Massively Distributed Datacube Processing

Vlad Merticariu and Peter Baumann
Jacobs University, v.merticariu@jacobs-university.de, p.baumann@jacobs-university.de

Abstract - **Datacubes provide a suitable paradigm for storing, accessing and processing large-scale, multi-dimensional spatio-temporal raster data. As hardware and distributed infrastructure, such as cloud and federations, become common, enabling datacubes to fully exploit the available capabilities is a crucial step in building scalable systems for complex query answering in real time. In this contribution, we describe an approach to distributed datacube processing that enables datacube engines – in our case rasdaman – to analyze, for every incoming query, a large number of equivalent distributed execution variants, and to pick an efficient one.**

Index Terms – array, datacube, distributed processing, query planning, rasdaman

INTRODUCTION

The datacube model is gaining more and more attention when dealing with Big Data challenges in a variety of domains such as remote sensing, climate simulations, geographic information systems, medical imaging or astronomical observations. Solutions provided by classical Big Data tools such as NoSql [1] and MapReduce [2] proved to be very efficient when dealing with simple, linear, data structures, however they are limited in domains associated with multi-dimensional data [3]. Traditional relational databases share the same problem: even though they are known for successfully handling data of any size, they lack the tools for dealing with multi-dimensional datasets. This problem has been addressed by the field of array databases, in which systems provide datacube services for raster data, without imposing limitations on the number of dimensions that a dataset can have. Examples of datasets usually handled by array databases include 1-dimensional sensor data, 2-D satellite imagery, 3-D x/y/t image time series as well as x/y/z geophysical voxel data, and 4-D x/y/z/t weather data. In life sciences, there is laser scan microscopy and brain scans. And this can grow as large as simulations of the whole universe when it comes to astrophysics [4].

Due to the complexity of array operations, and the high data volumes involved, several techniques are used for speeding up array queries, the most prolific ones being query optimization and parallel query processing. Both methods have been extensively studied in the relational database field, but only partially applied to array databases due to specific raster data properties which require different approaches. For example, one of the most characterizing properties of arrays is the well-defined Euclidean neighborhood, which has high impact on access locality (when a particular cell is accessed it is extremely likely that its neighbor pixels will also get accessed) and induces efficient partitioning techniques for storage [5].

RELATED WORK

Exploiting parallelism to process queries has been widely explored for improving query response times in relational databases [6][7]. As RDBMS operations are usually limited to retrieval, filtering and aggregations, most of the systems focus either on partitioning the data and applying the same query execution tree to smaller partitions or distributing a limited set of operators to different processors on the same machine [7]. While this represents a good starting point for exploiting parallelism in datacubes, further optimization opportunities are missed: non-trivial (non "embarrassingly parallel") tasks are common in the datacube world, and they are not targeted. Furthermore, heterogeneous infrastructures (e.g. part of the datacube sitting in a cloud, and part of it sitting on-board a satellite [8]), where hardware properties vary considerably, require a more detailed inspection of the possible distribution alternatives and their costs, which we achieve with the method introduced in the paper.

Another related framework is MapReduce [2]: a programming model for simplified parallel processing of large datasets. It has two components: a Map procedure that performs filtering and sorting, and a Reduce procedure for summarizing the results of the Map step. Hadoop [9], the open source MapReduce implementation, supports the execution of programmable user tasks. Its efficiency, however, is known to be poor when compared to parallel databases [10], and, when it comes to datacubes, specific raster data properties, such as Euclidian neighborhood of pixels, are ignored, and each pixel is processed individually, making it orders of magnitude slower [11].

Google Earth Engine combines a catalog of satellite imagery and geospatial datasets with analytics capabilities [12]. It builds on the tradition of Grid systems with files however, without providing datacube paradigm. Based on a functional programming language, without advanced parallelization capabilities (only "embarrassingly parallel" operations), users can submit code which is executed transparently in Google's own distributed environment, with a worldwide private network. The method that we present, on the contrary, handles distribution of any operation, in any network. To the best of our knowledge, no other framework achieves that in the datacube world.

QUERY PROCESSING

Storage-wise, datacubes are split into units of access called tiles [5], optimization which enables efficient data retrieval by minimizing the amount of unnecessary information which is read from disk, by retrieving only the tiles affected by the query. Tiles of the same datacube can sit on different machines, which, as we will see in the following sections, enhances the level of parallelism in processing.

Datacube operations are usually triggered via standardized web services, such as WCS, WMS and WCPS [13]. However, these services are only the interface to the client performing the operation, while the processing itself is handled at a lower level which enables optimized, efficient execution. In rasdaman, the geo layer intercepting the web requests representing datacube operations translates them into array database queries, which are then handled by rasdaman's query processing engine [14].

For example, computing the NVDI over Europe, on the 1st of July 2018, using Sentinel data stored in a datacube with the same name, and returning the result as a NetCDF file, can be achieved with the following WCPS query:

```
for $c in (Sentinel) return encode(
  (($c.nir – $c.red) / ($c.nir + $c.red))
  [ Lat(35:70), Long(10:40), ansi("2018-07-01") ],
 "image/tiff")
```

In this case, the datacube has 3 dimensions corresponding to Latitude, Longitude and Time (here expressed using ansi coordinate system) axes.

Traditionally, in relational systems, queries are modeled as query trees: tree data structures representing relational expressions [7]. The tables involved in the queries are represented as leaf nodes and the operations are represented as internal nodes. Similarly, in datacube systems, datacube expressions can be modeled as trees. The tiles touched by the query are represented as leaves, and the operations as internal nodes.

Prior to evaluation, queries undergo heuristic optimization based on rewriting rules (for example, subsets are pushed down in the query tree, in order to discard unnecessary data as early as possible). Once heuristic optimization concludes, cost-based approaches can be used to further explore execution alternatives. In the next sections, we present such an approach, which focuses on inspecting the cost of executing different parts of the heuristically optimized query tree on different machines available in the network.

COST MODEL

In order to evaluate different execution variants of the same query, a cost model has been established, taking into account three key factors: the amount of data accessed, the amount of data transported over the network, and the amount of processing. The processing costs are further differentiated into resource costs and time costs (e.g. averaging over 1GB of data sequentially has the same resource processing cost as averaging over 1GB of data under parallel load because the same amount of CPU cycles is spent but has a higher time processing cost because it takes longer). In future, this will allow for different optimization strategies, such as minimizing response times, minimizing internal data transport, or optimizing overall service load balance.

Formally, the cost is defined as a function taking as input a query tree, and returning the following quadruple:

- C_a: data access cost, measuring the total amount of data read from disk in the evaluation of the input query. This is measured by summing up the sizes of data accessed in the leaf nodes of the query tree representing tile access.
- C_r: processing resource cost, measuring the total amount of processing spent in the evaluation of the query. This is measured recursively in the operation nodes of the query tree, the cost of the parent node being the sum of the processing resource costs of all children of the node, plus operated data size multiplied by a constant that is different for each operation. The constant determines how much processing costs for a given operation on a particular machine (e.g. addition is cheaper than square root), and can be approximated by running a micro benchmark at application start. The final cost of the query is the cost of the root node.
- C_p: processing time costs, measuring the total processing time spent in the evaluation of the query. Similar to C_r, this is measured recursively, with the difference that when taking into account the cost of the children operations, instead of the sum, only the maximum one is considered.
- C_t: transport costs, measuring the total amount of data going over the network. This is done by summing data sizes of nodes representing subqueries to remote machines, as well as the size of the query result.

LOCATION AWARE PROCESSING

Given an array query tree, and a network, the goal is to determine the optimal execution location for each subexpression (or node), such that the overall cost is minimized. In order to efficiently represent the search space, we introduce a new data structure: location aware query trees. A location aware query tree is a query tree where each query tree node is decorated with the set of possible execution locations. In the query execution pipeline, a location aware query tree is created starting from a heuristically optimized query tree, by annotating all its nodes with a set of locations.

For example, in a network with 3 machines $m = \{S_1, S_2, S_3\}$, containing a datacube *A*, which in turn contains a single tile *a*, the query "for $c in (A) return $c + 1" is represented by the following query tree, where D represents the delivery node:

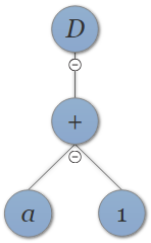

Figure 1. Query tree for "$c + 1".

Making this tree location aware is done by decorating each node with a set of possible execution locations. For example, the corresponding location aware tree that assumes that any node can be executed on any computer in the network is the following:

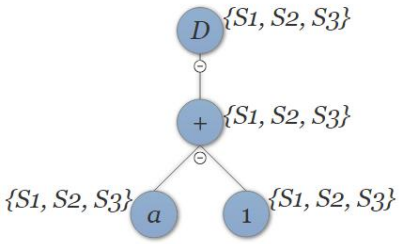

Figure 2. Location aware query tree.

However, not all possibilities are valid. For example, tile nodes (in this case *a*) can only be executed (i.e. loaded) from nodes on which they are stored, while the root node, which delivers the result back to the user, must reside on the node on which the query was issued. Furthermore, the node representing the constant value 1 can always be executed on the same machine where its parent resides, without incurring extra costs. Assume tile *a* sits on machine S_2, and the query is executed on machine S_1, the following refined location aware tree captures the above observations.

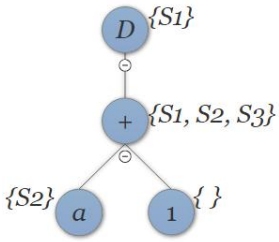

Figure 3. Refined location aware query tree.

The tree now indicates that the + node can be executed on any node in the network. The next steps are now to instantiate the corresponding tree for each variant, compute the corresponding costs, and pick the one that is the least expensive. However, in order to ensure that the tree can be in fact executed, the data delivered by the tile node *a* must be transported to the machine where the + node is to be evaluated, in case they sit on different computers. A similar step must be performed with the result of the + node, to ensure that in the end it reaches the machine where the delivery node sits, so the result can finally reach the user. The transportation of data is ensured by a special node (named T), which is added whenever a mismatch between the location of a parent node and the one of a child node occurs, leading to the following execution candidates.

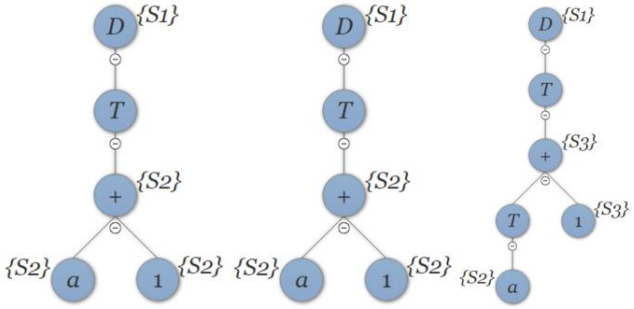

Figure 4. Resulting execution candidates E_1, E_2, E_3.

Using the previously discussed cost model, the engine can now rank the 3 candidates: they all access the same amount of data, and they all perform the same amount of processing. The amount of data transported through the network, however, is 1 tile for the E_1 and E_2 trees, and 2 tiles for the E_3 tree, making it a less desirable candidate. Thus, the engine will execute one of E_1 and E_2, which have the same cost.

PRUNING STRATEGIES

The method presented in the section above, while exhaustive in the number of possible location assignments for the query tree nodes, has the disadvantage of producing a potentially large number of candidates in case the query produces a tree with a large number of nodes, or in case the network is large. Heuristic strategies are used to reduce the number of candidates that need to be inspected. While presenting the exact details of the strategies is out of the scope of this paper, we briefly describe 2 of them, which have an increased practical relevance:

- Transport minimization: by pruning the execution locations of a node that are not part of the execution locations of its parent or children, the number of necessary network transfers is reduced [15]. This is relevant especially when federations of datacubes are involved, where transporting large amounts of data across datacenters is not feasible.
- Processing balancing: this strategy favors those candidates that achieve an equal distribution of processing tasks in the network, and is relevant in homogeneous environments with high bandwidth between the machines, such as clouds.

CONCLUSION

We presented a method that enables datacube engines to inspect and rank a large number of execution candidates, which allows determining suitable distributed execution plans for each incoming datacube query. While the set of possible candidates is sometimes too large for all its elements to be inspected individually, heuristic methods can be used for reducing it, depending on the particular situation. The optimization enables datacube engines to efficiently exploit distributed infrastructure in answering queries.

A real-life use-case of the optimization can be observed in the federation which was set up between the CODE-DE datacube precursor service established in the BigDataCube project [16], and the Alfred Wegener Institute for Polar and Marine Research (AWI). Among others, temperature datacubes are available at CODE-DE, and SeaIce datacubes are available at AWI. In one of the use-cases, scientists need to compute the sea ice distribution at different temperature intervals. The WCPS query that achieves that is presented below. Note that the involved datacubes contain data at different resolutions, and in different coordinate systems, so scale and reprojection operations are required before finally combining them to obtain the answer:

```
for $c in (SeaIce), $d in (T2m)
 return encode(
  coverage SeaIceByTemperature
  over $temp t(-40:0)
  values add(
       scale(
              $c[ ansi("2018-01-01")],
                  { Lat:"CRS:1"(0:360),
                    Long:"CRS:1"(0:719) } )
       *
       (($d[ansi("2018-01-01")] -273.15) > $temp)
       )
  , "csv")
```

The query iterates over temperature values in the interval (-40, 0), in Celsius. At each step, a binary mask is determined from the temperature data, corresponding to temperature pixels having values exceeding the iterator value. The sea ice data is reprojected and scaled to match the temperature data, then only pixels matching the binary temperature mask are considered. The values are then added to obtain, for each temperature step, the total sea ice value corresponding to it.

Because the setup is a federation, a strategy that minimizes the amount of data transported between locations is used. The winning execution plan is one where the scaling and the reprojection of the SeaIce data happens internally at AWI, balanced against the available processing nodes, and, similarly, the temperature binary mask generation happens at CODE-DE, yielding 2 intermediary results, which must end up on the same machine for the next step of the computation. Out of the 2 candidates, the binary masks is transported because of its smaller size (1 bit per pixel), as well as its suitability for compression. The final computation happens in the AWI network, and, in case the query was originally performed at CODE-DE, the final result, which is just 40 scalar values, is transported back. This allows the user to execute the same query in any of the 2 networks, without taking into account data placement, knowing that the engine will always pick an efficient execution schedule.

ACKNOWLEDGMENT

This work is being supported by H2020 LandSupport, H2020 EOSC-hub, and German BMWi BigDataCube.

REFERENCES

[1] Strauch, C. and Kriha W.. "NoSQL databases." *Lecture Notes, Stuttgart Media University*, 2011.

[2] Dean, J., and Ghemawat, S. "MapReduce: simplified data processing on large clusters." *Communications of the ACM*. 51.1, 2008, 107-113.

[3] Leavitt, N. "Will NoSQL databases live up to their promise?." *Computer*, 43.2, 2010, 12-14.

[4] N.n. “Big Science Data Coming to SQL Databases”. http://slashgeo.org/2014/06/27/big-science-data-coming-sql-databases/.

[5] Baumann P., et al. “Putting Pixels in Place: A Storage Layout Language for Scientific Data”. *Proc. IEEE ICDM Workshop on Spatial and Spatiotemporal Data Mining*, 2010.

[6] Pirahesh H., et al.”Parallelism in relational data base systems: architectural issues and design approaches”, *DPDS*, 1990.

[7] Hong, M. et al. "Query processing in a parallel object-relational database system." *Data Engineering*, 3, 1996.

[8] Baumann, P. et al. “Breaking the big data barrier by enhancing on-board sensor flexibility” *Proc. ACM BigSpatial*, 2013.

[9] N.n. “Apache Hadoop”. http://hadoop.apache.org/.

[10] Rusu, F., and Cheng, Y. "A survey on array storage, query languages, and systems". *arXiv preprint*, arXiv:1302.0103, 2013.

[11] Research Data Alliance. “Array Databases: Concepts, Standards, Implementations” *Research Data Alliance (RDA) Working Group Report*, dx.doi.org/10.15497/RDA00024, 2018.

[12] N.n. “Google Earth Engine”. https://earthengine.google.com/.

[13] N.n “Web Coverage Processing Service”. https://www.opengeospatial.org/standards/wcps.

[14] ISO, “Information technology — Database languages — SQL — Part 15: Multi-Dimensional Arrays”, ISO 9075-15:2018

[15] Dumitru, A. et al. “Exploring cloud opportunities from an array database perspective”. *Proc. ACM SIGMOD DanaC*, 2014.

[16] N.n., “BigData Cube”, http://www.bigdatacube.org/

AUTHOR INFORMATION

Vlad Merticariu, PhD Student, Department of Computer Science and Electrical Engineering, Jacobs University.
Peter Baumann, Professor, Department of Computer Science and Electrical Engineering, Jacobs University.