1. **Sphinx Components:**

1.1 **Data Warehouse (Janguk, Pradeep, Laukik)**

1.1.1 **Policy Tables**

<table>
<thead>
<tr>
<th>ID</th>
<th>Resource</th>
<th>Entity</th>
<th>Property_i</th>
<th>TimeStamp</th>
</tr>
</thead>
</table>

**ID:** INT UNSIGNED NOT NULL PRIMARY KEY

**Resource:** INT UNSIGNED NOT NULL REFERENCES Resource(ID)

**Entity:** INT UNSIGNED NOT NULL REFERENCES Entity(ID)

**Property_i:** VARCHAR(255)

**TimeStamp:** VARCHAR(23) NOT NULL

1.1.2 **Accounting Tables**

<table>
<thead>
<tr>
<th>ID</th>
<th>Resource</th>
<th>Entity</th>
<th>Property_i</th>
<th>TimeStamp</th>
</tr>
</thead>
</table>

**ID:** INT UNSIGNED NOT NULL PRIMARY KEY

**Resource:** INT UNSIGNED NOT NULL REFERENCES Resource(ID)

**Entity:** INT UNSIGNED NOT NULL REFERENCES Entity(ID)

**Property_i:** VARCHAR(255)

**TimeStamp:** VARCHAR(23) NOT NULL

1.1.3 **Resource Property Tables**

<table>
<thead>
<tr>
<th>ID</th>
<th>Level</th>
<th>Name</th>
<th>Num_Machine</th>
<th>Property_i</th>
<th>TimeStamp</th>
</tr>
</thead>
</table>

**ID:** INT UNSIGNED NOT NULL PRIMARY KEY

**Level:** INT UNSIGNED NOT NULL

**Name:** VARCHAR(255) NOT NULL

**Num_Machine:** INT UNSIGNED NOT NULL

**Property_i:** VARCHAR(255)

**TimeStamp:** VARCHAR(23) NOT NULL
### 1.1.4 Resource Hierarchy Table

<table>
<thead>
<tr>
<th>ID</th>
<th>Parent</th>
<th>Child</th>
</tr>
</thead>
</table>

- **ID:** INT UNSIGNED NOT NULL PRIMARY KEY
- **Parent:** INT UNSIGNED NOT NULL REFERENCES Resource(ID)
- **Child:** INT UNSIGNED NOT NULL REFERENCES Resource(ID)

### 1.1.5 Resource Distance Table

<table>
<thead>
<tr>
<th>ID</th>
<th>Source</th>
<th>Destination</th>
<th>Distance</th>
</tr>
</thead>
</table>

- **ID:** INT UNSIGNED NOT NULL PRIMARY KEY
- **Source:** INT UNSIGNED NOT NULL REFERENCES Resource(ID)
- **Destination:** INT UNSIGNED NOT NULL REFERENCES Resource(ID)
- **Distance:** INT UNSIGNED NOT NULL REFERENCES Distance(ID)

### 1.1.6 Distance Table

<table>
<thead>
<tr>
<th>ID</th>
<th>Type</th>
</tr>
</thead>
</table>

- **ID:** INT UNSIGNED NOT NULL PRIMARY KEY
- **Type:** VARCHAR(100) NOT NULL

### 1.1.7 Entity Name Table

<table>
<thead>
<tr>
<th>ID</th>
<th>Level</th>
<th>Name</th>
</tr>
</thead>
</table>

- **ID:** INT UNSIGNED NOT NULL PRIMARY KEY
- **Level:** INT UNSIGNED NOT NULL
- **Name:** VARCHAR(255) NOT NULL
### 1.1.8 Entity Hierarchy Table

<table>
<thead>
<tr>
<th>ID</th>
<th>Parent</th>
<th>Child</th>
</tr>
</thead>
</table>

- **ID**: INT UNSIGNED NOT NULL PRIMARY KEY
- **Parent**: INT UNSIGNED NOT NULL REFERENCES Entity(ID)
- **Child**: INT UNSIGNED NOT NULL REFERENCES Entity(ID)

### 1.1.9 Job-Entity Property Table

<table>
<thead>
<tr>
<th>ID</th>
<th>Job</th>
<th>Entity</th>
<th>Property_i</th>
</tr>
</thead>
</table>

- **ID**: INT UNSIGNED NOT NULL PRIMARY KEY
- **Job**: INT UNSIGNED NOT NULL REFERENCES Job(ID)
- **Entity**: INT UNSIGNED NOT NULL REFERENCES Entity(ID)
- **Property_i**: VARCHAR(255)

### 1.1.10 Job Table

<table>
<thead>
<tr>
<th>ID</th>
<th>DAG</th>
<th>State</th>
<th>ExecSite</th>
<th>MountPoint</th>
<th>Manager</th>
<th>Transfer</th>
<th>TimeStamp</th>
</tr>
</thead>
</table>

- **ID**: INT UNSIGNED NOT NULL PRIMARY KEY
- **DAG**: INT UNSIGNED NOT NULL REFERENCES Dag(ID)
- **State**: TINYINT NOT NULL
- **ExecSite**: VARCHAR(255)
- **MountPoint**: VARCHAR(255)
- **Manager**: VARCHAR(255)
- **Transfer**: VARCHAR(255)
- **TimeStamp**: VARCHAR(23) NOT NULL
### 1.1.11 DAG Table

<table>
<thead>
<tr>
<th>ID</th>
<th>Entity</th>
<th>State</th>
<th>TimeStamp</th>
</tr>
</thead>
</table>

- **ID**: INT UNSIGNED NOT NULL PRIMARY KEY
- **Entity**: INT UNSIGNED NOT NULL REFERENCES Entity(ID)
- **State**: TINYINT UNSIGNED NOT NULL
- **TimeStamp**: VARCHAR(23) NOT NULL

### 1.1.12 Data Table

<table>
<thead>
<tr>
<th>ID</th>
<th>LFN</th>
<th>PFL</th>
<th>Path</th>
</tr>
</thead>
</table>

- **ID**: INT UNSIGNED NOT NULL PRIMARY KEY
- **LFN**: VARCHAR(255) NOT NULL
- **PFL**: VARCHAR(255)
- **Path**: VARCHAR(255)

### 1.1.13 Message Table

<table>
<thead>
<tr>
<th>ID</th>
<th>Direction</th>
<th>Entity</th>
<th>Type</th>
<th>Value</th>
<th>TimeStamp</th>
</tr>
</thead>
</table>

- **ID**: INT UNSIGNED NOT NULL PRIMARY KEY
- **Direction**: BOOL NOT NULL
- **Entity**: INT UNSIGNED NOT NULL REFERENCES Entity(ID)
- **Type**: TINYINT NOT NULL
- **Value**: BLOB NOT NULL
- **TimeStamp**: VARCHAR(23) NOT NULL

### 1.1.14 Relevant Application Tables

Transformation Catalog Snapshot Table: A transformation is an executable program, which is associated with attributes like executable name, location, arguments, environment etc.

Sphinx snapshot table contains Poolname, Logical transformation, physical transformation, env

TC_SNAPSHOT

<table>
<thead>
<tr>
<th>ID</th>
<th>Poolname</th>
<th>Logical transformation</th>
<th>Physical transformation</th>
<th>Environment</th>
</tr>
</thead>
</table>
### 1.1.15 Relevant Data Tables

Replica Catalog Snapshot Table: A replica catalog maintains logical to physical name translation. Sphinx maintains a snapshot of this table with columns: LFN, pool, SFN. Currently, SFN and PFN are same.

<table>
<thead>
<tr>
<th>RC_SNAPSHOT</th>
<th>ID</th>
<th>LFN</th>
<th>Poolname</th>
<th>SFN</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID:</td>
<td>INT UNSIGNED NOT NULL PRIMARY KEY</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LFN:</td>
<td>VARCHAR(255) NOT NULL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Poolname:</td>
<td>VARCHAR(255) NOT NULL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SFN:</td>
<td>VARCHAR(1024) NOT NULL</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Metadata Catalog Snapshot Table: Metadata is data about data. For our purposes, it means, data about files. It includes LFN (will point to RLS snapshot table), file size, creator, timestamp, derivation info (may point to VDS tables)

<table>
<thead>
<tr>
<th>MC_SNAPSHOT</th>
<th>ID</th>
<th>LFN</th>
<th>File size</th>
<th>Creator</th>
<th>TimeStamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID:</td>
<td>INT UNSIGNED NOT NULL PRIMARY KEY</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LFN:</td>
<td>VARCHAR(255) NOT NULL REFERENCES RC_SNAPSHOT(ID)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>File size:</td>
<td>INT UNSIGNED NOT NULL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Creator:</td>
<td>VARCHAR(255) NOT NULL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TimeStamp:</td>
<td>VARCHAR(23)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The MCS from Pegasus can be used to obtain information. Currently, MCS provides a C API for accessing the service. Eventually, it will provide a SOAP interface.

Data Access Pattern Table: This table contains the data access patterns that can be used by DMC for making optimal choices. For accurate data transfer prediction, DMC needs to know about time spent in 'real data transfer' (wire performance), 'disk I/O' (both at client and server side) and past transfer times. Another aspect is to maintain access statistics. I am proposing two tables. One is a snapshot of complete monitoring information maintained by monitoring component. The table contains: LFN, Client site, server site, data transfer performance (Mbps/sec), client disk I/O, server disk I/O. Note that same sites can have many rows and same LFN can have multiple rows describing performance.
### MONITOR_SNAPSHOT

<table>
<thead>
<tr>
<th>ID</th>
<th>LFN</th>
<th>Client Site</th>
<th>Server Site</th>
<th>Bandwidth</th>
<th>Client disk I/O BW</th>
<th>Server disk I/O BW</th>
</tr>
</thead>
</table>

**ID:** INT UNSIGNED NOT NULL PRIMARY KEY  
**LFN:** VARCHAR(255) NOT NULL REFERENCES RC_SNAPSHOT(ID)  
**Client Site:** VARCHAR(255) NOT NULL  
**Server Site:** VARCHAR(255) NOT NULL  
**Bandwidth:** INT UNSIGNED NOT NULL  
**Client disk I/O BW:** INT UNSIGNED NOT NULL  
**Server disk I/O BW:** INT UNSIGNED NOT NULL

Second is the access statistics table. It contains: LFN, no. of times accessed in past k time

### STATISTICS

<table>
<thead>
<tr>
<th>ID</th>
<th>LFN</th>
<th>NACCESS</th>
</tr>
</thead>
</table>

**ID:** INT UNSIGNED NOT NULL PRIMARY KEY  
**LFN:** VARCHAR(255) NOT NULL REFERENCES RC_SNAPSHOT(ID)  
**NACCESS:** INT UNSIGNED NOT NULL

The snapshot tables are purged over time and filled with new entries. Ideally, we would like to have all the information related to current dags and jobs. If a relevant entry is not present and a sphinx component requires it, a trigger will be generated and the snapshot is populated.

### 1.1.16 Job Tracking Tables

<table>
<thead>
<tr>
<th>CondorID</th>
<th>JobID</th>
<th>RunTime</th>
</tr>
</thead>
</table>

**CondorID:** INT UNSIGNED NOT NULL PRIMARY KEY  
**JobID:** INT UNSIGNED NOT NULL references JobTable(ID)  
**RunTime:** Varchar(32)
1.1.17 Grid Weather Tables

The resources to be monitored would be architected in a hierarchical fashion and would be similar to the hierarchy of the resource properties. The following tables would be used to identify the resources (by resources here we mean grid (eg. USCMS), domain (e.g. UF), sites, clusters and machines etc.)

**SPH_TB_RESOURCE**

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
</table>

ID: INT UNSIGNED NOT NULL PRIMARY KEY
Name: VARCHAR(255) NOT NULL

**SPH_TB_RESOURCE_HIERARCHY**

<table>
<thead>
<tr>
<th>ID</th>
<th>Parent</th>
<th>Child</th>
</tr>
</thead>
</table>

ID: INT UNSIGNED NOT NULL PRIMARY KEY
Parent: INT UNSIGNED NOT NULL REFERENCES SPH_TB_RESOURCE (ID)
Child: INT UNSIGNED NOT NULL REFERENCES SPH_TB_Resource (ID)

(We have similar tables used by Jang and given the same architecture (hierarchy) of the grid, same tables can be used.)

Then each parameter that is to be monitored can be identified using the following table:

**SPH_TB_GW_PARAMETER_RESOURCE**

<table>
<thead>
<tr>
<th>ID</th>
<th>ParamName</th>
<th>ResourceID</th>
</tr>
</thead>
</table>

ID: INT UNSIGNED NOT NULL PRIMARY KEY
ParamName: VARCHAR(32) NOT NULL
ResourceID: INT UNSIGNED NOT NULL REFERENCES SPH_TB_RESOURCE (ID)
Then the history of each of the parameter will be maintained in the following tables:

**SPH_TB_GW_PARAM_HIST_ID*_LEVEL*_**

<table>
<thead>
<tr>
<th>Value</th>
<th>Timestamp</th>
</tr>
</thead>
</table>

Value: INT UNSIGNED NOT NULL  
Timestamp: VARCHAR(32) NOT NULL

**1.1.18 Schedule Tables**
1.2 Control Process

1.2.1 Modules which change Job State

<table>
<thead>
<tr>
<th>Job Control Processes</th>
<th>Initial state</th>
<th>Final state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Interface</td>
<td>unpredicted</td>
<td>unpredicted</td>
</tr>
<tr>
<td>Job Predictor</td>
<td>unpredicted</td>
<td>unaccepted</td>
</tr>
<tr>
<td>Job Admission Controller</td>
<td>unaccepted</td>
<td>unplanned</td>
</tr>
<tr>
<td>Job Execution Planner</td>
<td>unplanned</td>
<td>unsent</td>
</tr>
<tr>
<td>Job Submitter</td>
<td>unsent</td>
<td>unfinished</td>
</tr>
<tr>
<td>Job Tracker</td>
<td>unfinished</td>
<td>remove</td>
</tr>
</tbody>
</table>

1.2.1.1 Message Interface (done)
This module receives an abstract DAG from the Sphinx Client, puts the DAG in the DAG table, decomposes the DAG into abstract jobs and puts all jobs into the Job table. The DAG is given an initial state of unreduced and each job is given a state unpredicted.

1.2.1.2 Job Predictor (place holder for now)
This module operates on any job which has state unpredicted AND for which the corresponding DAG state is unpredicted. The module estimates the resource requirements of the job based upon historical information and/or user provided requirements. The module then changes the state of the job to unaccepted.

1.2.1.3 Job Admission Controller (Janguk)
This module operates on any job which has state unaccepted AND for which the corresponding DAG has state unaccepted. The module utilizes the users requested QoS, predicted job resource usage, policy criteria, Sphinx schedule for the k time steps, etc and determines if the job is allowed to execute and if so, with what estimated QoS. The job state is then changed to unplanned.

1.2.1.4 Job Execution Planner (Janguk)
This module operates on any job which has state unplanned AND for which the corresponding DAG has state unfinished. Based upon the users requested QoS, predicted application resource usage, policy criteria tables, grid weather tables, resource availability tables, availability of input data tables, etc. the job is resolved onto physical resources. For data transfer jobs (created by the DAG Data Planner), the job is similarly resolved onto physical resources. The job state is then changed to unsent.

1.2.1.5 Job Submitter (Janguk)
This module operates on any job which has state unsent. The module creates a DAG of tasks containing any necessary environment setup tasks, input data transfers, publication of input data transfer to DMC, application execution, output data transfers, publication of output data transfer to DMC, and cleanup of the execution site. The job (i.e. DAG of tasks) may correspond to an application or it may correspond to a data transfer. The module then submits the job to Condor-G/DAGMan and changes the state of the job to unfinished.

1.2.1.6 Job Tracker (Laukik)
This module operates on any job which has state unfinished. The module monitors the status of each task in the job (“waiting”, “running”, “held”, “completed”) and records the start/end time and resource usage for each task in the Job Table. Once a job has completed (i.e. the last task has completed), any resource usage summary information is transferred to the Accounting and Application Tables. The state of the job is then changed to remove.
1.2.2 Modules which change DAG State

<table>
<thead>
<tr>
<th>DAG Control Processes</th>
<th>Initial state</th>
<th>Final state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Interface</td>
<td>unreduced</td>
<td>unpredicted</td>
</tr>
<tr>
<td>DAG Reducer</td>
<td>unreduced</td>
<td>unpredicted</td>
</tr>
<tr>
<td>DAG Predictor</td>
<td>unpredicted</td>
<td>unaccepted</td>
</tr>
<tr>
<td>DAG Admission Controller</td>
<td>unaccepted</td>
<td>unplanned</td>
</tr>
<tr>
<td>DAG Data Planner</td>
<td>unplanned</td>
<td>unfinished</td>
</tr>
<tr>
<td>DAG-Job Tracker</td>
<td>unfinished</td>
<td>remove</td>
</tr>
</tbody>
</table>

1.2.2.1 Message Interface (done)
This module receives an abstract DAG from the Sphinx Client, puts the DAG in the DAG table, decomposes the DAG into abstract jobs and puts all jobs into the Job table. The DAG is given an initial state of unreduced and each job is given a state unpredicted.

1.2.2.2 DAG Reducer (Pradeep)
This module operates on any DAG, which has state unreduced, queries a Replica System for the existence of any required input data for any job in the DAG, and removes all jobs for which their output data already exists. The module then changes the state of the DAG to unpredicted.

1.2.2.3 DAG Predictor (place holder for now)
This module operates on any DAG which has state unpredicted AND for which all corresponding jobs have state unaccepted. The module estimates the resource usage of the DAG, based upon the resource usage prediction for each individual job. The module then changes the state of the DAG to unaccepted.

1.2.2.4 DAG Admission Controller (Janguk)
This module operates on any DAG which has state unaccepted AND for which all corresponding jobs have state unplanned. If any job has state remove, then the module rejects the DAG and changes the state of the DAG to remove, otherwise the module changes the state of the DAG to unplanned.

1.2.2.5 DAG Data Planner (Pradeep)
This module operates on any DAG, which has state unplanned. The module examines the I/O needs of each job and determines if any data replication jobs are needed. Each data replication job is created and entered into the Job Table (corresponding to the DAG) with job status unplanned. The module then changes the state of the DAG to unfinished.

Note that, the data replication jobs may also be replication jobs for executables. The replication job may or may not contain specific locations to replicate data. Currently, the planner will decide the replication locations. How many nodes to create, where in the dag to create is an open research problem.

1.2.2.6 DAG-Job Tracker (Laukik)
This module operates on any DAG which has state unfinished AND for which all corresponding jobs have state remove. The module simple changes the DAG state to remove.
### 1.2.3 Example for a 2 node DAG

<table>
<thead>
<tr>
<th>Control Process</th>
<th>Initial DAG</th>
<th>Initial Job</th>
<th>Final DAG</th>
<th>Final Job</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Interface</td>
<td>unreduced</td>
<td>predicted</td>
<td>unreduced</td>
<td>unpredicted</td>
</tr>
<tr>
<td>DAG Reducer</td>
<td>unreduced</td>
<td>predicted</td>
<td>unreduced</td>
<td>unpredicted</td>
</tr>
<tr>
<td>Job Predictor</td>
<td>predicted</td>
<td>predicted</td>
<td>unpredicted</td>
<td>accepted</td>
</tr>
<tr>
<td>Job Predictor</td>
<td>predicted</td>
<td>predicted</td>
<td>unpredicted</td>
<td>accepted</td>
</tr>
<tr>
<td>DAG Predictor</td>
<td>unpredicted</td>
<td>accepted</td>
<td>accepted</td>
<td>accepted</td>
</tr>
<tr>
<td>Job Admission Controller</td>
<td>accepted</td>
<td>accepted</td>
<td>accepted</td>
<td>unplanned</td>
</tr>
<tr>
<td>Job Admission Controller</td>
<td>accepted</td>
<td>accepted</td>
<td>accepted</td>
<td>unplanned</td>
</tr>
<tr>
<td>DAG Admission Controller</td>
<td>accepted</td>
<td>unplanned</td>
<td>unplanned</td>
<td>unplanned</td>
</tr>
<tr>
<td>DAG Data Planner</td>
<td>unplanned</td>
<td>unplanned</td>
<td>unfinished</td>
<td>unplanned</td>
</tr>
<tr>
<td>Job Execution Planner</td>
<td>unfinished</td>
<td>unplanned</td>
<td>unfinished</td>
<td>unsent</td>
</tr>
<tr>
<td>Job Execution Planner</td>
<td>unfinished</td>
<td>unplanned</td>
<td>unfinished</td>
<td>unsent</td>
</tr>
<tr>
<td>Job Execution Planner</td>
<td>unfinished</td>
<td>unplanned</td>
<td>unfinished</td>
<td>unsent</td>
</tr>
<tr>
<td>Job Execution Planner</td>
<td>unfinished</td>
<td>unplanned</td>
<td>unfinished</td>
<td>unsent</td>
</tr>
<tr>
<td>Job Submitter</td>
<td>unfinished</td>
<td>sent</td>
<td>unfinished</td>
<td>unfinished</td>
</tr>
<tr>
<td>Job Submitter</td>
<td>unfinished</td>
<td>sent</td>
<td>unfinished</td>
<td>unfinished</td>
</tr>
<tr>
<td>Job Submitter</td>
<td>unfinished</td>
<td>sent</td>
<td>unfinished</td>
<td>unfinished</td>
</tr>
<tr>
<td>Job Submitter</td>
<td>unfinished</td>
<td>sent</td>
<td>unfinished</td>
<td>unfinished</td>
</tr>
<tr>
<td>Job Tracker</td>
<td>unfinished</td>
<td>finished</td>
<td>unfinished</td>
<td>remove</td>
</tr>
<tr>
<td>Job Tracker</td>
<td>unfinished</td>
<td>finished</td>
<td>unfinished</td>
<td>remove</td>
</tr>
<tr>
<td>Job Tracker</td>
<td>unfinished</td>
<td>finished</td>
<td>unfinished</td>
<td>remove</td>
</tr>
<tr>
<td>Job Tracker</td>
<td>unfinished</td>
<td>finished</td>
<td>unfinished</td>
<td>remove</td>
</tr>
<tr>
<td>DAG Tracker</td>
<td>unfinished</td>
<td>remove</td>
<td>remove</td>
<td>remove</td>
</tr>
</tbody>
</table>
1.3 Automated Data Management Component (Pradeep)

This component automatically replicates data across the Sphinx domain according to access patterns. It uses all the three sources (transfer time, disk i/o, past info) for predicting times and then tries to replicate data to reduce this time. Also, data is replicated without considering job handling in sphinx. Meaning, DMC just looks at the grid, data locations and does the hot-spot management.

DMC input tables: TC, RC, MCS, Data Access Pattern table (internal), Dag Table
DMC output tables: TC snapshot, RC snapshot, MCS snapshot, Updated data access pattern table, Updated Dag Table

I am listing two algorithms here. One is specific and will be implemented in first version. The second one is generic and tries to do management on the basis of some rules.

from the job table find the files transferred in last k steps. /* k will be decided by a heuristic */
for each file in the list
  do
    query data access pattern (DAP) table to find out no.of accesses in last k steps.
    if(n_accesses > HIGH_NACCESS) /* this data is quite hot */
      then
        create replication job
        add it to job table
        update status of the job to unplanned
        update the DAP to reflect that the data is being replicated
      end
  end

Second algo:

User/administrator or Sphinx or Sphinx DMC write rules for data management. These rules are specified in the code for now. Later we can create a simple rule language. The rules would be some thing like, if a job named ‘FFT’ comes in get executables for ‘FFT’ for at specific sites or replicate the executables in at least 3 sites. Another rule might be, as soon as a user logs in, pre-fetch particular data. He/She usually works on that data.

for each rule in the list
  do
    create a job describing the rule
    add it to job table
    update status of the job to unplanned
  end
1.4 Information Gathering Component (Laukik)

1.4.1 Job Tracking (view generation)

The function of this module is to track the status of the Sphinx jobs that are submitted (i.e. status = unfinished) and change the status to removed on completion. The association between the DAGs and their corresponding Jobs and then messages and condor-ids can be understood from the following figure:

DAG-ID $\rightarrow$ (1:n) JOB-ID $\rightarrow$ MESSAGE $\rightarrow$ (1:n) CONDOR-IDs

That is, when the Job Submitter module submits a job, it is submitted as a dag (a job consists of env, stage-in, exe, stage-out, clean-up etc) using the condor_submit_dag command. The Job Submitter module will have to register the CONDOR-Ids of the job that it has submitted with the JOB-ID.

This registration needs to be stored in a database table so that this job tracking module can track the condor-ids and so the corresponding jobs can be tracked.

1.4.2 Grid Weather (view generation)

This module will act as an interface between the underlying (possibly multiple) grid weather services.

**Start-up**

At start-up, all the GW tables are created and initialised as per the configuration files.

```plaintext
Start-up
{
    /* Read configuration file */
    /* set up the resource hierarchy */
    /* Identify the parameters to be monitored at each resource and populate the parameter table */
    /* Create necessary PARAM_HIST tables for each entry in the parameter table */
    /* Deploy information gatherer threads - one for each of the resources */
    /* Deploy time-smoothening threads */
}
```

**Multi-Level History**

The PARAM_HIST tables are multi-level. Each next level values are derived by aggregating values over time from lower level tables. So the levels are nothing but indicative of the time hierarchy such as sec, minutes, hours, days, weeks etc.

So, a level 0 table for a parameter may contain values sampled at every second, a level 1 table for this parameter may contain values for every minute, then level 2 for every hour and so on.

A folded time-series can be obtained for a parameter by considering a few starting values from tables at each level.
e.g. 20 entries from 1-sec table, 10 entries from the 1-min table, 4 entries from the 1-hour table etc. This table would thus have logarithmic sorts of scale as we proceed down the number of rows and could be helpful in prediction purposes.

Run Time

At run-time, this module would populate the level-0 tables for each parameter.

The information-gatherer threads would operate at various layers of the underlying monitoring system, read the monitored data values and populate the appropriate database tables.

e.g. The thread deployed as an information gatherer for a node at layer 0 in GEMS would collect the raw data at the predefined intervals (the interval would be the same as the interval in the level-0 data base table entries).

Pseudo-code

/* Instantiate an object of the monitoring interface for a particular layer and an instance in that layer */
while (true)
{
   /* for each parameter do the following: */
   {
      /* obj.getParameter () */
      /* populate this parameter in the database table */
   }
}

Similarly, information gatherer threads deployed at layer 1 in GEMS would collect the data fused by the aggregation functions in GEMS and populate the table (level –0 itself) for the (parent) resource in the hierarchy.

The time-smoothening threads would act on the lower level data tables to produce the higher-level data tables in real time.

The algorithms to be used for this operation needs further research.

1.4.3 Resource Properties (view generation)

The Resource properties are going to be much “static” as compared to the other parameters that are going to be monitored like the CPU-speed and MEM-FREE.

So getting information about the resource properties at the frequency of monitoring these other resources would be an overhead.

So we have 2 options:
Monitor these resource properties at a lower frequency (using the same monitoring system)
Use a different system (which is more efficient for this purpose; like the Glue Schema by MDS)

A quick solution would be to monitor these resource properties like other parameters.

Note: Parameters currently supported by the GEMS version that I have:
- LoadAvg
- MemFree
- SwapFree
- NetActivity
- DiskActivity
- PageActivity

In the long run, this module will make use of the MDS-Glue schema to get the information about the resource properties.
More exploration needs to be done in this area:
http://www.globus.org/mds/glueschema.html

1.5 Appendix: Pseudo-code

1.5.1 Job Admission Controller (Janguk)

```plaintext
Job_Admin_Controller() {
    /**
     * Sphinx Control module executes this function when there is a job with the state
     * unaccepted in Job table.
     **/
    Query Job table for all unaccepted jobs;
    FOR each unaccepted job i
    DO
        /* Resource Capacity Constraint */
        FOR each resource in Site table
        DO
            Calculate the load when the job is assigned to the resource;
            /**
             * This calculation is based on predicted job resource usage and the
             * current load of the resource
             **/
            Determine if the resource will be overloaded with this job;
            /**
             * This determination is based on the pre-defined capacity watermark
             **/
            END

            IF all the resource will be overloaded
            THEN
```
The state unaccepted will be remained. The state change of this job will be evaluated in the next controller cycle. 

/* QoS-Reservation Constraint */

/**
Determine estimated QoS of system utilizing the users requested QoS, predicted job resource usage, policy constraint and schedule.
**/

/**
Generate an outgoing message
Send the message to the job submitter
**/

Make an outgoing message with the estimated QoS;
Query an Outgoing Message table for inserting the outgoing message;

END

} // END of Admin_Controller()

Job_Admin_Controller (Job ID, Message){

/**
Incoming message module calls this function with a job ID and a message. 
The message is response to the estimated QoS that has been sent previously.
**/

Query Job table to get the job with the ID;

SWITCH (Message)

CASE accept:
Query Job table for changing the state of the job to unplanned;
Break;

CASE reject:
/**
This case will remove this job and the DAG that has the job. 
Removing a DAG means that deleting all the jobs in the DAG.
**/
Query Job table for changing the state of the job to remove;
Break;

CASE tryagain:
/**
The job submitting entry wants the admonition controller re-
evaluate the QoS. Remain unaccepted state of the job. QoS of the 
job will be re-evaluated in the next controller cycle.
**/
Break;

END
}

1.5.2 Job Execution Planner (Janguk)

Policy_Based_Planner {}{

/**
Decide a job to be planned in the next planner cycle according to Data Dependency Constraint, generate a property vector of the job, and find a submitter of the job
**/
Query Job table for all unplanned jobs;
FOR each unplanned job i
DO
  FOR each input data
Query RLS table for data availability;

IF Data Dependency Constraint is satisfied
THEN
Generate job property vector from Job-Entity property table;
Find the entity that submitted it from a Job-Entity property table;
GOTO Scheduling;
END

/**
All the unplanned jobs are not ready to be planned in the current planning cycle.
Skip this planning cycle.
**/
EXIT();

Scheduling:

/**
Update a policy constraint matrix based on policy information from a policy
constraint table. A system administrator or policy maker updated the table with
new policy in the unit of time defined in time policy. The policy is created on the base of resource property information from Resource
Property table.
**/
IF policy constraint matrix exists
THEN
  Compare timestamps of the matrix and Policy Constraint table;
  IF timestamps are not matched
  THEN
    Update policy constraint matrix based on policy information;
    Match the two timestamps;
  ELSE
    Update policy constraint matrix with the data from Account table
  ELSE
    Create policy matrix based on the information from the Policy table
    Match the two timestamps;

/**
Find a feasible solution set of resources for the job satisfying policy
constraints
**/
Construct a set of linear equations representing Policy Constraint;
Solve the equations subject to Job Requirement Constraint;
Generate a feasible solution matrix from the solution;

/**
Find one out of resources in the feasible solution set, which provides an optimal
solution to an objective function
**/
FOR each resource r indexed in the feasible solution matrix
DO
  Find a solution to a pre-defined objective function;
END

Find the optimal solution to the function; Ties are broken by taking the first
one;
Set a final assignment variable with the resource providing the solution;

/**
Decide replication location of remote input data consulting with RLS table
**/
FOR each input data i
DO
  Query RLS table with available locations for replication;
  Select a location that has the closest replica
  Set input_rep[i] with the location
END
1.5.3 Job Submitter (*Janguk*)

**Submission Generator()**

Sphinx control module calls this function when there is a job with the state of unsent in Job table.

Query **Job table** for checking out all the jobs in the state of unsent;

FOR each unsent job

DO

/* DAG submission file creation */

Create a DAG submission file of the job workflow and a submission file of each subtask in the DAG based on pre-defined templates;

/*
The job has concrete information of execution site and input/output data replication locations.
The DAG consists of seven subtasks; startup, input, execsetup, job, output, cleanup, register
*/

/* Submit workflow */
Submit the DAG submission file using 'condor_submit_dag'

Listen to the output from the command
Listen to any errors from the attempted command

END

1.5.4 Job Tracker (*Laukik*)

The job-tracking module would have the following logic:

/* For every job in the job table having status unfinished, do the following: */

/* For every CondorJob corr to this JobID */

/* Query the status. Condor_history command can be used for this purpose*/
/* update runtime in the database table */

if condor status = removed or complete

{ /* change status to removed */
}
else

{ /* set flag to signal job not completed */
}

/* if job_not_completed flag is not set, set the status of the Sphinx as removed and update the job statistics */

Note: As per our discussion on Friday, both removed and complete statuses of a condor job are considered to be *removed* for our job as it cannot be determined whether a job has really succeeded.
1.5.5 DAG Reducer (*Pradeep*)

select first unreduced DAG  
for each jobID in DAGID  
do  
  read output files list  
  for each output file in list  
  do  
    check RC for file  
    if file exists  
    then  
      filecount=filecount+1  
    else  
      continue  
    end  
  if filecount=number of output files  
  delete job ID from DAG ID  
end  
set DAG state to UNPREDICTED

1.5.6 DAG Admission Controller (*Janguk*)

DAG_Admin_Controller(){  
  Query DAG table to check out all the DAG’s in the state of unaccepted;  
  FOR each unaccepted DAG  
  DO  
    Query Job table with the DAG ID to check out all jobs belong to the DAG  
      FOR each job  
      DO  
        Check the state of the job;  
        IF the state is equal to remove  
        THEN  
          /**  
          If a single job in a DAG is removed then the whole DAG is removed  
          **/  
          Query DAG table to change the DAG state to remove;  
        END  
        Query Job table to change the state to remove;  
      END  
    END  
  END  
  Query DAG table to change the DAG state to UNPLANNED;  
}

1.5.7 DAG Data Planner (*Pradeep*)

select first unplanned DAG  
for each jobID in DAGID  
do  
  read input files list  
  for each input file in the list  
  do  
    query RLS snapshot table and/or RLS to find out no.of locations the input  
    is available  
    if (nlocations < LMARK_NLOCATIONS)  
      /* LMARK_NLOCATIONS a heuristic for deciding  
      whether to add nodes or not */  
    then  
      decide whether to specify output location or
if(specify_output_location == true)
then
    create a replication job with output location specified.
else
    create a replication job with no output location /* planner will decide the location later in planning state */
end
add the job to job table
update the job state to unplanned
else
    /* do nothing */
end /* each input file for loop */
end
update the DAG state to unfinished.

1.5.8 DAG-Job Tracker (Laukik)
This module will do the following operations:

1. It will look for DAGs that are in unfinished state
2. For any DAGs that are unfinished, it will do a look-up for the corresponding JOBs.
3. These Job-Ids are then used to look up their status.

(If we use 2 states for DAG – INCOMPLETE and COMPLETE and 2 separate states for JOBs – REMOVED and COMPLETE)

4. If any of the corresponding jobs are REMOVED, then
   4.1 The status of this DAG is changed to INCOMPLETE

5. If ALL the corresponding jobs have status COMPLETE, then
   5.1 The status of this DAG is changed to COMPLETE

(If we use just 1 final state for DAG – REMOVED and 1 final state for JOBs – REMOVED)

6. If ALL the corresponding jobs have status REMOVED, then
   6.1 The status of this DAG is changed to REMOVED