UNIVERSITY OF CALIFORNIA,
IRVINE

# Hyracks Console:
# A Visual UI for Monitoring the Hyracks Runtime Platform

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Computer Science

by

Ching-Wei Huang

2011

# TABLE OF CONTENTS

Page

1 INTRODUCTION ........ 1

2 HYRACKS OVERVIEW ........ 2

2.1 Hyracks Architecture ........ 3

2.1.1 Cluster Controller (CC) ........ 3

2.1.2 Node Controller (NC) ........ 4

2.1.3 Initialize Hyracks ........ 4

2.2 Hyracks Jobs ........ 5

2.2.1 Hyracks Job Specification ........ 7

2.2.2 Hyracks Job Plan ........ 9

2.2.3 Hyracks Runtime Tasks ........ 10

3 CONSOLE DESIGN GOALS ........ 12

3.1 Hyracks End Users ........ 12

3.2 Hyracks Operator Implementers ........ 13

4 HYRACKS CONSOLE ARCHITECTURE ........ 14

4.1 Hyracks Console Server (HCS) ........ 15

4.2 Hyracks Console Visualization (HCV) ........ 17

Page

5 IMPLEMENTATION ........................................18

5.1 REST API for Requesting the Hyracks Monitoring Information ........................................19

5.1.1 Hyracks Jobs........................................19

5.1.2 Hyracks Cluster Health ........................................20

5.2 Data Representation Mechanism ........................................21

5.3 Data Representation Modules........................................22

5.3.1 Hyracks Jobs........................................22

5.3.2 Hyracks Cluster Health ........................................38

5.4 Publish/Subscribe API for Server Pushing Method........................................50

6 HYRACKS CONSOLE USER MANUAL ........................................53

6.1 Setting up the Hyracks Console........................................53

6.1.1 Hyracks Cluster Installation Prerequisites for Collecting and Storing Data...........53

6.1.2 Installing JSP ........................................53

6.1.3 Running the Hyracks Console on a Browser........................................54

6.2 Hyracks Console Web UI........................................58

6.2.1 Hyracks Job Browser ........................................60

6.2.2 Hyracks Job Profile........................................61

6.2.3 Hyracks Cluster Browser ........................................68

Page

6.2.4 Hyracks Node Controller Profile ..... 71

7 CONCLUSION ..... 73

# LIST OF FIGURES

Page

Figure 1 : Hyracks System Architecture ....................................3

Figure 2: Initialize a Hyracks Cluster ....................................5

Figure 3: Representations for a Hyracks Job ....................................6

Figure 4 : Hyracks Job Plan ( HAN Graph) ....................................8

Figure 5 : Hyracks Job Specification (HOD Graph) ....................................8

Figure 6 : Hyracks Runtime Tasks (HONs Graph) ....................................11

Figure 7: Hyracks Console Architecture ....................................15

Figure 8 : Hyracks Console Visualization Web Server ....................................17

Figure 9 : URL-Request Paths for Hyracks Jobs Execution Information ....................................19

Figure 10 : URL-Request Paths for Monitoring Hyracks Cluster ....................................20

Figure 11 : Data Retrieval and Representation ....................................21

Figure 12 : Procedure for Generating a Hyracks Job Summary ....................................24

Figure 13 : Example Job Specification Graph Node in the DOT Language ....................................26

Figure 14 : Edge of the Job Specification Graph in the DOT Language ....................................27

Figure 15 : The Procedure in Generating the Job Specification Graph ....................................28

Figure 16 : Node of the HAN Graph in DOT Language ....................................30

Page

Figure 17 : Sub-graph (Hyracks Stage) of the HAN graph in the DOT Language and Example Result....................................................................................................................31

Figure 18 : Example Blocking Edge of the HAN Graph in the DOT Language..........................32

Figure 19 : Example of the HAN Graph in DOT Language ......................................................33

Figure 20 : Procedure for Generating Hyracks Activity Node Graphs.......................................34

Figure 21 : Job Pie Chart and Job Time Chart (1) .....................................................................36

Figure 22 : Job Pie Chart and Job Time Chart (2) .....................................................................37

Figure 23 : Procedure for Generating the CC Configuration Table.............................................39

Figure 24 : Example List of Registered Nodes Component.......................................................41

Figure 25 : Hyracks Jobs Timeglider .........................................................................................43

Figure 26 : Procedure in Generating the NC Configuration Table.............................................45

Figure 27 : Node Controller Job Summary Module ...................................................................47

Figure 28 : NC Resource Monitoring Module............................................................................49

Figure 29 : /jobs channel ............................................................................................................51

Figure 30 : /stages/<job-id> channel...........................................................................................52

Figure 31 : /attempts/<job-id> channel .......................................................................................52

Figure 32 : Initial Web Page (Job Browser Page) of the Hyracks Console .................................55

Figure 33 : Setup the Connection between the HCV and the HCS .............................................55

Page

Figure 34 : Setup the *cc-IP* and *cc-http-port* values ....56

Figure 35 : Example Job Browser Page ....57

Figure 36 : Hyracks Console Visualization Sitemap ....59

Figure 37 : Example Result After Deploying a Hyracks Job ....60

Figure 38: Overview of the Job Profile Page ....61

Figure 39 : Hyracks Job Specification (left) and Hyracks Activity Node Graph (right) ....62

Figure 40 : Hyracks Activity Node Graph of the Completed Hyracks Job....64

Figure 41: Updating the Hyracks Activity Node Graph on the Model Popup Page....64

Figure 42: HAN Profile of the "FileScan" Activity (left) and the "HashBuildActivityNode" Activity (right) on the Model Popup Page ....65

Figure 43: Hyracks Activity Node Graph with Data Movement Information on the Popup Page66

Figure 44 : Performance Pie Chart....67

Figure 45 : Performance Time Chart ....67

Figure 46 : CC Configuration for "vanilla.ics.uci.edu" Hyracks Cluster....68

Figure 47 : Hyracks Jobs Timeglider....69

Figure 48 : Only the Failure Jobs in the Hyracks Jobs Timeglider ....70

Figure 49 : Job's Description in the Hyracks Jobs Timeguilder ....70

Figure 50 : Registered NCs (left) and NC's CPU Utilization Legend (right)....71

Page

Figure 51 : Hyracks Node Controller Configuration of the Node-ID “nc2” ................................72

Figure 52 : Node Controller Jobs Summary ....................................................................72

Figure 53 : Hyracks Machine Resources Consumption ..............................................................73

# ACKNOWLEDGMENTS

I would like to express my gratitude to many individuals, as the Hyracks Console could not have been built without them. I would like to thank my advisor, Michael Carey. Without his help, suggestion, and encouragement, exploring and mastering the world of the Hyracks console would have been difficult to achieve.

Through this research, I have discovered a new topic that has fascinated me which is web UI development. I would also like to thank RaresVernica, Raman Grover, Alexander Bem, Nicola Onose, and Yingyi Bu, the members of the Hyracks and ASTERIX group at University of California, Irvine, for their assistance with all the levels of this research project. I also would like to thank Vinayak Borkar in particular for the direction and feedback that he provided all along.

Last but not least, I must acknowledge my partner and friend, Siripen Pongpaichet, for always being great company throughout this project, and for our implementation debates, exchanges of knowledge, skills, and venting of frustration during our research process, all of which helped to enrich the experience.  Like the wise men say, "Education is all the matter of building bridges", and now I'm ready to cross.

# ABSTRACT OF THE THESIS

Hyracks Console:

A Visual UI for Monitoring the Hyracks Runtime Platform

By Ching-Wei Huang
Master of Science in Computer Science
University of California, Irvine, 2011
Professor Michael J Carey Irvine, Chair

High-level programming frameworks for data intensive processing are being increasingly adopted, as they provide a simple programming interface allowing programmers to create and run data-intensive applications on commodity hardware. Unfortunately, monitoring and debugging applications in this framework is far more complicated than for sequential programs, and often leads to much difficulty and time loss on the part of the programmer. This difficulty arises not only from the complex global state of data-intensive applications but also from the nondeterministic nature of parallel systems. The core challenges involved in creating the *Hyracks Console* were: to identify the information useful to the users, gathering this data at runtime, and ultimately, presenting the data in a visual and scalable representation for the user. This paper covers in detail the design and implementation of the *Hyracks Console*, and it explains how the Hyracks Console can be used to monitor and understand not only the parallel processes of the applications but the *Hyracks* partitioned-parallel runtime platform itself.

# 1 INTRODUCTION

In recent years, the world has seen an explosion of information due to the growth of the Internet. At the same time, due to the declining cost of hardware, companies are now able to set up large clusters of independent computers to store and process the growing data. The emergence of high-level programming platforms for large-scale distributed systems, such as *Apache Hadoop* [1], *Google MapReduce* [2], and *Microsoft Dryad* [3], have led to a lot of followers in the development of the data-intensive applications. The successes of these high-level programming platforms are due to the fact that they conceal the complexity of the underlying distributed systems from the programmer by providing a simple interface through which programmers can parallelize common data-intensive tasks.

Unfortunately, understanding and monitoring these programming platforms is still a difficult task for most of the programmers. In order to diagnose an application failure, users must understand the internal structure of their distributed jobs and the mappings between their internal job's execution and their platforms distributed architecture. A number of ways to solve this problem have been proposed, such as a local debugging mode in *Pig* [4] and *DryadLINQ* [5], and a powerful set of monitoring and debugging tools provided in *Cloudera* [6] and *Karmasphere* [7]. In addition, *Chukwa*, a data collection system for monitoring and analyzing large distributed systems [8] is used to monitor Hadoop clusters. Chukwa also includes a set of powerful toolkits for displaying, monitoring and analyzing result, in order to make the best use of the collected data. *Hyracks* is a new partitioned-parallel software platform designed to run data-intensive computations on large shared-nothing clusters of the computers [9]. However, it does not have any existing or in-built tools for debugging and monitoring. Consequently, Hyracks users might face challenges in understanding and analyzing their Hyracks job execution process. Therefore, in order to speed up the development process and minimize the effort from the users, an efficient monitoring tool with visualization support for the Hyracks system has been developed.

In this thesis, we present the design and implementation of a new "*Hyracks Web-based Console".* Our ultimate goal is to build an interactive tool set that can help Hyracks users reduce the time and effort required for understanding, implementing and optimizing their Hyracks jobs.

Since humans are highly visually-sensitive creations, our monitoring system is intended to present the internal process of the Hyracks jobs' execution and the status of Hyracks clusters, in a visual way. All of the source code for this project will be available in a forthcoming release of Hyracks [10]. The rest of this thesis is organized as follow. Section 2 provides a quick overview of the Hyracks computational model using a simple example query. Section 3 discusses the goals and requirements of the Hyracks console. Section 4 presents the system architecture and design. Section 5 explains the implementation of the Hyracks console. Section 6 provides the Hyracks console's user manual including the system installation and the user guide. Finally, Section 7 summarizes the thesis and discusses our ideas for the future improvement.

# 2 HYRACKS OVERVIEW

Hyracks is a flexible software platform for data-intensive processing tasks that runs on shared-nothing hardware clusters. Large data computations can be dealt with by partitioning the data in a parallel fashion. Hyracks is based on a richer model than MapReduce or Dryad because it extends the limited model of MapReduce and supports a variety of common data communication patterns and operators. A built-in collection of operators and connectors can be used to create Hyracks jobs without being limited them to only Map and Reduce functions. Hyracks also includes a Hadoop compatibility layer that allows the user to execute existing Hadoop jobs without any changes. This section sets the stage for explaining the Hyracks Console by reviewing the main features of Hyracks.

The two essential components of the Hyracks architecture are the *Node Controller* and *Cluster Controller*. The main function of the node controller is to manage the processing of tasks on partitions of overall jobs. Cluster controllers manage the health of the node controllers by periodically sensing the heartbeat to each node controller. This mechanism can be used to detect and recover from the failure of a Hyracks job through restarts of the Hyracks task schedule. A Hyracks job is the basic unit of the Hyracks programming model, and it consists of a graph of operators and connectors. For a given operator, Hyracks uses one or several activities to carry out the work. During job processing, an operator's activity nodes operate in parallel on partitions of the data. The job's connectors re-partition the partitions of the output of the operators that are then fed to the next operator. The next section provides a quick introduction to the Hyracks architecture and provides a simple example to explain the programming model of the Hyracks platform. A more extensive explanation of Hyracks, along with a first evaluation of its performance, can be found in [9].

## 2.1 Hyracks Architecture

Hyracks runs on a shared-nothing cluster of commodity computers with local CPUs, memories, and disks. The Hyracks software architecture consists of a cluster controller and a node controller. The cluster and node controllers work together to perform Hyracks job scheduling as well as to track and recover from failures of Hyracks jobs. Figure 1 summarizes the software architecture of Hyracks.

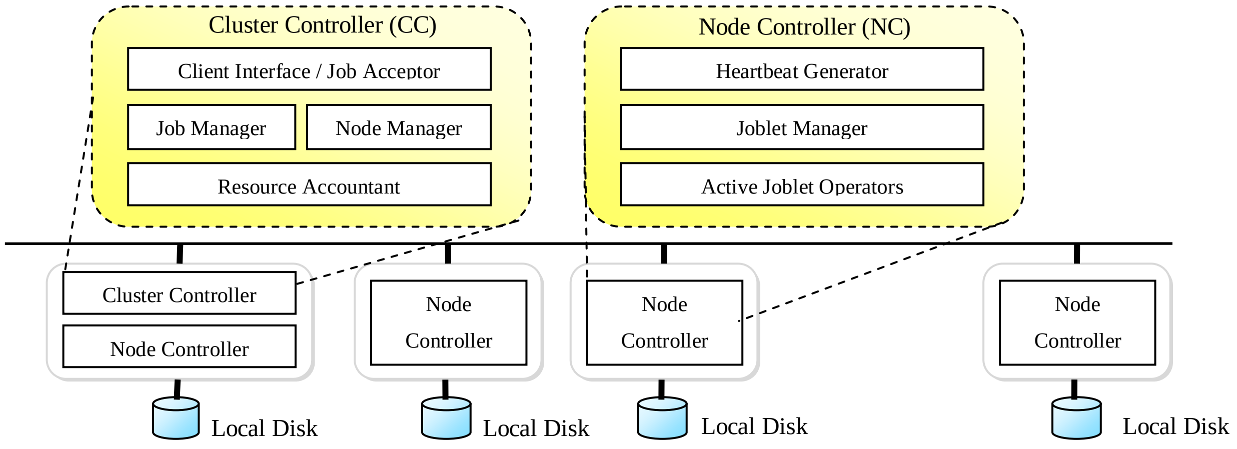


Figure 1 : Hyracks System Architecture

### 2.1.1 Cluster Controller (CC)

A cluster controller is used to manage a Hyracks cluster. In order to join a Hyracks cluster, a node initiates a Node Controller (NC) process by registering it with the Hyracks cluster controller. In the NC registration request, each NC also sends out information regarding its machines' resource configuration. After a successful registration, the CC begins to monitor the health status of the NC by periodically receiving its heartbeat. The CC is also responsible for monitoring the status of the entire cluster by keeping track of resource utilization metrics for each NC.

Cluster controllers also handle client interactions. When a client submits a Hyracks job, the cluster controller receives the job specification and internally expands it into an evaluation strategy. The evaluation strategy determines the parallelization and placement of operators in each stage of the job. Each stage is separated by a *blocking edge*, and each operator internally consists of one or several activities. A schedule will be created by the CC and will determine which NC will participate in the processing of each of the stages. The CC will then push messages to those scheduled NCs, in parallel, in order to activate the tasks for the final job.

Cluster controllers are also responsible for *fault handling*. For instance, in order to guarantee a job completion, in the event of a failure, the basic fault recovery response of the job's CC will be to re-plan and re-execute some or all of the tasks for this job.

### 2.1.2 Node Controller (NC)

For the job execution, each machine that participates in a Hyracks cluster runs one or more Node Controller processes. NCs are used to evaluate Hyracks jobs and each of the NCs sends its health status and data resource utilization metrics in a heartbeat signal to the Cluster Controller. NCs are responsible for accepting job tasks from a CC and executing the job tasks by using their local resources. At runtime, a given job is divided into one or more stages. The CC activates the stage by sending messages called "Task Activations" to the set of NCs chosen to participate in the stage. The CC then waits for the completion of the stage (or a failure event).

A *Joblet* is the collection of tasks of a job that is deployed on a particular NC. A joblet includes a collection of operators that perform processing on this specific NC. During the task execution on a set of NCs, the unit of data that is being produced by Hyracks tasks is called a *Frame*. A frame represents a fixed-sized chunk of contiguous bytes, and it is a serialized data format used for efficient data movement from sender to receiver.

### 2.1.3 Initialize Hyracks

In order to run Hyracks, users first need to start a CC process on a machine that is assigned to be its cluster controller node. Three optional arguments, *port*, *http-port* and *profile-dump-period* can be specified by the users when they start the CC process. The "http-port" is a

Jetty server port number in the Hyracks system that is used for communicating with the Hyracks console. By default, the CC listens for communication data from NCs through port 1099. If the CC does not listen on the default port, an optional *port* argument is required. Second, each worker node participating in this Hyracks cluster has to start an NC process. When an NC is started, it gets three required arguments: *node –id* (unique node's ID), *cc-host* (CC's IP address) and *data-ip-address* (NC's IP address). Last, a client machine connected to the cluster has to start a *hrackscli* process. This client machine is then responsible for deploying applications and submitting Hyracks jobs to the cluster. Figure 2 summarizes this step sequence.

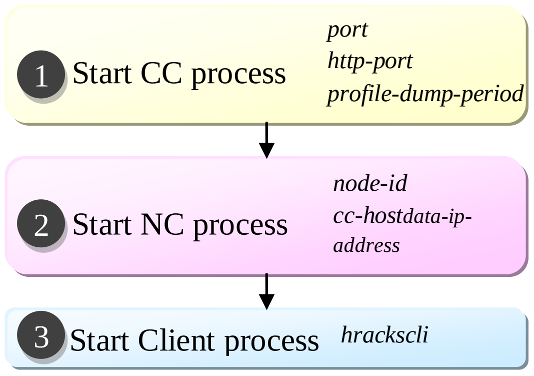


Figure 2: Initialize a Hyracks Cluster

## 2.2 Hyracks Jobs

In order to understand the data representation and Hyracks job visualization needs of the Hyracks console, this section gives a brief introduction to the logical design and implementation of Hyracks. We use an example query to detail the steps involved in Hyracks job execution.

A Hyracks job is submitted by a client as a *Hyracks Job Specification* that is represented as a directed acyclic graph (DAG) dataflow. The Hyracks Job Specification consists of a set of Hyracks Operator Descriptors (HODs) and Connector Descriptors (CDs). Each HOD node in the DAG represents a partitioned computation operator, and each CD edge connects two HOD nodes and represents a dataflow path from one operator to another operator. CDs also provide for data re-distribution by shuffling data from the set of sending operator partitions to the set of consuming operator partitions. Note that HODs or CDs in the Hyracks jobs do not provide logical or semantic information to Hyracks.

In order to provide Hyracks with the required information about control (scheduling) dependencies of Hyracks jobs, the CC internally expands each HOD into one or more sub-activities, or phases, called Hyracks Activity Nodes (HANs). Based on the resulting HAN graph, the CC then divides a given job into one or several stages and plans in which order the stages will be executed. At runtime, the CC decides the amount of parallelism for each stage and expands each HAN into a set of identical Hyracks Operator Nodes (HONs). Essentially, HONs are clones of a HAN and are responsible for executing the HAN logic on individual data partitions to produce the results of the HAN. Terminology-wise, the HOD graph is also called the Hyracks Job Specification, the HAN graph is also called the Job Plan, and the HON graph is called the Hyracks Runtime Tasks. Figure 3 summarizes these concepts.

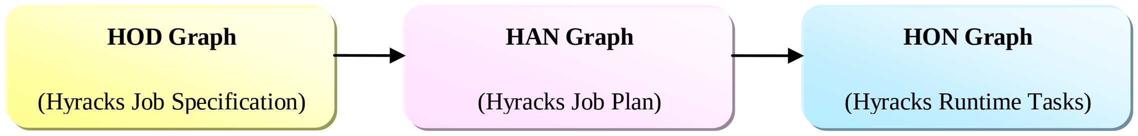


Figure 3: Representations for a Hyracks Job

Hyracks includes two broad classes of users: (1) Hyracks end users, who use the Hyracks library to assemble Hyracks jobs that solve their problems, and (2) Hyracks operator implementers who create new operators for Hyracks end users. The Hyracks library has a number of built-in common operators and connectors that are provided with Hyracks for the end users' use. Pre-existing operators such as "file scan", "file writer", "map", "sort", "join", "group" and "aggregate" are included in this core library. Connectors such as "1:1", "M: N Hash-Partition", "M: N Hash-Partition-Merge", "M: N Range-Partition", and "M: N Replicator" connectors are also part of the Hyracks core library. Besides the core library, Hyracks provides a rich API that enables operator implementers to create new operators and/or connectors. For more details about the Hyracks end user model, operator implementation model, and built-in library, see the recent Hyracks paper [9] and the Hyracks Google code web pages [10].

In the rest of this section, we will use a simple query to describe how Hyracks internally executes a job in detailed steps. This sample job is based on querying two files containing CUSTOMER and ORDERS data from the TPC-H dataset [12]. The sample job computes the total price of orders placed by customers belonging to the same market segment, and the query results are sorted by the total amount spent in each market segment. To be more precise, this job can be summarized by the following equivalent SQL query:

```
select C_MKTSEGMENT, sum (O_TOTALPRICE) as MKTSEGMENT_TOTAL

from CUSTOMER join ORDERS on C_CUSTKEY = O_CUSTKEY

group by C_MKTSEGMENT

order by sum (O_TOTALPRICE) as MKTSEGMENT_TOTAL
```

### 2.2.1 Hyracks Job Specification

A *Hyracks Job Specification* for our example query can be constructed as shown in Figure 5. Each of the nodes displays a HOD’s name and arguments, and each edge between two operators is annotated with the details of the data distribution strategy that is to be used for directing partitioned data from the set of senders to the intended receivers. The two “File Scan” operators also have additional metadata that specifies where the files are located. Because Hyracks partitions and stores data collections locally on different nodes in the cluster, in order to allow the file scanner operator to access the files, the CC must schedule the file scan runtime tasks on the machines where the file partitions are available.

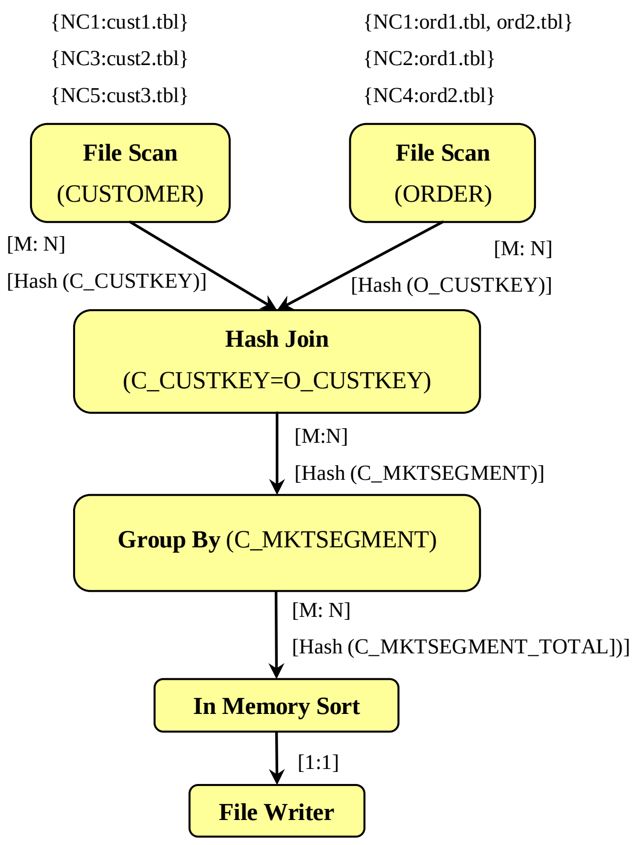


Figure 4: Hyracks Job Specification

(HOD Graph)

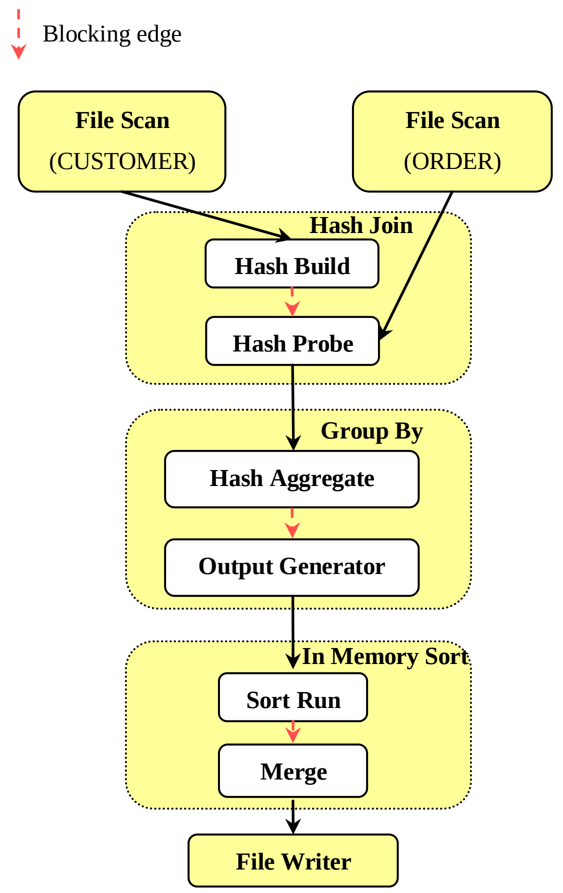


Figure 5: Hyracks Job Plan

( HAN Graph)

For instance, Figure 4 shows that the CUSTOMER data is partitioned into three files (cust1.tbl, cust2.tbl and cust3.tbl) that are stored locally on nodes NC1, NC3 and NC5, respectively. The ORDERS data also is partitioned, into two files (ord1.tbl and ord2.tbl), and each of the ORDERS partitions is *two-way replicated*. Hence, ord1.tbl can be accessed on either node NC1 or NC2, and ord2.tbl can be accessed on either node NC1 or NC4. Replication gives the CC multiple options for scheduling the file scan tasks of ORDERS data.

Each of the "File Scan" HODs in Figure 4 is used to read from a data source (CUSTOMER or ORDERS) separately and send its stream of data to the "Hash Join" HOD using an *M:N hash-based partition* connector. This connector uses a provided hash function to distribute the data produced by M senders to N receivers. In this case, the number of senders M is the number of partitioned files in each of data resources, so M=3 for CUSTOMER data and

M=2 for ORDERS data. In order to compute the "Hash Join" HOD in a partitioned manner, we need to ensure that CUSTOMER and ORDERS instances that satisfy the join condition (C_CUSTKEY = O_CUSTKEY) will be routed to the same join task. Thus, the hash partitioning of the CUSTOMER and the ORDERS instances is based on C_CUSTKEY and O_CUSTKEY, correspondingly.

Figure 4 shows that the "Hash Join" HOD receives two streams of data (one with CUSTOMER instances and one with ORDERS instances) and produces a stream of CUSTOMER-ORDERS pairs that satisfies the join condition (C_CUSTKEY = O_CUSTKEY). The result is then re-distributed to a "Group By" HOD using an *M:N hash-based partition* connector on the C_MKTSEGMENT field. This ensures that all CUSTOMER-ORDERS pairs that share a given value for that field will be routed to the same partition for aggregation. The "Group By" HOD aggregates the result of the join on the C_MKETSEGMENT field and uses a SUM aggregation function to sum the O_TotalPrice values within a group to compute a MKTSEGMENT_TOTAL field. An "M:N" hash-based partition connector is then used again to repartition the output partitions from the "Group By" HOD to the "In Memory Sort" HOD on the MKTSEGMENT_TOTAL field. Each sort task then sorts its input stream in descending order on the MKTSEGMENT_TOTAL field. Lastly, the final result of the job will be created as a set of partitions (with as many partitions as "In Memory Sort" HOD clones being used) by using a 1:1 connector between the "In Memory Sort" HOD and the "File writer" HOD. In the Hyracks library, a 1:1 connector leads Hyracks to create the same number of partitioned tasks for the sender HODs and receiver HODs, and to route the results pair-wise without re-distribution.

### 2.2.2 Hyracks Job Plan

When Hyracks starts to execute a job, it internally expands each HOD in the job's specification into a set of HANs. The results can be represented as a *Hyracks Activity Node graph* (also called a *Job Plan*) that is a somewhat more detailed *directed acyclic graph* (DAG), as shown in Figure 5. The expansion of each HOD indicates all sub-phases involved in each HOD along with the sequencing dependencies among the phases. The figure's dotted arrows denote blocking edges between pairs of activities, so each depended-upon activity must finish before the dependent activity on begin in Figure 5.

In the example shown, the “Hash Join” HOD is expanded into two HANs, “Hash Build” and “Hash Probe”. The “Hash Build” HAN builds a hash table on the input stream (usually the one with less data). The “Hash Probe” HAN then probes the resulting hash table with the other input stream in order to produce the result stream of join pairs. Note that the build phase of the “Hash Join” HOD thus needs to complete before the probe phase can begin in order for this to work properly. This sequencing constraint is shown as the dotted arrow from the “Hash Build” HAN to the “Hash Probe” HAN in Figure 5. The hash-based “Group By” HOD is similarly expanded into two HANs, the “Hash Aggregate” and “Output Generator” HANs. The blocking edge between these HANs represents the fact that the aggregator cannot begin to produce output until it has received all of its input data. Likewise, the “In Memory Sort” HOD can be expanded into a “Sort Run” HAN that precedes a “Merge” HAN. The former HAN generates a set of sorted runs, and the latter HAN produces the final sorted result by merging all of the sorted runs generated by its predecessor HAN. No output can be produced by the “Merge” HAN until the “Sort Run” HAN completes. Lastly, the “File Write” HOD becomes one corresponding HAN without any sequencing constraints.

### 2.2.3 Hyracks Runtime Tasks

The reason for having HODs expose their internal HANs to Hyracks in the manner described is to provide the Hyracks scheduler with insight into the sequencing dependencies of each part of a Hyracks job in order to facilitate execution planning and coordination. At runtime, HANs that are transitively connected to other HANs in a job through non-blocking edges can be grouped together to form a single executable stage. Internally, a stage is a set of connected HANs that can be co-scheduled (e.g., to be execute in a pipelined manner). A given stage is ready to execute when all HANs involved in its prior dependencies have been executed completely and successfully. The Hyracks scheduler decides on the degree of parallelism and the NC allocation of the various HAN instances based on their node affinities and resource requirements. For instance, all “File Scan” HAN tasks must be assigned to NCs where the files containing the partitioned data are located.

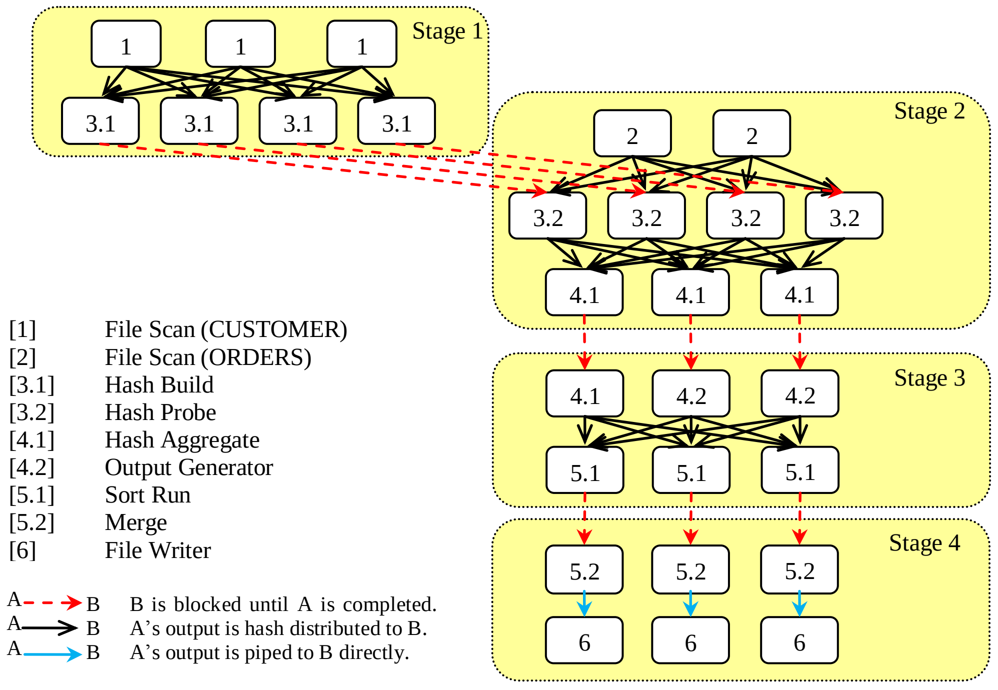


Figure 6 : Hyracks Runtime Tasks (HONs Graph)

When a given stage is scheduled, it is expanded by Hyracks into a set of runnable tasks called *Hyracks Operator Nodes* (HONs). In fact, HONs are simply runnable clones of the HANs. Each HON is created at runtime and is responsible for performing the actual computation for a HAN at chosen worker node. This is depicted in Figure 6. Each of the dotted boxes in Figure 6 indicates a group of HONs that belongs to the same stage. Although Figure 6 shows the HONs of all four stages at one time, in reality Hyracks expands each stage into a set of parallel HONs just prior to the stage's execution. At execution time, then, the four stages in our example will be activated in sequence.

In the first stage of our example Hyracks job in Figure 6, we assume that the CUSTOMER data is partitioned into three files and that Hyracks decides to use four nodes to execute the "Hash Join" HOD. Hyracks will start by running the three "File Scan" HONs of CUSTOMER data along with the four "Hash Build" HONs to produce hash tables of CUSTOMER data. Once the first stage is completed, the next stage will be planned and executed. The second stage will probe the hash table resulted from the first stage using the ORDER data, and will pipeline the join-pair results into the "Hash Aggregate" HONs. We assume in the second stage that ORDER data is divided into two files with two-way replication and that Hyracks decides to use three nodes for the “Group By” HOD. Thus, in the second stage of Figure 6, there are two clones of the “File Scan” HON, four clones of the “Hash Probe” HON, and three clones of the “Hash Aggregate” HON. When the second stage has been executed completely, Hyracks will begin to plan the third stage, composed of the “Output Generator” HONs and the “Sort Run” HONs. Assuming that the Hyracks decides to use three nodes to perform the “In Memory Sort” HOD, so there are three “Sort Run” HONs. Finally, after completing the third stage, the “Merge” HONs of the “Sort” HOD along with the “File Writer” HONs will be activated and executed to produce the final result of this example job. The job’s execution is then successfully completed, and the result will be located (in partitioned files) at the “File Writer” HON’s nodes.

# 3 CONSOLE DESIGN GOALS

As briefly mentioned earlier, Hyracks includes two broad classes of users, *Hyracks end users* and *Hyracks operator implementers*. We expect the Hyracks console to be used by both kinds of Hyracks users. These two different groups have different functional requirements for our Hyracks console. We designed the Hyracks console based on the results of an informal survey of Hyracks developers’ functional requirements and performance demands. This section summarizes the design goals that were identified for the Hyracks console.

## 3.1 Hyracks End Users

Hyracks end users use the existing Hyracks operators and connectors to assemble Hyracks jobs for solving their problems. They might encounter the following difficulties when they execute their Hyracks jobs:

- **Failure events:** When an error occurs during a job’s execution, users gain limited information of what caused the problem and failure. Thus, it can be hard and time consuming for end users to debug their Hyracks jobs and figure out at which stage an error has occurred.
- **Run-time status:** After end users deploy a Hyracks job, they might face a difficulty in observing the current status of their jobs; for instance, which stage is currently running

and which stage(s) is completed. End users also may want to know how far along their jobs are. Besides difficulty in monitoring current job's status, end users are often not able to get the complete picture of which nodes participating in Hyracks cluster and how their jobs' tasks have been assigned to these nodes.

## 3.2 Hyracks Operator Implementers

Hyracks Operator Implementers are responsible for creating new Hyracks operators and connectors for Hyracks end users. The operators and connectors are either implemented in the system's core library or in the user's additional library. This class of Hyracks users are likely to face the following problems.

- **Failure events:** Here the Hyracks operator implementers face amount of similar problems as the Hyracks end users. They have to spend a significant time on testing and debugging their Hyracks jobs. The time and effort can be extensive if the developers need to manually find the location of the errors.
- **Internal processing:** As mentioned earlier, each HOD consists of one or several HAN(s). When Hyracks operator implementers execute their Hyracks jobs, they need to ensure that the internal processing of the HODs and/or CDs is correct, e.g., that their HODs and CDs are expanded and executed correctly, and that the dataflow is partitioned and distributed nicely across the NCs.

To address all these difficulties of Hyracks users, we set up several goals for our Hyracks Console. The ultimate goal of the console is to reduce the time and effort required in understanding, implementing and optimizing Hyracks jobs. Hyracks users should be able to monitor the health of the Hyracks Clusters and to track the internal processes involved in a Hyracks job's execution. In addition, the Hyracks Console should satisfy the following characteristics:

- **Real-time status:** It will be very beneficial for programmers, if the Hyracks console is able to represent the real-time status of a Hyracks cluster and Hyracks jobs (and whether that status is initial, running, finished, or failed). Hence, we should consider how and when to update the latest status of the Hyracks system.

- **Information scalability:** Addressing scalability covers two related requirements, the scalability of the architecture and the scalability of the visualization. We need to consider cases when the Hyracks system may be running on a cluster with thousands of machines and executing thousands of Hyracks jobs simultaneous.
- **Interactive GUI:** Users should be able to access all the information that they need in the fastest way. The Hyracks console should minimize the effort from its users by creating an interactive *Graphical User Interface* (GUI).

# 4 HYRACKS CONSOLE ARCHITECTURE

This section explains the internal architecture of the Hyracks Console along with related design decisions and implementation techniques. It is aimed at readers wishing to understand how the console works inside. The Hyracks Console is based on a *Representational State Transfer* (REST) Web-based architecture [13]. The REST architecture is targeted by many distributed information systems because it provides a set of architectural constraints that emphasize the scalability of component interactions, simplicity of interfaces, as well as the independence of client and server deployments.

Resources are the heart of the REST architecture. REST defines the identity of a resource via a *Uniform Resource Identifier* (URI) using the http protocol. Interaction with a resource takes place at its URI. In the Hyracks Console, the resources are separated into two parts: *Hyracks Jobs* and *Hyracks Clusters*. Each part has its own static and dynamic information. Resources are conceptually independent from their presentations returned to the client. For example, a resource may involve information stored in a plain text format, but the Web server can return that resource data in a *JavaScript Object Notation* (JSON) format [14]. In the Hyracks console, JSON is used as a data-interchange format to transmit the information between a server and Web application. For readers not familiar with JSON, it is primarily used as an alternative to the *Extensible Markup Language* (XML). JSON has a much smaller grammar than XML and maps more directly to the data structures used in modern programming languages. JSON objects are built up from two structures: *name/value pairs* and *ordered lists of values*. These are universal data structures found in most programming languages, so the JSON format is compatible with many modern languages.

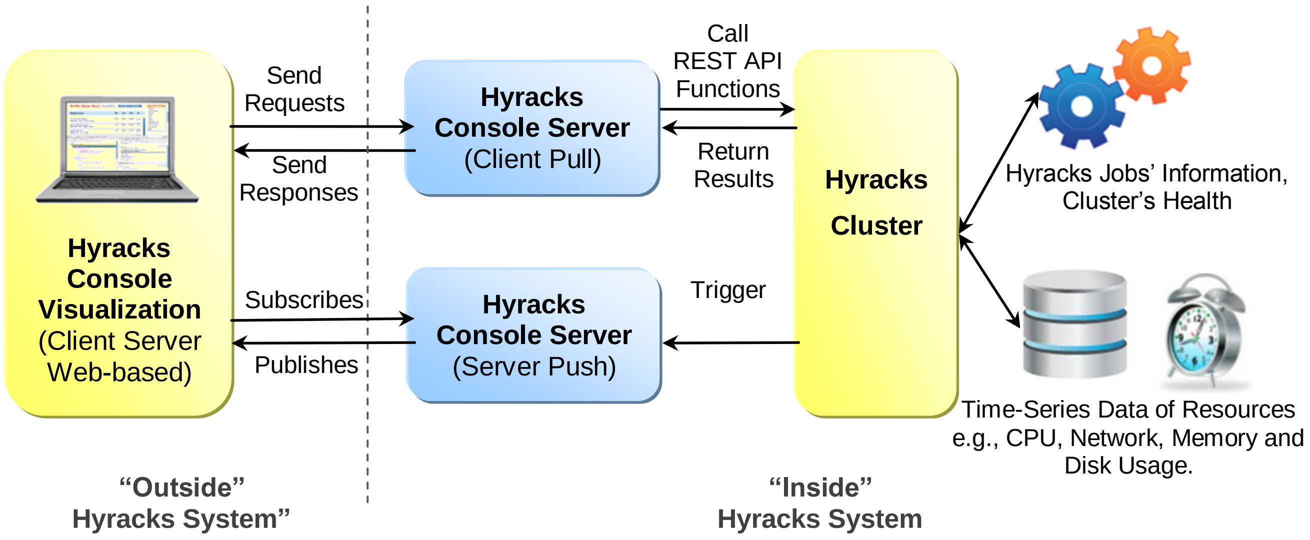

Hyracks Console Visualization (Client Server Web-based)
Send Requests
Send Responses
Subscribes
Publishes
Hyracks Console Server (Client Pull)
Hyracks Console Server (Server Push)
Call REST API Functions
Return Results
Trigger
Hyracks Cluster
Hyracks Jobs' Information, Cluster's Health
Time-Series Data of Resources e.g., CPU, Network, Memory and Disk Usage.
"Outside" Hyracks System"
"Inside" Hyracks System

Figure 7: Hyracks Console Architecture

Figure 7 provides an overview of the Hyracks Console architecture. Based on the REST style, the Hyracks Console is conceptually separated into two main components: the *Hyracks Console Server* (HCS) and *Hyracks Console Visualization* (HCV). The first component is responsible for collecting and transferring the monitoring information from the Hyracks system to the second component (HCV). The HCV is then responsible for presenting the current state of the Hyracks system in real-time to users. The Hyracks Console information and control flow starts from the Hyracks users asking the HCV to send requests for monitoring information of the Hyracks system to the HCS through a unique URI. The HCS then responds with the requested information. In this section, we provide further details of each component of the Hyracks Console architecture.

## 4.1 Hyracks Console Server (HCS)

The first console component is the *Hyracks Console Server* (HCS) implemented "inside" the Hyracks system. The HCS is responsible for collecting and delivering useful information related to the current state of Hyracks jobs and Hyracks clusters to the second component, the *Hyracks Console Visualization* (HCV). Two mechanisms have been used for transmitting the required data: client-pull and server-push. To support these two different techniques, we have implemented two Web servers within the Hyracks system.

The first communication technique between client and server is called *client-pull* [15]. To produce only the necessary data, a client must initiate a request for transmission of information from the HCS, and the HCS then receives the request and uses a REST API function to respond with data in the JSON format. Considering the scalability issue, we decided to use Jetty [17], an open-source project providing an HTTP server together with the `javax.servelet` container for the web server. Monitoring information of the Hyracks cluster is gathered from two main sources. The first type of data is the Hyracks jobs' information and the Hyracks cluster's health, each of which is collected and temporally stored by the *Hyracks Cluster Controller (CC)*. The second type of data is time-series data that is collected by the Ganglia system [18, 19] and archived on the *RRDtool* (*Round-Robin Database*) [20]. Ganglia collects data based on the physical view of the cluster, so there is one and only one set of data for each machine. Unlike Ganglia, one machine in the Hyracks cluster can install one or more *Hyracks Node Controllers (NCs)*. To map the logical NC nodes to the real physical nodes of a cluster, we implemented an extra file which represents and stores this mapping information as pairs of a machine's IP address and a location of its RRD table that stores the machine information.

The other communication technique is called the *server-push* [16] mechanism which is used to support a real-time monitoring. To provide this service, we use *CometD* [21], which supports a scalable HTTP-based push technology known as *Comet* [22] over a *Bayeux* protocol [23]. The Bayeux protocol can transport asynchronous messages with low latency between a web server and a web client. If any significant event occurs in the Hyracks system, HCS will initiate the communication and send message to the clients independently without first needy requests from them. Clients have to "subscribe" or "listen" to a particular "channel" to receive a message. When any new information is available, the HCS "publishes" that information to that "channel". We use this server-push mechanism to deliver any/all dynamic information that should be presented in advance, such as a notification of a new Hyracks job or a notification of completed stages and/or failed stages of a given Hyracks job. Further explanation of internal console data collection issues, together with the set of REST API functions implemented in the Hyracks console, can be found in [24].

## 4.2 Hyracks Console Visualization (HCV)

To ease analysis of the collected data, we have built a flexible, “portal-style” graphical web interface in the *Hyracks Console Visualization (HCV)*. This component is on the “outside” of the Hyracks system and is responsible for representing the Hyracks information to users through a Web interface. The HCV is independent of the Hyracks clusters. By using the HCV, a Hyracks user is able to monitor any Hyracks cluster either locally or remotely through any types of Web browsers, and multiple Hyracks users are able to monitor the same Hyracks cluster at the same time. The only assumption is that the user’s machine is able to reach a *Hyracks Console Server (HCS)* inside the Hyracks clusters of interest.

Since the Hyracks system is written in the Java programming language, Java Server Pages (JSP) [25] was chosen as the most appropriate technology for implementing the HCV. JSP is a Java technology that provides developers with a simplified and easy way to create a dynamic Web content. JSP technology is designed to address the problem that the Java programming environment does not provide enough support for Web applications. The JSP Java code and certain pre-defined actions are interleaved with the static Web markup and content, with the resulting page being compiled and executed on the server to deliver an HTML or XML document. To deploy and run JSP in HCV, the *Apache Tomcat* [26] web server is used. Tomcat implements the Java Servlet and JSP specifications from Sun Microsystems, and provides a "pure Java" HTTP web server environment for Java code to run in.

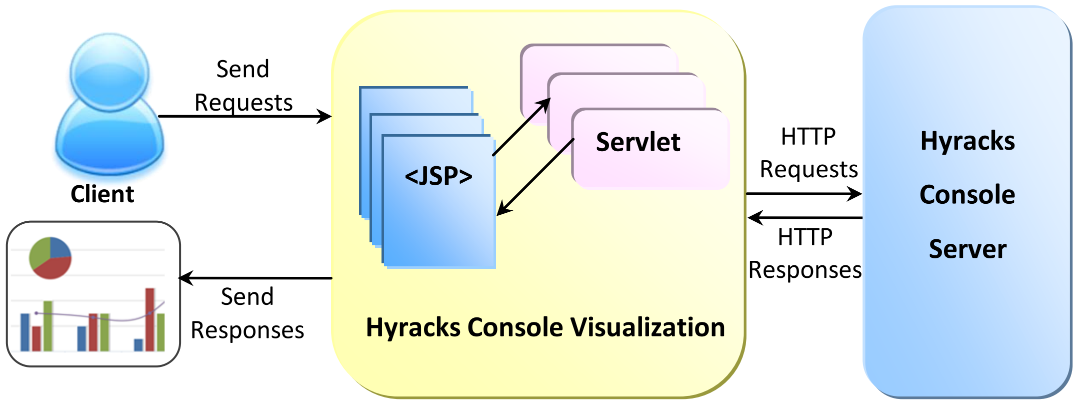


Figure 8 : Hyracks Console Visualization Model

The HCV uses the JSP and Servlet technologies to interact with the client. By using a REST-style architecture, once the client sends request through JSP, the Web server will trigger the data representation mechanism in the HCV to send an HTTP request to the Hyracks Console Server (HCS) inside the Hyracks cluster. The HCS then returns an HTTP response containing the resource information related to the user’s request in mostly JSON format. Finally, the HCV presents the returned information to the client using an appropriate visualization method. Figure 8 summarizes the above concepts. The data representation mechanism of the HCV will be explained in more detail in Section 5.2.

The main responsibility of the HCV is to represent the current state of Hyracks clusters and Hyracks jobs. Humans are a visually-oriented species, which means that they are generally more sensitive to pictures or diagrams versus plain text. Therefore, the console uses various visualization methods, including tables, charts, graphs and dataflow diagrams, to represent the more complicated information in the simplest possible way. Many visual representation tools have been used in the HCV, including Graphviz [27], TimeGuilder [28], Hightchart [29] and DataTable [30]. These tools all support the interactive GUI and together present the necessary Hyracks data in a visualized manner. The Hyracks Console not only represents the high-level abstraction of the Hyracks cluster, but also allows users to drill down into the internal details of any Hyracks job.

To provide intuitive and useful visual representations of Hyracks information via Web pages, there were severed challenges involving the design of a convenient user interface and interactions between information requests and server responses. The visual composition and temporal behavior of Graphical User Interface (GUI) should have the ability to scale well (i.e., the visualization should be intelligible at the large future scale of Hyracks clusters and Hyracks jobs) and the ability to be dynamic (the user interface should be updated in real-time when the current state of the Hyracks system changes).

## 5 IMPLEMENTATION

In this section, the implementation of the Hyracks Console Visualization (HCV) component is described in more detail. We first give a quick introduction to the REST API for requesting each type of Hyracks monitoring information. Next, we describe the general mechanism used for data representation in the HCV. We depict the benefits and usage of each type of monitoring data along with real example results returned to the HCV component. Some details of the server push mechanism for updating the current status of the Hyracks are then discussed. Throughout this section, we assume that the IP address of the master node running the Hyracks Cluster Controller process is “vanilla.ics.uci.edu” and that its http-port number is “2099”. All of the URI examples shown here are based on this assumption.

## 5.1 REST API for Requesting the Hyracks Monitoring Information

Before we can start to monitor the Hyracks, the REST API has to be setup at the *Hyracks Console Server (HCS)* for requesting resources via a set of global URLs. The resources are divided into two main categories: Hyracks job execution information and Hyracks cluster behavior. The *Hyracks Console Visualization (HCV)* can navigate through every resource easily since the “URL-Request” paths are organized based on the tree structure show in Figure 9 and Figure 10.

### 5.1.1 Hyracks Jobs

The first goal of the console is to help the Hyracks users understand the internal details of Hyracks job executions. Therefore, we decided to provide the following information about Hyracks jobs. (1) A summary of all the jobs in the Hyracks cluster. Each Hyracks job has a (2) job specification, (3) job plan, (4) job profile, and (5) job stage. Figure 9 shows the URL paths that the HCV can use for requesting different types of information.

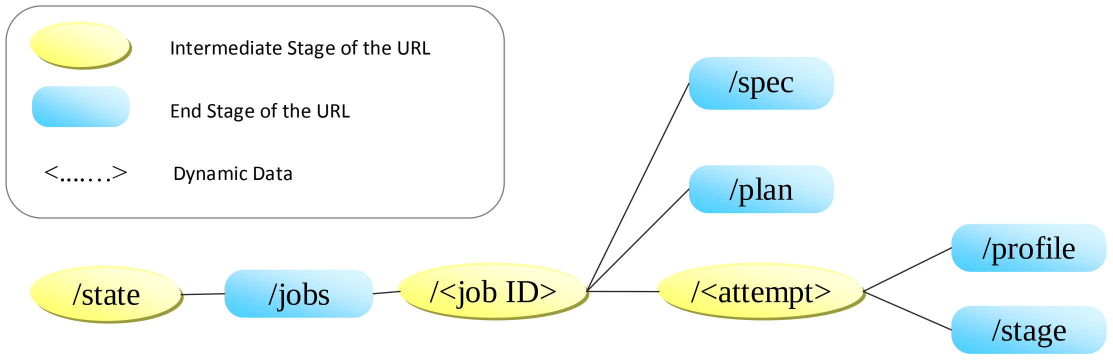


Figure 9 : URL-Request Paths for Hyracks Jobs Information

An example of the URL path for requesting a summary of all Hyracks jobs in a given Hyracks clusters is shown in the box below.

**URL Path**

```
http://<cc-ip-address>:<http-port>/state/jobs
```

**Example**

```
http://vanilla.ics.uci.edu:2099/state/jobs
```

### 5.1.2 Hyracks Cluster Health

Another kind of useful monitoring data provided by the Hyracks Console is Hyracks cluster health information. Any Hyracks cluster consists of two types of node: a Hyracks Cluster Controller (CC) and one or more Hyracks Node Controllers (NCs). In reality, a large distributed clusters usually consist of hundreds or even thousands of nodes. At that scale, it is inefficient and almost impossible to monitor the current state of the entire cluster by accessing each physical node one-by-one. As a result, the URL-Request paths in Figure 10 are generated to request Hyracks Cluster monitoring information. The information related to the Hyracks cluster includes (1) CC configuration (2) NC summary, (3) NC configuration, (4) NC jobs summary, and (5) NC resources.

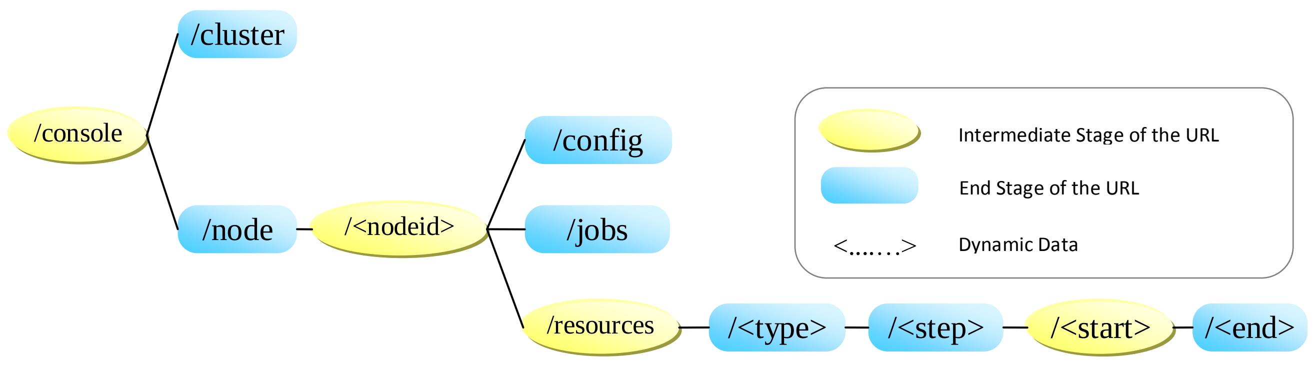


Figure 10: URL-Request Paths for Hyracks Cluster Health Information

Further details of the implementation of the REST API and the collection of Hyracks resources at the HCS can be found in [24].

## 5.2 Data Representation Mechanism

When Hyracks users send a request to the *Hyracks Consoler Server (HCS)*, most of the data returned from the server is presented in the JSON format. Although data in the JSON format is fairly easy to read and comprehend, typical users still prefer information represented visually. Moreover, when such data become large, it is almost impossible for users to gain knowledge by searching through thousands lines of text. Thus, a *data representation mechanism* to summarize and present the JSON data has been implemented in the *Hyracks Console Visualization (HCV)*. As noted earlier, the HCV is responsible for sending requests to the HCS and representing the returned results from the HCS visually. The data representation mechanism consists of three simple steps: First, the HCV sends a request to the HCS via one of the URL-Request paths. Then, when the HCS receives the request, it returns the JSON data based on the type of the URL-Request. Second, the HCV loads the JSON data by choosing a selected visualization tool. Third, this selected visualization tool presents the Hyracks information in an appropriate format, such as a diagram, a table or a chart, to the Hyracks user. Figure 11 summaries these steps.

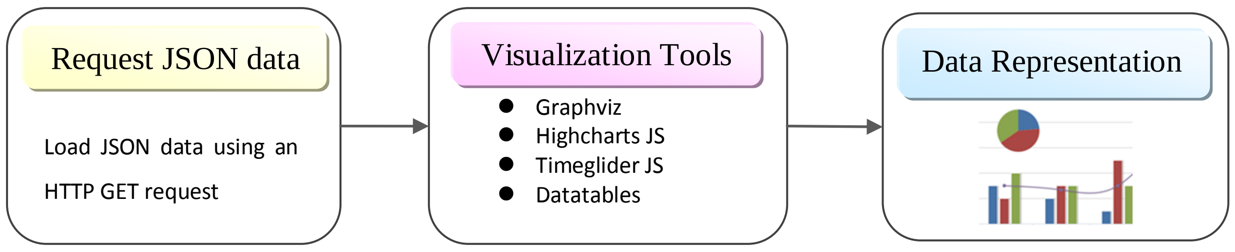


Figure 11: Data Retrieval and Representation

## 5.3 Data Representation Modules

As we mentioned earlier, Hyracks monitoring information is divided into two main categories: Hyracks job information and Hyracks cluster health information. In the next section, we describe in more detail, of the data representation in each category based on real examples.

### 5.3.1 Hyracks Jobs

The Hyracks Cluster Visualization (HCV) provides a *Hyracks Job Summary* component to present information about all Hyracks jobs that have been submitted to a given Hyracks cluster. The HCV presents the information about each Hyracks Job by separating into several modules, including a *Hyracks Job Specification*, *Hyracks Activity Node Graph*, *Hyracks Job Pie Chart*, and *Hyracks Job Time Chart.*

- **Hyracks Job Summary**

The *Hyracks Job Summary* module summarizes all of the Hyracks jobs that have been submitted to a given Hyracks cluster. There are five attributes presented about each Hyracks job: the *Job Name, Application Name, Start Time, Duration*, and job *Attempt* times. Each Hyracks job has a unique job-id (UUID) [31]which is generated by the Hyracks system and used in communications between the HCV and the HCS. However, as this job-id is not friendly to read and understand, the job's display name (generated by the HCS based on the start time of each job) is presented instead. This name starts from "job_000000001" and increases by one value at a time.

To implement the *Hyracks Job Summary* component, the HCV follows the data representation procedure shown in Figure 11. The visualization tool for representing the Hyracks jobs summary is *DataTables* [30]. *DataTables*, a plug-in for the jQuery JavaScript library, is an open sources tool for adding interaction controls to any HTML table, including sorting, searching and paging. The HCV sends a request to the HCS through the "Jobs-Summary" URL and represents the returned data in the tables using the DataTables jQuery tool. The data for each job is categorized into one of four independent tables based on the real-time status of the Hyracks job (i.e., running, failed, completed, or initialized). The maximum number of Hyracks jobs in each page in each table is ten by default.

In our example, the HCV first sends a request to the HCS through the *http://vanilla.ics.uci.edu:2099/state/jobs* URL. The HCS returns Hyracks job summary data in a JSONArray format. The HCV then uses the DataTables tool to present and distribute information about each Hyracks job in the received JSONArray into one of the four job tables based on the current job's status. In addition, the jQuery Toggle class is added to each table to allow users to hide or show the tables content. To achieve the scalability issue of the table content, users can search for their jobs within the table's contents by putting a keyword in search box. By default, job information in each table is sorted by the job name field; however, users can choose to sort a job table based on other attributes such as an *application's name*, *start time,* or *duration time*. This Hyracks Job Summary component allows users to easily browse all of the Hyracks jobs belonging to a particular Hyracks cluster in one place, as well as to search for a particular job that they want to monitor. Figure 12 summarizes the steps involved in generating the Job Browser using Hyracks job "job00000002" as an example.

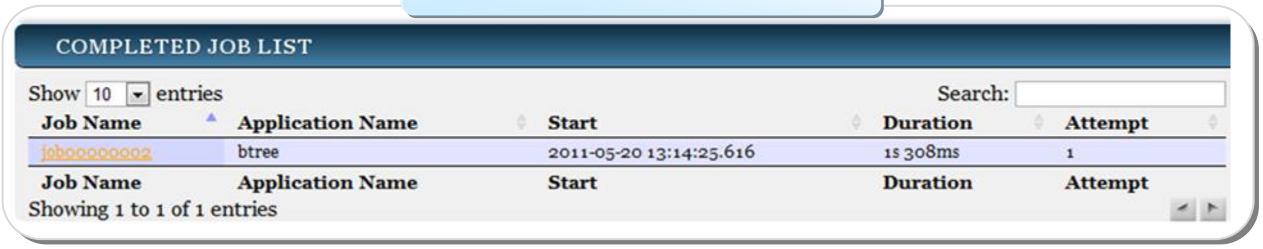
Job-Summary URL

**URL Path**
http://<cc-ip-address>:<http-port>/state/jobs
**Example**
http://vanilla.ics.uci.edu:2099/state/jobs

Job Summary JSONArray

```
{
        result: [{
                id: "5be03555-cbe0-4d30-808f-91d4e92659c3"
                application: "btree"
                display-name: "job00000002"
                status: "TERMINATED"
                events: [
                        {
                        status: "RUNNING"
                        system-time: 1305922465621
                        date: "2011-05-20 13:14:25.621"
                        }
                        {
                        status: "INITIALIZED"
                        system-time: 1305922465616
                        date: "2011-05-20 13:14:25.616"
                        }
                        {
                        status: "TERMINATED"
                        system-time: 1305922466929
                        date: "2011-05-20 13:14:26.929"
                        }
                ]
                attempts: 1
                type: "job-summary"
        }]
}
```

Job Lists from DataTables



COMPLETED JOB LIST

Show 10 entries　　Search:

| Job Name | Application Name | Start | Duration | Attempt |
|---|---|---|---|---|
| job00000002 | btree | 2011-05-20 13:14:25.616 | 1s 308ms | 1 |
| Job Name | Application Name | Start | Duration | Attempt |

Showing 1 to 1 of 1 entries

Figure 12 : Procedure for Generating a Hyracks Job Summary

- **Hyracks Job Specification**

The Hyracks Job Specification is a DAG dataflow diagram which consists of a set of Hyracks Operator Descriptions (HODs) and Hyracks Connector Descriptions (CDs)**.** The HCV sends a request to the HCS via the "Job-Spec" URL for gathering the Hyracks job specification for a particular job. The result returned is a JSONArray containing a set of HODs and a set of CDs. Based onto that information, the HCV is able to create a Hyracks job specification diagram by using *Graphviz* [27]. Graphviz is a package of open source tools initiated by *AT&T Labs Research* for drawing graphs that are specified in the DOT language [32]. To generate a Hyracks job specification diagram, we use the layout tool in the Graphviz software called *dot* [33], this tool will draw a directed graph as a graph file or a graphic format such as GIF, PNG, SVG or PostScript (which can be converted to PDF). The *dot* tool accepts DOT language scripts as an input and processes the scripts to generate graphics. The DOT language consists of three kinds of objects: graphs, nodes, and edges. The main (outermost) graph can be a directed (digraph) or undirected graph. The HCV uses the Graphviz Java API to generate a DOT script for the Hyracks Job Specification.

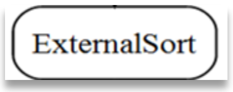
In the HCV, a servlet page named *JobSpecServlet* is implemented to produce the DOT script of the Hyracks Job Specification and then call the *dot* tool to draw the DAG diagram. *JobSpecServlet* requests *Job Specification* JSONArray from the HCS via the Job-Spec URL. The returned data is divided into two sets: a set of HODs and a set of CDs. Each HOD is then assigned as a node object in the DOT script. The node's ID is the HOD's ID and the node's label is the HOD's Java class name. An example of the node object in the DOT language script is shown in Figure 13.

**Node of the Job Specification Graph in the DOT language:**

“operator-id” [label= “<operator-class>”]

**Example:**

"4e53dd1c-c169-476c-ac80-245c818c3206" [label="ExternalSort" ]



**Node of the Job Specification Graph in the DOT language:**

“operator-id” [label= “<operator-class>”]

**Example:**

"92b74c8c-66f1-4a80-a5c2-a9c92e9abc46" [label="BTreeBulkLoad" ]

BTreeBulkLoad

Figure 13 : Example Node of the Job Specification Graph in the DOT Language

The next step is to generate the edge objects in the DOT script from the in-operator-id and out-operator-id fields of each CD and to use the Java class name of each CD as the edge’s label. An example of an edge object in the DOT language script that connects between two nodes (Figure 13) is shown in Figure 14.

**Edgeof the Job Specification Graph in the DOT language:**

“input-operator-id” - > “output-operator-id” [label= “connector-class”]

**Example:**

"4e53dd1c-c169-476c-ac80-245c818c3206" -> "92b74c8c-66f1-4a80-a5c2-a9c92e9abc46" [label="1:1"];

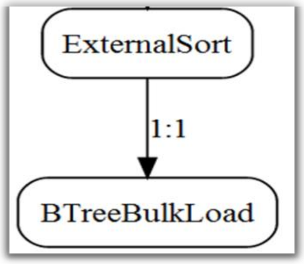


Figure 14 : Edge of the Job Specification Graph in the DOT Language

As the last step, the *JobSpecServlet* submits the generated DOT script to the *dot* tool. If the Job Specification Graph is created successfully, the *JobSpecServlet* will return the DAG in the *Scalable Vector Graphics* (SVG) format [33]. Figure 15 summarizes the procedure in generating the Job Specification Graph by using a simple Hyracks BTree-related job as an example.

JobSpecServlet Requests JSON data

**URL Path** http://<cc-ip-address>:<http-port>/state/jobs/<job-id>/spec
**Example** http://vanilla.ics.uci.edu:2099/state/jobs/5be03555-cbe0-4d30-808f-91d4e92659c3/spec

Job Specification JSONArray

```
{
result: {
connectors: [
            {in-operator-id: "ODID:4c8dfe24-880b-427e-bbd5-f5d290c1db66"
            connector: {id: "f4ec1ecc-2b73-4fae-abb2-061c29f3daca"
                        java-class:
"edu.uci.ics.hyracks.dataflow.std.connectors.MToNHashPartitioningConnectorDescriptor"}
            out-operator-id: "ODID:4e53dd1c-c169-476c-ac80-245c818c3206"}
            {in-operator-id: "ODID:4e53dd1c-c169-476c-ac80-245c818c3206"
            connector: {id: "cb473b23-bac4-4192-9a1a-f7c9008539d5"
                        java-class: "edu.uci.ics.hyracks.dataflow.std.connectors.OneToOneConnectorDescriptor"}
            out-operator-id: "ODID:92b74c8c-66f1-4a80-a5c2-a9c92e9abc46"}
            ]
operators: [
            {id: "92b74c8c-66f1-4a80-a5c2-a9c92e9abc46"
            java-class: "edu.uci.ics.hyracks.storage.am.btree.dataflow.BTreeBulkLoadOperatorDescriptor"}
            {id: "4c8dfe24-880b-427e-bbd5-f5d290c1db66"
            java-class: "edu.uci.ics.hyracks.examples.btree.helper.DataGenOperatorDescriptor"}
            {id: "4e53dd1c-c169-476c-ac80-245c818c3206"
            java-class: "edu.uci.ics.hyracks.dataflow.std.sort.ExternalSortOperatorDescriptor"}
            ]
}}
```

Parse to DOT Language

```
digraph hyracks_job {
        "92b74c8c-66f1-4a80-a5c2-a9c92e9abc46" [label="BTreeBulkLoad" ];
        "4c8dfe24-880b-427e-bbd5-f5d290c1db66" [label="DataGen" ];
        "4e53dd1c-c169-476c-ac80-245c818c3206" [label="ExternalSort"];
        "4c8dfe24-880b-427e-bbd5-f5d290c1db66" -> "4e53dd1c-c169-476c-ac80-245c818c3206" [label="M:N Hash"];
        "4e53dd1c-c169-476c-ac80-245c818c3206" -> "92b74c8c-66f1-4a80-a5c2-a9c92e9abc46" [label="1:1"];
}
```

Job Specification Graph from Graphviz

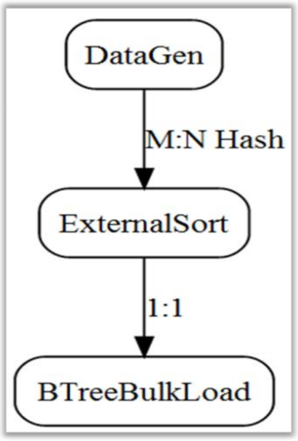


Figure 15 : The Procedure in Generating the Job Specification Graph

- **Hyracks Activity Node Graph (Job Plan)**

As mentioned before, one of the main goals of the Hyracks Console is offering an easy way to understand the internal execution process of the Hyracks job. To accomplish this goal, the console supports an interactive Hyracks Activity Node Graph (HAN graph or Job Plan) that provides insight of the Hyracks job's execution process. Recall that when the Hyracks system receives the Job Specification from the users, it internally expands each Hyracks Operator Descriptor (HOD) into a set of Hyracks Activity Nodes (HANs) and each of HANs has a set of input and /or output Hyracks connectors. All HANs that connect to each other with non-blocking edges are grouped together to form a stage. A Hyracks Job is divided into one or several stages, and every stage is executed in its order.

The procedure for generating the Job Plan diagram is based on the same approach as for the Job Specification diagram. To generate the DOT language script for the HAN Graph, the *JobPlanServlet* is used to load two JSON streams: the *Job Plan* and *Job Stage* JSONArray. The *Job Plan* JSONArray consists of all HANs expanded from the set of HODs from a Job Specification. Each HAN has a set of input and/or output CDs. In addition, the "depends-on" field in the *Job Plan* JSONArray shows the "ID" of the blocking activities to specify the dataflow constraints (blocking edges). Unlike the *Job Plan* or *Job Specification* JSONArray, which is static information, when the HCV requests the Job-Stage URL, it gets runtime data about the progress of the job's execution stage by stage. The *Job Stage* JSONArray is composed of several important entities; for example, the current status of each stage, the set of activities in each stage, the list of nodes assigned to process each activity, and the timestamp of the stage's events (e.g., `init-time, finish-time,` and `fail-time)`. When the *JobPlanServlet* loads the *Job Plan* and the *Job Stage* JSONArray, it first converts each of HANs into node objects in the DOT language script. As we mentioned in the Hyracks overview, each activity is cloned into a set of Hyracks Operator Nodes (HONs) that are scheduled to run on a set of Hyracks Node Controllers (NCs). The number of nodes at each HAN indicates the degree of parallelism decided by the Hyracks system. To provide better insight regarding the HON level at each HAN, a JavaScript function called *showNC* is embedded on each node of the HAN graph. This function displays the HAN's class name, the number of partitions, and the partition nodes when the user clicks on a particular node of the HAN graph. An example node in the HAN graph with its *showNC* function in the DOT language is shown in Figure 16.

**Node of the HAN Graph in the DOT Language:**

```
“HAN-id”
[id= “HAN-id” label= “HAN’sjava class”
URL= “javascript:showNC(HAN class, partition number, partition nodes )”];
```

**Example:**

```
"ANID:c4de682b-30d3-42aa-b138-b9987d90863c"
[id="ANID:c4de682b-30d3-42aa-b138-b9987d90863c" label="ExternalSort\n(SortActivity)"
URL="javascript:showNC('ExternalSort\n(SortActivity)',4,'[nc1,nc2,nc3,nc4]')" ];
```

Figure 16 : Example Node of the HAN Graph in DOTLanguage

The *JobPlanServlet* groups a set of HANs that are connected to each other with non-blocking edges into a sub-graph to form a Hyracks stage. In this step, the *JobPlanServlet* identifies the current status of the given stage and assigns the associated reference color to the sub-graph based on the stage’s status. For example, a light blue color represents a finished stage, a light green color represents a running stage, a gray color represents a pending stage, and a salmon color represents a failed stage. The *JobPlanServlet* organizes the order of the HANs inside each stage (sub-graph) by using the information from the Hyracks connectors. Each connector has the IDs of its input HAN and/or output HAN and these are used to construct the diagram.

In addition to process data, each connector between a sender and receiver HAN contains data movement information at the task-execution level from where each HAN has been cloned into Hyracks Operator Nodes (HONs). The unit of data produced and consumed by HONs is called a “Frame”. HCS provides a URL path that allows the HCV to request data movement information from each connector as a graphical image of a matrix representing data flow between senders and receivers, as shown in Figure 17. In this example, there is one HON on the sender side running on the node-id “nc1”, and four HONs on the receiver side running on the node-ids “nc1”, “nc2”, “nc3” and “nc4”. The number of frames sent from the sender “nc1” to the receivers “nc1”, “nc2”, “nc3”, and “nc4” are 33, 92, 101, and 190 respectively. The total number of frames is 416. The *JobPlanServlet* embeds a Javascript function called *showConnector()* into each connector (each edge in the Job Plan Graph) to request the communication matrix image. A detailed description of this matrix can be found in [24]. An example HAN sub-graph in the DOT langue is shown in Figure 17.

**Sub-graph (a Hyracks Stage) of the HAN Graph in DOT Language:**

```
subgraph cluster_number {
    id="stage-id" color=”stage-status-color”;
    ["input-HAN-id" -> “output-HAN-id"
        [label=”connector-class"
        URL="javascript:showConnector(‘connectorworkflow-url’, 'connector-class')",];
    ]
}
```

**Example:**

```
subgraph cluster_1 {
    id="sid_f0fafa9d10a64347a240cb9f954c8542"  color=lightblue;
    ["ANID:2f56ac5f-2cc4-42b7-bd45-3cbba572d919" ->
      "ANID:c4de682b-30d3-42aa-b138-b9987d90863c"
        [label="M:N Hash"
        URL="javascript:showConnector('http://vanilla.ics.uci.edu:2099/profile/5be03555-cbe0-
        4d30-808f-91d4e92659c3/0/f4ec1ecc-2b73-4fae-abb2-061c29f3daca', 'M:N Hash')"];
    ]
}
```

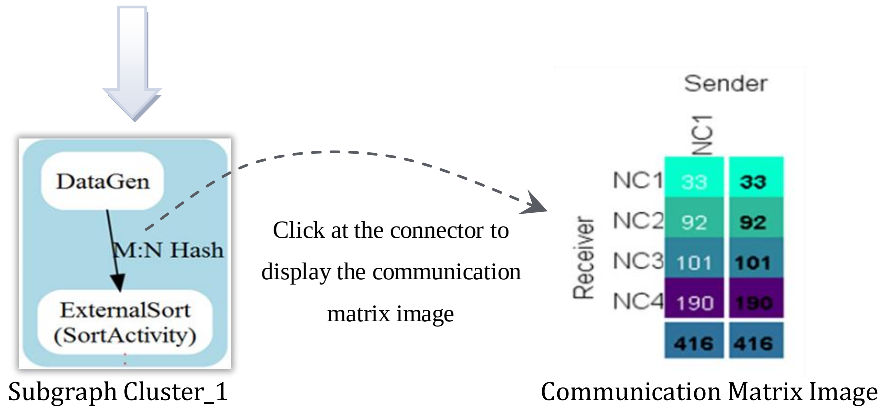

DataGen
M:N Hash
ExternalSort (SortActivity)
Subgraph Cluster_1
Click at the connector to display the communication matrix image
Sender
NC1
Receiver
NC1 33 33
NC2 92 92
NC3 101 101
NC4 190 190
416 416
Communication Matrix Image

Figure 17 : Example Sub-graph (a Hyracks Stage) of the HAN graph in the DOT Language and Communication Matrix Image

The *JobPlanServlet* will connect each stage (sub-graph) together with blocking edges. A blocking edge displays as a dotted red edge in the HAN graph. The input HAN-id of the blocking edge is the HAN-id of the last HAN node of the former stage, and the output HAN-id is the

HAN-id of the first HAN node of the latter stage. An example HAN blocking edge in the DOT language is shown in the middle of Figure 18.

**Blocking Edge of the HAN Graph in DOT Language:**

```
“input-HAN-id" -> "output-HAN-id" [style=dotted,arrowhead=vee, arrowsize=2,color=red]
```

**Example:**

```
ANID:2df02bc7-d6ef-4d61-aa5f-2ae45f1ed6e2"  ->  "ANID:22c232fb-b785-4839-aa19-de4c1952c115"
[style=dotted,arrowhead=vee,arrowsize=2,color=red];
```

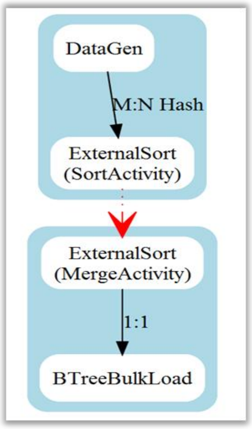
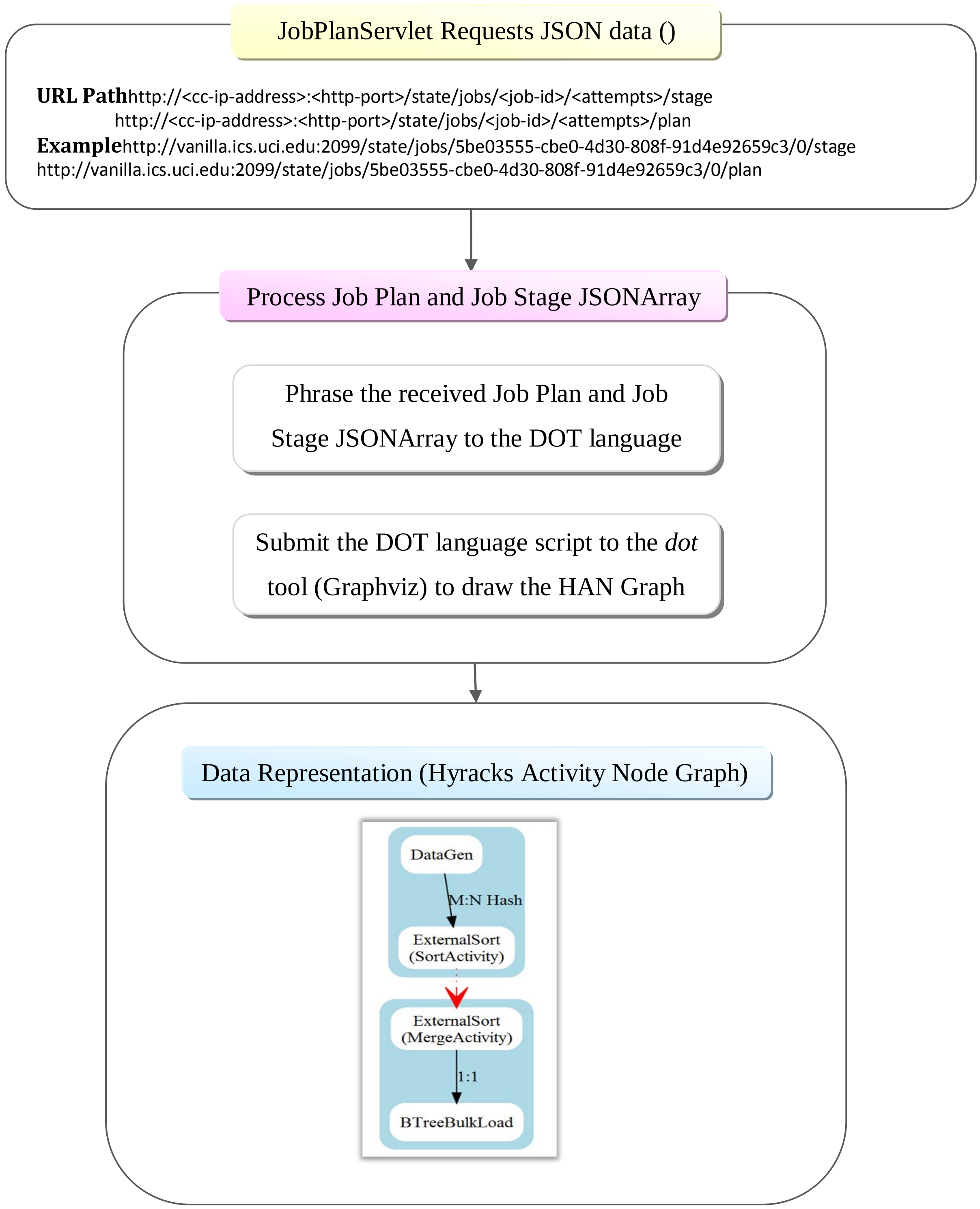


Figure 18 : Example Blocking Edge of the HAN Graph in the DOT Language

A complete example of DOT language scrip for a HAN Graph is shown in Figure 19.The JavaScript functions of the Hyracks Activity Node Graph will be explained in more detail in Section 6.2. Figure 20 summarizes the procedure for generating the interactive HAN Graph.

```
digraph hyracks_job {
size = "20,20"; rankdir = "TB";
"ANID:22c232fb-b785-4839-aa19-de4c1952c115" [id="ANID:22c232fb-b785-4839-aa19-de4c1952c115"
label="HashGroup\n(OutputActivity)"
URL="javascript:showNC('HashGroup\n(OutputActivity)',2,'[nc1,nc2]')"];
"ANID:f37cdbb0-2947-4a8b-a5b7-aa888b9db4ad" [id="ANID:f37cdbb0-2947-4a8b-a5b7-aa888b9db4ad"
label="InMemoryHashJoin\n(HashBuildActivityNode)"
URL="javascript:showNC('InMemoryHashJoin\n(HashBuildActivityNode)',1,'[nc1]')"];
"ANID:7f72f554-89a6-4525-a63a-68e84af65284" [id="ANID:7f72f554-89a6-4525-a63a-68e84af65284"
label="FileScan" URL="javascript:showNC('FileScan',2,'[nc1,nc2]')"];
"ANID:e9f4aebd-b04b-4eb6-8aef-f8699e557e1d" [id="ANID:e9f4aebd-b04b-4eb6-8aef-f8699e557e1d"
label="FileScan" URL="javascript:showNC('FileScan',2,'[nc1,nc2]')"];
"ANID:e95df9e1-a8db-46e3-8609-d84a2cad587a" [id="ANID:e95df9e1-a8db-46e3-8609-d84a2cad587a"
label="InMemoryHashJoin\n(HashProbeActivityNode)"
URL="javascript:showNC('InMemoryHashJoin\n(HashProbeActivityNode)',1,'[nc1]')"];
"ANID:2df02bc7-d6ef-4d61-aa5f-2ae45f1ed6e2" [id="ANID:2df02bc7-d6ef-4d61-aa5f-2ae45f1ed6e2"
label="HashGroup\n(HashBuildActivity)"
URL="javascript:showNC('HashGroup\n(HashBuildActivity)',2,'[nc1,nc2]')"];
"ANID:218446b2-ebea-415e-b279-f565db1fabd0" [id="ANID:218446b2-ebea-415e-b279-f565db1fabd0"
label="FrameFileWriter" URL="javascript:showNC('FrameFileWriter',2,'[nc1,nc2]')"];
subgraph cluster_0 {  id="sid_38b8589bd0584564951d53c8eae0873c"  style="filled,rounded"; node
[style=filled, fillcolor=white]; label = "";
 color=lightblue;
["ANID:e9f4aebd-b04b-4eb6-8aef-f8699e557e1d" -> "ANID:f37cdbb0-2947-4a8b-a5b7-aa888b9db4ad"
[label="M:N Hash" URL="javascript:showConnector('http://vanilla.ics.uci.edu:2099/profile/a965fe5a-ba0b-
4a01-a13d-1b4dcc57adc1/0/0f641b5b-1624-4e59-8ff4-ad1e2bdd0e9a', 'M:N Hash')",];
]}
subgraph cluster_1 {  id="sid_56081892010f4507ba4cdb3bdcefb009"  style="filled,rounded"; node
[style=filled, fillcolor=white]; label = "";
 color=lightgrey;
[]}
subgraph cluster_2 {  id="sid_0689036c64124484a024c535e29d993a"  style="filled,rounded"; node
[style=filled, fillcolor=white]; label = "";
 color=lightblue;
["ANID:22c232fb-b785-4839-aa19-de4c1952c115" -> "ANID:218446b2-ebea-415e-b279-f565db1fabd0"
[label="1:1" URL="javascript:showConnector('http://vanilla.ics.uci.edu:2099/profile/a965fe5a-ba0b-4a01-
a13d-1b4dcc57adc1/0/7f0c5be7-cb19-4bc2-90e5-30b436272826', '1:1')",];
]}
subgraph cluster_3 {  id="sid_37ceb645c042480b8a30c6b425f396ea"  style="filled,rounded"; node
[style=filled, fillcolor=white]; label = "";
 color=lightblue;
["ANID:7f72f554-89a6-4525-a63a-68e84af65284" -> "ANID:e95df9e1-a8db-46e3-8609-d84a2cad587a"
[label="M:N Hash" URL="javascript:showConnector('http://vanilla.ics.uci.edu:2099/profile/a965fe5a-ba0b-
4a01-a13d-1b4dcc57adc1/0/af0852fc-4123-4b61-9723-a0d645361230', 'M:N Hash')",];
, "ANID:e95df9e1-a8db-46e3-8609-d84a2cad587a" -> "ANID:2df02bc7-d6ef-4d61-aa5f-2ae45f1ed6e2"
[label="M:N Hash" URL="javascript:showConnector('http://vanilla.ics.uci.edu:2099/profile/a965fe5a-ba0b-
4a01-a13d-1b4dcc57adc1/0/50917f45-f3c2-47ba-8e15-a4bec7556be8', 'M:N Hash')",];
]}
"ANID:2df02bc7-d6ef-4d61-aa5f-2ae45f1ed6e2" -> "ANID:22c232fb-b785-4839-aa19-de4c1952c115"
[label="",style=dotted,arrowhead=vee,arrowsize=2,color=red];
"ANID:f37cdbb0-2947-4a8b-a5b7-aa888b9db4ad" -> "ANID:e95df9e1-a8db-46e3-8609-d84a2cad587a"
[label="",style=dotted,arrowhead=vee,arrowsize=2,color=red];
}
```

Figure 19 : Example of the HAN Graph in DOT Language


JobPlanServlet Requests JSON data ()

**URL Path** http://<cc-ip-address>:<http-port>/state/jobs/<job-id>/<attempts>/stage
http://<cc-ip-address>:<http-port>/state/jobs/<job-id>/<attempts>/plan

**Example** http://vanilla.ics.uci.edu:2099/state/jobs/5be03555-cbe0-4d30-808f-91d4e92659c3/0/stage
http://vanilla.ics.uci.edu:2099/state/jobs/5be03555-cbe0-4d30-808f-91d4e92659c3/0/plan

Process Job Plan and Job Stage JSONArray

Phrase the received Job Plan and Job Stage JSONArray to the DOT language

Submit the DOT language script to the *dot* tool (Graphviz) to draw the HAN Graph

Data Representation (Hyracks Activity Node Graph)

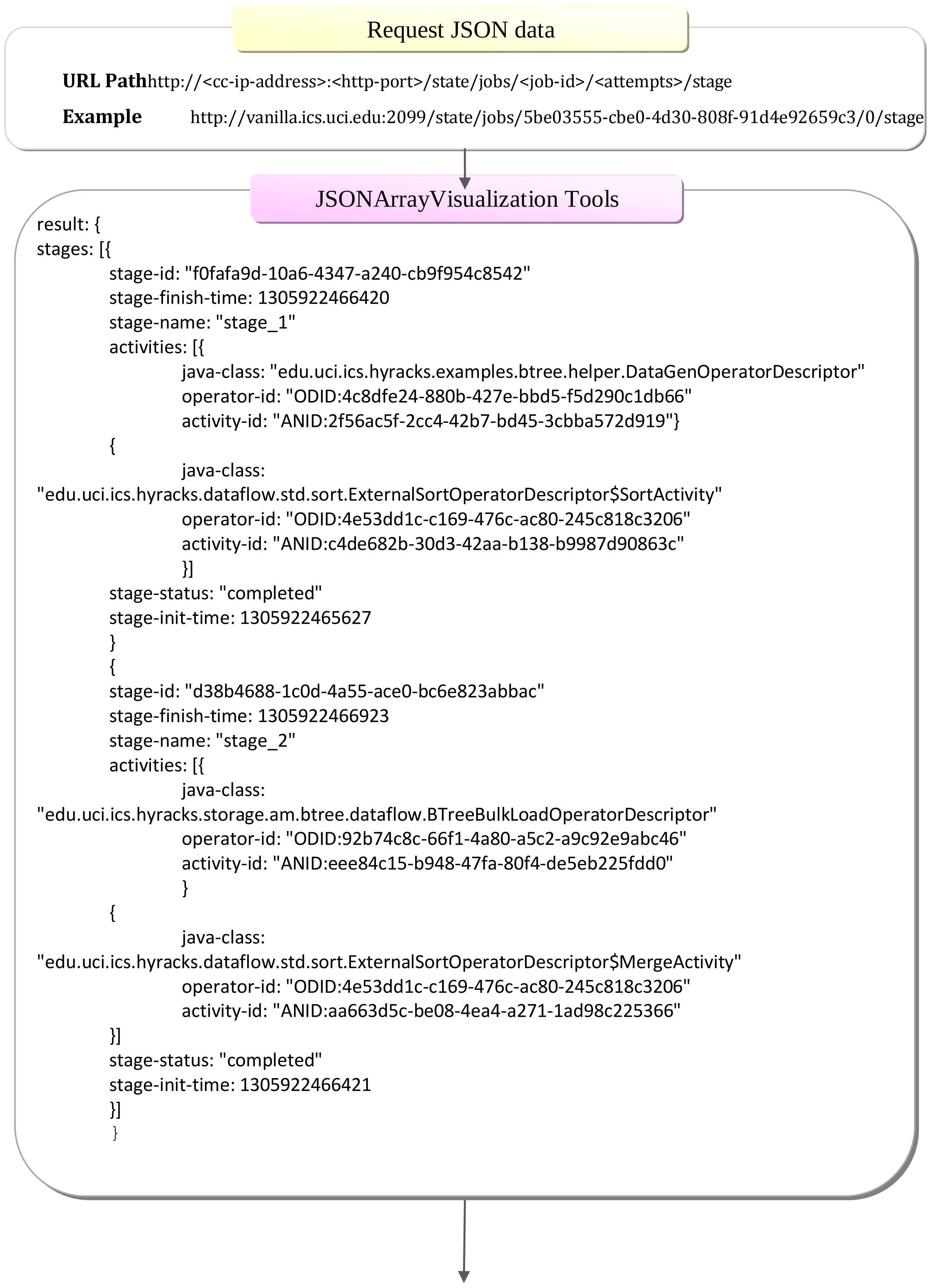
Figure 20 : Procedure for Generating Hyracks Activity Node Graphs

- **Hyracks Job Pie Chart and Hyracks Job Time Chart**

The Hyracks Console includes two additional features, the *Hyracks Job Pie Chart* and the *Hyracks Job Time Chart*, to allow users to examine the performance of their Hyracks jobs. The *Job Pie Chart* displays the time usage of each Hyracks job stage using a pie graph, and the *Job Time Chart* shows the relation between the time and the parallelism degree of each job stage by using a line graph.

For generating these two components, we choose the *Highchart* tool [29] to present information returned from the HCS. The Hightchart library written in pure JavaScript offers an easy way to add interactive charts in the web application. Currently, the Highchart library supports several types of chart such as line, spline, area, area spline, column, bar, pie and scatter chart. This tool not only allows the users to export the chart into many formats including PNG, JPG, PDF or SVG format, but also lets them to print the chart directly from the web pages.

To implement the *Hyracks Job Pie Chart* and the *Hyracks Job Time Chart* components, the HCV follows the mechanism as shown in Figure 11 to generate the data representation. Both components require the same *Job Stage* JSONArray data as an input. The HCV first sends the request to the HCS through the "Job-Stage" URL. The *Highchart* tool then parses the JSONArray of the Job Stage returned from the HCS and generates either a pie chart or a line chart in the SVG format. When the *Hightchart* tool generates the chart successfully, the graphic/chart in the SVG format is ready for the HCV to load and present it on the web page. In Figure 21, a simple process in generating the Job Pie Chart and the Job Time Chart is presented. This example Hyracks job consists of two job's stages. Most of the time spends on the first stage (61%) comparing to the second stage (39%). The numbers of the parallelism degree on Y axis (Node Partition) at stage 1 and stage 2 are both equal to four partitions.



**URL Path**http://<cc-ip-address>:<http-port>/state/jobs/<job-id>/<attempts>/stage

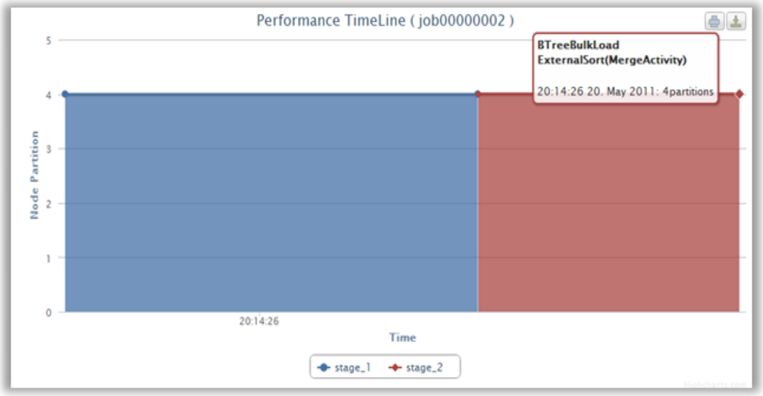
**Example** http://vanilla.ics.uci.edu:2099/state/jobs/5be03555-cbe0-4d30-808f-91d4e92659c3/0/stage

```
result: {
stages: [{
            stage-id: "f0fafa9d-10a6-4347-a240-cb9f954c8542"
            stage-finish-time: 1305922466420
            stage-name: "stage_1"
            activities: [{
                        java-class: "edu.uci.ics.hyracks.examples.btree.helper.DataGenOperatorDescriptor"
                        operator-id: "ODID:4c8dfe24-880b-427e-bbd5-f5d290c1db66"
                        activity-id: "ANID:2f56ac5f-2cc4-42b7-bd45-3cbba572d919"}
            {
                        java-class:
"edu.uci.ics.hyracks.dataflow.std.sort.ExternalSortOperatorDescriptor$SortActivity"
                        operator-id: "ODID:4e53dd1c-c169-476c-ac80-245c818c3206"
                        activity-id: "ANID:c4de682b-30d3-42aa-b138-b9987d90863c"
                        }]
            stage-status: "completed"
            stage-init-time: 1305922465627
            }
            {
            stage-id: "d38b4688-1c0d-4a55-ace0-bc6e823abbac"
            stage-finish-time: 1305922466923
            stage-name: "stage_2"
            activities: [{
                        java-class:
"edu.uci.ics.hyracks.storage.am.btree.dataflow.BTreeBulkLoadOperatorDescriptor"
                        operator-id: "ODID:92b74c8c-66f1-4a80-a5c2-a9c92e9abc46"
                        activity-id: "ANID:eee84c15-b948-47fa-80f4-de5eb225fdd0"
                        }
            {
                        java-class:
"edu.uci.ics.hyracks.dataflow.std.sort.ExternalSortOperatorDescriptor$MergeActivity"
                        operator-id: "ODID:4e53dd1c-c169-476c-ac80-245c818c3206"
                        activity-id: "ANID:aa663d5c-be08-4ea4-a271-1ad98c225366"
            }]
            stage-status: "completed"
            stage-init-time: 1305922466421
            }]
            }
```

(Continue with next page)

Figure 21 : Job Pie Chart and Job Time Chart (1)

Data Representation (Job Time & Job Pie)



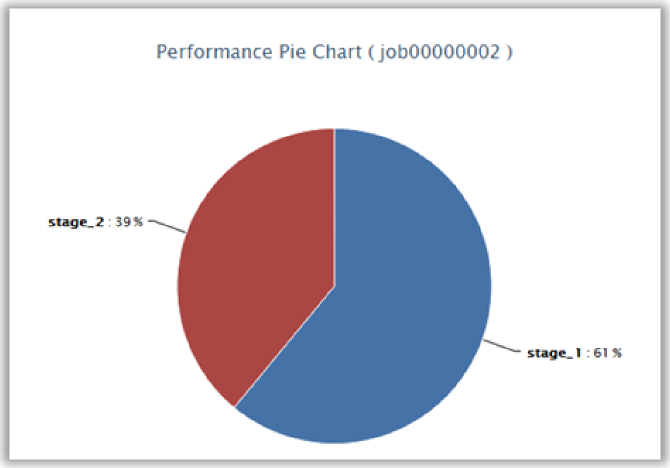


Figure 22 : Job Pie Chart and Job Time Chart (2)

### 5.3.2 Hyracks Cluster Health

Every Hyracks cluster consists of two types of nodes: Hyracks Cluster Controller (CC) and Hyracks Node Controllers (NCs). For information related to the CC, the HCV provides following information: the *CC configuration parameters*, a list of *Registered Nodes,* and a *Job Time Guilder*. For each NC, the HCV offers: *NC configuration parameters*, a list of *running* and *completed jobs,* and *Node Resources Monitoring* information.

- **CC Configuration**

The Hyracks Console provides read access to the CC Configuration as static information that consists of the CC's configuration variables, i.e., "cc-port", "http-port", "heartbeat-period", "max-heartbeat-lapse-period", "default-max-job-attempts", and "profile-dump-period". In addition, this feature displays a list of the Hyracks applications deployed in a given Hyracks cluster. Note that these configuration values cannot be changed unless users restart the CC process. The data visualization for this static information is simple. The HCV requests the CC configuration data from the HCS via the CC configuration URL path. The HCV then displays the returned results in an HTML table. An example result is shown in Figure 23.

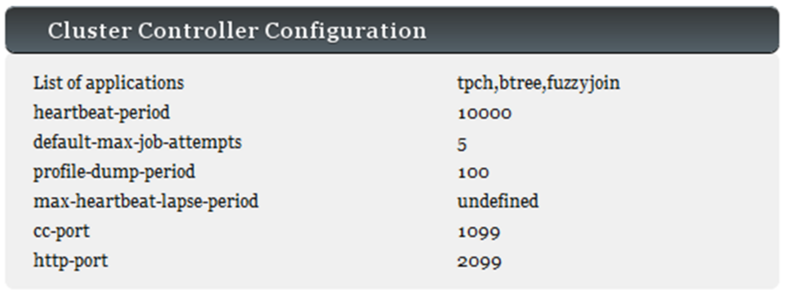
Request JSON

**URL Path**http://<cc-ip-address>:<http-port>/console/cluster
**Example** http://vanilla.ics.uci.edu:2099/console/cluster

CC Configuration JSONArray

```
{
    result: {
        cc-config: {
            heartbeat-period: 10000
            default-max-job-attempts: 5
            profile-dump-period: 100
            max-heartbeat-lapse-periods: 5
            cc-port: 1099
            type: "cc-config"
            http-port: 2099
        }
        app-name: [
            "tpch"
            "btree"
            "fuzzyjoin"
        ]
        applications: [
            {
                application-root-dir: "edu.uci.ics.hyracks.control.cc. ClusterControllerService/applications/tpch"
                application-name: "tpch"
                created-at: "2011-05-20 13:12:06.425"
                initialized-at: "2011-05-20 13:12:09.799"
            }
            {
                application-root-dir: "edu.uci.ics.hyracks.control.cc. ClusterControllerService/applications/btree"
                application-name: "btree"
                created-at: "2011-05-20 13:13:48.735"
                initialized-at: "2011-05-20 13:13:48.881"
            }
            {
                application-root-dir: "edu.uci.ics.hyracks.control.cc. ClusterControllerService/applications/fuzzyjoin"
                application-name: "fuzzyjoin"
                created-at: "2011-05-20 13:12:59.933"
                initialized-at: "2011-05-20 13:13:00.580"
            }
        ]}
    }
```

Data Representation (CC Configuration)



**Cluster Controller Configuration**

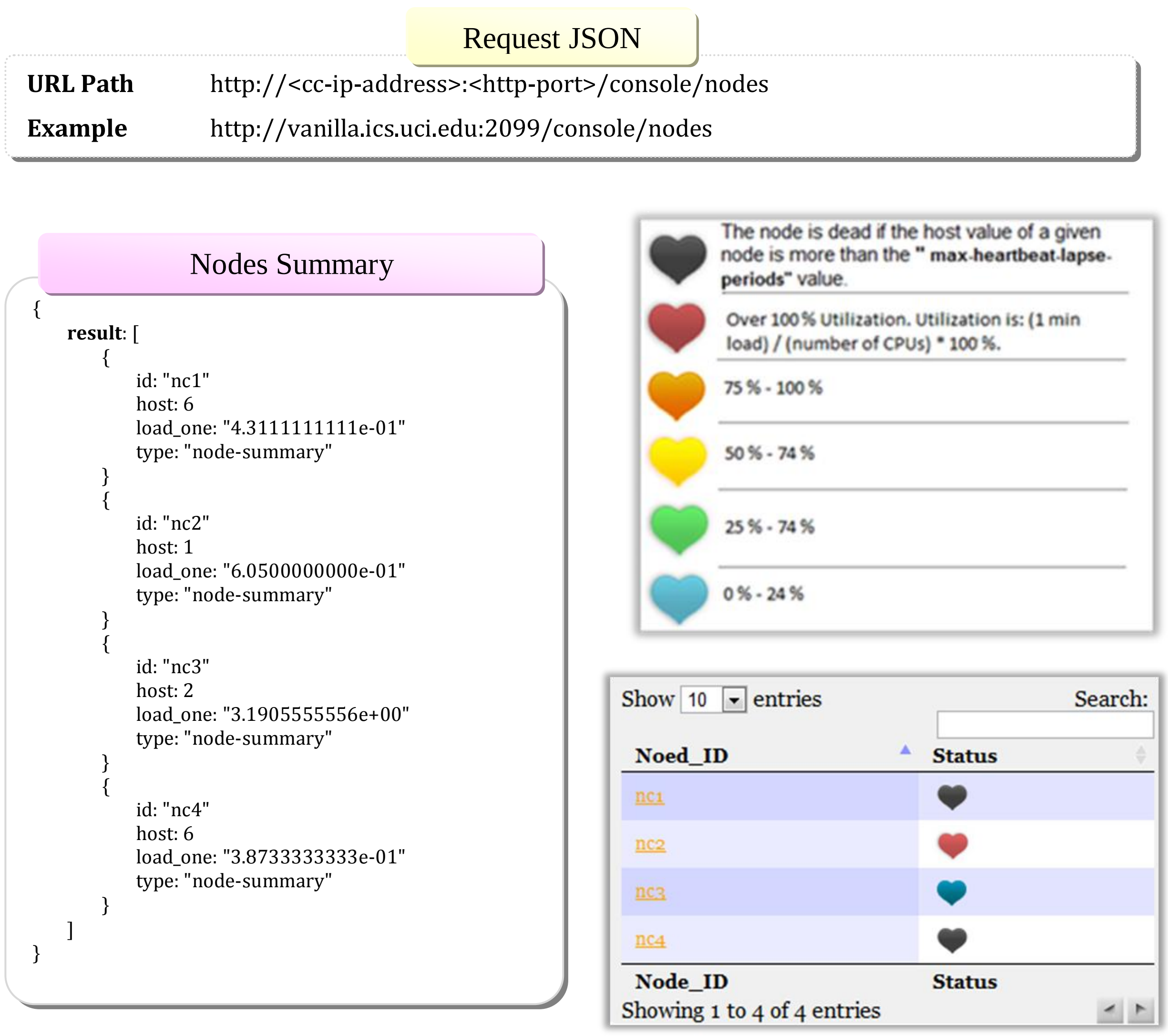
| | |
|---|---|
| List of applications | tpch,btree,fuzzyjoin |
| heartbeat-period | 10000 |
| default-max-job-attempts | 5 |
| profile-dump-period | 100 |
| max-heartbeat-lapse-period | undefined |
| cc-port | 1099 |
| http-port | 2099 |

Figure 23 : Procedure for Generating the CC Configuration Table

- **Registered Nodes**

Another component related to the Hyracks Cluster Controller is a list of registered NCs. The NC nodes are managed by the CC and used to execute the partitions of tasks as schedulable by the CC. This component provides an overall picture of the health status of each registered NC node.

To generate this information, the HCV needs to fetch the *Node Controllers Summary* JSON data from the HCS and present the returned results in a table using the DataTables jQuery tool. An example result is shown in Figure 24. DataTables was selected to be the data presenter because it has nice scalability characteristics. Although our example shows only four registered nodes, in reality, this component is able to support thousands of nodes.



Request JSON

| | |
|---|---|
| **URL Path** | http://<cc-ip-address>:<http-port>/console/nodes |
| **Example** | http://vanilla.ics.uci.edu:2099/console/nodes |

Nodes Summary

```
{
    result: [
        {
            id: "nc1"
            host: 6
            load_one: "4.3111111111e-01"
            type: "node-summary"
        }
        {
            id: "nc2"
            host: 1
            load_one: "6.0500000000e-01"
            type: "node-summary"
        }
        {
            id: "nc3"
            host: 2
            load_one: "3.1905555556e+00"
            type: "node-summary"
        }
        {
            id: "nc4"
            host: 6
            load_one: "3.8733333333e-01"
            type: "node-summary"
        }
    ]
}
```

| | |
|---|---|
| 🖤 | The node is dead if the host value of a given node is more than the " **max-heartbeat-lapse-periods**" value. |
| ❤️ | Over 100% Utilization. Utilization is: (1 min load) / (number of CPUs) * 100 %. |
| 🧡 | 75 % - 100 % |
| 💛 | 50 % - 74 % |
| 💚 | 25 % - 74 % |
| 💙 | 0 % - 24 % |

Show 10 entries Search:

| Noed_ID | Status |
|---|---|
| nc1 | 🖤 |
| nc2 | ❤️ |
| nc3 | 💙 |
| nc4 | 🖤 |
| **Node_ID** | **Status** |

Showing 1 to 4 of 4 entries

Figure 24 : Example List of Registered Nodes Component

- **Hyracks Jobs Timeglider**

This module displays all of the Hyracks jobs that have been submitted to the monitored cluster over time. The information shown in this module is actually comes from the same Job Summary JSONArray that was discussed in the Hyracks Job Summary component. While the Job Summary component divides up the Hyracks jobs based on the jobs' status, the Hyracks Jobs Timeglider offers another perspective by using a timeline to present all Hyracks jobs. The *Timeglider* tool [28] is a data-driven interactive timeline component which is excellent for historical projects and project planning information. It is an open-source JavaScript/jQuery plug-in with an MIT license. The current version of the Timeglider component is fed time-series data from a JSON file; hence, the original data that is received in the console from the Job Summary URL has to be reorganized into another structure to accomplish the Timeglider tool's requirement. After the HCV requests and receives the Job Summary JSONArray from the HCS, it uses the *TimelineSeverlet* to propose the format for showing the data received. The Timeglider tool then loads the resulting JSON data and presents the interactive timeline in a Web page. Figure 25 shows the partial data of the JobTimegliderServlet.json file generated by *TimelineServlet* and the resulting screenshot of the Timeglider visualization that represents one Hyracks Job, "job00000005".

Request JSON

**URL Path** http://<cc-ip-address>:<http-port>/state/jobs

**Example** http://vanilla.ics.uci.edu:2099/state/jobs

JobTimegliderServlet.json

```
[{
            id: "ccjobhistory",
            initial_zoom: "5"
            title: "TimeLine of Jobs in vanilla.ics.uci.edu:2099"
            events: [{
            id: "eb444563-311f-469f-9874-2705cf3bccda"
            icon: "TERMINATED.png"
            title: "job00000005"
            startdate: "2011-05-20 13:15:50"
            importance: 10
            description: "Status: TERMINATED, Application: btree ,start:2011-05-20 13:15:50 end:2011-05-20
            13:15:51"
            link: "JobProfile.jsp?jobid=eb444563-311f-469f-9874-
            2705cf3bccda&jobName=job00000005&attempts=1&status=TERMINATED"
            enddate: "2011-05-20 13:15:51"
}]
```

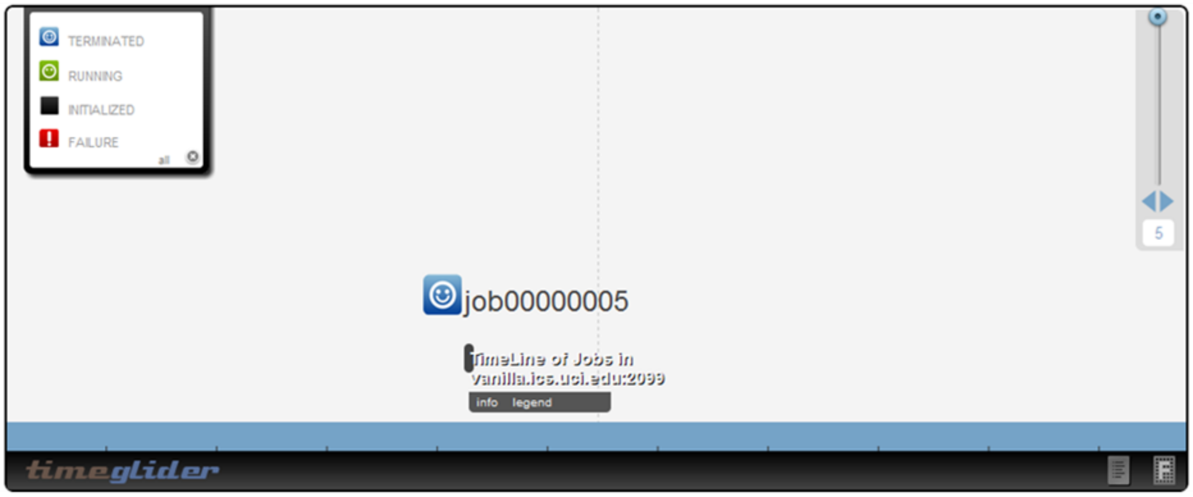


Figure 25 : Hyracks Jobs Timeglider

- **Node Controller configuration (NC-config)**

Similar to the CC configuration module, this NC-config module displays all configuration variables and then values for a given node controller i.e., “node-id”, “frame-size”, “cc-host”, “cc-port”, “dcache-client-path”, “io-devices”, “data-ip-address”, and “dcache-client-servers”. The NC configuration cannot be changed unless the users restart the NC process. The data representation mechanism of the NC Configuration is similar to the CC configuration as shown in Figure 26. The HCV requests the NC Configuration URL from the HCS and presents the returned data in the HTML table.

Request JSON

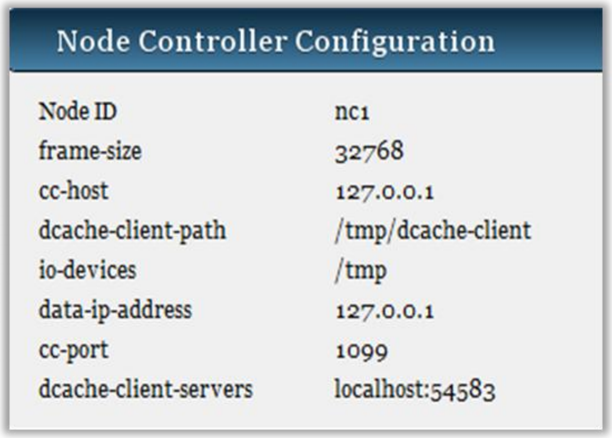
**URL Path**http://<cc-ip-address>:<http-port>/console/nodes/<node-id>/config

**Example** http://vanilla.ics.uci.edu:2099/console/nodes/nc1/config

NC Configuration JSONArray

```
{
    result: {
        id: "nc1"
        frame-size: 32768
        cc-host: "127.0.0.1"
        dcache-client-path: "/tmp/dcache-client"
        io-devices: "/tmp"
        data-ip-address: "127.0.0.1"
        cc-port: 1099
        type: "node-config"
        dcache-client-servers: "localhost:54583"
    }
}
```

Data Representation (NC Configuration)



Node Controller Configuration

| | |
|---|---|
| Node ID | nc1 |
| frame-size | 32768 |
| cc-host | 127.0.0.1 |
| dcache-client-path | /tmp/dcache-client |
| io-devices | /tmp |
| data-ip-address | 127.0.0.1 |
| cc-port | 1099 |
| dcache-client-servers | localhost:54583 |

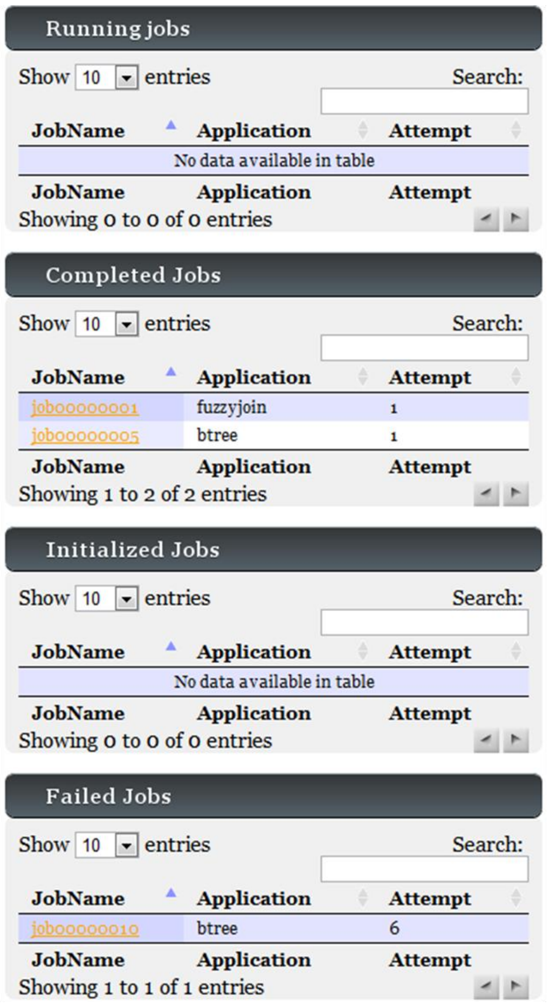
Figure 26 : Procedure in Generating the NC Configuration Table

- **Node Controller Jobs Summary**

For each of the NCs in the cluster, the HCV provides a *Node Controller Jobs Summary (NC Jobs Summary)* module. This module consists of a list of all Hyracks jobs that a given node is participated in. All executed jobs are classified into four categories based on their current status i.e., `INITIALIZED`, `RUNNING`, `TERMINATED`, and `FAILURE`. This information allows users to understand the performance and workload of each NC. To generate this module, the HCV uses the *Node Controller Jobs Summary* URL-Request provided by the HCS and the *DataTables* jQuery tool. The procedure for generating the *NC Jobs Summary* module is illustrated in Figure 27. There are two finished jobs ("job00000001" and "job00000005") and one failed job ("job00000010") that the node "nc1" is participated in.

Request JSON

**URL Path** http://<cc-ip-address>:<http-port>/console/nodes/<node-id>/jobs
**Example** http://vanilla.ics.uci.edu:2099/console/nodes/nc1/jobs



Running jobs

Show 10 entries Search:

| JobName | Application | Attempt |
|---|---|---|
| No data available in table | | |

Showing 0 to 0 of 0 entries

Completed Jobs

Show 10 entries Search:

| JobName | Application | Attempt |
|---|---|---|
| job00000001 | fuzzyjoin | 1 |
| job00000005 | btree | 1 |

Showing 1 to 2 of 2 entries

Initialized Jobs

Show 10 entries Search:

| JobName | Application | Attempt |
|---|---|---|
| No data available in table | | |

Showing 0 to 0 of 0 entries

Failed Jobs

Show 10 entries Search:

| JobName | Application | Attempt |
|---|---|---|
| job00000010 | btree | 6 |

Showing 1 to 1 of 1 entries

```
{
    result: {
        failed-jobs: {
            job-id: "54d395c4-289f-4de9-ab54-c429b006a968"
            application: "btree"
            display-name: "job00000010"
            status: "FAILURE"
            attempts: 6
            start-time: "2011-05-20 13:56:15.041"
            end-time: "2011-05-20 14:07:08.659"
        }
        finish-jobs: [
            {
                job-id: "eb444563-311f-469f-9874-2705cf3bccda"
                application: "btree"
                display-name: "job00000005"
                status: "TERMINATED"
                attempts: 1
                start-time: "2011-05-20 13:15:50.536"
                end-time: "2011-05-20 13:15:51.045"
            }
            {
                job-id: "17159a0c-0ae9-414d-9553-928a9b42a12d"
                application: "fuzzyjoin"
                display-name: "job00000001"
                status: "TERMINATED"
                attempts: 1
                start-time: "2011-05-20 13:13:26.969"
                end-time: "2011-05-20 13:13:27.962"
            }
        ]
    }
}
```

Figure 27 : Node Controller Job Summary Module

- **Node Controller Resource Monitoring**

In the Hyracks system, the resource usage of each worker machine is beneficial for the Hyracks users to monitor as a measure of the Hyracks cluster's health and to help detect any job execution failures caused by regarding node failures. The *NC Resource Monitoring* module presents a line chart of time-series data of the resource consumption on a given worker machine. This resource consumption data is actually collected by the *Ganglia* software and stored in the *RRD* database. The HCS is responsible for fetching this time-series data from the RRD database and sending that data in the JSON format to the HCV when the HCV requests it via the NC-Resource URL. The HCV can specific the type of requested resource in the NC-Resource URL from the following five main categories: CPU (cpu_idle, cpu_nice, cpu_system, cpu_user and cpu_wio), DISK (disk_free and disk_total), LOAD (load_one, cpu_num, proc_run and proc_total), NETWORK (bytes_in, bytes_out, pkts_in and pkts_out), and MEM (mem_bufferes, mem_cached, mem_free, mem_shared and mem_total). For the NC-Resource URL, the time resolution <step> for the resource data is every 15 seconds, and the <end-time> is the current time by default. To minimize the data loading time, the number of records returned to the HCV is about 30 records per type of resource. To be more precise, the default resource monitoring URL returns resource usage data in the last 450 (=15*30) seconds. Sample memory usage for the node "nc1" is shown in Figure 28.

For visualizing the time-series resource data, we use the Highchart tool to display a data in an interactive line chart. The horizontal axis is time and the vertical axis is a resource consumption value. The result is presented in five tabs, i.e., "all", "mem", "cpu", "disk", and "network". Each tab represents a resource category. This interactive line chart not only gives a visual overview of resource usage, but it also allows users to check the actual values at each point in the chart by using a mouse-over action. In addition, users can filter the content of the line chart by selecting the sub-types or resource of interest. They also are able to zoom in/out in time by dragging their mouse over the line chart.

Request JSON

**URL Path**
http://<cc-ip-address>:<http-port>/console/nodes/ <node-id>/resources/<type>/ (<step>)/(<start-time>)/(<end-time>)
**Example**
http://vanilla.ics.uci.edu:2099/console/nodes/nc1/resources/mem

```
{
    result: [
        {
            type: "mem"
            sub-type: "mem_buffers"
            count: 30
            data: [
                {
                    num: "1.0000000000e+00"
                    time: "06:51:45 PM"
                    value: "2.4786666667e+03"
                }
                {
                    num: "1.0000000000e+00"
                    time: "06:52:00 PM"
                    value: "2.5160000000e+03"
                }
                {
                    num: "1.0000000000e+00"
                    time: "06:52:15 PM"
                    value: "2.5160000000e+03"
                }
                …
                …
                …
            ]
        }
          (continue on the right figure)
```

```
          (continue from the left figure)
        {
            type: "mem"
            sub-type: "mem_cached"
            count: 30
            data: [
                {
                    num: "1.0000000000e+00"
                    time: "06:51:45 PM"
                    value: "6.1557333333e+04"
                }
                {
                    num: "1.0000000000e+00"
                    time: "06:52:00 PM"
                    value: "6.1560000000e+04"
                }
                {
                    num: "1.0000000000e+00"
                    time: "06:52:15 PM"
                    value: "6.1560000000e+04"
                }
                …
                …
                …
            ]
        }
    ]
}
        (Note: this picture includes only part of data)
```

Data Representation(Resources Monitoring)

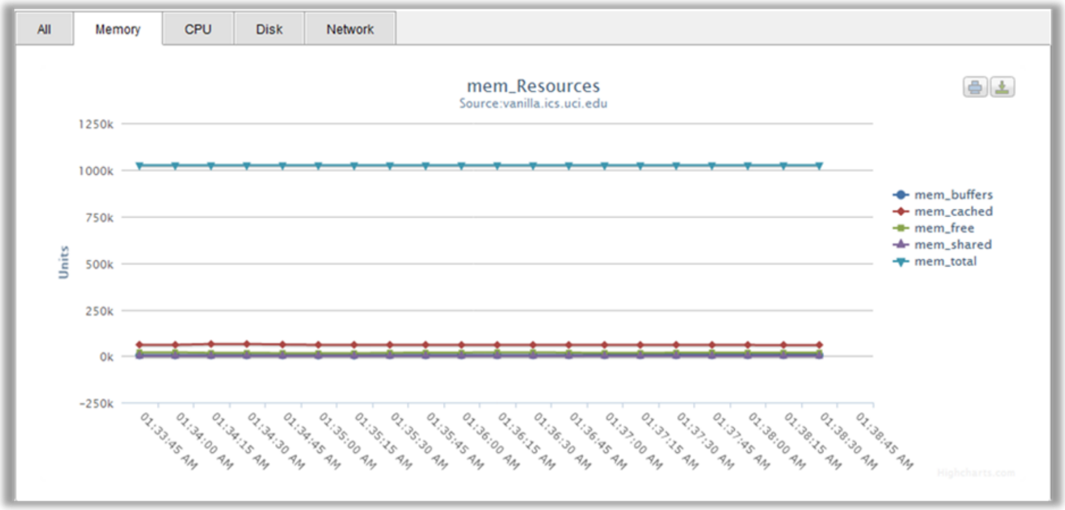


Figure 28 : NC Resource Monitoring Module

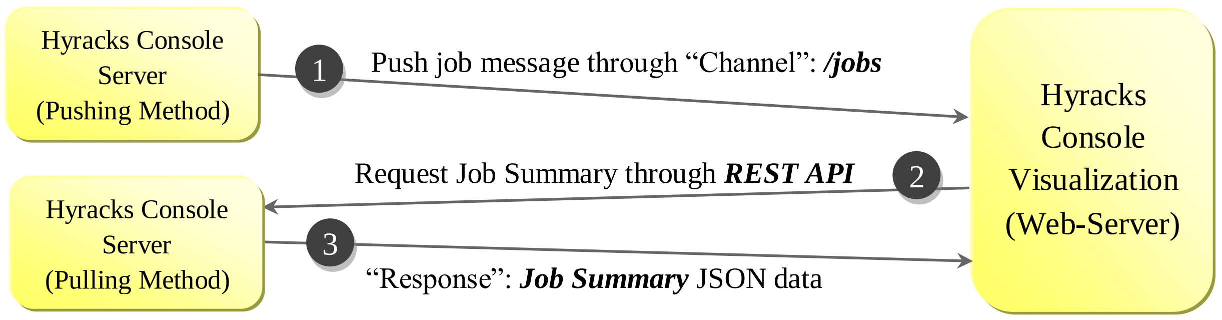
## 5.4 Publish/Subscribe API for Server Pushing Method

One of our design goals of the Hyracks Console is to monitor the Hyracks system in "real-time". Many real-time web-based monitoring systems set up a timer to update their sites automatically on a periodic basis. However, if we stand on the user's side, they might not want their Webpage to keep refreshing periodically without any change of the content. To be more precise, a user's web page should be updated only when any new resource information in related to that particular web page is generated. To support this feature, an additional web-server is implemented in the console to support *a server-push mechanism* that intelligently responds to significant events in the Hyracks system at runtime.

Instead of having the Hyracks Console Visualization (HCV) periodically pull for the data from the Hyracks Console Server (HCS), a web-server on the HCS side initiates the communication and publishes a message to the HCV via a specific "channel" when any significant event occurs in the Hyracks system. To receive the message, the HCV has to subscribe/listen to that particular channel. Thus, when a user opens a Hyracks Console web page that requires this feature, the HCV automatically subscribes to the related channel using the CometD service. When the HCV receives a message from the HCS, it will update the visual modules that are relevant to this message. In the HCS, there are three such channels implemented to interact with different types of events as shown below.

- **The /jobs channel** is responsible for transmitting the event messages when there is an event related to the set of Hyracks jobs, such as *JobCreateEvent, JobStartEvent, JobAbortEvent, JobAttemptStartEvent,* and *JobCleanupEvent.* For example, when a new job is submitted to the Hyracks cluster, the HCS-Push method publishes a "job" message to the */job's* channel to trigger the HCV server to reload the *Job Browser*[1] page by requesting Job Summary data from the HCS-Pull method through the REST API. The HCV then updates the Job Browser page by using the new Job Summary JSON data. Figure 29 summaries this concept.

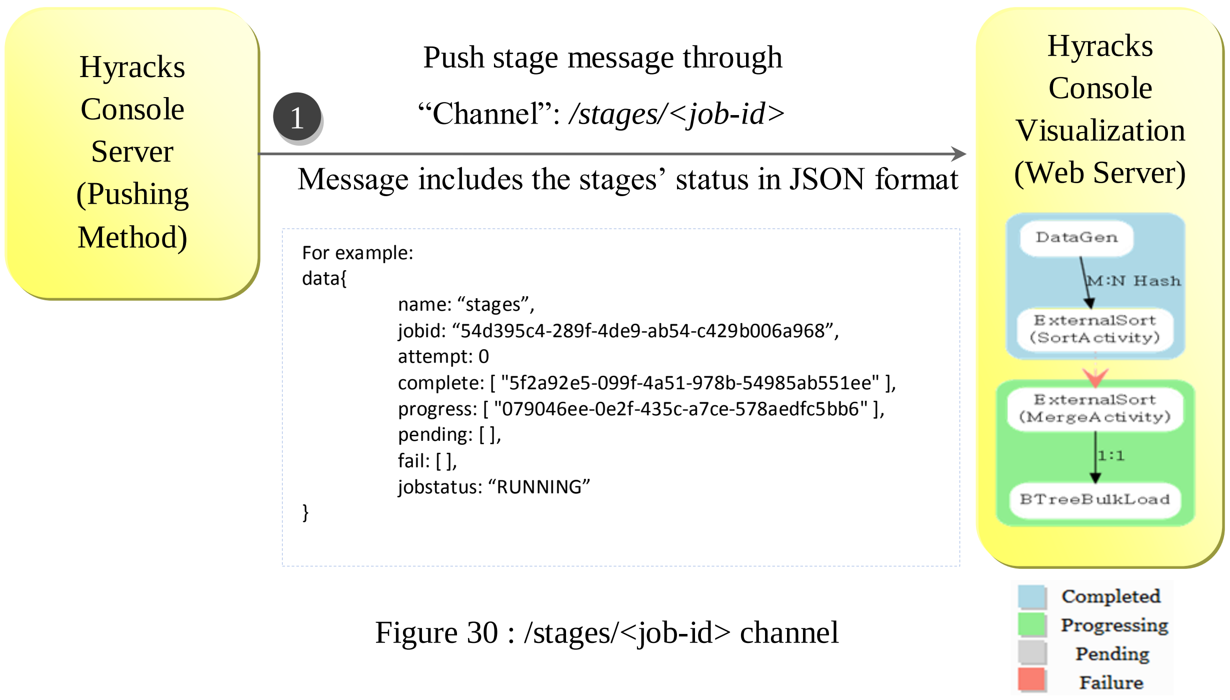
[1] The *Job Browser* page displays the tables of all Hyracks jobs that have been submitted to a particular Hyracks cluster. An example of this page can be found in Section 6.2.1.



Hyracks Console Server (Pushing Method)
Hyracks Console Server (Pulling Method)
1
Push job message through "Channel": /jobs
Request Job Summary through REST API
2
3
"Response": Job Summary JSON data
Hyracks Console Visualization (Web-Server)

Figure 29 : /jobs channel

- **The /stages/<job-id> channel** is responsible for transmitting event messages that are related to the stage of a particular monitored Hyracks job, such as *ScheduleRunableStagesEvent*, *StageletFailureEvent*, *JobAttemptStartEvent*, *JobAbortEvent*, and *JobCleanupEvent*. When the stage of a monitored job is updated, the HCS sends a message through this channel to the HCV server. This "stage" message includes several information units such as the job's ID, attempt number, list of completed/ progressing/ pending/ failed stages' ID, and current job's status. Furthermore, the HCV web-server uses a JavaScript function to assign the color of each stage in the *Job Activity Node Graph* module as shown in Figure 23.The "light blue", "light green", "gray" and "salmon" colors represent a completed stage, progressing stage, pending stage, and failed stage respectively. With this server push mechanism, the Hyracks user can see a real-time animation of the *Job Activity Node Graph* via the web interface.


Hyracks Console Server (Pushing Method)
1
Push stage message through
"Channel": /stages/<job-id>
Message includes the stages' status in JSON format
Hyracks Console Visualization (Web Server)
DataGen
M:N Hash
ExternalSort (SortActivity)
ExternalSort (MergeActivity)
1:1
BTreeBulkLoad
Completed
Progressing
Pending
Failure

```
For example:
data{
            name: "stages",
            jobid: "54d395c4-289f-4de9-ab54-c429b006a968",
            attempt: 0
            complete: [ "5f2a92e5-099f-4a51-978b-54985ab551ee" ],
            progress: [ "079046ee-0e2f-435c-a7ce-578aedfc5bb6" ],
            pending: [ ],
            fail: [ ],
            jobstatus: "RUNNING"
}
```

Figure 30 : /stages/<job-id> channel

- **The /attempts/<job-id> channel** is used when the monitored job starts a new attempt for its execution (JobAttempStartEvent). As was mentioned in the Hyracks overview, if an error occurs during a job's execution, the Hyracks system will try to restart the process again until reaching the maximum number of attempts. The push mechanism of the */attempt/<job-id>* channel is similar to the */jobs* channel. The "attempt" message from the HCS-push will trigger the HCV sever to request the *JobPlan* and *JobStage* JSON data via the HCS-Pull API and then reload the *Job Profile*[2] page. This is shown in Figure 31.

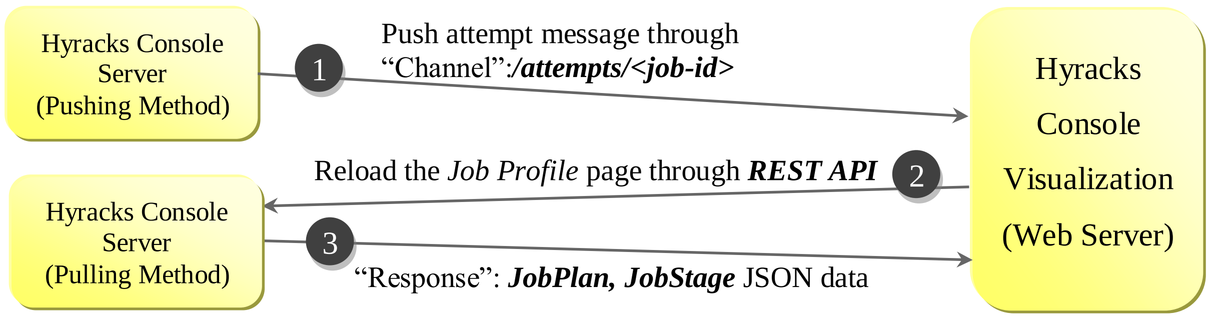

Hyracks Console Server (Pushing Method)
1
Push attempt message through
"Channel":/attempts/<job-id>
Hyracks Console Visualization (Web Server)
Reload the Job Profile page through REST API
2
Hyracks Console Server (Pulling Method)
3
"Response": JobPlan, JobStage JSON data

Figure 31 : /attempts/<job-id> channel

[2] The Job Profile page presents several modules related to a particular Hyracks job such as its Job Specification, Job Plan, Job Pie Chart, and Job Time Chart modules. An example of this page can be found in Section 6.2.2

# 6 HYRACKS CONSOLE USER MANUAL

This section explains in detail of how to install and run the Hyracks Console. In addition, we demonstrate how to use the Hyracks Console to monitor the behavior of the Hyracks Cluster based on the real example throughout the section. This section is aimed at readers using to understand how to use the Hyracks Console.

## 6.1 Setting up the Hyracks Console

The Hyracks Console is made up of two parts, the Hyracks Console Visualization (HCV) and the Hyracks Console Server (HCS) components. The HCV is used for remotely viewing live and historical statistics (such as CPU load averages or network utilization) of the Hyracks cluster. The HCS source code is included in the Hyracks system package. The HCS component of the Hyracks system is activated automatically after the Hyracks Cluster Controller process is started. For using the Hyracks Console, the only requirement is the users need to install and run the console on the machine that is able to connect to a machine which is running the Hyracks Cluster Controller process of the monitored Hyracks cluster.

### 6.1.1 Hyracks Cluster Installation Prerequisites for Collecting and Storing Data

Before starting the Hyracks Console, the *Ganglia* software should be installed and be runnable on every machine in the Hyracks cluster to collect the time-series data for the resource consumption (i.e., CPU, disk, network, and load) of each machine. This data is required for generating the Node Controller Resource Monitoring module. However, if the user chooses not to install Ganglia, the Hyracks console will notify them by annotate "the Ganglia software is missing" on that visual module. The process of collecting resource usage data is independent of the Hyracks job's execution, so the user will still be able to monitor all other aspects of the Hyracks jobs without installing Ganglia.

### 6.1.2 Installing JSP

In order to run Java Server Pages, to deploy and run, a compatible web server with servlet container is required. There are a number of JSP technology implementations for different web servers. The latest information on officially-announced support can be found at http://java.sun.com/products/jsp/industry.html. In this implementation, the Hyracks Console chooses the Apache Tomcat server to support JSP.

### 6.1.3 Running the Hyracks Console on a Web Browser

The Hyracks Console Visualization (HCV) is a client-server web application that is accessible over a network. HCV is implemented using a browser-support language which is reliant on a common web browser to render the application executable. This section helps Hyracks users to start the Hyracks Console and setup the connection between HCV and HCS.

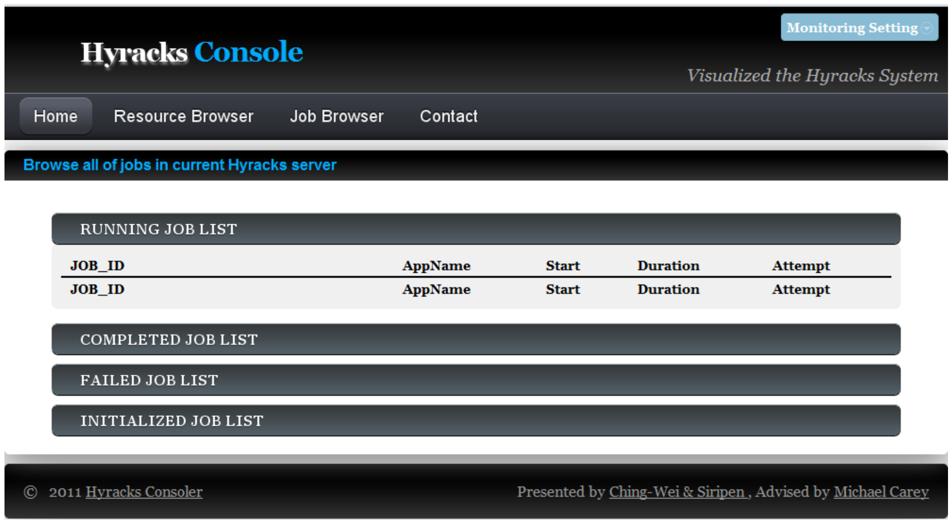
**Prerequisites**

- Google Chrome web browser (or equivalent, e.g., Firefox)

**Steps**

1) Open a web browser
2) Load this following URL: "http://localhost:<webserver-port>/HyracksConsole/web" where the <web server port> is the network port running the Tomcat Server

   When the Hyracks Console Visualization (HCV) is loaded successfully, the user should see an initial web page (Job Browser Page) as shown in Figure 32. Notice that the content in the table is empty since the connection between the HCV and the HCS is not setup yet.



| JOB_ID | AppName | Start | Duration | Attempt |
|---|---|---|---|---|
| JOB_ID | AppName | Start | Duration | Attempt |

Figure 32 : Initial Web Page (Job Browser Page) of the Hyracks Console

3) Setup the connection between the HCV and the HCS

We assume that the IP address of the master node running the Hyracks Cluster Controller of the monitored Hyracks system is “vanilla.ics.uci.edu” and the http-port number is “2099”. Users can setup the HCV/HCS connection by following these steps.

a. Click on “Monitoring Setting” button on the top right as shown in Figure 33
b. Fill the *cc-IP* as *vanillia.ics.uci.edu* and *cc-http-port* as *2099* as shown in Figure 34.
c. Click on “Save”

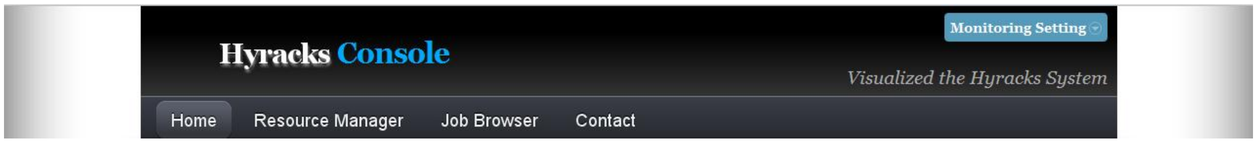
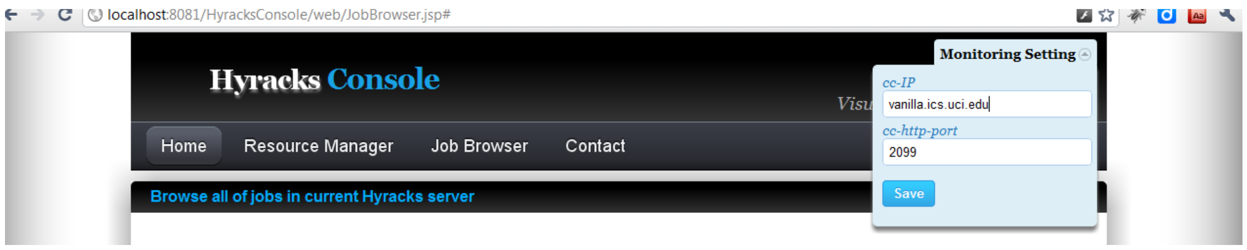


Figure 33 : Setup the Connection between the HCV and the HCS

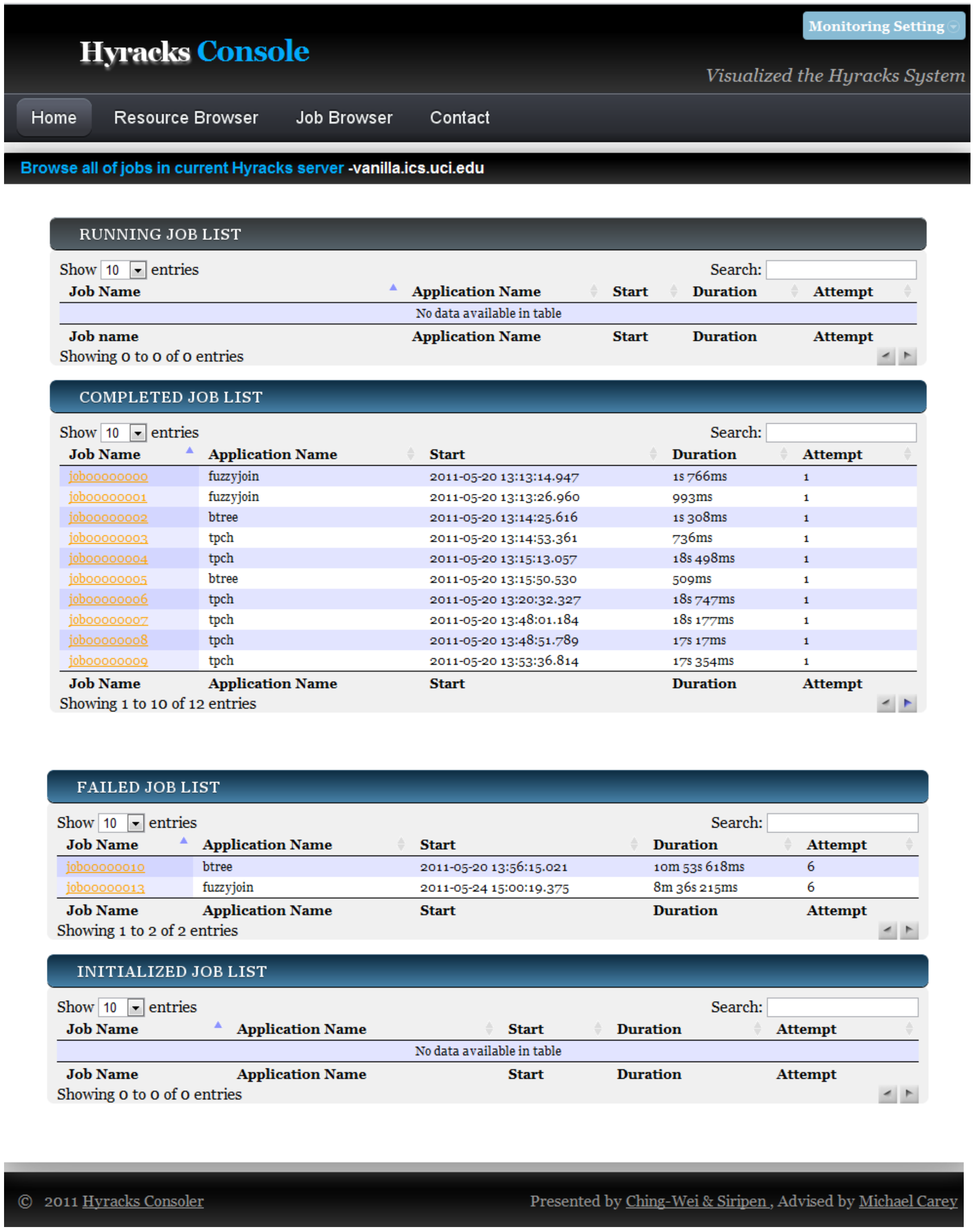

localhost:8081/HyracksConsole/web/JobBrowser.jsp#

Browse all of jobs in current Hyracks server

cc-IP: vanilla.ics.uci.edu

cc-http-port: 2099

Figure 34 : Setup the *cc-IP* and *cc-http-port* values


Hyracks Console

Monitoring Setting

Visualized the Hyracks System

Home | Resource Browser | Job Browser | Contact

Browse all of jobs in current Hyracks server -vanilla.ics.uci.edu

RUNNING JOB LIST

Show 10 entries Search:

| Job Name | Application Name | Start | Duration | Attempt |
|---|---|---|---|---|
| No data available in table | | | | |
| Job name | Application Name | Start | Duration | Attempt |

Showing 0 to 0 of 0 entries

COMPLETED JOB LIST

Show 10 entries Search:

| Job Name | Application Name | Start | Duration | Attempt |
|---|---|---|---|---|
| job0000000000 | fuzzyjoin | 2011-05-20 13:13:14.947 | 1s 766ms | 1 |
| job0000000001 | fuzzyjoin | 2011-05-20 13:13:26.960 | 993ms | 1 |
| job0000000002 | btree | 2011-05-20 13:14:25.616 | 1s 308ms | 1 |
| job0000000003 | tpch | 2011-05-20 13:14:53.361 | 736ms | 1 |
| job0000000004 | tpch | 2011-05-20 13:15:13.057 | 18s 498ms | 1 |
| job0000000005 | btree | 2011-05-20 13:15:50.530 | 509ms | 1 |
| job0000000006 | tpch | 2011-05-20 13:20:32.327 | 18s 747ms | 1 |
| job0000000007 | tpch | 2011-05-20 13:48:01.184 | 18s 177ms | 1 |
| job0000000008 | tpch | 2011-05-20 13:48:51.789 | 17s 17ms | 1 |
| job0000000009 | tpch | 2011-05-20 13:53:36.814 | 17s 354ms | 1 |
| Job Name | Application Name | Start | Duration | Attempt |

Showing 1 to 10 of 12 entries

FAILED JOB LIST

Show 10 entries Search:

| Job Name | Application Name | Start | Duration | Attempt |
|---|---|---|---|---|
| job0000000010 | btree | 2011-05-20 13:56:15.021 | 10m 53s 618ms | 6 |
| job0000000013 | fuzzyjoin | 2011-05-24 15:00:19.375 | 8m 36s 215ms | 6 |
| Job Name | Application Name | Start | Duration | Attempt |

Showing 1 to 2 of 2 entries

INITIALIZED JOB LIST

Show 10 entries Search:

| Job Name | Application Name | Start | Duration | Attempt |
|---|---|---|---|---|
| No data available in table | | | | |
| Job Name | Application Name | Start | Duration | Attempt |

Showing 0 to 0 of 0 entries

© 2011 Hyracks Consoler

Presented by Ching-Wei & Siripen , Advised by Michael Carey

Figure 35 : Example Job Browser Page

Figure 35 depicts an example of the Job Browser Page when the HCV successfully creates a connection to the HCS of a given Hyracks System. There is no running or initialized Hyracks job in the figure, but there are twelve completed Hyracks jobs and two failed Hyracks jobs.

## 6.2 Hyracks Console Web UI

The HCV generates dynamic web content by using *Java Server Pages (JSP)* technology. The entire sitemap of the HCV is presented in Figure 36. There are four main pages: *Hyracks Cluster Browser, Hyracks Node Controller Profile, Hyracks Job Browser* and *Hyracks Job Profile*. The top level of the Hyracks console is conceptually divided into two parts, Hyracks Cluster Browser and Hyracks Job Browser. Users can navigate from the Hyracks Cluster Browser page to the Hyracks Node Controller Profile page and from the Hyracks Job Browser page to the Hyracks Job Profile page. Moreover, they can navigate back and forth between the Hyracks Node Controller Profile page and the Hyracks Job Profile page. Table 1 shows the list of contents or modules presented on each web page. Throughout this section, a simple Hyracks job is used as an example to demonstrate how the Hyracks Console monitors the behavior of the Hyracks system. We assume that the *cc-IP* is *vanilla.ics.uci.edu* and the *cc-http-port* is *2099*.

Table 1: Hyracks Console Web Pages Description

| Page Name | Description | Contents/ Modules |
|---|---|---|
| Hyracks Job Browser | Presents all Hyracks Jobs that have been submitted to a particular Hyracks cluster. | • Running Jobs<br>• Finished Jobs<br>• Initialized Jobs<br>• Failed Jobs |
| Hyracks Job Profile | Presents more detail for a particular Hyracks job. | • Hyracks Job Specification Graph<br>• Hyracks Activity Node Graph<br>• Time Usage Report in Pie-Chart<br>• # of Partitions in Timeline<br>• Resources Consumption |
| Hyracks Cluster Browser | Presents an overview for a particular Hyracks Cluster including both Hyracks nodes view and Hyracks jobs view. | • Cluster Controller Configuration<br>• Registered Node Controllers & Node Health<br>• Hyracks Job Execution Timeline |
| Hyracks Node Controller Profile | Presents information related to a particular Hyracks Node Controller. | • Node Controller Configuration<br>• Running Jobs<br>• Finished Jobs<br>• Initialized Jobs<br>• Failed Jobs<br>• Machine Resource Consumption |

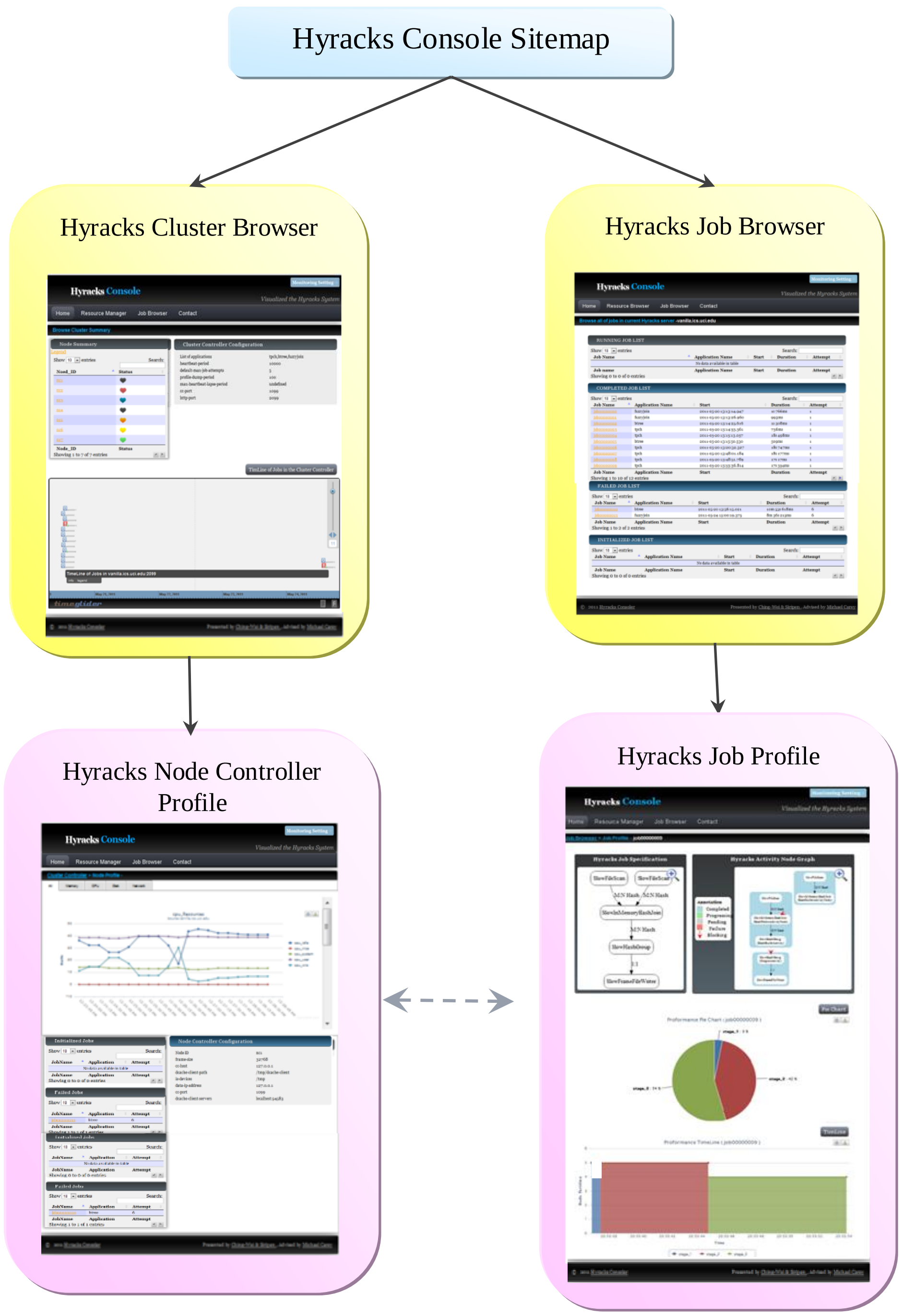
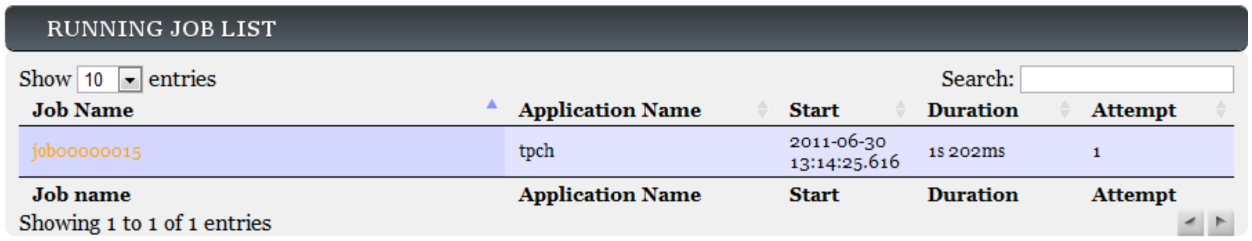


Figure 36 : Hyracks Console Visualization Sitemap

### 6.2.1 Hyracks Job Browser

The Hyracks Job Browser page displays a summary of all of the Hyracks jobs that are deployed in the given Hyracks cluster. There are four tables included in this page to present each category of a job's status, i.e., initialized, running, completed, and failed. An example of the Job Browser page is shown in Figure 35. There we see no jobs in the RUNNING or INITIALIZED JOB LIST tables, twelve jobs in the COMPLETED JOB LIST table, and two jobs in the FAILED JOB LIST table. For each job, the following data is presented: *Job Name, Application Name, Start time, Duration*, and number of job *Attempt* times.

To support real-time monitoring, the Job Browser page embeds a CometD Client JavaScript to subscribe the HCV client to the "/jobs" channel in order to receive messages pushed from the HCS whenever the events related to the Hyracks cluster's job occur. For example, when a new Hyracks job of the "tpch" application is deployed in the "vanilla.ics.uci.edu" Hyracks cluster, the HCS pushes a message through the "/jobs" channel and there by triggers the HCV to update the Job Browser page with low latency. Assuming that the new job is running immediately after being submitted to the cluster, the Hyracks Console user will observe a new entry in the "RUNNING JOB LIST" table as shown in Figure 37. In this example, the new entry is "job00000015" and the start time is "2011-06-30 13:14:25.616". In addition, there is a link at the job's name that allows the user to click and navigate to its Hyracks Job Profile page to see more details related to that job. Let's click on the "job00000015" link and navigate to its profile page as described in the next section.


RUNNING JOB LIST

Show 10 entries    Search:

| Job Name | Application Name | Start | Duration | Attempt |
|---|---|---|---|---|
| job00000015 | tpch | 2011-06-30 13:14:25.616 | 1s 202ms | 1 |
| Job name | Application Name | Start | Duration | Attempt |

Showing 1 to 1 of 1 entries

Figure 37 : Example Result of RUNNING JOB LIST Table after Deploying a Hyracks Job

### 6.2.2 **Hyracks Job Profile**

The Hyracks Job Profile page consists of four visualization modules, (1) *Hyracks Job Specification*, (2) *Hyracks Activity Node Graph*, (3) *Hyracks Job Pie Chart*, and (4) *Hyracks Job Time Chart*. Figure 38 is an overview of the Hyracks Job “job00000015” Profile page.

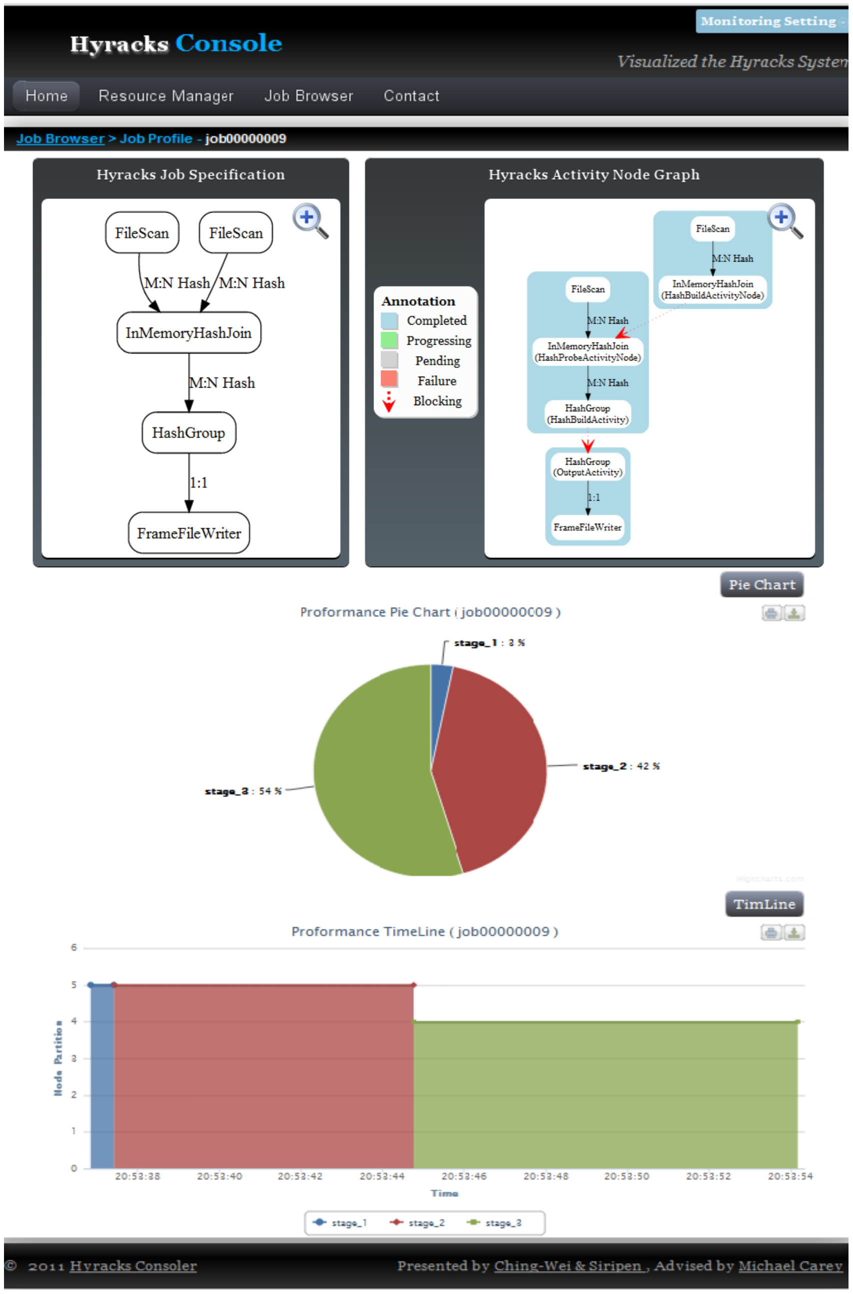


Figure 38: Overview of the Job Profile Page

The top two boxes present the *Hyracks Job Specification* (called *JobSpec*) and the corresponding *Hyracks Activity Node Graph* (called *JobPlan*) of a particular Hyracks job. The Hyracks Job Specification provides users with an overview of how their jobs are structured, while the Hyracks Activity Node Graph explains how each HOD internally expands into a set of corresponding HANs. These two diagrams are offering an easy way for the users to understand the design and execution process of their job. Figure 39 shows these two visual modules. Note that the complete *Job Specification* and *Job Plan* JSONArray along with the DOT script of this example can be found in the appendix section.

When the users are interested in greater details, they can click on the 🔍 button. Then, a model-box is popped up to display the moveable and zoomable diagram (🔍: zoom out and 🔍: zoom in). Each box in the Hyracks Job Specification component represents a HOD and the description inside the box indicates the class of this HOD. Each arrow that connects a pair of HODs represents a CD and the annotation of the arrow indicates the class of this CD. For example, from the Hyracks Job Specification of the Hyracks job “job00000015” shown in Figure 39 (left), there are five HODs assembled to perform this job: two “FileScan”, one “InMemoryHashJoin”, “HashGroup”, and “FrameFileWriter”. Each “FileScan” HOD connects to the “InMemoryHashJoin” HOD by an “M:N Hash Partition” connector. The “InMemoryHashJoin” and “HashGroup” HODs are also connected by an “M:N Hash Partition” connector. Last, the “HashGroup” and “FramFileWriter” HODs are connected by a “1:1” connector.

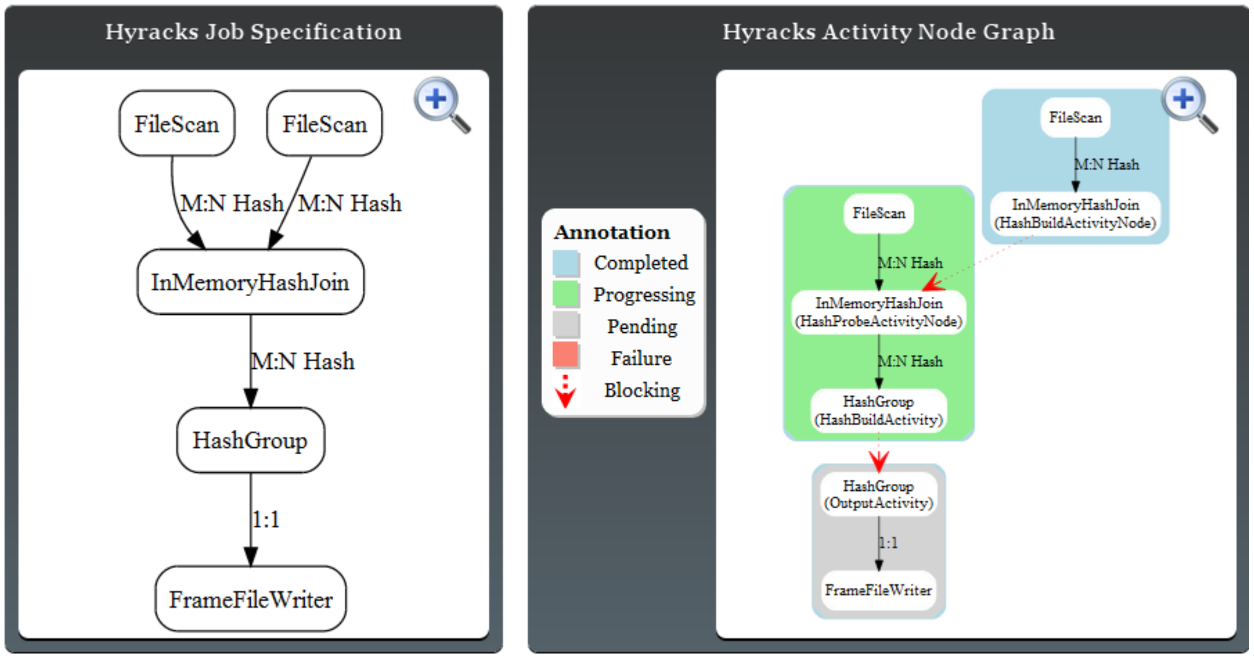


Figure 39 : Hyracks Job Specification (left) and Hyracks Activity Node Graph (right)

When the Hyracks system starts to execute a job, internally, it expands each HOD in the job's specification into a set of HANs which can be presented in the Hyracks Activity Node Graph. The expansion of each HOD indicates all sub-phases involving in each HOD along with the sequence dependencies among phases. In Figure 39 (right), the Hyracks Activity Node graphical view uses red dotted arrows to denote the blocking edges between pairs of HANs, which means that the sender HAN must finish before the receiver HAN can begin. In our example job, the "InMemoryHashJoin" HOD in the Hyracks Job Specification is expanded into "HashBuildAcitvityNode" and "HashProbeAcitvityNode" HANs. The "HashBuildAcitvityNode" HAN must finish before the "HashProbeAcitvityNode" HAN can begin, so there is a blocking edge (red arrow) between these two activities. Each colored box that contains one or several HANs is represented as a stage. The first stage in Figure 39 (right) consists of "FileScan" and "InMemoryHashJoin" HANs which can be executed in a pipelined manner. The different colors denote the runtime status of each stage (i.e., light blue – Completed, light green – Progressing, gray – Pending, red – Failure). Therefore, from the Activity Node Graph, users can recognize easily that the first stage is already completed, the second stage is still progressing, and the third stage is pending.

In addition, the Job Profile page embeds the CometD Client JavaScript to listen to the "/stages/<job-id>" channel. When the current status of any stages of the monitored job is changed, the HCS pushes a message through the "/stages/<job-id>" channel to trigger the HCV to update the color of the stages in the Activity Node Graph. For instance, when our example job finishes the execution, the color of all stages in the Activity Node Graphic will be changed to be the light blue color as shown in Figure 40. This technique represents the current status of each stage by calling JavaScript function to update only the color of each box in the diagram without refreshing the entire page. Thus, the users can stay wherever they are on the page without being perturbed by status color update. For example, if a user opens the model popup page and enlarge the diagram to 200%, then the last stage fails, a user will still be able to observe this change on the last stage as shown in Figure 41.

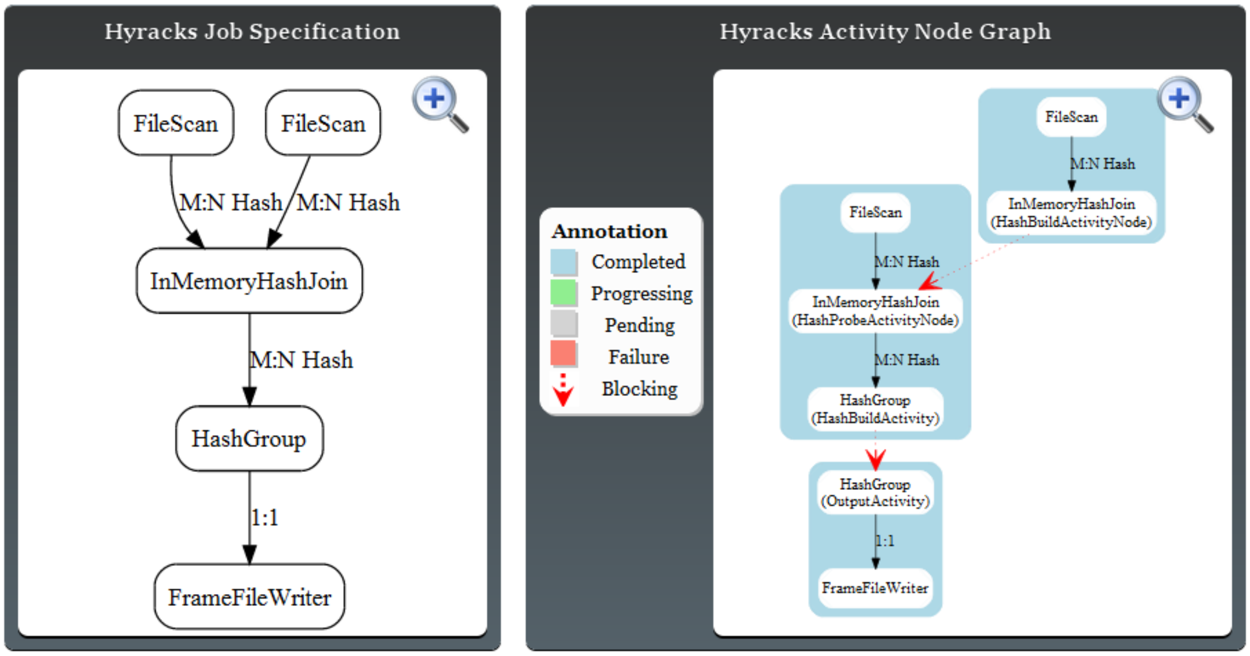


Figure 40 : Hyracks Activity Node Graph of the Completed Hyracks Job

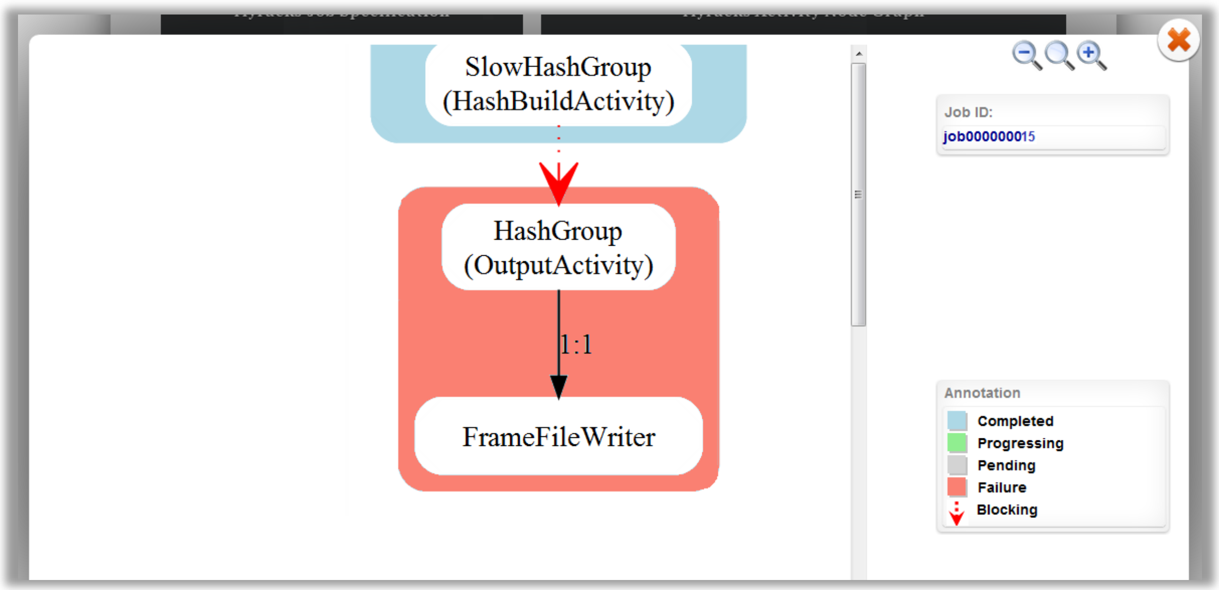
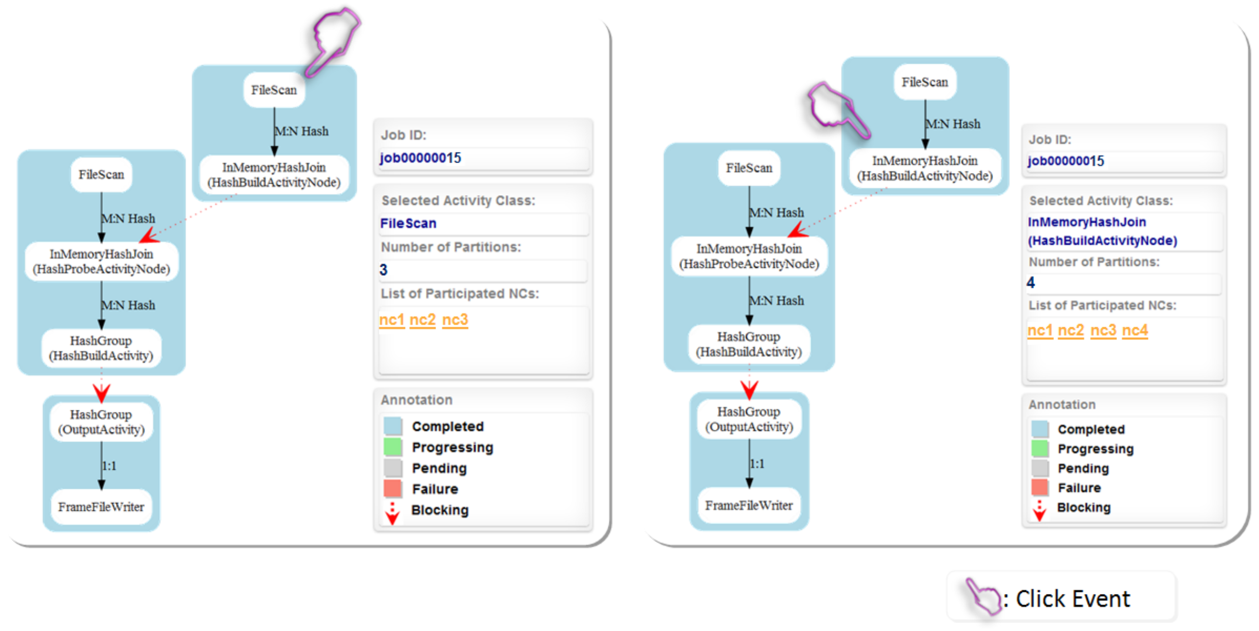


Figure 41: Updating the Hyracks Activity Node Graph on the Model Popup Page

As briefly mentioned earlier, each HAN actually represents a set of parallel tasks scheduled to execute on the machines in the Hyracks cluster. To observe the job's task information, users can interact with the Activity Node Graph. For instance, when a user click on the first "FileScan" HAN, extra HAN's information will be presented on the right panel including the number of partitions and the list of participated NCs, which for example are equal to "3" and "nc1, nc2, nc3" respectively in Figure 41 (left). This means the "FileScan" HAN has three parallel tasks which are scheduled to run on the"nc1", "nc2" and "nc3". Continuing with Figure 41 (right), when the user clicks on the "HashBuildAcitvityNode" HAN in the first stage, the number of partitions is changed to "4" and the list of participated NCs is changed to "nc1, nc2, nc3, nc4", which means the "HashBuildAcitvityNode" HAN has four tasks which is scheduled on the node-id "nc1", "nc2", "nc3" and "nc4".

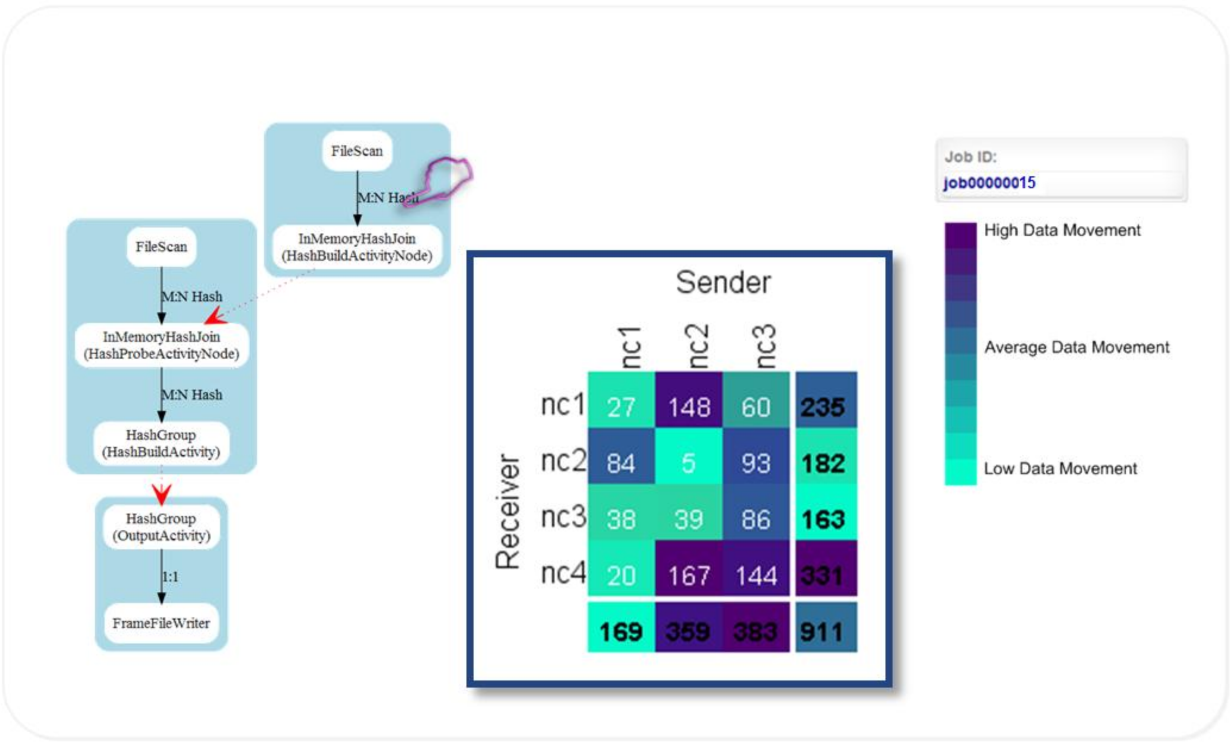

FileScan
M:N Hash
InMemoryHashJoin (HashBuildActivityNode)
FileScan
M:N Hash
InMemoryHashJoin (HashProbeActivityNode)
M:N Hash
HashGroup (HashBuildActivity)
HashGroup (OutputActivity)
1:1
FrameFileWriter
: Click Event

Job ID:
job00000015
Selected Activity Class:
FileScan
Number of Partitions:
3
List of Participated NCs:
nc1 nc2 nc3
Annotation
Completed
Progressing
Pending
Failure
Blocking

Job ID:
job00000015
Selected Activity Class:
InMemoryHashJoin (HashBuildActivityNode)
Number of Partitions:
4
List of Participated NCs:
nc1 nc2 nc3 nc4
Annotation
Completed
Progressing
Pending
Failure
Blocking

Figure 42: HAN Profile of the “FileScan” Activity (left) and
the “HashBuildActivityNode” Activity (right) on the Model Popup Page

In addition, the graphical view of each connector of the Activity Node Graph embeds a communication matrix image presenting the volume of data movement at runtime between the senders and receivers HANs. From the previous “tpch” Hyracks job example, the communication matrix of the “M:N Hash” connector between the “FileScan” HAN and the “HashBuildActivityNode” HAN in the first stage is presented in Figure 43. There are three instances of the “FileScan” HAN and four instances of the “HashBuildActivityNode” HAN. Note that this matrix graph visualization will allow the users to view the data movement on each connector even if the data is shuffled among hundreds or perhaps thousands of nodes.



| | nc1 | nc2 | nc3 | |
|---|---|---|---|---|
| nc1 | 27 | 148 | 60 | 235 |
| nc2 | 84 | 5 | 93 | 182 |
| nc3 | 38 | 39 | 86 | 163 |
| nc4 | 20 | 167 | 144 | 331 |
| | 169 | 359 | 383 | 911 |

Figure 43: Hyracks Activity Node Graph with Data Movement Information on the Popup Page

The Next two modules are the job's performance Pie Chart and Time Chart. The Pie Chart summarizes the percentage of time usage for each job's stages. Each slice of the pie represents a different job stage. If a user moves the mouse over a slice, it will pop out a bubble to indicate the set of HANs that belongs to the selected stage. Figure 44 shows a screen of the Pie Chart when the user has moved his/her mouse over the second stage which consumed 42% of the total execution time, and which consists of three HANs: "HashBuildActivity", "FileScan", and "HashProbeActivityNode". The Time Chart depicts the parallelism of each stage based on the timeline. The X-axis of this chart is time, which begins at the job's start time and ends at the job's finish time. If the job is currently running, the end time is the current time. The Y-axis is the number of partition nodes in each stage. In Figure 45, the number of partitions of the first and the third stages are both four, while the second stage has five partitions. When the user moves the mouse over the second stage in the Time chart, the popup bubble denotes that this stage consists of three HANs ("HashBuildActivity", "FileScan", and "HashProbeAcitvityNode").

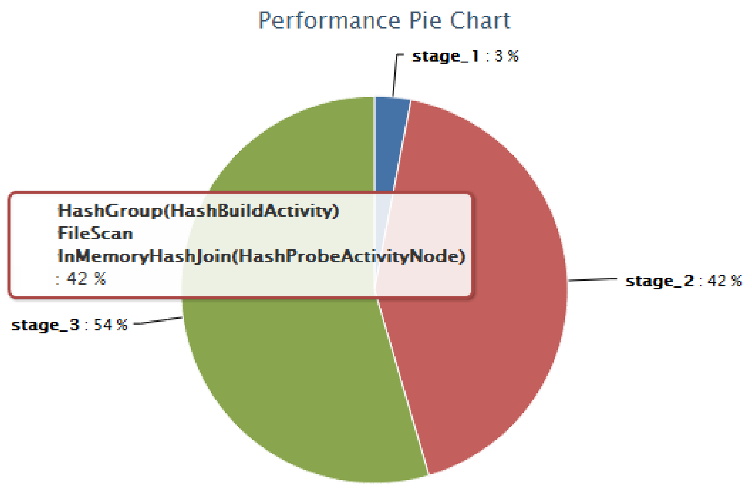


Figure 44 : Performance Pie Chart

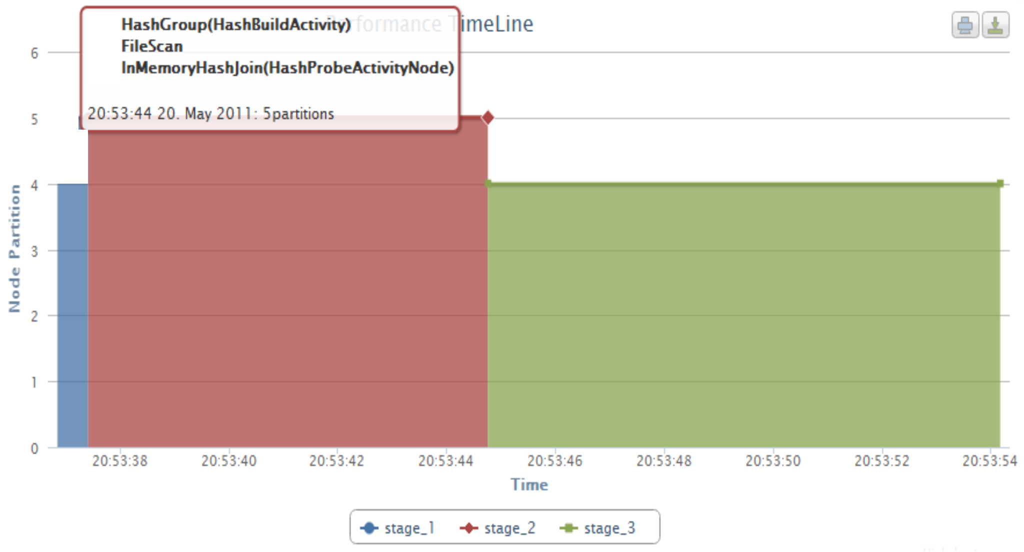
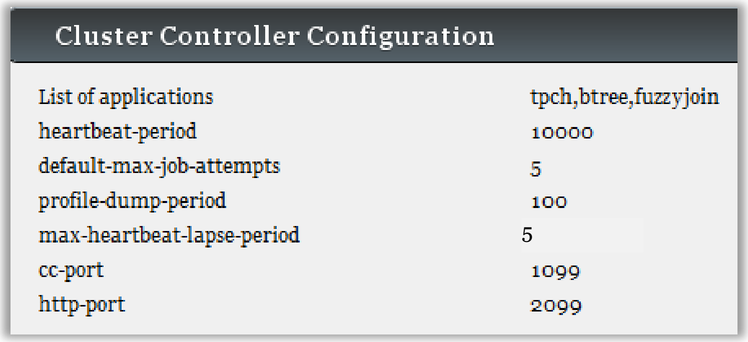


Figure 45 : Performance Time Chart

### 6.2.3 Hyracks Cluster Browser

The Hyracks Cluster Browser page displays an overview of the entire Hyracks clusters. It consists of three visualization modules, (1) *Hyracks Cluster Controller Configuration*, (2) *Registered Hyracks Node Controllers*, and (3) *Hyracks Job Timeglider*. Continuing with our example server, let's click on the Cluster Browser menu to open the Hyracks Cluster Browser page. The first module, the Hyracks Cluster Controller (CC) Configuration, shows configuration information. The CC configuration of our example server, "vanilla.ics.uci.edu", is shown in Figure 46. There are seven variables included in the CC configuration including: *list of applications*, *heartbeat-period*, *default-max-job-attempts*, *profile-dump-period*, *max-heartbeat-lapse-periods*, *cc-port*, and *http-port*. The *list of applications* in the figure shows that there are three applications deployed on the Hyracks system, which are "tpch", "btree", and "fuzzyjoin". The *heartbeat-period* is the duration between two heartbeats of each node controller in milliseconds. The *default-max-job-attempts* number is the maximum numbers of times that the CC will restart a job's execution if any error occurs. The *profile-dump-period* is the time duration between two profile dumps from each node controller, in milliseconds, and the default value is 0 which means it is disabled. In this example, we assigned the profile-dump-period is set to be every 100 milliseconds. The *max-heartbeat-lapse-period* is the maximum number of missed heartbeats before a node is marked as dead. In this example it is unassigned, so it shows "5" as the default. Also, we can see the *cc-port* that was setup for listening for connections from node controllers. Lastly, the *http-port* is a network port number specified for communicating between the HCS and the HCV.



| | |
|---|---|
| List of applications | tpch,btree,fuzzyjoin |
| heartbeat-period | 10000 |
| default-max-job-attempts | 5 |
| profile-dump-period | 100 |
| max-heartbeat-lapse-period | 5 |
| cc-port | 1099 |
| http-port | 2099 |

Figure 46 : CC Configuration for "vanilla.ics.uci.edu" Hyracks Cluster

The second module is the *Hyracks Jobs Timegulider* that represents the history of jobs executions on the monitored Hyracks Cluster. The timeline runs from the first deployed job to the last deployed job. The users can see the status of the jobs from the symbols. An example Hyracks Jobs Timeglider is shown in Figure 47.

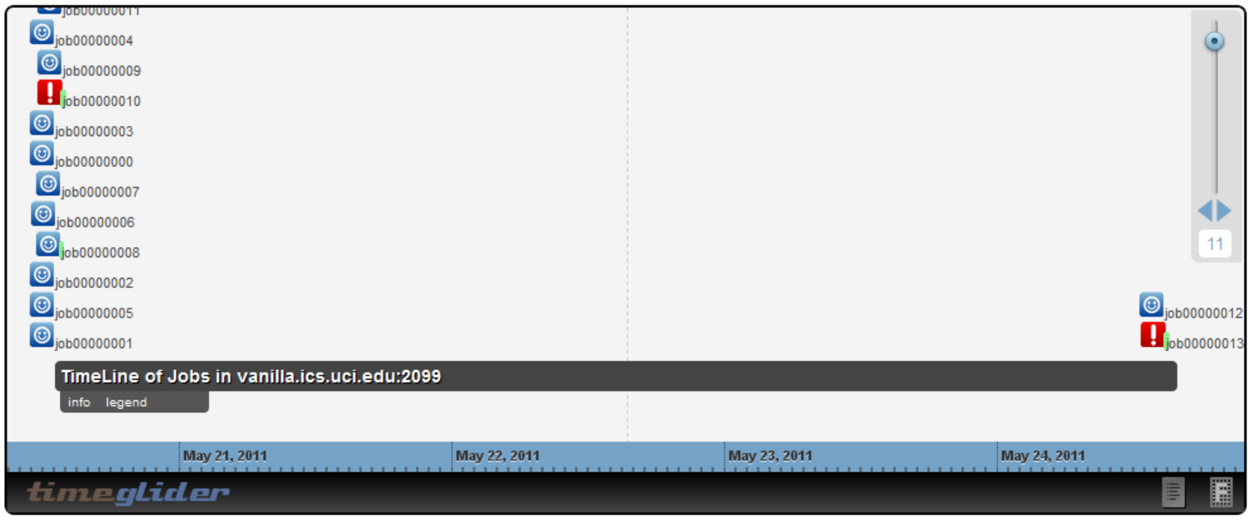


Figure 47 : Hyracks Jobs Timeglider

The legend of the Hyracks Jobs Timegulider explains the symbols and the referred job's status. Each symbol can also be used as a button to toggle the associated jobs from the screen; for instance, after clicking on the failure icon, the screen only displays the failure jobs and hides uninterested jobs as shown in Figure 48.

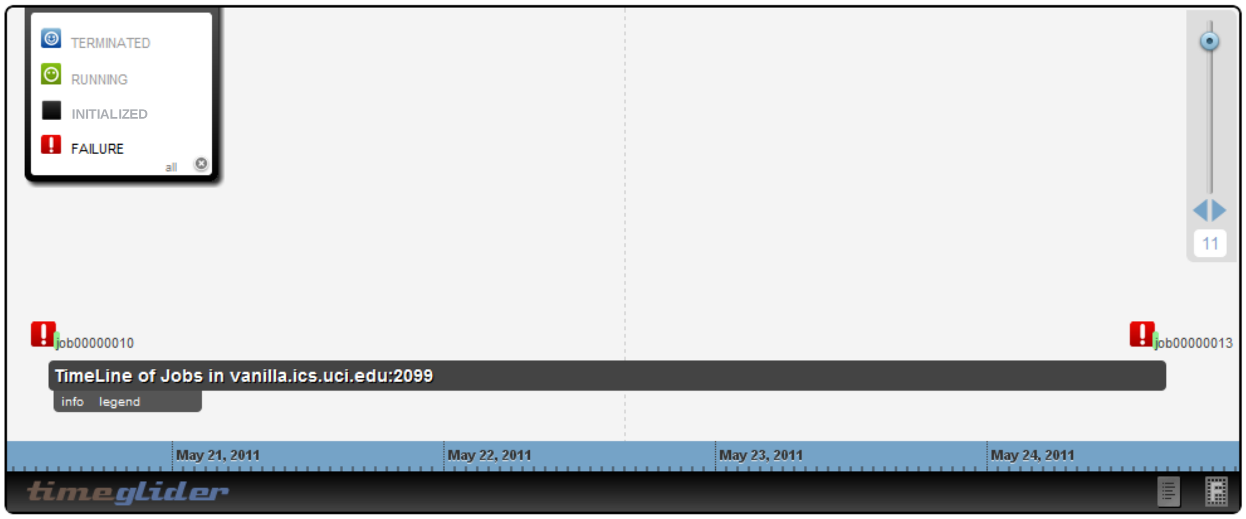
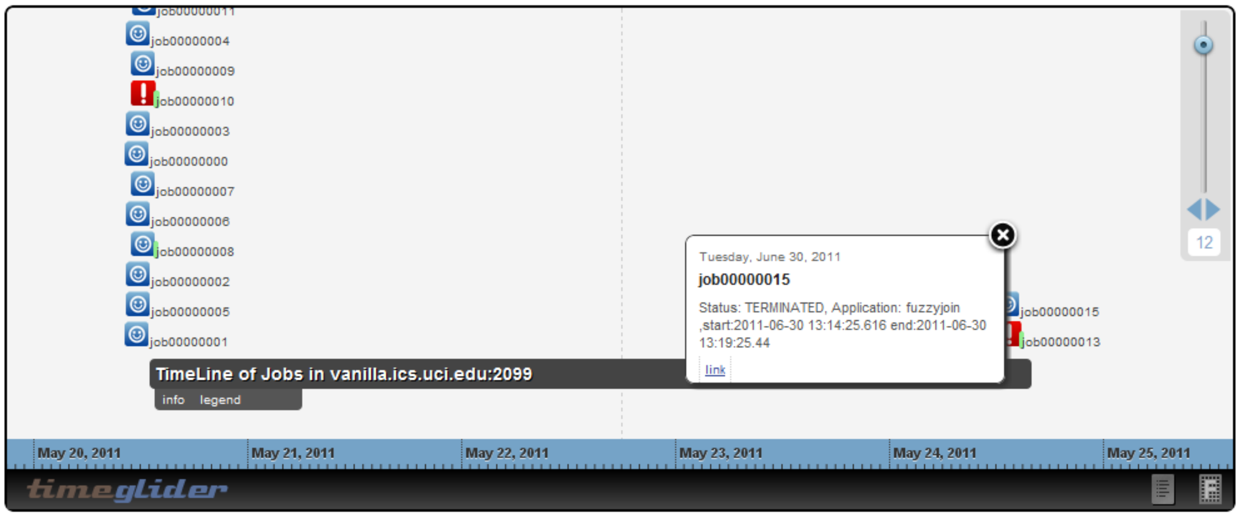


Figure 48 : Only the Failure Jobs in the Hyracks Jobs Timeglider

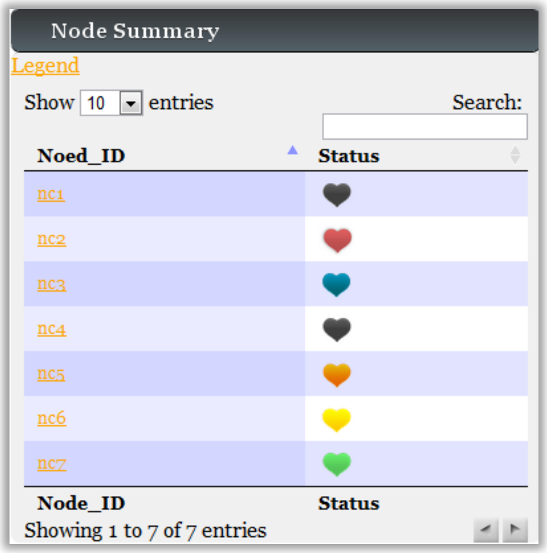
In addition, each job's name on the Timeglider is a clickable link that invokes a detailed description box. In Figure 49, the detail box of the "job00000015" indicates that the job's status is "TERMINATED", its application is "tpch", and its execution started at "2011-06-30 13:14:25.616" and ended at"2011-06-30 13:19:25.44". The hyperlink "link" in the job's detail box navigates to its Hyracks Job Profile page.



Tuesday, June 30, 2011
job00000015
Status: TERMINATED, Application: fuzzyjoin ,start:2011-06-30 13:14:25.616 end:2011-06-30 13:19:25.44
link

Figure 49 : Job's Description in the Hyracks Jobs Timeguilder

The last page, the *Registered NCs* module summarizes all of the NCs that have been registered to the CC. The color of each NC's heart image indicates the CPU utilization of the given NC as per the legend shown in Figure 50. In this example, there are seven nodes in the Hyracks cluster. As the gray hearts indicate in the figure, node "nc1" and "nc4" are dead. Node "nc2" has a red heart, which means that its CPU utilization is close to 100%. By using this module, users can get an overview of the workload of their Hyracks cluster and know which resources are available for their future Hyracks jobs. A user can click on the node-id hyperlink to navigate to the Hyracks Node Controller Profile page. Let's indeed click on the node "nc2" and drill down to its profile page.


Node Summary

Legend

Show 10 entries Search:

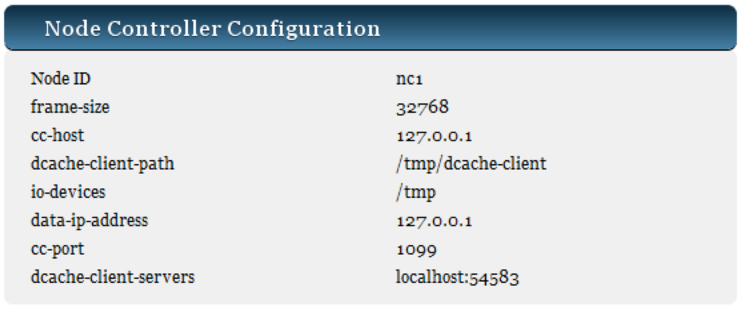
| Noed_ID | Status |
|---|---|
| nc1 | ♥ |
| nc2 | ♥ |
| nc3 | ♥ |
| nc4 | ♥ |
| nc5 | ♥ |
| nc6 | ♥ |
| nc7 | ♥ |
| Node_ID | Status |

Showing 1 to 7 of 7 entries

Figure 50 : Registered NCs (left) and NC's CPU Utilization Legend (right)

### 6.2.4 Hyracks Node Controller Profile

The Hyracks Node Controller Profile page is a child page of the Hyracks Cluster Browser page. The first module in this page is the *Node Controller Configuration*, as shown in Figure 51. All node configuration variables and their values for a given node controller are included in its profile page. The configuration variables include: *node-id, frame-size, cc-host, dcache-client-path, io-devices, data-ip-address, cc-port,* and *dcache-HCVs*.

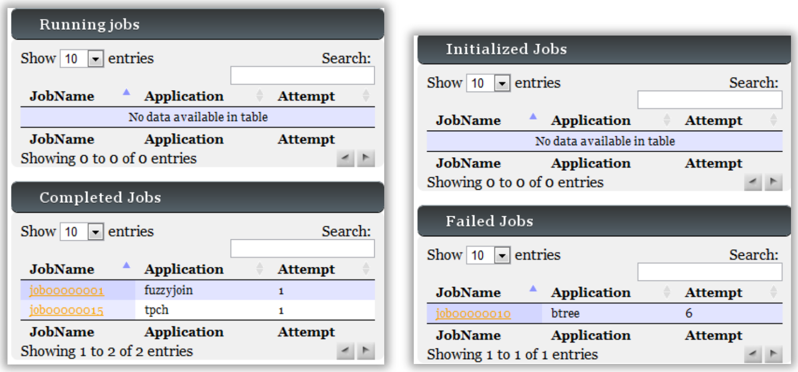


**Node Controller Configuration**

|  |  |
|---|---|
| Node ID | nc1 |
| frame-size | 32768 |
| cc-host | 127.0.0.1 |
| dcache-client-path | /tmp/dcache-client |
| io-devices | /tmp |
| data-ip-address | 127.0.0.1 |
| cc-port | 1099 |
| dcache-client-servers | localhost:54583 |

Figure 51 : Hyracks Node Controller Configuration of the Node-ID "nc2"

The second visualization module is the *Node Controller Jobs Summary*, which presents the list of Hyracks jobs executed on a given node. Each job is classified into one of four categories based on their current status: `INITIALIZED`, `RUNNING`, `TERMINATED`, or `FAILURE`. This information helps users to understand the performance and workload of each node. For instance, Figure 52 shows an example Node Jobs Summary for the node "nc2". The user can see that the job "job00000015" exists on the completed jobs list, for example, meaning that this job was executed on node "nc2" and is already finished.



**Running jobs**

Show 10 entries Search:

| JobName | Application | Attempt |
|---|---|---|
| No data available in table | | |

Showing 0 to 0 of 0 entries

**Completed Jobs**

Show 10 entries Search:

| JobName | Application | Attempt |
|---|---|---|
| job00000001 | fuzzyjoin | 1 |
| job00000015 | tpch | 1 |

Showing 1 to 2 of 2 entries

**Initialized Jobs**

Show 10 entries Search:

| JobName | Application | Attempt |
|---|---|---|
| No data available in table | | |

Showing 0 to 0 of 0 entries

**Failed Jobs**

Show 10 entries Search:

| JobName | Application | Attempt |
|---|---|---|
| job00000010 | btree | 6 |

Showing 1 to 1 of 1 entries

Figure 52 : Node Controller Jobs Summary

The last visualization module is the *Node Controller Resource Monitoring*. Information about the resource consumption of each worker machine is beneficial in order to monitor the cluster's health and detect any job execution failures caused by node failures. There are six types of resources views (e.g., "mem" for memory, "cpu", "disk", "load", "network" and "all") represented in different tabs. By default, users are able to monitor the current resources usage of the given node in the last 450 seconds. Figure 53 is an example of the CPU resource usage for machine “nc2”. When the mouse is moved over specific points in the line graph, a bubble box is popped out to show the exact value at that point. In Figure 53, the pop-out bubble denotes that at “11:00:30 PM” the value of “cpu-user” is “39.4%”.

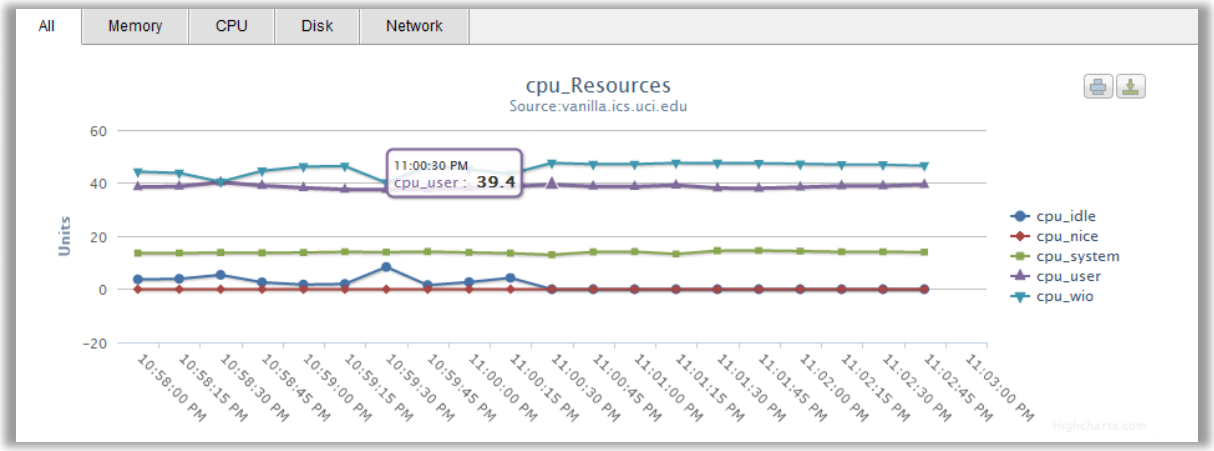


Figure 53 : Hyracks Machine Resources Consumption

## 7 CONCLUSION

The Hyracks partitioned-parallel platform has been developed at UCI to support data-intensive processing and analysis on large-scale distributed clusters. The Hyracks system allows the programmers to execute dataflow applications from a simple interface and conceals the complexity of their internal processes. However, when a failure occurs, in their Hyracks system, or performance is not as expected, it will be beneficial for users if they can understand and analyze the intermediate steps or individual execution processes for their Hyracks jobs. For this purpose, the *Hyracks Console* has been developed to accelerate the development process of large-scale data processing application and provide better insights into the Hyracks platform in real-time. This thesis has described the design and client-side visualization implementation of the Hyracks Console.

The Hyracks Console has been designed to support two main types of users, *Hyracks End Users* and *Hyracks Operator Implementers*, in monitoring the Hyracks system and its jobs. The

Hyracks Console system architecture is conceptually divided into two main components, *Hyracks Console Server (HCS)* and *Hyracks Console Visualization (HCV)*, which are independent of one another.

The first component, the *Hyracks Console Server (HCS)*, runs inside the Hyracks system and is responsible for collecting, storing, and delivering the information related to the Hyracks cluster and its jobs at runtime. To efficiently provide the necessary data, two communication techniques between the client (HCV) and the server (HCS) have been implemented. The first technique is called *client-pull*, where the HCV sends a request to the HCS through the REST API and the HCS then responds with required data in the JSON format. The second technique, *server-push*, is implemented to support real-time monitoring. When significant events happen in the Hyracks system, the HCS initiates the communication and sends a message to the HCV without first reading a request from it. This is used for event notification purposes and triggers page refreshes if/as needed.

The second component, the *Hyracks Console Visualization (HCV)*, receives and converts the plain JSON data into a visual representation. The implementation of the visualization, the data representation mechanism, and chosen data presentation tools have all been discussed in this thesis. The HCV also provides a Graphical User Interface (GUI) that allows users to interact freely with the Hyracks Console. The HCV minimizes the Hyracks users' effort in monitoring the Hyracks system by parsing the JSONArray data into dataflow diagrams, pie charts, timeline charts or tables. We expect the resulting Hyracks Console will facilitate existing users in their understanding of the logical construction and the internal execution of their Hyracks jobs, as well as reducing the time needed for the new Hyracks users to learn about the programming using Hyracks on large clusters.

# REFERENCES

[1] Apache Hadoop. http://hadoop.apache.org.

[2] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation 2004 (OSDI '04)*, pages 137–150, San Francisco, California, USA, December 2004.

[3] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, Lisbon, Portugal, March 2007.

[4] Pig. http://hadoop.apache.org/pig.

[5] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *the 8th Symposium on Operating Systems Design and Implementation 2008 (OSDI '08)*, San Diego, California, USA, December 2008.

[6] Cloudera. http://www.cloudera.com/products-services/tools.

[7] Karmasphere Studio. http://www.karmasphere.com.

[8] J.Boulon A.Konwinski, R.Qi,A. Rabkin E. Yang, and M.Yang. Chukwa: A Large-Scale Monitoring System. In *the 5th International Conference on Computability and Complexity in Anaysis (CCA 2008)*, Hagen, Germany, August 2008.

[9] V. Borkar, M. J. Carey, R. Grover, N. Onose, R. Vernica. Hyracks: A Flexible and Extensible Foundation for Data-Intensive Computing. In *IEEE International Conference on Data Engineering 2011 (ICDE 2011)*, Hanover, Germany, April 2011.

[10] Hyracks. http://code.google.com/p/hyracks.

[11] A. Behm, V. R. Borkar, M. J. Carey, R. Grover, C. Li, N. Onose, R. Vernica, A. Deutsch, Y. Papakonstantinou and V. J. Tsotras. ASTERIX: Towards a Scalable, Semistructured Data Platform for Evolving-world Models. In *Distributed and Parallel Databases, Vol. 29,* pages 185-216, 2011.

[12] TPC-H. http://www.tpch.org/tpch.

[13] R. T. Fielding and R. N. Taylor. Principled Design of the Modern Web Architecture. In *Proceedings of the 2000 International Conference on Software Engineering (ICSE 2000)*, pages 407–416, Limerick, Ireland, June 2000.

[14] JSON (JavaScript Object Notation). http://www.json.org.

[15] "Pull Technology." *Wikipedia, The Free Encyclopedia.* Wikimedia Foundation, Inc. 6 February 2011. Web. 26 June. 2011.

[16] "Push Technology." *Wikipedia, The Free Encyclopedia.* Wikimedia Foundation, Inc. 17 June 2011. Web. 26 June. 2011.

[17] Jetty. http://www.eclipse.org/jetty.

[18] Ganglia Monitoring System. http://ganglia.sourceforge.net

[19] M. L. Massie, B. N. Chun, and D. E. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. In *Parallel Computing, 30(7),* pages 817-840, 2004

[20] RRDtool. http://www.mrtg.org/rrdtool/tut/rrdtutorial.en.html.

[21] CometD. http://cometd.org.

[22] "Comet (programming)." *Wikipedia, The Free Encyclopedia.* Wikimedia Foundation, Inc. 11 June 2011. Web. 26 June. 2011.

[23] Bayeux Protocol. http://svn.cometd.com/trunk/bayeux/bayeux.html.

[24] S. Pongpaichet. Hyracks Console: Monitoring the Hyracks Partitioned-Parallel Runtime Platform. Master's thesis, Department of Computer Science, UC Irvine, August 2011.

[25] JavaServer Pages Technology. http://www.oracle.com/technetwork/java/javaee/jsp/index.html.

[26] Apache Tomcat. http://tomcat.apache.org.

[27] Graphviz: Graph Visualization Software. http://www.graphviz.org.

[28] TimeGlider. http://timeglider.com.

[29] Highcharts JS. http://www.highcharts.com.

[30] DataTables. http://www.datatables.net.

[31] Java Class UUID.http://download.oracle.com/javase/6/docs/api/java/util/UUID.html.

[32] The DOT Language.http://www.graphviz.org/doc/info/lang.html.

[33] E. Gansner, E. Koutsofios, and S. North.Drawing Graphs with dot.http://www.graphviz.org/Documentation/dotguide.pdf.

[34] Scalable Vector Graphics (SVG).http://www.w3.org/Graphics/SVG.

# APPENDIX

In this section, we provide the complete results of the “Job Specification” and “Job Plan” JSONArray of the “tpch” example job (“job_00000015”) presented in Section 6 along with their DOT Scripts used for generating interactive graphical view. In addition, we summarize the URL-Request paths patterns and their usage.

## A. JSONArray : “Job Specification” of the “tpch” Example Job

```
{
result: {
        connectors: [{
                in-operator-port: 0
                in-operator-id: "ODID:73c64eb3-079e-46e2-bc97-19bee4089832"
                connector: {
                        id: "a08f8b12-f793-4348-8986-d671afd2691b"
                        java-class:
                        "edu.uci.ics.hyracks.dataflow.std.connectors.OneToOneConnectorDescrip
                        tor
                        type: "connector"
                        }
                type: "connector-info"
                out-operator-port: 0
                out-operator-id: "ODID:59260ba1-8777-4f58-9710-a3db96b1788d"
                }
                {
                in-operator-port: 0
                in-operator-id: "ODID:6ed0a038-2558-4fed-ae40-f602877bea02"
                -
                connector: {
                        id: "4813880e-89cb-410f-bc60-96b10a8b6d21"
                        java-class:
                        "edu.uci.ics.hyracks.dataflow.std.connectors.MToNHashPartitioningConn
                        ectorDescriptor"
                        type: "connector"
                }
                type: "connector-info"
                out-operator-port: 0
                out-operator-id: "ODID:616ebe36-317f-466e-8677-ea2db022e3dd"
                }
                {
                        in-operator-port: 0
                        in-operator-id: "ODID:c5f3312e-e9c7-49d2-8d01-f3a6a4d0eb37"
                        -
                        connector: {
                                id: "05903646-df96-4165-bdc1-07ed48eb988b"
                                java-class:
                                "edu.uci.ics.hyracks.dataflow.std.connectors.MToNHashPartitio
                                ningConnectorDescriptor"
                                type: "connector"
                        }
                        type: "connector-info"
                        out-operator-port: 1
                        out-operator-id: "ODID:616ebe36-317f-466e-8677-ea2db022e3dd"
                }
```

(Continue on the next page)

```
        {
            in-operator-port: 0
            in-operator-id: "ODID:616ebe36-317f-466e-8677-ea2db022e3dd"
            connector: {
                id: "3fa6003f-060a-4fc6-8da6-2468ff838699"
                java-class:
                "edu.uci.ics.hyracks.dataflow.std.connectors.MToNHash
                PartitioningConnectorDescriptor"
                type: "connector"
            }
            type: "connector-info"
            out-operator-port: 0
            out-operator-id: "ODID:73c64eb3-079e-46e2-bc97-19bee4089832"
        }
    ]
    operators: [
        {
            id: "6ed0a038-2558-4fed-ae40-f602877bea02"
            in-arity: 0
            java-class:
            "edu.uci.ics.hyracks.dataflow.std.file.FileScanOperatorDescriptor"
            out-arity: 1
            type: "operator"
        }
        {
            id: "73c64eb3-079e-46e2-bc97-19bee4089832"
            in-arity: 1
            java-class:
            "edu.uci.ics.hyracks.dataflow.std.group.HashGroupOperatorDescriptor
            "
            out-arity: 1
            type: "operator"
        }
        {
            id: "59260ba1-8777-4f58-9710-a3db96b1788d"
            in-arity: 1
            java-class:
            "edu.uci.ics.hyracks.dataflow.std.file.FrameFileWriterOperatorDescr
            iptor"
            out-arity: 0
            type: "operator"
        }
        {
            id: "616ebe36-317f-466e-8677-ea2db022e3dd"
            in-arity: 2
            java-class:
            "edu.uci.ics.hyracks.dataflow.std.join.InMemoryHashJoinOperatorDesc
            riptor"
            out-arity: 1
            type: "operator"
        }
        {
            id: "c5f3312e-e9c7-49d2-8d01-f3a6a4d0eb37"
            in-arity: 0
            java-class:
            "edu.uci.ics.hyracks.dataflow.std.file.FileScanOperatorDescriptor"
            out-arity: 1
            type: "operator"
        }
    ]
    type: "job"
    }
}
```

## B. JSONArray: “Job Plan” of the “tpch” Example Job

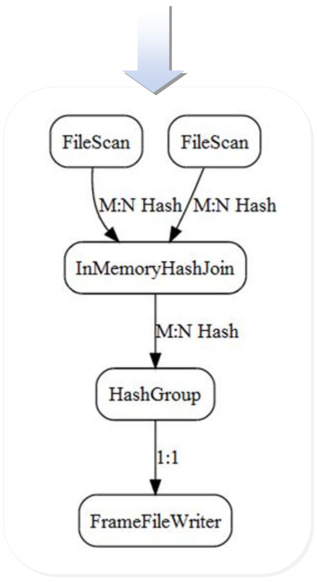
```
{result: {
    id: "f196e2ba-a236-41c3-a2ee-3881157b0fd9"
    flags: "[PROFILE_RUNTIME]"
    activities: [
             {
             id: "ANID:8494ccfe-f5c2-4e25-9d71-2dad14daf01e"
             owner-id: "ODID:6ed0a038-2558-4fed-ae40-f602877bea02"
             java-class: "edu.uci.ics.hyracks.dataflow.std.file.FileScanOperatorDescriptor"
             outputs: [
                  {
                  connector-id: "CDID:4813880e-89cb-410f-bc60-96b10a8b6d21"
                  type: "activity-output"
                  connector-java-class:
                  "edu.uci.ics.hyracks.dataflow.std.connectors.MToNHashPartitioningConnectorDescriptor"
                  output-port: 0
                  }
                  ]
                  type: "activity"
             }
             {
             id: "ANID:637582c0-3d09-41e9-95ce-4d6038f1d562"
             owner-id: "ODID:73c64eb3-079e-46e2-bc97-19bee4089832"
             inputs: [
             {
                  input-port: 0
                  connector: {
                           id: "3fa6003f-060a-4fc6-8da6-2468ff838699"
                           java-class:
                           "edu.uci.ics.hyracks.dataflow.std.connectors.MToNHashPartitioningConnectorDescriptor"
                           type: "connector"
                      }
                           connector-id: "CDID:3fa6003f-060a-4fc6-8da6-2468ff838699"
                           type: "activity-input"
                           connector-java-class:
                           "edu.uci.ics.hyracks.dataflow.std.connectors.MToNHashPartitioningConnectorDescriptor"
                      }
                      ]
                  java-class: "edu.uci.ics.hyracks.dataflow.std.group.HashGroupOperatorDescriptor$HashBuildActivity"
                  type: "activity"
             }
             {
             id: "ANID:061df810-5121-4c6c-86e7-fad70146541a"
             owner-id: "ODID:616ebe36-317f-466e-8677-ea2db022e3dd"
             inputs: [
             {
                  input-port: 0
                           connector: {
                           id: "4813880e-89cb-410f-bc60-96b10a8b6d21"
                           java-class:
                           "edu.uci.ics.hyracks.dataflow.std.connectors.MToNHashPartitioningConnectorDescriptor"
                           type: "connector"
                      }
                           connector-id: "CDID:4813880e-89cb-410f-bc60-96b10a8b6d21"
                           type: "activity-input"
                           connector-java-class:
                           "edu.uci.ics.hyracks.dataflow.std.connectors.MToNHashPartitioningConnectorDescriptor"
             }
             ]
                  java-class:
             "edu.uci.ics.hyracks.dataflow.std.join.InMemoryHashJoinOperatorDescriptor$HashBuildActivityNode"
                  type: "activity"
         }
         {
                  id: "ANID:41bcb50f-0891-4e68-8e57-4511e402dbee"
                  owner-id: "ODID:c5f3312e-e9c7-49d2-8d01-f3a6a4d0eb37"
                  java-class: "edu.uci.ics.hyracks.dataflow.std.file.FileScanOperatorDescriptor"
                  outputs: [
                           -
                  {
                           connector-id: "CDID:05903646-df96-4165-bdc1-07ed48eb988b"
                           type: "activity-output"
                           connector-java-class:
                           "edu.uci.ics.hyracks.dataflow.std.connectors.MToNHashPartitioningConnectorDescriptor"
                           output-port: 0
                  }]
                  type: "activity"
         }
```

(Continue on the next page)

```
{
        id: "ANID:ab78cb09-a1b6-440f-aa92-9ed64064cbf9"
        owner-id: "ODID:616ebe36-317f-466e-8677-ea2db022e3dd"
        inputs: [{
                nput-port: 0
                connector: {
                        id: "05903646-df96-4165-bdc1-07ed48eb988b"
                        java-class:
                        "edu.uci.ics.hyracks.dataflow.std.connectors.MToNHashPartitioningConnectorDescriptor"
                        type: "connector"
                }
                        connector-id: "CDID:05903646-df96-4165-bdc1-07ed48eb988b"
                        type: "activity-input"
                        connector-java-class:
                        "edu.uci.ics.hyracks.dataflow.std.connectors.MToNHashPartitioningConnectorDescriptor"
        }]
        java-class:
        "edu.uci.ics.hyracks.dataflow.std.join.InMemoryHashJoinOperatorDescriptor$HashProbeActivityNode"
        outputs: [
        {
                        connector-id: "CDID:3fa6003f-060a-4fc6-8da6-2468ff838699"
                        type: "activity-output"
                        connector-java-class:
                        "edu.uci.ics.hyracks.dataflow.std.connectors.MToNHashPartitioningConnectorDescriptor"
                        output-port: 0
        }
        ]
        type: "activity"
        depends-on: [
                "ANID:061df810-5121-4c6c-86e7-fad70146541a"
        ]}
{
        id: "ANID:c21d2a96-8d19-41fc-a1e3-4430adeed1dd"
        owner-id: "ODID:59260ba1-8777-4f58-9710-a3db96b1788d"
        inputs: [
        {
                        input-port: 0
                        connector: {
                            id: "a08f8b12-f793-4348-8986-d671afd2691b"
                            java-class:
                            "edu.uci.ics.hyracks.dataflow.std.connectors.OneToOneConnectorDescriptor"
                            type: "connector"
                                }
                            connector-id: "CDID:a08f8b12-f793-4348-8986-d671afd2691b"
                            type: "activity-input"
                            connector-java-class:
                            "edu.uci.ics.hyracks.dataflow.std.connectors.OneToOneConnectorDescriptor"
                        }
                            ]
                            java-class:
                            "edu.uci.ics.hyracks.dataflow.std.file.FrameFileWriterOperatorDescriptor"
                            type: "activity"
                    }
                    {
                            id: "ANID:a0da67cd-e36d-433c-803f-cc70f1e86e96"
                            owner-id: "ODID:73c64eb3-079e-46e2-bc97-19bee4089832"
                            java-class:
                            "edu.uci.ics.hyracks.dataflow.std.group.HashGroupOperatorDescriptor$OutputActivity"
                            outputs: [
                            {
                                connector-id: "CDID:a08f8b12-f793-4348-8986-d671afd2691b"
                                type: "activity-output"
                                connector-java-class:
                                "edu.uci.ics.hyracks.dataflow.std.connectors.OneToOneConnectorDescriptor"
                                output-port: 0
                }
                            ]
                            type: "activity"
                            -
                            depends-on: [
                                    "ANID:637582c0-3d09-41e9-95ce-4d6038f1d562"
                            ]
                    }
            ]
            type: "plan"
    }
    }
```

## C. DOT Language Script: “Job Specification” of the “tpch” Example Job

```
digraph hyracks_job {
    size = "20,20"; rankdir = "TB"; // initialize the size of the node box

    //initialize the node of the HAN graph
    "fbb74415-c6ea-404a-9788-5affab4e4fd5" [label="FrameFileWriter"];
    "8abbfff4-8b39-4469-9551-7ccf3ce27cef" [label="FileScan"];
    "afe0ba86-efc9-4249-8f6b-2b1e5229c000" [label="FileScan”];
    "7ad96b95-2c02-47e6-833e-3d454263be5b" [label="HashGroup"];
    "da2c467a-ed31-4166-b9bb-0a9bcfcae0e5" [label="InMemoryHashJoin"];

    //give the order of the every two connected nodes.
    "8abbfff4-8b39-4469-9551-7ccf3ce27cef"->"da2c467a-ed31-4166-b9bb-0a9bcfcae0e5" [label="M:N Hash"];
    "afe0ba86-efc9-4249-8f6b-2b1e5229c000"->"da2c467a-ed31-4166-b9bb-0a9bcfcae0e5" [label="M:N Hash"];
    "da2c467a-ed31-4166-b9bb-0a9bcfcae0e5"->"7ad96b95-2c02-47e6-833e-3d454263be5b" [label="M:N Hash"];
    "7ad96b95-2c02-47e6-833e-3d454263be5b"->"fbb74415-c6ea-404a-9788-5affab4e4fd5" [label="1:1"];
}
```

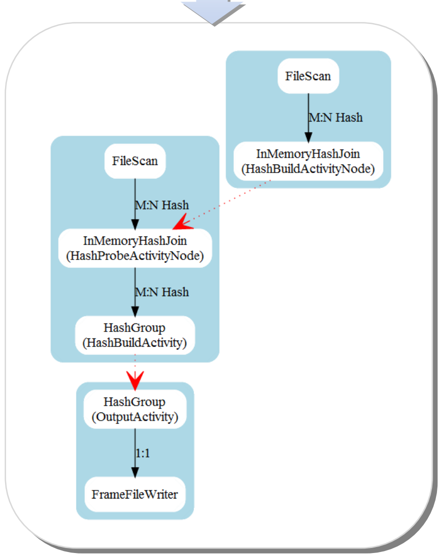


## D. DOT Language Script: “Job Plan” of the “tpch” Example Job

```
digraph hyracks_job {
        //initialize the size of the node box
        size = "20,20";rankdir = "TB";

        //initialize the node of the HAN graph(JobPlan)
        "ANID:22c232fb-b785-4839-aa19-de4c1952c115"[id="ANID:22c232fb-b785-4839-aa19-de4c1952c115"
        label="HashGroup\n(OutputActivity)"
        URL="javascript:showNC('HashGroup\n(OutputActivity)',2,'[nc1,nc2]')"];

        "ANID:f37cdbb0-2947-4a8b-a5b7-aa888b9db4ad"[id="ANID:f37cdbb0-2947-4a8b-a5b7-aa888b9db4ad"
        label="InMemoryHashJoin\n(HashBuildActivityNode)"
        URL="javascript:showNC('InMemoryHashJoin\n(HashBuildActivityNode)',1,'[nc1]')"];

        "ANID:7f72f554-89a6-4525-a63a-68e84af65284" [id="ANID:7f72f554-89a6-4525-a63a-68e84af65284"
        label="FileScan" URL="javascript:showNC('FileScan',2,'[nc1,nc2]')"];

        "ANID:e9f4aebd-b04b-4eb6-8aef-f8699e557e1d" [id="ANID:e9f4aebd-b04b-4eb6-8aef-f8699e557e1d"
        label="FileScan"URL="javascript:showNC('FileScan',2,'[nc1,nc2]')"];

        "ANID:e95df9e1-a8db-46e3-8609-d84a2cad587a" [id="ANID:e95df9e1-a8db-46e3-8609-d84a2cad587a"
        label="InMemoryHashJoin\n(HashProbeActivityNode)"
        URL="javascript:showNC('InMemoryHashJoin\n(HashProbeActivityNode)',1,'[nc1]')"];

        "ANID:2df02bc7-d6ef-4d61-aa5f-2ae45f1ed6e2" [id="ANID:2df02bc7-d6ef-4d61-aa5f-2ae45f1ed6e2"
        label="HashGroup\n(HashBuildActivity)"
        URL="javascript:showNC('HashGroup\n(HashBuildActivity)',2,'[nc1,nc2]')"];

        "ANID:218446b2-ebea-415e-b279-f565db1fabd0" [id="ANID:218446b2-ebea-415e-b279-f565db1fabd0"
        label="FrameFileWriter" URL="javascript:showNC('FrameFileWriter',2,'[nc1,nc2]')"];

        //Group a set of HANs into different stage s and also give the order of nodes in each stage.
        subgraph cluster_0 {
                id="sid_38b8589bd0584564951d53c8eae0873c"  style="filled,rounded";
                node [style=filled, fillcolor=white]; color=lightblue;
                [
                "ANID:e9f4aebd-b04b-4eb6-8aef-f8699e557e1d" -> "ANID:f37cdbb0-2947-4a8b-a5b7-
        aa888b9db4ad" [label="M:N Hash"
                URL="javascript:showConnector('http://vanilla.ics.uci.edu:2099/profile/a965fe5a-ba0b-4a01-
        a13d-1b4dcc57adc1/0/0f641b5b-1624-4e59-8ff4-ad1e2bdd0e9a', 'M:N Hash')",];
                ]
        }
```

(Continue on the next page)

```
        subgraph cluster_1 {
                id="sid_0689036c64124484a024c535e29d993a"  style="filled,rounded";
                node [style=filled, fillcolor=white]; color=lightblue;
                [
                "ANID:22c232fb-b785-4839-aa19-de4c1952c115" -> "ANID:218446b2-ebea-415e-b279-
        f565db1fabd0" [label="1:1"
                URL="javascript:showConnector('http://vanilla.ics.uci.edu:2099/profile/a965fe5a-ba0b-
        4a01-a13d-1b4dcc57adc1/0/7f0c5be7-cb19-            4bc2-90e5-30b436272826', '1:1')",];
                ]
        }
        subgraph cluster_2 {
                id="sid_37ceb645c042480b8a30c6b425f396ea"  style="filled,rounded";
                node [style=filled, fillcolor=white]; color=lightblue;
                [
                "ANID:7f72f554-89a6-4525-a63a-68e84af65284" -> "ANID:e95df9e1-a8db-46e3-8609-
        d84a2cad587a" [label="M:N Hash"
                URL="javascript:showConnector('http://vanilla.ics.uci.edu:2099/profile/a965fe5a-ba0b-
        4a01-a13d-1b4dcc57adc1/0/af0852fc-4123-4b61-9723-a0d645361230', 'M:N Hash')",];,
                "ANID:e95df9e1-a8db-46e3-8609-d84a2cad587a" ->"ANID:2df02bc7-d6ef-4d61-aa5f-
        2ae45f1ed6e2" [label="M:N Hash"
                URL="javascript:showConnector('http://vanilla.ics.uci.edu:2099/profile/a965fe5a-ba0b-
        4a01-a13d-1b4dcc57adc1/0/50917f45-f3c2-47ba-8e15-a4bec7556be8', 'M:N Hash')",];
                ]
        }

        //use the blocking edges to connect the subgraphs together
        "ANID:2df02bc7-d6ef-4d61-aa5f-2ae45f1ed6e2" -> "ANID:22c232fb-b785-4839-aa19-de4c1952c115"
        [style=dotted ,arrowhead=vee, arrowsize=2,color=red];
        "ANID:f37cdbb0-2947-4a8b-a5b7-aa888b9db4ad"-> "ANID:e95df9e1-a8db-46e3-8609-
        d84a2cad587a"[style=dotted,arrowhead=vee,arrowsize=2,color=red];
}
```


FileScan
M:N Hash
InMemoryHashJoin (HashBuildActivityNode)
FileScan
M:N Hash
InMemoryHashJoin (HashProbeActivityNode)
M:N Hash
HashGroup (HashBuildActivity)
HashGroup (OutputActivity)
1:1
FrameFileWriter

## E. Summary of the URL-Requests

| URL-Request Path | Usage | Console Page/Feature |
|---|---|---|
| `/state/jobs` | Get list of Hyracks jobs that have been submitted to the Hyracks cluster | Job Browser/Job List |
| `/state/jobs/<job-id>/spec` | Get for a particular Hyracks job's specification | Job Profile/HOD graph |
| `/state/jobs/<job-id>/plan` | Get for a particular Hyracks job's plan ( activity node graph) | Job Profile/HAN graph |
| `/state/jobs/<job-id>/<attempts>/stage` | Get progress of a Hyracks job in stage-by-stage basis | Job Profile/HAN graph |
| `/state/jobs/<job-id>/<attempts>/profile` | Get profile counter of a particular Hyracks job | Job Profile/HAN graph |
| `/profile/<job-id>/<attempts>/<connector-id>/(number)` | Get dataflow matrix image of a particular Hyracks connector in a specific Hyracks job. Optional "number" at the end of the URL is used when users want to show the actual number data frames moving at the Hyracks connector. | Job Profile/HAN graph |
| `/console/cluster` | Get Cluster Controller Configuration | Cluster Browser/CC configuration |
| `/console/nodes/<node-id>/config` | Get Node Controller Configuration | NC Profile/NC configuration |
| `/console/nodes` | Get list of all nodes associated with the interested Cluster Controller | Cluster Browser/Registered NC |
| `/console/nodes/<node-id>/jobs` | Get list of Hyracks jobs on a given node | NC Profile/NC Job Summary |
| `/console/nodes/<node-id>/resources/<type>/(<step>)/(<start-time>)/(<end-time>)` | Get physical resource consumption at a given Node Controller | NC Profile/Machine Resource Consumption |

Note: every URL-Request path starts with "`http://<cc-ip-address>:<http-port>`"