UNIVERSITY OF CALIFORNIA,
IRVINE

Hyracks Console:
Monitoring the Hyracks Partitioned-Parallel Runtime Platform

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Computer Science

by

Siripen Pongpaichet

Thesis Committee:
Professor Michael J Carey, Chair
Associate Professor Chen Li
Professor Ramesh Jain

2011

DEDICATION

To

my family and friends

TABLE OF CONTENTS

Page

LIST OF FIGURES

Page

ACKNOWLEDGMENTS

The Hyracks Console could not have been built without the assistance of many individuals and teams. We are especially grateful to our advisor Michael Carey for his helpful suggestions, encouragement, and for the kindness that he offered us throughout this process. We also would like to thank the members of the Hyracks and ASTERIX group at University of California, Irvine, for their feedback and support in overcoming obstacles: Vinayak Borkar, Rares Vernica, Raman Grover, Alexander Behm, Nicola Onose and Yingyi Bu. Without them, this project would not be what it is today.

Last, but not least, I would like to offer my warmest gratitude to my friend, Ching-Wei Huang, for always being great company throughout this project, and for her splendid work on the terrific interface design that has often left everyone speechless.

ABSTRACT OF THE THESIS

Hyracks Console:
Monitoring the Hyracks Partitioned-Parallel Runtime Platform

By

Siripen Pongpaichet

Master of Science in Computer Science

University of California, Irvine, 2011

Professor Michael J Carey Irvine, Chair

Use of high-level programming frameworks for data-intensive computation on large-scale distributed clusters is prevalent in the parallel programming community, as the complexity of an underling cluster can be concealed. However, debugging and optimizing applications in this framework often leads to much difficulty and time loss on the part of the programmer. This paper presents the *Hyracks Console*, which is designed to assist Hyracks users in monitoring and understanding sets of parallel processes running on the *Hyracks* partitioned-parallel runtime platform. The core challenges involved in creating the Hyracks Console were: identify which system data will be most useful to users, gathering this data in real time, and ultimately, presenting the data in a visual and scalable representation for the user.

1 INTRODUCTION

In the past five years, high-level programming frameworks for large-scale parallel programs, such as *Hadoop* [1] from Apache, *MapReduce* [2] from Google, *Dryad* [3] from Microsoft, *Pig* [4] from Yahoo!, and *Hive* [5] from Facebook, have gained a lot of interest and attention, both from academic researchers and real world businesses. The ability of these frameworks to shield programmers from the complexity of an underlying distributed cluster and to provide a simple programming interface allows programmers to create highly scalable applications and run them in parallel on large-scale distributed clusters. Despite the success of the MapReduce model, the model adds complexity to the computation design process of the problem being solved; users need to translate their problems into jobs involving only *Map* and *Reduce* primitives. In response, an alternative infrastructure called "*Hyracks*" [6], has been proposed by researchers at the University of California, Irvine. Hyracks is an open-source, *partitioned-parallel platform* built to support data-intensive computations on large *shared-nothing* clusters.

Debugging and monitoring remains a great challenge for these high-level programming platforms. To diagnose the cause of a program failure, users have to understand the internal structure of their distributed job and the mappings that exist between their fragments of sequential source code and their jobs distributed execution. Given that program failures are common, this means that the complexity of the distributed system inevitably becomes visible to users. Several systems, such as *Pig* and *DryadLINQ* [7] provide a local debugging mode which is useful for finding some types of bugs, but not for finding failures caused by large scale data. *Cloudera* [8] and *Karmasphere* [9] provide a rich set of tools to simplify the development experience for Hadoop users. In addition, *Chukwa*, a log collection framework [10], can be used to monitor Hadoop clusters (including some visualization tools). *Hyracks Users* face similar difficulties in understanding and analyzing the execution processes of their *Hyracks jobs*. To accelerate the development process and provide better insight into the Hyracks platform, a monitoring system is essential.

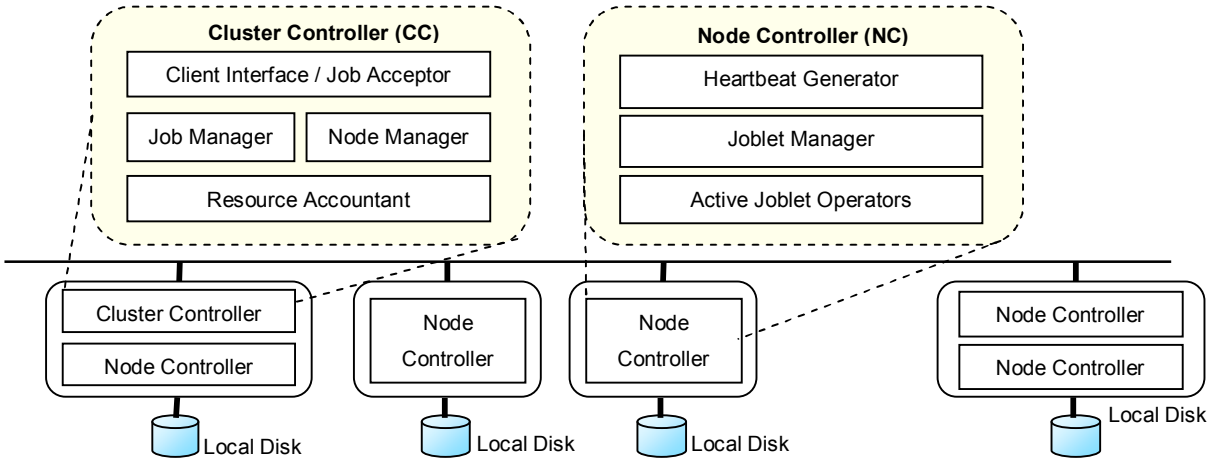
In this thesis, the design and implementation of a new Hyracks Console is presented. The ultimate goal of this component of Hyracks is to reduce the time and effort required for understanding, implementing and optimizing *Hyracks jobs*. Programmers can use the Hyracks console both to monitor their clusters' health and to track the details of their Hyracks jobs' execution. This monitoring system has some important characteristics, including: *identifying real-time job and system status* and *supporting large-scale clusters*. All source codes for this project will be available in a furthering release of Hyracks [11]. While the Hyracks Monitoring Console is designed to monitor the Hyracks system, its approach is quite universal and could be applied to other popular distributed platform such as MapReduce or Dryad. In Section 2, a quick overview of the Hyracks platform is provided. Section 3 delves deeper into the motivations and goals of the Hyracks Console. Section 4 presents the Hyracks Console architecture and design. Section 5 explains the implementation of the Hyracks Console system. Finally, Section 6 summarizes the thesis and discusses a few other features that can be added in the future.

2 HYRACKS OVERVIEW

Hyracks is an *open-source, partitioned-parallel* runtime platform written in the Java programming language. Hyracks is designed to support data-intensive computations on large *shared-nothing* distributed clusters. Computations on large collections of data, which are distributed across multiple machines, are efficiently divided into smaller computations that execute on each partition of the data separately (in parallel). Hyracks is in the same general space as the Hadoop and Dryad infrastructures for data-parallel problems. Unlike the Hadoop framework, which offers the MapReduce computational paradigm based on user-provided functions, Hyracks provides an out-of-the-box set of common operators and connectors for creating general data-oriented computation tasks. Hyracks also allows users to build their own operator and/or connector types. In addition, Hyracks includes a Hadoop compatibility layer that enables users to run existing Hadoop MapReduce jobs without changing their existing code. In this section, we provide a quick overview of the Hyracks system architecture and describe the steps involved in Hyracks job execution using a simple example of data query. A more extensive explanation of Hyracks, along with a first evaluation of its performance, can be found in [6].

2.1 Hyracks Architecture

Hyracks is designed to work on a shared-nothing cluster of commodity machines with local CPUs, memories, and disks. As shown in Figure 1, Hyracks clusters are composed of two types of nodes: *Hyracks Cluster Controllers (CCs)* and *Hyracks Node Controllers (NCs)*. Every Hyracks cluster is managed by one-and-only-one Cluster Controller process. In addition, any machine in the Hyracks cluster can run one or more Node Controller processes. To setup a Hyracks cluster, one machine is designated as a master node and runs the Hyracks Cluster Controller. This master machine must be accessible from all worker machines in the cluster as well as from the client machines that will be used to submit jobs to Hyracks. Worker machines that want to participate in the Hyracks cluster must run a Hyracks Node Controller.



Source: ASTERIX: towards a scalable, semistructured data platform for evolving-world models [12]

Figure 1: Hyracks Cluster Architecture

When clients submit job execution requests by way of the client interface, the Cluster Controller is responsible for accepting those requests, setting up their evaluation plans, and then scheduling the job-tasks to run on selected machines in the cluster. When any failure occurs, the Cluster Controller is also responsible for re-planning and re-executing some or all of the job-tasks. In addition, the Cluster Controller is used to monitor the overall cluster health and to keep track of the resource loads at the worker machines.

Each worker machine in a Hyracks cluster has to run a Node Controller process which is used to accept task execution requests from the Cluster Controller and which reports its health

(e.g., liveness and resource usage levels) periodically via a heartbeat mechanism. Hyracks uses a push-mechanism to interact between the Cluster Controller and the Node Controllers: the Cluster Controller pushes task execution requests to selected Node Controllers, and each Node Controller pushes a notification (either complete or fail) message back to the Cluster Controller when a computation at the worker machine is finished. In addition, to handle system failures that might occur accidentally during the evaluation of a job, Hyracks provides a built-in mechanism for fault detection and recovery through the use of heartbeats.

When an NC is started, it gets three required arguments: `node-id` (which has to be unique), `cc-host` (CC's IP address) and `data-ip-address` (NC's IP address). By default, the Cluster Controller listens to communication data from the NCs through port 1099, but users can setup a different port by passing optional parameters (`port`) when users start the Cluster Controller. Besides the communication port between the CC and the NCs, there is another port called `http-port`; this is a port for a Jetty web server with the HTTP protocol that runs inside the Hyracks cluster. This port number is very important for the Hyracks Monitoring Console, as all information displayed on the console is retrieved through this server port.

2.2 Hyracks Jobs

A Hyracks job is started as a directed acyclic graph (DAG) representing a dataflow that consists of a set of Hyracks Operator Descriptors (HODs) and Connector Descriptors (CDs). The HOD nodes in the graph represent the partitioned computation operators of the job, and the CD edges between the HOD nodes represent the data flow paths from one operator to another. CDs also provide Hyracks with data-distribution information for shuffling data from the set of input operators to the set of output operators. Internally, each HOD consists of one or more sub-activities or phases called Hyracks Activity Nodes (HANs). Hyracks expands the HOD graph into a more detailed HAN graph in order to divide the job into several stages and plan the order in which the stage is going to be executed. The HOD graph is also called the Hyracks Job Specification, and the corresponding HAN graph is also called the Hyracks Job Plan. At runtime, Hyracks decides on the amount of parallelism necessary for each stage and then creates a set of that many identical Hyracks Operator Nodes (HONs). These HONs are each clones of a HAN

and are responsible for performing their processing on individual partitions of the data coming to them. Figure 2 summarizes these concepts.

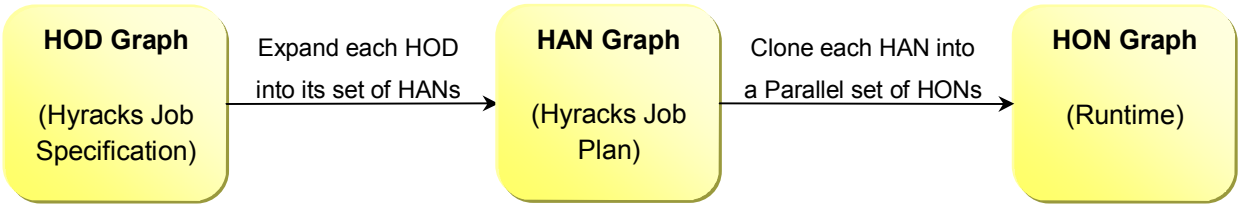


Figure 2: Graph Representations for a Hyracks Job

As mentioned earlier, Hyracks provides a standard set of HOD and CD libraries that are commonly used in building data-oriented programs so that end users and query compliers can simply use those built-in libraries to assemble their jobs. Important operators such as "file scan", "file writer", "map", "sort", "join", "group", and "aggregate" operators 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 allows end users to implement additional sets of HODs and CDs specialized to their needs. For example, ASTERIX, a data-intensive storage and computing platform based on the Hyracks platform [12], has implemented a "limit operator" and an "N:1 Merge connector" for its special requirements.

Throughout this section, we will use a simple query as a Hyracks job example to describe the internal processes of job execution from the beginning to the final evaluation step. This Hyracks job is based on two files containing CUSTOMER and ORDERS data from the TPC-H dataset [13]. This job aims to calculate the number of orders placed by customers belonging to various market segments, and the results are sorted by their market segments. The example Hyracks job is equivalent to the SQL query in the top of Figure 3. In addition, Figure 3 shows the TPC-H schema of the two tables related to the query, namely the CUSTOMER and ORDERS tables.

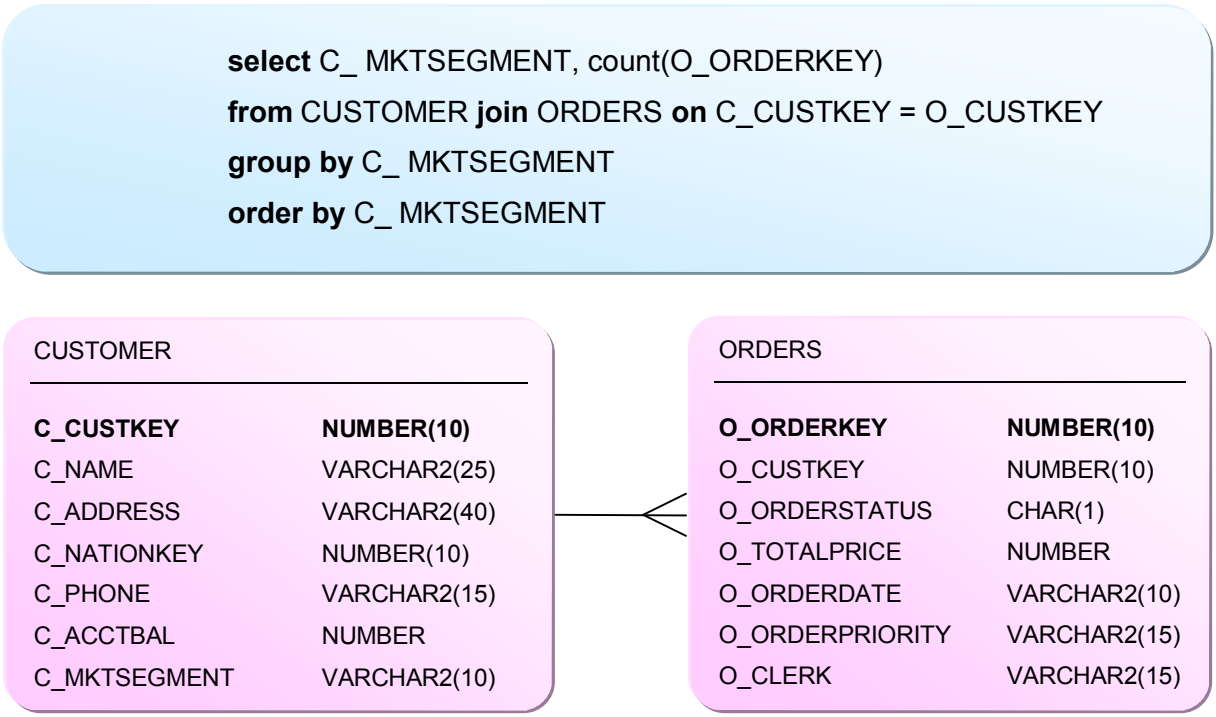


Figure 3: SQL Query of the Example Hyracks Job and TPC-H Schema

2.3 Hyracks Job Specification

The simple Hyracks job's specification can be constructed as shown in Figure 4. Each node shows a HOD's name and argument(s), and each edge between two HODs indicates the data distribution logic to be used for routing partitioned data from the set of senders to their intended receivers. Additional metadata is provided in the File Scan operators to specify where the files are located. Since data collections in Hyracks are partitioned and stored locally on different nodes in the cluster, in order to access files using a file scan, the runtime tasks for the scans have to be scheduled and executed on the machine(s) where the files are available.

For this example, Figure 4 shows that the CUSTOMER data is partitioned into two files (customer-part1.tbl and customer-part2.tbl) that are stored locally on nodes NC1 and NC2, respectively. The ORDERS data is partitioned into three files (orders-part1.tbl, orders-part2.tbl, and orders-part3.tbl), and each partition is *two-way replicated*. Thus, the first file can be accessed on either node NC2 or NC3, the second file on either node NC3 or NC4, and the last file on either node NC4 or NC5. This replication information gives Hyracks multiple choices for scheduling the file scan HOD's tasks on ORDERS data.

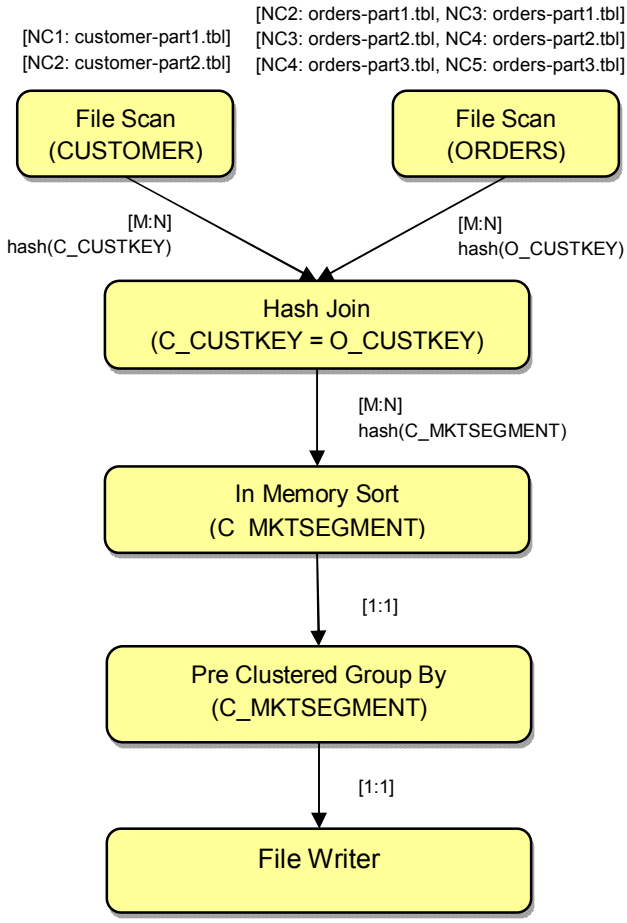


Figure 4: Example Hyracks Operator Descriptor Graph (Hyracks Job Specification)

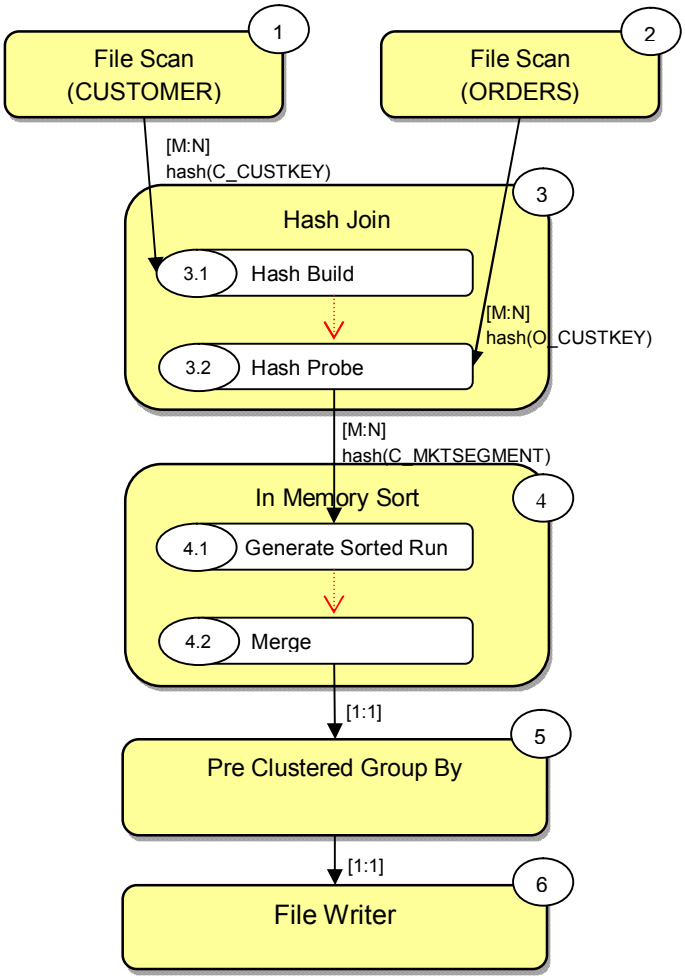


Figure 5: Example Hyracks Activity Node Graph (Hyracks Job Plan)

Each File Scan HOD in Figure 4 is used to read data from a data source file (CUSTOMER or ORDERS) and sends its stream of data to the Hash Join HOD using an *M:N hash-based partitioning* connector. This connector distributes each datum produced by each of M senders, using a provided hash function, to one of N receivers. In this case, the number of senders (M) is the number of partitioned files in each data source, so M=2 for CUSTOMER data and M=3 for ORDERS data. To compute the join in a partitioned manner, we need to ensure that CUSTOMER and ORDERS instances that match on the join condition will be routed to the same

join task. Therefore, hash partitioning of the CUSTOMER and ORDERS instances is done on C_CUSTKEY and O_CUSTKEY, respectively.

The Hash Join HOD in Figure 4 receives these two streams of data (one with CUSTOMER instances and one with ORDERS instances) and produces a stream of CUSTOMER-ORDERS pairs that satisfy the joining condition (C_CUSTKEY = O_CUSTKEY). The result is re-distributed to an In-Memory Sort HOD using an *M:N hash based partitioning* connector on the C_MKTSEGMENT field, in order to ensure that all CUSTOMER-ORDERS pairs that agree on the that field will be routed to the same sort task. Then, each sort task locally sorts the input stream into descending order on the C_MKTSEGMENT field and passes it to a Pre-Clustered Group By HOD using a *1:1* connector. This 1:1 connector means that the number of sender tasks will equal the number of receiver tasks and that data is routed pair-wise without repartitioning. The Pre-Clustered Group By HOD can be used in this example since the input data of this grouping HOD is clustered on the grouping field. Aggregation can then be done in a partitioned manner, locally at each task, by invoking a COUNT aggregation function to count the number of O_ORDERKEY occurrences within each group. Finally, the result from the Pre-Clustered Group By HOD is sent to a File Writer HOD using, again, a 1:1 connector.

2.4 Hyracks Job Execution

At the beginning of a job's execution, Hyracks will expand each HOD in the *Hyracks Job Specification* into a set of HANs. For example, Figure 5 shows the Hyracks activity graph for the Hyracks job specification in Figure 4. The expansion of each HOD indicates all sub-phases involved in each operator along with the sequencing dependency information among phases. In our example, the Hash Join HOD is expanded into two HANs; Hash Build and Hash Probe activities. The first activity builds a hash table on one input stream (usually the one with the smaller amount of data). To produce the result stream of join pairs, the other input stream will then be used in the second activity to probe the hash table (the result from Hash Build activity) with its value on the hash field. Note that the build phase of the Hash Join HOD needs to finish before the probe phase can begin. This sequencing constraint is indicated with the dotted arrow (blocking edge) from the Hash Build HAN to Hash Probe HAN in Figure 5.

Continuing with Figure 5, the In-Memory Sort HOD is similarly expanded into two HANs: the Generate Sorted Runs and Merge activities. The former activity is used for generating a set of sorted runs, and the latter activity is used to produce the sorted result by merging all sorted runs generated by the first activity. The merge phase is not able to start until the first phase completes; therefore, there is a blocking edge between those two activities. The rest of the HODs in Figure 4 consist of a single activity each, so there is only one HAN in Figure 5 for each of Figure 4's HODs.

A Hyracks Activity Node graph provides Hyracks with insight into the sequencing dependencies of each part of a Hyracks job. Hyracks uses this information to facilitate execution planning and coordination, which is why a HAN graph is also called *Hyracks Job Plan*. All HANs that connect to one another with non-blocking edges are grouped together to form a job stage. For each stage, the set of activities in it can be co-scheduled (e.g., to run in a pipelining manner). Stages are planned and executed by Hyracks in the order in which they become ready to run. A stage is ready to execute when all of its dependent stages have been executed completely and successfully. Hyracks decides on the degree of parallelism and location of HAN instances based on the job's resource requirements and node affinities. For example, all File Scan HAN tasks/instances have to be assigned to worker machines where the files of partitioned data are located.

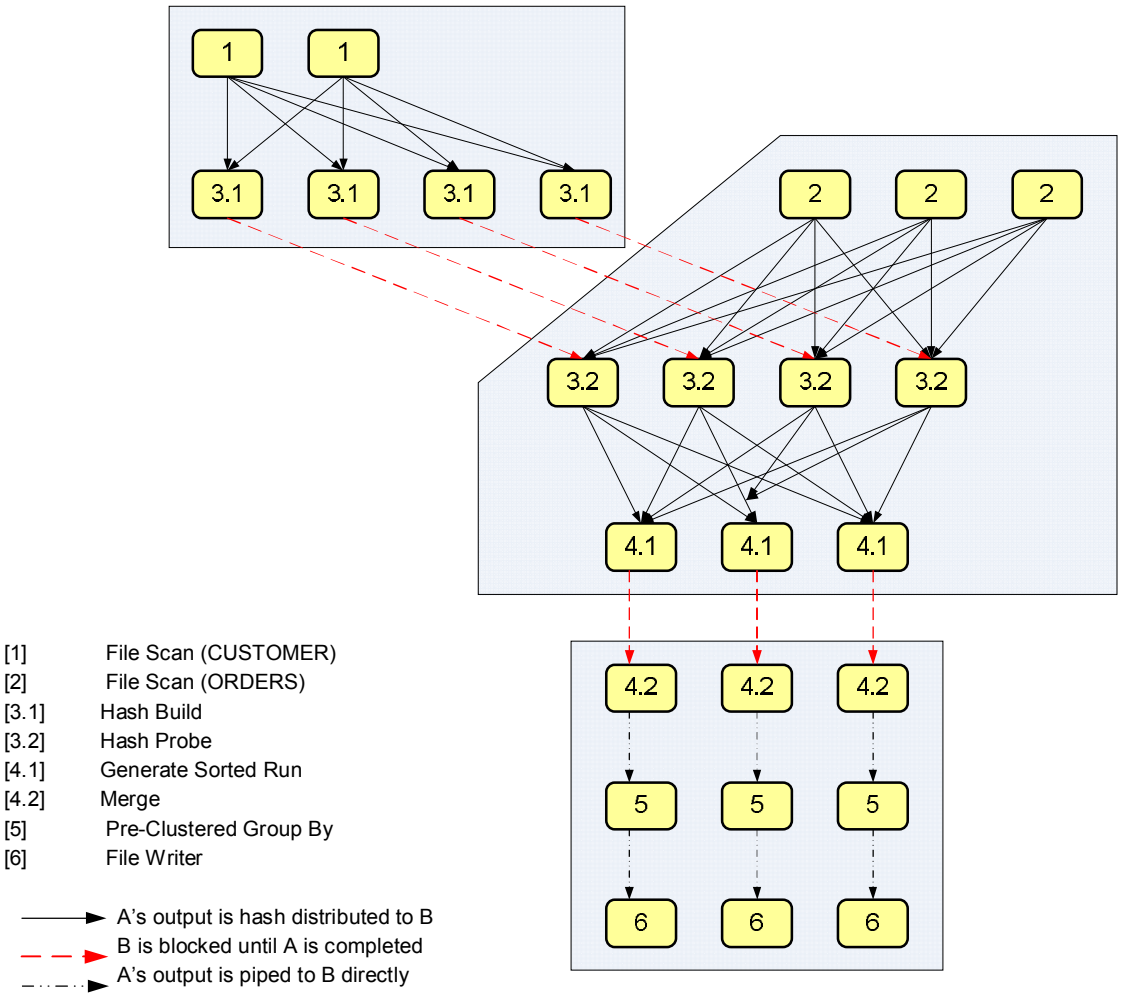


Figure 6: Example Hyracks Operator Node Graph at runtime

When each stage is scheduled, *Hyracks Operator Nodes (HONs)* are created at runtime to be responsible for the actual computation at each worker machine. In fact, HONs are simply clones of HANs. Figure 6 shows an example of a runtime HON graph resulting from stage planning. The gray boxes in the figure indicate groups of operator nodes that belong to the same stage. In reality, Hyracks will plan the degrees of parallelism and the locations of the HONs for each stage just prior to the stage's execution. At runtime, the three stages in our example will be executed stage by stage based on their readiness to run. The first stage consists of two activities: the File Scan activity on CUSTOMER data and the Hash Build activity as part of the hash join operator. In this example, we assume that CUSTOMER data is partitioned into two files and that Hyracks decides to use four nodes to compute the hash join. Hyracks begins to run the two File

Scan HONs along with the four Hash Build HONs to produce hash tables of CUSTOMER data in stage 1.

After the first stage is run successfully, the next stage, which probes the hash tables using ORDERS data and then performs run generation for the sort operator, is planned and executed. Here, we assume that ORDERS data is divided into three separate files with two-way replication and that Hyracks decides to use three nodes for sorting. Thus, in the second stage, there are three clones of the File Scan activity, four clones of the Hash Probe activity, and three clones of the Generate Sorted Run activity, as shown in Figure 6. After Hyracks completes the second stage, it begins to plan the last stage. This stage is composed of a set of Merge, Group By, and File Writer HONs. The (local) Merge HONs receive the sorted runs from the previous stage, and they send their sorted output streams to their local Group By HONs. These HONs aggregate their inputs using a count aggregation function and produce the final results, which are written to partitions of the output file by the File Writer HONs. The execution of this example job is then successfully complete and terminated. Note that the 1:1 connectors between the activities in the last stage cause the degree of parallelism of the Group By and File Writer activities to be the same as the Sort activity.

In the current Hyracks version (0.1.3), fault recovery from a node failure is very simple. A job has a number indicating the maximum number of attempts for the job to be executed before it is aborted by the Hyracks system. Any jobs that are impacted by a node failure will be restarted from the beginning. (The Hyracks developers are now developing a more sophisticated recovery technique that allows the system to restart a job at the granularity of each stage instead of the entire job.)

3 HYRACKS CONSOLE GOALS

The Hyracks platform has two main types of users

a) ***Hyracks End Users*** are users who create Hyracks jobs by assembling Hyracks operators and connectors together, subsequently executing their jobs on the Hyracks clusters. These users are likely to face the following problems:

- When a given job encounters problems during execution, users are given a limited indication of what caused the an error; it can be hard for them to determine at which stage the job's error occurred. Consequently, it is often hard and time consuming for end users to debug their Hyracks jobs.
- Users face difficulty in getting an overall picture of which nodes are participating in the Hyracks cluster and how the various tasks in their job have been mapped to these nodes by Hyracks.

b) ***Hyracks Operator Implementers*** are generally responsible for creating new Hyracks operators and connectors, either in the system's core library or in a user's additional library. These developers may have these following difficulties:
   - Similar to the problems experienced by the end users, operator implementers have to spend a significant amount of time implementing and debugging Hyracks jobs in order to test their new operators and/or connectors.
   - As mentioned earlier, each HOD in a Hyracks job consists of one or more HAN(s). At runtime, each HAN is cloned into a parallel set of HONs. Operator implementers often need to examine the internal processes of a Hyracks job's execution to ensure that their operators and/or connectors are performing correctly and that input/output data are nicely partitioned and distributed across nodes.

The Hyracks Console is built to assist both kinds of Hyracks users to deal with the above mentioned difficulties. The ultimate goal of the Hyracks console is to reduce the time and effort required in monitoring and debugging Hyracks jobs. The different types of users have different functional requirements and demands. Fortunately, these needs can be reduced to two major domains, which are the Hyracks job's execution processes and the Hyracks cluster's performance. The Hyracks Console is implemented to collect information needed for analyzing the Hyracks system and displays that information on the visualization interface. In addition, the console provides the following features.

- **Real-Time Status**: the Hyracks Console enables users to immediately detect changes in the states of the Hyracks system. It allows users to observe the progress of their job's execution in real-time.

- **Scalability**: Hyracks is designed to run on a large-scale cluster. A production cluster may consist of thousands of nodes and may execute thousands of complicated Hyracks jobs at the same time. The console is designed to efficiently collect, organize and present a large amount of monitored data.

4 HYRACKS CONSOLE ARCHITECTURE

To achieve the stated system requirements and goals, we designed this system based on a Representational *State Transfer (REST)* web-based architecture [14]. It consists of two major components, which are *web clients* and *web servers*. The REST architecture is used to implement the Hyracks console because of its scalability of component interaction, its independence in terms of deployment of client and server components, and the simplicity of its interfaces. A critical concept of REST is the existence of resources, each of which is referred to with a global identifier (e.g., a URI in the HTTP protocol). In our system, these resources include the static and dynamic states of both *Hyracks Jobs* (e.g., each *job's specification, plan, status, current state,* etc.) and *Hyracks Clusters* (e.g., the *Cluster Controller* and the *Node Controller* configurations). To manipulate these resources appropriately, clients and servers are required to communicate via a standardized interface such as HTTP and exchange representations of those resources. The resources at the server are conceptually separated from the presentations returned to the client.

For most of the data-interchanges between clients and servers, we decided to use the *JavaScript Object Notation (JSON)* format [15] rather than the *Extensible Markup Language (XML)* format because of several advantages of JSON. JSON is a lightweight alternative to XML, and JSON is smaller, faster and thus better for data interchanges. JSON is a text format that is completely language independent and compatible with many programming languages (e.g., Java, JavaScript, C, C++, C#, Python and many others). JSON objects are built up from two structures: name/value pairs and ordered lists of values. These are universal data structures found in most modern programming languages, which make the JSON format quite compatible with many other programming languages.

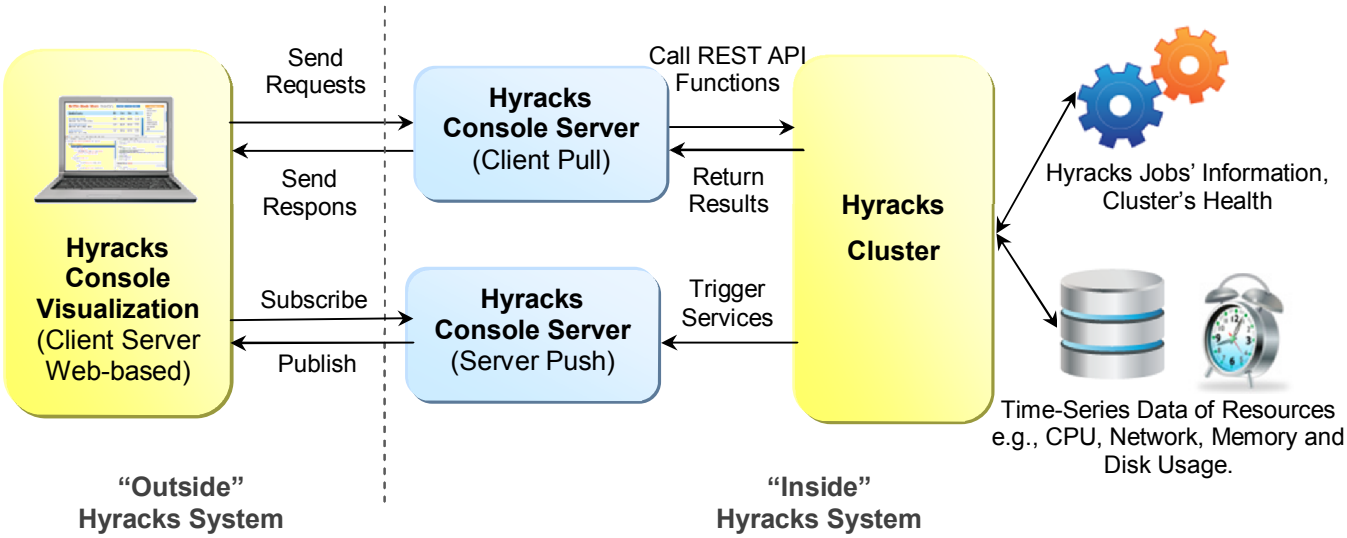


Figure 7: Hyracks Console System Architecture

4.1 Hyracks Console Visualization (HCV)

As shown in Figure 7, the Hyracks console architecture is conceptually divided into two parts: "inside" and "outside" the Hyracks system. On the outside, we have implemented the *Hyracks Console Visualization (HCV)* component to present the current state of the Hyracks system in real-time. Hyracks users can access HCV with any web browser (e.g., Chrome, Firefox, or IE). This component is totally separate from the Hyracks cluster. Any HCV can connect to any cluster, and multiple HCVs can connect to the same cluster provided that the user's machine can reach the web server inside the Hyracks system.

The HCV component provides a visualization of the current state of the Hyracks clusters and its Hyracks jobs. It has the ability to represent information such that users can easily and quickly understand and interpret it. Humans are a *visually-oriented* species, meaning that they prefer images or diagrams over plain text. Therefore, it is vital to have appropriate visualization methods available for representing different types of data, such as tables, charts, graphs and data flow diagrams, in order to explain and describe the complex information in the simplest possible way. This component should not only provide the high-level abstraction of an entire Hyracks cluster and Hyracks jobs, but should also allow users to drill-down into the granular details of each node in a given cluster and each stage of a given job. To this end, many visual representation tools have been used such as GraphViz [16, 20], TimeGlider [17] and DataTables

[18]. There were several challenges involved in building this component, including achieving *scalability* (the visualization should be intelligible at a large scale), as well as the ability of the system to function *dynamically* (users should be able to see graphics changing in real-time). Further explanation of the visualization issues, together with the set of animation tools implemented in the Hyracks console system, can be found in [19].

4.2 Hyracks Console Server (HCS)

The second component of the Hyracks console is the *Hyracks Console Server (HCS)* "on the inside" of the Hyracks system. The HCS is responsible for collecting and delivering monitoring information of the Hyracks system to the first component, *Hyracks Console Visualization (HCV)*. In fact, there are two web servers implemented in the Hyracks system to support two different techniques for communication between clients and servers.

4.2.1 Client Pull Mechanism

The first communication technique is called a *client-pull* [21], in which a client initiates a request for transmission of information from the server. The server receives the request and calls a REST API function to generate the result, mostly in the JSON format. Finally, the result is delivered to the client via HTTP protocol. Our goal is to provide the exact data needed when a user requests them; no more, no less. To connect to the web server, clients need to know the server port number, which has to be setup through the `http-port` argument when users start the Hyracks Cluster Controller process. To implement the Hyracks web server, we decided to use Jetty, an open-source project providing an HTTP server, together with the `javax.servlet` container [23], since it is embeddable, extensible, and enterprise scalable.

To come up with an appropriate set of useful REST API functions, we first had to understand the behavior of the Hyracks platform in fine detail, and estimate what kinds of information are both significant and necessary. There have been various existing console systems used to support other parallel platforms, which we used as guidelines, for example, *Cloudera* and *Karmasphere Studio* for the Hadoop platform, *Nova* workflow manager [24] for *Pig* and *Daphne* for *DryadLINQ* programs [25]. Combining the common functionalities of the existing console systems with the survey of Hyracks users' requirements, the set of necessary information was eventually finalized. Our Hyracks console design is also flexible enough to allow users to

define their own REST API functions for generating and passing the resulting data through the web server inside the Hyracks system.

Monitoring information of the Hyracks cluster is gathered from two main sources. First, data related to the state of the Hyracks jobs and the health of the Hyracks cluster are collected by the *CC (Hyracks Cluster Controller)* and stored temporarily in the memory of the master node. This kind of data is deleted whenever the CC is restarted. The second type of data is time-series data stored on the local disk of the master node. Examples of this data are the CPU load, network in/out, memory usage and disk consumption of all machines participating in the Hyracks cluster. To collect this time-series data, we installed and used *Ganglia*, a scalable distributed monitoring system for high performance computing systems [26, 27]. The Ganglia system uses *RRDtool* (*Round-Robin Database*) [28] to store and archive all time-series data. 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, a given machine in a Hyracks cluster can install one or more *NCs (Hyracks Node Controllers)*. To map the logical NC nodes to the real physical nodes of the cluster, we created an extra file that represents and stores this mapping information (as pairs of a machine's IP address and a path of an RRD table).

4.2.2 Server Push Mechanism

Another technique used for client-server communications in the console is a *server-push* mechanism [22], in which the server sends messages independent from any requests from clients. Whenever new information is available, the server "publishes" that information in a specific "channel." To receive a message, clients have to "subscribe" or "listen" to that "channel." This service is used to deliver any dynamic information that should be expressed in advance (without any requests), including notification of a new Hyracks job and notification of completed stages and/or failure stages of a given Hyracks job. Even though this technique increases the number of messages on the server, we adopted this approach to provide real-time support and dynamic information for the Hyracks console. We implemented this service using *CometD*, a scalable HTTP-based with a push technology pattern know as *Comet* over a *Bayeux* protocol [29, 30]. The Bayeux protocol can transport asynchronous messages with low latency between a web server and a web client [31]. These messages are routed via named channels and can be delivered in many directions (servers to clients, clients to servers, and clients to clients).

5 IMPLEMENTATION

This section describes the implementation of the Hyracks Console in greater detail. It describes the benefits and usages of each type of monitoring data along with actual results returned by the console. 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 the `http-port` number is "`2099`." All of the URL examples shown here are based on that assumption.

5.1 REST API for Client Pulling Method

The Jetty HTTP web server is embedded in the Hyracks console architecture to support communication between the *Hyracks Console Server (HCS)* and the outside world including the *Hyracks Console Visualization (HCV)* component. This server is started automatically when users start the *Hyracks Cluster Controller (CC)* process and stopped when users stop the CC process. The communication port number is setup based on the optional argument, `http-port` number, when starting the CC process (the default value is 2099). To handle the client-pull mechanism in the Jetty server, we simply build server connectors and handlers without using the Servlets API. Each handler added in this server is responsible for generating different types of results depended on the client requests. The sample code of creating "/state", "/profile" and "/console" handlers is presented in Figure 8.

```
private final HandlerCollection handlerCollection;

private void addHandlers() {
  ContextHandler handler = new ContextHandler("/state");
  RoutingHandler rh = new RoutingHandler();
  rh.addHandler("jobs", new JSONOutputRequestHandler(new JobsRESTAPIFunction(ccs)));
  handler.setHandler(rh);
  addHandler(handler);

  handler = new ContextHandler("/profile");
  handler.setHandler(new AdminConsoleProfileHandler(ccs));
  addHandler(handler);

  handler = new ContextHandler("/console");
  rh = new RoutingHandler();
  rh.addHandler("nodes", new JSONOutputRequestHandler(new NodesRESTAPIFunction(ccs)));
  rh.addHandler("cluster", new JSONOutputRequestHandler(new ClusterRESTAPIFunction(ccs)));
  handler.setHandler(rh);
  addHandler(handler);
}

public void addHandler(Handler handler) {
  handlerCollection.addHandler(handler);
}
```

Figure 8: addHandlers() and addHandler() Methods in WebServer.java

The information that typical Hyracks users request for monitoring the Hyracks system can be boiled down into two main categories: Hyracks jobs' execution information and Hyracks cluster behavior. Given the REST-style architecture, we create a global URL for each type of resource. Accordingly, the better navigating throughout every resource using the `RoutingHandler` object, we organize the "URL-Request" paths based on the "Tree Structure" as depicted in Figure 9 (as well as Figure 21 in Section 5.1.2).

As we mentioned earlier, most of the data returned from the HCS is in the JSON format. Consequently, to provide a simple standard to the system, we build a Java object interface for outputting the result, `IJSONOutputFunction`. Implementing an interface allows a class to become more formal about the behavior that it promises to provide. Any classes that claim to implement an interface must develop all methods defined by that interface. Interfaces also form a contract between the classes and the outside world, and this contract is enforced at build time by

the compiler. The Hyracks Console creates three classes from the `IJSONOutputFunction` interface: `JobsRESTAPIFunction`, `NodesRESTAPIFunction` and `ClusterRESTAPIFunction`. The first class is used for creating all information related to a Hyracks job's execution, and the rests are responsible for producing results regarding the Hyracks cluster's health.

5.1.1 Hyracks Jobs

To assist Hyracks users monitor and understand the internal actions of each Hyracks job's execution, we provide the following monitoring information via the `JobsRESTAPIFunction`: (1) summary of all jobs submitted to the cluster, as well as each job's (2) specification, (3) plan, (4) profile, and (5) stages. Figure 9 presents the URL paths that can be used for requesting different types of information.

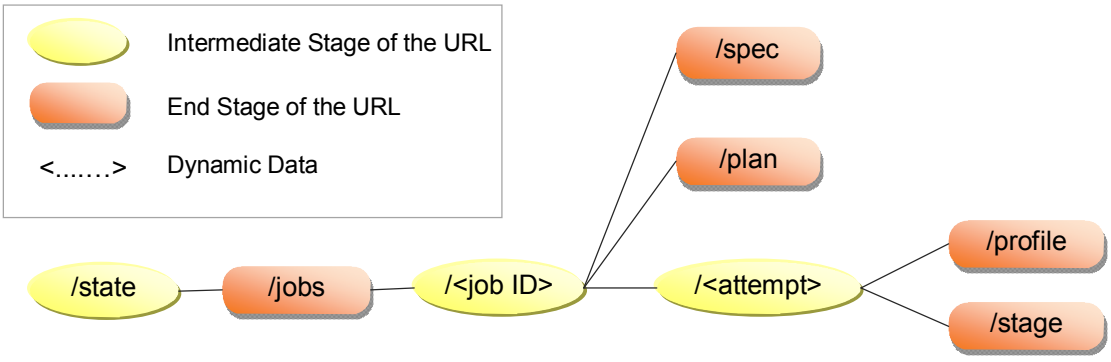


Figure 9: URL-Request Paths for the Hyracks Jobs Execution Information

In this section, we are going to examine each type of information found in the Hyracks Console and describe its usage based on real examples. Note that most of the data delivered from the *Hyracks Console Server (HCS)* is presented in the *JSON* format.

- **Summary of Hyracks Jobs**: When a client in the *Hyracks Console Visualization (HCV)* sends a request to the *Hyracks Console Server (HCS)* through the *Job-Summary URL*, it receives a JSONArray containing all Hyracks jobs that have been submitted to the Hyracks system. Each job presents some important job-related entities such as job's "ID", job's "display-name", job's "application" name, job's current "status" (i.e., `INITIALIZED`, `RUNNING`, `TERMINATED`, or `FAILURE`), job's "events" with time-stamp (e.g., job is terminated event), and the number of "attempts." Figure 10 is an example result for a Jobs Summary URL request.

URL Path

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

Example

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

```
{
    result: [
            {
                    id: "a965fe5a-ba0b-4a01-a13d-1b4dcc57adc1"
                    application: "tpch"
                    display-name: "job00000003"
                    status: "TERMINATED"
                    events: [
                            {
                                    status: "RUNNING"
                                    system-time: 1305922493368
                                    date: "2011-05-20 13:14:53.368"
                            }
                            {
                                    status: "INITIALIZED"
                                    system-time: 1305922493361
                                    date: "2011-05-20 13:14:53.361"
                            }
                            {
                                    status: "TERMINATED"
                                    system-time: 1305922494104
                                    date: "2011-05-20 13:14:54.104"
                            }
                    ]
                    attempts: 1
                    type: "job-summary"
            }
            {
                    id: "54d395c4-289f-4de9-ab54-c429b006a968"
                    application: "btree"
                    display-name: "job00000010"
                    status: "FAILURE"
                    events: [
                            {
                                    status: "RUNNING"
                                    system-time: 1305924975041
                                    date: "2011-05-20 13:56:15.041"
                            }
                            {
                                    status: "FAILURE"
                                    system-time: 1305925628659
                                    date: "2011-05-20 14:07:08.659"
                            }
                            {
                                    status: "INITIALIZED"
                                    system-time: 1305924975021
                                    date: "2011-05-20 13:56:15.021"
                            }
                    ]
                    attempts: 6
                    type: "job-summary"
            }
    ]
}
```


Figure 10: Example Result from Hyracks Jobs Summary URL

- **Job Specification**: When a client sends a request to the HCS via the *Job-Spec URL* to gather the job specification for a particular job, it receives a JSONArray containing two types of data: a set of Hyracks connectors and a set of Hyracks operators. From that information, users are able to recreate a Hyracks jobs' specification (a DAG dataflow diagram) to ensure that their jobs are constructed correctly. For example, in Figure 11 our example job's specification consists of three operators and four connectors. Each connector in the `connectors` JSONArray is used to connect between two operators in the `operators` JSONArray indicated by these two entities: `in-operator-id` and `out-operator-id`. Figure 12 (left) shows the job's specification diagram generated from the information in Figure 11.

URL Path

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

Example

`http://vanilla.ics.uci.edu:2099/state/jobs/54d395c4-289f-4de9-ab54-c429b006a968/spec`

```
{
  result: {
    connectors: [
      {
        in-operator-port: 0
        in-operator-id: "ODID:3b93bca7-b93a-4fab-87a6-ede060abb96e"
        connector: {
          id: "f4046a40-e78a-4492-af15-b25ff310994a"
          java-class: "edu.uci.ics.hyracks.dataflow.std.connectors.MToNHashPartitioningConnectorDescriptor"
          type: "connector"
        }
        type: "connector-info"
        out-operator-port: 0
        out-operator-id: "ODID:52c2e2a7-6aef-4fe3-94e3-dfa77ecb4faf"
      }
      {
        in-operator-port: 0
        in-operator-id: "ODID:52c2e2a7-6aef-4fe3-94e3-dfa77ecb4faf"
        connector: {
          id: "e865252b-ba52-499f-a789-9084b1329edf"
          java-class: "edu.uci.ics.hyracks.dataflow.std.connectors.OneToOneConnectorDescriptor"
          type: "connector"
        }
        type: "connector-info"
        out-operator-port: 0
        out-operator-id: "ODID:7b34c851-fa5f-4d1b-94c3-cd8fa1ae5ce6"
      }
    ]
    operators: [
      {
        id: "3b93bca7-b93a-4fab-87a6-ede060abb96e"
        in-arity: 0
        java-class: "edu.uci.ics.hyracks.examples.btree.helper.DataGenOperatorDescriptor"
        out-arity: 1
        type: "operator"
      }
      {
        id: "52c2e2a7-6aef-4fe3-94e3-dfa77ecb4faf"
        in-arity: 1
        java-class: "edu.uci.ics.hyracks.dataflow.std.sort.ExternalSortOperatorDescriptor"
        out-arity: 1
        type: "operator"
      }
      {
        id: "7b34c851-fa5f-4d1b-94c3-cd8fa1ae5ce6"
        in-arity: 1
        java-class: "edu.uci.ics.hyracks.storage.am.btree.dataflow.BTreeBulkLoadOperatorDescriptor"
        out-arity: 0
        type: "operator"
      }
    ]
    type: "job"
  }
}
```


Figure 11: Example Result from Hyracks Job's Specification URL

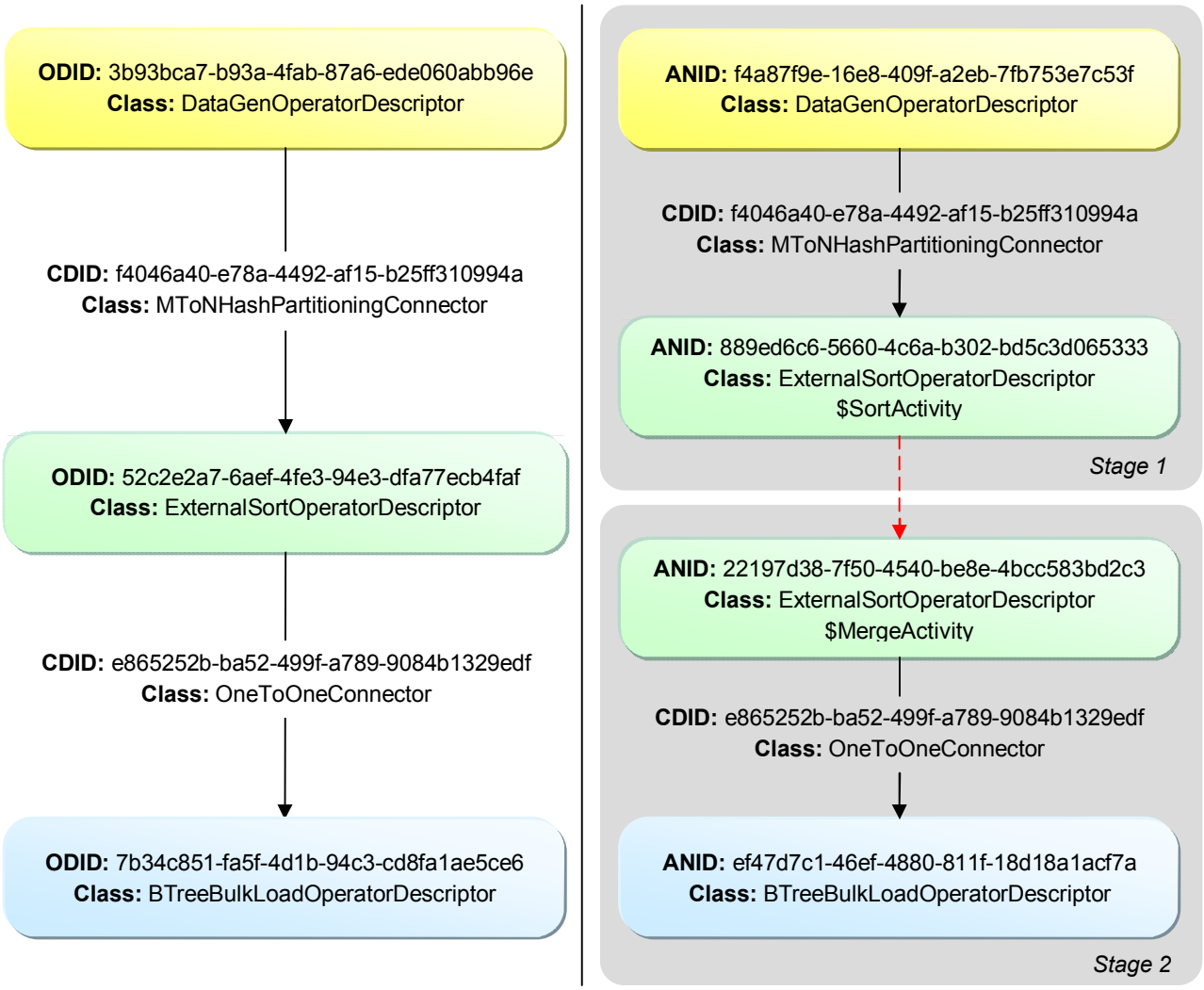


Figure 12: Example of Job's Specification (left) and Job's Plan (right)

- ***Job Plan***: When clients send a Job-Plan URL request, they receive a JSONArray of the job's plan information. This result consists of all of the job's Hyracks activities, which are expanded from the set of the Hyracks operators in the job's specification. Each activity has a set of input and/or output Hyracks connectors that are used to construct the *Hyracks Activity Node Graph (Hyracks Job Plan)*. In addition, the `depends-on` entity in the result shows the "ID" of the blocking activities, specifying the dataflow constraint between each activity. All activities connected by non-blocking edges are grouped together to form a job's stage. Recall that one Hyracks job is normally divided into many stages, and every stage is executed, one-by-one, based on its readiness to run. The job's plan gives insight into the Hyracks job's execution steps. For example, Figure 12 (right) is the plan diagram generated from the information in Figure 13. The "ExternalSortOperatorDescriptor" operator in Figure 12 (left) has been replaced with two activities, "SortActivity" and "MergeActivity," that are connected to each other by a blocking edge. Consequently, this job's plan is divided into two stages in which the second stage cannot be executed until the first stage is completed.

URL Path

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

Example

```
http://vanilla.ics.uci.edu:2099/state/jobs/54d395c4-289f-
4de9-ab54-c429b006a968/plan
```

```
{
  result: {
    id: "54d395c4-289f-4de9-ab54-c429b006a968"
    activities: [
      {
        id: "ANID:ef47d7c1-46ef-4880-811f-18d18a1acf7a"
        owner-id: "ODID:7b34c851-fa5f-4d1b-94c3-cd8fa1ae5ce6"
        inputs: [ {
            input-port: 0
            connector-id: "CDID:e865252b-ba52-499f-a789-9084b1329edf"
            type: "activity-input"
            connector-java-class:"edu.uci.ics.hyracks.dataflow.std.connectors.
                                  OneToOneConnectorDescriptor"
          } ]
        java-class: "edu.uci.ics.hyracks.storage.am.btree.dataflow.BTreeBulkLoadOperatorDescriptor"
        type: "activity"
      }
      {
        id: "ANID:22197d38-7f50-4540-be8e-4bcc583bd2c3"
        owner-id: "ODID:52c2e2a7-6aef-4fe3-94e3-dfa77ecb4faf"
        java-class: "edu.uci.ics.hyracks.dataflow.std.sort.ExternalSortOperatorDescriptor
                     $MergeActivity"
        outputs: [ {
            connector-id: "CDID:e865252b-ba52-499f-a789-9084b1329edf"
            type: "activity-output"
            connector-java-class: "edu.uci.ics.hyracks.dataflow.std.connectors.
                                   OneToOneConnectorDescriptor"
            output-port: 0
          } ]
        type: "activity"
        depends-on: [ "ANID:889ed6c6-5660-4c6a-b302-bd5c3d065333" ]
      }
      {
        id: "ANID:f4a87f9e-16e8-409f-a2eb-7fb753e7c53f"
        owner-id: "ODID:3b93bca7-b93a-4fab-87a6-ede060abb96e"
        java-class: "edu.uci.ics.hyracks.examples.btree.helper.DataGenOperatorDescriptor"
        outputs: [ {
          connector-id: "CDID:f4046a40-e78a-4492-af15-b25ff310994a"
          type: "activity-output"
          connector-java-class: "edu.uci.ics.hyracks.dataflow.std.connectors.
                                 MToNHashPartitioningConnectorDescriptor"
          output-port: 0
          } ]
        type: "activity"
      }
      {
        id: "ANID:889ed6c6-5660-4c6a-b302-bd5c3d065333"
        owner-id: "ODID:52c2e2a7-6aef-4fe3-94e3-dfa77ecb4faf"
        inputs: [ {
            input-port: 0
            connector-id: "CDID:f4046a40-e78a-4492-af15-b25ff310994a"
            type: "activity-input"
            connector-java-class: "edu.uci.ics.hyracks.dataflow.std.connectors.
                                   MToNHashPartitioningConnectorDescriptor"
          } ]
        java-class: "edu.uci.ics.hyracks.dataflow.std.sort.ExternalSortOperatorDescriptor
                     $SortActivity"
        type: "activity"
      }
    ]
    type: "plan"
  }
}
```


Figure 13: Example Result from Hyracks Job's Plan URL

- ***Job Stages***: Unlike the results of the job's plan or job's specification which are static data, when users request a job's Job-Stage URL, they get dynamic data about the progress of the job's stage by stage execution. Here, the result is composed of several important entities; for example, the current status of each stage (e.g., `pending`, `in-progress`, or `completed`), the set of activities in each stage, the list of nodes assigned to process each activity, and the time-stamp of the job's state change events (e.g., `init-time`, `finish-time`, and `fail-time`) on a stage by stage basis. As shown in Figure 14, users can monitor the progress of their jobs' execution by observing each stage's status. Job execution is terminated while there is no "pending" stage. Figure 15 provides the sense of parallelism of each activity in the stage. As mentioned in the Hyracks overview, each activity is cloned into a set of tasks that are scheduled to run on a set of nodes. The number of nodes at each activity, therefore, indicates the degree of parallelism decided by the Hyracks system. In particular, the `node-events` presents the actual time that each node spends for executing a Hyracks job at each stage.

URL Path

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

Example

```
http://vanilla.ics.uci.edu:2099/state/jobs/54d395c4-289f-
4de9-ab54-c429b006a968/0/stage
```


```
{
  result: {
    attempt: 0
    pending-stage-ids: [
      "5ce9c97c-ba11-48e0-8ff0-01f9033d37fa"
    ]
    in-progress-stage-ids: [
      "079046ee-0e2f-435c-a7ce-578aedfc5bb6"
    ]
    completed-stage-ids: [
      "5f2a92e5-099f-4a51-978b-54985ab551ee"
    ]
    type: "stage-attempt"
```


Figure 14: Example Result from Hyracks Job's Stage URL

```
stages: [
  {
    stage-id: "5f2a92e5-099f-4a51-978b-54985ab551ee"
    stage-finish-time: 1305924975320
    stage-init-time: 1305924975053
    stage-status: "completed"
    stage-name: "stage_1"
    activities: [
      {
        activity-id: "ANID:f4a87f9e-16e8-409f-a2eb-7fb753e7c53f"
        java-class: "edu.uci.ics.hyracks.examples.btree.helper.
                     DataGenOperatorDescriptor"
        operator-id: "ODID:3b93bca7-b93a-4fab-87a6-ede060abb96e"
        node-names: [
          "nc1"
        ]
      }
      {
        activity-id: "ANID:889ed6c6-5660-4c6a-b302-bd5c3d065333"
        java-class: "edu.uci.ics.hyracks.dataflow.std.sort.
                     ExternalSortOperatorDescriptor$SortActivity"
        operator-id: "ODID:52c2e2a7-6aef-4fe3-94e3-dfa77ecb4faf"
        node-names: [
          "nc1"
          "nc2"
          "nc3"
          "nc4"
        ]
      }
    ]
    nodes-events: [
      {
        init-time: 1305924975058
        finish-time: 1305924975318
        node-id: "nc1"
      }
      {
        init-time: 1305924975058
        finish-time: 1305924975319
        node-id: "nc2"
      }
      {
        init-time: 1305924975058
        finish-time: 1305924975305
        node-id: "nc3"
      }
      {
        init-time: 1305924975058
        finish-time: 1305924975316
        node-id: "nc4"
      }
    ]
  }
```


Figure 15: Example Result from Hyracks Job's Stage URL (cont.)

- ***Job Profile:*** This information represents data movement at the task-execution level. The unit of data produced and consumed by Hyracks tasks is called a "Frame." Each task-runtime only operates on one stream of frames (per input), without consideration of any repartitioning needed for the output. The repartitioning of the frames is handled separately by a Hyracks connector. Each connector-runtime has two sides: the sender and the receiver, which each can consist of several instances. A job's profile provides a counter of how many times each sender instance sends a frame to each receiver instance. Figure 16 depicts a sample result from a sending job-profile URL request; it can be interpreted into the diagram as shown in Figure 17.

  In Figure 16, the format of the profile counters is

  "`<connector-id>.sender.<sender-id>.<receiver-id>.nextFrame.`"

  To turn on the collection of profiling data, which is disabled by default, Hyracks users need to set a `profile-dump-period` argument (in the millisecond unit) while first starting the Hyracks Cluster Controller process (e.g., `hyrackscc -profile-dump-period 10000`, which means that the profiling data should be updated every 10 seconds).

URL Path

```
http://<cc-ip-address>:<http-port>/state/jobs/<job-id>/
<attempts>/profile
```

Example

```
http://vanilla.ics.uci.edu:2099/state/jobs/54d395c4-289f-
4de9-ab54-c429b006a968/0/profile
```

```
{
  result: {
    job-id: "54d395c4-289f-4de9-ab54-c429b006a968"
    attempt: 0
    joblets: [
      {
        node-id: "nc1"
        type: "joblet-profile"
        stagelets: {
          stage-id: "5f2a92e5-099f-4a51-978b-54985ab551ee"
          counters: [
            {
              name: "f4046a40-e78a-4492-af15-b25ff310994a.sender.0.0.open"
              value: 1
            }
            {
              name: "f4046a40-e78a-4492-af15-b25ff310994a.sender.0.1.open"
              value: 1
            }
            {
              name: "f4046a40-e78a-4492-af15-b25ff310994a.sender.0.2.open"
              value: 1
            }
            {
              name: "f4046a40-e78a-4492-af15-b25ff310994a.sender.0.3.open"
              value: 1
            }
            {
              name: "f4046a40-e78a-4492-af15-b25ff310994a.sender.0.0.nextFrame"
              value: 33
            }
            {
              name: "f4046a40-e78a-4492-af15-b25ff310994a.sender.0.1.nextFrame"
              value: 92
            }
            {
              name: "f4046a40-e78a-4492-af15-b25ff310994a.sender.0.2.nextFrame"
              value: 101
            }
            {
              name: "f4046a40-e78a-4492-af15-b25ff310994a.sender.0.3.nextFrame"
              value: 190
            }
            {
              name: "f4046a40-e78a-4492-af15-b25ff310994a.sender.0.0.close"
              value: 0
            }
            {
              name: "f4046a40-e78a-4492-af15-b25ff310994a.sender.0.1.close"
              value: 0
            }
            {
              name: "f4046a40-e78a-4492-af15-b25ff310994a.sender.0.2.close"
              value: 0
            }
            {
              name: "f4046a40-e78a-4492-af15-b25ff310994a.sender.0.3.close"
              value: 0
            }
```


Indicate that the node-id of the sender-id “0” is “nc1”

(Note: This figure continues on the next page)

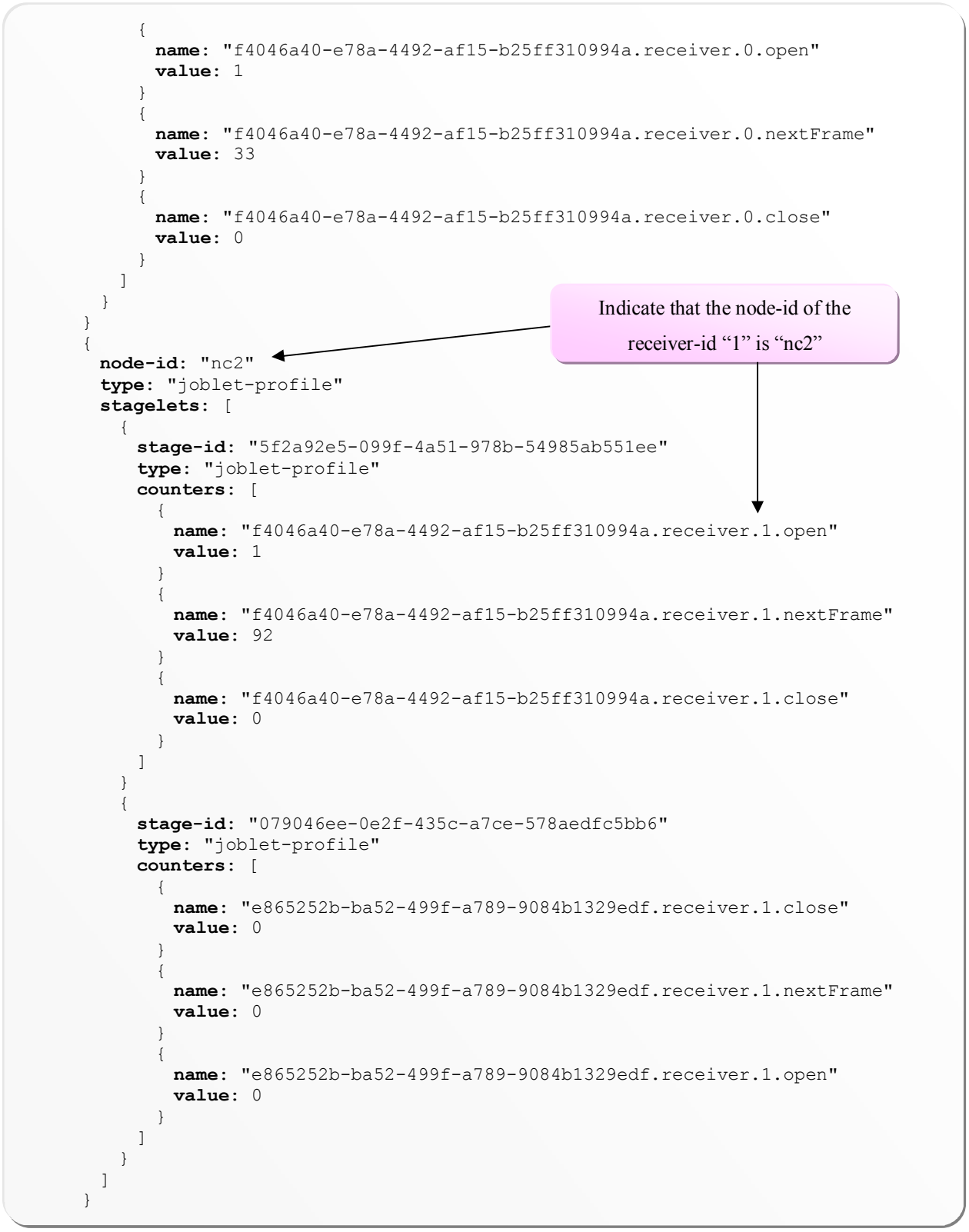


Figure 16: Example Result from Hyracks Job's Profile URL

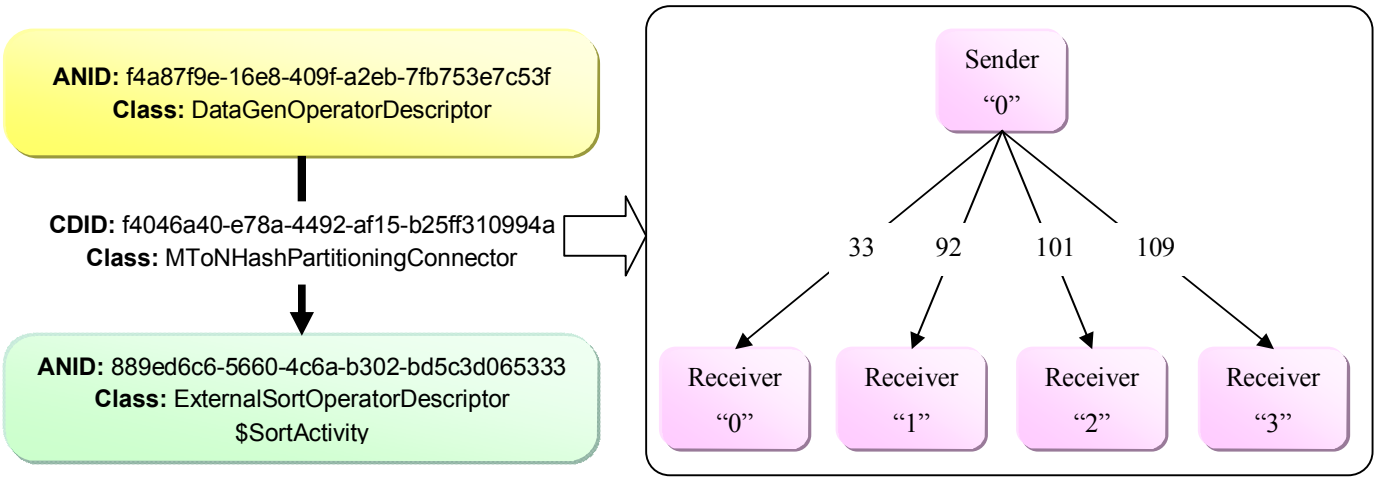


Figure 17: Example of Data Movement at the M to N Hash Partitioning Connector
(This example connector consists of one instance on the sender side and four instances on the receiver side. There are 33, 92, 101 and 190 frames of data sent from sender "0" to receiver "0", "1", "2" and "3" respectively.)

It is clear from the figure that, this profiling information can become extremely large if data is shuffled among thousands of nodes, and users may not be able to get an understandable overview of the data movement at each connector from the data at this fine granularity. Therefore, another URL path (shown in the box below) is provided to request the profiling data at each connector via graphical image of a matrix representing data flow between senders and receivers, as displayed in Figure 18 and Figure 19. The optional `number` in the URL path below says that users also want to show the actual number of frames, which are moved from senders to receivers, on the profile image.

URL Path

```
http://<cc-ip-address>:<http-port>/profile/<job-id>/
<attempts>/<connector-id>/(number)
```

Example

```
http://vanilla.ics.uci.edu:2099/profile/54d395c4-289f-4de9-
ab54-c429b006a968/0/f4046a40-e78a-4492-af15-b25ff310994a/number
```

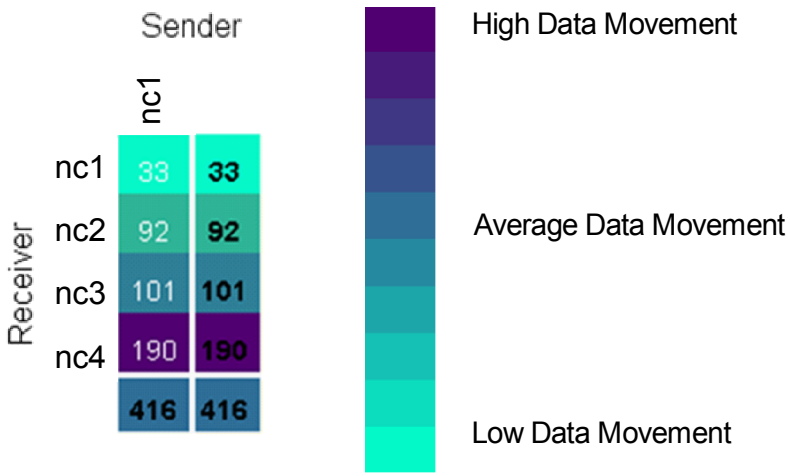


Figure 18: Small Scale Communication Matrix Image and the Gradient Scale Representing the Partitioned Data Movement from Low to High

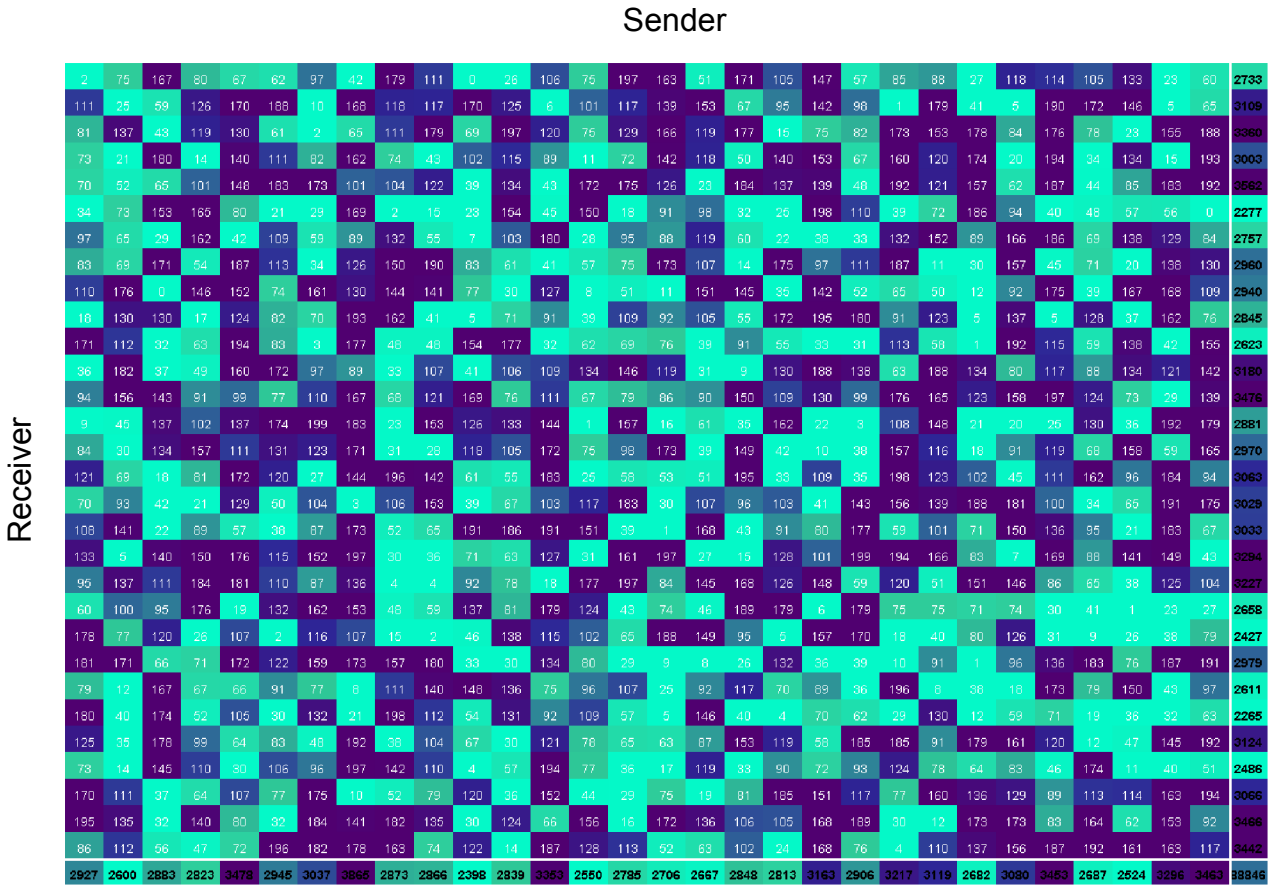


Figure 19: Large Scale Communication Matrix Image (30 sender instances and 30 receiver instances)

Figure 18 shows a communication matrix image of a given Hyracks connector-runtime of a Hyracks job running on a small-scale Hyracks cluster. The data is sent from one sender instance to four receiver instances. To assume that the sender instance is operated by the node-id "nc1" and each receiver instance is operated by the node-id "nc1", "nc2", "nc3" and "nc4." There are 33, 92, 101 and 190 frames sent from the sender node "nc1" to the receiver nodes "nc1", "nc2", "nc3" and "nc4" respectively. The total number of frames that has been sent is 416. Figure 19 shows a communication image of a particular Hyracks connector of a Hyracks job running on a large-scale Hyracks cluster where there are 30 instances on the sender side and 30 instances on the receiver side. The last column in the matrix image presents the total number of frames that each receiver obtains. Moreover, the last row shows the total number of frames which each sender delivers.

The color of each cell in the communication matrix image reflects the number of frames that is sent among sender and receiver nodes. The colors range from the light green color [RGB: 4 249 200] (low data movement) to the dark purple color [RGB: 80 0 112] (high data movement). If data transfer is partitioned equally among all participating nodes, every cell in the profile image will have the same color, which is the navy color [RGB: 46 110 151] (the middle color in the gradient scale). These colors are being computed by increasing or decreasing RBG (Red Green Blue) value of the middle color (navy color) toward high (purple) or low (green) color. JHeatChart [32], a simple Java library for generating heat map charts, has been modified and used to generate the communication matrix image.

From Figure 18, the color of the second cell in the matrix image can be computed as shown below. The source code to calculate a cell color is also presented in Figure 20.

1) Calculating an average number of frames transfer from every sender instance to every receiver instance.

   E.g., avg = (33 + 92 + 101 + 190) / 4 = 104 frames

2) Finding a minimum and maximum value in the matrix and calculate a different between the average value to the minimum value and maximum values.

   E.g., rangeDown = avg – min = 104 – 33 = 71

   rangeUp = max - avg = 190 – 104 = 86

3) Calculating a value distance between a matrix cell's value and the average value.

   E.g., the value of the second cell is 92. Thus, distance = |data – avg| = |92 – 104| = 12

4) Finding a proportion distance of the cell's value toward the minimum or maximum value. If the cell's value is larger or equal to the average value, the rangeUp value (from step 2) is used. Otherwise, the rangeDown value will be used.

   E.g., value of the second cell, 92, is less than the average value, so
   percentPosition = distance / rangeDown = 12 / 71 = 0.169

5) If the cell's value is higher or equal to the average, we calculate the color value distance between the navy color [RGB: 46 110 151] (reflects the average value) and the dark purple color [RGB: 80 0 112] (reflects the maximum value). If not, the color value distance is computed from the distance between the navy color and the light green color [RGB: 4 249 200] (reflects the minimum value).

   E.g., value of the second cell, 92, is less than the average, so

$$\text{colourValueDistance} = |R_{navy} - R_{green}| + |G_{navy} - G_{green}| + |B_{navy} - B_{green}|$$
$$= |46 - 4| + |110 - 249| + |151 - 200| = 42 + 139 + 49 = 230$$

6) Calculating a number of times that the color value is required to shift.

   E.g., colourPosition = colourValueDistance * percentPosition = 230 * 0.169 ≈ 39

7) The red (R), green (G) and blue (B) values of a given cell start with same value as the red, green and blue values of the average color (in this example, the values are 46, 110, and 151 respectively). Then, the cell's R, G and B values are changed toward either the low-end color or the high-end color by decreasing or increasing either R, G or B value one-by-one until the number of shifting times reaches the colourPosition number (from step 6). In each round, we compute three color distance pairs between the R, G and B of the given cell and the R, G and B of the low/high-end color. Then, a color (R, G or B color of the given cell) with the largest distance is selected to be added or deducted one value at a time.

   E.g., toward the low-end color, the distance of the green (G) value between the light green and the navy is much larger than the distance of the read (R) and blue (B) colors, thus the green value is kept increasing from 110 to 149 (increase one value at a time until reach 39 times).

8) Finally, generating a new color from the Red, Green, and Blue color from setp 7.

   E.g., the result color of the second cell (value 92) in the matrix image is
   a dark green color [RGB: 46 149 151] as shown in Figure 18.

```
private Color getCellColour(double data, double min, double max, double avg) {

   double rangeUp = max - avg;
   double rangeDown = avg - min;
   double distance = Math.abs(data - avg);

   int r = midValueColour.getRed();
   int g = midValueColour.getGreen();
   int b = midValueColour.getBlue();

   // What proportion of the way through the possible values is that.
   double percentPosition;
   int r2, g2, b2;
   if(data >= avg){
      percentPosition = distance / rangeUp;
      r2 = highValueColour.getRed();
      g2 = highValueColour.getGreen();
      b2 = highValueColour.getBlue();
   }
   else{
      percentPosition = distance / rangeDown;
      r2 = lowValueColour.getRed();
      g2 = lowValueColour.getGreen();
      b2 = lowValueColour.getBlue();
   }
   int colourValueDistance = Math.abs(r - r2) + Math.abs(g - g2) + Math.abs(b - b2);
   int colourPosition = (int) Math.round(colourValueDistance * percentPosition);

   // Make n shifts of the colour, where n is the colourPosition.
   for (int i=0; i<colourPosition; i++) {
      int rDistance = r - r2;
      int gDistance = g - g2;
      int bDistance = b - b2;

      if ((Math.abs(rDistance) >= Math.abs(gDistance))
            && (Math.abs(rDistance) >= Math.abs(bDistance))) {
         // Red must be the largest.
         r = changeColourValue(r, rDistance);
      } else if (Math.abs(gDistance) >= Math.abs(bDistance)) {
         // Green must be the largest.
         g = changeColourValue(g, gDistance);
      } else {
         // Blue must be the largest.
         b = changeColourValue(b, bDistance);
      }
   }
   return new Color(r, g, b);
}

private int changeColourValueDown(int colourValue, int colourDistance) {
   if (colourDistance < 0) {
      return colourValue-1;
   } else if (colourDistance > 0) {
      return colourValue+1;
   } else {
      // This shouldn't actually happen here.
      return colourValue;
   }
}
```

Figure 20: Source Code for Calculating Cell Color

5.1.2 Hyracks Cluster Health

Besides the Hyracks jobs' information, the Hyracks Console allows users to monitor the overall Hyracks Cluster Health. Every Hyracks cluster consists of two types of nodes: *Hyracks Cluster Controllers* (CCs) and *Hyracks Node Controllers* (NCs). In reality, production clusters will generally consist of thousands of nodes. At that scale, users will not be able to access each physical node, one-by-one, to gather an overview of the current state of the cluster. As a result, the following Request-URL paths in Figure 21 are created to assist users retrieving cluster monitoring information. The data related to CC and NC(s) in the cluster is very important to diagnose the overall cluster health. This data is divided into five main categories: *CC configuration*, *NC configuration*, *NC summary*, *NC jobs summary*, and *NC resources*. We will explain the further details below.

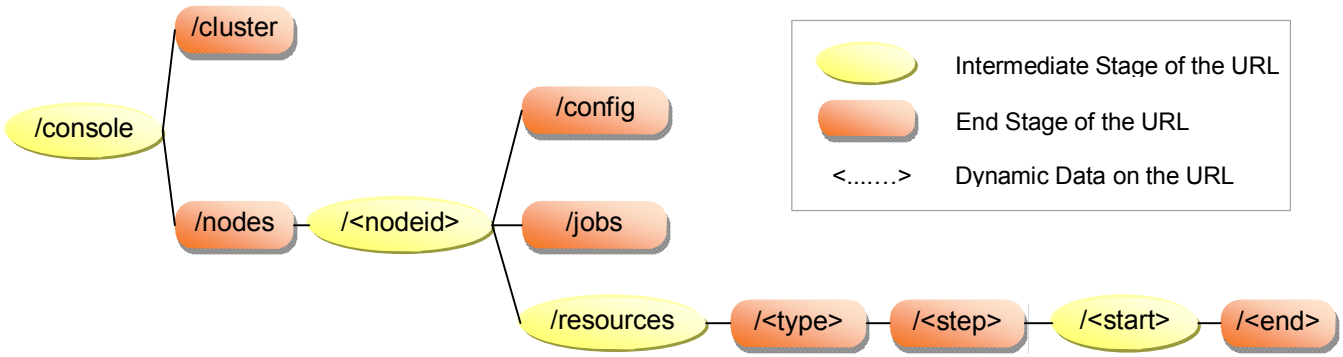


Figure 21: URL Paths for Monitoring the Hyracks Cluster

- **Hyracks Cluster Controller Configuration:**

  The *Hyracks Cluster Controller (CC)* plays a significant role in the Hyracks system. It is the main process to handle job control communications between the master node and worker nodes in the Hyracks cluster and schedules all jobs' execution. To ensure that the CC is setup correctly, users can check with the `cc-config` entity which reports all configuration variables, including `cc-port`, `http-port`, `heartbeat-period`, `max-heartbeat-lapse-periods`, `default-max-job-attempts` and `profile-dump-period`. Note that the CC configuration values cannot be changed unless users restart the CC process. The result of requesting the CC-Config URL also includes significant information about applications deployed in the clusters. Shown in Figure 22 is the result of an example result

for the Hyracks Cluster Controller. There are three applications deployed, which are `tpch`, `btree` and `fuzzyjoin`.

URL Path

```
http://<cc-ip-address>:<http-port>/console/cluster
```

Example

```
http://vanilla.ics.uci.edu:2099/console/cluster
```


```
{
   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"
         }
      ]
   }
}
```


Figure 22: Example Result from CC Configuration URL

- **Hyracks Node Controller Configuration:**

  All node configuration variables and their values for a given node controller are returned from the Hyracks Console Server when clients send Node-Config URL request, as shown in

  Figure 23. (Note that if users intend to adjust its configuration values, they need to restart the Node Controller.)

URL Path

```
http://<cc-ip-address>:<http-port>/console/nodes/
<node-id>/config
```

Example

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


```
{
   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"
   }
}
```


Figure 23: Result from Node Configuration URL

- **Hyracks Node Controllers Summary:**

  When clients send a Node-Summary URL request, they get a `JSONArray` containing a list of all nodes associated with the monitored Cluster Controller, as shown in Figure 24. Each node consists of four entities, `id`, `host`, `load_one` and `type`. The `host` entity at each node indicates the liveness of each node. If the `host` value of a given node is less than the `max-heartbeat-lapse-periods` value (in CC configuration), the node is still alive. The

`load_one` entity is one-minute load average.[1] The load average is recalculated every five seconds based on the current system load. The system load is the number of processes that are running on the machine plus the number of processes that are waiting to run. On the lightly loaded node, the current system load varies between 1 and 0.

URL Path

`http://<cc-ip-address>:<http-port>/console/nodes`

Example

`http://vanilla.ics.uci.edu:2099/console/nodes`


```
{
   result: [
      {
         id: "nc1"
         host: 0
         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: 1
         load_one: "3.8733333333e-01"
         type: "node-summary"
      }
   ]
}
```


Figure 24: Result from Nodes Summary URL

[1] http://book.opensourceproject.org.cn/enterprise/cluster/linuxcluster/opensource/11021/bbl0140.html

- **Hyracks Node Controller Jobs Summary:**

  The node controller jobs summary URL represents the list of Hyracks jobs on a given node. Each job is classified into 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. An example result is presented in Figure 25.

URL Path

```
http://<cc-ip-address>:<http-port>/console/nodes/
<node-id>/jobs
```

Example

```
http://vanilla.ics.uci.edu:2099/console/nodes/nc1/jobs
```


```
{
   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 25: Result from Node Jobs Summary URL

- **Hyracks Node Controller Resources:**

The CC process on the master node is significant for controlling the Hyracks system, but information about physical resource consumption at the master node itself is less important, since the computation is actually done at the worker nodes that run the NC process. Information about the resource usage of each worker machine is beneficial in order to monitor the cluster's health and detect any job execution failures caused by node failures. When clients send a request via the NC-Resources URL, they receive the time series data for resource consumption at the specified Node Controller nodes.

The time series data is collected by the Ganglia software and stored in the RRD database. By default, there are five main categories of the resource; for example, 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). Users can more specify about the desired data by adding these following filters at the end of the URL: resource type(s), frequency of data (e.g., every 15 or 90 seconds) and a time interval.

URL Path

```
http://<cc-ip-address>:<http-port>/console/nodes/
<node-id>/resources/<type>/(<step>)/(<start-time>)/(<end-time>)
```

Example

```
(1)http://vanilla.ics.uci.edu:2099/console/nodes/nc1/resources/mem
(2)http://vanilla.ics.uci.edu:2099/console/nodes/nc1/resources/load
/90/1303333720/1303334720
```

As above display shown the first example URL asks for the memory consumption of the node-id "nc1" in the Hyracks cluster. By default, the resolution `step` of resource data is every 15 seconds, and the `end-time` is "now". The number of records returned by the web-server is about 30 records per type of resource. In other words, the default resource usage URL returns the resource consumption in the last 450 seconds (= 15 seconds * 30 records). The sample result (only the first ten data records) from this URL is shown in Figure 26. The `type` entity represents the resource category and `sub-type` entity indicates the sub-type of

the resource data in a given category; for instance, the MEM resource type has mem_buffers and mem_cached as a sub-type. Each `data` record consists of three entities, `num`, `time` and `value` to represent a value of resource usage at a specific time. The `num` entity shows a number of machines (always equal to 1 for every NC).


```
{
  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"
        }
        {
          num: "1.0000000000e+00"
          time: "06:52:30 PM"
          value: "2.5506666667e+03"
        }
        {
          num: "1.0000000000e+00"
          time: "06:52:45 PM"
          value: "2.5680000000e+03"
        }
        {
          num: "1.0000000000e+00"
          time: "06:53:00 PM"
          value: "2.5680000000e+03"
        }
        {
          num: "1.0000000000e+00"
          time: "06:53:15 PM"
          value: "2.6200000000e+03"
        }
        {
          num: "1.0000000000e+00"
          time: "06:53:30 PM"
          value: "2.6200000000e+03"
        }
        {
          num: "1.0000000000e+00"
          time: "06:53:45 PM"
          value: "2.6426666667e+03"
        }
        {
          num: "1.0000000000e+00"
          time: "06:54:00 PM"
          value: "2.6880000000e+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"
        }
        {
          num: "1.0000000000e+00"
          time: "06:52:30 PM"
          value: "6.1538666667e+04"
        }
        {
          num: "1.0000000000e+00"
          time: "06:52:45 PM"
          value: "6.1528000000e+04"
        }
        {
          num: "1.0000000000e+00"
          time: "06:53:00 PM"
          value: "6.1528000000e+04"
        }
        {
          num: "1.0000000000e+00"
          time: "06:53:15 PM"
          value: "6.1380000000e+04"
        }
        {
          num: "1.0000000000e+00"
          time: "06:53:30 PM"
          value: "6.1380000000e+04"
        }
        {
          num: "1.0000000000e+00"
          time: "06:53:45 PM"
          value: "6.2880000000e+04"
        }
        {
          num: "1.0000000000e+00"
          time: "06:54:00 PM"
          value: "6.5880000000e+04"
        }
        …
        …
        …
      ]
    }
  ]
}
```


Figure 26: Result from the First Example of Node Controller Resources URL (MEM Category)

The second example URL asks for the system load of the node-id "nc1." The `step` is set to be 90, so the result will show the resource data in every 90 seconds. The `start-time` and `end-time` is also set. A time in seconds since epoch (1970-01-01) format is required. The time interval is 1,000 seconds (1303334720 – 1303334720 = 1,000 seconds). Thus, the number of records returned by the web-server is about 12 records (1,000 / 90 ≈ 12). The sample result from this URL is shown in Figure 27.

```
{
   result: [
      {
         type: "load"
         sub-type: "load_one"
         count: 12
         data: [
            {
               num: "1.0000000000e+00"
               time: "02:09:00 PM"
               value: "4.3111111111e-01"
            }
            {
               num: "1.0000000000e+00"
               time: "02:10:30 PM"
               value: "1.8833333333e+00"
            }
            {
               num: "1.0000000000e+00"
               time: "02:12:00 PM"
               value: "4.4528888889e+00"
            }
            {
               num: "1.0000000000e+00"
               time: "02:13:30 PM"
               value: "3.1905555556e+00"
            }
            {
               num: "1.0000000000e+00"
               time: "02:15:00 PM"
               value: "1.2870000000e+00"
            }
            {
               num: "1.0000000000e+00"
               time: "02:16:30 PM"
               value: "6.0500000000e-01"
            }
            {
               num: "1.0000000000e+00"
               time: "02:18:00 PM"
               value: "7.7000000000e-01"
            }
            {
               num: "1.0000000000e+00"
               time: "02:19:30 PM"
               value: "9.3388888889e-01"
            }
            {
               num: "1.0000000000e+00"
               time: "02:21:00 PM"
               value: "7.8722222222e-01"
            }
            {
               num: "1.0000000000e+00"
               time: "02:22:30 PM"
               value: "6.0000000000e-01"
            }
            {
               num: "1.0000000000e+00"
               time: "02:24:00 PM"
               value: "4.7622222222e-01"
            }
            {
               num: "1.0000000000e+00"
               time: "02:25:30 PM"
               value: "3.8733333333e-01"
            }
         ]
         dsname: "nc1"
      }
```


(continue on the right figure)

(continue from the left figure)


```
   {
      type: "load"
      sub-type: "proc_run"
      count: 12
      data: [
         {
            num: "1.0000000000e+00"
            time: "02:09:00 PM"
            value: "0.0000000000e+00"
         }
         {
            num: "1.0000000000e+00"
            time: "02:10:30 PM"
            value: "0.0000000000e+00"
         }
         {
            num: "1.0000000000e+00"
            time: "02:12:00 PM"
            value: "0.0000000000e+00"
         }
         {
            num: "1.0000000000e+00"
            time: "02:13:30 PM"
            value: "7.0000000000e-01"
         }
         {
            num: "1.0000000000e+00"
            time: "02:15:00 PM"
            value: "1.8666666667e+00"
         }
         {
            num: "1.0000000000e+00"
            time: "02:16:30 PM"
            value: "8.8888888889e-02"
         }
         {
            num: "1.0000000000e+00"
            time: "02:18:00 PM"
            value: "8.7777777778e-01"
         }
         {
            num: "1.0000000000e+00"
            time: "02:19:30 PM"
            value: "1.6666666667e-01"
         }
         {
            num: "1.0000000000e+00"
            time: "02:21:00 PM"
            value: "1.0000000000e+00"
         }
         {
            num: "1.0000000000e+00"
            time: "02:22:30 PM"
            value: "6.1111111111e-01"
         }
         {
            num: "1.0000000000e+00"
            time: "02:24:00 PM"
            value: "5.4444444444e-01"
         }
         {
            num: "1.0000000000e+00"
            time: "02:25:30 PM"
            value: "3.5555555556e-01"
         }
      ]
   }
]
}
```


Figure 27: Result from the Second Example of Node Controller Resources URL (LOAD Category)

5.2 Publish/Subscribe API for Server Pushing Method

One of the most significant features of any monitoring system is its "real-time" support. To satisfy this characteristic, we implement an additional web-server in the *Hyracks Console Server (HCS)* to support a server-push mechanism that responds to vital Hyracks events. With this server, the system can avoid having clients periodically poll for data. This second server initiates communication between the *Hyracks Console Server (HCS)* and the *Hyracks Console Visualization (HCV)* and publishes messages to a particular "channel" whenever a vital event occurs. Any clients that subscribe/listen to that channel will receive the message and perform appropriate actions. This web-server is implemented on port number [`http-port + 1`]. (E.g., if the `http-port` setting is 2099, this web-server communicates over port 2100.)

The key events in the Hyracks system are handled by messages that flow between the *Hyracks Cluster Controller (CC)* and one or more of the *Hyracks Node Controllers (NCs)*. For example, when a stage is ready to run, the CC sends messages to the set of NCs participating in that stage's execution in order to start task activations, and the CC is informed of the failure or success of a stage by the NCs. The significant Hyracks events can be classified into three main categories; therefore, we create three channels responding to each category: (1) `/jobs`, (2) `/stages/<job-id>` and (3) `/attempts/<job-id>` channel.

As we mentioned in the system architecture section, the *CometD* service is employed to support server-push communications between clients and servers via the *Bayeux* protocol. To implement this service in the Jetty server, `CometdServlet` and `JobStageServlet` objects are added to the Jetty servlet container and initiated when the server starts. The first object is responsible for the CometD and Bayeux configuration, and the second object is responsible for invoking CometD services in the `JobStageService` class. This class extends the CometD class `org.cometd.server.AbstractService`, which specifies the Bayeux channels the service is interested in.

When an important event in the Hyracks system occurs, depending on what type of event it is, one of these three methods - `pushJobs()`, `pushStages()` or `pushAttempts()`- is invoked. Each method requests `JobStage` Servlet URL along with different parameters and values. The servlet then generates the result/message in the JSON format and invokes

`JobStageService`, which then publishes the message on the particular channel. This process can be summarized as shown in Figure 28.

JobStage Servlet URL Pattern

```
"http://<host>:<port>/jobstage?name=jobs";

"http://<host>:<port>/jobstage?name=attempts&jobid=<jobId>";

"http://<host>:<port>/jobstage?name=stages&jobid=<jobId>&attempt=<attempt>";
```

Figure 28: Server-Push Mechanism

5.2.1 `/jobs` channel:

A message is published to this channel whenever there is an event related to Hyracks Jobs, for example: `JobCreateEvent`, `JobStartEvent`, `JobAbortEvent`, `JobAttemptStartEvent`, and `JobCleanupEvent`.

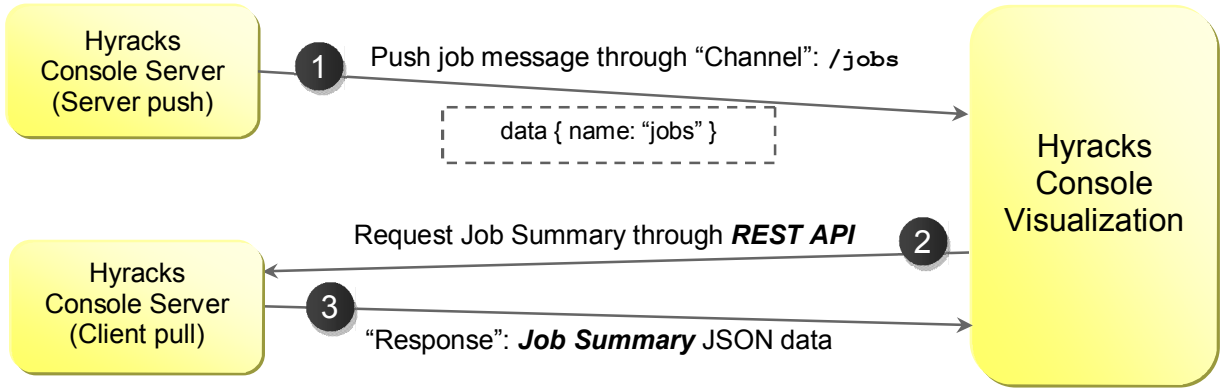


Figure 29: Example of /jobs Channel

Figure 29 shows an example action when a new job is submitted to the system. The HCS (supporting server-push) calls the `pushJobs()` method to publish a message to the `/jobs` channel. The HCV, which subscribes to this channel, receives the message and reloads a `JobBrowser` page by sending a new request to HCS (supporting client-pull). The latter server processes the client request and responds back with the summary of jobs in the JSON format. In this case, the push mechanism is used to trigger a client pull action.

5.2.2 `/stages/<job-id>` channel:

When an event related to some change in the states of a job stage occurs, a message, including the current status of each stage, is published to this channel. Some examples of events related to job stages are: `ScheduleRunnableStagesEvent`, `StageletFailureEvent`, `JobAttemptStartEvent`, `JobAbortEvent`, and `JobCleanupEvent`.

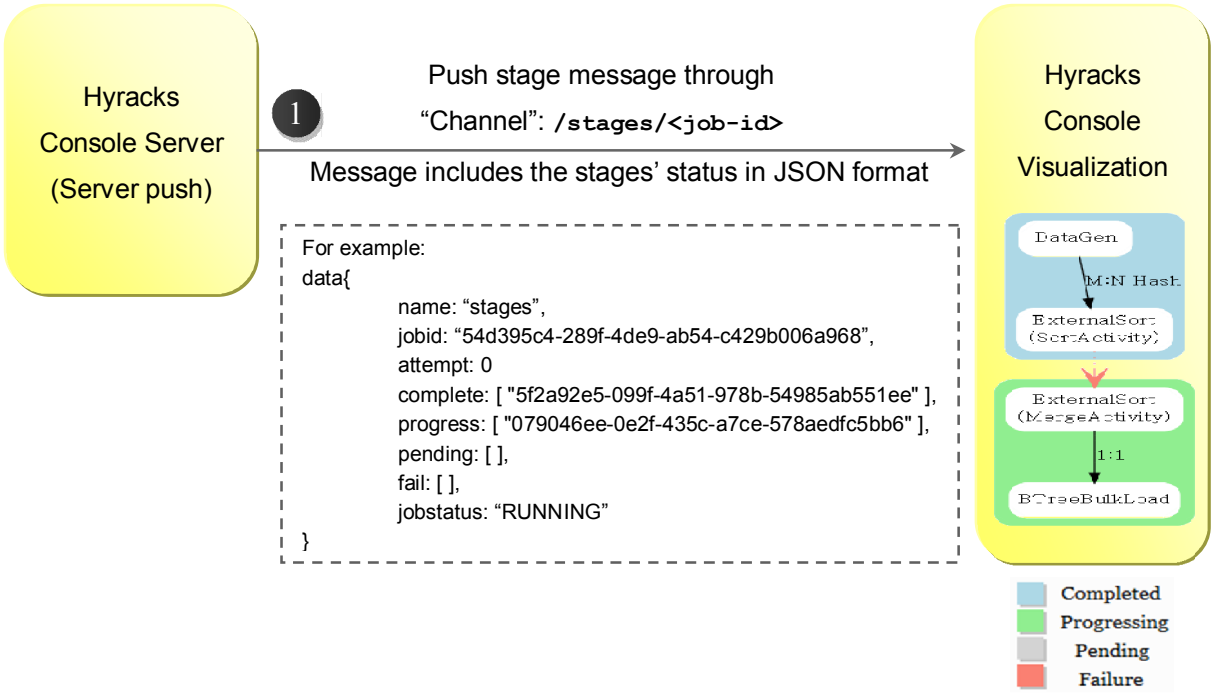


Figure 30: Example of /stages/<job-id> channel

Figure 30 depicts an action when the stage of job-id "`54d395c4-289f-4de9-ab54-c429b006a968`" is changed. The web-server calls the `pushStage()` method to publish the

message to the `/stages/54d395c4-289f-4de9-ab54-c429b006a968` channel. Then, the HCV that listens to the channel receives and passes the message to the JavaScript function in order to assign the color of each stage in the *Job Activity Node Graph (Job Plan)* diagram. The message includes several entities, e.g., a job's ID, attempt number, list of completed stages' ID, list of progressing stages' ID, list of pending stages' ID, list of failed stages' ID, and job's status. The color notation of completed, progressing, pending and failed stage is light blue, light green, gray, and salmon respectively. Note that this action, unlike the previous one, doesn't request other resources. This is because all needed information is already included in the message.

5.2.3 `/attempts/<job-id>` channel:

When 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. This channel is used when a specific job begins a new attempt for its execution (`JobAttemptStartEvent`). The HCS calls the `pushAttempts()` method to publish a message to the `/attempts/<job-id>` channel. When the HCV receives the message, it reloads a `JobProfile` page as shown in Figure 31.

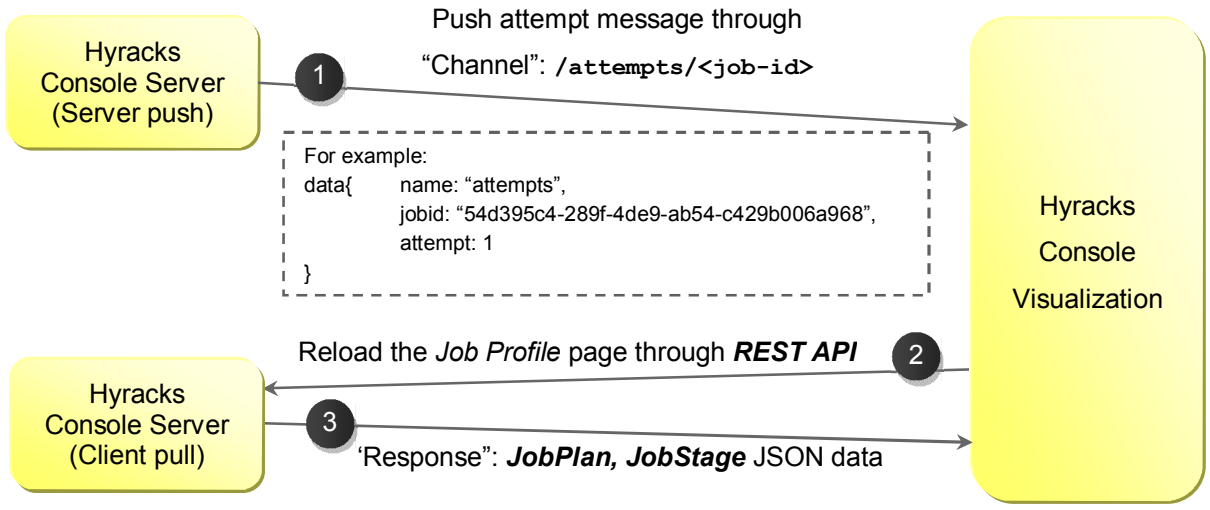


Figure 31: Example of /attempts/<job-id> channel

5.3 Sample Web UI

Even though data in the JSON format is fairly easy to read and comprehend, typical users prefer information represented visually. In addition, when such data becomes large, it is almost impossible for the users to gain knowledge by searching through thousands lines of text. In this section, we provide some samples of the web interface of our Hyracks Console Visualization (HCV) system. The entire sitemap of the HCV is presented in Figure 32. There are four main pages: *Hyracks Cluster Browser*, *Hyracks Node Controller Profile*, *Hyracks Job Browser* and *Hyracks Job Profile*. 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. In addition, 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 presented on each web page. A full description of the web user interface along with the user manual of the web UI can be found in [19].

Table 1: Hyracks Web Pages Description

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

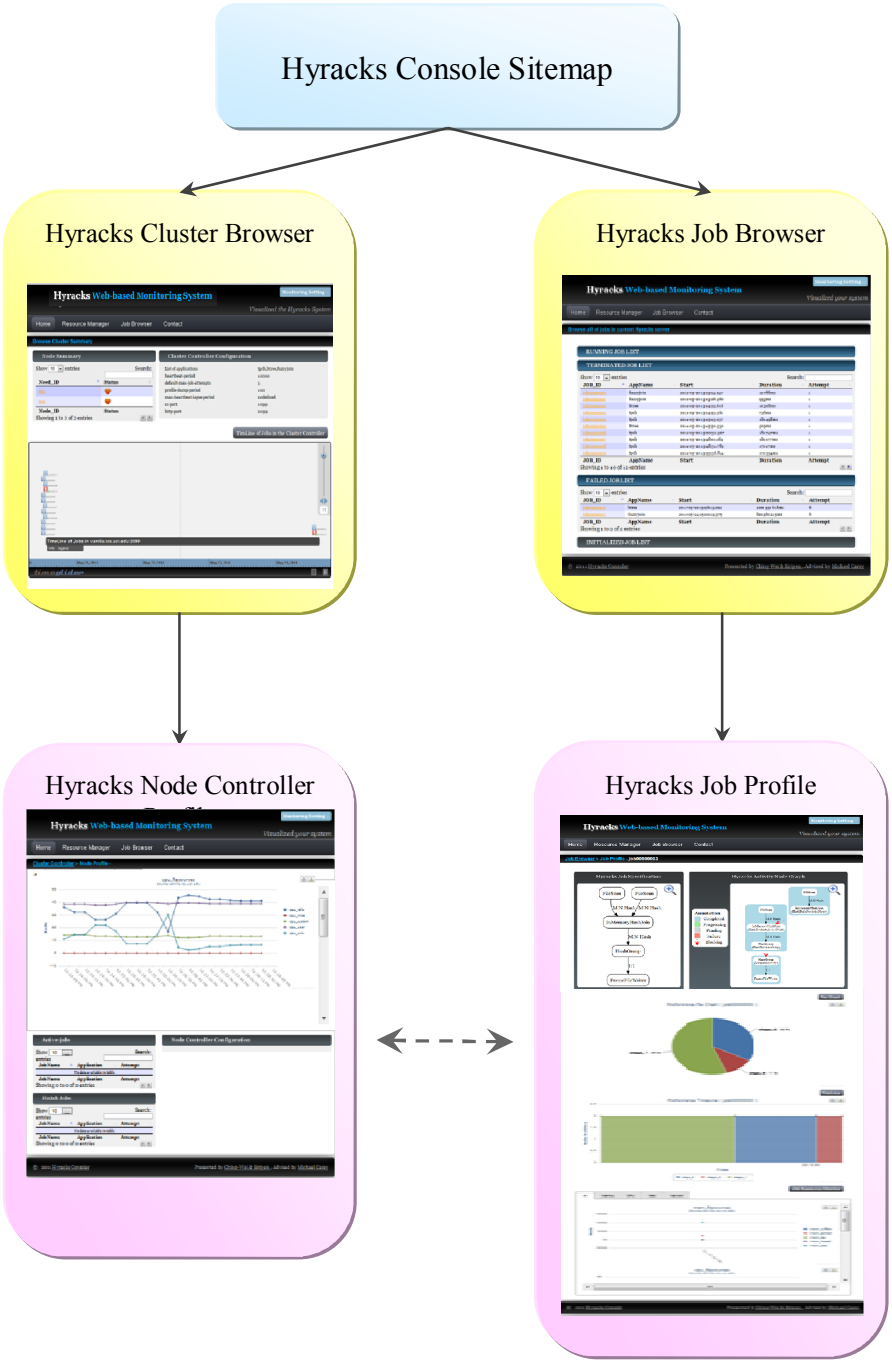


Figure 32: Hyracks Console Visualization Sitemap

```
{
result: [
   {
      id: "a965fe5a-ba0b-4a01-a13d-
      1b4dcc57adc1"
      application: "tpch"
      display-name: "job00000003"
      status: "TERMINATED"
      events: [
         {
            status: "RUNNING"
            system-time: 1305922493368
            date: "2011-05-20
            13:14:53.368"
         }
         {
            status: "INITIALIZED"
            system-time: 1305922493361
            date: "2011-05-20
            13:14:53.361"
         }
         {
            status: "TERMINATED"
            system-time: 1305922494104
            date: "2011-05-20
            13:14:54.104"
         }
      ]
      attempts: 1
      type: "job-summary"
   }
   {
      id: "54d395c4-289f-4de9-ab54-
      c429b006a968"
      application: "btree"
      display-name: "job00000010"
      status: "FAILURE"
      events: [
         {
            status: "RUNNING"
            system-time: 1305924975041
            date: "2011-05-20
            13:56:15.041"
         }
         {
            status: "FAILURE"
            system-time: 1305925628659
            date: "2011-05-20
            14:07:08.659"
         }
         {
            status: "INITIALIZED"
            system-time: 1305924975021
            date: "2011-05-20
            13:56:15.021"
         }
      ]
      attempts: 6
      type: "job-summary"
   }
   ...
   ...
   ...
]
}
```

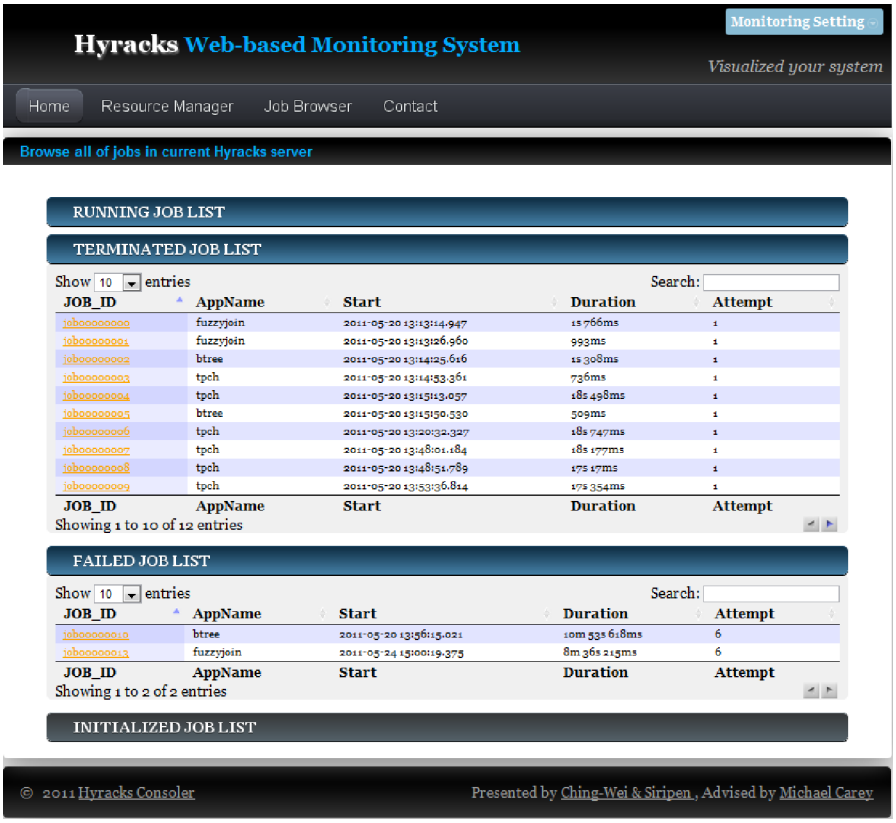



Figure 33: Job Summary in JSON format (left) and Job Browser Screenshot (right)

Figure 33 compares data presentation methods between the JSON format and the visualization web interface. On the left is a (partial) summary of jobs presented in the JSON format, and on the right is a corresponding screenshot of "Job Browser" page. This page requests the "Jobs-Summary" URL and presents the returned data in the table using the *DataTables* jQuery tool [18], which allows users to search and sort within the contents of the table. The data is categorized into four tables based on the job's status (i.e., `initialized`, `running`, `completed`, or `failed`). To handle the scalability issue, users can limit the number of jobs presented by putting a keyword on the search box. In addition, we include the *CometD Client JavaScript* in this page in order to subscribe the client to the "`/jobs`" channel. Whenever events

related to the Hyracks cluster's jobs occur (e.g., a new job is submitted to the cluster), the Hyracks Console Server will push a message to the "`/jobs`" channel. Users thus see all changes in real-time with low delay.

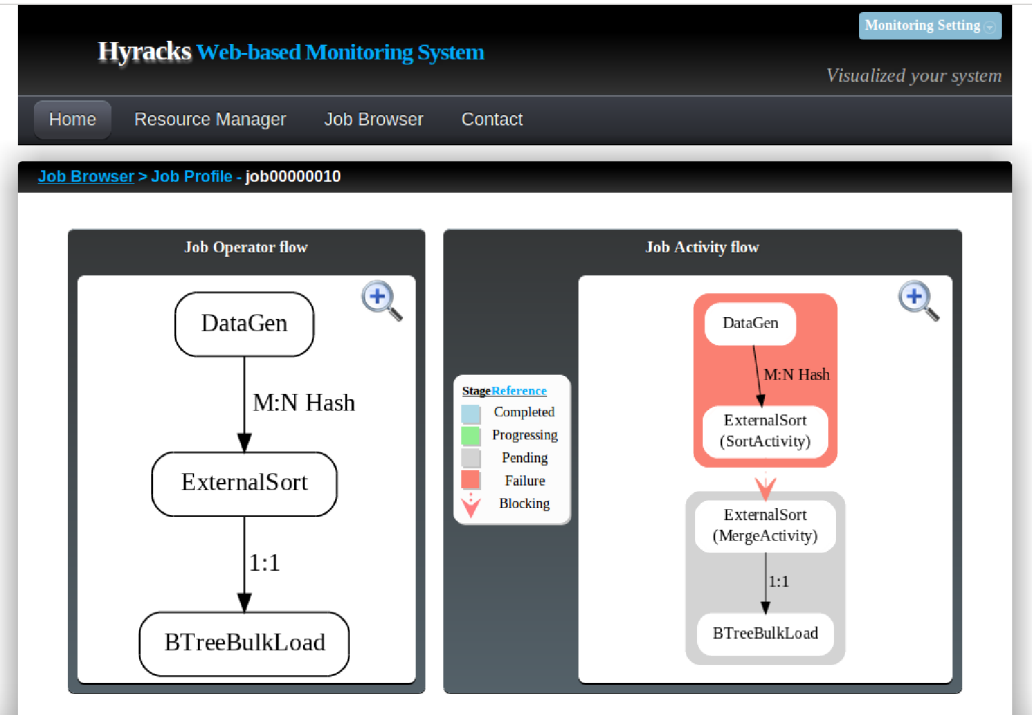


Figure 34: the Screenshot of One Part of Job Profile Page (bottom)

Figure 34 is the screenshot of the one part of the "Job Profile" page that presents a Hyracks Job Specification (called JobSpec) and the corresponding Hyracks Activity Node Graph (called JobPlan). These diagram representations are generated by the Graphviz tool [16]. From the Hyracks Job Specification of the "job00000010", there are three HODs assembled to perform one job: DataGen, ExternalSort, and BTreeBulkLoad. The first two HODs are connected by an M:N Hash Partition connector, and the last two HODs are connected by a 1:1 connector. The ExternalSort HOD in the job's specification is expanded into two activities – SortActivity and MergeActivity. The MergeActivity cannot be started until the SortActivity is completed, so there is a blocking edge (red arrow) between these two activities. This blocking edge is also used to divide a job into several job stages. As shown in the figure, the first stage consists of two HANs: the DataGen and SortActivity HANs. The second stage consists of the MergeActivity and BTreeBulkLoad HANs. The color in the diagram represents the current status of each job's

stage. E.g., a light blue color presents finished stage, a light green color presents running stage, a gray color presents pending stage, and a salmon color presents failed stage.

6 CONCLUSION AND FUTURE THOUGHTS

To support large-scale data processing and analysis, the *Hyracks* partitioned-parallel platform is being developed at UCI. When users execute a job on Hyracks, the internal processes between the input state and the output state can seem like a "black box" to users, in that they are not able to see the intermediate steps or individual processes. When an error occurs in their program, or its performance is not as expected, users need to understand and analyze the execution process of their jobs. To accelerate the development process of data-intensive applications and to provide better insight into the runtime state of the Hyracks platform, the *Hyracks Console* has been developed. This thesis has described the design and server-side implementation of the Hyracks Console.

The Hyracks Console has been designed to assist both *Hyracks End Users* and *Hyracks Operator Implementers* in monitoring the Hyracks system. The system architecture is conceptually divided into two main components, the *Hyracks Console Server (HCS)* and the *Hyracks Console Visualization (HCV)*. The HCS is responsible for collecting and delivering useful information related to the current state of the Hyracks jobs and the Hyracks clusters. To produce the precise, necessary data, clients must send requests to the HCS, and the HCS then responds with data related to the request, mostly in the JSON format. This technique is called a *client-pull* communication between client and server. In addition, a *server-push* mechanism is implemented to support real-time monitoring. When significant events occur in the Hyracks system, the HCS initiates the communication and sends a message to the clients without requests from them. The second component, the *Hyracks Console Visualization* (HCV), converts and presents plain-text data into a visual representation that users can easily and quickly understand. Several representation tools available to the HCV have been discussed in this paper. The HCV component also allows users to interact freely with the Hyracks Console system. The resulting Hyracks Console should greatly aid existing users in their understanding of the execution of their Hyracks jobs as well as helping new users overcome the steep learning curve of programming on large clusters.

There are still a number of ideas and future features that could further increase the functionality of the Hyracks Console. It would be nice for programmers if they could deploy their applications and execute their jobs via the Hyracks Console interface instead of the command line. It would also be beneficial to display more information at the runtime-task level to provide better insight about data movement at runtime. The console should also display an error log of job failures, and it should show the input/output locations of results of successful jobs on the Hyracks Console screen. The Hyracks Console must continue grow in parallel with the ongoing evolution of the Hyracks platform.

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] Hive. http://hadoop.apache.org/hive.

[6] 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.

[7] 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.

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

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

[10] 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 Analysis (CCA 2008)*, Hagen, Germany, August 2008.

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

[12] 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 Database*, *Vol. 29,* pages 185-216, 2011.

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

[14] 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.

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

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

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

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

[19] C. Huang. Hyracks Console Visualization. Master thesis, Department of Computer Science, UC Irvine (in preparation), July 2011.

[20] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull. GraphViz and Dynagraph: static and dynamic graph drawing tools. In M. Junger and P. Mutzel, editors, *Graph Drawing Software*, pages 127-148. Springer-Verlag, 2003.

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

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

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

[24] C. Olston et al. Nova: Continuous Pig/Hadoop Workflows. In *ACM SIGMOD 2011 International Conference on Management of Data (Industrial Track)*, Athens, Greece, June 2011.

[25] V. Jagannath, Z. Yin and M. Budiu. Monitoring and Debugging DryadLINQ Applications with Daphne. In *International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*, Anchorage, AK, May 2011

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

[27] 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

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

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

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

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

[32] JHeatChart. http://www.javaheatmap.com.

[33] D. J. Dewitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. GAMMA: a high performance dataflow database machine. In *Proceedings of the Very Large Data Bases (VLDB) 1986*, pages 228-237, Kyoto, Japan, August 1986.

[34] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *ACM SIGMOD 2008 International Conference on Management of Data (Industrial Track)*, Vancouver, Canada, June 2008.

APPENDIX

In this section, we provide summary information about Cluster Controller Configuration, Node Controller Configuration, and the URL-Request paths.

Table A: Cluster Controller Configuration

Variable	Default Value	Usage
-port	1099	Sets the port to listen for connections from node controllers.
-http-port	2099	Sets the http port for the Cluster Controller.
-heartbeat-period	10000	Sets the time duration between two heartbeats from each node controller in milliseconds.
-max-heartbeat-lapse-periods	5	Sets the maximum number of missed heartbeats before a node is marked as dead.
-profile-dump-period	0	Sets the time duration between two profile dumps from each node controller in milliseconds. 0 to disable.
-default-max-job-attempts	5	Sets the default number of job attempts allowed if not specified in the job specification.

Table B: Node Controller Configuration

Variable	Default Value	Usage
-cc-host	-	Cluster Controller host name
-cc-port	1099	Cluster Controller port
-node-id	-	Logical name of node controller unique within the cluster
-data-ip-address	-	IP Address to bind data listener
-frame-size	32768	Frame Size to use for data communication
-iodevices	java.io.tmpdir	Comma separated list of IO Device mount points
-dcache-client-servers	localhost:54583	Sets the list of DCache servers in the format host1:port1,host2:port2,...
-dcache-client-server-local	-	Sets the local DCache server, if one is available, in the format host:port
-dcache-client-path	/tmp/dcache-client	Sets the path to store the files retrieved from the DCache server

Table C: Summary of URL-Request Path

URL-Request Path	Usage
/state/jobs	Get a list of Hyracks jobs that have been submitted to a Hyracks cluster
/state/jobs/<job-id>/spec	Request for a particular Hyracks job's specification
/state/jobs/<job-id>/plan	Request for a particular Hyracks job's plan (or an activity node graph)
/state/jobs/<job-id>/<attempts>/stage	Get a progress of a Hyracks job in stage by stage basis
/state/jobs/<job-id>/<attempts>/profile	Get a profile counter of a particular Hyracks job
/profile/<job-id>/<attempts>/<connector-id>/(number)	Get a 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 of dataflow on the image
/console/cluster	Request for a Cluster Controller Configuration
/console/nodes/<node-id>/config	Request for a Node Controller Configuration
/console/nodes	Get a list of all nodes associated with the interested Cluster Controller
/console/nodes/<node-id>/jobs	Get a list of Hyracks jobs on a given node
/console/nodes/<node-id>/resources/<type>/(<step>)/(<start-time>)/(<end-time>)	Request for physical resource consumption at a given Node Controller

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