EXTASCI: An Extensible System for Scientific Data Analysis

Yu Cheng, Florin Rusu

Email: *ycheng4@ucmerced.edu, frusu@ucmerced.edu*

Introduction & Background

Data generated through scientific experiments and measurements has unique characteristics that require special processing techniques. The size of the data is extremely large, with terabytes of new data generated daily. The structure of the data is diverse, not only relational, but rather multi-dimensional arrays with different degrees of sparsity. Analyzing scientific data requires complex processing that can be seldom expressed in a declarative language such as SQL.

EXTASCI

Architecture

EXTASCI is a scalable distributed system for large scale scientific data processing. It uses a shared-nothing architecture which is shown in Figure 1. Typically, EXTASCI is deployed over a cluster of computers containing their local storage. There is a master node that is not only responsible for managing the system metadata, but is also in charge of the communication between other nodes when running a query. Computer nodes are independent from each other, but when running a query, the nodes compose a dynamic hierarchy structure (parent and children) cluster, shown in Figure 2, based on their computing resource. The distributed nodes can easily register in or log out from the cluster.

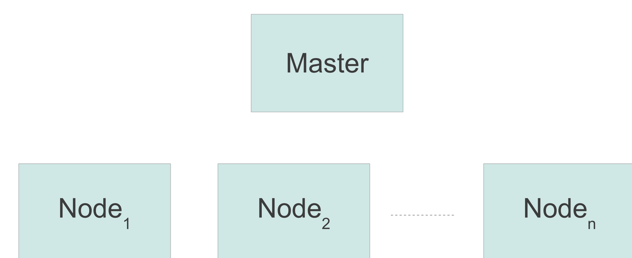

Fig 1. Initial Structure

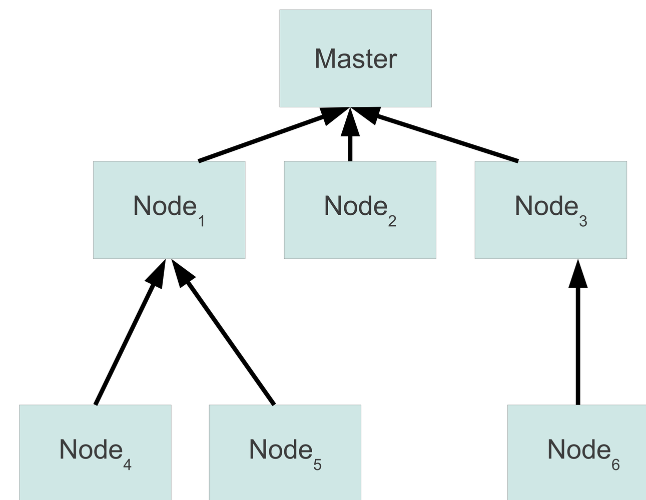

Fig 2. Hierarchical Structure

Storage Manager

- `Array Model`
  Arrays are a natural data model for a significant subset of science users specifically astronomy, oceanography, fusion, and remote sensing. For example, astronomy scientists use high-resolution images in their research. Within each image it is almost certain to have a large number of distinguishable features corresponding to light from objects such as stars, galaxies, and nebula. These images are all typically 2-D arrays, shown in Figure 3. In EXTASCI, the storage manager is responsible for storing the data in the proper format. It supports two types of data models: relational model and array model. Users can easily choose the array model at the data loading step by defining the dimensions and attributes, then the system can automatically load the data in the right format.
- `Chunk Data`
  Chunking is a partition technique mainly used for the array data model. Each chunk contains a horizontal partition of the data with the same order inside, as represented in Figure 4. In EXTASCI, each node stores the subset of the whole array data locally and executes the queries using the local data which makes the queries run in parallel. Further, inside each chunk, data is stored column-oriented to vertically partition the attributes, which decreases the amount of data to be read when executing queries requiring a small number of attributes.
- `Index`
  One chunk can be viewed as an independent data unit which is a subset of the whole array. In many cases, scientific users only care about some special regions in which some attribute is above a given threshold. In order to decrease the amount of transferred data, we design an index for every chunk which contains the minimum and the maximum value for each dimension and attribute. When running queries, we can omit the chunks not satisfying the selection condition.

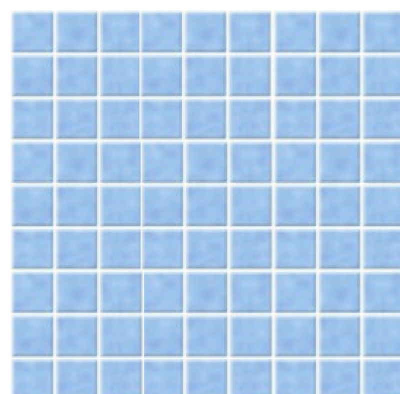

Fig 3. Array Structure

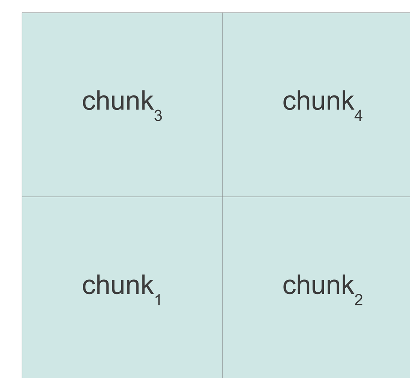

Fig 4. Chunk Structure

User-Defined Aggregates (UDA)

EXTASCI provides an abstract interface that enables users to implement complicated operations easily. Users do not need to care about parallelism/distribution and I/O issues like disk, memory, and cache. What users need to do is to implement some core API described in the following:

BeginChunk() is called when each chunk is handled for the first time. In this function, users can do some initialization work.

Accumulate() offers an interface to let users handle each tuple from the data easily. For the array data model, the tuples are read in order.

Merge() allows users to do a merging operation between two separate states which is crucial when implementing aggregation operations such Group-By.

EndChunk() is called after a chunk has been processed. It can be used to do ending or cleaning tasks.

SS-DB – Standard Benchmark for Scientific Databases

We evaluate the expressiveness of the task specification interface and the performance of our system by implementing the SS-DB benchmark, a standard benchmark for scientific data analysis. In our poster, we choose to handle the astronomy images acquired using telescopes, to detect and measure star-like and galaxy-like features, as shown in Figure 5. The raw data is represented as 2-D arrays which are all in the same local coordinate system that starts from $(0, 0)$, increases in both dimensions, and ends at $(7499, 7499)$. Each cell in the array contains 11 attributes, $V_1 \ldots V_{11}$, which are 32-bit integers. In these attributes, 5 of them are typical "interesting values" in the total array space with data ("background") of less interest. Each array is approximately *2.48GB* in size.

Moreover all the images are in a global coordinate system, which goes from 0 to 10^8 in each dimension. Each raw image has a start point (I, J) in this coordinate system, which is much larger than can be observed by a practical sensor at a single point in time. So images taken at different times have different starting points, as depicted in Figure 6. The benchmark contains arrays taken at distinct times $T_1, T_2, \ldots, T_n$. These arrays are arranged into cycles according to their acquisition times, with 20 images per cycle, which is similar to the scientific data which is processed at periodic intervals.

Fig 5. Raw Image

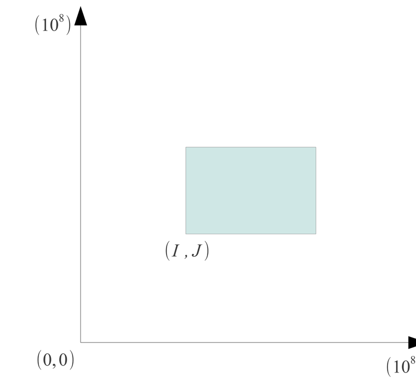

Fig 6. Coordinate System

The benchmark contains both data loading and scientific data analysis tasks as described in the following:

- `Observations`
  The images are all ordered arrays, which are stored in the array model. Also, because there are many operations based on the attributes, we use column-oriented storage. We handle the raw 2-D arrays and produce a collection of observations (stars in astronomy) which contain multiple neighbouring points having high values in some attributes. In order to find an observation, a cell in the array can not be consider alone, we need to consider the points next to it in the other 8 directions, such as up, down, left, right, and so on. Also, we need to calculate some attributes for each observation, such as its center coordinates, polygon boundary, avgdist and pixelsum, calculated from the corresponding points. Observations, being derived from raw image data and represented as a series of sparse 2-D arrays, contain some attributes, are independent with each other, so we represent them as relations.
- `Groups`
  A physical phenomena can exist at different times, in different arrays, because it may be moving or the sensor may be in a different position. So we need to gather all the observations of same physical phenomena in different arrays together. We use the same algorithm, considering two observations B_1 and B_2 at times N_1 and N_2 with centers (C1x, C1y) and (C2x, C2y). These observations cannot be in the same group unless $\sqrt{(C2x - C1x)^2 + (C2y - C1y)^2} < D3 \times (N2 - N1)$. D3 is the maximum allowable velocity in time period T for the objects in the group. Group data are typically relations, so we store them using relational tables. When the groups are formed from observations, scientific users need to retrieve the raw data concerning this group in some specific region for further research. In this situation, we need to search the group data in relation tables and then draw the raw data from array storage, which poses an interesting challenge.

Queries

In addition to observations and groups, the benchmark also contains several queries grouped in three categories:

- Queries on raw data
- Queries on observations
- Queries on groups

All queries are executed on a specific rectangular region defined by a starting point $[U, V]$ and the length len_i in each dimension. There are also other parameters controlling the complexity of the queries such as $D1, D2, \ldots, D6$. We will shortly describe the requirement of each query and our implementation using the core API derived from the UDA interface:

- **Q1 (Aggregation)**
  Compute the average value of V_i for a random i from the 20 images in each cycle. Through this query, we can calculate the average background noise in the raw imagery, for example.

  Implementation:
  BeginChunk() – define the values for $U1, V1$
  Accumulate() – record the sum and count for the attribute
  Merge() – merge the sum and count
  EndChunk() – nothing
  Finalize() – calculate the result

- **Q2 (Recooking)**
  Change the threshold and recook the raw data in a specific region of size $[U1, V1]$ starting at $[X1, Y1]$ and generate a new collection of observations in that region.

  Implementation:
  BeginChunk() – define the values of $U1, V1$
  Accumulate() – ingest the cell according to the former points around, keeping the complete observation in one map and saving the incomplete observations in another map containing the direction index
  Merge() – merge the complete observations map together; merge the incomplete observations in the corresponding direction and move the complete observations out.
  EndChunk() – flush the complete observations out
  Finalize() – flush all the observations in the maps out

- **Q3 (Regridding)**
  Regrid the cells in a specific region size of $[U1, V1]$ starting at $[X1, Y1]$, with a ratio $10 : 3$.

  Implementation:
  BeginChunk() – define the value of $U1, V1$
  Accumulate() – calculate all the griding points value corresponding to this cell; all the griding points is saved in a map using their coordinate as key
  Merge() – merge the map saving the griding points
  EndChunk() – nothing
  Finalize() – flush all the result out

- **Q4 (Observation Aggregation)**
  Calculate the average value of Oi of observations in a region defined by U2, V2 from specific cycle. The handling method of this query is similar to Q1.

- **Q5 (Polygons)**
  Select observations whose polygons overlap the slab size of $[U2, V2]$ starting at $[X1, Y1]$.

  Implementation:
  BeginChunk() – define the value of $U2, V2$
  Accumulate() – check all the observations one by one, using external library to help us doing the overlapping logic; summing up the attributes up whose polygon meets the requirement
  Merge() – merge the sum and count
  EndChunk() – nothing
  Finalize() – calculate the result

- **Q6 (Density)**
  Firstly grouping the observations in the specific region size of $[U2, V2]$ starting at $[X1, Y1]$ spatially into $D4$ by $D4$ tiles, and find tiles containing more than $D5$ observations. The handling method of this query is similar to the Q4.

- **Q7 (Centroid)**
  Find the groups whose center falls in the slab of size $[U2, V2]$ starting at $[X1, Y1]$. This query is similar to Q5.

- **Q8 (Trajectory)**
  For a region of size $[U3, V3]$ starting at $[X2, Y2]$, produce raw data by $D6$ by $D6$ tile centered on each center for all images that intersect the slab. This query contains two stages filtering operation on group in relation model and selecting operation in array model. For the first stage, we can simply implement this similar to Q5, then we could get all the centers. In the second stage, we need to go over all the images to select proper raw data one by one which is time-consuming process. we can use the meta data of minimal value and maximal value in a chunk, to sharply decrease the number of images we need to handle. Further, we run as many queries as possible parallel which typically 55 queries at the same time, to increase the capability of the system.

  Implementation:
  BeginChunk() – define the value of $U2, V2, X2, Y2$, and $D6$. Load all the centers into system.
  Accumulate() – create a map for each tile using their center coordinate as key; when handling an point, add it to the corresponding tile; in order to alleviate the memory size, flush the tile out when it is fully filled
  Merge() – merge the map
  EndChunk() – do nothing
  Finalize() – flushing all the tiles out

Experimental Results

System. We executed all our experiments on a single node having 2 AMD Opteron 8-core processors for a total of 16 cores running at $2GHz$, $16GB$ of memory, 4 $1TB$ hard-drives, and runs Ubuntu 10.10 64-bit.

Data. We use the image generator from SS-DB to generate the small size raw data arrays that are $3,750$ by $3,750$ in size, totalling *99GB*. The data contains 160 arrays taken at distinct times $T_1, T_2, \ldots, T_{160}$. These arrays are arranged into 8 cycles according to their taking times, with 20 arrays per cycle.

Parameters.

Parameter	D1	D2	D3	D4	D5	D6	[U1, V1]	[U1, V1]	[U1, V1]
Value	50	50	0.1	100	20	100	3750	10000	1000

Results.
We run each query for 5 times and take the average running time. In order to illustrate the system more accurately, we split the running time into two parts execution time and compile time.

Query	Total Time (seconds)		
	Execution Time	Compilation Time	Other Time
Loading	2,531.00	/	/
Cooking	61.96	212.69	0.01
Grouping	93.00	32.00	0.01
Q1	52.39	175.43	0.01
Q2	2.24	10.27	0.01
Q3	12.73	27.67	0.01
Q4	0.05	2.45	0.01
Q5	0.05	3.61	0.01
Q6	0.03	3.16	0.01
Q7	0.31	3.15	0.01
Q8	39.12	297.13	0.02