Process accounting collection

Student:
Goran Cetušić

Supervisor:
Ricardo da Silva

CERN openlab
December 8, 2011

Contents

Abstract

The objective is the collection of data generated by the standard psacct package as opposed to a modified version currently in use on most lxbatch and lxplus machines, and the storage of the data to a central repository for easier access. This enables automatic report generation and security audits as well as some usage summaries per user/machine/process.

As one of the goals is to use standard tools, the architecture is based on the transmission of documents over HTTP to a proxy authenticated by Kerberos and stored in a central NoSQL database with predefined views.

1 Introduction

Most of the machines in lxbatch and lxplus clusters use sysacct, a modified version of psacct available from standard Scientific Linux RPM repositories. The modifications were made mostly because the accounting files generated by psacct are stored to AFS repositories and their collection is required to work properly with system log rotation mechanisms. This lacks report generation so a better solution is required. The conclusion was that the process accounting software should use the standard Scientific Linux psacct package with a repository capable of generating reports.

Languges	**Databases**	**Services**	**Protocols**
Bash	MongoDB	Apache	JSON
Python	CouchDB	Kerberos	XML
Javascript			HTTP
C			GSSAPI

Figure 1: Technologies used and researched

Using some of the technologies in Figure 1, the software should collect all data from accounting files, send them to a central repository and generate reports by sending queries to the repository.
Basic questions the queries should answer are:

- Which commands did a user execute?
- On which machines was he/she active?
- What time was he/she active?
- Where was this command executed, by whom and at what time?
- What is the first and last time a user was active on a machine?
- What time did he/she execute a command on a machine?

An important thing to remember is that the queries will rarely be executed; mostly when security incidents occur or daily to generate activity reports. On the other hand, the write load on the repository will grow as more machines are added.

2 Collector

A potential solution for collecting psacct data was Gratia, an open source project for cluster monitoring from Open Science Grid. It consists of a collector (central repository) writen in Java and several probes that send data from a machine to the collector. The available Gratia probes are mostly written in Python and support several methods of data retrieval apart from psacct (dCache, Condor and Hadoop).

Record data from Gratia's psacct probe are sent to the central repository as XML documents. However, the documents represent summaries for a particular user; how much of the overall resources did a user consume, first and last time the user executed a command etc. No information about particular commands needed for successful security audits is stored with Gratia.

The Gratia probe-collector architecture uses a custom protocol for XML file transfers. It was apparent that OSG Gratia did not conform to the project specification because of the custom protocols. A decision was made to reuse the Python code from Gratia's psacct probe to collect record data and store it inside a NoSQL database as opposed to the Gratia collector written in Java.

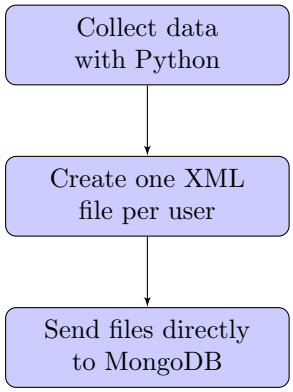


Figure 2: First draft

When the probe is started it checks for files in the default psacct folder. It moves any accounting files inside the directory that were archived by logrotate to its own directory folder specified inside the configuration file. The newest pacct file is also moved unless it is empty and an empty file is created in its place to prevent logrotate from creating additional archived files. Accounting is then started on a file inside the probe directory also specified in the configuration file. Record collection is started on all accounting files inside the probe directory.

Several modifications were made to the Gratia psacct probe. The original database chosen for record storage was MongoDB but the records were formatted as XML documents. Since MongoDB can only receive JSON files, the Gratia probe was changed to store records as JSON files, not XML. In addition to summaries, one JSON document per user with complete information about commands is sent to the database every time the probe is executed. If an ac-

counting file was created less than an hour ago, the file will not be processed and sent to the repository.

If there was a problem sending JSON files during last probe execution, the files are stored and an attempt to send them during next probe execution will be made until they are sent or a timeout has been reached. Accounting is restarted on a new file and the old one is processed. The probe checks if the files can be archived before sending them or aborts execution if there is an error.

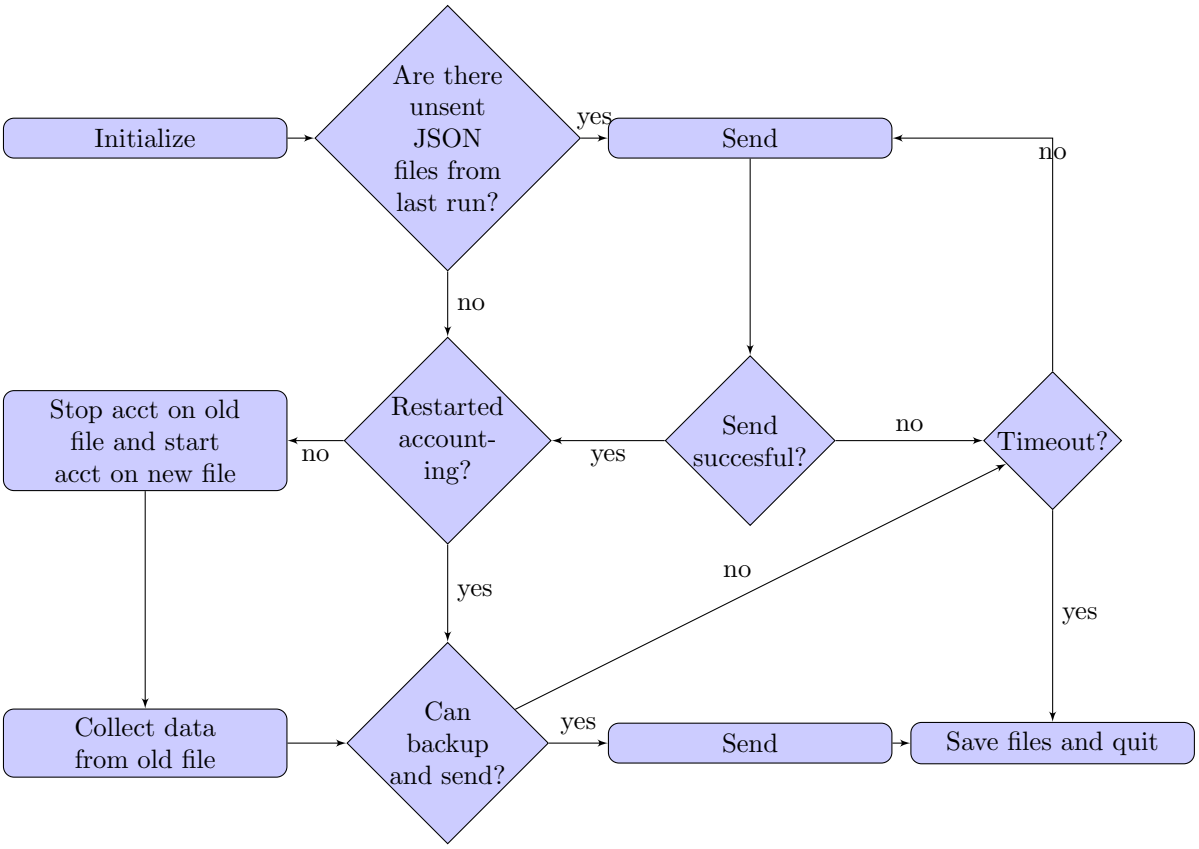


Figure 3: Code diagram

The main files for the probe are:

- PSACCTProbe.py - basically just calls PsAcct() in PSACCTProbeLib.py or prints help
- PSACCTProbeLib.py - moves files, does backups, creates record objects and calls Send() in GratiaCore.py
- Gratia.py - mostly wrapper functions for records
- GratiaCore.py - does the actual json creation and sending (by using GratiaAuth.py)

- GratiaAuth.py - Kerberos authentication, HTTP request creation and server connections are handled here
- GetProbeConfigAttribute.py - configuration file parser
- collector.conf - configuration file containing database addresses, folder locations and other settings
- psacct_probe.cron.sh - cronjob script

The recommended usage of the probe is to run the psacct_probe.cron.sh script every hour by using crontab. The script will then invoke PSACCTProbe.py. For a large number of machines sending records to the database, the crontab entry should be changed for every machine the probe is installed to so the records are not sent at the same time which could potentially cause a denial of service from the database.

Another problem that occured during development was that NoSQL databases have no real network security. MongoDB only supports username and password authentication without encryption. It is expected to run in a completely trusted environment which was not an option for the lxbatch and lxplus clusters. One of the other popular NoSQL databases, CouchDB, has SSL encryption but only in most recent versions which are not available in SCL 5 repositories. But even with SSL encryption, it still uses username/password authentication which could not be integrated with the current CERN infrastructure. The implemented solution is a reverse proxy (Apache) that supports Kerberos authentication and forwards HTTP requests to the database. The problem with this setting was that Apache would send JSON files over HTTP. MongoDB, the first database chosen for record storage, used a custom protocol to send JSON files. Instead of stripping HTML before forwarding the request to the database, MongoDB was replaced by CouchDB with a RESTful API.

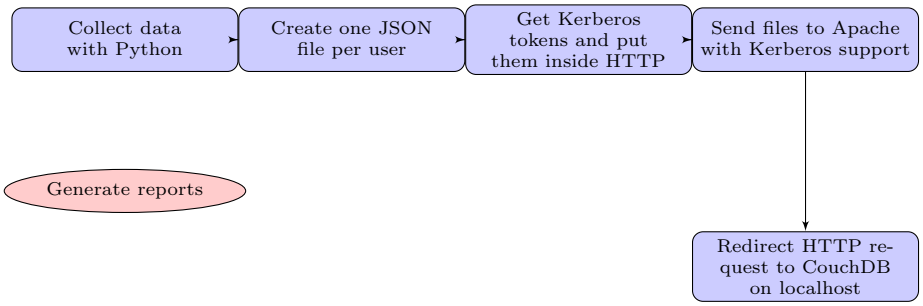


Figure 4: Final architecture

3 Authentication

3.1 Kerberos

The probe uses a Python kerberos module to request the service ticket from Kerberos. One thing to note is that the module used for CouchDB does not exist in SCL 5 repositories. There is a custom module in nonstandard CERN repositories but it is no longer maintained and after several tests it was concluded that it does not work with the current setup. The client connecting to Apache must already have a Kerberos TGT in its cache that is generated by calling "kinit -k" or some other command. This command is executed by one of the scripts executed by cronjob. It uses the service ticket for the Apache server instead of user/pass authentication so it integrates seamlessly into current lxbatch and lxpluss authentication mechanisms.

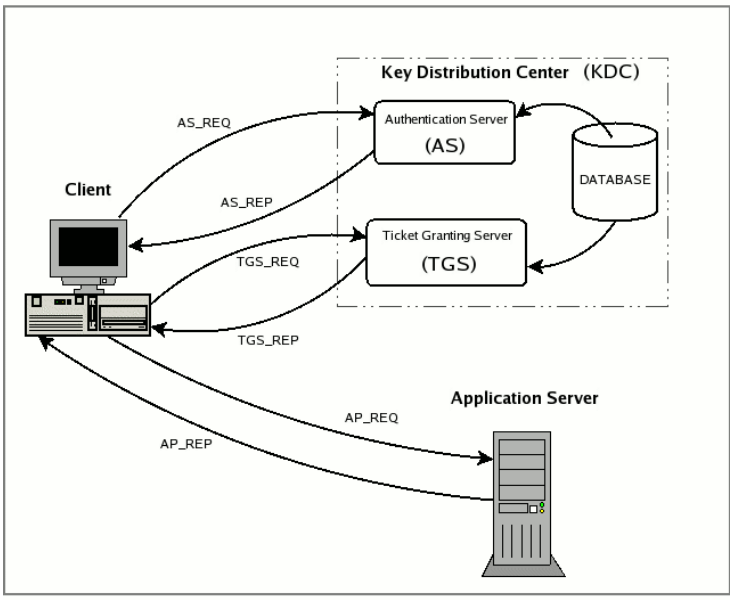
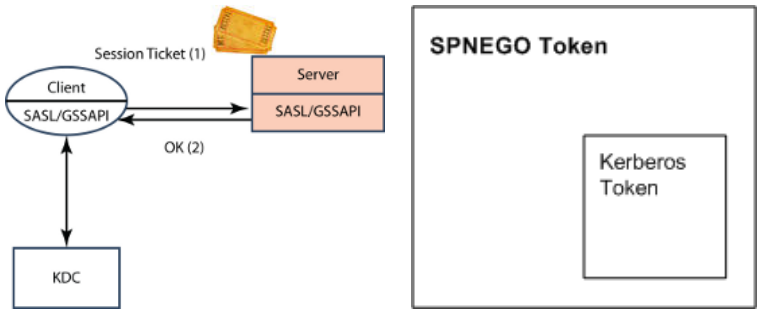


Figure 5: Kerberos token exchange

SPNEGO (Simple and Protected GSSAPI Negotiation Mechanism) is a GSSAPI "pseudo mechanism" that is used to encapsulate Kerberos tickets. SPNEGO's most visible use is in Microsoft's "HTTP Negotiate" authentication extension that is used by Apache. Authentication is implemented inside GratiaAuth.py.



(a) GSSAPI (b) SPNEGO

Figure 6: GSSAPI Authorization with SPNEGO

GratiaAuth.py retrieves the Kerberos token from the Kerberos server, encapsulates it inside a GSSAPI token which is the value field of the SPNEGO scheme. The final result is a HTTP request with headers similar to the header in Figure 7:

```
GET /private/index.html HTTP/1.1
Host: localhost
Authorization: Negotiate QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

Figure 7: HTTP Authorization with SPNEGO

3.2 Apache

Once the records are collected on a particular machine and the GSSAPI token is encapsulated inside the HTTP request which carries the actual record data, they are forwarded to an Apache server acting as a reverse proxy. Apache uses the mod_auth_kerb module for Kerberos support and forwards the request to the repository (CouchDB) since CouchDB does not support authentication mechanisms other than usernames and passwords.

```
<Proxy *>
    Order deny,allow
    Deny from all
    Allow from cern.ch
</Proxy>
<Location />
    #SSLRequireSSL
    AuthType Kerberos
    AuthName "CERN Login"
    KrbMethodNegotiate On
    KrbMethodK5Passwd Off
    KrbAuthRealms CERN.CH
    Krb5KeyTab /etc/krb5.keytab
    KrbVerifyKDC Off
    KrbServiceName host/lxfsrd0714.cern.ch@CERN.CH
    require valid-user
</Location>
ProxyPass / http://localhost:5984/ retry=0 nocanon
ProxyPassReverse / http://localhost:5984/
RequestHeader unset Authorization
```

Figure 8: Apache configuration

As can be seen from the configuration file in Figure 8 above, Apache needs read access to the keytab file to authenticate the request with the Kerberos server. In this particular setup Apache only checks if the host requesting access is really who it claims to be. Authorization per machine should be implemented outside of Kerberos. The KrbMethodNegotiate parameter ensures that Apache searches for the Negotiate token inside the HTTP request. Reminder: this makes authentication without usernames and passwords (not suitable for a large number of machines running jobs) possible. With KrbMethodK5Passwd turned off, users cannot access the database with their username and password, meaning they cannot access CouchDB's web interface from a host that is not added to Kerberos. Once the request has been validated, Apache removes the request's Authorization header containing Kerberos information so CouchDB can receive a clean HTTP request CouchDB can process.

4 Repository

4.1 Databases

CouchDB provides a RESTful JSON API than can be accessed from any environment that allows HTTP requests. CouchDB's built in Web administration console (Futon) speaks directly to the database using HTTP requests issued from the browser. Apache CouchDB is a document-oriented database that can be queried and indexed using JavaScript in a MapReduce fashion. Every record received by CouchDB is stored as a JSON document. Since the queries executed represent complex questions, the records are sent as summaries (resource usage per user from the time the records were last sent and now) and detailed records (usage per command by the user).

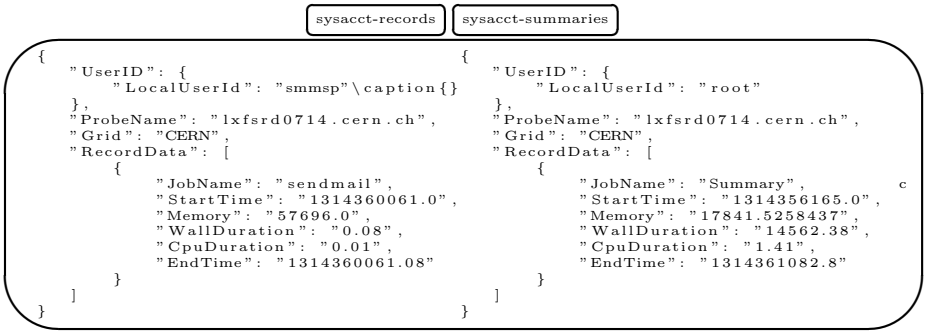


Figure 9: Database layout

The setup in Figure 9 has two databases: sysacct-summaries and sysacct-records. Every time the probe collects data it generates JSON files to be sent as HTTP requests to CouchDB. For every user that was active on the machine it sends two files. One is a summary that tells the overall resource usage and the other is a list of every command the user executed when he was active, the time at which the command was executed, the resource usage of the command etc.
One document is sent to be written to sysacct-summaries and the other to sysacct-records. This setup requires more storage space for JSON files but enables faster response times for certain queries. Let's look again at the questions the queries are supposed to answer:

1. Which commands did a user execute?
2. On which machines was he/she active?
3. What time was he/she active?
4. Where was this command executed, by whom and at what time?
5. What is the first and last time a user was active on a machine?
6. What time did he/she execute a command on a machine?

The first question requires us to know the exact names of commands executed so the queries have to be directed to the sysacct-records database since sysacct-summaries contains only summaries, not the specific information about commands.
Suppose a security breach has occured involving a certain user. A logical course of action would be to find on which machines and when the user was active to check if there has been tampering with those machines. One solution would be to execute a query on sysacct-records to go through all the JSON files and search for the machine names the user was active on. But the database has to search through a much larger collection of data that are irrelevant for this query (e.g. the commands the user executed). It is simpler to search through the summaries which have only one record (the summary) so the response is much faster.

4.2 Views

The CouchDB equivalent to SQL queries are views. Views are actually functions written mostly in Javascript that get executed every time the user wants to retrieve some information from the database. There are third party query servers for other languages like Python but since Javascript is part of CouchDB functions written in Javascript are still the fastest.

Query speed can be an issue with CouchDB, especially with large amounts of data that do not comply to the map/reduce principle. An explanation will be given in this document. CouchDB queries have two functions: a map and an optional reduce function. As opposed to SQL databases, CouchDB will always return an object representing a table with two attributes: a key and a value.

```
                    function(doc) {
                      emit(doc._id, doc);
                    }

function(doc) {
  if (doc.Type == "customer") {
    emit(doc.LastName, {FirstName: doc.FirstName});
    emit(doc.FirstName, {LastName: doc.LastName});
  }
}
```

Figure 10: Map functions

A view function should accept a single argument: the document object. To produce results, it should call the implicitly available emit(key, value) function. For every invocation of that function, a result row is added to the view (if neither the key nor the value are undefined). The rows in the computed table are updated automatically with any changes that have been made (additions, edits, or deletions) since the view was created.

Views are the primary tool used for querying and reporting on CouchDB documents. There are two different kinds of views: permanent and temporary views. Permanent views are stored inside special documents called design documents. Temporary views are not stored in the database, but rather executed

on demand.
NOTE: Temporary views are only good during development. Final code should not rely on them as they are very expensive to compute each time they are called and they get increasingly slower the more data you have in a database.

Every call to emit generates a row with a key and a value and the view is executed for every document in the database (not the entire CouchDB database but the JSON database the query is accessing). Suppose we have a database of JSON documents with every document representing a user with their home address and we want the number of users living in Geneva, Zurich etc. The easiest way to get the desired data is to write a view with a map function that generates a row with the name of the city as the key and a "1" as the value. The optional reduce function can then be used to group the keys and summarize all the 1's per key like in the graphical representation below.

key	value
"Geneva"	1
"Lyon"	1
"Geneva"	1
"Chamonix"	1

$\Longrightarrow$

key	value
"Geneva"	2
"Zurich"	1
"Chamonix"	1

Figure 11: Key grouping

Note that the keys and values can be more complex but the reduce function should only be used to lower the overall number of values. When choosing between a larger number of keys and a large number of values per key one should always choose the former. The performance issues detected when executing queries were mostly caused by attempts to retrieve neatly structured data from CouchDB without any need to go through the data and format it before presenting it as a report - NoSQL databases are ideally suited to perform map/reduce calculations that reduce the number of values but do not fare so well with complex data structures.

```
# User was active on machines X,Y,Z... (summaries)

def fun(doc):
  for record in doc["RecordData"]:
    yield [doc["UserID"]["LocalUserId"],doc["ProbeName"]],None

def fun(keys, values):
  return None
```

Figure 12: View functions with more complex results

This particular view (Figure 12) was written in Python by using a third party view server and answers the question on which machines the user was active. WARNING: The Python view server is much slower and is not maintained by CouchDB so languages other than Javascript should be used only when speed is not an issue. In this example the database returns a JSON object representing a table with several rows and each one containing a list and the None Python

value. This generates a large amount of rows but is several times faster than putting the names of the machines as the value and then reducing them to a single value for one user before giving an output. The database will always be occupied with writing the JSON documents because of the large number of nodes available in clusters with each one frequently sending data. The external program that executes the views should do most of the work of formatting the data and the database should only take care of delivering all the necesarry data to the program as fast as possible.

```
# Command was executed by user X on machine Y at time Z (records)
def fun(doc):
  if doc["ProbeName"] and doc["RecordData"] and doc["UserID"]:
    for command in doc["RecordData"]:
      yield [command["JobName"],doc["UserID"]["LocalUserId"],doc["ProbeName"]],
            command["StartTime"]

def fun(keys, values):
  return values
```

Figure 13: Map and reduce functions written in Python

The values in Figure 13 represent timestamps of the commands that were executed. This is slow and inefficient because it generates keys with an enormous number of values grouped together. Although this is closer to the final representation, a faster solution is to put all the data inside keys and format it afterwards.

key	**value**
["sendmail", "ssmp", "spock.cern.ch", "1333333333"]	null
["sh", "root", "kirk.cern.ch", "1333333334"]	null
["sh", "root", "spock.cern.ch", "1333333335"]	null

Figure 14: Return results with complex structures as keys

Another issue are ranges. As opposed to SQL queries, views are static and cannot dynamically change the parameters inside their functions. An inefficient but possible solution would be to create permanent views for the most frequently requested ranges and use temporary views for the rest. Since the keys generated by view functions are sorted, a more viable solution for the program executing views would be to send key ranges as HTTP url parameters. An example with Bash and curl is given in Figure 15.

```
$ curl -X GET 'http://localhost:5984/sysacct_records/_design/
commands/_view/exectimes?startkey=\["sendmail"\]\&endkey=\["sh", \{\}]'
```

Figure 15: View ranges

4.3 Indexing

CouchDB uses a data structure called a B-tree to index its documents and views. Only documents created after the last time a certain view was executed are indexed. This represents a problem when the views are executed infrequently and large amounts of data are written to the database between view executions, like in lxplus and lxbatch clusters, because CouchDB indexes documents before returning the requested data. To alleviate this a few different possibilities have been suggested. Most rely on registering with CouchDB's update notifications and triggering reads automatically.

```
...
class ViewUpdater(object):
    # The smallest amount of changed documents before the views are updated
    MIN_NUM_OF_CHANGED_DOCS = 50

    # Set the minimum pause between calls to the database
    PAUSE = 5 # seconds

    # URL to the DB on the CouchDB server
    URL = "http://localhost:5984"

    # One entry for each design document
    # in each database
    VIEWS = {
        'sysacct_records': {
            'commands': [
                'exectimes',
                # ...
            ]
        }
    }
...
```

Figure 16: Periodic view regeneration

Views are stored as design documents inside the database along with other JSON documents. Because indexing is performed on every view in a document, views should be stored inside different design documents to avoid a possible slowdown.

5 Conclusion

During development and testing several conclusion have been made regarding the data collection and report generation.

As the number of machines sending records grows so does the load on the repository (CouchDB). Albeit this is rarely a problem in relatively small clusters, CERN clusters are large and could potentially crash the database server. The script started periodically by cronjob should not be executed on the hosts at the same time.

The probe is written in Python and uses the kerberos module to authenticate with Apache. The module and especially its usage with GSSAPI and HTTP Negotiate is poorly documented and the projects using it should get involved with writing the documentation. A good starting point are examples written during development of the psacct probe.

Psacct data is collected from the output of the dump-acct command. The probe expects that output to be of a specific format which is different on other Linux platforms but it should be fairly easy to make the probe compatible with other platforms.

Response time of database queries can slow down to a halt if the view functions are poorly written and if languages other than the native Javascript are used. These guidelines should be followed to get maximum response times from the database:

1. Reduce functions should be used to reduce the number of values and when choosing between adding more load to the repository or local desktop computers retrieving data during audits, the latter should always be chosen since most of the time those computers are idle

2. It is better to retrieve a large number of keys than a smaller number of keys with large numbers of different values per key

3. Indexes should be regenerated as often as possible to avoid slowdowns during queries, either by calling views with external scripts or with update notifications

4. Views should be written in separate documents to avoid another slowdown because of too many views being indexed at the same time which are not really needed

In general, NoSQL databases require a very different approach from SQL databases but are ideally suited for data that do not follow a specific format like SQL tables do. This makes it easier to expand the probe to send additional data and not just psacct records. The architecture described in this document can be used for any kind of accounting collection as long as there is an external authentication mechanism or until NoSQL databases are expanded with other authentications like Kerberos. For a completely trusted environment this is not needed and data can be sent directly to the repository.

References

[1] J. Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: The Definitive Guide*, 1st edition. Sebastopol, Canada: O'Reilly Media Inc., 2010.

[2] Migeon, Jean-Yves. *The MIT Kerberos Administrator's How-to Guide*, 1st edition. Massachusetts, USA: MIT Kerberos Consortium, 2008.

[3] Open Science Grid. *OSG Accounting Twiki*. Fermilab (2011). https://twiki.grid.iu.edu/bin/view/Accounting/WebHome

[4] Paltomaki, Atte. *Setting up a Kerberos proxy with Apache*. modauthkerb-help@lists.sourceforge.net, (November 24th, 2011). http://permalink.gmane.org/gmane.comp.apache.mod-auth-kerb.general/2164

[5] Orton, Joe. *Kerberos and Single Sign-On With HTTP*. ApacheCon (2008). http://eu.apachecon.com/eu2008/program/materials/kerb-sso-http.pdf

[6] Radez, Dan. *python + kerberos + apache GSSAPI Example*. JADDOG (July 6th, 2009). http://www.jaddog.org/2009/07/06/python-kerberos-kinit-apache-gssapi-example