Astronomical Data Processing in EXTASCID

Yu Cheng
UC Merced
5200 N Lake Road
Merced, CA 95343
ycheng4@ucmerced.edu

Florin Rusu
UC Merced
5200 N Lake Road
Merced, CA 95343
frusu@ucmerced.edu

ABSTRACT

Scientific data have dual structure. Raw data are preponderantly ordered multi-dimensional arrays or sequences while metadata and derived data are best represented as unordered relations. Scientific data processing requires complex operations over arrays and relations. These operations cannot be expressed using only standard linear and relational algebra operators, respectively. Existing scientific data processing systems are designed for a single data model and handle complex processing at the application level.

EXTASCID is a complete and extensible system for scientific data processing. It supports both array and relational data natively. Complex processing is handled by a metaoperator that can execute any user code. As a result, EXTASCID can process full scientific workflows inside the system, with minimal data movement and application code. We illustrate the overall process on a real dataset and workflow from astronomy—starting with a set of sky images, the goal is to identify and classify transient astrophysical objects.

1. INTRODUCTION

Science represents one of the most important sources of Big Data. While effectively storing the data is a challenge in itself, the main problem scientists face is how to efficiently process the data in order to obtain novel insights and gain knowledge. There are multiple reasons for this. First, the structure of the raw data and that of metadata and derived data are different—ordered arrays and relations, respectively. Second, processing involves complex operations that go beyond linear and relational algebra. As a result, it is common in scientific domains to have an infrastructure consisting of two separate systems—one for raw data and one for metadata and derived data. The drawbacks of such a solution are evident—two systems to manage, data residing in two separate places, and cross-system data processing at application layer, to name a few.

To illustrate this situation, we provide a motivating example from astronomy. The *Palomar Transient Factory (PTF)* project [1] aims to identify and automatically classify transient astrophysical objects such as variable stars and supernovae in real-time. High-resolution images of the night sky represent the raw data. The first step in the processing pipeline is to extract a set of astrophysical objects from each image and to store them together with the image metadata into a relational database. The images are stored separately and processed outside of the database due to their ordered array structure—not a good fit for the unordered relational data model. In subsequent pipeline stages, the bulk of the processing is executed at the application layer and involves extracting the image metadata and the identified objects from the database, reading the actual image from a different storage source – typically a file on disk – executing application code across the raw image and the corresponding objects, and storing newly generated results into the database. Similar situations are encountered in other scientific projects, for example *Sloan Digital Sky Survey (SDSS)* [2] and the data vaults example presented in [13].

SSDBM '13, July 29 - 31 2013, Baltimore, MD, USA

Contributions. EXTASCID (**EXT**ensible system for **A**nalyzing **SCI**entific **D**ata) evolves around two fundamental design objectives: *completeness* and *extensibility*. EXTASCID is a complete system with native support both for array data as well as for relational data. EXTASCID provides unlimited extensibility by making the execution of arbitrary user code a central part of its design through the well-established User-Defined Aggregate (UDA) mechanism. As a result, EXTASCID supports in-database processing of full scientific workflows over both raw and derived data.

EXTASCID is built around the massively parallel GLADE [8] architecture for data aggregation. While it inherits the extensibility provided by the original UDA interface implemented in GLADE, EXTASCID enhances this interface considerably with functions specific to scientific processing. This requires significant extensions to the GLADE execution strategy in order to provide additional flexibility and to optimize array processing. The design of the EXTASCID parallel storage manager with native support for relations and arrays is entirely novel—GLADE works only for relational data. As far as we know, this is the first storage manager with native support for relations and arrays in the literature.

Given its descent from GLADE, EXTASCID also satisfies the standard requirements for scientific data processing—support for massive datasets and parallel processing. Contrary to existent scientific data processing systems designed for a target architecture, typically shared-nothing, EXTASCID is *architecture-independent*. It runs optimally both on shared-memory, shared-disk servers as well as on shared-nothing clusters. The reason for this is the exclusive use of thread-level parallelism inside a processing node while process-level parallelism is used only across nodes.

Related work. EXTASCID is part of a long series of parallel systems for scientific data processing. Titan [6] and T2 [7] are the first systems designed with extensibility in mind. They adopt an execution strategy closely related to the Map-Reduce [10] paradigm. More recently, SciHadoop [5] implements array processing on top of the popular Hadoop Map-Reduce framework. The main differ-

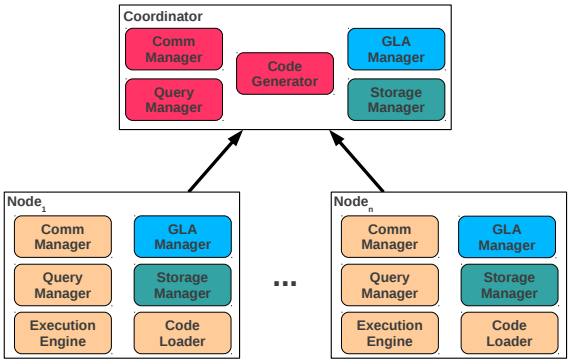

Figure 1: EXTASCID system architecture.

ences between EXTASCID and these systems are the different execution strategy, i.e., UDA vs. Map-Reduce, and the native support for arrays and relations in EXTASCID. RasDaMan [4] is a general middleware for array processing with array chunks stored as BLOBs in a back-end database. The processing is specified through a limited number of second-order operators integrated into SQL and executed entirely inside the middleware. RasDaMan is targeted only at array data – relational data have to be processed either at the application level or inside the middleware – and it is not parallel. The RAM [17] and SRAM [9] systems provide support for array processing on top of the MonetDB [12] columnar database. They do not provide native support for arrays since arrays are represented as relations and array operations are mapped over relational algebra operators. RAM and SRAM are not parallel. SciQL [18] is an array extension to SQL integrating relations and arrays into the same declarative language. While the language is well-defined through examples across multiple application domains, there is little known about its implementation beyond that it is based on MonetDB. SciDB [16] is the system EXTASCID resembles the most. Both are parallel systems designed to be extensible. SciDB supports natively only arrays. EXTASCID provides native support both for arrays and relations. The execution strategy in EXTASCID is well-defined through the UDA interface which makes reasoning about parallelism clear. The same is not true in SciDB where a series of User-Defined Functions (UDF) are arbitrarily interconnected.

2. SYSTEM DESIGN

In a nutshell, EXTASCID is a parallel data processing system that executes any computation specified as a Generalized Linear Aggregate (GLA) [8] using a merge-oriented execution strategy supported by a push-based storage manager. The storage manager is designed with special consideration for multi-dimensional range-based data partitioning in order to support efficient array processing. To allow for wide extensibility in terms of the supported user code and to extract maximum performance, GLAs are dynamically compiled inside EXTASCID at runtime following the optimized code generation mechanism proposed in the DataPath system [3].

As shown in Figure 1, EXTASCID consists of two types of entities: a coordinator and one or more executor processes. The coordinator is the interface between the user and the system. Since it does not manage any data except the catalog metadata, the coordinator does not execute any data processing task. These are the responsibility of the executors, typically one for each physical processing node. It is important to notice that the executors act as completely independent entities, in charge of their data and of the physical resources. The coordinator as well as the executors consist of multiple components, depicted in Figure 1. While the components are inherited from the GLADE [8] architecture, significant enhancements are required to the Storage Manager and GLA Manager modules in order to support extensible array storage and processing in addition to the native relational data model.

2.1 Storage Manager

The storage manager is responsible for organizing data on disk, reading, and delivering the data to the execution engine for processing. There are multiple aspects that distinguish EXTASCID from traditional database storage managers. First, it supports natively relational data as well as multi-dimensional arrays. Second, and most important, the storage manager operates as an independent component that reads data asynchronously and pushes it for processing. It is the storage manager rather than the execution engine in control of the processing through the speed at which data is read from disk. And third, in order to support a highly-parallel execution engine consisting of multiple execution threads, the storage manager itself uses parallelism for simultaneously reading multiple data partitions.

Chunking or *tiling* is the name used for range-based partitioning in the context of multi-dimensional arrays [14]. Essentially, the array dimensions are used as partitioning attributes and the resulting data segments are called chunks. Possible chunking strategies for arrays are presented in [11, 15]. Issues that need to be addressed include the shape of the chunk, the order in which to store the chunks on disk, and how to distribute chunks across processing nodes.

The shape of the chunk can be fixed across the entire array – regular chunking – or there can be multiple shapes, each of them containing the same number of array cells—irregular chunking [11, 15]. Regular chunking is better suited for dense arrays, also known as grids, since each cell in the array contains the same data. The main issue with regular chunking is how to determine the optimal shape. The immediate alternative is to make the size of the chunk along each dimension proportional to the domain size of the corresponding dimension—aligned tiling [11] uses the same scaling factor across each dimension. Another alternative is to determine the shape based on the query workload as the solution to the optimization formulation that minimizes the overall number of chunks read from disk [14]. Irregular chunking is better suited to sparse arrays. The objective is to create chunks that contain the same number of data points rather than to have chunks with the same shape. This results in similar processing time across chunks and load balancing across processes—an important aspect for parallel processing. EXTASCID supports all these types of chunking. It chooses the appropriate strategy based on the type of data and other available information such as the query workload.

Once the chunk shape is determined, two additional problems require attention—how to order the chunks on disk and how to distribute the chunks across multiple processing nodes. It is important to notice that no matter what order is chosen, there will be tasks with suboptimal performance. Thus, the idea is to optimize the placement for a given workload or in the average case. Random placement of chunks on disk and across nodes is optimal in the average case. When workload information is available, the order of chunks on disk – the order in which dimensions are considered – can be chosen such that chunks that are accessed together are placed continuously on disk. This results in larger sequential scans and fewer seeks, thus better I/O performance. Larger chunk sizes have a somehow similar effect. The assignment of chunks to nodes involves a more complicated tradeoff. On one side, we aim for maximum parallelism. On the other, the amount of data trans-

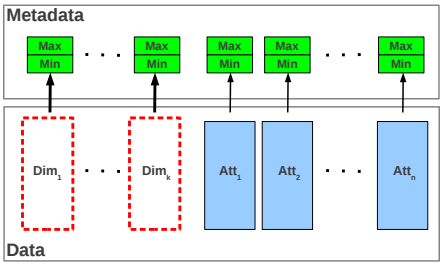

Figure 2: Chunk structure.

ferred between nodes has to be minimized. Thus, it is not clear if chunks that are accessed together should be assigned to the same or different nodes. It depends on the actual task to be executed. The problem becomes even more complicated in EXTASCID due to the thread-level parallelism inside each processing node. In this situation, we opt for random chunk placement on disk and random chunk assignment to processing nodes as our default strategy. The user is given the possibility to change this and specify an arbitrary placement though.

Figure 2 depicts the generic structure of an EXTASCID chunk containing metadata to support range-based data partitioning. It is important to point out that this structure is directly applicable both to arrays as well as relational data. The metadata contain the minimum and maximum values for each dimension and attribute and are stored in the system catalog. This information is computed at loading time – it does not require maintenance since the data are read-only – and it is sufficient to determine if a chunk is required for processing in a subsample query. The actual data are vertically partitioned, with each column stored in a separate set of disk blocks. This allows only for the required columns to be read for each query, thus minimizing the I/O bandwidth required for processing.

Given the generic chunk structure, it is important to determine what optimizations can be applied for different types of data. We are specifically interested in sparse and dense ordered arrays and unordered relations. While in the case of sparse arrays and relations there is not much beyond using the metadata to determine if a chunk is required for processing in a subsample or selection query, dense arrays provide further optimization opportunities. To be precise, the dimensions can be discarded altogether if the data inside the chunk are stored sorted along a known order of the dimensions. This optimization is known as *dimension* or *index suppression* and can reduce the amount of data read from disk even further. Notice that although index suppression reduces the amount of stored data, we do not consider it as a compression method. Compression techniques are orthogonal to chunk organization and they can be applied at column level. Currently, EXTASCID does not support compression.

2.2 GLA Execution

EXTASCID adopts a merge-oriented execution strategy, facilitated by the push-based storage manager and the GLA interface for complex task specification. Merging is supported by two components of the system—a GLA metaoperator that is part of the execution engine and the GLA manager. As all the other operators in the execution engine, the GLA metaoperator takes as input chunks. Unlike other operators though, its functionality is not restricted to a pre-determined template with a reduced number of configuration parameters. Instead, the GLA metaoperator can execute arbitrary user code as long as it is expressed using the GLA interface [8]. The role of the GLA manager is to merge together GLAs created

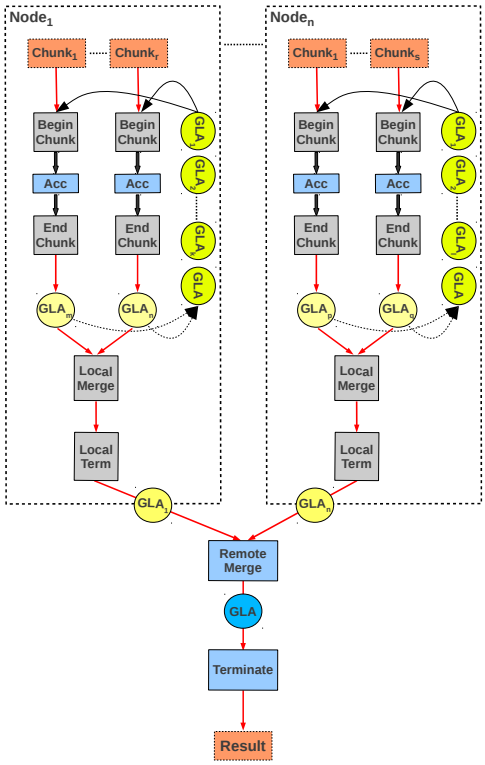

Figure 3: EXTASCID merge-oriented execution strategy. The gray rectangles correspond to methods specific to array processing in the extended GLA interface.

at different nodes. Merging is executed on arbitrary tree structures, determined independently for each query.

Figure 3 depicts the stages of the merging strategy expressed in terms of the extended GLA interface specific to array processing. The semantic of the standard GLA methods is presented elsewhere [8]. In the following, we focus on the array-specific GLA methods we propose starting from the PTF use case. `BeginChunk` is invoked before the data inside the chunk are processed, once for every chunk. `EndChunk` is similar to `BeginChunk`, invoked after processing the chunk instead. These two methods operate at chunk granularity. They are the places where side-effect operations are executed. For example, in `BeginChunk` data can be sorted according to a dimension that makes the processing more efficient. In `EndChunk`, data that are part of the GLA state and do not require further merging can be materialized to disk resulting in significant reduction in memory usage. Merging is invoked in two places. In the GLA metaoperator, `LocalMerge` puts together local GLAs created on the same processing node, while in the GLA manager `RemoteMerge` is invoked for GLAs computed at different nodes. This distinction provides optimization opportunities depending on the chunking strategy—when chunks corresponding to the same array are stored on the same node, only `LocalMerge` is required. `Terminate` is called after all the GLAs are merged together in order to finalize the computation, while `LocalTerminate` is invoked after the GLAs at a processing node are merged. `LocalTerminate` allows for optimizations when the processing is confined to each node and no data transfer is required. It is important to notice that not all the interface methods have to be implemented for every type of processing.

3. SYSTEM IMPLEMENTATION

EXTASCID is implemented as a GLADE [8] extension. The storage manager and the GLA metaoperator are the components that require significant re-implementation efforts since in GLADE relations are fully scanned from disk for every query and operators do not consider the order when iterating over tuples. The EXTASCID storage manager has to support arrays and range-based data partitioning. Arrays are stored using the generic chunk structure depicted in Figure 2. Range information is included in the dimension metadata. It is used to determine which chunks have to be processed for a query given a range condition on dimensions. This is an array-specific optimization – a lightweight form of spatial indexing – to reduce the amount of data read from disk. Dimensional range pruning is implemented using the same principles of dynamic code generation and compilation present in the execution engine since it is a runtime operation specific to every query and it depends on the data at each node. The implementation of the GLA metaoperator is enhanced with calls to the methods in the extended GLA interface (Figure 3). To optimize execution based on the internal chunk organization, the order of the cells, e.g., row-major or column-major, and their absolute coordinates have to be known in the GLA methods. They are available in the chunk metadata and can be extracted by the GLA metaoperator in `BeginChunk`. Moreover, even the chunk organization can be altered in `BeginChunk` prior to iterating over the array cells.

4. THE DEMONSTRATION

Demo participants are presented the entire process EXTASCID performs for the detection and classification of astrophysical transient objects in the PTF project[1]. The input to the entire process is represented by a series of reference images of the sky and a set of newly acquired images. The goal is to identify and classify astrophysical objects visible in the new images and inexistent in the corresponding reference image in real-time. Since the images have high resolution and the processing is complex, extensive use of parallelism is required to make this possible, as reflected by the deep pipeline consisting of multiple applications that is currently in place. In our solution, the entire pipeline is executed inside EXTASCID, thus reducing the management complexity considerably. The GLA implementation and the execution inside EXTASCID are shown for each task in the pipeline. The following processing steps are presented to the demo audience:

- *Image subtraction and calibration*. A new image is generated by subtracting the reference image from the newly acquired image. Then it is calibrated using a series of complex image processing algorithms. This is an array operation that preserves the original chunking. It is implemented entirely inside EXTASCID by parallelizing the execution across multiple chunks of the same image and across multiple images.
- *Object identification*. Candidate objects are extracted from the subtracted images using a parametrized clustering-based algorithm operating at array cell level. The resulting candidates and their corresponding properties are stored as relations in EXTASCID. Identification takes as input a chunked array and generates as output a partitioned relation. It is a perfect representative for the merge-oriented processing strategy implemented in EXTASCID.
- *Object classification*. The set of transient objects determined at the previous step is further processed to identify and classify only the real objects. This is a multi-stage operation over relational data that consists in selecting only those candidate objects appearing in multiple images. Once identified, a scoring function is applied to compute the likelihood that the object is real.

A demo participant can execute the above tasks on a set of real PTF images based on a pre-defined script. Starting with a set of sky images, the participants experience each step of the processing, observe the intermediate results, and obtain the most likely transient objects in the end. Participants can also customize different parameters of the processing during the demonstration. For example, they can choose the set of images and regions where to search for transient objects. Or they can test the effect different data partitioning and chunking strategies have on the execution.

A second demo scenario explores the EXTASCID storage manager. Specifically, we evaluate possible array implementations inside a parallel relational storage manager. The first alternative, and most common, is to represent arrays on top of relations. Dimensions have to be represented explicitly in this case since there is no ordering information available. The second alternative is to support arrays directly by preserving the order—the EXTASCID solution. Dimensions can be discarded altogether in this case since the ordering information is implicit. Demo participants are shown the effects each representation has on data organization, I/O throughput, and execution time.

[1]The authors would like to thank Kesheng Wu and Peter Nugent from Lawrence Berkeley National Laboratory for introducing the PTF project and providing the dataset.

5. REFERENCES

[1] Palomar Transient Factory. `www.astro.caltech.edu/ptf`.
[2] Sloan Digital Sky Survey. `www.sdss3.org`.
[3] S. Arumugam and al. The DataPath System: A Data-Centric Analytic Processing Engine for Large Data Warehouses. In *SIGMOD 2010*.
[4] P. Baumann and al. The Multidimensional Database System RasDaMan. In *SIGMOD 1998*.
[5] J. B. Buck and al. SciHadoop: Array-based Query Processing in Hadoop. In *SC 2011*.
[6] C. Chang and al. Titan: A High-Performance Remote Sensing Database. In *ICDE 1997*.
[7] C. Chang and al. T2: A Customizable Parallel Database for Multi-Dimensional Data. *SIGMOD Rec.*, 27(1), 1998.
[8] Y. Cheng and al. GLADE: Big Data Analytics Made Easy. In *SIGMOD 2012*.
[9] R. Cornacchia and al. Flexible and Efficient IR using Array Databases. *VLDBJ*, 17, 2008.
[10] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1), 2008.
[11] P. Furtado and P. Baumann. Storage of Multidimensional Arrays Based on Arbitrary Tiling. In *ICDE 1999*.
[12] S. Idreos and al. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Eng. Bull.*, 35(1), 2012.
[13] M. Ivanova and al. Data Vaults: A Symbiosis between Database Technology and Scientific File Repositories. In *SSDBM 2012*.
[14] S. Sarawagi and M. Stonebraker. Efficient Organization of Large Multidimensional Arrays. In *ICDE 1994*.
[15] E. Soroush and al. ArrayStore: A Storage Manager for Complex Parallel Array Processing. In *SIGMOD 2011*.
[16] M. Stonebraker and al. The Architecture of SciDB. In *SSDBM 2011*.
[17] A. R. van Ballegooij. RAM: A Multidimensional Array DBMS. In *EDBT Workshops 2004*.
[18] Y. Zhang and al. SciQL: Bridging the Gap between Science and Relational DBMS. In *IDEAS 2011*.