculate a threshold where the chosen device can be sub-optimal, but improving the current throughput. Performance evaluations achieved a speedup of 1.6 times if compared against GPU versions without such query optimization heuristics.

In Breß (2014), it is presented a detailed cost estimation procedure to evaluate a multidimensional query cost, enabling CPU or GPU query allocations dynamically. The processing cost for each query and it's estimated performance in each type of processor (GPU and CPU, respectively) are calculated and used to schedule a query to the most suitable processor type.

The cost estimation procedure uses hardware details, such as thread organization and memory architecture to infer accurate results. Performance results against MontetDB Nes and Kersten (2012) demonstrated that the presented approach could be 1.8 times faster when its CPU-GPU designs are adopted in conjunction. The work reinforced the bottleneck caused by multiple CPU-GPU data transfers, but no alternatives using multiple GPUs in a single machine or in a cluster were detailed.

The work of Wang et al. (2018) was the only one found to present a system that integrates distributed computing and GPU-based acceleration to the OLAP context. The authors present a solution based on Hadoop HDFS and Map Reduce to perform the data distribution and aggregation calculation. A GPU-based Reducer algorithm is presented. To mitigate the elevated number of I/O operations imposed by Hadoop when mapping the data, there is a compression stage that preventively aggregates the base relation and creates an inverted index. The main idea of the algorithm is to create the complete data cube and in query time, filter the cuboids that match the predicates along with the aggregation function. All the operations are performed using the Divide and Conquer algorithm model through a series Map Reduce operations and the GPU is used to provide the speed up on the cube construction, query filtering and cuboid selection and aggregation computation. On the experiments conducted by the authors, they don't consider varying the node count on the cluster, experimenting only with different base sizes. When experimenting with the GPU-based distributed cube algorithm, the authors vary the base relation size and the time cost increases along linearly. On the overall process, the GPU algorithm is 2× faster than the CPU equivalent.

Nowadays the interest in implementing efficient alternatives to build multidimensional query results has been reduced significantly. Instead, efficient scheduling strategies to allocate queries in hybrid multi-core-CPU and GPU systems became the common OLAP research interest (Riha et al. (2013); Breß et al. (2013); Breß (2014)). The queries workload partition, how to avoid multiple CPU-GPU transfers, how updates work and so forth are user responsibilities, using the literature improvements, for instance. The recent works of Karnagel and Habich (2017); Appuswamy et al. (2017) reinforced this assumption. The JCL-GPU-Cubing approach innovates in different direction, i.e., we are interested in efficient algorithms to build multidimensional queries from huge data cubes over a cluster of CPUs-GPUs devices, therefore the decision to work cooperatively CPUs-GPUs or separately is not our focus because the queries investigated manipulate huge results and consume vast amount of processor's cycles, therefore CPUs and GPUs are always used together.

Table 2.1 presents a comparison of the related work described above, summarizing the implemented requirements by each paper and the requirements our work implements and will implement upon future work. Each requirement can be fulfilled in three levels (✓, ✓✓ and ✓✓✓), where ✓ indicates basic implementations, ✓✓ indicates fundamental ones and ✓✓✓ indicates advanced designs. The '–' wildcard indicates that no information was found about the ability of the approach in attending the requirement.

Table 2.1: Comparison of the related work using the previously defined requirements.

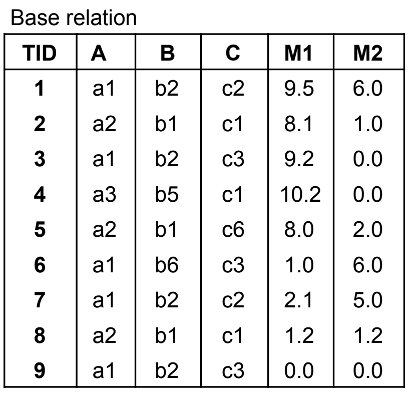
Related Work	**SoMG**	**MCD**	**LBR**	**CGHS**	**HM**	**HD**	**US**	**CDT**	**RQO**
JCL-GPU-Cubing	✓✓✓	✓✓✓	✓	✓✓✓	✓✓	✓✓	✓✓	–	✓✓✓
Lauer et al. (2010)	✓✓	✓	✓✓	–	–	–	–	–	–
Riha et al. (2011)	✓	✓	✓✓	✓✓✓	–	–	–	–	–
Kaczmarski (2011)	✓✓	✓	✓✓	–	–	–	–	–	–
Kaczmarski and Rudny (2011)	✓	✓	✓✓	–	–	–	–	✓	–
Wang and Zhou (2012)	✓	✓	✓✓	–	–	–	–	–	✓
Sitaridi and Ross (2012)	✓	✓	✓✓✓	–	–	–	–	–	–
Zhang et al. (2012)	✓	✓	✓✓	–	–	✓	–	✓✓	–
Malik et al. (2012)	✓	✓	✓✓	✓✓✓	–	–	–	–	–
Riha et al. (2013)	✓	✓	✓✓	✓✓✓	–	–	–	–	–
Breß et al. (2013)	✓	✓	–	✓✓✓	–	–	–	–	–
Zhang et al. (2014)	✓	✓	✓✓	✓✓✓	–	–	–	–	–
Breß (2014)	✓	✓	✓✓	–	–	✓	–	✓✓	–
Karnagel and Habich (2017)	✓	–	–	✓✓✓	–	–	–	–	–
Sato and Usami (2017)	✓	✓	✓✓	–	–	–	–	✓	–
Appuswamy et al. (2017)	✓	✓✓	–	✓✓✓	–	–	–	–	–
Wang et al. (2018)	✓✓	✓✓	✓✓	–	–	–	–	–	–

According to Table 2.1, the most attended requirements are: i) single GPU support (SoMG), ii) multicore deployment (MCD) and iii) medium size base relations (LBR). The most rare requirements in the literature are: v) holistic measures support (HM), vi) high dimensionality support (HD) and vii) update support (US). The remaining requirements (CGHS, CDT and RQO), although partially attended, are equally important. The big data requirements (volume, velocity and variety, for instance) are not addressed by the GPU OLAP literature, precisely single devices deployments are not sufficient when volume increases, the recurrent updates of a data cube crash its internal representation and algorithms in all related work, and finally no work creates a unified data cube representation with dimensions, measures and hierarchies for text, spatial, stream and other data types. In the next section, we present an alternative solution to attend some of requirements stated in the beginning of this section.

Chapter 3

Development

In this chapter, we present the main components of the JCL-GPU-Cubing approach. For the next sections, the indexing and query pipelines are detailed with basic examples to illustrate the main ideas, discussions about the design decisions and pseudo-codes of the implemented algorithms. The examples presented in this chapter consider the base relation of Figure 3.1 as the input for the algorithms. Columns A, B and C represent three dimensions and columns M1 and M2 the two measures. There is one primary-key or tuple identification per tuple, called "TID". The examples used in this chapter consider a cluster with four devices.



Base relation

TID	A	B	C	M1	M2
1	a1	b2	c2	9.5	6.0
2	a2	b1	c1	8.1	1.0
3	a1	b2	c3	9.2	0.0
4	a3	b5	c1	10.2	0.0
5	a2	b1	c6	8.0	2.0
6	a1	b6	c3	1.0	6.0
7	a1	b2	c2	2.1	5.0
8	a2	b1	c1	1.2	1.2
9	a1	b2	c3	0.0	0.0

Figure 3.1: The input base relation for the example.

3.1 Indexing

Given the base relation from Figure 3.1, it must be partitioned across the multiple devices in the cluster. The current strategy simply partitions the base relation into chunks with identical size in terms of tuples. These chunks are stored across the cluster devices, as seen in Figure 3.2.

Device 1

TID	A	B	C	M1	M2
1	a1	b2	c2	9.5	6.0
2	a2	b1	c1	8.1	1.0
3	a1	b2	c3	9.2	0.0

Device 2

TID	A	B	C	M1	M2
4	a3	b5	c1	10.2	0.0
5	a2	b1	c6	8.0	2.0
6	a1	b6	c3	1.0	6.0

Device 3

TID	A	B	C	M1	M2
7	a1	b2	c2	2.1	5.0
8	a2	b1	c1	1.2	1.2
9	a1	b2	c3	0.0	0.0

Figure 3.2: The input base relation partitioned over a cluster.

After the base relation is partitioned, the indexes are created. Those index compose what we named a "partial cube". Figure 3.3 illustrates the indexes created for the partition of the base relation stored in Device 1, but it is important to note that all devices of the cluster create multiple indexes in this step simultaneously. The Inverted Index of each tuple and consequently the Inverted Index of the complete base relation can be defined from an ordinary tuple t_1= $\{id_1, a_1, b_1, \ldots m_1, m_2, \ldots\}$, so it can be defined as it_1=$\{$ $a_1 : id_1$, $b_1 : id_1$, $c_1 : id_1$, $\ldots\}$, where id_1 represents the tuple identification or primary-key, $(a_1, b_1, c_1, \ldots)$ represents the tuple dimensional values and $(m_1, m_2, \ldots)$ its measure values. The second data source of a partial cube is named "measures" and it is responsible for storing the measure values and from which tuple they come from. Finally, the third data source named "tuples" represents the tuple storage without its measures. Both "measures" and "tuples" data sources represent the base relation partitioned horizontally over a cluster, i.e., it implements a tuple partition and not a column partition. These three data sources are sufficient for answering queries efficiently in parallel. After the "index" method call, several partial cube representations are created in the cluster, as Figure 3.3 illustrates.

Inverted Index

Dim. Value	TIDs
a1	[1, 3]
a2	[2]
b1	[2]
b2	[1, 3]
c1	[2]
c2	[1]
c3	[3]

Measure Index

TID	Measure Values
1	[9.5, 6.0]
2	[8.1, 1.0]
3	[9.2, 0.0]

Tuples

TID	A	B	C
1	a1	b2	c2
2	a2	b1	c1
3	a1	b2	c3

Figure 3.3: The Inverted Index, Measure Index and Tuples Index of Device 1.

The Index Construction is a multi-thread algorithm, thus concurrency is introduced inside each device to create the three indexes. The Inverted Index and Measure Indexes are used to speedup the values lookup during the query processing and the Tuples Index is used to maintain the structure of dimensions of each tuple in main memory to improve dimensional and measure filtering during the query processing. Those three structures are called "partial cubes", as Figure 3.4 illustrates. Basically, the partition and split operations run indefinitely, i.e., while there is a tuple to be consumed from a base relation. A sequential process reads

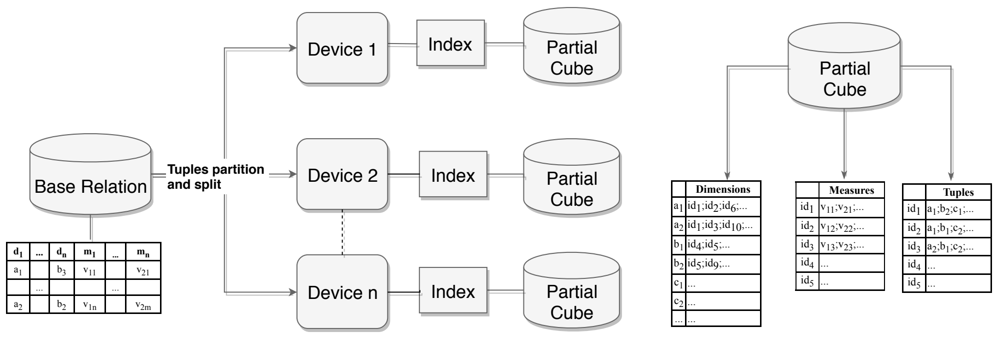


Figure 3.4: The indexing phase.

the input data repeatedly and sends chunks of tuples at a time to each device of the cluster. This is done once and can take hours or even days for the first load of a huge amount of data. This process is illustrated in Figure 3.4. It is important to understand that there is no order in the base relation to guarantee the data partition, consequently data redundancies (in terms of dimensional values, but tuples with different TIDs) may occur in the cluster, but they are eliminated during the query execution. Another important aspect of the indexing algorithm is that there is no synchronization barriers, so even a cluster composed of multi-core CPUs can be adopted without synchronization drawbacks during the partial data cube indexing.

Algorithm 1: The Indexing algorithm that runs in each device or in each device core of a cluster

```
Input: $R$ with a set of tuples $t$, where $t$ = ($TID$, $D_1$, $D_2$, ..., $D_n$, $M_1$, $M_2$, ..., $M_m$),
       where $n$ is the number of dimensions and $m$ is the number of measures of $R$;
Output: $DI$ with the dimensions Inverted Index of entries $e$, where
        $e$ = ($D_i\_v_j$, $TIDs$), $i = 1..n$ and $j = 1..|D_i|$ and $TIDs$ is a list of TIDs for all
        occurrences of value $v_i$ at the dimension $D_i$;
        $MI$ with the Measure Index of entries $e$, where $e$ = ($TID$, $M_1$, $M_2$, ..., $M_m$);
        $TI$ with the Tuple Index of entries $e$, where $e$ = ($TID$, $D_1$, $D_2$, ..., $D_n$);
1  receive $R$ from the cluster;
2  initialize DI and MI as empty mappings;
3  for each $t$ in $R$ do
4      id = t[TID];
5      for each dimension $D_i$ in $t$ do
6          DI[$D_i\_v_j$] = DI[$D_i\_v_j$] $\cup$ id;
7      end
8      for each measure $M_k$ in $t$ do
9          MI[id] = MI[id] $\cup$ $M_k$;
10     end
11     TI[id] = all dimension values from $t$;
12 end
```

The algorithm 1 reads tuple per tuple in the loop - line 3 - splitting them into dimensions and measures, those partitions being stored in two data structures, the inverted indexes of dimensions and measures. The first loop creates the Inverted Index - line 5 - and the second loop creates the Measure Index - line 8. At end the end of each iteration of the main loop, the Tuple Index receives a new entry of dimension values from each tuple - line 11.

3.2 Query

The following subsections describe the multiple pipelines from the JCL-GPU-Cubing query. First the base relation is filtered according to a query, then multiple sub-cubes are constructed over the cluster and finally the measure calculus is performed to answer the query.

3.2.1 Filtering

After the Indexing, the next step is to perform successive queries. For that, each query has its filtering step to select some dimensional and sometimes measure values according to user needs and before it generates the sub-cube that answer the query, composed of several aggregated measure values organized hierarchically. Using the same example presented in the last chapter, Figure 3.5 shows the query Q with one predicate per dimension, where the predicate C =? represents the INQUIRE operator on dimension C and it means all dimension values individually and the aggregated value "*" - ALL.

After the partition of the base relation (Figure 3.2), Device 1 and Device 3 had identical dimensional values in their partitions, which can occur in real scenarios. This situation highlights the need of the reduce phase of the query algorithm at the end of the query pipeline, but for now all devices filtered their tuples independently and generate a result-set of tuples that are locally unique and satisfy all query predicates.

Q: **A**=a1 AND **B**=b2 AND **C**=?; AVG **M1**

Device 1

Tuples			TIDs
a1	b2	c2	[1]
a1	b2	c3	[3]

Device 3

Tuples			TIDs
a1	b2	c2	[7]
a1	b2	c3	[9]

Figure 3.5: The query filtering and the result as a set of tuples in Device 1 and 3.

3.2.2 Sub-cube Construction

The next step of the query pipeline is the sub-cube construction, the combinatorial algorithm. Figure 3.6 illustrates the sub-cubes created from the valid tuples filtered in Section 3.2.1. The Sub Cube Construction algorithm runs on each device or each core of a cluster.

Sub Cube Device 1

Tuples			TIDs
a1	b2	c2	[1]
*	b2	c3	[1]
*	b2	c2	[3]
a1	b2	c3	[3]
a1	*	c2	[1]
a1	*	c3	[3]
*	*	c2	[1]
*	*	c3	[3]
a1	b2	*	[1, 3]
*	b2	*	[1, 3]
a1	*	*	[1, 3]
*	*	*	[1, 3]

Sub Cube Device 3

Tuples			TIDs
a1	b2	c2	[7]
*	b2	c3	[7]
*	b2	c2	[9]
a1	b2	c3	[9]
a1	*	c2	[7]
a1	*	c3	[9]
*	*	c2	[7]
*	*	c3	[9]
a1	b2	*	[7,9]
*	b2	*	[7,9]
a1	*	*	[7,9]
*	*	*	[7,9]

Figure 3.6: The sub cubes created for the filtered tuples from Devices 1 and 3.

After the filtering step, a set of valid tuples is obtained, i.e., a set of tuples that meet the filters restrictions applying only the “AND” logical operator. The “OR” and nested “AND/OR” combinations are planed for future implementations of the presented approach. With all valid tuples, it is possible to transfer all tuples to GPU to perform the “sub-cube construction” or leave them in CPU to perform the same operation. In summary, the CPU-GPU data transfers must take less time than “sub-cube construction” to compensate the GPU usage, so there are many research studies innovating in new query scheduling ideas to allocate them in CPU or in GPU Riha et al. (2011); Malik et al. (2012); Riha et al. (2013); Breß et al. (2013); Breß (2014); Karnagel and Habich (2017); Appuswamy et al. (2017). As mentioned before, this work assumes that CPU-GPU data transfers always compensate the benefit of thousand of cores.

3.2.2.1 CPU based sub-cube construction

The CPU version receives all valid tuples and insert the wildcard “ALL” or “*” in all non-aggregated attribute value. To illustrate that, consider the tuple with $TID = 1$ from the base relation illustrated in Figure 3.1, which has the following values: $\{a_1, b_2, c_2, 9.5, 6.0\}$. This tuple is a valid one and the “sub-cube construction” algorithm inserts “*” in all the three dimensions A, B and C, creating the following new tuples t_1=$\{*, b_2, c_2, 9.5, 6.0\}$, t_2=$\{a_1, *, c_2, 9.5, 6.0\}$, t_3=$\{a_1, b_2, *, 9.5, 6.0\}$, t_4=$\{*, *, c_2, 9.5, 6.0\}$, t_5=$\{*, b_2, *, 9.5, 6.0\}$, t_6=$\{a_1, *, *, 9.5, 6.0\}$, t_7=$\{*, *, *, 9.5, 6.0\}$. As we can see, it is a costly processing and storage task, where a three dimensional query with just one tuple as a result after filtering creates seven new tuples plus, summing up 2^3 tuples as the final multidimensional result. Figure 3.7 illustrates the CPU-based implementation of the query pipeline.

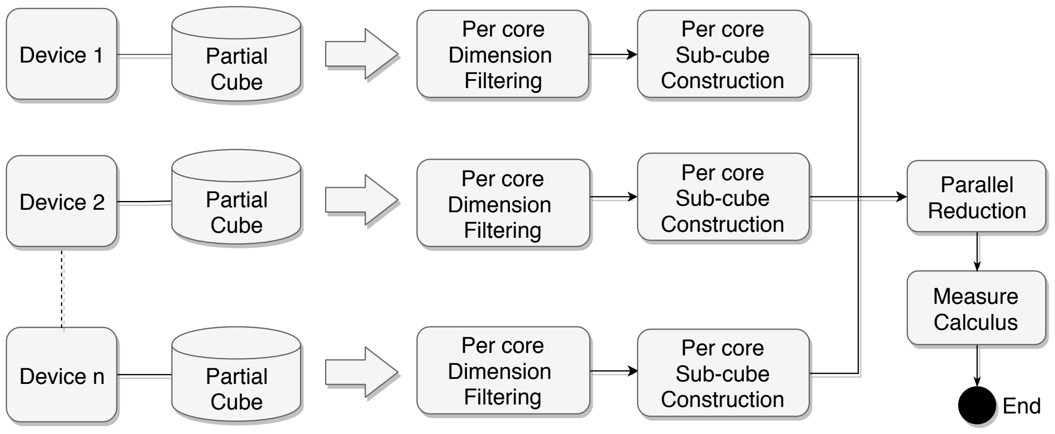

Device 1
Partial Cube
Per core Dimension Filtering
Per core Sub-cube Construction
Device 2
Partial Cube
Per core Dimension Filtering
Per core Sub-cube Construction
Parallel Reduction
Measure Calculus
Device n
Partial Cube
Per core Dimension Filtering
Per core Sub-cube Construction
End

Figure 3.7: The Query pipeline using the CPU version.

3.2.2.2 GPU based sub-cube construction

Due to the particularities of the GPU architecture, some steps of the query pipelines presented before had to be redesigned. The main difference, if compared with the CPU version, is that after filtering, each device must produce a unique set of tuples, so multi-core computer architectures must perform a synchronization barrier to produce a unique set of "TIDs" per device. This step is called "Result-set Reduction", shown in Figure 3.8 and it was implemented because the CPU to GPU data transfer tends to be highly costly, as described by many related works from Chapter 2, so instead of performing multiple data transfers with possibly redundant data, only a single data transfer per GPU card is performed to store all tuples in the GPU memory.

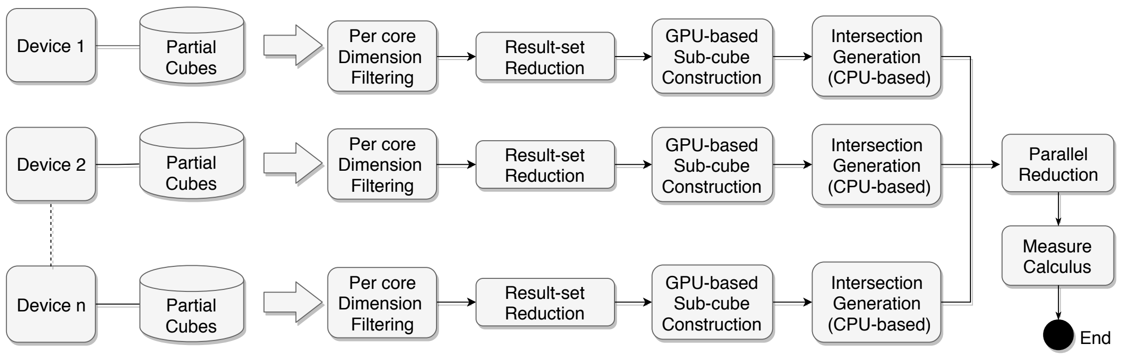

Device 1
Partial Cubes
Per core Dimension Filtering
Result-set Reduction
GPU-based Sub-cube Construction
Intersection Generation (CPU-based)
Device 2
Partial Cubes
Per core Dimension Filtering
Result-set Reduction
GPU-based Sub-cube Construction
Intersection Generation (CPU-based)
Parallel Reduction
Measure Calculus
Device n
Partial Cubes
Per core Dimension Filtering
Result-set Reduction
GPU-based Sub-cube Construction
Intersection Generation (CPU-based)
End

Figure 3.8: The Query pipeline using the GPU version.

After the "Result-set Reduction" step finishes, the "sub-cube construction" step starts in GPU and it is the same explained previously, but implemented over a particular memory organization that we must detail a bit more. The "sub-cube construction" inserts "ALL" or "*" value at different tuple dimension attribute, creating new aggregate tuples, this way in GPU we must know how much memory space will be required to store all aggregate tuples from all base tuples filtered. Figure 3.9 illustrates the GPU memory arrangement used in this work, where each tuple with n dimensions each produce exactly $2^n - 1$ new tuples with identical size, so it can preemptively allocated. An important consideration for this algorithm is that when the number of tuples surpass the number of threads available in GPU, the tuples are queued in GPU memory or even in CPU memory until there are available GPU threads, so sub-cubes can be constructed in batches with one or several batch transfers.

We are interested in high level of parallelism, so we decided to map each GPU thread to a single tuple not yet aggregated, the thinnest grain of parallelism, thus it generates all aggregate tuples from such a tuple, as Figure 3.9 illustrates. Identical aggregate tuples are generated during this GPU parallel operation, for instance the more aggregated tuple where all dimension attributes are equal “*”, so future parallel reductions are required to produce the final aggregated tuple, similar to previously explained CPU version of “sub-cube construction” algorithm. There are several other ways to map GPU threads to set of filtered tuples, so any coarse grained map strategy can be done and future investigations could indicate the best coarsity level in terms of tuples per GPU thread each multidimensional query should allocate.

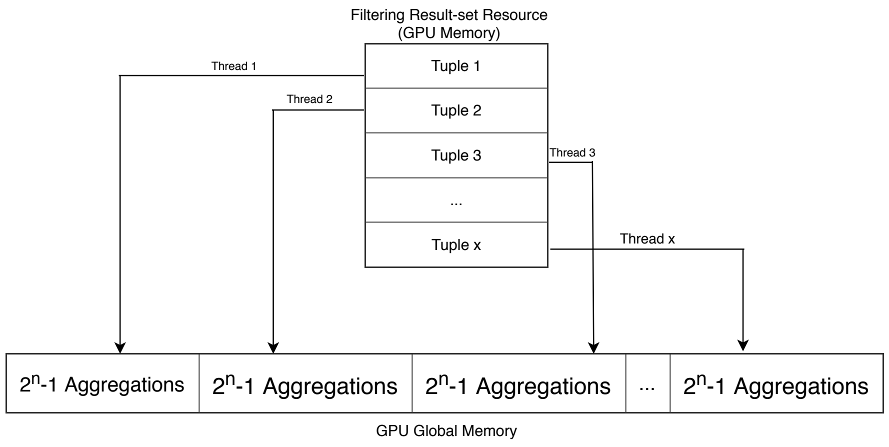


Figure 3.9: The GPU memory arrangement.

Before starts “Parallel Reduction” in CPU, the JCL-GPU-Cubing must associate the aggregate tuples to their “TIDs”. This pipeline step is also performed in CPU and done by “Intersection Generation” step in Figure 3.8. In the CPU resource there are several base tuples with their “TIDs”, this way it is possible to insert such “TIDs” on each aggregate tuple, that will come from the GPU, associated with the specific base tuple. This step is done sequentially, but it can be easily implemented in parallel. After the “Intersection Generation” step all tuples are ready to be reduced. The great advantage of this idea is that it avoids storing “TIDs” from aggregate tuples in GPU global memory and they represent most tuples in a data cube. Besides saving space, this decision allows the GPU algorithm to be simple and similar to the CPU version. Besides that, the final “Parallel Reduction” will operate over less data redundancies, since “Result-set Reduction” eliminated base tuple redundancies at a Device level, the Intersection Generation also eliminated redundancies of aggregate tuples and each device produces only a single resource, whereas the CPU version produces multiple sub-cubes per device.

Algorithm 2: The sub-cube construction algorithm for CPU or GPU devices

```
Input: $RS$ with the result-set of tuples $t$ that satisfy all query predicates, where $t$ has
       only the dimensions present in the query $q$;
Output: $C$ with the aggregation of the tuples from $RS$, where “*” is the wildcard to
        represent the ALL aggregation level on a given dimension $D_i$ of $t$;
1  receive $RS$ from the cluster;
2  if current device has an available GPU then
3      Aggregate all $RS$ from the current device to $RS^*$, removing all duplicated tuples;
4      $C = RS^*$;
5      $n$ = number of dimensions;
6      $O = 2^n$;
7      copy $C$, $n$ and $O$ to GPU memory;
8      allocate the list agg in the GPU memory to store the aggregations of all tuples;
9      for each $t$ in $C$, in the GPU and in parallel do
10         $gID$ = GPUThreadID;
11         generate all aggregations of $t$ for all dimensions;
12         store the aggregations in agg w.r.t. the offset $O + gID$;
13     end
14     copy agg back to CPU memory;
15     for each $t$ in $C$ do
16         originalTIDs = $t$[TIDs];
17         for each aggregation $a$ of $t$ in agg do
18             a[TIDs] = originalTIDs;
19             if $a$ has the same aggregations of any $t'$ already in $C$ then
20                 $t'$[TIDs] = $t'$[TIDs] $\cap$ $a$[TIDs];
21             end
22             add $t'$ to $C$;
23         end
24     end
25 end
26 else
27     C = RS;
28     for each dimension $D_i$ in $RS$ do
29         for each $t$ in $C$ do
30             create a new tuple $t'$;
31             $t^* = t$;
32             $t^*[D_i] = *$;
33             if $t^*$ has the same aggregations of any $t'$ in $C$ then
34                 $t'$[TIDs] = $t'$[TIDs] $\cap$ $t^*$[TIDs];
35             end
36             add $t'$ to $C$;
37         end
38     end
39 end
```

The sub-cube construction algorithm is detailed in the algorithm 2. Similar to the previous algorithms, this solution is executed distributed and in parallel over the cluster. The input is the result-set of tuples that were filtered on the previous step of the query pipeline. Those tuples from the result-set only have the dimensions that were considered by the query. Line 2 determines if the execution will occur on GPU or CPU.

If the device can run GPU code, the first step is to reduce all result-sets from the current device into a single resource, named RS^*, which is transferred to the GPU memory (line 3). When performing this reduction, the aforementioned data redundancies of base tuples are eliminated. As the number of new tuples that will be generated by each base tuple can by calculated, we can allocate the exact amount of GPU memory beforehand, as shown in line 8. The same calculation can be done to create an offset to be used to store the aggregations of each tuple in the GPU memory, as seen in the line 6. The loop in line 9 is actually executed using the GPU threads and it runs in parallel for each base tuple, so all aggregations are generated and stored in the GPU global memory, using the previously calculated offset. After creating the newly aggregate tuples, they are transferred back to the CPU main memory (line 14 and in CPU the intersections must be done to associate the base tuples' TIDs to these newly aggregate tuples. This is done on the loop 15, which uses the TIDs of the base tuples do produce the TIDs of the aggregate tuples.

If the device is not capable of running GPU code, the sub-cube is constructed using only the CPU and for that, the main loop, at line 26, applies the aggregation level "ALL" in each dimension of the result-set and generates the intersections (32). In contrast to the GPU procedure, the TIDs intersections can be generated as the aggregations are created, because the CPU version generates the aggregations sequentially, so at each iteration it can check if the aggregation already exists and merge the *TIDs* list of the aggregated tuple with the one that is already on the sub-cube.

3.2.3 Parallel reduction and Measure calculus

The next step of the query pipeline is responsible to produce a unique query response, this way multiple sub-cubes must be reduced to a single sub-cube, which is the final query response. Figure 3.10 illustrates the final sub-cube that contains all tuples from the two sub-cubes Devices 1 and 3. This final sub-cube is created and stored in a single device of the cluster.

Sub-Cube Device 1

Tuples			TIDs
a1	b2	c2	[1]
*	b2	c3	[1]
*	b2	c2	[3]
a1	b2	c3	[3]
a1	*	c2	[1]
a1	*	c3	[3]
*	*	c2	[1]
*	*	c3	[3]
a1	b2	*	[1, 3]
*	b2	*	[1, 3]
a1	*	*	[1, 3]
*	*	*	[1, 3]

Sub-Cube Device 3

Tuples			TIDs
a1	b2	c2	[7]
*	b2	c3	[7]
*	b2	c2	[9]
a1	b2	c3	[9]
a1	*	c2	[7]
a1	*	c3	[9]
*	*	c2	[7]
*	*	c3	[9]
a1	b2	*	[7,9]
*	b2	*	[7,9]
a1	*	*	[7,9]
*	*	*	[7,9]

Reduced Sub-Cube

Tuples			TIDs
a1	b2	c2	[1, 7]
*	b2	c3	[1, 7]
*	b2	c2	[3, 9]
a1	b2	c3	[3, 9]
a1	*	c2	[1, 7]
a1	*	c3	[3, 9]
*	*	c2	[1, 7]
*	*	c3	[3, 9]
a1	b2	*	[1, 3, 7, 9]
*	b2	*	[1, 3, 7, 9]
a1	*	*	[1, 3, 7, 9]
*	*	*	[1, 3, 7, 9]

Figure 3.10: The sub-cubes from devices 1 and 3 are reduced to a single sub-cube with all tuples from both devices and it's respective intersections.

The final step to answer the query illustrated in Figure 3.11 is to translate the TIDs to the respective measure values and apply the measure function (AVG, for instance) on all measure values. This can be done by consulting the Measure Indexes from the devices where the tuples of the Final Reduced Sub-cube came from. Figure 3.11 illustrates the final sub-cube representation with the measure calculus executed for each base and aggregate tuple, thus the query response is ready to be presented to the user.

Measure Index Device 1

TID	Measure Values
1	[9.5, 6.0]
2	[8.1, 1.0]
3	[9.2, 0.0]

Measure Index Device 3

TID	Measure Values
7	[9.5, 6.0]
8	[8.1, 1.0]
9	[9.2, 0.0]

Reduced Sub-Cube

Tuples			TIDs
a1	b2	c2	[1, 7]
*	b2	c3	[1, 7]
*	b2	c2	[3, 9]
a1	b2	c3	[3, 9]
a1	*	c2	[1, 7]
a1	*	c3	[3, 9]
*	*	c2	[1, 7]
*	*	c3	[3, 9]
a1	b2	*	[1, 3, 7, 9]
*	b2	*	[1, 3, 7, 9]
a1	*	*	[1, 3, 7, 9]
*	*	*	[1, 3, 7, 9]

Final Sub-Cube

Tuples			TIDs
a1	b2	c2	5.8
*	b2	c3	5.8
*	b2	c2	4.6
a1	b2	c3	4.6
a1	*	c2	5.8
a1	*	c3	4.6
*	*	c2	5.8
*	*	c3	4.6
a1	b2	*	5.2
*	b2	*	5.2
a1	*	*	5.2
*	*	*	5.2

Figure 3.11: The measure calculus from the query is applied over the measure values of the final reduced sub-cube's tuples.

In the worst case, each device of a cluster receives identical tuples, producing identical sub-cubes and consequently identical results after the algorithm "sub-cube construction", so a reduction must be executed to eliminate redundancies from all devices and perform the final aggregations of the measure values. This is done sequentially by a device of the cluster, so any device can perform the "parallel reduction" step of the pipeline, illustrated in Figure 3.7. Basically, this device receives all tuples and their corresponding set of TIDs from each device, so the redundancy is eliminated by uniting TIDs. This pipeline step can be implemented in distribute way, similar to Hive, Kylin and other Hadoop based middlewares Dean and

Ghemawat (2008); Ranawade et al. (2016); Thusoo et al. (2009) and it is part of JCL-GPU-Cubing future plans.

This query pipeline step is illustrated in Algorithm 3 and is called parallel reduction, but the current implementation does not consider any parallel strategy for the sub-cube reductions and its measure calculus. This algorithm simply reduces all sub-cubes redundancies from the cluster, whether it was a CPU or GPU execution, as seen in line 2. The loop in line 2 translates the TIDs from each tuple generated in the sub-cube into several measure values obtained from the Measure Index 4. After recovering all the measure values, the specified measure *FUNCTION* is applied to the list of measure values, as seen in line 5. In our example, the AVG measure function is applied, but any other measure function works, such as the spatial distance of two geo-objects, the ranking of several documents and so forth.

Algorithm 3: The measure calculus algorithm

```
Input: The multiple sub-cubes $C$ that were generated at each device of the cluster;
       The Measure Index generated at each device of the cluster;
       The measure function portion of the query $Q$ of the form: $M$ $FUNCTION$;
Output: The cube $C^*$ that has all intersections from all sub-cubes and each tuple has
        its measure values aggregated w.r.t. the query $Q$;
1 On a single device of the cluster, receive all sub-cubes $C$;
2 Aggregate all received $C$ in $C^*$ for each tuple $c$ in $C^*$ do
3     ids = c[TIDs];
4     recover all measure values of $M$ in $c$ associated with id, using the Measure Index;
5     apply the $FUNCTION$ over the measure values of $c$ and store the result in $C^*$;
6 end
7 return the resultant sub-cube $C^*$.
```

3.3 Discussions

In this chapter we presented the core ideas of the JCL-GPU-Cubing approach, developed to achieve high speedups while answering complex multidimensional queries from data cubes and over CPU-GPU clusters. We present two solutions, one for indexing and other for query multidimensional data. The query was also designed to work in CPU and GPU. The update algorithm follows mainly the indexing counterpart, so we omit its explanation now, postponing it for the masters course. As future improvements for this work, the following points can be addressed:

1. Improved filtering strategy: project and implement an improved version of the filtering step to eliminate the need of the "Tuples Index" data structure, improving memory consumption.

2. An alternative approach for when the sub-cube to be constructed in GPU doesn't fit its memory: project and implement an alternative solution to the tuple queuing in CPU to be processed in the GPU, demanding multiple CPU to GPU memory transfers.

3. Update support: we must detail the supported types of updates on the data cube and how the indexing algorithm is changed or extended to handle the different those types of update;
4. AND & OR logical operators: the query must be formally defined in this chapter and how AND and OR logical operators are used in a query must be also defined and exemplified;
5. Distributed solutions for the “Parallel Reduction” and “Measure Calculus” steps: both must be detailed;
6. Discussions: we must introduce deep discussions about JCL-GPU-Cubing strengths and limitations, always comparing it against the state-of-art in OLAP, precisely against the research frontier in GPU accelerators to speedup data cube queries.

Chapter 4

Experiments

In this chapter, we present various comparative scenarios to evaluate the CPU only and the CPU-GPU hybrid version. We first present the cluster and devices' configurations, the base relations and the query used to execute the experiments. Then the experimental results and discussions are presented for the indexing and query pipelines. The query pipeline is evaluated in several cluster setups to compare the CPU version to the GPU version of the sub-cube construction algorithm.

4.1 Setup

The experimental environment consists in an heterogeneous cluster with 6 devices of various processing and memory capacities. The devices are interconnected via an standard gigabit Ethernet switch. The configuration of each device is shown in Table 4.1 and organized in order, from the best to the worst device in terms of memory and processing capacities. Table 4.2 shows the cluster setups used. Note that, although setups A and B use the same devices, the query execution is done in CPU and GPU, respectively. The hybrid execution mode, in setup E, denotes that all available devices are used in the cluster with devices 1, 2 and 3 run in GPU mode. We tested the system under such setups with three base relations: BR1) with 1 million tuples, BR2) with 2 million tuples and BR3) with 4 million tuples. All base relations have ten dimensions and two measures. The dimensional values have no skew and they are integer numbers with carnality of 1000, so they can assume values ranging from 1 to 1000.

As we discussed on Chapter 3, we do not build the full data cube from the base relation; instead, we only build a partial data cube with partial aggregations and when a query is submitted all necessary aggregations to produce the data cube to answer the query are calculated on-the-fly. Therefore, the data cube built from query `Q` has 5 dimensions and 1 measure.

```
Q: MAX M1
WHERE A > ‘600’ AND B > ‘650’ AND C > ‘800’ AND INQUIRE D AND INQUIRE E;
```

Table 4.1: The Devices used during the experiments and the respective configuration. Listed from the best, to the worst device available.

Device	**O.S.**	**CPU Model**	**CPU Cores**	**System RAM**	**GPU Model**	**GPU Cores**	**GPU Memory**
1	Windows 7	Intel Core i5-2500 @ 3.30GHz	4	32GB	NVIDIA GeForce GTX TITAN Z	5760	12GB
2	Windows 7	Intel Core i7-4790 @ 3.60GHz	8	16GB	NVIDIA GeForce GTX 460 v2	336	1GB
3	Windows 7	Intel Core i5-2400 @ 3.10GHz	4	8GB	NVIDIA GeForce GTX 460 v2	336	1GB
4	Ubuntu 16.04	Intel Xeon E5405 @ 2.0GHz	8	16GB	NA	NA	NA
5	Ubuntu 16.04	Intel Xeon @ 3.00GHz	4	16GB	NA	NA	NA
6	Ubuntu 16.04	Intel Core i7-3720M @ 2,60GHz	8	8GB	NA	NA	NA

Table 4.2: The multiple cluster setups used in the experiments.

Cluster Setup	**Devices Used**	**Query Execution Mode**
A	1, 2, 3	CPU
B	1, 2, 3	GPU
C	4, 5, 6	CPU
D	1, 2, 3, 4, 5, 6	CPU
E	1, 2, 3, 4, 5, 6	Hybrid

We evaluated the index and the query pipelines in terms of total time, i.e., the time elapsed to index or query are presented, so network, stack and other times are omitted. The query was executed 3 times over a cluster with one of the configurations explained before. The final values used on graphics of this chapter are the average of the 3 query response times obtained, presenting also the standard deviation as the error bars in the plots. All cluster devices received the same number of tuples, so there is no partition strategy using, for instance, the base relation properties versus the devices configurations. Actually, we just adopted a circular list strategy to partition the base relation tuples among the cluster devices.

4.2 Experimental Evaluations

We first evaluate the JCL-GPU-Cubing indexing algorithm scalability under the multiple CPU cluster setups. Next, the scalability of queries was evaluated while running them in GPU mode, CPU mode and in hybrid mode, so it is possible to measure the GPU improvements.

4.2.1 Indexing

As discussed in Chapter 3, the indexing algorithm runs entirely in CPU, so we evaluated it with different CPU cluster setups A, C and D, respectively. Figure 4.1 illustrates such an algorithm indexing the base relations BR1, BR2 and BR3. The cluster setup D has six devices and it indexed all three base relations faster than the others, once it represents the union of clusters A and C. As the base relations increased the difference between cluster D and the others also increased and the reason is because there are more devices to run the indexing algorithm. Moreover, the tuples of all the base relations BR1, BR2 and BR3 had to be send to remote devices, so in a bigger cluster the queue time tends to be reduced during the tuples transfer.

The worse runtime was obtained by cluster C, since it represents the worse devices in terms of processing and storage. In general, the indexing runtimes increased linearly according to the base relation increase, except for the biggest base relation BR3 and the worst cluster C. For this scenario, the indexing algorithm degraded more than the number of new tuples, so we understand that the cluster C started to become saturated while indexing the base relation BR3 with 4 million tuples, requiring new devices to maintain the quality of the service.

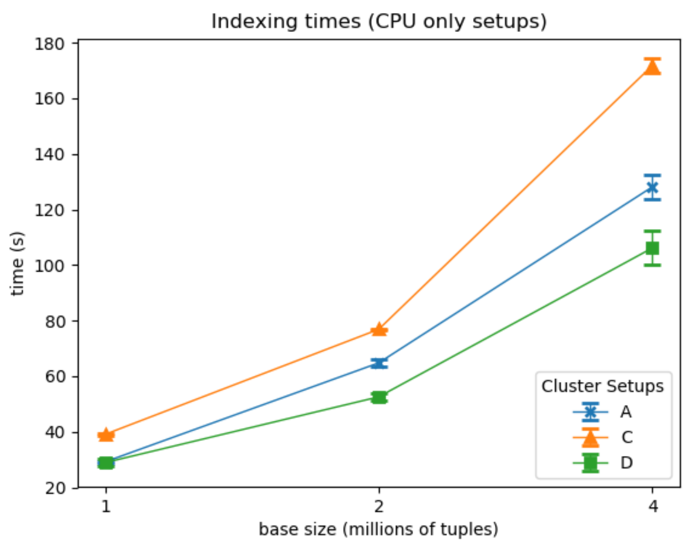


Figure 4.1: Indexing runtimes with the best devices (cluster setup A), the worst devices (cluster setup C) and with the complete cluster (setup D).

4.2.2 Query

To evaluate JCL-CPU-Cubing query algorithm, we compared its performance over multiple cluster setups, reinforcing the utilization of multiple GPUs to create the sub-cubes to answer the queries. The query `Q` was executed multiple times and it returned 33.114 tuples for BR1, 66.501 tuples for BR2 and 2.128.032 tuples in BR3. During a query of the BR3 base relation, for instance, 132.385 tuples satisfied `Q`, but to produce the complete sub-cube the algorithm generated approximately two million aggregated tuples where the wildcard ALL (*) was present.

4.2.2.1 Best CPU Devices x GPU Devices

The Figure 4.2 illustrates the query response time using the cluster setups A and B and it is possible to note a linear behaviour of query processing of cluster B using GPUs. The small size of base relations BR1 and BR2 demonstrated that the data transfer between CPU and GPU did not compensate the numerous cores present in each GPU, this way the cluster setup with the best devices running in CPU outperformed the GPU cluster when the small base relations are queried. In opposite direction, when the number of tuples to answer `Q` reached 4 million tuples using BR3, the GPU cluster outperformed its CPU counterpart, so it is possible to see the benefits of GPU while answering big multidimensional queries. Another important issue about the GPU implementation is that there is a reduction step of the base tuples in each device and before the sub-cube construction, therefore, the GPU query could be improved. However, when the parallel reduction is executed at the end of the query pipeline there are fewer redundancies to be eliminated, since some of them were already eliminated during the previous reduction step.

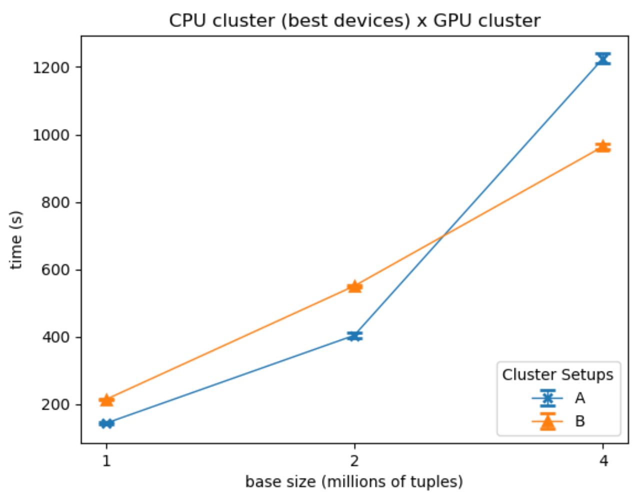


Figure 4.2: Query response times with the best devices running in CPU (setup A) and the best devices running in GPU (setup B).

4.2.2.2 Worst CPU Devices x GPU Devices

The Figure 4.3 illustrates the query comparisons of cluster setups B and C, where the setup C represents the worse devices. In this case, the GPU version outperformed the CPU version in all scenarios. The justification was because setup B had better devices with GPUs inside, so as the query becomes bigger the difference in terms of response time also increased, as observed with base relation BR3.

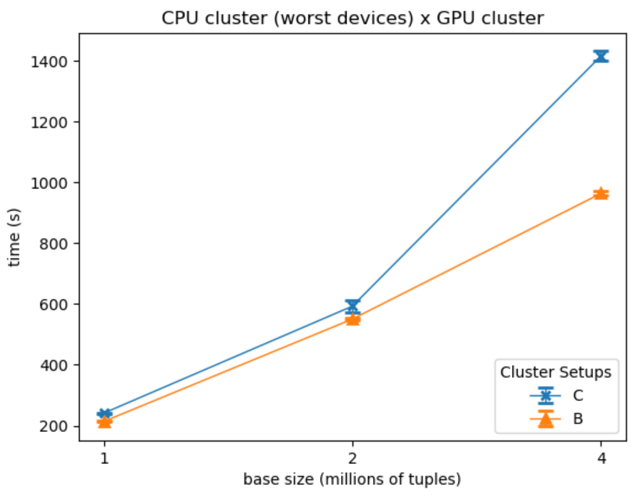


Figure 4.3: Query response times with the worst devices running in CPU (setup C) and the best devices running in GPU (setup B).

4.2.2.3 GPU Devices x Complete Hybrid Cluster

In this experiment, the query is executed using setup B and setup E, the last representing the deployment of the complete cluster with the best devices running in GPU and the worst devices in CPU. The Figure 4.4 illustrates that for the small base relations, precisely BR1 and BR2, setup E outperformed cluster setup B, since there are more tuples to transfer between memories in setup B than there are in setup E, where the workload is distributed across more devices in the cluster. When processing the base relation BR3 the setup B outperformed setup E since the devices running in CPU, precisely the worst devices, demanded more time to process the filtering and to create the sub-cube to answer the query. This experiment instigates further investigation to determine if more GPUs on the cluster is better as the base relation increases, since the GPU-only devices outperformed the hybrid cluster and this particular setup presented a linear scalability.

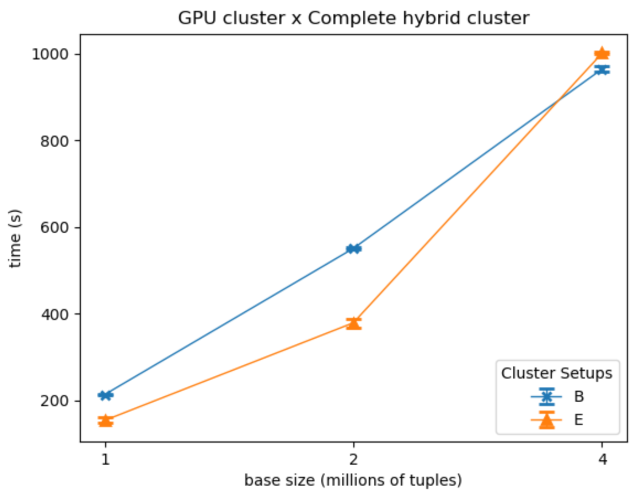


Figure 4.4: Query response times with the best devices ruining in GPU (setup B) and the the hybrid cluster (setup E).

4.2.2.4 Complete CPU Cluster x Complete Hybrid Cluster

The next experiment evaluated the performance of the GPU query execution using the complete cluster in hybrid mode and using the complete cluster running in CPU only mode. Figure 4.5 illustrates that setup E has a worst performance than setup D with BR1 and BR2. This is due to the data transfer overhead not being compensated by the sub-cube construction processing gains in GPU, as already discussed. Besides that, since setup D does not need to execute the data transfer before the sub-cube construction, it performed better with those base relations. The main difference with BR3 is that in the presence of more tuples to be processed, the GPU devices in setup E had the data transfer overhead compensated by the sub-cube construction. Besides that, the worst devices, precisely the devices 4, 5 and 6, became the bottleneck of the cluster, this way, when using them in any cluster setup it is expected to have some performance degradation, similarly to the previous experiment.

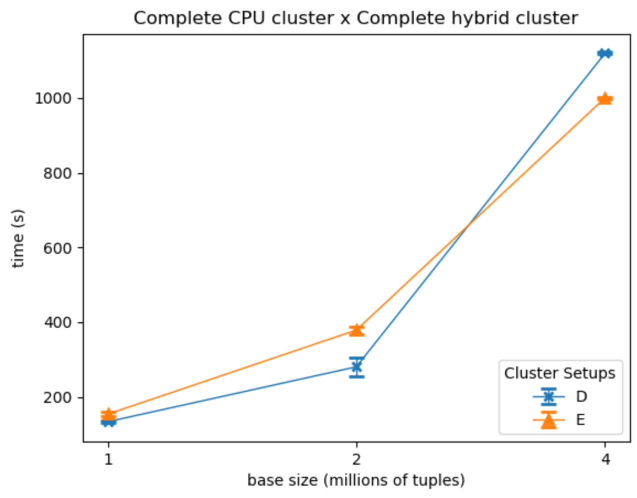


Figure 4.5: Query response times with the complete cluster running in CPU (setup D) and the the hybrid cluster (setup E).

4.2.2.5 Worst CPU Devices x Complete CPU Cluster

This experiment illustrates how the CPU version scales when the cluster increases. Figure 4.6 illustrates a cluster with first the worst devices deployed and then the best ones. When ruining with more devices in the cluster the workload is more distributed across the devices, causing a considerable performance gain with all base relations. More experiments with more devices to understand the query saturation, i.e., when a specific base relation size does not guarantee query scalability above a certain number of devices and the same can be done for GPU devices.

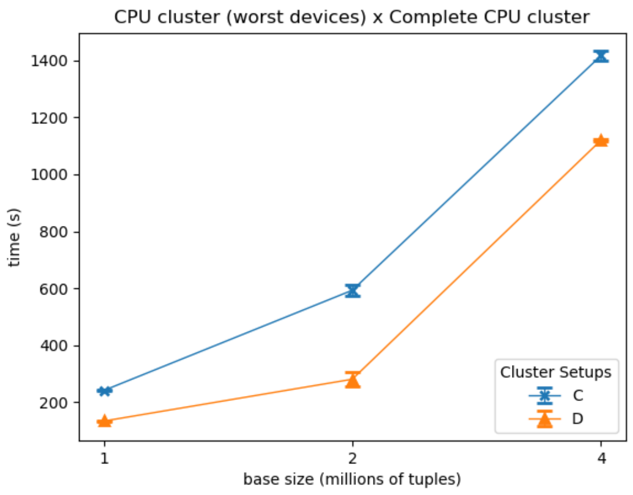


Figure 4.6: Query response times with the worst devices running in CPU (setup C) and the complete cluster running in CPU (setup D).

4.2.2.6 Best CPU Devices x Complete CPU Cluster

The final experiment evaluated the complete CPU cluster against the bet CPU devices, so a similar behaviour is obtained, i.e., Figure 4.7 illustrates that setup D outperformed the cluster setup A, since setup A has half the number of devices. It is important to note that the difference is less significant than the previous experiment, with closer query response times, because this experiment used the best devices. Although the CPU only version did not scale linearly as the number of tuples grow, this experiment reinforced the hypotheses that as the number of devices and the hardware quality improve, the query runtimes would be also improved.

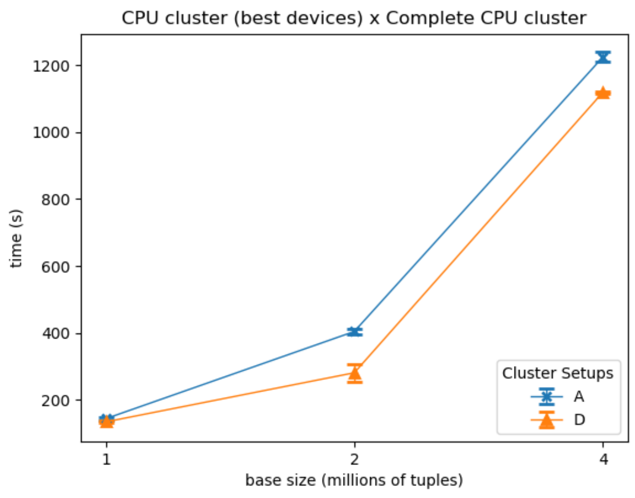


Figure 4.7: Query response times with the best devices running in CPU (setup A) and the complete cluster running in CPU (setup D).

4.3 Discussions

In this chapter we presented the experimental evaluations of JCL-GPU-Cubing approach. The current implementation uses the CPU to create the indexes, filter and the reduce of the input base relation, using sometimes the GPU to construct the sub-cube and answer the user query. The evaluation of the results presented above showed that the indexing algorithm scales when the base relation and cluster size grows, so when more devices are added to the cluster it is expected to have performance gains. For the query processing, the presented approach works only for big query results, since there are extra processing tasks in GPU implementation and there is also data transfer from CPU to GPU and vice-versa.

Further experiments can be made in order to enable more accurate and deeper evaluations:

1. Execute JCL-GPU-Cubing in a cluster with more GPU devices to verify the linear scalability;

2. Deploy a heterogeneous cluster and try to find a point of diminishing returns for the query processing with both CPU and GPU devices;

3. Test base relations with different skews, cardinality and high dimensionality;

4. Test JCL-GPU-Cubing with holistic measures versus other measure types;

5. Evaluate JCL-GPU-Cubing against both the related work and the benchmarks.

Chapter 5

Conclusion

This work presented the initial ideas of a solution to solve a fundamental problem in OLAP literature and consequently for the data cube relational operator: how to reduce complex multidimensional queries response times? The problem becomes even worse if recurrent updates in the base relation is supported and if there is a huge volume of high dimensional data to be analyzed.

This work presents an alternative solution to solve the problem stated before. We use GPU cards to speedup queries. This idea has been done before, but using a single device and two GPU cards at most. We are interested in cluster deployments of GPUs, so multiple private memories must be considered. Many related work require the base relation totally stored in GPU memory, which is unrealistic for large base relations. Sometimes they do not support holistic measures or range queries or hybrid CPU-GPU benefits. This work innovates in a solution designed for: i) high dimensionality; ii) holistic measures; iii) range queries; iv) hybrid CPU-GPU solution; v) cluster version and not only multi-core version to speedup queries.

Experimental results demonstrated that the GPU version scales when both the query result and the number of devices increase. These results reinforced our hypothesis that GPU clusters can speedup query response times. We plan future tests, including comparative ones with GPU and CPU versions. More experiments with synthetic and real data are necessary. The data cube proprieties, i.e., its number of dimensions, tuples, its cardinality and skew must be varied for a better understanding about JCL-GPU-Cubing benefits and limitations.

To summarize the future work, we can enumerate the following acclivities to be done:

1. Implement, detail and test the improved filtering strategy to eliminate the need of the "Tuples Index" data structure, improving memory consumption;

2. Implement, detail and test an alternative approach with multiple data transfers for a single query in GPU that does not fit in its memory;

3. Implement, detail and test the update support;
4. Implement, detail and test AND & OR logical operators and how the are used in the query processing;
5. Implement, detail and test the distributed solutions for the “Parallel Reduction” and “Measure Calculus” steps;
6. Execute JCL-GPU-Cubing in a cluster with more GPU devices to verify the linear scalability;
7. Use a heterogeneous cluster and try to find a point of diminishing returns for the query processing with both CPU and GPU devices;
8. Use base relations with different skews, cardinality and high dimensionality;
9. Evaluate JCL-GPU-Cubing with holistic measures.

Bibliography

Almeida, A. L. B., Cimino, L. d. S., de Resende, J. E. E., Silva, L. H. M., Rocha, S. Q. S., Gregorio, G. A., Paiva, G. S., Delabrida, S., Santos, H. G., de Carvalho, M. A. M., et al. (2018). A general-purpose distributed computing java middleware. *Concurrency and Computation: Practice and Experience*, page e4967.

Appuswamy, R., Karpathiotakis, M., Porobic, D., and Ailamaki, A. (2017). The case for heterogeneous htap. In *8th Biennial Conference on Innovative Data Systems Research.*

Benatallah, B., Motahari-Nezhad, H. R., et al. (2016). Scalable graph-based olap analytics over process execution data. *Distributed and Parallel Databases*, 34(3):379–423.

Bimonte, S., Tchounikine, A., and Miquel, M. (2006). Geocube, a multidimensional model and navigation operators handling complex measures: Application in spatial olap. In *ADVIS*, pages 100–109. Springer.

Bimonte, S., Tchounikine, A., Miquel, M., and Pinet, F. (2011). When spatial analysis meets olap: Multidimensional model and operators. *Exploring Advances in Interdisciplinary Data Mining and Analytics*, pages 249–277.

Breß, S. (2014). The design and implementation of cogadb: A column-oriented gpu-accelerated dbms. *Datenbank-Spektrum*, 14(3):199–209.

Breß, S., Siegmund, N., Bellatreche, L., and Saake, G. (2013). An operator-stream-based scheduling engine for effective gpu coprocessing. In *East European Conference on Advances in Databases and Information Systems*, pages 288–301. Springer.

Cuzzocrea, A. (2015a). Aggregation and multidimensional analysis of big data for large-scale scientific applications: Models, issues, analytics, and beyond. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*, SSDBM '15, pages 23:1–23:6, New York, NY, USA. ACM.

Cuzzocrea, A. (2015b). Data warehousing and olap over big data: a survey of the state-of-the-art, open problems and future challenges. *International Journal of Business Process Integration and Management*, 7(4):372–377.

Cuzzocrea, A., Bellatreche, L., and Song, I.-Y. (2013). Data warehousing and olap over big data: Current challenges and future research directions. In *Proceedings of the Sixteenth International Workshop on Data Warehousing and OLAP*, DOLAP '13, pages 67–70, New York, NY, USA. ACM.

Cuzzocrea, A., Simitsis, A., and Song, I.-Y. (2016). Big data management: New frontiers, new paradigms. *Information Systems.*

de Resende, J. E. E., de Souza Cimino, L., Silva, L. H. M., Rocha, S. Q. S., de Oliveira Correia, M., Monteiro, G. S., de Souza Fernandes, G. N., Almeida, S. G. M., Almeida, A. L. B., de Aquino, A. L. L., and de Castro Lima, J. (2017). Jcl android : Uma extensão iot para o middleware hpc jcl. In *Brazilian Symposium on Computing Systems Engineering.*

Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

Gonzalez, H., Han, J., and Li, X. (2006). Flowcube: constructing rfid flowcubes for multi-dimensional analysis of commodity flows. In *Proceedings of the 32nd international conference on Very large data bases*, pages 834–845. VLDB Endowment.

Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., Venkatrao, M., Pellow, F., and Pirahesh, H. (1997). Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data mining and knowledge discovery*, 1(1):29–53.

Han, J., Pei, J., and Kamber, M. (2011). *Data mining: concepts and techniques.* Elsevier.

Ho, C.-T., Agrawal, R., Megiddo, N., and Srikant, R. (1997). *Range queries in OLAP data cubes*, volume 26. ACM.

Jin, X., Han, J., Cao, L., Luo, J., Ding, B., and Lin, C. X. (2010). Visual cube and on-line analytical processing of images. In *Proceedings of the 19th ACM international conference on Information and knowledge management*, pages 849–858. ACM.

Kaczmarski, K. (2011). Comparing gpu and cpu in olap cubes creation. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 308–319. Springer.

Kaczmarski, K. and Rudny, T. (2011). Molap cube based on parallel scan algorithm. In *Advances in Databases and Information Systems*, pages 125–138. Springer.

Karnagel, T. and Habich, D. (2017). Heterogeneous placement optimization for database query processing. *it-Information Technology*, 59(3):117–123.

Lauer, T., Datta, A., Khadikov, Z., and Anselm, C. (2010). Exploring graphics processing units as parallel coprocessors for online aggregation. In *Proceedings of the ACM 13th international workshop on Data warehousing and OLAP*, pages 77–84. ACM.

Lima, J. C. and Hirata, C. M. (2011). Multidimensional cyclic graph approach: Representing a data cube without common sub-graphs. *Information Sciences*, 181(13):2626–2655.

Lin, C. X., Ding, B., Han, J., Zhu, F., and Zhao, B. (2008). Text cube: Computing ir measures for multidimensional text database analysis. In *2008 Eighth IEEE International Conference on Data Mining*, pages 905–910. IEEE.

Liu, M., Rundensteiner, E., Greenfield, K., Gupta, C., Wang, S., Ari, I., and Mehta, A. (2011). E-cube: multi-dimensional event sequence analysis using hierarchical pattern query sharing. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 889–900. ACM.

Malik, M., Riha, L., Shea, C., and El-Ghazawi, T. (2012). Task scheduling for gpu accelerated hybrid olap systems with multi-core support and text-to-integer translation. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1987–1996. IEEE.

Moreira, A. A. and Lima, J. C. (2012). Full and partial data cube computation and representation over commodity pcs. In *Information Reuse and Integration (IRI), 2012 IEEE 13th International Conference on*, pages 672–679. IEEE.

Moreno, F., Arango, F., and Fileto, R. (2009). Extending the map cube operator with multiple spatial aggregate functions and map overlay. In *Geoinformatics, 2009 17th International Conference on*, pages 1–7. IEEE.

Nes, S. I. F. G. N. and Kersten, S. M. S. M. M. (2012). Monetdb: Two decades of research in column-oriented database architectures. *Data Engineering*, 40.

Ranawade, S. V., Navale, S., Dhamal, A., Deshpande, K., and Ghuge, C. (2016). Online analytical processing on hadoop using apache kylin.

Riha, L., Malik, M., and El-Ghazawi, T. (2013). An adaptive hybrid olap architecture with optimized memory access patterns. *Cluster computing*, 16(4):663–677.

Riha, L., Shea, C., Malik, M., and El-Ghazawi, T. (2011). Task scheduling for gpu accelerated olap systems. In *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, pages 107–119. IBM Corp.

Silva, R. R., Hirata, C. M., and de Castro Lima, J. (2015). A hybrid memory data cube approach for high dimension relations. In *ICEIS (1)*, pages 139–149.

Silva, R. R., Lima, J. d. C., and Hirata, C. M. (2013). qcube: efficient integration of range query operators over a high dimension data cube. *Journal of Information and Data Management*, 4(3):469–482.

Sismanis, Y., Deligiannakis, A., Roussopoulos, N., and Kotidis, Y. (2002). Dwarf: Shrinking the petacube. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 464–475. ACM.

Sitaridi, E. A. and Ross, K. A. (2012). Ameliorating memory contention of olap operators on gpu processors. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, pages 39–47. ACM.

Souza, A. N. P., Fortes, R. S., and Lima, J. C. (2017). Dynamic topic hierarchies and segmented rankings in textual olap technology. *Journal of Convergence Information Technology (JCIT)*, 12(1):1–17.

Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., and Murthy, R. (2009). Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629.

Wang, G. and Zhou, G. (2012). Gpu-based aggregation of on-line analytical processing. In *Communications and Information Processing*, pages 234–245. Springer.

Wang, H., Wang, Z., Li, N., and Kong, X. (2018). Efficient olap algorithms on gpu-accelerated hadoop clusters. *Distributed and Parallel Databases*, pages 1–36.

Wang, Y., Song, A., and Luo, J. (2010). A mapreducemerge-based data cube construction method. In *Grid and Cooperative Computing (GCC), 2010 9th International Conference on*, pages 1–6. IEEE.

Wang, Z., Fan, Q., Wang, H., Tan, K.-L., Agrawal, D., and El Abbadi, A. (2014). Pagrol: parallel graph olap over large-scale attributed graphs. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 496–507. IEEE.

Wittmer, S., Lauer, T., and Datta, A. (2011). Real-time computation of advanced rules in olap databases. In *Advances in Databases and Information Systems*, pages 139–152. Springer.

Xin, D., Han, J., Li, X., Shao, Z., and Wah, B. W. (2007). Computing iceberg cubes by top-down and bottom-up integration: The starcubing approach. *IEEE Transactions on Knowledge and Data Engineering*, 19(1):111–126.

Zhang, D., Zhai, C., and Han, J. (2009). Topic cube: Topic modeling for olap on multidimensional text databases. In *SDM*, volume 9, pages 1124–1135. SIAM.

Zhang, J., You, S., and Gruenwald, L. (2012). High-performance online spatial and temporal aggregations on multi-core cpus and many-core gpus. In *Proceedings of the fifteenth international workshop on Data warehousing and OLAP*, pages 89–96. ACM.

Zhang, J., You, S., and Gruenwald, L. (2014). Parallel online spatial and temporal aggregations on multi-core cpus and many-core gpus. *Information Systems*, 44:134–154.

Zhao, P., Li, X., Xin, D., and Han, J. (2011). Graph cube: on warehousing and olap multidimensional networks. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 853–864. ACM.

Certifico que o aluno **Lucas Henrique Moreira Silva**, autor do trabalho de conclusão de curso intitulado "**Computing Data Cubes Over GPU Clusters**", efetuou as correções sugeridas pela banca examinadora e que estou de acordo com a versão final do trabalho.

[illegible]

Joubert de Castro Lima
Orientador

Ouro Preto, 17 de dezembro de 2018.