Process accounting collection

Goran Cetušić
Supervisor: Ricardo da Silva

University of Zagreb
CERN Openlab

September 1, 2011

CERN openlab

Openlab project

Original project name:
Review process accounting

sysacct

- Existing software:
  - modified version of psacct (CERN-cc-sysacct)
  - Bash scripts
  - AFS repository

sysacct

- Existing software:
  - modified version of psacct (CERN-cc-sysacct)
  - Bash scripts
  - AFS repository
- Project task:
  1. collect psacct data
  2. send to central repository
  3. generate reports

sysacct

- Existing software:
  - modified version of psacct (CERN-cc-sysacct)
  - Bash scripts
  - AFS repository
- Project task:
  1. collect psacct data
  2. send to central repository
  3. generate reports
- Research and solutions:

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

Objective

- Final objective is getting data from a central repository
- Some of the basic questions the queries should 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?
  7. ...
- The queries will be executed rarely
  - Mostly when security incidents occur
  - Or daily to generate activity reports

Collector

Phase I - collecting data

- Write code/collect data
- Existing solutions?
  - OSG Gratia

Phase I - collecting data

- Write code/collect data
- Existing solutions?
  - OSG Gratia
- Gratia:
  - Collector (Java)
  - Probes (Python)
    - condor, psacct, hadoop, dcache...

1. Probe (Python)
   - XML - ugly, overhead
   - Custom protocol
   - Only summaries

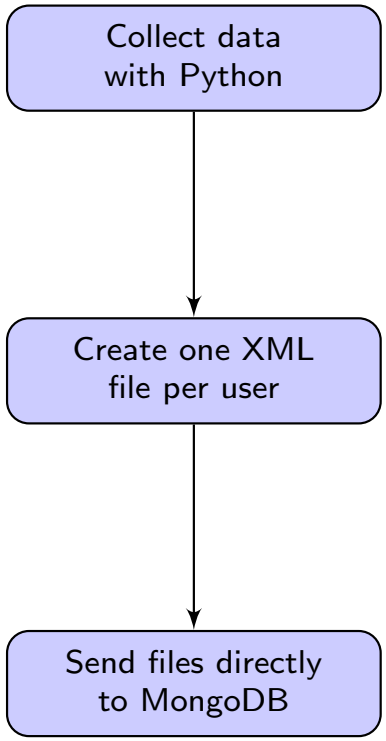
Phase I - collecting data

- Write code/collect data
- Existing solutions?
  - OSG Gratia
- Gratia:
  - Collector (Java)
  - Probes (Python)
    - condor, psacct, hadoop, dcache...

1. Probe (Python)
   - XML - ugly, overhead
   - Custom protocol
   - Only summaries
2. Repository
   - ~~Gratia Java collector~~
   - NoSQL $\surd$

First draft

- Guideline:
  - reuse existing Gratia code
- First problem:
  - Collector $\Longrightarrow$ XML
  - MongoDB $\Longrightarrow$ JSON
- Solution:
  - Collector $\Longrightarrow$ JSON
  - MongoDB $\Longrightarrow$ JSON



Accounting and log file handling

Accounting and log file handling

- When the collector starts it:
  - Moves current files for processing
  - Starts accounting on a new file

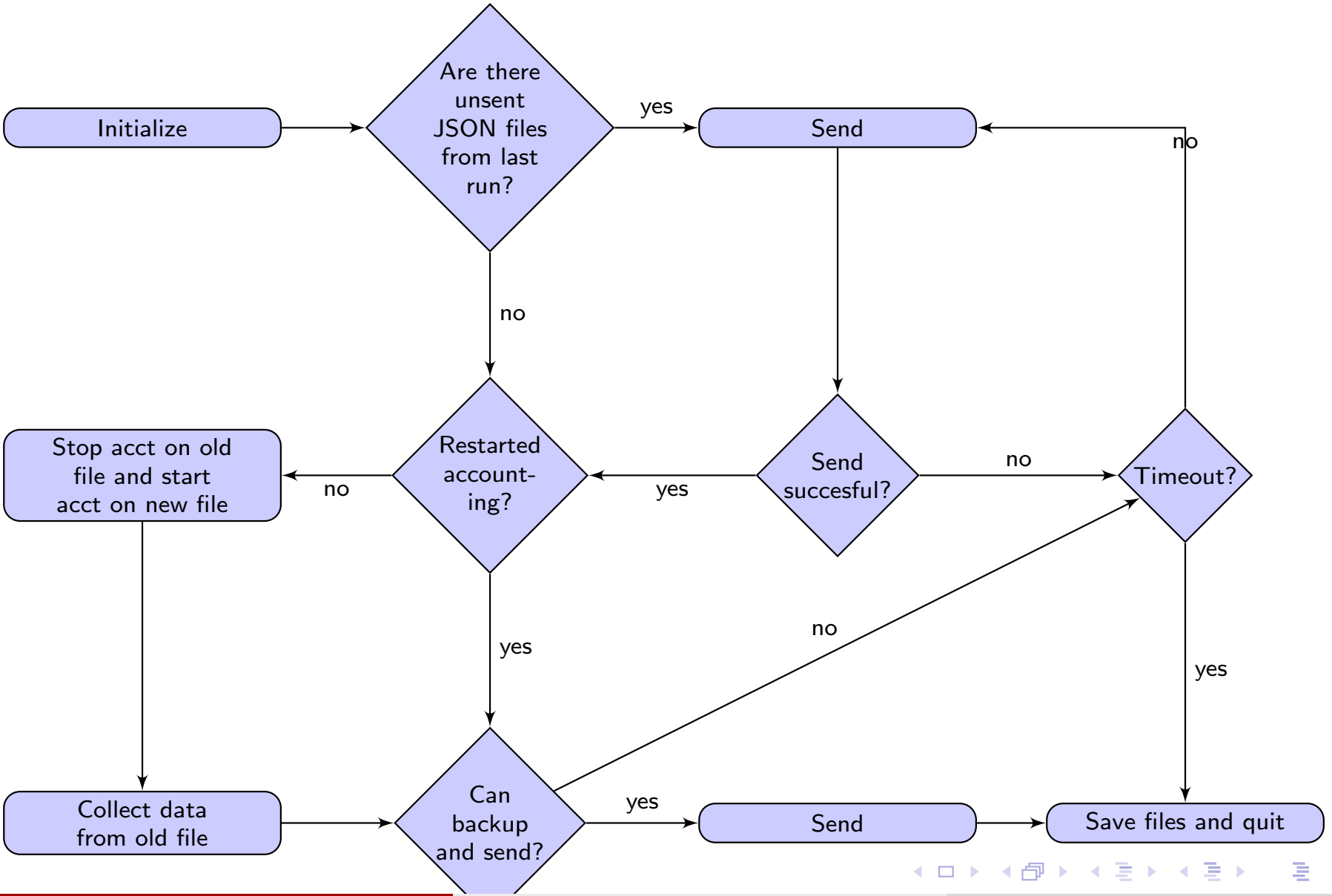
Accounting and log file handling

- When the collector starts it:
  - Moves current files for processing
  - Starts accounting on a new file
- If the sending of files fails:
  - The files are stored locally
  - The collector will try to resend them next time

Accounting and log file handling

- When the collector starts it:
  - Moves current files for processing
  - Starts accounting on a new file
- If the sending of files fails:
  - The files are stored locally
  - The collector will try to resend them next time
- Logrotate used to restart default accounting
  - Some old logrotate scripts may still do that
- Collector has it's own rotation mechanism
- New log is generated every day
- Sent files are stored locally for 90 days

Code


Initialize
Are there unsent JSON files from last run?
yes
Send
no
no
Restarted account-ing?
Stop acct on old file and start acct on new file
no
yes
Send succesful?
no
Timeout?
yes
no
yes
Collect data from old file
Can backup and send?
yes
Send
Save files and quit

Phase II - security: authentication

- Large problem - NoSQL databases have no real network security
  - MongoDB - only user/pass without encryption
  - CouchDB - user/pass with encryption in newest versions
  - ... - None(?) have Kerberos

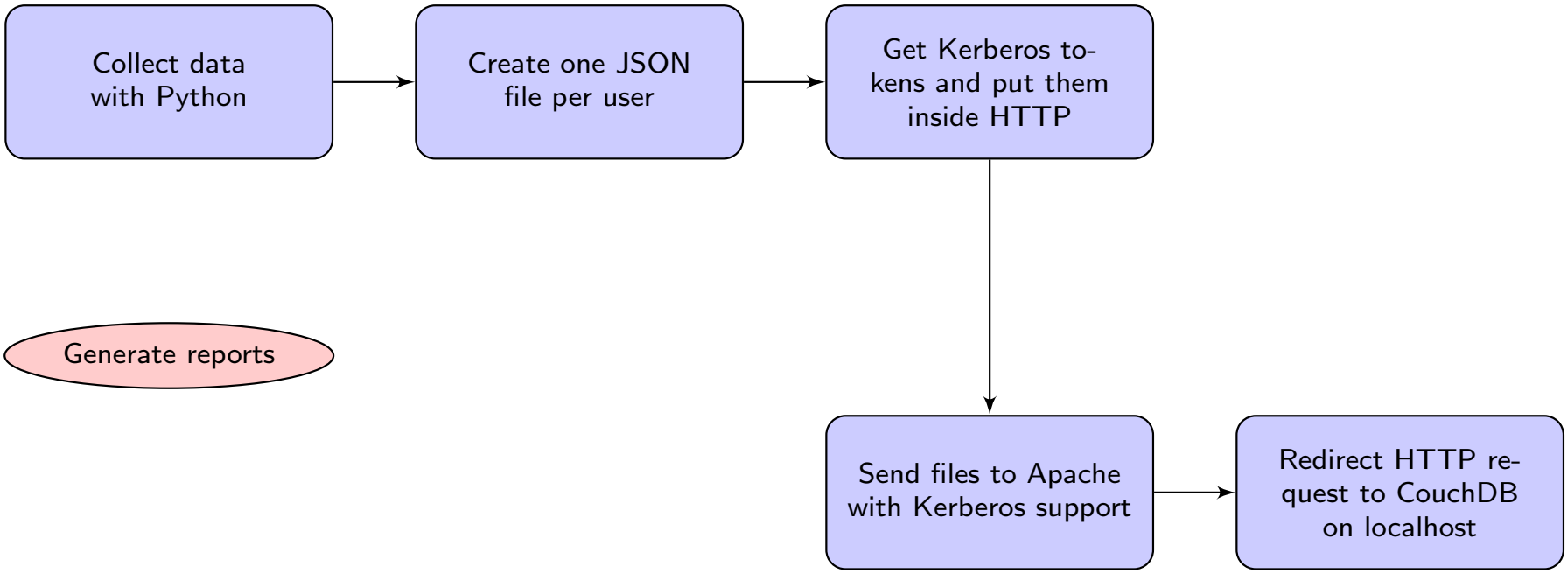
Phase II - security: authentication

- Large problem - NoSQL databases have no real network security
  - MongoDB - only user/pass without encryption
  - CouchDB - user/pass with encryption in newest versions
  - ... - None(?) have Kerberos
- Possible solution - reverse proxy with Kerberos support
  - ActiveMQ
  - Apache $\surd$

Phase II - security: authentication

- Large problem - NoSQL databases have no real network security
  - MongoDB - only user/pass without encryption
  - CouchDB - user/pass with encryption in newest versions
  - ... - None(?) have Kerberos
- Possible solution - reverse proxy with Kerberos support
  - ActiveMQ
  - Apache $\surd$
- Another problem:
  - MongoDB $\Longrightarrow$ custom transport protocol
  - mod_auth_kerb $\Longrightarrow$ HTTP
- "Quick" rewrite of the collector transport mechanism
  - CouchDB HTTP with Kerberos instead of MongoBD

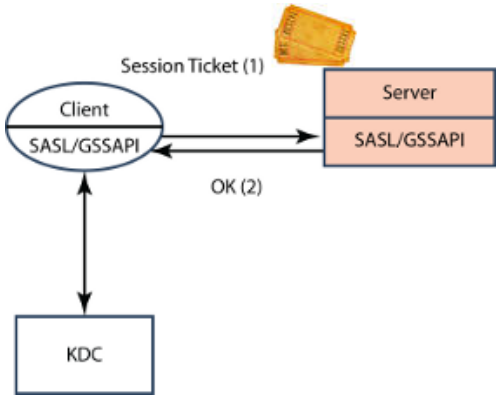
Final architecture

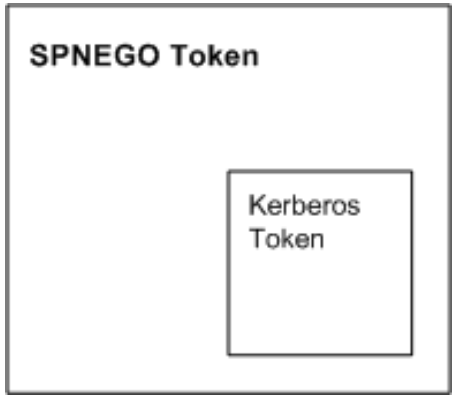


Kerberos+Apache

SPNEGO

- HTTP Negotiate header that uses GSSAPI with Kerberos tokens





1. Get a Kerberos service token
2. Wrap the token inside GSSAPI
3. Create a HTTP request filled with JSON records
4. Put the GSSAPI token inside the HTTP Negotiate header
5. Send the request to Apache

NOTE: The client connecting to Apache must already have a Kerberos ticket in its cache that is generated by calling "kinit -k" or some other command. It does not use user/pass authentication.

Proxy configuration

```
<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
```

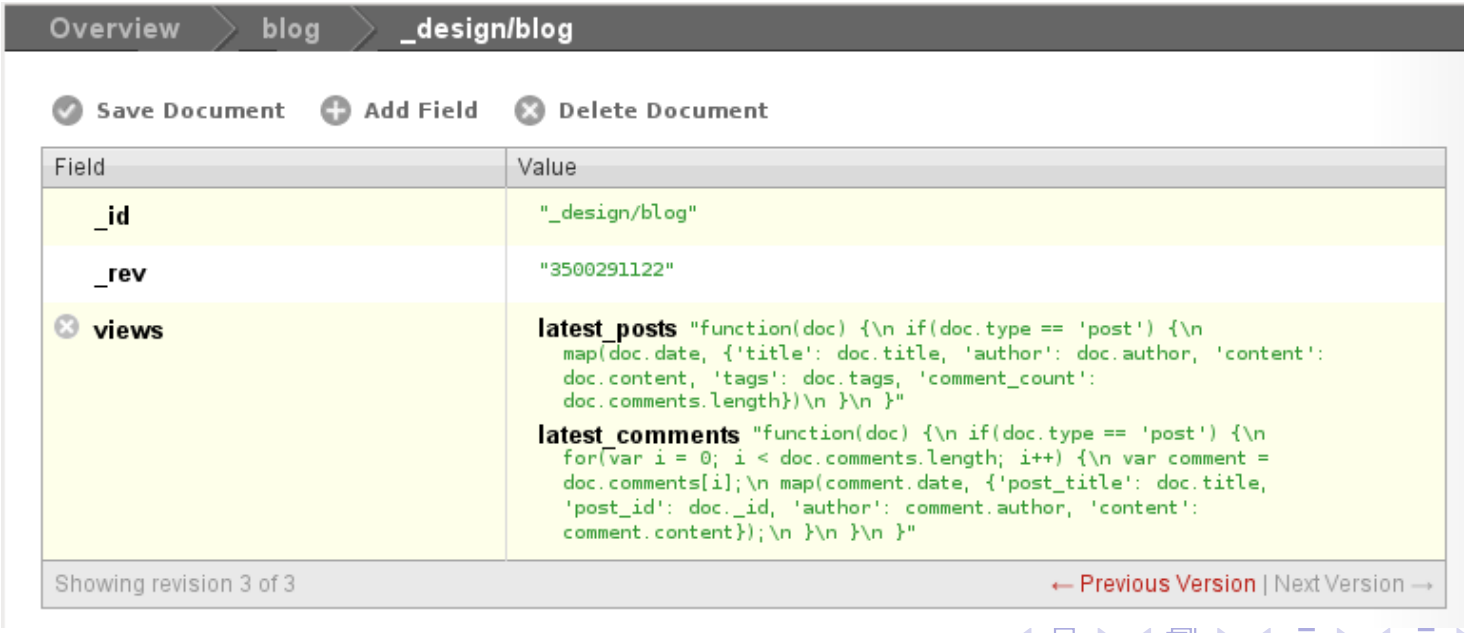
- The keytab must be readable by the user running the httpd daemon
- httpd must be enabled in SELinux

CouchDB

Retrieving data

- Any kind of query to CouchDB is represented as a view
- There are two different kinds of views:
  - Permanent views - stored inside special design documents
  - Temporary views - not stored in the database, but executed on demand
- Permanent views are stored in CouchDB design documents whose id begins with _design/ e.g. views for a blog are stored in _design/blog

NOTE: Temporary views are not adequate for production because they're very expensive to compute each time they're called



View functions

- Each database in CouchDB can store multiple design documents
  - The views inside a design document are executed only for documents inside that particular database
- The view is defined by a mandatory JavaScript map function that maps keys to values

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

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

- It is possible to use other languages than JavaScript by plugging in third-party view servers

map/reduce

- If a view has the optional reduce function, it is used to produce aggregate results for that view

```
function(doc) {
  emit(doc.machine, doc.cputime);
}
```

```
function (key, values) {
    return sum(values);
}
```

- Keys can be grouped (group=true)
  - Reduce function summarizes values of rows that share the same key

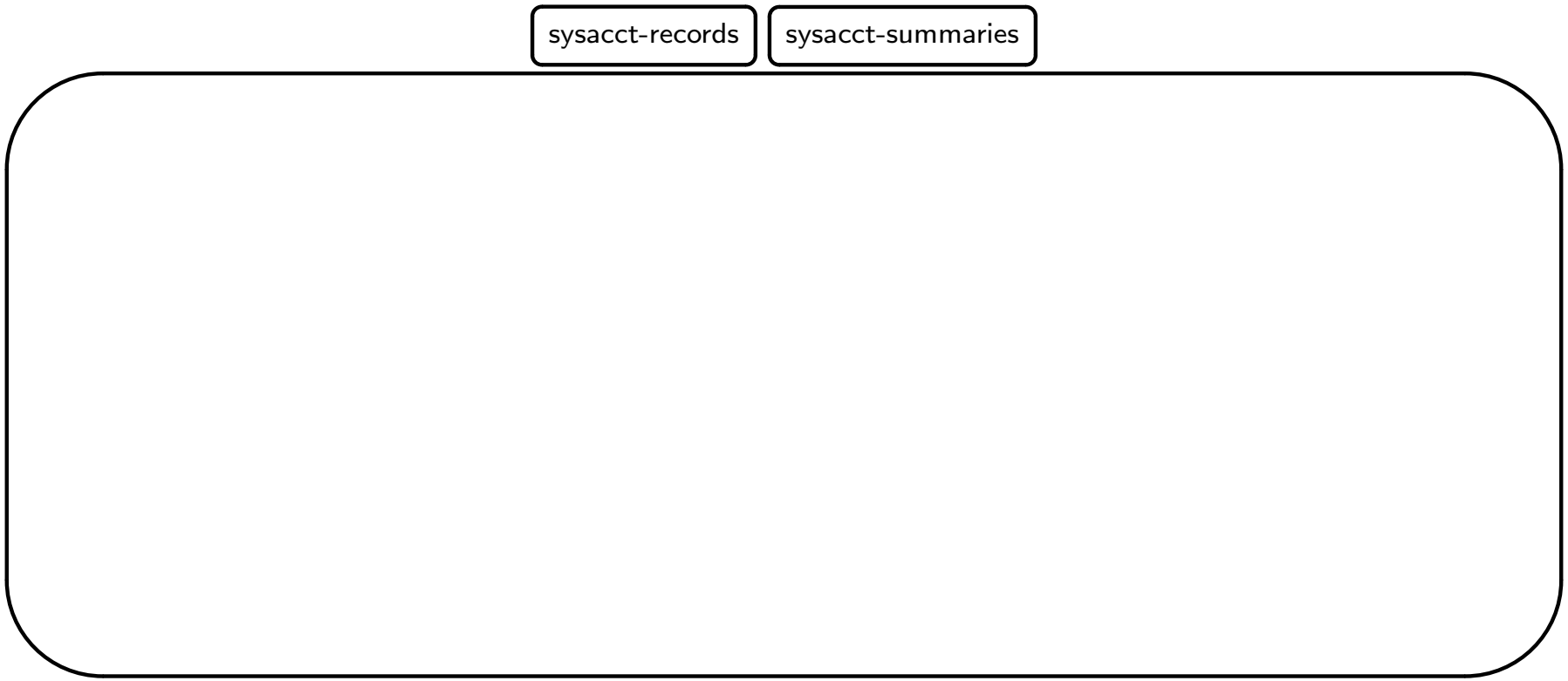
key	value
"spock.cern.ch"	2
"kirk.cern.ch"	4
"spock.cern.ch"	7
"picard.cern.ch"	8

$\Longrightarrow$

key	value
"spock.cern.ch"	9
"kirk.cern.ch"	4
"picard.cern.ch"	8

- Reducing and it's grouping mechanism can be set to false
  - Without grouping the reduce function above does nothing

Documents



Documents

sysacct-records | sysacct-summaries


```
{
    "UserID": {
        "LocalUserId": "smmsp"
    },
    "ProbeName": "lxfsrd0714.cern.ch",
    "Grid": "CERN",
    "RecordData": [
        {
            "JobName": "sendmail",
            "StartTime": "1314360061.0",
            "Memory": "57696.0",
            "WallDuration": "0.08",
            "CpuDuration": "0.01",
            "EndTime": "1314360061.08"
        }
    ]
}
```

Documents

sysacct-records

```
{
    "UserID": {
        "LocalUserId": "smmsp"
    },
    "ProbeName": "lxfsrd0714.cern.ch",
    "Grid": "CERN",
    "RecordData": [
        {
            "JobName": "sendmail",
            "StartTime": "1314360061.0",
            "Memory": "57696.0",
            "WallDuration": "0.08",
            "CpuDuration": "0.01",
            "EndTime": "1314360061.08"
        }
    ]
}
```

sysacct-summaries

```
{
    "UserID": {
        "LocalUserId": "root"
    },
    "ProbeName": "lxfsrd0714.cern.ch",
    "Grid": "CERN",
    "RecordData": [
        {
            "JobName": "Summary",          c
            "StartTime": "1314356165.0",
            "Memory": "17841.5258437",
            "WallDuration": "14562.38",
            "CpuDuration": "1.41",
            "EndTime": "1314361082.8"
        }
    ]
}
```

Documents

sysacct-records | sysacct-summaries


```
{
    "UserID": {
        "LocalUserId": "smmsp"
    },
    "ProbeName": "lxfsrd0714.cern.ch",
    "Grid": "CERN",
    "RecordData": [
        {
            "JobName": "sendmail",
            "StartTime": "1314360061.0",
            "Memory": "57696.0",
            "WallDuration": "0.08",
            "CpuDuration": "0.01",
            "EndTime": "1314360061.08"
        }
    ]
}
```

```
{
    "UserID": {
        "LocalUserId": "root"
    },
    "ProbeName": "lxfsrd0714.cern.ch",
    "Grid": "CERN",
    "RecordData": [
        {
            "JobName": "Summary",
            "StartTime": "1314356165.0",
            "Memory": "17841.5258437",
            "WallDuration": "14562.38",
            "CpuDuration": "1.41",
            "EndTime": "1314361082.8"
        }
    ]
}
```


- Two databases:
  - sysacct_records - detailed information about commands
  - sysacct_summaries - summarized information after each collection

Queries

- Some of the basic questions the queries should answer:

Queries

- Some of the basic questions the queries should answer:
  1. Which commands did a user execute?

Queries

- Some of the basic questions the queries should answer:
  1. Which commands did a user execute?
  2. On which machines was he/she active?

Queries

- Some of the basic questions the queries should answer:
  1. Which commands did a user execute?
  2. On which machines was he/she active?
  3. What time was he/she active?

Queries

- Some of the basic questions the queries should 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?

Queries

- Some of the basic questions the queries should 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?

Queries

- Some of the basic questions the queries should 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?

Queries

- Some of the basic questions the queries should 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?
- Some of the queries (2,3,5) can be done on both summaries or records, some (1,4,6) only on records
  - Command names
  - Activity time range
- Any kind of query that requires information for individual commands has to use the record documents

Views

- Let's start with number 2

```
# 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
```

Views

- Let's start with number 2

```
# 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
```

- Q: Why do we need the reduce function?

Views

- Let's start with number 2

```
# 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
```

- Q: Why do we need the reduce function?
- Summaries are generated for a user on a machine at 10pm, 11pm...
  - The view is called for every document in the database
  - yield generates keys for every document (keys are not unique)

Views

- Let's start with number 2

```
# 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
```

- Q: Why do we need the reduce function?
- Summaries are generated for a user on a machine at 10pm, 11pm...
  - The view is called for every document in the database
  - yield generates keys for every document (keys are not unique)
- A: For grouping

key	value
["root", "spock.cern.ch"]	null
["root", "kirk.cern.ch"]	null
["root", "spock.cern.ch"]	null
["smmsp", "spock.cern.ch"]	null

Views

- Let's start with number 2

```
# 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
```

- Q: Why do we need the reduce function?
- Summaries are generated for a user on a machine at 10pm, 11pm...
  - The view is called for every document in the database
  - yield generates keys for every document (keys are not unique)
- A: For grouping

key	value
["root", "spock.cern.ch"]	null
["root", "kirk.cern.ch"]	null
["root", "spock.cern.ch"]	null
["smmsp", "spock.cern.ch"]	null

$\Longrightarrow$

key	value
["root", "spock.cern.ch"]	null
["root", "kirk.cern.ch"]	null
["smmsp", "spock.cern.ch"]	null

Time ranges

- Where was this command executed, by whom and at what time?
  - Not nearly as easy to figure out as it seems
- If we want to get information for a particular command:
  - The command name has to be the first part of the key so it can be searched

```
# Command was executed by user X on machine Y at time Z (records)
# View: commands/exectimes
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["Sta
```

Time ranges

- Where was this command executed, by whom and at what time?
  - Not nearly as easy to figure out as it seems
- If we want to get information for a particular command:
  - The command name has to be the first part of the key so it can be searched

```
# Command was executed by user X on machine Y at time Z (records)
# View: commands/exectimes
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["Sta
```

- Again, there will be duplicate keys

Time ranges

- Where was this command executed, by whom and at what time?
  - Not nearly as easy to figure out as it seems
- If we want to get information for a particular command:
  - The command name has to be the first part of the key so it can be searched

```
# Command was executed by user X on machine Y at time Z (records)
# View: commands/exectimes
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["Sta
```

- Again, there will be duplicate keys
- Should we group to get unique keys?

Time ranges

- Where was this command executed, by whom and at what time?
  - Not nearly as easy to figure out as it seems
- If we want to get information for a particular command:
  - The command name has to be the first part of the key so it can be searched

```
# Command was executed by user X on machine Y at time Z (records)
# View: commands/exectimes
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["Sta
```

- Again, there will be duplicate keys
- Should we group to get unique keys?
- Is it better to have larger values or more keys?
  - Should the timestamps be part of the key or values?

DB output

- "Grow tall, not wide"
  - Reduce function should only be used to get a smaller number of values
- This is slow and inefficient:

```
# Command was executed by user X on machine Y at time Z (records)
# View: commands/exectimes
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
```

DB output

- "Grow tall, not wide"
  - Reduce function should only be used to get a smaller number of values
- This is slow and inefficient:

```
# Command was executed by user X on machine Y at time Z (records)
# View: commands/exectimes
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
```

- Option: get all the keys and format output by external scripts

Searches

- Row keys are sorted

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

Searches

- Row keys are sorted

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

- How would we search for the command sh?

Searches

- Row keys are sorted

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

- How would we search for the command sh?

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

Searches

- Row keys are sorted

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

- How would we search for the command sh?

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

- This will get us all the keys with "sh"
- For further filtering additional help is needed

Searches

- Row keys are sorted

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

- How would we search for the command sh?

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

- This will get us all the keys with "sh"
- For further filtering additional help is needed
  - A Python script that iterates through every key and creates a dictionary
  - Or we could use Bash directly with curl...

Local processing

- Questions arise:

Local processing

- Questions arise:
  - Is it better to give more work to the script or the database?

Local processing

- Questions arise:
  - Is it better to give more work to the script or the database?
  - Should the script get a smaller/larger number of non-unique keys?

Local processing

- Questions arise:
  - Is it better to give more work to the script or the database?
  - Should the script get a smaller/larger number of non-unique keys?
  - How should the final output be formatted?

Local processing

- Questions arise:
  - Is it better to give more work to the script or the database?
  - Should the script get a smaller/larger number of non-unique keys?
  - How should the final output be formatted?
- Guideline:
  - Database will always have a high load, query scripts on desktop computers should do as much work as possible

Local processing

- Questions arise:
  - Is it better to give more work to the script or the database?
  - Should the script get a smaller/larger number of non-unique keys?
  - How should the final output be formatted?
- Guideline:
  - Database will always have a high load, query scripts on desktop computers should do as much work as possible
- Calculated decisions (could be the wrong ones):

Local processing

- Questions arise:
  - Is it better to give more work to the script or the database?
  - Should the script get a smaller/larger number of non-unique keys?
  - How should the final output be formatted?
- Guideline:
  - Database will always have a high load, query scripts on desktop computers should do as much work as possible
- Calculated decisions (could be the wrong ones):
  - Make the queries as specific as possible (don't get everything!)

Local processing

- Questions arise:
  - Is it better to give more work to the script or the database?
  - Should the script get a smaller/larger number of non-unique keys?
  - How should the final output be formatted?
- Guideline:
  - Database will always have a high load, query scripts on desktop computers should do as much work as possible
- Calculated decisions (could be the wrong ones):
  - Make the queries as specific as possible (don't get everything!)
  - Choose more keys per value over less values per key (parse it later)

Local processing

- Questions arise:
  - Is it better to give more work to the script or the database?
  - Should the script get a smaller/larger number of non-unique keys?
  - How should the final output be formatted?
- Guideline:
  - Database will always have a high load, query scripts on desktop computers should do as much work as possible
- Calculated decisions (could be the wrong ones):
  - Make the queries as specific as possible (don't get everything!)
  - Choose more keys per value over less values per key (parse it later)
  - Avoid using reduce functions only for grouping (do it in the script)

Local processing

- Questions arise:
  - Is it better to give more work to the script or the database?
  - Should the script get a smaller/larger number of non-unique keys?
  - How should the final output be formatted?
- Guideline:
  - Database will always have a high load, query scripts on desktop computers should do as much work as possible
- Calculated decisions (could be the wrong ones):
  - Make the queries as specific as possible (don't get everything!)
  - Choose more keys per value over less values per key (parse it later)
  - Avoid using reduce functions only for grouping (do it in the script)
- Retrieval+presentation of data is achieved by a combination of views inside the database and scripts that call the views

Indexing

- The first time a view is executed CouchDB indexes results in a B-tree
  - It can take a long time for the first call to return results
  - Subsequent calls are much faster because a B-tree exists
- Our views are going to be rarely executed
  - We can update our B-tree periodically (warm up the views)

```
class ViewUpdater(object):
```

Indexing

- The first time a view is executed CouchDB indexes results in a B-tree
  - It can take a long time for the first call to return results
  - Subsequent calls are much faster because a B-tree exists
- Our views are going to be rarely executed
  - We can update our B-tree periodically (warm up the views)

```
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"
```

Indexing

- The first time a view is executed CouchDB indexes results in a B-tree
  - It can take a long time for the first call to return results
  - Subsequent calls are much faster because a B-tree exists
- Our views are going to be rarely executed
  - We can update our B-tree periodically (warm up the views)

```
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',
                # ...
            ]
        }
    }
```

Conclusion

- Most NoSQL databases have no security
- Debian distributions use a different psacct format
- Python documentation for Kerberos is obscure
- Reduce functions should reduce
- Grow tall not wide
- Views written in JavaScript are the fastest
- Views should be prewarmed
- Design documents should have less views
- Collector can be expanded to collect other data
- NoSQL databases require a very different approach

Questions?