A Lightweight Workbench for Database Benchmarking, Experimentation, and Implementation

Xinyuan Zhao and Shashi K. Gadia

Abstract—We have developed a platform, called Cyclone Database Implementation Workbench (CyDIW), that can be used to implement new database prototypes, use existing command-based systems, and conduct experiments. The workbench allows seamless integration of multiple systems and provides useful services. To support database implementation page-based storage and buffer managers are built-in. A scripting language for batches of commands is included. Experiments are encapsulated as batches of commands on multiple systems. A simple and easy to use GUI is available that acts as an editor and a launch-pad for execution of batches of commands. Emphasis in CyDIW is on simplifying the logistics surrounding setting up experiments that are comprehensive and self-contained. The benchmarking services in CyDIW can be used for lightweight benchmarking, where a benchmark consisting of a dataset and a suite of commands is given. A benchmarking experiment collects performance statistics from multiple systems based on varying parameters and plots benchmarking results without leaving the GUI. Setup for the system is easy. All configuration settings are recorded in XML documents that are highly portable and readily visible. Once installed, batches representing experiments can be exchanged as text files and executed on CyDIW on any computer.

Index Terms—Benchmarking, Experimentation, Database Implementation.

1 Introduction

We are engaged in ongoing implementation of several database prototypes. Our implementation style dictates that these prototypes be organized as self-contained command based systems. In order to meet our needs in instruction in baccalaureate and graduate courses, research prototyping, database implementation, and project development and management, we have developed a centralized platform called Cyclone Database Implementation Workbench (CyDIW). In addition to our prototypes, any command based system, e.g. Oracle [1], can be easily interfaced, registered, and used as a client on CyDIW.

A low profile but versatile GUI (graphical user interface), shown in Figure 2, with almost no learning curve is also available. The GUI serves as an editor as well as a launchpad for execution of batches of commands that are realized as simple text files. A command in CyDIW consists of a client prefix, a command from the client system, some environmental information on how the command is to be handled by CyDIW platform in order to dispense appropriate services, and an ending semicolon. The command is sent to the client associated with the command prefix for parsing and execution. The output can be shown in the output pane, directed to files, or displayed in popup windows. The console displays the status and error messages for the commands.

In order to make development of a batch of commands a high level activity, we have designed an appropriate scripting language to make it easy for the users to realize complex experiments as simple batches of commands.

It is desirable that an experiment encapsulated by a batch can be repeated. We note that even though the scripting language includes conditional and loop structures, currently their use is to help compose self-contained experiments rather than ability to alter the sequence or the environment in which commands are executed. Thus the experiments currently undertaken in CyDIW can be viewed as linear. Ability to repeat non-linear experiments, e.g., executing certain commands in random order and under changing environments is considered beyond the scope of this paper. Our emphasis is on making linear experiments as versatile and self-contained as possible through logistical support within CyDIW. Here are a few logistical features. (See Figures 2, 3 and 5 for interesting examples of batches.)

- A batch of commands can involve multiple client systems of CyDIW. For example, a MySQL / SQL and Saxon / XQuery commands can be executed sequentially.
- Commands can be executed selectively or as a batch. This allows interactive refining, development, and fine tuning of commands in an experiment.
- As stated above, the outputs of commands can be displayed in the output pane, redirected to files, or displayed in

- *Xinyuan Zhao is a Ph.D. student in Computer Science at Iowa State University, Ames, IA 50011. Email: xinyuan@iastate.edu.*
- *Shashi K. Gadia is with the Computer Science Department, Iowa State University, Ames, IA 50011. Email: gadia@cs.iastate.edu.*

Manuscript received (insert date of submission if desired). Please note that all acknowledgments should be placed at the end of the paper, before the bibliography.

pop-up windows.

• The output pane is an html viewer by default, but it can be toggled to display plain text as well. For example, an XQuery query can compute a result as an html table. The html code or the intended display in an html browser can be viewed without having to get out of the GUI and use text editor or an html browser.

• A facility for creating variables is included. Commands can be parameterized with variables and such commands can themselves be stored in variables for later execution. Parameters are substituted in commands before they are executed. Thus a single command can be used for execution on multiple client systems and multiple inputs files.

• Services are offered by CyDIW via options added to client system commands.

• CyDIW is itself seamlessly treated as a client with many useful commands at user's disposal. For example a for-loop is treated as a CyDIW command. Another example is that an XML-based expression tree of a query can be viewed in a variety of ways.

• The underlying operating system can also be treated as a client to help execute appropriate commands without having to leave the GUI. The if-statement in the scripting language can be used to make behavior of a batch independent of the operating system.

• In order to take the mystery out of configuration settings, such information is recorded in XML files. Such files are highly portable and independent of operating systems and language platforms. The CyDIW internals must read from and only from these XML files in order to extract the required settings. Often such information is buried deep in implementation code making the system quite opaque and behavior difficult to replicate.

• XML-based configuration files can be modified easily for experimentation.

• Contents of files that effect the dynamic environment of a batch can be viewed in output pane or popup windows. An example is an expression tree after compilation of a query.

• XML-based logging and reporting of performance benchmark stats is available. The nesting of stats in log files automatically mimics the nesting of for loops where the stats are collected.

• An XML-based log may contain multiple benchmarks of interest. The desired benchmarks can be extracted and preprocessed by executing XQuery queries. For this, an XQuery engine can be used as a client within a batch. Readily built adapters are available for interfacing several XQuery engines with CyDIW.

• The highly popular open source R environment can be used for graphical reporting of benchmarks from within a batch. R receives an XML document containing a benchmark and creates graphical reports that can be stored in pdf files grouped into folders.

• In implementation of our own prototypes, variable can be used as handles to deal with queries step by step. A query can be parsed and its XML-based expression tree can be visualized. The expression tree can then be executed in iterator style. Commands can be used to open an iterator, get output one object at a time and close the iterator. This is useful in database implementation as the system behavior can be tested interactively without having to leave the GUI.

• Adapters for interfacing command-based systems with CyDIW are easy to develop. Once created, a system registration is made quite intuitive and transparent via an XML-based centralized system configuration file. Adapters handle sending commands to the parser of a client system, setting up a conduit for receiving results, and receiving performance stats.

• The organization of the workbench is modular. The installation of the workbench only requires copying it in a directory of one's choice.

• OS is by default available as a client. As stated above the open source R environment and XQuery engines are helpful in making experiments self-contained. An adapter for R and its pre-registry are included in CyDIW. A user can install R on their system and update the registration entry to reflect its location and enable it to activate R. Several XQuery engines are available as open source as well. In particular the open source version of Saxon XQuery engine is included in CyDIW. (We caution the reader that for benchmarking experiment shown in Figure 2, a commercial version of the XQuery engine provided by Saxon is used.)

• Two forms of comments are available. Single line comments start with "//" and must not contain any semicolons except the one that is required at the end. These comments can be used for documentation and instructions to a user, and are displayed verbatim in the output pane. More interestingly, they can also be used to suppress execution of commands. Yet, a single command at a time can be executed by selecting it while carefully avoiding the initial "//" characters. Another type of comments are those that are delimited by "/*" and "*/". These comments are not displayed in the output pane. Such comments can have multiple occurrences of semicolons and hence can be used to comment out sequences of commands.

• Experiments, encapsulated by batches, can be shared among users and be repeated.

The repeatability of experiments is very illusive concept and requires a cautious consideration. An experiment would be repeatable if the environment in which it is executed remains the same between repetitions and each command has the same incremental outcome on the environment. In today's complex computer systems with multiple cores, array of caches residing in the CPU, main memory, and the disk, and invisible resident processes, it is difficult to guarantee the same performance when a command is repeated. The lack of ability to repeat execution time can be addressed to some extent by including warm-up mechanisms in a batch of commands and then collecting performance stats over multiple executions. XQuery can be used within CyDIW to average results of such executions (See Figure 3).

As an implementation platform, CyDIW includes instrumentation to measure performance in terms of requests for page accesses – a customary practice in databases. Such a performance measure is more independent of the computer platform under use.

• CyDIW provides infrastructural support for database implementation. Using CyDIW a page-based storage can be created on a laptop or span multiple disks on a desktop. The command-centric approach in CyDIW can be followed to automate many aspects of experimentation, including gener-

ation of synthetic data or loading of real datasets (if they are available via commands in the client systems), running commands, collecting performance statistics, and plotting the final results at the click of a button.

The rest of the paper is organized as follows. In Section 2 we discuss existing database benchmarking and experimentation tools. Section 3 describes the architecture of the CyDIW platform. Section 4 describes the built-in commands in CyDIW. Storage and Buffer management services are discussed briefly in Section 5, which can be used in implementation of database prototypes in the workbench. Some use case examples are shown in Section 6. Section 7 provides some further discussion. The paper is concluded in Section 8.

2 Related Work

As numerous relational and XML database systems exist in industry and academia, benchmarking tools have been indispensable to help users make comparisons among different database systems.

Some benchmarking tools to help evaluate the performance of relational database engines are OSDL DBT suite [10], BenchmarkSQL [11], Quest Software's Benchmark Factory for Databases [12], and so on [14]. Most of these tools implement the industry standard TPC-C benchmarks [13] and support performance evaluations and comparisons for popular relational database systems. For example, BenchmarkSQL [11] is a benchmarking tool which closely resembles the TPC-C standard for OLTP. Like many other benchmarking tools, it runs SQL commands on a database system through JDBC [18] and measures the execution time and the throughput in a database system.

For relational databases, JDBC can be used to connect to a database and send commands as strings. For example, consider an insert statement that is to be used for building a benchmark dataset, one tuple at a time. Various segments of the insert statement can be computed using complex mechanisms such as nested loops, conditionals and functions implemented in Java. Using JDBC, benchmarking tools can invoke full power offered by general-purpose programming language.

XQJ [19] is a counter part of JDBC for processing queries on XML data and can be used to prepare and run heavier benchmarks for XQuery engines from Java platform. Some benchmarking tools [15], [16], [20] have been using XQJ API to provide benchmarking services to evaluate XQuery engines.

XCheck [15] is a platform that can automate some tasks in benchmarking XQuery engines. A user can configure an experiment by setting up the information regarding XQuery engines, queries and input data in some XML configuration files, and XCheck can complete the experiment automatically. BumbleBee [16] is a test harness for validating compliance of XQuery engines with the language specification. In addition, it can also be used for performance evaluation. XQBench [20] provides a web environment to submit XML documents and XQuery queries, run them on different XQuery implementations and get the results. However, none of the above tools provides a user the ability to design new experiments according to various specific requirements. Besides, these tools can only be used to test command-based XQuery engines.

With the help of the scripting language in CyDIW and the client registration mechanism, CyDIW supports designing different experiments on all kinds of client systems for different purposes.

3 The Architecture of CyDIW

The CyDIW platform consists of two main parts: CyDIW central services and the client systems. CyDIW central services include CyDIW user interface, CyDIW command parser, clients manager, clients interface and storage services. Each client system needs to provide a client adapter which will be used as a bridge between CyDIW platform and the client system. The details on how to implement a client adapter and how to register a client system will be discussed later in this section.

The overall architecture of the CyDIW platform is shown in Figure 1. As we can see, an experiment, represented in terms of a batch of commands is sent to the main parser of CyDIW. The parser processes each command and with the help of the clients manager delivers it either to the CyDIW execution engine or to a specific client execution engine according to the prefix of that command. Both CyDIW execution engine and client systems can access the variables in CyDIW. Client system can also interact with the log files through CyDIW client interface and provide their performance data to the log files.

3.1 Client Dependency Decoupling

One major concern in designing the CyDIW platform is how to make the platform independent of a client system, so that the platform can be used to run commands from different new client systems without any change on CyDIW's side.

In order to decouple the dependencies between CyDIW platform and the client systems, an XML configuration file named SystemConfig.xml is used to maintain the classpaths information for CyDIW and client systems. With the help of this information, the system class loader can load all the client systems at the start of CyDIW. After this, CyDIW clients manager will be initialized using the information found from SystemConfig.xml, and will keep track of a list of all the clients registered in the system configuration file. When CyDIW is executing a client command, it will search the clients manager by the prefix of the client. If such a client is found, the clients manager will return the client adapter corresponding to that client and CyDIW will use it for processing the client command and redirect its output accordingly.

Using this mechanism, the development and maintenance of CyDIW and client systems can be physically separated. CyDIW and client systems can be stored in different locations specified in the system configuration file. A client adapter class is used as a wrapper class to a client

system, and is registered with the prefix of the client in SystemConfig.xml. Therefore, the client adapter becomes the only bridge between a client and CyDIW platform. Section 3.3 talks about the implementation of the client adapters and the CyDIW client interface.

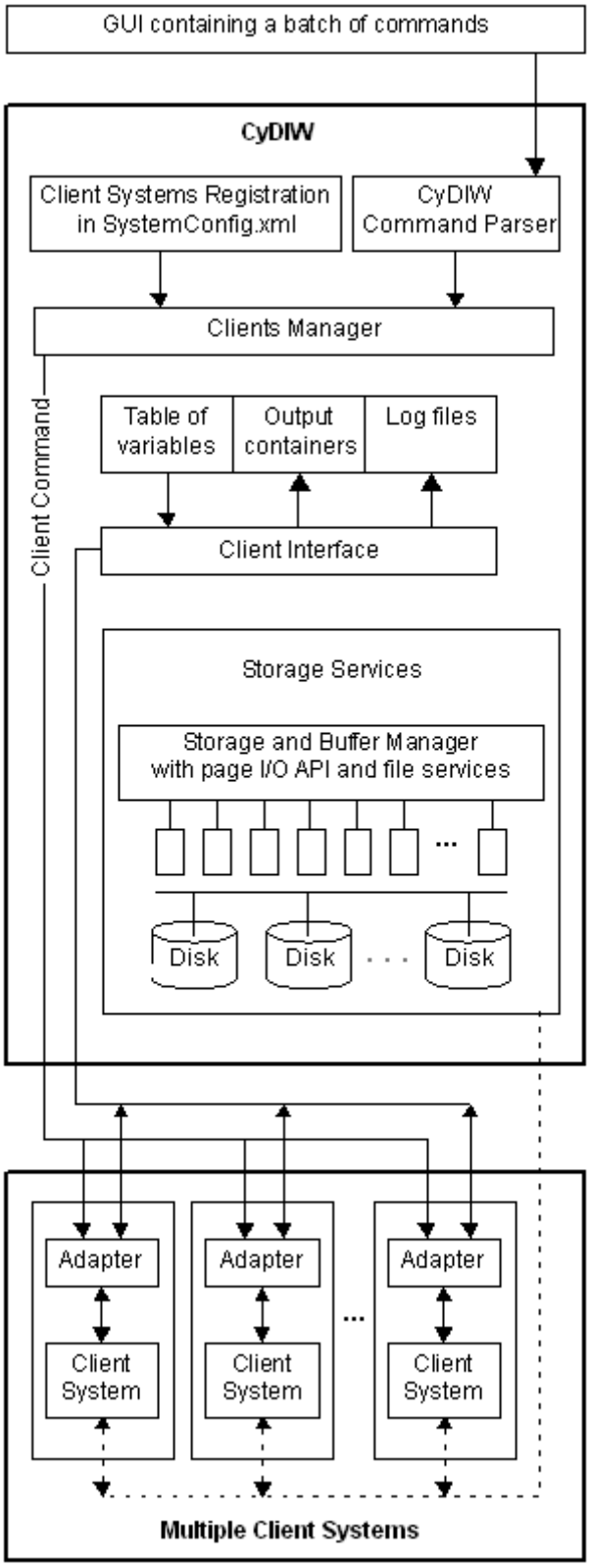

Fig. 1. The Cyclone Database Implementation Workbench (CyDIW) and client systems.

3.2 Client System Registration

The client registration process is managed by CyDIW client manager and consists of two steps. First, a client adapter needs to be implemented according to a common client interface provided by CyDIW. Second, a new client entry needs to be added to the CyDIW system configuration file SystemConfig.xml. Once these two steps are completed, the client system can be loaded when CyDIW starts and the client commands can be executed in CyDIW GUI with the designated prefix. No changes are required from CyDIW's side to add a new client.

Each client entry in SystemConfig.xml contains a fixed set of attributes that provide the necessary information to load a client system. These attributes include the name and prefix of the client, the paths of class files and library files (in case the client system is implemented in Java), the workspace path which stores the input and output files for the client, and the full path of the client adapter class.

For example, the client entry for the Saxon XQuery engine is displayed as follows:

```
<Client Name="Saxon" Prefix="Saxon" Enabled="yes"
    ClassPath="E:\CyDIW_Root\CyDIW_Clients"
    LibraryPath="E:\CyDIW_Root\CyDIW_Clients\saxon\lib"
    WorkspacePath="E:\CyDIW_Root\CyDIW_Workspace"
    ClientAdapter="clientsmanager.clients.SaxonAdapter"/>
```

The attribute "Enabled" indicates whether the client in the current entry will be loaded into the CyDIW platform. If the value of the "Enabled" attribute is "yes", CyDIW is going to load the Saxon client adapter and its executable code according to the class name specified in "ClientAdapter" and the locations specified in "ClassPath" and "LibraryPath". The "Prefix" attribute "Saxon" will be used as the unique identifier for this client in the CyDIW platform. Any command starting with the prefix "$Saxon" in CyDIW will be processed by the client adapter "SaxonAdapter". The result of the Saxon queries can be stored into the "WorkspacePath" if output redirection is used. The details on output redirection will be explained in section 4.1.

3.3 Client Adapter and Client Interface

As we have mentioned above, in order to register a new client to CyDIW, the client adapter is the only part in the client's source code that needs to be added or modified. To facilitate this, a client interface and a client factory class are provided in CyDIW public API and they define all the operations that need to be implemented in the client adapter. The definition of the interface is as follows:

```
public interface ClientInterface {
    public void initialize(CyGUI dbgui, int clientID);
    public void execute(int clientID, String command);
    public String getCustomLogData();
}
```

In the above definition, CyGUI is an interface that defines operations available in the CyDIW GUI instance, such as appending a string to the Output Pane, appending a string to the Console, accessing the clients manager, accessing the value of a variable, updating logging information, and so on. The integer parameter clientID is an internal ID assigned to a client in CyDIW, and can be used to access any information related to this client stored in the clients manager.

An abstract class ClientFactory implements ClientInterface and provides default implementations for initialize() and getCustomLogData() methods. Then, to implement a client adapter, a user can simply inherit the ClientFactory class, implements the abstract execute() method, and overrides some other methods if necessary.

The method initialize() is used to pass a CyGUI instance to the client and take care of the initialization work for the client adapter, so that it is ready to execute client commands at any time. The method execute() can be considered as the entry point of the client system in CyDIW. It takes a client command as input and processes it according to the procedures defined by the client system.

The getCustomLogData() method needs to be implemented, if a client needs to pass user defined logging information to CyDIW. The client adapter is responsible to monitor any user defined measurements and CyDIW will use this method to collect the logging information from the client. In this way, clients have the freedom to report their performance measurements in any granularity. For example, in the client adapter for Saxon XQuery engine, getCustomLogData() method can be implemented to measure only the query compilation time excluding the time spent in other stages. If a user decides to measure the performance of some other steps in Saxon, the user can easily do so by modifying Saxon client adapter.

When a client command is being executed in CyDIW, CyDIW platform will first find the client adapter by the prefix of the command, and use its execute() method to process the command. User defined performance measurements can be collected through its getCustomLogData() method and written into CyDIW's log files.

It needs to be noticed that although a client adapter has to be coded in Java to work with CyDIW public APIs, the client system itself can be implemented in any language. As long as a client adapter is implemented to wrap up the client system, CyDIW is able to use the client to process client commands. For example, R environment [24] can be used as a client system in CyDIW to generate graphical reports for experiments. To achieve this, a client adapter for R is implemented which simply uses R's command batch execution to run R code. If a client system is implemented in a language other than Java (such as c or c++) and logging information needs to be collected from it, then Java native interface can be used to pass information between the native client system and the client adapter in Java.

3.4 Integrated Client Services

Some client systems have also been integrated into CyDIW by default and can be used to facilitate benchmarking experiments.

The operating system commands starting with the prefix "$OS" are registered into CyDIW as a client by default, so that CyDIW can perform normal operating system commands such as file operations easily without leaving the platform. For example, in order to move a file named "data.xml" from the current working directory into a subdirectory named "CyDIW_Workspace", a user doesn't have to leave the CyDIW GUI and move files manually. All one needs to do is to run the command "$OS:> move data.xml CyDIW_Workspace" in Windows or "$OS:> mv data.xml CyDIW_Workspace" in Linux. If this step is required in the middle of an experiment, by executing the above OS command, the experiment can proceed automatically without the user's involvement.

Since the log files are maintained in XML format, XQuery queries can be used to process the data, apply aggregate functions to it and represent it in the way that takes preference of users and consumers into account. Currently, the Saxon XQuery engine, registered with the prefix "$Saxon", is used as the default client to process XML log files.

A client adapter for R environment [24] has also been implemented and will be registered into CyDIW by default at startup. R environment can be used to generate graphical reports for the experimental data, and needs to be pre-installed on a user's machine.

The R client in CyDIW starts its commands by the prefix "$R" and can run R code from a batch. In this mode, R environment will take input from an R code file and it can also pass arguments to the R code from the command line. The following is an example of such a command:

```
$R:> CMD BATCH "--args
  inputFile='CyDIW_Workspace/experimentData.xml'
  outputFile='CyDIW_Workspace/experiment1_plots.pdf'
  title='CyDIW_Experimental_Results'
  xlabel='XMark_Queries'
  ylabel='Execution_Time'
" R_Folder/R_code.txt;
```

A sample of R code is provided in CyDIW which can draw a group of side-by-side bar plots from a set of two-dimensional data stored in an XML file. The graph demonstrated in Section 6.2 is drawn using this R code. Users can modify this code or write completely new R code to fit their own needs, with some knowledge of R graphing functions. The detailed instructions on R programming and batch execution can be found in [24] and [25].

4 CyDIW Commands

A command in CyDIW generally consists of four components: a prefix, an actual command in a client system, an output redirection clause and a logging redirection clause. The latter two components are optional. The general structure of a command can be expressed as follows:

Prefix :> *Command* [*OutputClause*] [*LoggingClause*];

Here, *Prefix* is the prefix assigned to a client system in SystemConfig.xml and must always starts with a "$" character when being used in CyDIW. *Command* is an actual command that can be processed via the client adapter bound to the above prefix. *OutputClause* starts with the keyword "out >>" and informs CyDIW that the output of *Command* needs to be redirected to a designated output file. *LoggingClause* starts with the keyword "log time >>" or "log custom >>" and informs CyDIW that the associated command needs performance logging. If the option "time" is used, the execution time of *Command* will be collected and written to a designated log file. If the option "custom" is used, a user defined performance parameter will be

monitored and collected. The methods getCustomLogData() discussed in Section 3.3 are responsible for monitoring and reporting the user defined logging information to CyDIW.

In order to facilitate the experiments repeatability and benchmarking services, a special set of built-in commands is designed in CyDIW platform which include variable management, log file management, conditional statement and loop statement. All the CyDIW built-in commands start with the prefix "$CyDB", and their syntaxes and usages will be described in the following few sections. To check the syntax of all the "$CyDB" commands at run time, one can run the command "$CyDB:> list commands;" and a list of all the CyDIW built-in commands will be displayed in the Output Pane.

4.1 Logging and Output Redirection

As mentioned earlier, any client command in CyDIW can have an optional output clause *OutputClause* and an optional logging clause *LoggingClause*. Actually, some of the "$CyDB" commands discussed later in next few sections can have optional output and logging clauses as well, such as the "$CyDB:> run", "$CyDB:> if" and "$CyDB:> foreach" commands.

The structure of the *OutputClause* is "out >> *OutputFile*", where "out" and ">>" are keywords and *OutputFile* is the name of an output file. If the output redirection is not used, the result of a command will be displayed in the Output Pane by default. To use the output redirection clause in a client command, the client adapter has to support this feature. In case an output redirection clause is detected, CyDIW will extract the name of that output file and pass it to the client adapter together with the actual client command. The client adapter should then redirect output to the correct file or return an error message if output redirection is not supported.

In order to facilitate logging, XML-based log files can be created where commands can deposit their log entries as XML elements. If two commands are at the same level and are executed sequentially, their elements in the XML log file will be siblings to each other. If two commands are not at the same level, e.g., one command is a conditional or loop statement and the other command is inside the conditional or loop body, the elements of the two commands will have a parent and child relationship in the XML log file – the element of the latter command will become the child of the former one. To achieve this behavior, a log file in CyDIW always keeps track of the next writing location in the file. The location should move forward in the same level for normal commands, but need to go inside or leave a level when entering or leaving a conditional or loop statement. A log file constructed in this way is demonstrated in Figure 4 in Section 6.2.

To avoid the null reference error, CyDIW requires a log file to be created before it can be used. The command "$CyDB:> createlog *LogFile*;" can be used to create a new log file named *LogFile*. If the file already exists, the command will overwrite the existing one and create a new empty file. The command "$CyDB:> useLog *LogFile*;" is used to register an existing log file, so that it can be reused in CyDIW and new logging elements will be appended to the end of the log file. Once a log file is created or registered in CyDIW, it can be used by any of the CyDIW or client commands to record their logging information. Any number of log files can be used.

The structure of the *LoggingClause* is as follows:

log (time | custom) >> *LogTag LogFile*

The strings "log", "time", "custom" and ">>" are all keywords in CyDIW. *LogTag* is the XML tag explicitly surrounded with "<" and ">". The tag is used to embrace the logging information in the log file and can also contain XML attributes. *LogFile* is the name of a log file that has been created or registered in CyDIW.

Two options "time" and "custom" are supported for logging. The time option is controlled by CyDIW; it simply records the elapsed time in milliseconds between dispatch and return of the command when it is executed.

With custom logging it is up to the client to return whatever it wishes as a string of characters. Ideally, the client should keep in mind that the resulting log entry should be a legal XML element. For example, consider the command "log custom >> LogEntry BenchmarkA.xml". When it is executed a log entry shown below may be created.

```
<LogEntry>
   <PgesRead>10</PageRead>
   <PagesWritten>3</PagesWritten>
   <BuffersUsed>5</BuffersUsed>
   <Time unit = "millisecond">105</Time>
</LogEntry>
```

Here, the outer tag LogEntry is inferred from the command. The user has chosen to return a sequence of three XML elements that report the number of pages read, pages written, buffers used, and time. The time reported here is not to be confused by the time returned when "time" option is used in the command.

Note that whether or not to implement the output and logging redirection functions for a client adapter is a decision of the user's, depending on the potential scope of usage of a client system in the workbench. In Section 6.2, in order to compare the performance of three different XQuery engines, we have implemented output and redirections logging for all the three client systems. The usages of output and logging redirections are demonstrated clearly in that example.

4.2 Variables

CyDIW platform supports user defined variables. A user can declare a variable, assign values to it, and use it inside an expression or a command. These variables can be used as loop control variables in a loop statement, environment variables or parameters in a client system command, or simply to store values and compose expressions or commands.

To differentiate a variable in CyDIW platform from a variable in a client system, a special prefix "$$" is used for each variable. Thus, a variable named "x" will be represented as "$$x" in CyDIW. Currently CyDIW supports only four types of variables: string, integer, string array and integer array. An array in CyDIW uses zero-based indexes.

The syntax to declare a variable is as follows:

$CyDB:> declare *Type VariableName*;

For example, the following two statements declare an integer array named "i" with its length 10, and a string variable named "osType":

$CyDB:> declare int[10] $$i;

$CyDB:> declare string $$osType;

Any variable in CyDIW platform is defined in the global scope and maintained in a global variable table in CyDIW, which means once a variable is defined, it can be accessed by any client system commands and is valid until CyDIW terminates or the variable is removed from the global variable table by an "$CyDB:> undeclare" command. The syntax to remove a variable is as follows:

$CyDB:> undeclare *Type VariableName*;

The command "$CyDB:> list variables;" can be used to list all the variables currently in the variable table, and the command "$CyDB:> clear variables;" can be used to clear the current contents of all the variables in the table.

If a variable is not initialized at declaration, an integer variable will be initialized to the value zero and a string variable will be initialized as an empty string by default. The same rule applies to integer or string elements in arrays. The value of a variable can be assigned either at declaration or by the "$CyDB:> set" command after it's declared, as the following two examples show:

$CyDB:> declare int $$j := 5;

$CyDB:> set $$osType := Linux;

In the second statement above, a string value "Linux" is assigned to a string array element "$$osType". Note that when assigning a value to a string variable, the value doesn't need to be quoted to represent a string value. All characters between the assignment operator ":=" and the command terminator ";" – except the leading and trailing white spaces – will be assigned to the string variable. This is the only exception in CyDIW where a string constant is not surrounded by a pair of double quotes. This exception is made because the double quote character and some other special characters may be used frequently in client system commands, and it will be inconvenient and inefficient if one has to use lots of escape characters when assigning a command to a string variable. At any other place, a string needs to be surrounded by a pair of double quotes to represent a string constant. For example, the following conditional statement compares if the string variable "$$osType" is equal to the string value "Windows" and decides whether to execute a block of statements:

$CyDB:> if ($$osType == "Windows") { ... }

Since a string variable in CyDIW can be used to store a command string, a special command "$CyDB:> run" is designed to run the command stored inside a variable. Its syntax is as follows:

$CyDB:> run *PrefixVar CmdVar* [*OutputClause*] [*LoggingClause*];

PrefixVar is a string variable containing the prefix of a command and *CmdVar* is a string variable containing an actual client system command. The syntax and usage of *OutputClause* and *LoggingClause* are the same as what we've discussed in the previous sections.

For example, if two string variables are declared as follows:

$CyDB:> declare string $$p1 := $Saxon;

$CyDB:> declare string $$query1 := for $b in doc("auctions.xml")/site/regions//item return $b/name/text();

Then, executing the command "$CyDB:> run $$p1 $$query1;" has the same effect as executing the client command "$Saxon:> for $b in doc("auctions.xml")/site/regions//item return $b/name/text();". If we want to redirect the output or logging information of the above command, we can add output and logging redirection clauses to it.

In order to make the best use of variables in CyDIW, a special pattern "(*Variable*)" is defined to access the value of a *Variable* from anywhere in CyDIW code. Whenever CyDIW parser encounters the pattern "(*Variable*)", it will replace the pattern with the value of the *Variable*, no matter it appears inside the *OutputFile* in a *OutputClause*, the *LogTag* in a *LoggingClause*, or the string in an assignment command. Using the pattern, users can have more freedom and convenience in designing experiments and generating log files in the preferred formats.

For example, the following loop header is selected from the use case in Section 6.2. The command will add one "<loop2>" element into the log file "benchmark2.xml" in each of its 20 iterations, and the element has a "var" attribute with the value "query($$j)". Thus, the attribute value will become "query1" in the first iteration, "query2" in the second iteration, and so on.

$CyDB:> foreach $$j in [1, 20] log time >> <loop2 var="query($$j)"> benchmark2.xml { … }

4.3 Conditional statement

In order to make it possible to design experiments with more complicated logic, conditional and loop statements are built into CyDIW commands.

The syntax of a conditional statement is as follows:

$CyDB:> if (*BooleanExpression*) [*LoggingClause*] { *IfBlock* }
[else [*LoggingClause*] { *ElseBlock* }]

The conditional statement checks the result of the *BooleanExpression*. If the result is true, the block of commands in *IfBlock* will be executed; if the result is false and the optional else clause exists, the block of commands in *ElseBlock* will get executed. The *LoggingClause* in both if and else clauses are optional.

4.4 Loop statement

The loop statement in CyDIW starts with the keyword "foreach" and its two types of formats are as follows:

$CyDB:> foreach *Variable* in (*ExpressionList*) [*LoggingClause*]
{ *Block* }

$CyDB:> foreach *Variable* in [*Exp1*, *Exp2*] [*LoggingClause*]
{ *Block* }

In the above definition, *Variable* is the control variable of a loop statement and must be an integer. *ExpressionList* is a list of integer expressions and *Exp1* and *Exp2* are two integer expressions. The control variables of other data types haven't been implemented.

The function of the "$CyDB:> foreach" command is to let

a *Variable* iterate through each of the values in the *ExpressionList* (for the first format) or through the range between *Exp1* and *Exp2* (for the second format), and execute the block of commands in *Block* in each iteration. The *LoggingClause* is optional and will add a new pair of XML tags (e.g., <loopTag> and </loopTag>) to the designated log file in each iteration. As have been explained in Section 4.1, if the commands inside *Block* also write logging information to the same log file, the logging elements added by these commands will become the children of the <loopTag> element in each iteration.

5 Storage and Buffer Management Services

CyDIW platform also provides storage and buffer management services to facilitate implementation of database systems.

The user's view of a storage consists of a sequence of pages with integer addresses. Several independent storages can be created although only one can be used in a given batch. Currently page sizes of 1 to 64 kilobytes and page addresses of 32 bit unsigned integers are supported. Thus the maximum capacity of the storage is 256 terabytes. The storage can straddle multiple disks. We have experimented with storage consisting of a few megabytes to gigabytes on laptops and also a storage consisting of 1.5 terabytes that straddles four 500 gigabyte disks. The settings for a storage are stored in an XML based StorageConfig.xml file stored in the root of CyDIW.

The commands to create storage are akin to installing a disk in a system, formatting it with a desired page size, and allocation of a pool of buffers. On creation of storage a bit map to keep track of pages being use is created. Pages can be allocated and deallocated by clients. The storage can be reformatted with a different page size if that is necessary in experimentation. Of course in such a case the contents of the existing storage will be overwritten. However the new storage will have the same foot-print on the disk as the existing one – with the hope that it causes a minimal necessary change to the physical environment. All these features are available via CyDB commands available to clients.

In order to use the storage, the buffer manager has to be started using a CyDB command. CyDB commands to read and write pages – staple operations in database internals – are available. As customary and critical in databases these operations to access pages are mediated by the buffer manager. Every buffer is paired with some book-keeping information to facilitate buffer management. A mapping keeps track of which page is in which buffer. A pin count is associated with each buffer in order to keep track of how many clients (transactions) are currently using the page residing in the buffer. In CyDIW the responsibility to increment and decrement the pin count lies with the client. This simplifies the management but the clients are expected to act responsibly. A page with pin count of 0 makes it a candidate for replacement by the buffer manager operating under a buffer replacement policy. A bit is associated with every buffer to keep track of any change in the buffer. If the bit is set, it is implied that the change is intended to the page residing in the buffer. Therefore, the page is written to the corresponding page on the disk before the buffer is reused by the buffer manager according to its replacement policy. Buffer management via the concept of pinning is of pivotal importance in databases. Often, page-based storage and buffer management represent a major hurdle before any meaningful database implementation can take place. Therefore it is built-in CyDIW. Storage for a project can be created on the fly by executing CyDB commands.

Once a storage is created it can be used by any number clients. It is to be kept in mind that in database implementation the binary format and contents of a page are managed by clients of those pages. Storage is of course persistent and it retains its state after a session if over. Obviously the storage should not have to be created in every session. An existing storage can be used by executing the useStorage command. Inadvertent creation of a storage can be suppressed by commenting out the appropriate commands.

In our lab we develop experiments that use commands to create storage, load data if available or create synthetic data, create and display artifacts such as XML-based parse and expression trees for queries in popup windows or in graphical format. We have already seen other aspects of experiments. Experiment to create self-contained and comprehensive demo of prototypes are encouraged in our lab. Such demos can easily be repeated by others. The use case in Section 6.3 illustrates this for our NC94 prototype. The reader may keep in mind that this paper is on methodology of implementation – we do not consider a detailed description of the NC94 prototype itself in this paper.

6 Benchmarking Use Cases

Three use cases are considered. The first use case shows how SQL and XQuery queries can be executed from the same batch. A remote MySQL server is used in the illustration. Second use case shows benchmarking for execution of 20 XMark queries on three different XQuery engines. The R Environment is used to produce graphical reports. The third use case covers NC94 prototype that has been implemented on the top of CyDIW.

6.1 Use Case 1: SQL vs. XQuery

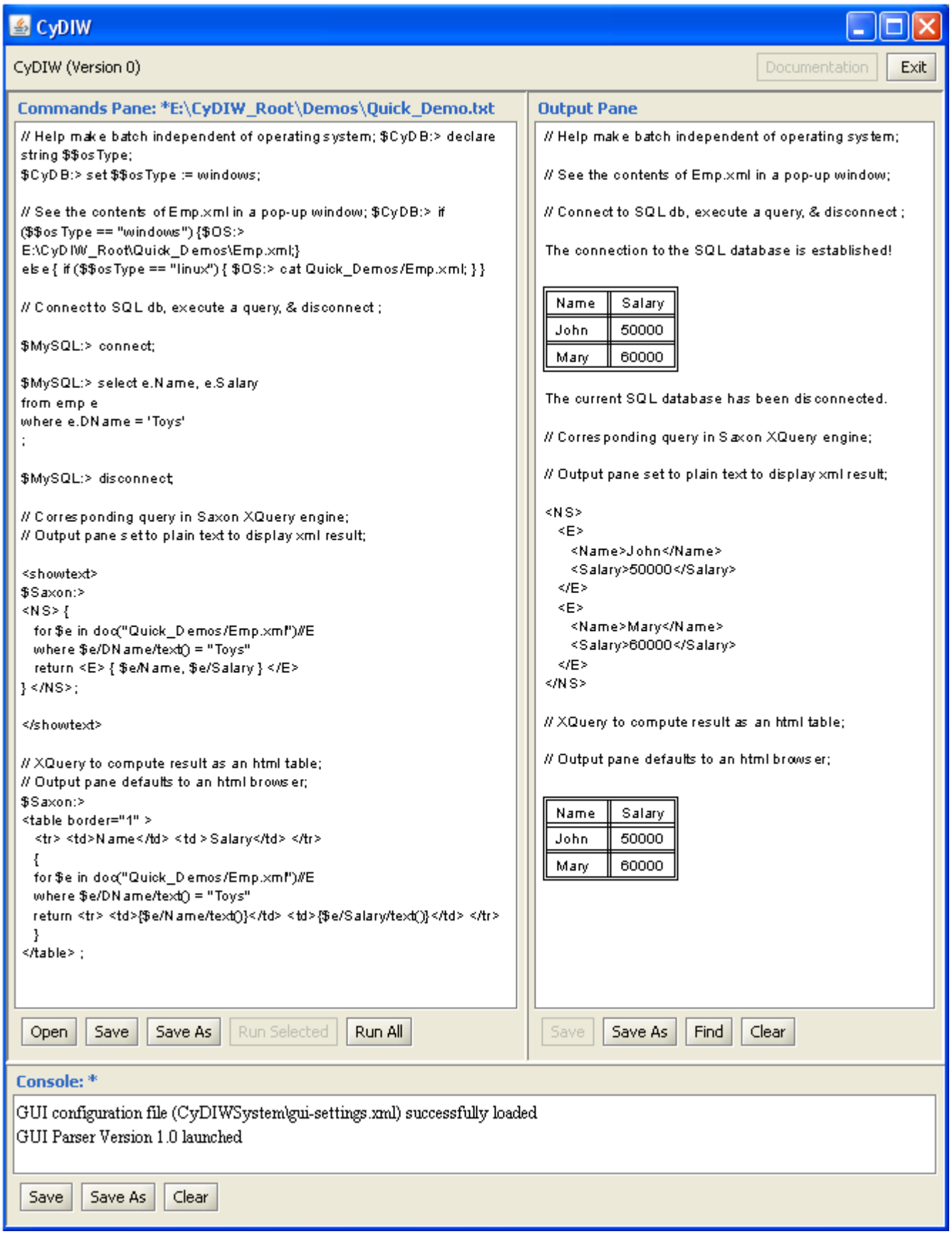

Fig. 2. CyGUI showing the Use Case 1 for Saxon XQuery and a remote MySQL engines.

This use case shows how any existing SQL and XQuery platforms can be used in the same batch. For those who know SQL, this is a useful way of learning XQuery. The demo illustrates several features of the scripting language. The Emp.xml file is displayed using an operating system command in a way that the batch is independent of the choice of the operating system. For SQL, a connection is made to MySQL server, a select statement is executed, and the connection is then closed. Two versions are included for queries in XQuery. By default the output pane is an html browser. To display the text based output the first XQuery query is entered between <showtext> and </showtext> tags. The second XQuery query computes an html table which is readily displayed in the output pane. The demo can be executed by pressing [Run All] button in the GUI.

6.2 Use Case 2: Experiment on XQuery Engines

An experiment in CyDIW is encapsulated in terms of a batch of commands. A complete benchmarking experiment in CyDIW generally consists of the following four steps:

1. Prepare the experiment environment, such as setting up variables and creating log files.
2. Run the experiment and collect performance statistics in XML-based log files.
3. Process the raw logging data using XQuery and reorganize the data in a user's preferred format.
4. Generate graphical reports for the data.

An experiment comparing the performance of three XQuery engines is demonstrated in this section as a use case of CyDIW platform. The experiment is expressed in terms a batch of commands shown in Figure 3, and is designed by following the above four-step procedure.

All the three XQuery engines have been registered in CyDIW as client systems and their client adapter classes are loaded at the start of CyDIW. In order to make a fair comparison among the engines, all the three engines are invoked through the XQJ API [19].

The suite of 20 XQuery queries from XMark benchmark [9] is used to test all three engines in the experiment and the size of the input XML document is 1MB. The queries are organized in nested loops using CyDIW's loop statements and the performance statistics are collected into a log file through the client interface. The logging data is processed in XQuery and plotted it as a graph using R environment.

As Figure 3 shows, the environment variables for the experiment are created in step 1. The string array variable $$prefix holds the prefixes of the three XQuery engines, and the string array $$query holds the 20 XMark queries to be executed. An XML-based log file named benchmarkQ.xml is created to collect the performance statistics from the queries. At the time of creation, the file contains an empty root element <root></root>. The variables $$i, $$j and $$k are used as control variables in the loop statements.

The main part of the experiment in step 2 consists of three levels of nested loops. In order to create an XML-based log file with the intended annotations, each loop header materializes a user- defined tag in the log file. In

```
// Step 1. Setup variables and log files;
$CyDB:> declare string[4] $$prefix;
$CyDB:> declare string[21] $$query;
$CyDB:> declare int $$i, $$j, $$k;
$CyDB:> createLog <root> benchmarkQ.xml;
$CyDB:> declare string $$xmldoc := doc("file:/E:/auctions.xml");
// Assign 20 XMark queries to variables $query[1] to $query[20];
$CyDB:> set $$query[1] :=
for $b in ($$xmldoc)/site/regions/africa/item[@id="item15"]
return $b/name/text();
$CyDB:> set $$query[2] :=
for $b in ($$xmldoc)/site/open_auctions/open_auction
return <increase> { $b/bidder[1]/increase/text() } </increase>;
…
$CyDB:> set $$query[20] :=
let $auction := ($$xmldoc) return <result> … </result>

// Step 2. Query execution and logging of benchmark statistics;
// Outer most looop to iterate through the prefixes of three clients;
$CyDB:> foreach $$i in (1, 2, 3) log time
            >> <loop1 var="($$prefix[$$i])"> benchmarkQ.xml {
    // Inner loop to iterate through 20 XMark queries;
    $CyDB:> foreach $$j in [1, 20] log time
            >> <loop2 var="query($$j)"> benchmarkQ.xml {
        // Execute the query 3 times for warm up;
        $CyDB:> foreach $$k in [1, 3] {
            $CyDB:> run $$prefix[$$i] $$query[$$j];
        }
        // Execute the query 10 times and log performance;
        $CyDB:> foreach $$k in [1,10] {
            $CyDB:> run $$prefix[$$i] $$query[$$j] out>>
              ($$prefix[$$i])_query($$j).xml log time
                  >> <query> benchmarkQ.xml;
        }
    }
}

// Step 3. Execute an XQuery to prepare experimentData xml, to be sent to
R, with average performance of each query for each engine;
$Saxon:>
for $e in doc("CyDIW_Workspace/benchmarkQ.xml")//loop1
    return <loop1> {
        $e/@var,
        for $f in $e/loop2
        return <loop2> {
            $f/@var,
            let $g := $f/query/text() return <avg> {avg($g)} </avg>
        } </loop2>
    } </loop1>
out >> experimentData.xml;

// Display contents of experimentData xml in a pop-up window;
$CyDB:> displayFile CyDIW_Workspace/experimentData.xml;

// Step 4. Use R to generate graphical reports for the experiment;
$R:> CMD BATCH "--args … " R_Folder/R_code.txt;
```

Fig. 3. Use case 2: benchmarking XQuery engines.

the outer most loop, the control variable $$i iterates through each of the three prefixes in the array $$prefix, and a tag "<loop1 var="($$prefix[$$i])">" is inserted into the log file benchmarkQ.xml for each iteration. In the inner loop, $$j iterates through each of the 20 XMark queries in the array $$query, and a tag "<loop2 var="query($$j)">" is added into the log for each iteration.

Two inner most loops are used to execute each query

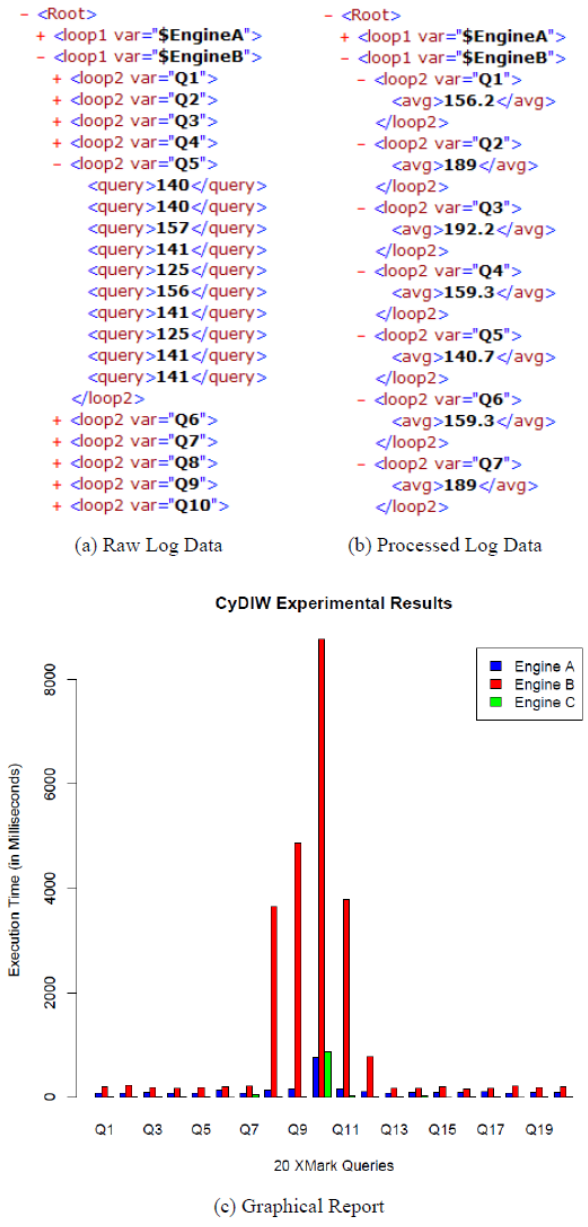

Fig. 4. Benchmark and graphical report.

multiple times for warming up and measuring performance. The first loop executes a query 3 times to warm up the cache. Since we don't care about the performance of the query in this step, the query does not have a logging clause. The second loop executes the query 10 times and deposits its execution time together with a user-defined tag into the log file. All the parameters in the experiment can be changed easily by a user to meet one's specific experimental requirement.

Therefore at the end of step 2, when the benchmarking process is finished, the log file benchmarkQ.xml will end up with a structure of nested elements <root>, <loop1>, <loop2> and <query>, as shown in Figure 4. At this time, the execution time for each query is recorded 10 times. Thus, an XQuery query is used to calculate the average execution time for each query and reorganize the data. This query is handled by the Saxon client. The data file experimentData.xml after applying the XQuery query in step 3 is shown in Figure 4.

In order to generate graphical reports for experimental data, R environment [24] is registered into CyDIW and used as a client system to process data. R scripts can be executed in command batch mode in CyDIW platform.

An R script is provided in CyDIW which can read data from an XML file and generate bar plots based on two-dimensional data. Some parameters can be passed to the code to set the title or labels. This R code can be reused to fit most of the needs on two-dimensional data. Users can also write R scripts by themselves to handle a variety of reporting needs.

The graph in Figure 4 is generated using the above R script. As we can see, the y-axis represents the execution time of a query in milliseconds, and the x-axis contains 20 groups of bars corresponding to 20 XMark queries, with each group consisting of results from 3 engines. The graphical report shows that Engine B does not perform very well from query 8 to query 12. If we want to take a closer look at the comparisons for some other queries, such as query 13 to query 18, we can simply modify the range of the query in the inner loop as "$CyDB:> foreach $$j in [13, 18]", and generate a separate graph for those queries.

It needs to be noted that although the use case in this section only shows an experiment for XQuery engines, CyDIW platform can be used to run experiments for any client system, as long as the client is registered into CyDIW.

6.3 Use Case 3: NC94 Prototype on CyDIW

The third use case covers NC94 prototype that has been implemented on the top of CyDIW. The details of NC94 prototype are beyond the scope of this paper. The objective of the use case here is how the entire behavior of an implemented system can be encapsulated in a batch of commands. Thus this use case is representative of the style in which one would use the CyDIW workbench for database implementation.

The demo starts with creation of page-based storage. Then geographical maps for the north central region of the United States, available as an XML file, is paginated and stored in our own binary format for XML documents. Spatiotemporal soil data is also loaded in our page based-storage. The internal directory of the storage is viewed and the database is opened.

A query is executed in two different ways, first directly as a whole. Then it is stored in a variable that acts as a handle for the query. The handle helps in stepping through the intermediate steps in execution of the query. The query is parsed and stored in an XML-based parse tree. The parse tree, an XML file, is viewed in a pop-up window. A graphical representation of the parse tree is generated on the fly from its XML representation and displayed in a popup window as well. Next, an XML-based expression tree is computed and then displayed. The expression tree is then executed in iterator style using open, getNextTuple, and close, available as commands. In order to harness this feature, all the intermediate steps have to be implemented as commands by the client as

```
// A demo for NC94: Covers creation of paginated storage, loading of
data, parsing and execution of query;
/* Either create new storage and use it
(Highlight commands and click [Run Selected] );
   $CyDB:> createrawstorage Quick_StorageConfig.xml;
   $CyDB:> formatStorage 16;
   $CyDB:> startBufferManager 100;
*/
/* Or use existing storage
(Highlight command and click [Run Selected] );
   $CyDB:> usestorage Quick_StorageConfig.xml;
*/
/* Load GML maps and Soil data
(Highlight and click [Run Selected] );
   $NC94:> LoadNC94GMLData
        E:\CyDIW_Root\Quick_Demos\Iowa_GML.xml;
   $NC94:> LoadNC94Data | Soil |
      E:\CyDIW_Root\Quick_Demos\Iowa_Soil.mdb;
   $CyDB:> showdirectory;
*/

// Open the NC94 database;
$NC94:> OpenNC94Database NC-94.xml;
// create a log file to collect performance stats;
$NC94:> createlog <MyLogRoot> MyLog.xml;

// Execution of NC94 Query in single step - Option 1;
$NC94Query:>
select C.FIPS, C.PctArable
from soil C
|| MyQueryOutput.txt log>> <stat1> MyLog.xml;

// Show compilation and execute step by step - Option 2;
// Store query in variable a[0] - use a[0] as a handle;
$CyDB:> declare string[1] $$a;
$CyDB:> set $$a[0] := $nc94:>
    select C.FIPS, C.PctArable from soil C;
// Parse the query and see parse & expression trees in popups;
$NC94:> ParseQuery $$a[0];
$CyDB:> DisplayParseTree $$a[0] xmlview;
$CyDB:> DisplayParseTree $$a[0] graphicalview;
$NC94:> BuildExpressionTree $$a[0];
$CyDB:> DisplayExpressionTree $$a[0] xmlview;
$CyDB:> DisplayExpressionTree $$a[0] graphicalview;

// Execute the query a[0] in iterator style;
$NC94:> OpenIterator $$a[0];
$NC94:> GetNextTuple $$a[0];
$NC94:> HasNextTuple $$a[0];
$NC94:> GetRemainingTuples $$a[0];
$NC94:> HasNextTuple $$a[0];
$NC94:> CloseIterator $$a[0];

// Close the NC94 database;
$NC94:>CloseNC94Database;
```

Fig. 5. Use case 3: a demo for NC94 prototype implemented on CyDIW.

well.

Finally the NC-94 database is closed. We have also implemented a getRemainingTuples to skip step-by-step execution. In subsequent sessions one may skip the creation of the storage and / or loading of datasets.

The three demos cover a broad range of features and usage of the CyDIW platform. It should be clear that the platform can be shared by students, instructors, developers, managers, and student advisors. The batches of commands can be shared and executed independently.

7 Discussion

All the experiments in this paper have been tested on a machine using AMD Athlon 64 X2 dual-core CPU and a 2GB memory with Windows XP operating system running on top. The CyDIW platform has also been tested on a Linux machine. Thus it is independent of operating system. If a client system is dependent on an operating system it can not be run on other operating systems via CyDIW.

Accuracy is an important issue for benchmarking service tools. In order to avoid the variations caused by caching or other random factors, the warming up and averaging mechanisms are recommended to users when designing an experiment in CyDIW, as have been discussed in the use case in Section 6.2.

We need to ensure that CyDIW platform does not introduce extra overhead to the performance of a client system. As we have mentioned earlier in Section 3.3, CyDIW only measures the total execution time for a client, and the client adapter itself is responsible for monitoring the execution time in stages or any other performance statistics and reporting them to CyDIW. Therefore, the only factor that may affect the accuracy of an experiment in CyDIW platform is due to the potential thread switching overhead added by CyDIW, since multiple threads may run when a client is invoked through CyDIW. To test this overhead, we have run Saxon and MXQuery engines independently in separate Java classes and measured their execution time. We could not notice the difference between the execution time of a client running independently and within CyDIW on a multicore machine.

8 Conclusion

We have presented the details of the Cyclone Database Implementation Workbench (CyDIW) for usage and experimentation on multiple database systems and implement new ones treated as clients.

The paper articulates an experiment as a batch of commands on multiple client systems. CyDIW makes this concept of experiment quite viable by providing a scripting language custom designed for batches of commands. Client commands are enhanced by redirection and logging services offered by CyDIW. Many other services are realized as commands on operating system, CyDIW itself, an XQuery engine, and statistical package R that are themselves realized as clients of CyDIW. Variables allow dynamic substitution of arguments in commands before they are executed – arguments such as queries, file for redirection, client platforms, XML tags in log entries, and even commands themselves. With all these facilities, experiments become self contained as well as fuller in their scope. These experiments, realized as plain text files, consisting of batches of commands, can be exchanged among

users and run on any machine where CyDIW and the client systems have been configured and installed. The installation of CyDIW is as simple as it can possibly be – one simply copies it in a directory of one's choice on a Windows or Linux system where Java is supported. CyDIW is pre-configured and ready to be used.

Existing database engines can be used as client systems by providing adapters and registration in CyDIW. Adapters do not attempt to alter the inner working of a client system they merely setup conduits for receiving commands from and return of output and logging information to the workbench. Several adapters and registration settings are pre-configured in CyDIW.

As stated above, a variable can be used as a container for arguments of a command and act as a handle. This can help decomposition of complex commands into smaller components to help gain useful insight into their inner workings. A good example is execution of a query in databases systems. A query goes through compilation, algebraic optimization, and plan generation with cost estimation before a optimal strategy is chosen for execution. The execution itself can be iterator based that reports one tuple at a time. These tasks can be accomplished via commands and help even developers to easily visualize intermediate artifacts such as expression trees for queries. Vendors can share some internals with customers.

Reporting of benchmarks can be a sensitive issue for vendors. The time option helps in measurement of overall time spent by commands. On the other hand consumers may feel the need for benchmarks with finer granularities that are based upon proprietary information. CyDIW leaves the door open so that in presence of voluntary industry standards a side-by-side comparison of systems would become easier for consumers. Logging can also support some creative uses other than benchmarking.

In implementation of database prototypes page based storage and buffer management are required lowest level services. This takes care of a major hurdle in realistic database implementation.

The overall behavior of a prototype can be easily understood through a batch of commands that starts from creation of a new storage on a disk or spanning multiple disks, paginating the storage with a given page size, creation of synthetic or loading of real datasets, to execution of commands using workbench wide services. This reduces learning curve for newcomers, makes maintenance easier, and also increases the longevity of prototypes. It would also allow group leaders to communicate their needs in a clearer manner to their team members on one hand and demonstrate the results in a concise manner on the other. This would help achieve an all around increase in clarity and productivity. The workbench can also be used to ensure clean delivery once a prototype has been implemented.

The GUI is intentionally kept as simple as possible avoiding unnecessary learning curve. It acts as an editor and a launch-pad for execution of batches of commands. Batches encapsulate system behavior at a level that seems far more abstract than APIs and suitable for sharing with colleagues, managers, executives, students, and faculty.

The use of XML is extensive all across the workbench and implementation of our prototypes. It is used for configuration settings, catalogs, logging, and artifacts such as expression trees. It helps in making the use of configuration and interface settings explicit taking mystery out. Use of XML also facilitates third party utilities such as R.

We have found CyDIW to be a useful tool in instruction. For example, in introductory database courses students can execute SQL and XQuery commands with great ease. This has lead to considerable savings of time for instructors, system support staff, teaching assistants, and – most of all – students. CyDIW has been used in the graduate level database implementation course as well. The platform is now also used for all our research and development efforts in implementation of multiple database prototypes.

Based on our experience we feel that the workbench can be used in instruction as well as research and development in databases; it should prove to be useful to instructors and students in academia as well as programmers, managers, and executives in industry. A version of CyDIW platform is available at research.cs.iastate.edu/cydiw/.

ACKNOWLEDGMENT

The authors are greatly thankful to the anonymous reviewers and editor for their thoughtful suggestions. The authors are thankful to R environment [24] for use of their software tool and appreciate the helpful information from the owners of Saxon [21], MXQuery [22] and XBird [23] projects. The authors are also thankful to the financial support from LASCAC at Iowa State University. Finally our thanks go to Samik Basu, Pete Boysen, and Hridesh Rajan at Iowa State University for his helpful suggestions.

REFERENCES

[1] S. Ma, "Implementation of a Canonical Native Storage for XML," *Master's thesis, Department of Computer Science*, Iowa State University, Dec. 2004.

[2] D. Patanroi, "Binary page implementation of a canonical native storage for XML," *Master's thesis, Department of Computer Science*, Iowa State University, Dec. 2005.

[3] S.K. Gadia, "A Homogeneous Relational Model and Query Languages for Temporal Databases," *ACM Trans. Database Systems*, vol. 13, no. 4, pp. 418-448, 1988.

[4] S.K. Gadia and S.S. Nair, "Algebraic Identities and Query Optimization in a Parametric Model for Relational Temporal Databases," *IEEE Trans. Knowledge and Data Eng*, vol. 10, no. 5, pp. 793-807, 1998.

[5] S.K. Gadia and S.S. Nair, "Temporal Databases: A Prelude to Parametric Data," *Temporal Databases*, pp. 28-66, 1993.

[6] Oracle Database, http://www.oracle.com/us/products/database/index.htm, 2010.

[7] A. Sahuguet, "Kweelt, The Making Of: The Mistakes Made and The Lessons Learned," *Technical Report, Department of Information and Computer Science*, University of Pennsylvania, 2000.

[8] D. Chamberlin, J. Robie, and D. Florescu, "QUILT: An XML Query Language for Heterogeneous Data Sources," *Proceedings of WebDB 2000 Conference in Lecture Notes in Computer*

Sciences, Springer-Verlag, pp. 1-25, 2000.

[9] A. Schmidt, F. Waas, M. Kersten, M.J. Carey, I. Manolescu, and R. Busse, "XMark: A Benchmark for XML Data Management," *Proc. 28th Int'l Conf. Very Large Data Bases*, pp. 974-985, 2002.

[10] OSDL DBT Suite, http://osdldbt.sourceforge.net, 2011.

[11] BenchmarkSQL, http://benchmarksql.sourceforge.net, 2011.

[12] Quest Software's Benchmark Factory for Databases, http://www.quest.com/benchmark-factory, 2011.

[13] TPC-C Benchmark http://www.tpc.org/tpcc, 2011.

[14] Database Benchmarking, Oracle Wiki, http://wiki.oracle.com/page/Database+Benchmarking, 2010.

[15] L. Afanasiev, M. Franceschet, M. Marx, and E. Zimuel, "XCheck: a Platform for Benchmarking XQuery Engines," *Proc. 32th Int'l Conf. Very Large Data Bases*, pp. 1247-1250, 2006.

[16] BumbleBee, http://www.xquery.com/bumblebee, 2011.

[17] XML/SWF Charts, http://www.maani.us/xml_charts, 2011.

[18] JDBC, http://java.sun.com/products/jdbc, 2011.

[19] XQJ, http://www.xqjapi.com/javadoc, 2011.

[20] XQBench, http://fifthelement.inf.ethz.ch:8083/xqbench/, 2011.

[21] Saxon, http://www.saxonica.com, 2011.

[22] MXQuery, http://mxquery.org/, 2011.

[23] XBird, http://code.google.com/p/xbird/, 2011.

[24] R Environment, http://www.r-project.org/, 2011.

[25] Batch Execution of R, http://stat.ethz.ch/R-manual/R-patched/library/utils/html/BATCH.html, 2011.

[26] Ramakrishnan, Raghu and Johannes Gehrke. Database Management Systems, McGraw-Hill.

Xinyuan Zhao obtained his B.E. in Automation and M.S. in Computer Science from Tsinghua University, Beijing, China in 2002 and 2005. He is currently working toward the PhD degree in Computer Science Department at Iowa State University. His research interests include XML databases, XQuery language, and database implementation.

Shashi K. Gadia obtained his B.S.(Hons) and M.S. in Mathematics from Birla Institute of Technology and Science, Pilani, India, and Ph.D. in Mathematics from University of Illinois, Urbana in 1977. His main research interest is in concept of non-atomic values in databases, for example, values with dimensions such as time, space, and beliefs on one hand and hierarchical values such as those in XML on the other. He has worked in temporal, spatial, and multilevel security databases, optimization, incomplete information, query languages, user interfaces, pattern matching, implementation, relational, object oriented databases, and semistructured databases, He has been a faculty in Computer Science Department at Iowa State University, since 1986.