Monitoring SLAC High Performance UNIX Computing Systems

Annette K. Lettsome
Office of Science, SULI Program

Bethune-Cookman College

Stanford Linear Accelerator Center
Menlo Park, California
August 19, 2005

Prepared in partial fulfillment of the requirements of the Office of Science, U.S. Department of Energy Science Undergraduate Laboratory Internship (SULI) Program under the direction of Mr. Adeyemi Adesayna in the Systems Group of the SLAC Computing Services (SCS) at Stanford Linear Accelerator Center (SLAC).

Participant: ________________________________
Signature

Research Advisor: ___________________________
Signature
Monitoring SLAC High Performance UNIX Computing Systems. ANNETTE K. LETTISOME
(Bethune-Cookman College, Daytona Beach, FL 32114) ADEYEMI ADESANYA (Stanford
Linear Accelerator Center, Menlo Park, CA 94025).

Knowledge of the effectiveness and efficiency of computers is important when working with
high performance systems. The monitoring of such systems is advantageous in order to foresee
possible misfortunes or system failures. Ganglia is a software system designed for high
performance computing systems to retrieve specific monitoring information. An alternative
storage facility for Ganglia’s collected data is needed since its default storage system, the round-
robin database (RRD), struggles with data integrity. The creation of a script-driven MySQL
database solves this dilemma. This paper describes the process took in the creation and
implementation of the MySQL database for use by Ganglia. Comparisons between data storage
by both databases are made using gnuplot and Ganglia’s real-time graphical user interface.
INTRODUCTION

Stanford Linear Accelerator Center (SLAC) is a large particle physics laboratory that performs various government-funded experiments. Computers, mainly UNIX platforms, aid some of our experiments by collecting data, storing results, and calculating the findings. When working in a large environment such as SLAC, with experiments and projects that spit out information to its corresponding workstation constantly, it is important to monitor the performance of those computer systems. UNIX has some special commands—like *top* (provides an ongoing look at processor activity in real time), *iostat* (monitors system input/output device loading by observing the time the devices are active in relation to their average transfer rates), *vmstat* (reports information about processes, memory, paging, block IO, traps, and cpu activity), and *ps* (gives a snapshot of the current processes)—that provide various monitoring information. The health and stability of any computer system—no matter its platform, memory capacity, or age—are very important and should be monitored to avoid and prevent mishaps from occurring.

*Ganglia* is a software system that provides monitoring capabilities for high performance computing systems. *Ganglia* is comprised of two daemons, a process that runs in the background generally unaware to and not affecting users, called *gmond* and *gmetad*. These daemons are responsible for the retrieval of various monitoring information, known as metrics, over specified intervals. Some examples of metrics are percent CPU usage, current process/load size, host name, last boot time, etc.

The *Ganglia* system works on a federation of clusters [1], meaning a group of clusters that are each comprised of computers. *Ganglia* uses its daemons in a chain of command method to retrieve the metrics. Each computer or host has its own *gmond* and it sends its monitoring info
in the form XML to the *gmond* of the computer in its cluster that has been designated ‘delegate’. It is then the delegate’s responsibility to retrieve the monitoring info and keep it until *gmetad* retrieves it. After *gmetad* gets the information, it stores it in a round-robin database.

![Ganglia Architecture](image)

**Figure 1 Ganglia Architecture** [6]

*Ganglia* currently works hand-in-hand with the round-robin database (RRD), a database storage system with a fixed size [2]. Therefore, the initialized size is the size it will remain; it is not capable of growing or shrinking. The advantage of this is that the problem of not having enough space to house its expanding info is never an issue. In addition, the RRD manages time-oriented measurements, so working with *Ganglia*’s periodic output is a strong suit. Unfortunately, the downfall of the RRD is its advantage. Being that the nature of the round-robin database is to remain fixed in size, when there is no longer space available for the incoming information, the not so current data that it presently holds is compacted. By doing so, the accuracy of the older information is lost. Therefore, when querying the database to find the memory usage of a particular host on a date exactly one year ago today, the resultant data would be an estimation rather than the exact figure. Since the round-robin database does this, it is
advantageous to find a more adequate storage system, even though other methods increase the risk of infinite growth.

The alternative way of storing the information is to use a traditional relational database, specifically MySQL. To design and create the tables, the monitoring info that needed storage—the XML dump of the metrics delivered by the gmond and gmetad daemons—was studied. The obvious relationships found to model the tables after are as follows:

- A ganglia system has many clusters.
- A cluster has many related hosts.
- A host has many monitored metrics.

**PROJECT OVERVIEW**

Two methods taken to study the Ganglia system and learn exactly what it does were by:

1. Logging on remotely to various hosts and using commands like top and iostat to see the monitoring info that Ganglia collected at the source, and by
2. Viewing the real-time graphical web interface Ganglia produced with the information gathered and stored in the round-robin database.

After the basic understanding of Ganglia was developed, the creation of a database that would hold the relevant monitoring information proceeded. The database’s purpose was to become a medium to query different monitoring information from various hosts, clusters, and metrics. The control of the database and its activities became the responsibility of a program, since the database is unable to differentiate between which information is to store where and how without assistance. The program’s task was to initially create tables as needed, fill the tables
with its corresponding information by deciphering the XML output from Ganglia and convert it into structured query language (sql) statements, and constantly update the tables with new information.

**DATABASE**

The design of a database is the first, most difficult, and longest stage in the creation of a database. This is because it involves planning and numerous revisions to make it normalized. “In relational database design, normalization is the process of organizing data to minimize redundancy. Normalization usually involves dividing a database into two or more tables and defining relationships between the tables. The objective is to isolate data so that additions, deletions, and modifications of a field can be made in just one table and then propagated through the rest of the database via the defined relationships” [3].

As stated earlier, when designing the database for Ganglia, the structure of the XML output helped in identifying the relationships. The first design plan consisted of six tables, which was to hold all the information ever created for every grid, cluster, host and metric. This first design wasted memory space because all of the values in the metric tables were to be stored as string, even though some values were actually numeric and would take up less if they were stored effectively.

![Figure 2 Design Schema #1](image)
With that epiphany, the table designs that followed spilt the metric table into smaller tables depending on its type. The latest design of this method consisted of eight tables accommodating string, numerical, floating point, and time metrics. This method’s downfall was its potential growth factor and scalability issues. Scalability “refers to how well a hardware or software system can adapt to increased demands. For example, a scalable network system would be one that can start with just a few nodes but can easily expand to thousands of nodes” [3]. This method had a scalability flaw since each metric table held more than one metric and the addition of rows due to updates overtime would slow down its search time when querying even if indexed.

When thinking on the potential growth of any individual table over time and its decrease in search time, it became evident not to concern so much about the amount of tables, but rather the size of each table. This new philosophy led to the final design of the tables. The idea of the
The current design of the tables (Fig. 4) is economical and scalable. It is economical in that each value type corresponds with the appropriate information being stored and scalable since only the specific metric info will be stored in a given table thus making expansion easier.
With the current design completed, the implementation proceeded. Direct MySQL implementation created the database itself and the four basic tables—cluster, host, metric and metricList. The program dealt with everything else—from the creation of the specific metric tables to the insertion and comparison of data.

Figure 4 Second to Last Design Schema

PROGRAMMING

The programming played an instrumental role in the project. Without a program, some data entry clerk would have the grueling task of reading through the XML output and manually inserting the new information into the database. That task in itself would not be a one-man’s job and would take forever especially since the metric values updates are available at least every 60 seconds.

A C++ structure guide for the program, created by my mentor, jump-started the coding process. The guide included the header file of the Expat XML Parser. A parser is “a program that dissect source code so that it can be translated into object code” [3]. Specifically, Expat “is a stream-oriented parser in which an application registers handlers for things the parser might find in the XML document (like start tags)” [4]. This parser, written in C, helped with the
deciphering of the XML code. The guide also used some functions from the MySQL C API, an application program interface that “allows C programs to access a database” [5]. After studying the guide for basic understanding, it became apparent that a good bit of the code needed the MySQL C API itself.

The program initially needed to read the XML code in order to populate the four basic tables (see Tables 2 thru 5 for descriptions). During the initial population of the metric and metricList tables, the specific metrics tables were to be simultaneously created and filled. To distinguish the specific metric tables from each other, a naming scheme was created. The metric name and host identification number of each specific metric table would be its identifier. For instance, if the hostID for gala002 was 144 and one of its metrics were mem_buffers, then that metric table’s name would be “mem_buffers_144.” This method allowed the creation of uniquely named tables that also served an identification purpose. (See Appendix A for the table creation code.)

In order to update the table with new info, a comparison between the last value and the newly retrieved value occurs. The word ‘update’ in this context refers to the addition of new rows rather than replacing old value with new ones. If that were the case, the database would not hold a history of info. The program is to parse through the XML code every 60 seconds, which is the minimum tmax for the metrics. Recall that some metrics have a tmax as great as 1200 and that the values do not necessarily change after each metric call. If there were no comparison statement, duplications would occur in the table and waste space. The code therefore compares the new info with the last input and if it has changed, it will be stored. Data redundancy in the aspect of rows 1, 5, & 8 being identical is wanted because the purpose of the database is to note the fluctuations. On the other hand, it is unwanted if rows 7, 8, & 9 are identical because the
data could have just been saved in one row instead of the three. This is the comparison code’s main purpose. (See Appendix B for update function containing the comparison code.)

The update function marks the end of the basic code. All programming details mentioned thus far lies in an infinite loop. With each iteration, the XML dump will be parsed and everything will repeat over again, after sleeping for 60 seconds. The addition of cluster, host, and metric information happens if and only if new information appears. Creation of new specific metric tables occurs whenever the dump reveals a new metric, while updates occur with justification of the comparison statement.

IMPLEMENTATION AND ANALYSIS

Implementation of the MySQL database with driver program proceeded on August 3. First, I manually created the basic four tables—cluster, host, metric and metricList. Next, the program was compiled and executed using the main gmetad daemon, ganglia01. The specificMetric tables were created as described and all tables received their appropriate data.

After running the program for a week, I noticed that my metricKey from both the metricList and corresponding specificMetric tables were larger than what my calculation said it should be. Realizing the problem, I knew exactly where the mishap occurred, fixed it, and restarted the implementation process on August 13.

Everything seems to be running smoothly thus far with the current implementation. The database has 68449 tables, being that it is storing info from 2349 hosts. It takes my program approximately 13 minutes to parse through the XML dump provided by Ganglia. I expect the database to increase by at least 224 tables since my previous implementation showed that there
are at least 2357 hosts being monitored. There is uncertainty with the amount of host being monitored because when a host has a heavy load it does not produce its heartbeat. The heartbeat is used by Ganglia to report whether a host is up and they are usually pegged as ‘down’ when the heartbeat is not retrieved. Moreover, I believe that if a host is marked as ‘down’ for a long time span Ganglia temporarily forgets that it was being monitored and decreases its host monitoring total.

**FUTURE / FURTHER WORK**

The database could probably be fine tuned sometime to decrease search time. The program has the ability to indefinitely create and update tables in the database as long as they are being monitored by Ganglia. Also, if there needs to be a change in the columns of a table to either stop collecting a particular piece of information or to add a piece, the code can be altered. The code has commented explanations of what is being done every step of the way. Therefore, anyone should be able to pick up where I left off and continue the project if further work needs to be done in the future.

If I had more time to with this project, I would figure out how to shorten the hostName for each host instead of adding the common extension for all 2349+ hosts. I would also add a grid table to show which clusters belonged to which grids. It was not included in my final design plan because at the time Ganglia did not provide that info in its XML dump. In addition, I would add a clause to the code that would only check for an update when that particular metric’s tmax was due instead of checking all specificMetric tables with each iteration. This method should decrease the current iteration length of 13 minutes. Finally, I wish I had more time to see my
program run. Being that its final implementation began so late, I was unable to get enough data to make a thorough comparison of the two storage systems. Maybe I would get a chance after a year’s span of time to see if the database accomplished its main goal of storing accurate data that could be retrieved in the future.

ACKNOWLEDGEMENTS

This project was conducted at the Stanford Linear Accelerator Center. First and foremost, I would like to thank GOD for giving me air to breathe, food to sustain me, and shelter to protect me while I was at this internship. I want to thank my mentor, Adeyemi Adesanya, for putting up with me for the entire eight weeks. I know that I can be a hand-full at times. I also want to thank my computer science professors at Bethune-Cookman College for telling me about this program and sending me off with acquired knowledge. Special thanks to my parents for answering their cell phones at least three times a day to speak with their lonely child. In addition, I want to thank the SULI program especially the people at SLAC for choosing me. Finally yet importantly, I would like to thank myself, Annette Kamilah, for accomplishing the goals that were set and succeeding.
REFERENCES


### FIGURES AND TABLES

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Key</th>
<th>Default</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>clusterID</td>
<td>int(10) unsigned</td>
<td>PRI</td>
<td>NULL</td>
<td>auto_increment</td>
<td></td>
</tr>
<tr>
<td>clusterName</td>
<td>varchar(20)</td>
<td>UNI</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2  Cluster Table Description

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Key</th>
<th>Default</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>hostID</td>
<td>int(10) unsigned</td>
<td>PRI</td>
<td>NULL</td>
<td>auto_increment</td>
<td></td>
</tr>
<tr>
<td>hostName</td>
<td>varchar(20)</td>
<td>UNI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IP</td>
<td>varchar(15)</td>
<td>UNI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>clusterID</td>
<td>int(10) unsigned</td>
<td>MUL</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3  Host Table Description

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Key</th>
<th>Default</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>hostID</td>
<td>int(10) unsigned</td>
<td>PRI</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>metricKey</td>
<td>int(10) unsigned</td>
<td>PRI</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4  Metric Table Description

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Key</th>
<th>Default</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>metricKey</td>
<td>int(10) unsigned</td>
<td>PRI</td>
<td>NULL</td>
<td>auto_increment</td>
<td></td>
</tr>
<tr>
<td>metricName</td>
<td>varchar(40)</td>
<td>UNI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>type</td>
<td>enum('uint8','int8','uint16','int16','uint32','int32', 'float','double','timestamp','string')</td>
<td>YES</td>
<td>MUL</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>units</td>
<td>varchar(15)</td>
<td>YES</td>
<td>NULL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>source</td>
<td>enum('gmond','gmetric')</td>
<td>YES</td>
<td>MUL</td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>

Table 5  metricList Table Description

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Key</th>
<th>Default</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>metricKey</td>
<td>int(10) unsigned</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hostID</td>
<td>int(10) unsigned</td>
<td>MUL</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lastUpdate</td>
<td>Timestamp</td>
<td>YES</td>
<td>PRI</td>
<td>CURRENT_TIMESTAMP</td>
<td></td>
</tr>
<tr>
<td>value</td>
<td>int(10) unsigned</td>
<td>[if type is integer]</td>
<td>MUL</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>value</td>
<td>varchar(25)</td>
<td>[if type is string]</td>
<td>MUL</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>value</td>
<td>double</td>
<td>[if type is floating point]</td>
<td>MUL</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>value</td>
<td>datetime</td>
<td>[if type is time]</td>
<td>MUL</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Table 6  specificMetric Tables Description
The graphs of Comparison #1 compare the input/output for bronco014 over a 12 hours span. Specifically, ["08/14/05 23:00:00";"08/15/05 11:15:00"].
The graphs of Comparison #2 compare various CPU stats for bbr-xfer05 over a 4-hour span. Specifically, ["08/15/05 08:00:00":"08/15/05 12:00:00"].
The graphs of Comparison #3 compare the total processes for pippin01 over a 24-hour span. Specifically, ["08/14/05 12:30:00";"08/15/05 12:30:00"].
void GangliaDB::createSpecificMetricTable(const char **attr, int hostID)
{
    char queryBuffer[BUFFERSIZE];
    char *setValue;

    // condition statements figure out which type the particular value needs to be stored as and saves it as setValue

    if(strcmp(attr[5],"uint8")==0) setValue = "tinyint unsigned NOT NULL";
    else if(strcmp(attr[5],"uint16")==0) setValue = "smallint unsigned NOT NULL";
    else if(strcmp(attr[5],"uint32")==0) setValue = "int unsigned NOT NULL";

    else if(strcmp(attr[5],"int8")==0) setValue = "tinyint NOT NULL";
    else if(strcmp(attr[5],"int16")==0) setValue = "smallint NOT NULL";
    else if(strcmp(attr[5],"int32")==0) setValue = "int NOT NULL";

    else if(strcmp(attr[5], "string") == 0) setValue = "varchar(25) NOT NULL";

    else if(strcmp(attr[5], "float")==0 || strcmp(attr[5], "double")==0) setValue = "double NOT NULL";

    else if(strcmp(attr[5], "timestamp")==0) setValue = "datetime NOT NULL";

    // statement allows the creation of tables along with its indices; setValue used here for database statement
    Sprintf (queryBuffer, "Create table IF NOT EXISTS %s_%u ( metricKey int unsigned , hostID int unsigned, lastUpdate timestamp PRIMARY KEY, value %s, INDEX indx1 (value, lastUpdate, metricKey, hostID), INDEX indx2 (value, metricKey, hostID), INDEX indx3 (lastUpdate, metricKey, hostID), INDEX indx4 (value, hostID, metricKey), INDEX indx5 (hostID, metricKey))", attr[1], hostID, setValue);

    if(mysql_real_query(mysqlSess, queryBuffer, strlen(queryBuffer)) ){
        cerr << "Create statement: " << queryBuffer << " did not work. " << endl;
    }
} // end void createSpecificMetric
void GangliaDB::updateSpecificMetricTable(const char **attr, int hostID, int mKey)
{
    char queryBuffer[BUFFERSIZE];

    int hID;       // holds holdID; used just for verification purposes
    char *mName;   // holds metricName; used just for verification purposes
    char* mValue;    // holds value from specificMetric
    int rowCounter;      // use to control the loop to place first values into tables

    sprintf(queryBuffer, "Select COUNT(metricKey) from %s_%u", attr[1], hostID);
    if(mysql_real_query(mysqlSess, queryBuffer, strlen(queryBuffer)) ){
        cerr << "Select statement: " << queryBuffer << " did not work. " << endl;
    }
    result = mysql_store_result(mysqlSess);
    if(!result) {
        cerr << "store_result Failed" << endl;
    }
    if(!mysql_num_rows(result)) {
        cerr << "No rows; count =0 " << endl;
    }

    row = mysql_fetch_row(result);
    rowCounter = atoi(row[0]);   // convert to integer

    // free results
    mysql_free_result(result);

    if(!rowCounter) { // meaning that the table is empty
        // insert into specificMetric table according to its type; must be converted differently if float, integer, etc.

        if (strcmp(attr[5],"uint8")==0 || strcmp(attr[5],"uint16")==0 || strcmp(attr[5],"uint32")==0) {
            sprintf(queryBuffer, "insert into %s_%u (metricKey, hostID, value) values(%u ,%u,  %u)", attr[1], hostID, mKey, hostID, atoi(attr[3]));
        }
else if(strcmp(attr[5],"int8")==0 || strcmp(attr[5],"int16")==0 || strcmp(attr[5],"int32")==0)
      {
        sprintf(queryBuffer, "insert into %s_%u (metricKey, hostID, value) values(%u, %u, %d)", attr[1], hostID, mKey, hostID, atoi(attr[3]));
      }
else if(strcmp(attr[5], "string") == 0 || strcmp(attr[5], "timestamp")==0)
      {
        sprintf(queryBuffer, "insert into %s_%u (metricKey, hostID, value) values(%u, %u, "%s")", attr[1], hostID, mKey, hostID, attr[3]);
      }
else if(strcmp(attr[5], "float")==0 || strcmp(attr[5], "double")==0)
      {
        sprintf(queryBuffer, "insert into %s_%u (metricKey, hostID, value) values(%u, %u, %f)", attr[1], hostID, mKey, hostID, atof(attr[3]));
      }
else{
      cout << "My unknown type is " << attr[5] << endl;
      // clear free queryBuffer just incase;
      sprintf(queryBuffer, "");
    }

    if(mysql_real_query(mysqlSess,queryBuffer,strlen(queryBuffer))) {
        cerr << " failed to execute mysql query: " << queryBuffer << endl;
    }

    // select rows from specified place and sort them with the most recent update first; then only retrieve that first rows and use it for comparison

    else{
      sprintf(queryBuffer, "select value, metricName, hostID from %s_%u INNER JOIN metricList ON %s_%u.metricKey = metricList.metricKey where lastUpdate < CURRENT_TIMESTAMP order by lastUpdate DESC", attr[1], hostID, attr[1], hostID);

      if(mysql_real_query(mysqlSess,queryBuffer,strlen(queryBuffer))) {
          cerr << " failed to execute mysql query: " << queryBuffer << endl;
      }

      result = mysql_store_result(mysqlSess);
      if(!result) {
          cerr << "store_result Failed" << endl;
      }
      if(!mysql_num_rows(result)) {
          cerr << "No rows, but why?; count =0 " << endl;
      }
  }
row = mysql_fetch_row(result);
mValue = row[0];
mName = row[1];
hID = atoi(row[2]);

// free results
mysql_free_result(result);

// verify that input goes into current table; comparison uses mValue at last part of if statement
if(hID == hostID && strcmp(mName,attr[1]) ==0) // this verification is probably not necessary
{
    if ((strcmp(attr[5],"uint8")==0 || strcmp(attr[5],"uint16")==0 || strcmp(attr[5],"uint32")==0) && atoi(mValue) != atoi(attr[3])){
        sprintf(queryBuffer, "insert into %s_%u (metricKey, hostID, value) values(%u, %u, %u)", attr[1], hostID, mKey, hostID, atoi(attr[3]));
    }
    else if((strcmp(attr[5],"int8")==0 || strcmp(attr[5],"int16")==0 || strcmp(attr[5],"int32")==0) &&  atoi(mValue) != atoi(attr[3])){
        sprintf(queryBuffer, "insert into %s_%u (metricKey, hostID, value) values(%u, %u, %d)", attr[1], hostID, mKey, hostID, atoi(attr[3]));
    }
    else if((strcmp(attr[5], "string") == 0 || strcmp(attr[5], "timestamp") == 0) && strcmp(mValue, attr[3]) != 0){
        sprintf(queryBuffer, "insert into %s_%u (metricKey, hostID, value) values(%u, %u, "%s")", attr[1], hostID, mKey, hostID, attr[3]);
    }
    else if((strcmp(attr[5], "float") == 0 || strcmp(attr[5], "double") == 0) && atof(mValue) != atof(attr[3])){
        sprintf(queryBuffer, "insert into %s_%u (metricKey, hostID, value) values(%u, %u, %f)", attr[1], hostID, mKey, hostID, atof(attr[3]));
    }
    else{ // mValue == attr[3], so do nothing, need to clear free queryBuffer; it should contain select statement;
        sprintf(queryBuffer, "\n");
    }

    if( mysql_real_query(mysqlSess,queryBuffer,strlen(queryBuffer)))
        cerr << " failed to execute mysql query because attr[3] == mValue " << endl;
}

} // end internal if statement

} // end else

} // end update specificMetric