End-to-End Experiment Management in HPC

***Abstract*—Experiment management requires a constant feedback loop cycling through planning, execution, measurement, and analysis. Experiment management in High Performance Computing (HPC) follows this general pattern with three additional characteristics. One, HPC applications must deal with frequent platform failures which can interrupt, perturb, or terminate experiments. Two, these applications typically use MPI. Three, a scheduling system typically acts as a gatekeeper. This paper introduces HPC Experiment Management (HExMan), an experimental management framework for HPC which simplifies all four phases of experiment management. Planning is simplified by allowing the user to merely describe their experimental goals instead of constructing parameters for each individual task. To simplify execution, HExMan dispatches the tasks itself thereby freeing the user from remembering the often arcane methods for interacting with various scheduling systems. HExMan provides transducers that automatically measure and record important information; these can be extended to collect measurements specific to each experiment. Finally, analysis is simplified by providing DBViz, a visualization tool enabling interactive data exploration.**

I. Introduction

Experiment management in any domain is challenging. There is a perpetual feedback loop cycling through planning, execution, measurement, and analysis. The lifetime of a particular experiment can be limited to a single cycle although many require myriad more cycles before definite results can be obtained. Within each cycle, a large number of *subexperiments* may be executed in order to measure the effects of one or more independent variables.

Experiment management in high performance computing (HPC) follows this general pattern but also has three unique characteristics. One, computational science applications running on large supercomputers must deal with frequent platform failures which can interrupt, perturb, or terminate running experiments. Two, these applications typically integrate in parallel using MPI as their communication medium; each subexperiment within HPC is typically executed with an *mpirun* command. Three, there is typically a scheduling system (*e.g.* Condor, Moab, SGE, *etc.*) acting as a gatekeeper for the HPC resources.

In this paper, we introduce HPC Experiment Management (HExMan), an experimental management framework simplifying all four phases of experiment management. HExMan simplifies experiment planning by allowing the user to describe their experimental goals without having to fully construct the individual parameters for each task. To simplify execution, HExMan dispatches the subexperiments itself thereby freeing the user from remembering the often arcane methods for interacting with the various scheduling systems. By providing *transducers*, HExMan automatically measures and records important information about each subexperiment; these transducers can easily be extended to collect additional measurements specific to each experiment. Transducers are implemented as wrappers that execute each subexperiment and can be extended to parse the output of each subexperiment; the term is borrowed from a similar concept in Semantic File Systems [1]. In both HExMan and Semantic File Systems, transducers automatically generate semantically meaningful information in key-value pairs which can then be easily searched. Finally, experiment analysis is simplified by providing DBViz a general database visualization framework that allows users to quickly and easily interact with their measured data. DBViz provides two important advanced features for improving data analysis: *outlier analysis* and *hidden difference search*. A typical experiment workflow is shown in Figure 1a; Figure 1b shows the same workflow augmented with the various components of HExMan.

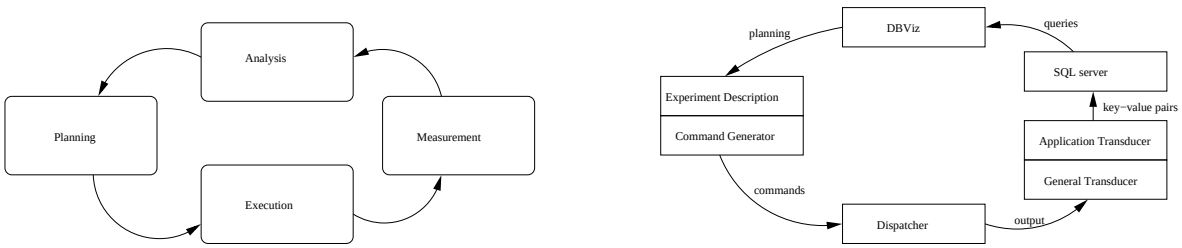
Although our experiment management framework has been designed for use at Los Alamos National Lab (LANL) and therefore generally for an HPC environment, we believe it can be easily modified to be suitable for experiment management in a Grid environment as well. Throughout this paper, we discuss how HExMan reduces the complexity of managing experiments that may span multiple supercomputers and the different scheduling systems on each. In terms of experiment management, supercomputers in HPC are very similar to clusters in the Grid. Although the Grid introduces additional complexity due to multiple administrative domains, we believe that HExMan can be useful in this environment as well but this is not a focus of this paper.

The remainder of the paper is organized as follows. We describe our design in Section II. In Section III, we offer a demonstration of our usage model. We provide a comparison to related work in Section IV, current status and future work in Section V, and a conclusion in Section VI.

II. HExMan Architecture

Simplifying the design, planning, execution, and analysis of experiments, HExMan is comprised of four main parts as shown in Figure 1b. A command generator reads a simple experiment description and produces a command to run each subexperiment. The subexperiments are then passed to the dispatcher who is responsible for dispatching each. As each subexperiment is executed, its output is parsed by *transducers* who insert parsed key-value pairs into a SQL server. The user then uses DBViz, which queries the SQL server, to visualize and analyze their results. Depending on this analysis, the user can modify their experiment description and repeat the cycle.

LANL Technical Release LA-UR 10-02150, April 28, 2010



(a) Typical Experiment Lifecycle

(b) HExMan Augmented Experiment Lifecycle

Fig. 1: **Experiment Lifecycle.** *The figure on the left depicts a typical experiment lifecycle progressing through planning, execution, measurement, and analysis. A user devises an experiment to study the effect of some number of independent variables, executes and measures each subexperiment, analyzes the results, and, depending on that analysis, can then revise the experiment and repeat the cycle. The figure on the right shows the workflow within HExMan. The user describes their parameter space to the command generator and specifies a path to an optional application transducer to capture application specific key-value pairs. HExMan adds a path to a general transducer to capture key-value pairs common to all experiments. The generated commands and their transducer(s) are then passed to the dispatcher which either submits the commands to a scheduling system or runs them synchronously in the foreground. As the commands run, key-value pairs are captured by the transducer(s) and inserted into a SQL server from which they can then be queried to visualize and analyze the results. Depending on this analysis the user can then modify their parameter space and repeat the cycle.*

A. Command Generation

HExMan provides a command generator engine which reads a description of the experiment, computes all subexperiments needed to complete the experiment, and produces a command to execute each subexperiment. For example, for an experiment wishing to study the effect of a set of independent variables, the user merely specifies an array of every target value for each independent variable. If multiple independent variables are specified, the command generator produces every possible combination of commands. The command generator assumes that each subexperiment will be executed with *mpirun*, although this can be over-ridden. The generated commands are then passed to the dispatcher who initially just lists the commands so that the user can verify that the commands have been correctly generated; once verified, the dispatcher can either submit the commands to the scheduling system or run them synchronously in the foreground.

The simplest approach is to write a text file with one mpirun command in each line; in this case, the command generator will pass each command directly to the dispatcher without modification. For more complicated parameter studies, HExMan provides a template where multiple list of parameters can be specified. Figure 2a shows an example of an experiment description file for the *mdtest* parallel metadata benchmark. This example shows how the optional *mpi_options*, *program_options*, *transducers*, and *transducer_arguments* are used. The command generator then creates all possible commands from the permutations of the lists of options and arguments as is shown in Figure 2b. The *–desc* argument is parsed by the transducer, is removed from the arguments passed to *mdtest*, and is used as a tag for the experiment to easily group all of the subexperiments for subsequent analysis.

Although the command generator can read from a simple list of mpirun commands, for any reasonable parameter sweep, a description file is a drastic simplification. Although this description file must be written in *Python*, the basic structure is already provided to the user in a template description file. So far two different users lacking any *Python* experience have used HExMan and were able to edit the provided template and start executing their experiments within an hour.

B. Dispatcher

HExMan provides an experiment manager program *run_expr.py*. To run an experiment, the user executes the experiment manager by passing it either a description file or a text list of commands. After invoking the command generator to obtain a list of commands from the file provided by the user, the experiment manager then passes these commands to the dispatcher. The experiment manager program takes several command line options which it passes to the dispatchersuch as the mode of dispatch, the number of times to run each command, and whether to randomly shuffle the commands before dispatching them. The dispatch mode specifies how the dispatcher should dispatch each command; by default it merely lists the generated commands so that the user can verify that the parameter study has been correctly configured in the description file. After verifying the list of commands, the user can ask the dispatcher (with command line options passed to the experiment manager) to run the jobs synchronously in the foreground or to submit them to a scheduling system.

Additional command line options can be used to pass additional information to the scheduling system. Currently, HExMan only supports the *moab* scheduler although it can easily be modified to support other schedulers as well; an earlier version of HExMan had support for both *moab* and *PBS*.

The dispatcher also pulls default parameters from a configuration file which is also written in Python. This configuration file provides different scheduling system parameters for different supercomputers. For example, at LANL we have populated our HExMan configuration file with information about the number of cores on each node in our different supercomputers. This information is then passed to the scheduling system which needs to know how many nodes

```
1 import sys
2 sys.path += '../lib'
3 import expr_mgmt
4
5 mpi_options = { "np"    : [ 4, 8 ], }
6 mpi_program = ( "/users/user1/mdtest/mdtest" )
7 program_options = {
8   "I" : [ 1, 10 ],
9   "d" : [ "/pfs/user1/mdtest.out" ],
10  "z" : [ 1, 2 ],
11  "b" : [ 1 ],
12 }
13 transducer = "/users/user1/mdtest/trans.py"
14 transducer_arguments = [
15   [ "--desc expr1" ]
16 ]
17
18 def get_commands():
19   return expr_mgmt.get_commands(
20     mpi_options=mpi_options,
21     mpi_program=mpi_program,
22     transducer=transducer,
23     transducer_arguments=transducer_arguments,
24     program_options=program_options )
```

(a) Experiment Description

```
1 /users/user1/mdtest/wrapper.py mpirun -np 2 \
2   /users/user1/mdtest/mdtest -I 1 -z 1 \
3   -b 2 -d /pfs/user1/mdtest.out --desc expr1
4 /users/user1/mdtest/wrapper.py mpirun -np 2 \
5   /users/user1/mdtest/mdtest -I 1 -z 2 \
6   -b 2 -d /pfs/user1/mdtest.out --desc expr1
7 /users/user1/mdtest/wrapper.py mpirun -np 2 \
8   /users/user1/mdtest/mdtest -I 10 -z 1 \
9   -b 2 -d /pfs/user1/mdtest.out --desc expr1
10 /users/user1/mdtest/wrapper.py mpirun -np 2 \
11   /users/user1/mdtest/mdtest -I 10 -z 2 \
12   -b 2 -d /pfs/user1/mdtest.out --desc expr1
13 /users/user1/mdtest/wrapper.py mpirun -np 4 \
14   /users/user1/mdtest/mdtest -I 1 -z 1 \
15   -b 2 -d /pfs/user1/mdtest.out --desc expr1
16 /users/user1/mdtest/wrapper.py mpirun -np 4 \
17   /users/user1/mdtest/mdtest -I 1 -z 2 \
18   -b 2 -d /pfs/user1/mdtest.out --desc expr1
19 /users/user1/mdtest/wrapper.py mpirun -np 4 \
20   /users/user1/mdtest/mdtest -I 10 -z 1 \
21   -b 2 -d /pfs/user1/mdtest.out --desc expr1
22 /users/user1/mdtest/wrapper.py mpirun -np 4 \
23   /users/user1/mdtest/mdtest -I 10 -z 2 \
24   -b 2 -d /pfs/user1/mdtest.out --desc expr1
```

(b) Generated Commands

Fig. 2: **Command Generation.** *On the left is a simple experiment description for the parallel metadata benchmark* mdtest*. Lists are specified the* -l *and the* -z *options whereas just a single value is specified for the* -d *and* -b *options. The options list for* mpi *specifies that each generated command should be run at different sizes (passed to* mpirun *with the* -np *option. This example also shows how transducers can be specified to the command generator. The right side shows the commands generated from this experiment description. Note that the user can either create an experiment description file as in the left column or can manually create a list of commands like those on the right side. The command generator will determine whether it needs to automatically generate commands from a description file or whether it merely reads the commands directly. Once the command generator either generates the commands or reads them in, it then passes them to the dispatcher.*

to allocate for different numbers of requested processes (*e.g.* to dispatch a 128 processor job on the LANL RRZ supercomputer, the dispatcher reads the configuration file to discover that RRZ has eight cores per node and therefore requests sixteen nodes from the scheduling system). We also include information about the default filesystem for each supercomputer; although the naming conventions on each supercomputer are mostly consistent, some supercomputers do have different paths to their parallel storage systems. By specifying a default value for each supercomputer, we can use a single experiment description file across multiple supercomputers without modifying it specifically for each.

Some of the functionality of the dispatcher might seem redundant since users can access scheduling systems directly; however, we believe that this redundancy is actually one contribution of HExMan in that it abstracts the scheduling system away. Users create an experiment description file, and a configuration file, and can then dispatch commands on any scheduling system on any supercomputer without having to modify any of their scripts since HExMan transparently interacts with the scheduling system for them.

C. Transducers

As command generation and dispatch assist the planning and execution phases of an experiment life-cycle, so do the HExMan *transducers* assist in the measurement phase. The term transducers is borrowed from Semantic File Systems (SFS) [1]. Just as transducers in SFS create searchable semantic information about files, transducers in HExMan create searchable semantic information about each subexperiment. HExMan provides a general transducer which collects information about every subexperiment; this transducer can be augmented by the user to collect information specific to the application. As is shown in Figure 2b, the transducer parses the output of each subexperiment, records the command line of each, collects additional information about the execution environment, and then inserts all of this collected information as key-value pairs into a SQL server. Some of the key-value pairs collected from each subexperiment are as follows:

- **application:** Name of the application
- **commandline:** Command line used
- **description:** Optional experiment tag
- **epoch:** UNIX epoch at time of execution
- **hostlist:** A list of each compute node
- **numhosts:** Number of compute nodes used
- **retval:** Return value
- **system:** Name of the supercomputer
- **username:** Username of the submitting user
- **walltime:** Total runtime

The provided transducer can be easily extended by users comfortable with regular expressions who can parse the application output to gather additional key-value pairs specific to their applications. For example, our transducer for *mdtest* requires only 200 lines to parse the output and collect semantic information about the configuration and results for each subexperiment.

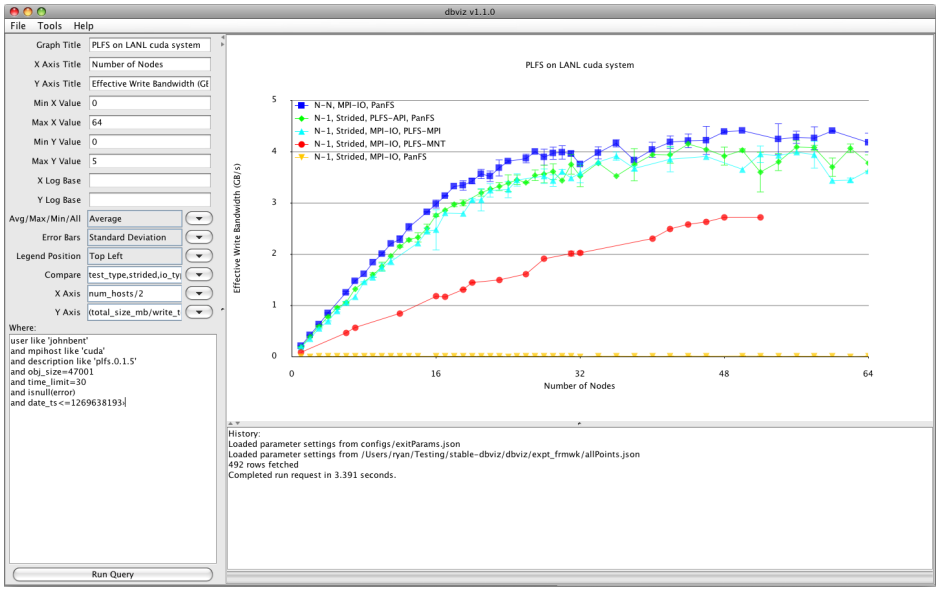


Fig. 3: **Interactive Data Exploration.** *This screenshot of DBViz shows how a graph is generated from a specified filter, x-axis, y-axis and a set of columns to compare.*

D. Visualization and Analysis

To enable interactive data exploration, we have developed DBViz, a data analysis tool written in *Java Swing*. This tool pulls data from any *SQL* database and plots it according to the user's specifications. The resulting graph is interactive which allows the user to more closely analyze the data. After ascertaining the table to be queried, DBViz fetches the schema and uses it to populate pull-down menus which are used to specify which fields to display on the axes and which fields to use to create comparisons (*i.e.* different lines in the resulting graph). This eliminates the need for the user to remember the exact schema. Although the user can use the pull-down menus as a shortcut, the user can also request that more complicated operations be performed. For example, an axis can be just a single field or it can be a function of multiple fields and constants; this could be used for instance to create a bandwidth by dividing a size field by a time field. When the user does create more complicated functions for the axes or the compare fields, these are remembered persistently by DBViz and are added to the pull-down menus so that they need be created only once.

After specifying the columns that should be displayed on the x-axis and the y-axis, a filter (*e.g.* an *SQL WHERE* clause) can also be specified. The filter allows the user to be more selective about the data that will be returned. By specifying a set of columns to be compared, different configurations can be compared. Currently, the available formats of the plots are linespoints, xy-scatter, and bar graphs as is illustrated in Figure 4. DBViz allows sessions to be saved so that interesting data explorations and interactive sessions can be interrupted and resumed. These saved sessions are stored in text files which allows them to be easily shared between users.

There are two important features in DBViz that make its interactivity so valuable. One is *outlier analysis*; the other is *hidden difference search*.

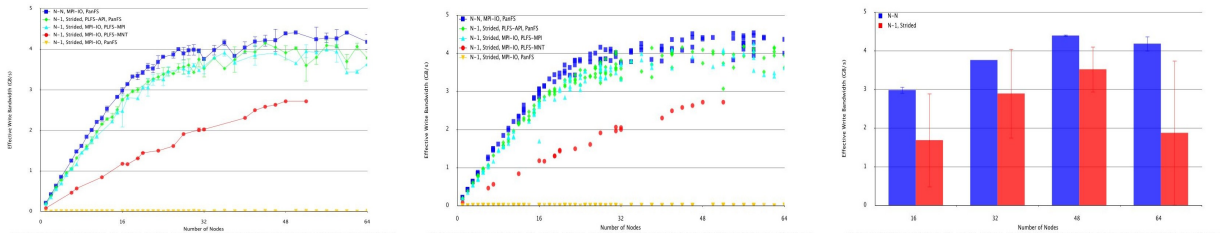
1) Outlier analysis: Outlier analysis is available in xy-scatter plots; the user can click on any point and DBViz will transparently query the database and present every key-value pair for that point thereby allowing the user to quickly discover the particular data for any subexperiment; we have found this to be of particular value for analyzing outliers.

2) Hidden difference search: Hidden difference search is another feature provided by DBViz to help the user refine their analysis. Hidden differences are caused by incompatible data being accidentally combined when the user specifies an imprecise filter or set of comparison fields. When the user suspects that the visualization lacks sufficient precision, hidden difference search can quickly identify suspect keys which need to be segregated or filtered to increase analytical precision.

Both outlier analysis and hidden difference search will be further illustrated by means of example in Section III.

E. Monitoring and Replay

By inserting executed commands into a *SQL* server, HExMan allows interactive data exploration as discussed above in Section II-D. In addition, HExMan provides two additional programs that utilize the *SQL* server. *replay_expr.py* takes a *SQL* filter (*i.e.* a WHERE statement), queries the database for all previously executed commands matching that filter, and passes them to the dispatcher. This is frequently utilized at our site to check differences in platform performance following a configuration change such as an upgrade to the operating system or changing the buffer size on a network switch. In such cases, the platform administrators will conduct a measurement, giving it a unique tag, before making the change; after the change, they merely



(a) Linespoints (b) XY-scatter (c) Bars

Fig. 4: **Visualization Flexibility.** *After HExMan populates a database with key-value pairs collected by transducers, DBViz allows interactive exploration of that data. These figures show typical visualizations where a filter has been applied to the database, and a visualization using the resulting rows is displayed. One line (or bar) is produced for each set of parameters that the user wishes to compare as a function of specified values for both axes. The axes values can be a single field or they can be arbitrarily complex combinations of multiple fields and constants (e.g. to show a bandwidth in GB/s, the user might specify that the y-axis should be a field dividing the number of bytes by a time and then dividing that again by a gigabyte). The figure on the left shows an aggregation of multiple points into mean values with standard deviations. In the middle is the same data in an xy-scatter plot; the scatter plot is useful because it allows the user to easily identify outlying data. The interactivity is vital here as a user can click on an outlying data point and DBViz will transparently query the database and display a table of all key-value pairs for that outlier. Finally, DBViz can produce bar-graphs as well as in the figure on the right.*

use *replay_expr.py* to rerun that same experiment. Following the replay, they can use DBViz to check for changes in performance. Although we use HExMan at our site primarily for platform measurement and observation, it is applicable to any HPC experiment.

A second tool that leverages the database is *check_expr.py* which takes both a *SQL* filter and an experiment description file. By using the command generator to generate the necessary subexperiments from the description file, *check_expr.py* can then determine which have not yet been dispatched by querying for each in the database. The uncompleted subexperiments can then optionally be passed to the dispatcher to resubmit them. Although this functionality can also be achieved by querying the scheduling system, doing so within HExMan allows the user a single abstraction for interacting with multiple different scheduling systems. Additionally, we frequently encounter situations where the scheduling systems' queues are purged by the administrators; in such cases, querying the database is the simplest way to check progress of the experiment. Finally, this approach drastically simplifies checking progress when the subexperiments have been spread across multiple computing sites, especially when each uses different scheduling systems.

III. Usage Model

To use HExMan, a user first creates an experiment description file. Because HExMan handles both command generation and command dispatch automatically, these description files need merely identify parameters and values of interest. An example of such a file is shown in Figure 2a. If the user desires more than the default set of key-value pairs captured by HExMan, they can additionally create an application-specific transducer. Table I lists the lines needed to create description files and transducers for several applications currently using our framework.

After defining descriptions and transducers, users can then begin to run their experiments with the dispatcher. The user can optionally ask the dispatcher to shuffle the subexperiments randomly before dispatching. The value of this randomness is shown in Figure 5. By utilizing randomness, users can more quickly see trends within their experiments than if the subexperiments are run ordered by the parameter sweeps. Note that utilizing randomness in this way is not a new result nor is it our contribution. Rather our contribution here is that the subexperiment tracking and monitoring provided by HExMan allow randomness to be exploited in this manner. Without such simple tracking and monitoring, it is much more difficult for the user to run subexperiments randomly; in the case of interrupted experiments, naive resubmission will rerun previously completed subexperiments and slow total time to completion for the experiment.

As the subexperiments begin to complete, the user can track their progress with the monitoring within HExMan, and can visualize the results with DBViz. The following describes an example interactive session with DBViz.

1) The dispatcher has already executed a set of subexperiments; each subexperiment was executed ten times to provide statistically meaningful results and variability measurements.
2) The entire experiment was tagged with a *description* of *expr1*.
3) The transducer has collected general key-value pairs and has parsed specific ones and has inserted them into a SQL server.
4) The user specifies "WHERE description like 'expr1'".
5) The user specifies the column *numhosts* as the x-axis and the column *walltime* as the y-axis.
6) The user asks to compare columns *depth* and *items* which were the parameters swept and which the transducer parsed from the command line.
7) A graph is created similar to that shown in Figure 3. The user notices unusually high variability and changes the display from lines to a scatter plot and

TEST	Description Lines	Keys Collected	Transducer Lines
mdtest	48	88	200
ior	45	101	143
LANL fs_test	36	149	2

TABLE I: **Defining Experiments.** *This table shows how many lines are necessary to describe typical parameter sweeps for three different applications which are currently using HExMan at LANL. It also shows the number of key-value pairs collected from each subexperiment, and the lines needed for the application-specific transducers which parse the output of each subexperiment and produce these key-value pairs. The LANL fs_test transducer is so small because LANL fs_test already produces output in key-value format so that transducer simply echoes its input to its output. The other transducers are larger because those applications do not have consistent output formatting so every key-value must be parsed differently.*

discovers a few outliers.

8) Using outlier analysis, the user pulls the complete row from the table for each outlier and notices that some of the outliers were from subexperiments that did not complete successfully. To remove them from the analysis, the user adds a successful *retval* condition to the WHERE clause.
9) The user returns to linespoints mode but the variability still looks higher than expected. The user queries the set of hidden differences and discovers that data from multiple supercomputers with drastically different performance characters has been combined. The user then either adds *system* to the comparison fields or refines the filter to examine subexperiments from only a single *system*.
10) The user discerns that performance has degraded from previous experiments. The user then combines multiple experiments and queries hidden differences again which reveals that *osversion* has changed. The user adds *osversion* to the comparison fields and identifies that as a reason for the performance degradation.

A screenshot of DBViz corresponding to step 7 above is shown in Figure 3. Notice that the key for the graph is constructed using each unique combination of the columns as specified in the *compare* field. By default, these keys would read as *k=v,k=v,...* but using regular expressions in the Preferences panel (not shown) allows the user to create semantically meaningful substitutions for key-value pairs. For example, in this screenshot, the user has created a substitution to replace *target=.*plfs.** with *PLFS-MNT*.

IV. Related Work

HExMan introduces a new way to manage experiments in a HPC environment, greatly simplifying the experimental process. HExMan submits and executes jobs, parses the results, inserts them into a database, and allows the user to graphically and interactively view the results. Similar efforts can be found, but none offer the holistic approach of HExMan nor its HPC focus.

GridBot [5] is a system for executing experiments across multiple grids and clusters. Its high-level goal is very similar to HExMan; both systems are designed to ease the burden of computational scientists to manage very large experiments with many subexperiments (subexperiments in GridBot are called *Bags of Tasks*). GridBot's focus however is on dispatch; its dispatch engine is designed for the Grid however where the dispatcher in HExMan is designed for an HPC environment. Focused on dispatch, GridBot lacks any analogues for transducers, DBViz, or the ability to monitor and replay experiments. Ultimately, the two systems are highly complementary: HExMan provides end-to-end experiment management for HPC; combining it with the grid-aware dispatching in GridBot would enable end-to-end experiment management for the Grid.

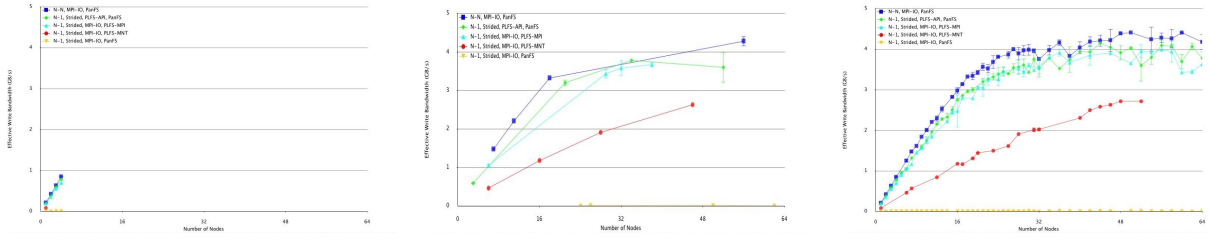
VGrADS [4] is a similar approach that provides a virtualized Grid execution system; like GridBot, it is geared for the Grid whereas HExMan is geared for HPC. VGrADS also attempts to provide reliable quality of service and fault tolerance using automated task replication. These features complement HExMan's more holistic approach.

Zoo [2] is another experiment management framework for desktop computing. Similar in concept to HExMan it acknowledges the experiment feedback loop and provides a framework for making the experimental process more manageable. Zoo has modules, such as Opossum and Frog, for each step of the process, from designing and setting up experiments through data collection and analysis. Although novel for its time, Zoo, and its focus on the desktop, provides insufficient management for the much more complex HPC environment. In addition, Zoo lacks the advanced features of DBViz, such as outlier analysis and hidden difference search.

GridDB [3] provides a data-centric overlay to process-centric grid middleware. The key differences between HExMan and GridDB exist in how they handle data processing and analysis. GridDB uses a declarative interface and type checking to deal with data processing and interactive query processing to support analysis. These techniques offer flexibility, but they also make the data more abstract. HExMan uses transducers to convert experimental results into usable data. Transducers are slightly less flexible, but they help to maintain the raw form of the data. DBViz offers easier data analysis and HExMan seeks to keep data processing and analysis more decoupled by using transducers and DBViz. HExMan provides a larger end-to-end focus whereas GridDB offers a greater emphasis on automated computational steering.

V. Current Status and Future Work

HExMan is used primarily by system integrators at Los Alamos National Lab to monitor several parallel storage sys-



(a) Ordered Dispatch (b) Random Dispatch (c) Final Results

Fig. 5: **Harnessing Randomness.** *These three graphs illustrate the value of randomizing the generated commands before dispatching them. The graphs on the left and the middle were created after only sixteen subexperiments were completed; the left-most is the graph produced when the commands are dispatched in the order of their generation whereas the middle graph was produced when the commands were randomly dispatched. Finally, the graph on the right shows the final results after all 492 subexperiments were completed. By providing a much better approximation of the final graph, randomized commands allow computational steering much earlier in the experiment lifecycle.*

tems on LANL supercomputers. It has been a valuable tool enabling much easier analysis of the impact of configuration changes and software upgrades on these systems. The ease by which experiments can be replayed allows significantly improved consistency of experiments enabling much cleaner *apples-to-apples* comparisons from before and after configuration changes. DBViz and the central SQL server simplify cross system comparisons that were previously extremely manually intensive. Additionally, HExMan is increasingly being adopted by other groups at LANL and has been used by several vendors to benchmark new systems.

We are working with one code team in particular at LANL that has current workflow inefficiencies in their regression testing that could be drastically improved using HExMan. They currently use an ad-hoc set of tools that has some of HExMan's functionality but it is not well consolidated nor organized and other functionality is entirely absent. This team runs nightly, weekly, and monthly tests where various metrics such as CPU utilization and I/O bandwidth are measured. The data is held in a database but the team lacks the capability to visualize the data interactively. When measured values fall outside of predetermined thresholds, the team is alerted to this but lacks an easy method to identify a root cause. Interactive sessions with DBViz would enable the team to quickly isolate, identify, and analyze the particular subexperiments that fell outside of the thresholds.

Their static visualization tools are very lacking in this regard; we have seen instances in which they have missed tests of interest because their predefined graphing utilities left some subexperiments literally off the graphs. Also, they have been plagued by an abundance of data which overwhelms their graphs with noise; DBViz can allow them to interactively filter and focus their analysis. Finally, even when they have ultimately identified root cause for performance anomalies, they lack utilities to rerun just those subexperiments. The replay ability of HExMan to apply a query to fetch a particular set of subexperiments to resubmit will drastically improve their workflow and hasten their analysis of performance anomalies.

DBViz is publicly available through SourceForge; the other components of HExMan will also be made publicly available pending the copyright process.

We are continuing to improve HExMan through the development of additional features. In the future, we aim to add a web-based interface to centralize experiment management and more easily allow multiple users to interact with collaborative experiments. Additionally, we plan to modularize the database integration to remove the SQL-only restriction.

VI. Conclusion

Experiment management is a cumbersome process. Users must plan, execute, monitor, and analyze multiple subexperiments. This process is often ad-hoc and each new experiment requires the development of new experiment management procedures. In computational science, and High Performance Computing in particular, developing a new experiment often requires that multiple new scripts are written: some to generate multiple subexperiments, some to execute them, and some to parse their outputs to enable analysis.

HExMan simplifies this process by providing a general and modular framework by which users can describe their experiments and HExMan will automatically generate subexperiments, execute them, and collect the resulting data. Additionally, HExMan provides a simple interface to augment the data collection and to monitor and replay experiments. A visualization component allows interactive data exploration greatly simplifying analysis and experiment tuning. Production use at Los Alamos National Labs has demonstrated the value of HExMan in enabling end-to-end experiment management in HPC.

References

[1] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. O'Toole. Semantic file systems. In *SOSP*, pages 16–25, 1991.

[2] Y. E. Ioannidis, M. Livny, A. Ailamaki, A. Narayanan, and A. Therber. Zoo: a desktop experiment management environment. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 580–583, New York, NY, USA, 1997. ACM.

[3] D. T. Liu and M. J. Franklin. Griddb: a data-centric overlay for scientific grids. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 600–611. VLDB Endowment, 2004.

[4] L. Ramakrishnan, C. Koelbel, Y.-S. Kee, R. Wolski, D. Nurmi, D. Gannon, G. Obertelli, A. YarKhan, A. Mandal, T. M. Huang, K. Thyagaraja, and D. Zagorodnov. Vgrads: enabling e-science workflows on grids and clouds with fault tolerance. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, New York, NY, USA, 2009. ACM.

[5] M. Silberstein, A. Sharov, D. Geiger, and A. Schuster. Gridbot: execution of bags of tasks in multiple grids. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, New York, NY, USA, 2009. ACM.