Running N-body Use Cases on Myria

by

Lee Lee Choo

Supervised by Magdalena Balazinska

A senior thesis submitted in partial fulfillment of the requirements for the degree of Bachelor of Science With Departmental Honors

Computer Science & Engineering

University of Washington

March 2014

Presentation of work given on Friday, March 21st, 2014

Thesis and presentation approved by __________________________

Date Friday, March 21st, 2014
Running N-body use case on Myria

Lee Lee Choo
Department of Computer Science and Engineering
University of Washington
March 20, 2014

1. Contents

1. Contents
2. Introduction
3. Overview of Myria
4. Overview of the Astronomy Application
5. Data
   5.1. Data for Progenitor Query
       5.1.1. The data for each snapshot
       5.1.2. Groups of interest
       5.1.3. Mass table
6. Running Progenitor Query on SQL Server
   6.1. The given queries
   6.2. The modified queries
7. Running Progenitor Query on Myria
   7.1. The steps to run progenitor query in Myria
   7.2. Progenitor Query in JSON
8. Evaluation of the queries
   8.1. The runtime
   8.2. Future work
9. Building a web gadget ASCOT on Myria
   9.1. The New Gadget
       9.1.1. ASCOT Setup
       9.1.2. Interacting with Myria
       9.1.3. The user interface
       9.1.4. Incorporate the visualization
10. Conclusion
11. Acknowledgement
12. References
13. Appendices
   13.1. Appendix A: Schema for each Snapshot
2. Introduction
In today’s information age, humans generate massive amounts of data. However not all data is created equal. In order to properly quantify data, we use the 3V’s (velocity, volume and variety). Velocity describes the rate of increase of data. Volume describes the raw amount of data. Last but not least, variety is used to describe heterogeneous data.

Unfortunately, simply collecting all the data does not automatically improve the understanding of the subject matter. It is also important to know how do we make full use of the data?

There are people whose full time job is to understand and to analyze the data in order to extract useful insight or pattern to better understand the world. These people are called data scientists. Data scientists use big data systems such as Hadoop, and Spark to help them. There are also commercial solutions such as Amazon’s Redshift and Google’s BigQuery. Some solutions require extensive programming knowledge in order to use. This paper proposes the solution to be Myria, a service from the University of Washington Database Group that lets users be self-sufficient in analyzing the big volume data and to provide efficient, scalable, and reliable big data management as a Cloud service.

Our goal is to show that Myria can perform well in real world situations such as astronomy real use cases. This work makes three research contributions. First, it contributes to the testing and developing of Myria. Second, it helps to benchmark Myria against SQL Server, a commercial database management engine on a real application and real data from the astronomy domain. Third, it shows that developer can easily develop web applications that use Myria as the database management system. Lastly, this project contributes to the astronomy field. The application developed in the context of this project is designed in collaboration with astronomers in the astronomy department. They intend to use the application in their works.

Given that our main goal of the project is to allow astronomers from the UW Astronomy department to efficiently run queries on a large N-body dataset, Myria was selected for its speed and familiarity to the scientists. In order to achieve that on Myria, we had to express queries through query plans in Myria. This is also a great opportunity to truly benchmark Myria’s performance. Finally, in order to complete the link between astronomers and Myria, we implemented a gadget on ASCOT for the uses cases to allow astronomers to interact with the Myria database management system without writing code. An in-depth explanation of ASCOT and the use cases implementation is explained in Section 8. The web interface should allow the astronomers to be self-sufficient in analyzing the n-body data.

3. Overview of Myria
Myria offers big data management as an easy to use Cloud service. It is a parallel
system so it can be scaled out easily.

To use Myria, the first step is to import all the data into Myria. The data is actually being loaded into underlying relational database system. Users can ingest different types of data into Myria using different ways. The different ways are being demonstrated in the sections that explain the setup processes for both use cases. The current underlying relational database system for Myria web interface is PostgreSQL. After data is being ingested successfully, users can either submit query plans directly expressed in JSON or start issuing queries using supported languages to the Myria web UI. Currently, Myria supports query language such as SQL, Datalog and MyriaL (a Pig-Latin like language). The first use case in this paper is the example of issuing query to Myria using JSON query plan and the second use case is the example of issuing query to Myria through the Myria’s web interface. Figure 1 shows the architecture of Myria.
4. Overview of the Astronomy Application

Lauren and Sarah from the Astronomy Department shared with us two use cases that they hope to run on Myria. The data for the use cases that they hope to run is from N-body simulation. An N-body simulation is used to calculate the position of particles as a function of time [1]. One example of N-body simulation can be seen on Figure 2. They provided us with the data for each simulation snapshot. The data in each simulation snapshot is clustered into groups called halos. The ultimate goal is to follow the evolution of these halos. The data consists of several parts: TIPSY format binary files.
and CSV files. TIPSY is a data visualization package that is used to quickly represent and display N-body simulations. TIPSY basic data structure is array. It contains three separate arrays to represent Cosmological N-body simulations particles which are collisionless particles, smoothed-particle hydrodynamics (SPH) particles, and star particles [2]. We are going to describe the data that they shared with us in detail in Section 5.

Figure 2: The visualization of the simulation provided by Lauren which the astronomers hope to load into the database in the future. Lauren and her team refer to this simulation as Vulcan. It is 80 million light years on a side and has high enough resolution to resolve the morphologies of low mass galaxies.

Halo finder on the workflow in Figure 3 is one of the use cases that they shared with us. We refer to it as progenitor use case. Progenitor, in this case is the term they use to describe group that has the most particles in common as the present days halos. Figure 4 is the illustration of finding progenitor from one of the snapshots. This use case involves finding progenitor from 27 snapshots that represents 27 different timesteps of the universe. We explain the conditions that we are using in looking for progenitors in Section 5 and Section 6.
Figure 3: The workflow provided by Lauren that shows where merger trees fit in to the astronomers’ analysis.

Figure 4: An illustration of finding progenitor from timestep(snapshot) j. Image extracted from Jennifer Ortiz’s “Progenitor” presentation slides.
Figure 5: An illustration of merger tree for halo group 16. Image extracted from “Applying Parallel Database Systems in Astrophysics: Creating Galactic Merger Trees using Myria” poster.

We can build a merger tree after we find the progenitors, which make merger tree the second use case that the astronomers shared with us. Figure 5 shows the illustration of a merger tree for a selected halo group from the present day. Astronomers are using a merger tree to describe the evolution of halos. From the workflow in Figure 3, we can see that being able to successfully build the merger tree is an important stepping stone for them to discover more about the universe such as the star formation history, the star formation efficiency, the gas temperature and the morphology.

We are going to explain how we approached the two important use cases in the following sections.

5. Data

5.1. Data for Progenitor Query
There are three parts of the data: files that contain data for each snapshot in TIPSY format, the group of interest file and the mass table file for each snapshot.

5.1.1. The data for each snapshot
There are 27 relations in the dataset. Each relation represents a snapshot of the observable universe at a particular point in time. Each row in the snapshot contains a particle’s attributes. Each particle can either be of type gas, star or dark. The data for each snapshot involves three parts. The first part is called TIPSY data.
The second part is a CSV file that contains iOrder which corresponds to every particle in the TIPSY file. iOrder represents particle’s id that is unique to each particle for all the snapshots. It is possible for a particle to appear in more than one snapshots. We identify the particles that appear in different snapshots through their iOrders.

The third part is a CSV file that contains the group number that corresponds to every particle in the TIPSY file. More than one particle may be in the same group for each snapshot. Particle that is in group 0 means it does not belong to any group. Unlike iOrder, group number for every particle is not constant over time. For example, particle that belongs to group 1 in time step 1 might belong to group 3 in time step 2.

The data is available both in SQL Server (loaded previously by the Myria team) and as a set of files in original format. Due to the sheer volume of the data, it is not possible for us to export them as the form of CSV file from SQL server. In addition to the access to SQL Server that contains N-body tables, we know the location of 27 files in TIPSY format, 27 CSV files that contains iOrders, 27 CSV files that contains group number, and legacy code. We reverse engineered the code and wrote TipsyFileScan operator for Myria. This operator takes in a TIPSY file, an iOrder file and a group number file from each snapshot. It then ingest all the data through round-robin fashion into Myria’s workers.

The schema for each snapshot is [iOrder], [mass], [x], [y], [z], [vx], [vy], [vz], [rho], [temp], [hsmooth], [metals], [tform], [eps], [phi], [grp], [type]. [iOrder] and [grp] are from the CSV files. We inserted the type based on where they are located in the given TIPSY file. Based on the hints from the legacy python code, every TIPSY file follows the specific format. The first few bytes in the TIPSY file tells us what is the total number of particles, what is the number of dark particles, the number of gas particles and the number of star particles. Different types of particles have different attributes. In TipsyFileScan, we union all the attributes into one schema. In the union process, we assigned value 0 for particles that do not have values for some attributes as Myria does not support NULL at that moment. Table 1 shows the attributes for each type of particles in the TIPSY file.

<table>
<thead>
<tr>
<th></th>
<th>mass</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>vx</th>
<th>vy</th>
<th>vz</th>
<th>rho</th>
<th>temp</th>
<th>hsmooth</th>
<th>metals</th>
<th>tform</th>
<th>eps</th>
<th>phi</th>
</tr>
</thead>
<tbody>
<tr>
<td>star</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dark</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>gas</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Table 1: The attributes for each type of particle in the TIPSY file.

However, the attributes that we are most concerned with for the progenitor query are type, grp and mass.

5.1.2. Groups of interest
Groups of interest data is a text file that contains one column: group number. That is a table that represents the halos from present day that the astronomers want to know
about. They are halo groups in cosmo50_00512 (the latest snapshot in the 27
snapshots) that fulfill this condition: 4.0e9 <= \( \text{SUM(mass of star particles)} \) * 1.84793e16 <= 8.0e10 .
We can thus get the groups of interest from Query 1.

```sql
SELECT grp
FROM cosmo50_00512
WHERE grp <> 0
    AND TYPE = 'star'
GROUP BY grp HAVING 1.84793e16 * \text{SUM(mass)} >= 4.0e9
    AND 1.84793e16 * \text{SUM(mass)} <= 8.0e10
ORDER BY grp ASC;
```
Query 1: Computes groups of interest.

We exported the output from the above query as a text file from SQL Server.

5.1.3. Mass table
Mass table data is a text file that contains two columns: the group number and the total
mass of that particular group. In Progenitor query case, there are 27 mass tables, each
correspond to the mass table of one snapshot. The total mass for each valid group is
calculated using this formula: 1.84793e16 * \text{SUM(mass)} AS \text{tot_mass}. Query 2 is the example
to get mass table for cosmo50_00512, substitute cosmo50_00512 to other snapshot
table name to get the mass table for other snapshot.

```sql
SELECT grp,
    1.84793e16 * \text{SUM(mass)} AS \text{tot_mass}
FROM cosmo50_00512
WHERE grp <> 0
GROUP BY grp;
```
Query 2: Computes mass table for a given snapshot.

6. Running Progenitor Query on SQL Server

6.1. The given queries
One of our goals is to benchmark SQL Server against Myria. To achieve this goal, one of
our team members wrote a query that find progenitor from a single snapshot. It first finds
the potential progenitor query and then the progenitors. Note that snapshot from the
present day is being referred to as base snapshot in the queries. Query 3 computes
potential progenitors and Query 4 uses potential progenitors to get the group that
contributes the most particles. The detail explanation of potential progenitor query and
progenitor query are at the section that explains Query 5 and Query 6, which are the
modified version of Query 3 and Query 4. The modification is necessary to find
progenitors for every snapshot.

```sql
SELECT DISTINCT (base_snapshot.grp) AS grp,
    pgtr_snapshot.grp AS pgtr,
    COUNT(*) AS aggr_col
FROM Cosmo50_00512 base_snapshot
JOIN Cosmo50_00512 pgtr_snapshot ON base_snapshot.iOrder = pgtr_snapshot.iOrder
```
JOIN grps_of_interest ON grps_of_interest.grp = base_snapshot.grp
WHERE pgtr_snapshot.grp <> 0 AND base_snapshot.type = 'star'
GROUP BY base_snapshot.grp,
    pgtr_snapshot.grp;

Query 3: Computes potential progenitors by joining base snapshot with base snapshot.

SELECT 0 AS pgtr_time,
    selected_pgtrs.grp AS grp, 
    selected_pgtrs.pgtr AS pgtr,
    pgtr_mass_table.tot_mass AS tot_mass
FROM
(SELECT potential_pgtrs.grp AS grp, 
    MIN(potential_pgtrs.pgtr) AS pgtr
FROM
(SELECT grp, 
    MAX(p.aggr_col) AS max_aggr_value
FROM
(SELECT DISTINCT (base_snapshot.grp) AS grp, 
    pgtr_snapshot.grp AS pgtr, 
    COUNT(*) AS aggr_col
FROM dbo.cosmo50_00512 base_snapshot 
JOIN dbo.cosmo50_00512 pgtr_snapshot ON base_snapshot.iOrder = pgtr_snapshot.iOrder 
JOIN grps_of_interest ON grps_of_interest.grp = base_snapshot.grp 
WHERE pgtr_snapshot.grp <> 0 
    AND base_snapshot.type = 'star'
GROUP BY base_snapshot.grp,
    pgtr_snapshot.grp) p
GROUP BY grp) AS grps_with_max_vals
JOIN
(SELECT DISTINCT (base_snapshot.grp) AS grp, 
    pgtr_snapshot.grp AS pgtr, 
    COUNT(*) AS aggr_col
FROM dbo.cosmo50_00512 base_snapshot 
JOIN dbo.cosmo50_00512 pgtr_snapshot ON base_snapshot.iOrder = pgtr_snapshot.iOrder 
JOIN grps_of_interest ON grps_of_interest.grp = base_snapshot.grp 
WHERE pgtr_snapshot.grp <> 0 
    AND base_snapshot.type = 'star'
GROUP BY base_snapshot.grp,
    pgtr_snapshot.grp) potential_pgtrs ON potential_pgtrs.grp = grps_with_max_vals.grp
AND potential_pgtrs.aggr_col = grps_with_max_vals.max_aggr_value
GROUP BY potential_pgtrs.grp) AS selected_pgtrs
JOIN cosmo50_00512_mass pgtr_mass_table ON selected_pgtrs.pgtr = pgtr_mass_table.grp;

Query 4: Progenitor query for base snapshot joins with base snapshot.

6.2. The modified queries
In order to run the whole use case, we need to join base snapshot with the previous snapshot and that is then joined with its previous snapshot. We combined all the mass tables into a single view called all_mass_tables and assigned each mass table an extra column that indicates their timestamps. For example, the last (base, cosmo50_00512) snapshot has timestamp 0, and the previous snapshot which is cosmo50_00504 has timestamp 1 and so on until the very first snapshot which is cosmo50_00024 having timestamp 26. Similarly, we combined all the snapshots into a single view called
all_snapshots  and assigned each snapshot table an extra column that indicates their timestamps. After that, we modified the potential progenitor query and progenitor query to have an extra join condition based on timestamp. The potential progenitor query is being made into a view, as seen in query 5 to simplify the progenitor query. Being made as a view does not improve the runtime in any way because it is a virtual view and what SQL server does is inserting that part of the query into every section in progenitor query that mentions potpro [3].

```
CREATE VIEW potpro AS
SELECT base.grp AS grp,
    pgtr.all_grp AS pgtr,
    pgtr.time AS time,
    count(*) AS aggr_col
FROM cosmo50_00512 AS base,
    all_snapshots AS pgtr,
    grps_of_interest
WHERE base.grp = grps_of_interest.grp
    AND base.iOrder = pgtr.iOrder
    AND base.type = 'star'
    AND pgtr.all_grp != 0
GROUP BY base.grp,
    pgtr.all_grp,
    pgtr.time;
```

Query 5: A virtual view for potential query that includes every snapshot.

```
SELECT time,
    grp,
    pgtr,
    tot_mass
FROM
(SELECT potpro.grp AS grp,
    potpro.time AS time,
    MIN(pgtr) AS pgtr
FROM
(SELECT grp,
    time,
    max(aggr_col) AS max_aggr_col
FROM potpro
GROUP BY grp,
    time) AS grps_with_max_vals,
potpro
WHERE grps_with_max_vals.grp = potpro.grp
    AND grps_with_max_vals.time = potpro.time
    AND max_aggr_col = aggr_col
GROUP BY potpro.grp,
    potpro.time) AS selected_pgtr
JOIN all_mass_tables ON mass_time = time
AND pgtr = massGRP;
```

Query 6: The progenitor query that includes every snapshot. potpro is the view in query 5.
Figure 6: An illustration of identifying potential progenitors for each timestep/snapshot. Image extracted from Jennifer Ortiz’s “Progenitor” presentation slides.

Figure 7: An illustration of finding progenitors from potential progenitors of each timestep/snapshot. The blue circles are the potential progenitors and the orange circles are the progenitors. Image extracted from Jennifer Ortiz’s “Progenitor” presentation slides.
Potential progenitor query joins base snapshot with all the snapshot which includes itself based on iOrder of the particles. We only care about star particles that are in groups of interest, so we filtered out particles in base snapshot that are not star and do not belong to group of interests. We also filtered out particles in all_snapshots that do not belong to any group because it is not needed for the use case. Note that particles in base snapshot that has the same iOrder as the particles in all_snapshots do not necessarily have the same group number. From Figure 6, we can see that there are multiple potential progenitors from each snapshot, we want to find one progenitor from each snapshot among the potential progenitors, as shown in Figure 7. The whole purpose of the progenitor query is to find group from each snapshot that contributes the largest mass to the latest valid groups in base snapshot. Thus, in the progenitor query, we first find the group with maximum particles in each snapshot from potential progenitor table. All those groups are under grps_with_max_vals variable. After that we join grps_with_max_vals with potential progenitor pot_pro in order to break tie in grps_with_max_vals. There might be more than a group from each snapshot that have the maximum particles, and the astronomers break ties by choosing the smaller group number as in this dataset which is cosmo50; they arranged the groups such that group with smaller number has larger mass than group with bigger number. After that join, we are guaranteed to have only one group with maximum particles from each snapshot under variable selected_pgttr. The final step is to get the total mass for all the group in selected_pgttr. To achieve that, we joined selected_pgttr to all_mass_tables that contains group number and the total number for that group in each snapshot. The main difference of this modification is the extra timestamp, the timestamp allows the query to differentiate the snapshot when joining. The result outputs the timestamp which represents each snapshot, group number in base snapshot, group number (pgtr) in the timestamp that contributes most mass to the particles in the respective base snapshot group number and the total mass of particles in pgtr.

7. Running Progenitor Query on Myria
To communicate with Myria system, we translated SQL queries in Section 5.2 into JSON because that is the only option to issue a query to Myria at that time. In general, users do not need to write their queries in JSON to use Myria as there are easier options which will be explained in Section 8. Unlike running the query in SQL Server that already has the dataset, there are extra steps involved to run the query in Myria and the overall procedure is very different.

7.1. The steps to run progenitor query in Myria
The first step is to setup the cluster by running the given script and provide our deployment.cfg file: /setup_cluster deployment.cfg

deployment.cfg file contains information about our username, the location to place the result on the machines, the master machine, the worker machines and ports involved. We can list as many worker machines as we want. Both Myria and the SQL server is running on similar hardware.

The second step is to launch master and workers: /launch_cluster deployment.cfg
The third step is to ingest the data to all the workers. There are three sets of data to be ingested. The cosmo tables, the groups of interests table and the mass tables. Two different formats are involved in ingesting all the data required for this experiment. First, cosmo tables are in the TIPSY format, therefore we need to tell Myria to use our custom TipsyFileScanOperator to ingest the data. An example of a JSON file used to ingest cosmo data looks like the following:

```
{
  "grp_filename": "/projects/db8/dataset_astro_2011/cosmo50cmb.256g2MbwK.00512.amiga.grp",
  "iorder_filename": "/projects/db8/dataset_astro_2011/cosmo50cmb.256g2MbwK.00512.iord",
  "relation": {
    "program_name": "astro",
    "relation_name": "cosmo512",
    "user_name": "leelee"
  },
  "tipsy_filename": "/projects/db8/dataset_astro_2011/cosmo50cmb.256g2MbwK.00512"
}
```

In Section 5.1.1, we mentioned that cosmo data for each snapshot involved three files. We can see that the JSON file content consists of the three file paths. Note that the file path is the location where the file is located among the workers.

The command to ingest cosmo512 data assumed the JSON file above is named ingest_coso512.json:

```
curl -i -XPOST beijing.cs.washington.edu:8753/dataset/tipsy -H "Content-type: application/json" -d @./ingest_coso512.json
```

beijing is the machine name that we specified as master in the deployment.cfg file and 8753 is also the port number that we specified. tipsy is the keyword that ask the server to parse ingest_coso512.json to TipsyFileScan.

An example of JSON file to ingest a mass table:

```
{
  "relation": {
    "user_name": "leelee",
    "program_name": "astro",
    "relation_name": "masstable512"
  },
  "schema": {
    "column_types": {
      "INT_TYPE",
      "DOUBLE_TYPE"
    },
    "column_names": {
      "grp",
      "tot_mass"
    }
  },
  "file_name": "/projects/db8/leelee_coso_masses/cosmo512_mass.txt",
  "delimiter": ","
}
```
Unlike ingest_cosmo512.json which is using customized TipsyFileScanOperators as the parser, this JSON file content needs to specify the schema and delimiter because the server is using a generic FileScan operator to read non-TIPSY files.

The command to ingest the above mass table assumed the JSON file is named ingest_512mass.json:

curl -i -XPOST beijing.cs.washington.edu:8753/dataset -H "Content-type: application/json" -d @./ingest_cosmo512.json

The command is almost the same as the command to ingest cosmo data except that in this command we exclude the tipsy keyword.

Except for the file name and file path, the JSON file format and the command to ingest groups of interest data are the same as the JSON file and command to ingest a mass table as both of them consist of single file and no special processing is needed on the server side to put the data on Myria workers.

The fourth step is to run the query. The command to run the query:

curl -i -XPOST beijing.cs.washington:8753/query -H "Content-type: application/json" -d @./progenitor.json

progenitor.json in the command is the JSON file that contains progenitor query. We describe the contents of this file in Section 7.2.

The fifth step is to access the result in the assigned worker's database.

Do note that the JSON file format in communicating with Myria might have changed. What we were using might not work with the latest version of Myria.

7.2. Progenitor Query in JSON
A JSON query consists of operator encodings, such as “scan”, a simple operator encoding to more complex operator encoding such as “shuffle”.

Initially, we scan each snapshots and unions them into a variable called UnionAll. When scanning each snapshots, we added a new field that contains their timestamps and also filtered out those that are in group 0. The way to decide what value to be put as the timestamp is the same as Section 6.2. The next step is to scan groups of interest table and base snapshot table. When scanning the base snapshot, we only want the particles that are of type star. Groups of interest table and base snapshot table are then being shuffled to workers based on the group number. After that, we join both of the tables based on group number and named the result BaseGrpsOfInterest. Before each worker join UnionAll and BaseGrpsOfInterest locally based on iOrder, we need to shuffle both tables based on iOrder. Shuffle the result of joining those two relations on BaseGrpsOfInterest’s group number (grp), BaseGrpsOfInterest’s timestamp (time) and
UnionAll’s group number (pgtr). Each worker then groups the result on time, grp and pgtr to produce the count of particles for each grouping (time, grp, pgtr). The result of the group by is the potential progenitor that contains time, grp, pgtr and aggr_col which is the count of all the particles in each grouping of (time, grp, pgtr). Figure 8 shows the simplified query plan for executing potential progenitor query in a non-distributed fashion.

Potential progenitor is then being split into two operators using LocalMultiwayConsumer, namely PotPro1, PotPro2. Shuffle PotPro1 based on time and grp then group it on time and grp to produce the grp that has the maximum number of aggr_col and call the result GrpsWithMaxVals (time, grp, max_aggr_col). Shuffle GrpsWithMaxVals on time, grp and max_aggr_col. Shuffle PotPro2 on time, grp and max_aggr_col. Each worker then join GrpsWithMaxVals with PotPro2 on time, grp, max_aggr_col/aggr_col. The result is then being grouped on grp to get the minimum pgtr from each grouping and called the result SelectedProgenitorGroupedBy. The reasoning behind getting the minimum pgtr is explained in Section 6.2.

Next, scan every mass table and union them into a relation called UnionAllMassTable (time, grp, tot_mass). Similarly to scanning cosmo tables, timestamp is being added to every row in UnionAllMassTable based on the same standard. Shuffle
UnionAllMassTable and SelectedProgenitorGroupedBy on time and grp. Join both the table on time and grp to get the tot_mass for every grp in SelectedProgenitorGroupedBy. Send the result (time, grp, min_pgtr, tot_mass) to worker 1. Figure 9 shows the simplified query plan for executing progenitor query in a non-distributed fashion.

![Query Plan Diagram](image_url)

Figure 9: The query plan for progenitor query in a non-distributed fashion.

Example of one of the operators in Myria JSON API:

```json
{
  "arg_child1":"SC_Scan_base_Rename",
  "arg_child2":"SC4",
  "arg_column11": [
    "1"
  ],
  "arg_column2": [ 
    "0"
  ],
  "arg_select1": [ 
    "0",
    "1"
  ],
  "arg_select2": [ 
    "0"
  ],
  "op_name":"BaseGrpsOfInterest",
}
```
Rename operators are being used to rename certain fields in some relations as Myria requires fields to have unique names when doing a join. The total lines of the JSON file is about 1440.

8. Evaluation of the queries

8.1. The runtime

Figure 1 shows the average runtime of progenitor query on SQL Server and on Myria with 2 workers, 3 workers and 12 workers respectively.

![Progenitor query runtime on SQL Server and Myria](image)

Figure 10: Bar chart showing the average progenitor query runtime on SQL Server and Myria.

From Figure 10, we can see that SQL Server takes 80 minutes to produce the result of the progenitor query. Myria with 2 workers, 3 workers and 12 workers take about 25 minutes, 19 minutes and 4 minutes respectively. The point of the experiment is to prove that Myria is capable of running a real use case query.

In addition, we do not observe a linear relationship between average runtime and the number of workers. If the relationship is linear, we would expect Myria 12 workers to use (25 - (6*10)) minutes to run the query as Myria 3 workers is taking about 6 minutes less in running the query. It makes sense that it is not linear because as we are using more workers, there are more overhead in shuffling the data, especially when there are 10
shuffling processes in this query.

8.2. Future work
Union of the data is messy. We can consider storing all the union in a new table and just start the query from there. That will save us all the time and complexity in partitioning and using a union operator over 27 tables. We can do that in SQL Server by inserting the result of union into a new table. Similarly for Myria, we could run a query to re-partition the data and store it to disk.

Another optimization is to filter out particles that belong to group 0 in the process of producing the union of all snapshots. We will then have less number of rows to join.

In addition, a custom REST API is being created to ingest TIPSY data because Myria is under production so it was the only way to do this task at this moment. Further thoughts should be put on this to generalize data ingestion.

All the code and JSON files are pushed to UW Myria Github with proper documentation. The result of the experiment is also submitted to UW Myria Github Wiki.

9. Building a web gadget ASCOT on Myria
Astronomical Collaborative Toolkit (ASCOT) is a dashboard creation tool for astronomy gadget.

One of our goals is to create an ASCOT gadget that allows astronomers to analyze the merger-tree data without the need for them to code or write query. They just need to select and filter based on the options we provided on the user interface.

The data set that we are using for this use case is the same as that of progenitor query. In addition, Jennifer has created two tables based on the data set to simplify the query for this use case: ParticleTable (iOrder, HaloID, Mass, Type) and HaloTable(HaloID, Timestep, GrpID).

9.1. The New Gadget
9.1.1. ASCOT Setup
Unlike the progenitor use case which we had to write the JSON query manually, we have the option to use MyriaL for this use case. This is because by this time, the Myria team had implemented a web interface located at demo.myria.cs.washington.edu that is capable of taking in a MyriaL query. On top of that, it would be cleaner and easier to use MyriaL.

For this use case, we have ingested the dataset in advance. Users need not worry about the setup at all. There are different steps involved in the set up as compared to the previous use case. The Myria web application runs on a different cluster, the star cluster, separate to where the cosmo50 data is located. Hence, there are extra steps involved in order to ingest the data. First, we have to install Hadoop client on our machine, and
transfer the data to star cluster HDFS. We need to figure out that the master node for that Hadoop cluster is located at vega.cs.washington.edu port 8020. After that, we have to ingest the data using different JSON file format. We were ingesting using file source for progenitor use case, whereas this time we were ingesting using URI at the HDFS. The JSON file format for this type of ingestion:

```json
{
    "relationKey": {
        "userName": "public",
        "programName": "adhoc",
        "relationName": "particlestable"
    },
    "schema": {
        "columnTypes": ["INT_TYPE", "INT_TYPE", "FLOAT_TYPE", "STRING_TYPE"],
        "columnNames": ["iOrder", "haloID", "mass", "type"]
    },
    "source": {
        "dataType": "URI",
        "uri": "hdfs://vega.cs.washington.edu:8020//datasets/mergertree/particlestable.txt"
    }
}
```

9.1.2. Interacting with Myria

In general, it is a breeze to issue queries through the Myria web application. This is because Myria provides REST API for ingesting dataset and issuing queries.

After getting the selection of halo group from user, we will format MyriaL query accordingly and send it to to Myria web application through proper REST API call. Query 7 is the query that we formed when user selected halo group 16. We will then get the query id from the response and use it to check the status of the query from Myria. When the status of the query is “SUCCESS”, a query will be issued to download the result which is being stored at a specific table. Other possible query statuses include “ERROR”, “ACCEPTED” and “RUNNING”. The downloaded result is then being displayed using d3 visualization library.

The whole process of figuring out the appropriate MyriaL query for this use case and issuing query to Myria web-application helped discover new bugs in Myria such as type conversion error and null pointer exception. It also helped push on certain functionality that has not yet existed in Myria before this use case such as ‘CASE WHEN’ operator.

```sql
p1 =
(SELECT p.iOrder, 
p.haloID, 
h1.grp, 
h1.timestep 
FROM SCAN(particlestable) AS p, 
SCAN(halotable) AS h 
WHERE p.haloID = h1.haloID);
p2 =
(SELECT p.iOrder, 
p.haloID, 
p.mass,
```
p.type, h1.grp, h1.timestep
FROM SCAN(particlestable) AS p, SCAN(halotable) AS h1
WHERE p.haloID = h1.haloID;

base =
(SELECT p.iOrder
FROM SCAN(particlestable) AS p, SCAN(halotable) AS h1
WHERE p.haloID = h1.haloID
AND h1.grp = userSelectedGroup
AND h1.timestep = 1
AND h1.);

J =
(SELECT p1.timestep AS timestep,
   p1.grp AS currentTimeGroup,
   p2 grp AS NextTimeStepGroup,
   SUM(p2.mass) AS p2mass,
   COUNT(p1.iOrder) AS cnt,
   SUM(CASE WHEN p2.type = "dark" THEN 1 ELSE 0) AS darkcount
FROM base,
   p1,
   p2
WHERE p1.iOrder = p2.iOrder
AND base.iOrder = p1.iOrder
AND p1.timestep + 1 = p2.timestep;
finalAns =
SELECT *
FROM J
WHERE cnt > 100
   AND darkcount >= 64
   AND p2mass * 184793000000000000.0 > 2940000000.0
   AND p2mass * 184793000000000000.0/4435032038548450.0 > (1/1000);

Query 7: MyriaL query for merger tree use case that produces merger tree edges for halo group 16.

Query 7 on average takes about 6 minutes to execute. We can do better by precomputing the edges for every halo group and store it in Myria. Then we simply need to issue a normal filter query to the said table whenever user asked for the edges of the selected halo group. The improved version takes only a few seconds.

Myria is taking 6 minutes to run the above is because the filter of h1.grp = userSelectedGroup is not being pushed down. We can consider fixing that and get a better runtime.

9.1.3. The user interface
ASCOT is being developed about 3 years ago. There are some design decisions that the developer made which restricted what we can implement and how we can implement. For instance, the interface by default created width and height for each gadget when the page is loaded, so that means we cannot add new components to our
gadget after the width has been assigned, if we did, then we would have to make some adjustments such that overflow gadget can then be scrollable. Then it would complicate the whole design, as the page is scrollable and inside the page there is a gadget that has multiple scrollable components. The workaround is that we created the skeleton of the components that we intend to add later and assign them with reasonable heights and weights. In addition, ASCOT is composed of iframes. Given that every gadget is an iframe, we are restricted to implement components and add functionalities within an iframe.

The current input user interface of our gadget allows user to key in the halo group that he or she is interested in.

After that, the appropriate MyriaL query is formed and sent to the Myria web application. The result we get is displayed as a collapsible tree, as shown in Figure 11 and 12.

![Figure 11: A screenshot of the gadget that we have created.](image)

There are a lot of potentials for output user interface. For instance, we have developed a bubble chart for progenitor query visualization using d3 library on ASCOT gadget just to try it out. The visualization of progenitors can be seen on Figure 13. We have also developed a typical table output format on ASCOT gadget.

Last but not least, there are three extra things that we have added to make the UI more user friendly, that is to display error messages when the input is not valid, display a loading image while waiting for result to load and display an error message if somehow
we failed to get the result.

Figure 12: The close-up of the collapsible tree.
Figure 13: A screenshot of progenitors visualization on the ASCOT gadget that we have created.

9.1.4. Incorporate the visualization
Jennifer and Laurel have their code for the visualization of merger tree for a particular halo group. We took their interaction_collapse.js and make it into a closure, such that we can call the function only when we need it to display result. We also modified some tiny details to make the visualization display properly on our gadget. The other two crucial modifications are changing the code to use raw JSON for the data and create merger tree for any halo groups, not a hard-coded group.

10. Conclusion
Throughout this year long journey, it is pleasing to see that Myria is getting more mature. One year ago, in order to run a query on Myria, we need to write a raw JSON query plan. But now, we can run a query easily by sending a MyriaL query to the Myria web interface. It is fair to say that Myria has improved a lot over the course of one year.

We have achieved our goals by demonstrating that Myria is capable of running real use cases efficiently and it has an excellent runtime when being compared against a commercial database engine. The next step would be to compare Myria against more advanced distributed database systems and let Myria to truly shine in the database world.

Last but not least, with the ASCOT gadget that we have implemented, we have made the
solid first step in allowing astronomers to be self-sufficient in analyzing data. We made it easier for them to discover more about the mysterious universe.

11. Acknowledgement
I would like to thank Professor Magdalena Balazinska for giving me the opportunity to work on this project and overseeing my effort for it. I would also like to thank Daniel Halperin, Director of Research for Scalable Analytics for giving me valuable suggestions, answering my questions and reviewing my code. I would also like to thank each of the graduate student who worked with me on this project, including Jennifer Ortiz, Jingjing Wang, Ryan Mass and Shengliang Xu who answered my Myria related questions, gave me constructive suggestions, met with me to make sure that I am on track, and allowed me to work very independently. I would also like to thank TerChmg Ng for answering my technical questions and Vaspol Ruambivoonsuk for answering my Myria related questions. Not to forget, Lauren Anderson and Sarah Loebman from the Astronomy Department, University of Washington who had worked closely with Jennifer and I on the use cases. Last but not least, I thank the whole Myria team for their contribution and support.

12. References
13. Appendices

13.1. Appendix A: Schema for each Snapshot

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type in SQL Server</th>
<th>Type in Myria</th>
</tr>
</thead>
<tbody>
<tr>
<td>iOrder</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>mass</td>
<td>float</td>
<td>double</td>
</tr>
<tr>
<td>x</td>
<td>float</td>
<td>double</td>
</tr>
<tr>
<td>y</td>
<td>float</td>
<td>double</td>
</tr>
<tr>
<td>z</td>
<td>float</td>
<td>double</td>
</tr>
<tr>
<td>vx</td>
<td>float</td>
<td>double</td>
</tr>
<tr>
<td>vy</td>
<td>float</td>
<td>double</td>
</tr>
<tr>
<td>vz</td>
<td>float</td>
<td>double</td>
</tr>
<tr>
<td>rho</td>
<td>float</td>
<td>double</td>
</tr>
<tr>
<td>temp</td>
<td>float</td>
<td>double</td>
</tr>
<tr>
<td>hsmooth</td>
<td>float</td>
<td>double</td>
</tr>
<tr>
<td>metals</td>
<td>float</td>
<td>double</td>
</tr>
<tr>
<td>tform</td>
<td>float</td>
<td>double</td>
</tr>
<tr>
<td>eps</td>
<td>float</td>
<td>double</td>
</tr>
<tr>
<td>phi</td>
<td>float</td>
<td>double</td>
</tr>
<tr>
<td>grp</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>type</td>
<td>char(4)</td>
<td>string</td>
</tr>
</tbody>
</table>