Data Aggregation System Documentation

Release development

Valentin Kuznetsov

February 06, 2014

Contents

- **Version**: development
- **Last modified**: February 06, 2014

CHAPTER 1

Contents:

1.1 Introduction

DAS is an opensource virtual data service integration platform. The acronym DAS stands for *Data Aggregation System*. It provides a a common layer above various data services, allowing end users to query one (or more) of them with a more natural, search-engine-like interface. It is being developed for the [CMS] High Energy Physics experiment on the [LHC], at CERN, Geneva, Switzerland. Although it is currently only used at the CMS experiment, it was designed to have a general-purpose architecture which would be applicable to other domains.

It provides several features:

1. a caching layer for underlying data-services
2. a common meta-data look up tool for these services
3. an aggregation layer for that meta-data

The main advantage of DAS is a uniform meta-data representation and consequent ability to perform look-ups via free text-based queries. The DAS internal data-format is JSON. All meta-data queried and stored by DAS are converted into this format, according to a mapping between the notation used by each data service and a set of common keys used by DAS.

1.2 How DAS works?

1.2.1 Resolving queries over data services

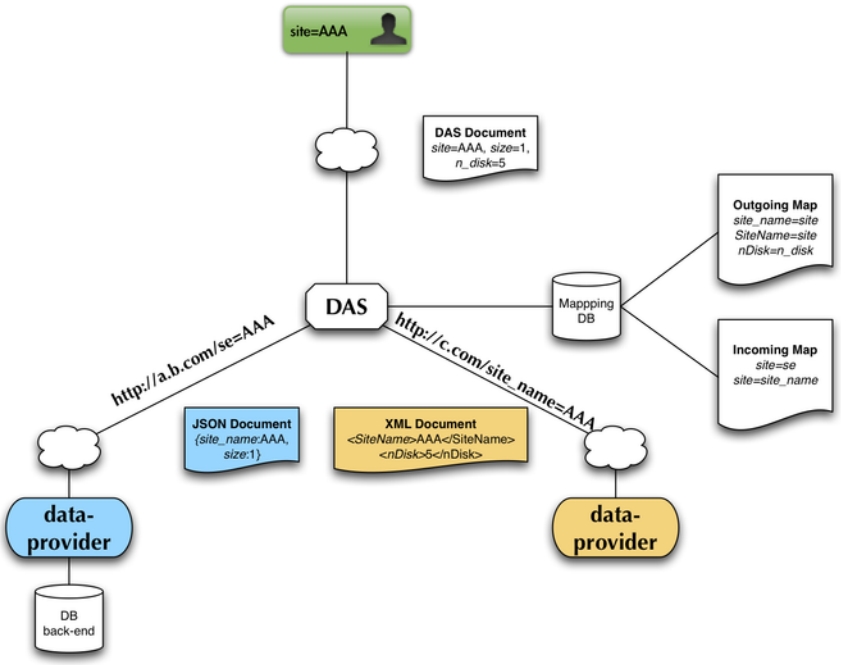
This diagram illustrates the request flow triggered by a user query.

The user makes the query *site=AAA*. DAS resolves it into several requests, by looking at the *daskeys* map for each available data service, which indicates that two services understand an input key *site*, which is transformed into queries for each of those services with parameter *se* and *site_name*.

DAS then makes the API calls (assuming the data isn't already available).

- *http://a.b.com/se=AAA*
- *http://c.com/site_name=AAA*

and retrieves the results, which are re-mapped into DAS records according to the *notation* map for each of those services.



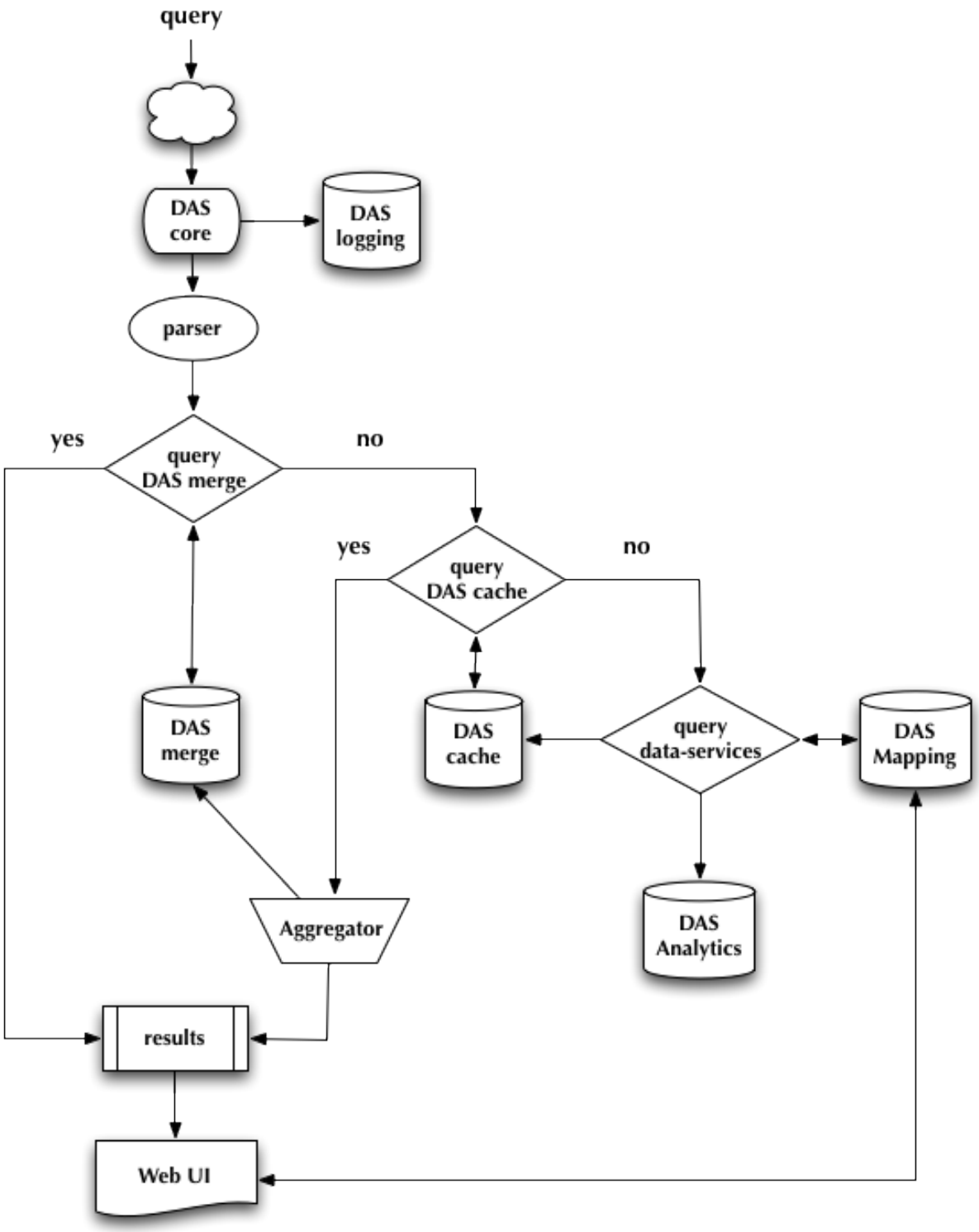
1.2.2 Query processing steps

DAS workflow consists of the following steps:

- parse input query
  - look-up for a superset of query conditions in DAS merge cache (ie, have we recently handled a query guaranteed to have returned the data being requested now)
    - if unavailable
      - query raw cache
      - query services, to get the records that aren't available
      - write new records to raw cache
      - perform aggregation
      - write aggregation results to merged cache
    - get results from merged cache
- present DAS records in Web UI (converting records if necessary)

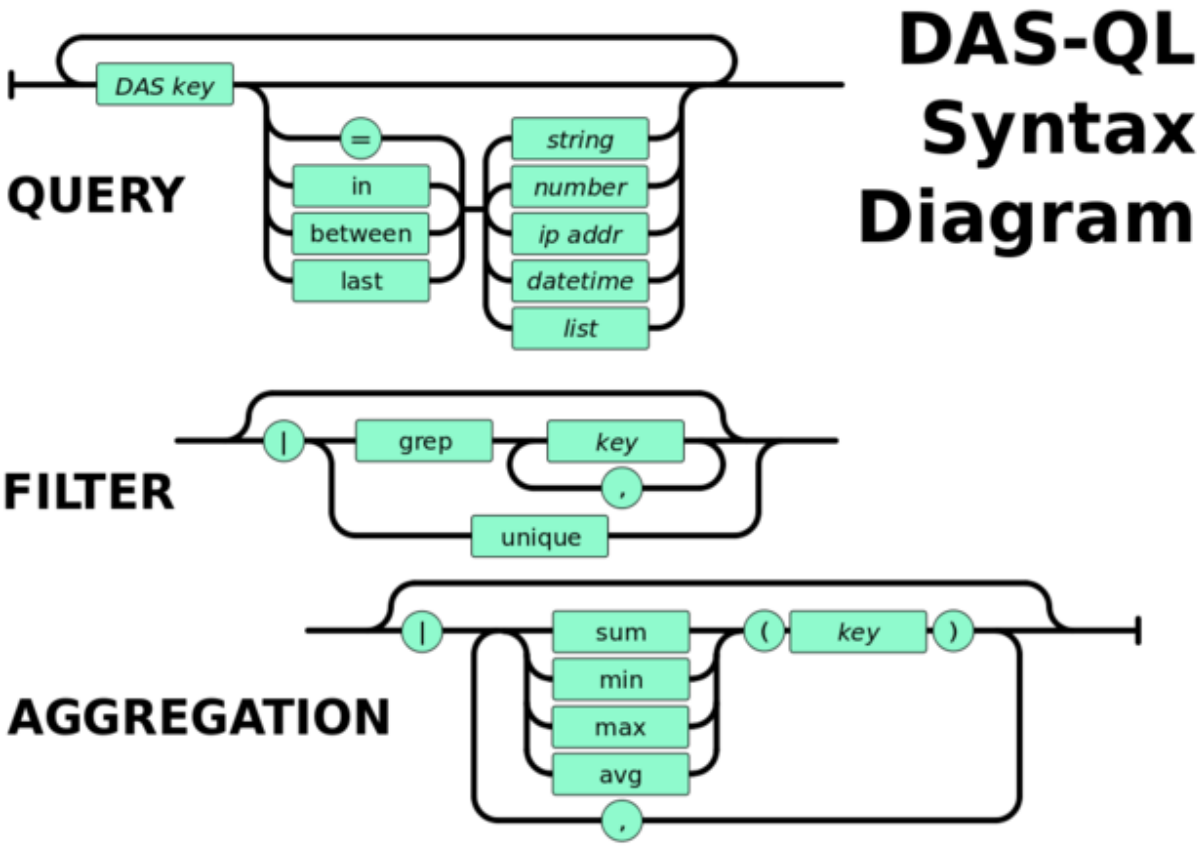
1.3 How to use DAS?

DAS can be used from web interface, through a command line interface, or accessed programically from other applications.


query
DAS core
DAS logging
parser
yes
query DAS merge
no
yes
query DAS cache
no
DAS merge
DAS cache
query data-services
DAS Mapping
Aggregator
DAS Analytics
results
Web UI

1.3.1 DAS Query Language

DAS Query Language (DAS-QL) provides intuitive, easy to use text based queries to underlying data-services. Its syntax is represneted in the following diagram



We can represent it via set of pipes, similar to UNIX pipes, the key look-up fields followed by filter and/or aggregator:

```
<key> <key> = <value> ... | <filter> <key.att> <op> <value>, ... | <aggregator>, ...
<key> date last <value h|m>
<key> date between [YYYYMMDD, YYYYMMDD]
```

Here the <*key*> is a valid DAS key, e.g. *dataset, run* and <*op*> is a valid DAS operator, e.g. <, >, <=, >=. The <*key.att*> represents a key attribute. They are deduced form the DAS records. For instance, if file record contain *size, id, md5 fields*, all of them can be used as attributed of the *file* key, e.g. *file.md5*. But the attributes cannot appear in look-up part of the query.

The DAS query can be supplemented either by filters or aggregator functions. The filter consists of filter name and *key/key.attributes* value pair or *key/key.attributes* fields. The support filters are: *grep, unique, sort*. To *unique* filter does not require a *key*.

The supported aggregator functions are: *sum, count, min, max, avg, median*. A wild-card patterns are supported via asterisk character.

Here is just a few valid examples how to construct DAS query using file and dataset keys and file.size, dataset.name key attributes:

```
file dataset=/a/b/c | unique
file dataset=/a/b/c | grep file.name, file.size
file dataset=/a/b/c | sort file.name
file dataset=/a/b/c | grep file.name, file.size, | sort file.size
file dataset=/a/b/c | sum(file.size)
file dataset=/a/b/c | grep file.size>1 | sum(file.size)
file dataset=/a/b*
```

Special keywords

DAS has a several special keywords: *system, date, instance, records*.

- The *system* keyword is used to retrieve a records only from specified system (data-service), e.g. DBS.
- The *date* can be used in different queries and accepts values in YYYYMMDD format as well as can be specified as *last* value, e.g. *date last 24h, date last 60m*, where h, m are hours, minutes, respectively.
- The *records* keyword can be used to retrieve DAS records regardless from their content. For instance, if one user place a query *site=T1_CH_CERN**, the DAS requests data from several data-services (Phedex, SiteDB), while the output results will only show site related records. If user wants to see which other records exists in DAS cache for given parameter, he/she can use *records site=T1_CH_CERN** to do that. In that case user will get back all records (site, block records) associated with given condition.

Sample queries used at CMS

Here we provide concrete examples of DAS queries used in CMS.

Find primary dataset

```
primary_dataset=Cosmics
primary_dataset=Cosmics | grep primary_dataset.name
primary_dataset=Cosmics | count(primary_dataset.name)
```

Find dataset

```
dataset primary_dataset=Cosmics
dataset dataset=*Cosmic* site=T3_US_FSU
dataset release=CMSSW_2_0_8
dataset release=CMSSW_2_0_8 | grep dataset.name, dataset.nevents
dataset run=148126
dataset date=20101101
```

Find block

```
block dataset=/Wgamma/Winter09_IDEAL_V12_FastSim_v1/GEN-SIM-DIGI-RECO
block=/ExpressPhysics/Commissioning10-Express-v6/FEVT#f86bef6a-86c2-48bc-9f46-2e868c13d86e
block site=T3_US_Cornell*
block site=srm-cms.cern.ch | count(block.name), sum(block.replica.size), avg(block.replica.size), me
```

Find file

```
file dataset=/Wgamma/Winter09_IDEAL_V12_FastSim_v1/GEN-SIM-DIGI-RECO
file block=/ExpressPhysics/Commissioning10-Express-v6/FEVT#f86bef6a-86c2-48bc-9f46-2e868c13d86e
file dataset=/Wgamma/Winter09_IDEAL_V12_FastSim_v1/GEN-SIM-DIGI-RECO | grep file.name, file.size
file dataset=/Wgamma/Winter09_IDEAL_V12_FastSim_v1/GEN-SIM-DIGI-RECO | grep file.name, file.size>150
```

```
file dataset=/Wgamma/Winter09_IDEAL_V12_FastSim_v1/GEN-SIM-DIGI-RECO | sum(file.size), count(file.nar
file block=/ExpressPhysics/Commissioning10-Express-v6/FEVT* site=T2_CH_CAF
file run=148126 dataset=/ZeroBias/Run2010B-Dec4ReReco_v1/RECO
file dataset=/ExpressPhysics/Commissioning10-Express-v6/FEVT | grep file.size | max(file.size),min(fi
```

Find lumi information

```
lumi file=/store/data/Run2010B/ZeroBias/RAW-RECO/v2/000/145/820/784478E3-52C2-DF11-A0CC-0018F3D0969A.
```

Find parents/children of a given dataset/files

```
child dataset=/QCDpt30/Summer08_IDEAL_V9_v1/GEN-SIM-RAW
parent dataset=/QCDpt30/Summer08_IDEAL_V9_skim_hlt_v1/USER
child file=/store/mc/Summer08/QCDpt30/GEN-SIM-RAW/IDEAL_V9_v1/0000/1EAE7A08-187D-DD11-85B5-001CC47D03
parent file=/store/mc/Summer08/QCDpt30/USER/IDEAL_V9_skim_hlt_v1/0003/367E05A0-707E-DD11-B0B9-001CC4A
```

Find information in local DBS instances

```
instance=cms_dbs_ph_analysis_02 dataset=/QCD_Pt_*_TuneZ2_7TeV_pythia6/wteo-qcd_tunez2_pt*_pythia*
```

Find run information

```
run=148126
run in [148124,148126]
run date last 60m
run date between [20101010, 20101011]
run run_status=Complete
run reco_status=1
run dataset=/Monitor/Commissioning08-v1/RAW
```

Find site information

```
site=T1_CH_CERN
site=T1_CH_CERN | grep site.admin
```

Jobsummary information

```
jobsummary date last 24h
jobsummary site=T1_DE_KIT date last 24h
jobsummary user=ValentinKuznetsov
```

1.3.2 DAS Command Line Interface (CLI) tool

The DAS Command Line Interface (CLI) tool can be downloaded directly from DAS server. It is python-based tool and does not require any additional dependencies, althought a python version of 2.6 and above is required. Its usage is very simple

```
Usage: das_client.py [options]
For more help please visit https://cmsweb.cern.ch/das/faq

Options:
  -h, --help            show this help message and exit
  -v VERBOSE, --verbose=VERBOSE
                        verbose output
```

```
  --query=QUERY         specify query for your request
  --host=HOST           host name of DAS cache server, default is
                        https://cmsweb.cern.ch
  --idx=IDX             start index for returned result set, aka pagination,
                        use w/ limit (default is 0)
  --limit=LIMIT         number of returned results (default is 10), use
                        --limit=0 to show all results
  --format=FORMAT       specify return data format (json or plain), default
                        plain.
  --threshold=THRESHOLD
                        query waiting threshold in sec, default is 5 minutes
  --key=CKEY            specify private key file name
  --cert=CERT           specify private certificate file name
  --retry=RETRY         specify number of retries upon busy DAS server message
  --das-headers         show DAS headers in JSON format
  --base=BASE           specify power base for size_format, default is 10 (can
                        be 2)
```

The query parameter specifies an input *DAS query <das_queries>*, while the format parameter can be used to get results either in JSON or plain (suitable for cut and paste) data format. Here is an example of using das_client tool to retrieve information about dataset pattern

```
python das_client.py --query="dataset=/ZMM*/*/*"

Showing 1-10 out of 2 results, for more results use --idx/--limit options

/ZMM_14TeV/Summer12-DESIGN42_V17_SLHCTk-v1/GEN-SIM
/ZMM/Summer11-DESIGN42_V11_428_SLHC1-v1/GEN-SIM
```

And here is the same output using JSON data format, the auxilary DAS headers are also requested:

```
python das_client.py --query="dataset=/ZMM*/*/*" --format=JSON --das-headers


{'apilist': ['das_core', 'fakeDatasetPattern'],
 'ctime': 0.0015709400177,
 'data': [{'_id': '523dcd7f0ec3dc12198a44c5',
           'cache_id': ['523dcd7f0ec3dc12198a44c3'],
           'das': {'api': ['fakeDatasetPattern'],
                   'condition_keys': ['dataset.name'],
                   'expire': 1379782315.848377,
                   'instance': 'cms_dbs_prod_global',
                   'primary_key': 'dataset.name',
                   'record': 1,
                   'services': [{'dbs': ['dbs']}],
                   'system': ['dbs'],
                   'ts': 1379782015.863179},
           'das_id': ['523dcd7d0ec3dc12198a4498'],
           'dataset': [{'created_by': '/DC=ch/DC=cern/OU=computers/CN=wmagent/vocms216.cern.ch',
                        'creation_time': '2012-02-24 01:40:40',
                        'datatype': 'mc',
                        'modification_time': '2012-02-29 21:25:52',
                        'modified_by': '/DC=org/DC=doegrids/OU=People/CN=Alan Malta Rodrigues 4861',
                        'name': '/ZMM_14TeV/Summer12-DESIGN42_V17_SLHCTk-v1/GEN-SIM',
                        'status': 'VALID',
                        'tag': 'DESIGN42_V17::All'}],
           'qhash': 'e5ced95dd57a5cfe1a3126a22a85a301'},
          {'_id': '523dcd7f0ec3dc12198a44c6',
```

```
            'cache_id': ['523dcd7f0ec3dc12198a44c4'],
            'das': {'api': ['fakeDatasetPattern'],
                    'condition_keys': ['dataset.name'],
                    'expire': 1379782315.848377,
                    'instance': 'cms_dbs_prod_global',
                    'primary_key': 'dataset.name',
                    'record': 1,
                    'services': [{'dbs': ['dbs']}],
                    'system': ['dbs'],
                    'ts': 1379782015.863179},
            'das_id': ['523dcd7d0ec3dc12198a4498'],
            'dataset': [{'created_by': 'cmsprod@cmsprod01.hep.wisc.edu',
                         'creation_time': '2011-12-29 17:47:25',
                         'datatype': 'mc',
                         'modification_time': '2012-01-05 17:40:17',
                         'modified_by': '/DC=org/DC=doegrids/OU=People/CN=Ajit Kumar Mohapatra 867118'
                         'name': '/ZMM/Summer11-DESIGN42_V11_428_SLHC1-v1/GEN-SIM',
                         'status': 'VALID',
                         'tag': 'DESIGN42_V11::All'}],
            'qhash': 'e5ced95dd57a5cfe1a3126a22a85a301'}],
 'incache': True,
 'mongo_query': {'fields': ['dataset'],
                 'instance': 'cms_dbs_prod_global',
                 'spec': {'dataset.name': '/ZMM*/*/*'}},
 'nresults': 2,
 'status': 'ok',
 'timestamp': 1379782017.68}
```

Using DAS CLI tool from other applications

It is possible to plug DAS CLI tool into other python applications. This can be done as following

```
from das_client import get_data

# invoke DAS CLI call for given host/query
# host: hostname of DAS server, e.g. https://cmsweb.cern.ch
# query: DAS query, e.g. dataset=/ZMM*/*/*
# idx: start index for pagination, e.g. 0
# limit: end index for pagination, e.g. 10, put 0 to get all results
# debug: True/False flag to get more debugging information
# threshold: 300 sec, is a default threshold to wait for DAS response
# ckey=None, cert=None are parameters which you can used to pass around
# your GRID credentials
# das_headers: True/False flag to get DAS headers, default is True

# please note that prior 1.9.X release the return type is str
# while from 1.9.X and on the return type is JSON

data = get_data(host, query, idx, limit, debug, threshold=300, ckey=None,
cert=None, das_headers=True)
```

Please note, that aforementioned code snippet requires to load *das_client.py* which is distributed within CMSSW. Due to CMSSW install policies the version of *das_client.py* may be quite old. If you need up-to-date *das_client.py* functionality you can follow this recipe. The code below download *das_client.py* directly from cmsweb site, compile it and use it in your application:

```
import os
import json
import urllib2
import httplib
import tempfile

class HTTPSClientHdlr(urllib2.HTTPSHandler):
    """
    Simple HTTPS client authentication class based on provided
    key/ca information
    """
    def __init__(self, key=None, cert=None, level=0):
        if  level:
            urllib2.HTTPSHandler.__init__(self, debuglevel=1)
        else:
            urllib2.HTTPSHandler.__init__(self)
        self.key = key
        self.cert = cert

    def https_open(self, req):
        """Open request method"""
        #Rather than pass in a reference to a connection class, we pass in
        # a reference to a function which, for all intents and purposes,
        # will behave as a constructor
        return self.do_open(self.get_connection, req)

    def get_connection(self, host, timeout=300):
        """Connection method"""
        if  self.key:
            return httplib.HTTPSConnection(host, key_file=self.key,
                                           cert_file=self.cert)
        return httplib.HTTPSConnection(host)

class DASClient(object):
    """DASClient object"""
    def __init__(self, debug=0):
        super(DASClient, self).__init__()
        self.debug = debug
        self.get_data = self.load_das_client()

    def get_das_client(self, debug=0):
        "Download das_client code from cmsweb"
        url  = 'https://cmsweb.cern.ch/das/cli'
        ckey = os.path.join(os.environ['HOME'], '.globus/userkey.pem')
        cert = os.path.join(os.environ['HOME'], '.globus/usercert.pem')
        req  = urllib2.Request(url=url, headers={})
        if  ckey and cert:
            hdlr = HTTPSClientHdlr(ckey, cert, debug)
        else:
            hdlr = urllib2.HTTPHandler(debuglevel=debug)
        opener = urllib2.build_opener(hdlr)
        fdesc = opener.open(req)
        cli = fdesc.read()
        fdesc.close()
        return cli

    def load_das_client(self):
        "Load DAS client module"
```

```
        cli = self.get_das_client()
        # compile python code as exec statement
        obj   = compile(cli, '<string>', 'exec')
        # define execution namespace
        namespace = {}
        # execute compiled python code in given namespace
        exec obj in namespace
        # return get_data object from namespace
        return namespace['get_data']

    def call(self, query, idx=0, limit=0, debug=0):
        "Query DAS data-service"
        host = 'https://cmsweb.cern.ch'
        data = self.get_data(host, query, idx, limit, debug)
        if  isinstance(data, basestring):
            return json.loads(data)
        return data

if __name__ == '__main__':
    das      = DASClient()
    query    = "/ZMM*/*/*"
    result   = das.call(query)
    if  result['status'] == 'ok':
        nres = result['nresults']
        data = result['data']
        print "Query=%s, #results=%s" % (query, nres)
        print data
```

Here we provide a simple example of how to use das_client to find dataset summary information.

```
# PLEASE NOTE: to use this example download das_client.py from
# cmsweb.cern.ch/das/cli

# system modules
import os
import sys
import json

from das_client import get_data

def drop_das_fields(row):
    "Drop DAS specific headers in given row"
    for key in ['das', 'das_id', 'cache_id', 'qhash']:
        if  row.has_key(key):
            del row[key]

def get_info(query):
    "Helper function to get information for given query"
    host    = 'https://cmsweb.cern.ch'
    idx     = 0
    limit   = 0
    debug   = False
    data    = get_data(host, query, idx, limit, debug)
    if  isinstance(data, basestring):
        dasjson = json.loads(data)
    else:
        dasjson = data
    status  = dasjson.get('status')
```

```
    if  status == 'ok':
        data = dasjson.get('data')
        return data

def get_datasets(query):
    "Helper function to get list of datasets for given query pattern"
    for row in get_info(query):
        for dataset in row['dataset']:
            yield dataset['name']

def get_summary(query):
    """
    Helper function to get dataset summary information either for a single
    dataset or dataset pattern
    """
    if  query.find('*') == -1:
        print "\n### query", query
        data = get_info(query)
        for row in data:
            drop_das_fields(row)
            print row
    else:
        for dataset in get_datasets(query):
            query = "dataset=%s" % dataset
            data = get_info(query)
            print "\n### dataset", dataset
            for row in data:
                drop_das_fields(row)
                print row

if __name__ == '__main__':
    # query dataset pattern
    query = "dataset=/ZMM*/*/*"
    # query specific dataset in certain DBS instance
    query = "dataset=/8TeV_T2tt_2j_semilepts_200_75_FSim526_Summer12_minus_v2/alkaloge-MG154_START52_
    get_summary(query)
```

1.4 Installation

1.4.1 Quick installation with pip and virtualenv

Basically on a Debian-like system you may just copy&paste all the commands below into a bash.

```
# get or install virtualenv, see http://www.virtualenv.org/en/latest/virtualenv.html#installation
# easiest is through apt-get or "pip install virtualenv" if available
sudo apt-get install python-virtualenv

# create virtual env
virtualenv dasenv # will use dir "dasenv"

# activate the virtualenv -- has to be run in every new shell
source dasenv/bin/activate

# get DAS source code
git clone git://github.com/dmwm/DAS.git
```

```
cd DAS

# install dependencies
export PYCURL_SSL_LIBRARY=gnutls  # depending on your distro try: openssl or gnutls
pip install -r requirements_freezed.txt
python -m nltk.downloader -e words stopwords wordnet

# you also need to connect to or install a MongoDB server, e.g.
sudo apt-get install mongodb-server
# set MongoDB port to 8230 in /etc/mongodb.conf or change dasconfig
# sed -i -e 's/8230/27017/g'  etc/das.cfg
# sed -i -e 's/8230/27017/g'  bin/das_db_import


# install DAS
python setup.py install
source ./init_env.sh

# initialize database with (default) service mappings
das_create_json_maps src/python/DAS/services/maps
das_update_database src/python/DAS/services/maps/das_maps_dbs_prod.js

# download YUI (the location is set in init_env.sh)
(curl -o yui.zip -L http://yuilibrary.com/downloads/yui2/yui_2.9.0.zip && \
 unzip yui.zip && mkdir -p  $YUI_ROOT && cp -R yui/* $YUI_ROOT/ )

# run the tests
source ./init_env.sh
touch /tmp/x509up_u$UID  # or use grid-proxy-init if you require certs...
python setup.py test

# finally start das server
# P.S. don't forget "source dasenv/bin/activate" when restarting in a new shell
source ./init_env.sh
bash das_server start
# Finally you can access it at: http://localhost:8212/das/
```

If, while running tests or starting DAS, you experience problems like "pycurl: libcurl link-time ssl backend (gnutls) is different from compile-time ssl backend (openssl)", try reinstalling (pycurl) using a different PYCURL_SSL_LIBRARY, e.g.:

```
PYCURL_SSL_LIBRARY=openssl
pip uninstall pycurl && pip install -r requirements_freezed.txt
```

For more details continue reading below.

1.4.2 DAS installation (old)

To get DAS release just clone it from GIT repository:

```
git clone git://github.com/dmwm/DAS.git
export PYTHONPATH=<install_dir>/DAS/src/python
```

DAS configuration is located in DAS/etc/das.cfg file and DAS executables can be found in DAS/bin area. To run DAS server you need to setup five configuration parameters, DAS_CONFIG, DAS_CSSPATH, DAS_TMPLPATH,

DAS_IMAGEPATH and DAS_JSPATH. The former is location of your das.cfg, while later should point to DAS/src/{css,templates,images,js} directories.

For your convenience you may create setup.sh script which will setup your environment, its context should be something like:

```
#!/bin/bash
dir=<install_dir> # put your install dir here
export PATH=$dir/install/bin:$dir/mongodb-linux-x86_64-2.2.2/bin:$PATH
export PATH=$dir/soft/DAS/bin:$PATH
export DAS_CONFIG=$dir/soft/DAS/etc/das.cfg
export DAS_CSSPATH=$dir/soft/DAS/src/css
export DAS_TMPLPATH=$dir/soft/DAS/src/templates
export DAS_IMAGESPATH=$dir/soft/DAS/src/images
export DAS_JSPATH=$dir/soft/DAS/src/js
export YUI_ROOT=$dir/soft/yui
export PYTHONPATH=$dir/soft/DAS/src/python
export PYTHONPATH=$dir/install/lib/python2.6/site-packages:$PYTHONPATH
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$dir/install/lib
```

where I installed all packages under local $dir/install area and keep DAS under $dir/soft/DAS.

Source code

DAS source code is freely available from [DAS] github repository. To install it on your system you need to use *git* which can be found from [GIT] web site.

Prerequisites

DAS depends on the following software:

- MongoDB and pymongo module
- libcurl library
- YUI library (Yahoo UI)
- python modules:
  - yajl (Yet Another JSON library) or cjson (C-JSON module)
  - CherryPy
  - Cheetah
  - PLY
  - PyYAML
  - pycurl

To install MongoDB visit their web site [Mongodb], download latest binary tar ball, unpack it and make its bin directory available in your path.

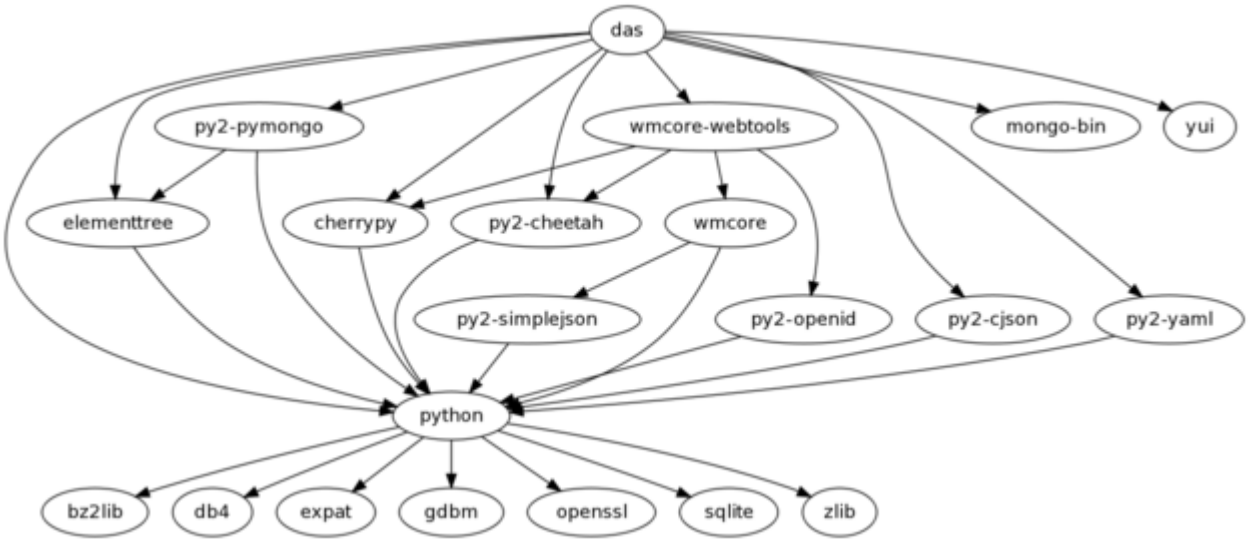
To install libcurl library visit its web site [CURL] and install it on your system.

To install YUI library, visit Yahoo developer web site [YUI] and install version 2 of their yui library.

To install python dependencies it is easier to use standard python installer *pip* (see above).

1.4.3 Dependencies

DAS is written in python and relies on standard python modules. The design back-end is MongoDB, which provides *schema-less* storage and a generic query language. Below you can see current set of dependencies for DAS within the CMS environment:



The *wmcore* and *wmcore-webtools* modules are a CMS general purpose web framework based on CherryPy. It is possible to run DAS both within this framework and in an entirely standalone version using only CherryPy.

Below we list all dependencies clarifying their role for DAS

- *python*, DAS is written in python (2.6), see [Python];
- *cherrypy*, a generic python web framework, see [CPF];
- *yui* the Yahoo YUI Library for building richly interactive web applications, see [YUI];
- *elementtree* and its *cElementTree* counterpart are used as generic XML parser in DAS, both implementations are now part of python (2.5 and above);
- *MongoDB*, a document-oriented database, the DAS DB back-ends, see [Mongodb] and [MongodbOverview];
- *pymongo*, a MongoDB python driver, see [Pymongo];
- *yaml*, a human-readable data serialization format (and superset of JSON), a python YAML library is used for DAS maps and server configurations, see [YAML], [PyYAML];
- *Cheetah*, a python template framework, used for all DAS web templates, see [Cheetah];
- *sphinx*, a python documentation library servers all DAS documentation, see [Sphinx];
- *ipython*, an interactive python shell (optional, used in some admin tools), see [IPython];
- *cjson*, a C library providing a faster JSON decoder/encoder for python (optional), see [CJSON];
- *yajl*, a C library providing a faster JSON decoder/encoder for python (optional), see [YAJL];
- *pycurl* and *curl*, python module for libcurl library, see [PyCurl], [CURL];
- *PLY*, python Lexer and Yacc, see [PLY];

1.5 Setting up and customizing DAS installation

First set basic configuration on database to use and other server parameters.

To integrate DAS with your own custom services the biggest and most time consuming task is preparing the service mappings which define the services to be integrated.

1.5.1 DAS configuration file

DAS configuration consists of a single file, $DAS_ROOT/etc/das.cfg. Its structure is shown below:

```
[das]                    # DAS core configuration
verbose = 0              # verbosity level, 0 means lowest
parserdir = /tmp         # DAS PLY parser cache directory
multitask = True         # enable multitasking for DAS core (threading)
core_workers = 10        # number of DAS core workers who contact data-providers
api_workers = 2          # number of API workers who run simultaneously
thread_weights = 'dbs:3','phedex:3' # thread weight for given services
error_expire = 300       # expiration time for error records (in seconds)
emptyset_expire = 5      # expiration time for empty records (in seconds)
services = dbs,phedex    # list of participated data-providers

[cacherequests]
Admin = 50               # number of queries for admin user role
Unlimited = 10000        # number of queries for unlimited user role
ProductionAccess = 5000  # number of user for production user role

[web_server]             # DAS web server configruation parameters
thread_pool = 30         # number of threads for CherryPy
socket_queue_size = 15   # queue size for requests while server is busy
host = 0.0.0.0           # host IP, the 0.0.0.0 means visible everywhere
log_screen = True        # print log to stdout
url_base = /das          # DAS server url base
port = 8212              # DAS server port
pid = /tmp/logs/dsw.pid  # DAS server pid file
status_update = 2500     #
web_workers = 10         # Number of DAS web server workers who handle user requests
queue_limit = 200        # DAS server queue limit
adjust_input = True      # Adjust user input (boolean)
dbs_daemon = True        # Run DBSDaemon (boolean)
dbs_daemon_interval = 300# interval for DBSDaemon update in sec
dbs_daemon_expire = 3600 # expiration timestamp for DBSDaemon records
hot_threshold = 100      # a hot threshold for powerful users
onhold_daemon = True     # Run onhold daemon for queries which put on hold after hot threshold

[dbs]                    # DBS server configuration
dbs_instances = prod,dev # DBS instances
dbs_global_instance = prod # name of the global DBS instance
dbs_global_url = http://a.b.c # DBS data-provider URL

[mongodb]                # MongoDB configuration parameters
dburi = mongodb://localhost:8230 # MongoDB URI
bulkupdate_size = 5000   # size of bulk insert/update operations
dbname = das             # MongoDB database name
lifetime = 86400         # default lifetime (in seconds) for DAS records
```

```
[dasdb]                  # DAS DB cache parameters
dbname = das             # name of DAS cache database
cachecollection = cache  # name of cache collection
mergecollection = merge  # name of merge collection
mrcollection = mapreduce # name of mapreduce collection

[loggingdb]
capped_size = 104857600
collname = db
dbname = logging

[analyticsdb]            # AnalyticsDB configuration parameters
dbname = analytics       # name of analytics database
collname = db            # name of analytics collection
history = 5184000        # expiration time for records in an/ collection (in seconds)

[mappingdb]              # MappingDB configuration parameters
dbname = mapping         # name of mapping database
collname = db            # name of mapping collection

[parserdb]               # parserdb configuration parameters
dbname = parser          # parser database name
enable = True            # use it in DAS or not (boolean)
collname = db            # collection name
sizecap = 5242880        # size of capped collection

[dbs_phedex]             # dbs_phedex configuration parameters
expiration = 3600        # expiration time stamp
urls = http://dbs,https://phedex # DBS and Phedex URLs
```

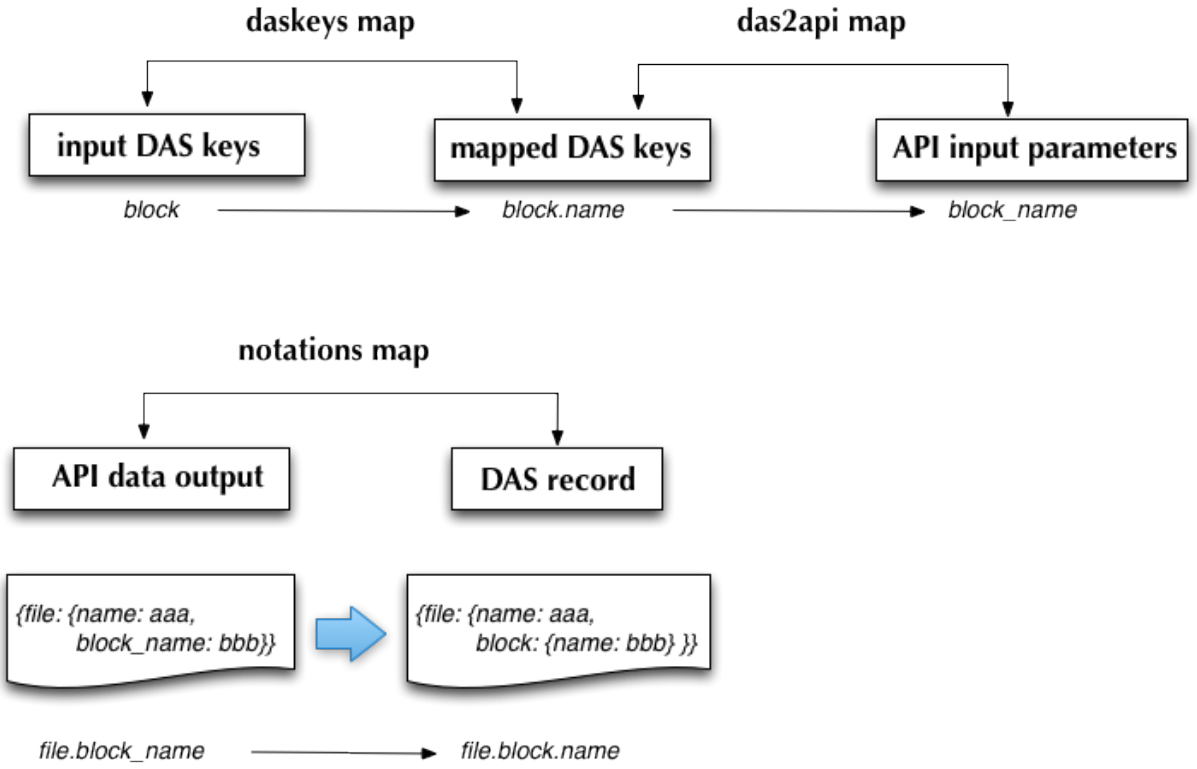
For up-to-date configuration parameters please see *utils/das_config.py*

1.5.2 DAS Mapping DB

DAS uses Query Language (QL) to look-up data from data-providers as well as its own cache. The data provided by various data services can come in variety of form and data formats, while DAS cache stores data records in JSON data format. Therefore we need to define certain mappings between DAS QL and data-provider API calls as well as DAS QL and data records in DAS cache. To serve this goal DAS relies on its Mapping DB which holds information about all the data-service APIs which are used by DAS, and the necessary mappings between DAS and API records

Each mapping file holds the following schema:

- **system**, the name of data-provider
- **format**, the data format used by data-provider, e.g. XML or JSON
- series of maps for APIs used in DAS workflow, where each API map has the following entries
  - **urn**, API alias name
  - **url**, the API URL
  - **params**, the set of input parameters for API in question
  - **lookup**, the name of DAS look-up key the given API serves
  - **das_map**, the list of maps which covers DAS QL keys, record names and API input argument, along with optional pattern; every map has the following parameters
    - **das_key**, the name of DAS QL key, e.g. run



        - **rec_key**, the DAS record key name, e.g. run.number
        - **api_arg**, the corresponding API input argument name, e.g. run_number
        - **pattern**, optional regex for input argument
    - **wild_card**, the optional notation for wild-card usage in given API, e.g. `*` or `%`
    - ckey and cert, the path to GRID credentials
- **notations** map which transforms API output into das record keys; this map consists of maps with the following structure
    - **api_output**, the key name returned by API record
    - **das_key**, the DAS record key
    - **api**, the name of API this mapping should be applied for

DAS also uses presentation map to translate DAS records into human readable form, e.g. to translate *file.nevents* into *Number of events*

The DAS maps use YAML data-format. Each file may contain several data-service API mappings, as well as auxilary information about data-provider, e.g. data format, expiration timestamp, etc. For example here is a simple mapping file for google map APIs

```
system : google_maps
format : JSON
---
urn : google_geo_maps
url : "http://maps.google.com/maps/geo"
expire : 30
```

```
params : { "q" : "required", "output": "json" }
lookup : city
das_map : [
    {"das_key":"city","rec_key":"city.name","api_arg":"q"},
]
---
urn : google_geo_maps_zip
url : "http://maps.google.com/maps/geo"
expire : 30
params : { "q" : "required", "output": "json" }
lookup : zip
das_map : [
    {"das_key":"zip","rec_key":"zip.code","api_arg":"q"},
]
---
notations : [
    {"api_output":"zip.name", "rec_key":"zip.code", "api":""},
    {"api_output":"name", "rec_key":"code", "api":"google_geo_maps_zip"},
]
```

As you can see it defines the data-provider name, `google_maps` (DAS call it system), the data format `JSON` used by this data-provider as well as three maps (for each API usage), separated by tripple dashes. The first one defines mapping for geo location for a given city key, the second defines geo location for a given zip key and mapping for notations used by DAS workflow. In particulat, DAS will map `zip.name` into `zip.code` for any api, and `name` into `code` for `google_geo_maps_zip` api (the meaning of these translation will become clear when we will discuss concrete example below).

As you may noticed, every mapping (the code between tripple dashes) has repeated strucute. It defines *urn, url, expire, params, lookup, das_map* values. The *urn* stands for uniform resource name, this alias is used by DAS to distinguish APIs and their usage pattern, the *url* is canonical URL for API in question, the *params* defines a dictionary of input parameters accepted by API, the *das_map* is mapping from DAS keys into DAS data records, and finally, *lookup* is the name of DAS key this map is designed for.

To accommodate different use cases of API usage the `params` structure may contain three types of parameter values: the **default**, **required** and **optional** values. The default value will be passed to API *as is*, the required value must be substituted by DAS workflow (it will be taken from the query provided by DAS user, if it will not be provide the API call will be discarded from DAS workflow) and the optional value which can be skipped by API call.

Example

In this section we show a concrete example of mappings used by DAS workflow for one of the data-services. Let's take the following DAS queries:

```
file file=X
file file=X status=VALID
```

These queries will correspond to the following DAS record structure:

```
{"file" : {"name": "X", "size":1, "nevents": 10, ...}}
```

The dots just indicate that structure can be more comlpex.

The `file` DAS key is mapped into `file.name` key.attribute value within DAS record, here the period divides key from attribute in aforementioned dictionary. Therefore `file.name` value is `X`, `file.size` value is 1, etc.

Here is an example of one of the DAS mapping records which can serve discussed DAS queries (please note that it may be several data-services which may provide the data for given DAS query).

```
urn: files
url : "https://cmsweb.cern.ch/dbs/prod/global/DBSReader/files/"
expire : 900
params : {
        "logical_file_name":"required",
        "detail":"True",
        "status": "optional",
}
lookup : file
das_map : [
    {"das_key": "file", "rec_key":"file.name", "api_arg":"logical_file_name",
     "pattern": "/.*.root"},
    {"das_key": "status", "rec_key":"status.name", "api_arg":"status",
     "pattern": "(VALID|INVALID)"},
]
```

This record defines `files` API with given URL and expire timestamp. It specifies the input parameters (`params`), in particular, `logical_file_name` is required by this API, the `detail` has default value True and `status` is an optional input parameter. The daskeys mapping defines mapping between DAS keys used by end-user and DAS record keys. For example

```
file file=X
```

will be mapped into the following API call:

```
https://cmsweb.cern.ch/dbs/prod/global/DBSReader/files?logical_file_name=X&detail=True
```

while:

```
file file=X status=VALID
```

will be mapped into:

```
https://cmsweb.cern.ch/dbs/prod/global/DBSReader/files?logical_file_name=X&detail=True&status=VALID
```

In both case, the data-provider will return back the following data-record, e.g.:

```
{"logical_file_name: "X", "size": 1, ...}
```

therefore we need another mapping from API data record into expected DAS record structure (as we discussed above):

```
{"file": {"name": "X", "size": 1, ...}}
```

To perform such translation DAS workflow consults `das2api` maps which defines them, e.g. `logical_file_name` maps into `file.name`, etc.

Sometimes, different data-services provides data records who have different notations, e.g. `fileName`, `file_name`, etc. To accommodate this differences DAS consults notation map to perform transation from one into another notation.

Finally, to translate DAS records into human readable form we need another mapping, the presentation one. It defines what should be presented at DAS UI level for a given DAS record. For example, we may want to display "File name" at DAS UI, instead of showing `file.name`. To perform this translation DAS uses presentation map.

1.5.3 How to add new data-service

DAS has pluggable architecture, so adding a new CMS data-service should be a relatively easy procedure. Here we discuss two different ways to add a new service into DAS.

Plug and play interface

This work is in progress.

A new data-service can register with DAS by providing a file describing the interface and available APIs. This configuration includes the data-service URL, the data format provided, an optional expiration timestamp for the data, the API name, necessary parameters and optional mapping onto DAS keys.

A new DAS interface will allow this information to be added via a simple configuration file. The data-service configuration files should be presented in [YAML] data-format.

An example configuration follows [1]:

```
# SiteDB API mapping to DAS
system : sitedb
format : JSON

# API record
---
# URI description
urn : CMSNametoAdmins
url : "https://a.b.com/sitedb/api"
params : {'name':''}
expire : 3600 # optional DAS uses internal default value

# DAS keys mapping defines mapping between query names, e.g. run,
# and its actual representation in DAS record, e.g. run.number
daskeys : [
    {'key':'site', 'map':'site.name', 'pattern':''},
    {'key':'admin', 'map':'email', 'pattern':''}
]

# DAS search keys to API input parameter mapping
das2api : [
        {'das_key':'site', 'api_param':'se', 'pattern':""}
]
---
# next API
---
# APIs notation mapping maps data-service output into
# DAS syntax, e.g
# {'site_name':'abc'} ==> {'site':{'name':'abc'}}
notation : [
        {'notation':'site_name', 'map': 'site.name', 'api': ''}
]
```

The syntax consists of key:value pairs, where value can be in a form of string, list or dictionary. Hash sign (#) defines a comment, the three dashes (—) defines the record separator. Each record starts with definition of system and data format provided by data-service.

[1] This example demonstrates flexibility of YAML data-format and shows different representation styles.

```
# comment
system: my_system_name
format: XML
```

Those definitions will be applied to each API defined later in a map file. The API section followed after the record separator and should define: *urn*, *url*, *expire*, *params* and *daskeys*.

```
# API section
---
urn: api_alias
url: "http://a.b.com/method"
expire: 3600 # in seconds
params: {} # dictionary of data-service input parameters
daskeys: [{}, {}] # list of dictionaries for DAS key maps
```

- the *urn* is the API name or identifier (any name different from the API name itself) and used solely inside of DAS
- the *url* defines the data-service URL
- the *params* are data-service input parameters
- the *daskeys* is a list of maps between data-service input parameters and DAS internal key representation. For instance when we say *site* we might mean site CMS name or site SE/CE name. So the DAS key will be *site* while DAS internal key representation may be *site.name* or *site.sename*. So, each entry in *daskeys* list is defined as the following dictionary: {'key':value, 'map':value, 'pattern':''}, where pattern is a regular expression which can be used to differentiate between different arguments where they have different structures.
- the (optional) *das2api* map defines mapping between DAS internal key and data-service input parameter. For instance, *site.name* DAS key can be mapping into _name_ data-service input parameter.

The next API record can be followed by the next record separator, e.g.

```
---
# API record 1
urn: api_alias1
url: "http://a.b.com/method1"
expire: 3600 # in seconds
params: {} # dictionary of data-service input parameters
daskeys: [{}, {}] # list of dictionaries for DAS key maps
---
# API record 2
urn: api_alias2
url: "http://a.b.com/method2"
expire: 1800 # in seconds
params: {} # dictionary of data-service input parameters
daskeys: [{}, {}] # list of dictionaries for DAS key maps
```

At the end of DAS map there is an optional *notation* mapping, which defines data-service output mapping back into DAS internal key representation (including converting from flat to hierarchical structures if necessary).

```
---
# APIs notation mapping maps data-service output into
# DAS syntax, e.g
# {'site_name':'abc'} ==> {'site':{'name':'abc'}}
notation : [
        {'notation':'site_name', 'map': 'site.name', 'api': ''}
]
```

For instance, if your data service returns runNumber and in DAS we use run_number you'll define this mapping in *notation* section.

To summarize, the YAML map file provides

- system name
- underlying data format used by this service for its meta-data
- the list of apis records, each record contains the following:
  - urn name, DAS will use it as API name
  - url of data-service
  - expiration timestamp (how long its data can live in DAS)
  - input parameters, provide a dictionary
  - list of daskeys, where each key contains its name *key*, the mapping within a DAS record, *map*, and appropriate pattern
  - list of API to DAS notations (if any); different API can yield data in different notations, for instance, siteName and site_name. To accommodate these syntactic differences we use this mapping.
- notation mapping between data-service provider output and DAS

Add new service via API

You can manually add new service by extending `DAS.services.abstract_service.DASAbstractService` and overriding the *api* method.

To do so we need to create a new class inherited from `DAS.services.abstract_service.DASAbstractService`.

```
class MyDataService(DASAbstractService):
    """
    Helper class to provide access to MyData service
    """
    def __init__(self, config):
        DASAbstractService.__init__(self, 'mydata', config)
        self.map = self.dasmapping.servicemap(self.name)
        map_validator(self.map)
```

optionally the class can override .. function:: def api(self, query) method of `DAS.services.abstract_service.DASAbstractService` Here is an example of such an implementation

```
def api(self, query):
    """My API implementation"""
    api     = self.map.keys()[0] # get API from internal map
    url     = self.map[api]['url']
    expire  = self.map[api]['expire']
    args    = dict(self.map[api]['params']) # get args from internal map
    time0   = time.time()
    dasrows = function(url, args) # get data and convert to DAS records
    ctime   = time.time() - time0
    self.write_to_cache(query, expire, url, api, args, dasrows, ctime)
```

The hypothetical function call should contact the data-service and fetch, parse and yield data. Please note that we encourage the use of python generators [Gen] in function implementations.

1.5.4 DAS CMS Operations

Here we outline CMS specific operations to build/install and maintain DAS system. Please note, all instructions below refer to $DAS_VER as DAS version.

Building RPM

For generic CMS build procedure please refer to this page. We build SLC5 RPMs on vocms82 build node using 64-bit architecture. Here is a list of build steps:

```
# clean the area
rm -rf bin bootstraptmp build BUILD cmsset_default.* \
    common RPMS slc5_amd64_gcc434 SOURCES SPECS SRPMS tmp var

# get PKGTOOLS and CMSDIST
cvs co -r $DAS_VER PKGTOOLS
cvs co -r dg20091203b-comp-base CMSDIST
cvs update -r 1.59 CMSDIST/python.spec

# update/modify appropriate spec's, e.g.
# cvs up -A CMSDIST/das.spec

# perform the build
export SCRAM_ARCH=slc5_amd64_gcc434
PKGTOOLS/cmsBuild --architecture=$SCRAM_ARCH --cfg=./build.cfg
```

The build.cfg file has the following content:

```
[globals]
assumeYes: True
onlyOnce: True
testTag: False
trace: True
tag: cmp
repository: comp
[bootstrap]
priority: -30
doNotBootstrap: True
repositoryDir: comp
[build das]
compilingProcesses: 6
workersPoolSize: 2
priority: -20
#[upload das]
#priority: 100
#syncBack: True
```

Build logs and packages are located in BUILD area. Once build is complete all CMS RPMs are installed under slc5_amd64_gcc434 area. Please verify that DAS RPMs has been installed. Once everything is ok, request from CERN operator to upload DAS RPMs into CMS COMP repository. It can be done using the following set of commands:

```
eval `ssh-agent -s`
ssh-add -t 36000
export SCRAM_ARCH=slc5_amd64_gcc434
PKGTOOLS/cmsBuild --architecture=$SCRAM_ARCH --cfg=./upload.cfg
```

where upload.cfg is similar to build.cfg with last three lines commented out.

Installing RPMs

DAS follows CMS build/install generic procedure. Here we outline all necessary steps to install CMS DAS RPMs into your area

```
export PROJ_DIR=/data/projects/das # modify accordingly
export SCRAM_ARCH=slc5_amd64_gcc434
export APT_VERSION=0.5.15lorg3.2-cmp
export V=$DAS_VER

mkdir -p $PROJ_DIR

wget -O$PROJ_DIR/bootstrap.sh http://cmsrep.cern.ch/cmssw/cms/bootstrap.sh
chmod +x $PROJ_DIR/bootstrap.sh
# perform this step only once
$PROJ_DIR/bootstrap.sh -repository comp -arch $SCRAM_ARCH -path $PROJ_DIR setup
cd $PROJ_DIR
source $SCRAM_ARCH/external/apt/$APT_VERSION/etc/profile.d/init.sh

apt-get update
apt-get install cms+das+$V
```

Updating RPM

Please following these steps to update DAS RPM in your area:

```
export PROJ_DIR=/data/projects/das
export SCRAM_ARCH=slc5_amd64_gcc434
export APT_VERSION=0.5.15lorg3.2-cmp
export V=$DAS_VER

cd $PROJ_DIR
source $SCRAM_ARCH/external/apt/$APT_VERSION/etc/profile.d/init.sh

apt-get update
apt-get install cms+das+$V
```

setup.sh

In order to run DAS installed via CMS RPMs you need to setup your environment. Here we provide a simple steps which you can follow to create a single setup.sh file and use it afterwards:

```
echo "ver=$V" > $PROJ_DIR/setup.sh
echo "export PROJ_DIR=$PROJ_DIR" >> $PROJ_DIR/setup.sh
echo "export SCRAM_ARCH=$SCRAM_ARCH" >> $PROJ_DIR/setup.sh
echo -e "export APT_VERSION=$APT_VERSION\n" >> $PROJ_DIR/setup.sh
echo 'source $SCRAM_ARCH/external/apt/$APT_VERSION/etc/profile.d/init.sh' >> $PROJ_DIR/setup.sh
echo 'source $PROJ_DIR/slc4_ia32_gcc345/cms/das/$ver/etc/profile.d/init.sh' >> $PROJ_DIR/setup.sh
```

1.5.5 DAS operations

Running DAS services

DAS consists of multi-threaded DAS web and DAS analytics servers. Please refer to *DAS CMS operations* section for deployment instructions and *DAS configuration* section for description of configuration parameters.

By CMS conventions DAS uses the following ports

- 8212 for DAS web server
- 8213 for DAS analytics server

In order to start each of server you need to setup your environment, see *setup.sh* file. For that use the following steps:

```
cd /data/projects/das/ # change accordingly
source setup.sh
```

Setting up DAS maps

Data-services are registered via the service mappings (DAS Maps) which define the relationships between the services and how their inputs or outputs shall be transformed.

To initialize or update the DAS maps (along with other metadata) call the bin/das_update_database with location of maps. The CMS maps are located in $DAS_ROOT/src/python/DAS/services/cms_maps directory.

Apache redirect rules

Here we outline Apache redirect rules which can be used to serve DAS services. Please note we used localhost IP, 127.0.0.1 for reference, which should be substituted with actual hostname of the node where DAS services will run.

Rewrite rules for apache configuration file, e.g. httpd.conf

```
# Rewrite rules
# uncomment this line if you compile apache with dynamic modules
#LoadModule rewrite_module modules/mod_rewrite.so
# uncomment this line if you compile your modules within apache
RewriteEngine on

# DAS rewrite rules
RewriteRule ^(/das(/.*)?)$  https://127.0.0.1$1 [R=301,L]

Include conf/extra/httpd-ssl.conf
```

Rules for SSL rewrites:

```
RewriteRule ^/das(/.*)?$ http://127.0.0.1:8212/das/$1 [P,L]
```

MongoDB server

DAS uses MongoDB as its back-end for Mapping, Analytics, Logging, Cache and Merge DBs.

RPM installation will supply its proper configuration file. You must start cache back-end before starting the DAS cache server.

MongoDB server operates via standard UNIX init script:

```
$MONGO_ROOT/etc/profile.d/mongo_init.sh start|stop|status
```

The MongoDB database is located in $MONGO_ROOT/db, while logs are in $MONGO_ROOT/logs.

DAS web server

DAS web server is multi-threaded server. It can be started using the following command

```
das_server start|stop|status|restart
```

The das_server should be in your path once you setup your CMS DAS environment, see *setup.sh*, otherwise please locate it under $DAS_ROOT/bin/ area.

It consits of N threads used by DAS web server, plus M threads used by DAS core to run data retrieval processes concurrently. DAS configuration has the following parameters

```
config.web_server.thread_pool = 30
config.web_server.socket_queue_size = 15
config.web_server.number_of_workers = 8
config.web_server.queue_limit = 20
```

The first two (thread_pool and socket_queue_size) are used by underlying CherryPy server to control its internal thread pool and queue, while second pair (number_of_workers and queue_limit) are used by DAS core to control number of worker threads used internally. The DAS core multitasking can be turned off by using

```
config.das.multitask = False
```

parameter.

DAS analytics server

The DAS analytics is complementary set of daemons who can populate DAS cache based on provided tasks, see *DAS analytics*. Their server can be started as simple as

```
python -u $DAS_PYTHONPATH/DAS/analytics/analytics_controller.py $DAS_ANALYTICS_CFG
```

where analytics_controller.py expects a valid DAS analytics config file. Below you can find a simple version of analytics configuration file (in CMS configuration format):

```
logfile_rotating_size = 100000
log_to_stdout = 0
log_to_file = 0
web_history = 10000
minimum_interval = 60
log_format = ""
max_retries = 1
```

```
retry_delay = 60
logfile = "/tmp/das_analytics.log"
logfile_rotating_interval = 24
no_start_offset = False
web = True
web_port = 8213
logfile_mode = "None"
workers = 4
log_to_stderr = 0
web_base = "/analytics"
logfile_rotating_count = 0
pid = "/tmp/analytics.pid"

# delta = finish-start (finish = start+period)
# period: the calls to be considered by analytics, default 1 month
# allowed_gap: is maximum gap in the summary record we are happy to ignore, default 1h
# interval: interval of the task, suggested value 14400 (4 hours)

# here interval=14400, 4 hours
# while preempt argument specify at which time task should be run before results are expired.
Task("SiteHotspot", "ValueHotspot", 14400, key="site.name")
Task("DatasetHotspot", "ValueHotspot", 3600, key="dataset.name")
```

DAS administration

DAS RPMs provide a set of tools for administration tasks. They are located at $DAS_ROOT/bin.

- das_server is a DAS server init script;
- das_cli is DAS stand-along CLI tool, it doesn't require neither cache or web DAS servers;
- das_code_quality.sh is a bash script to check DAS code quality. It is based on pylint tool, see [PYLINT].
- das_config is a tool to create DAS configuration file;
- das_update_database is a tool to update DAS maps, and initialize other metadata
- das_maps_yml2json is a tool that generates json DB dumps from YML dasmaps.
- das_db_import is a helper used to import the DB dumps including dasmaps, keylearning, inputvals
- das_mapreduce is a tool to create map/reduce function for DAS;

1.6 DAS architecture and internals

This section describes DAS internal modules and datastructures (including records stored in the database).

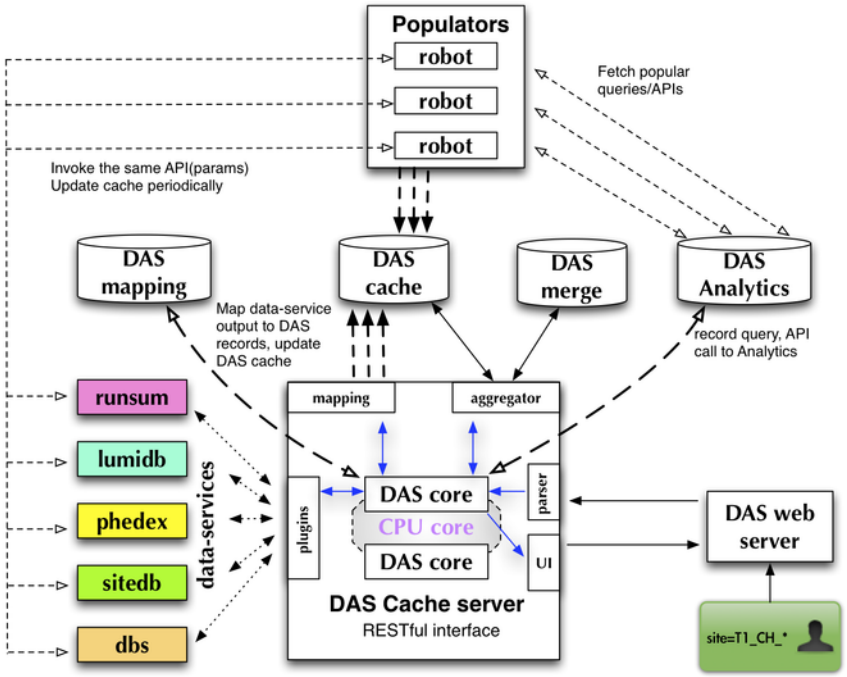
1.6.1 DAS architecture

DAS architecture is based on several components:

- common core library
  - analytics DB
  - mapping DB
  - caching DB

  - merging DB
  - logging DB
- data-service plugins, each plugin contains
  - data-service handler class
  - API map
  - notation map
- cache server
- client web server

The last two components, cache and client servers, are optional. The code itself can work without cache/client servers with the CLI tool which uses core libraries. But their existence allows the introduction of DAS pre-fetch strategies, DAS robots, which can significantly improve responsiveness of the system and add multi-user support into DAS. The following picture represents current DAS architecture:



It consists of DAS web server with RESTful interface, DAS cache server, DAS Analytics/Mapping/Cache DBs and DAS robots (for pre-fetching queries). The DAS cache server uses multithreading to consume and work on several user requests at the same time. All queries are written to the DAS Analytics DB. A mapping between data-service and DAS notations is stored in the DAS Mapping DB. Communication with end-users is done via set of REST calls. User can make GET/POST/DELETE requests to fetch or delete data in DAS, respectively. The DAS workflow can be summarised as:

- DAS cache-server receives a query from the client (either DAS web server or DAS CLI)
- The input query is parsed and the selection key(s) and condition(s) identified and mapped to the appropriate data-services
- The query is added to the DAS Analytics DB

- The DAS cache is checked for existing data
  - if available
    - Data is retrieved from the cache
  - otherwise:
    - The necessary data services are queried
    - The results are parsed, transformed and inserted into the DAS cache
    - The user receives a message that the data is being located
- The data is formatted and returned to the user

For more information please see the *DAS workflow* page. The DAS DBs use the MongoDB document-oriented database (see [Mongodb]), although during design/evaluation process we considered a number of other technologies, such as different RDMS flavors, memcached [Memcached] and other key-value based data stores (eg CouchDB [Couchdb]), etc.

1.6.2 DAS server

DAS server is multi-threaded application which runs within CherryPy web framework. It consists of the following main componens:

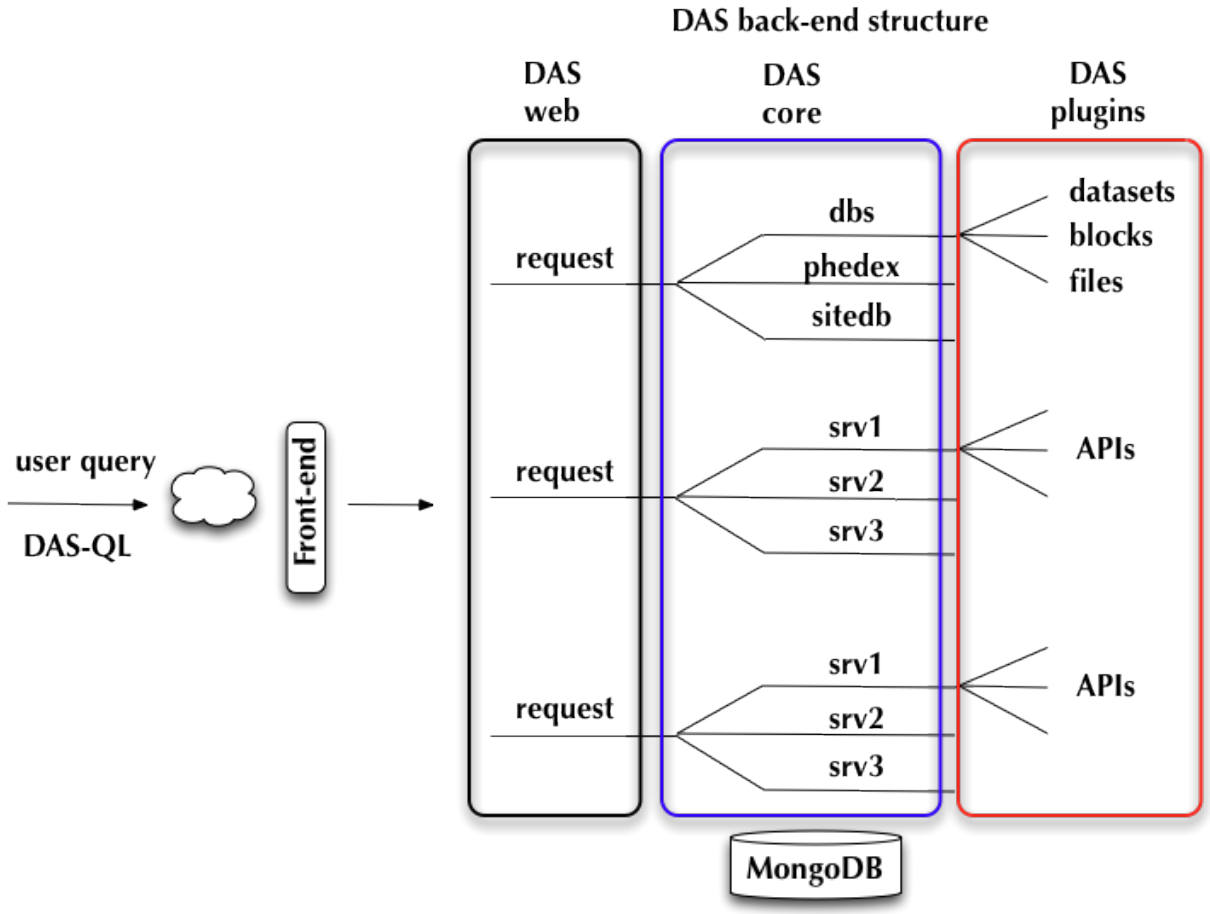
- DAS web server
- DAS core engine, which by itself consist of
  - DAS abstract data-service
  - Data-provider plugins (data-service implementations)
- DAS analytics daemons
- DAS command-line client
- Data-provider daemons, e.g. DBS daemon, etc.
- Various monitors, e.g. MongoDB connection monitor, etc.
- MongoDB

Upon user request, the front-end validates user input and pass it to DAS web server. It decompose user query into series of requests to underlying core engine, who by itself invokes multiple APIs to fetch data from data-provider and place them into MongoDB. Later this data are serverd back to the user and stay in cache for period of time determined by data-providers, see Figure:

Thread structure of DAS server

Below we outline a typical layout of DAS server threads:

- 1 main thread
- 1 HTTP server thread (CherryPy)
- 1 TimeoutMonitor thread (CherryPy)
- 1 dbs_phedex_monitor thread (dbs_phedex combined service)
- 1 dbs_phedex worker thread (dbs_phedex combined service)
- 1 lumi_service thread (lumi service)


DAS back-end structure
DAS web
DAS core
DAS plugins
user query
DAS-QL
Front-end
request
request
request
dbs
phedex
sitedb
srv1
srv2
srv3
srv1
srv2
srv3
datasets
blocks
files
APIs
APIs
MongoDB

- N_{CP} CherryPy threads
- N_{DAS} worker threads, they are allocated as following:
  - N_{web} threads for web workers
  - N_{core} threads for DAS core workers
  - N_{api} threads for DAS service APIs

In addition DAS configuration uses das.thread_weights parameter to weight certain API threads. It is defined as a list of srv:weight pairs where each service gets N_{api} number of threads.

Therefore the total number of threads is quite hight (range in first hundred) and it is determined by the following formula

$$N_{threads} = N_{main} + N_{CP} + N_{DAS}$$
$$N_{DAS} = N_{web} + N_{core} + N_{api}$$

N_{main} equals to sum of main, timeout, http, dbs_phedex and lumi threads N_{CP} is defined in DAS configuration file, typical value is 30 N_{web} is defined in DAS config file, see web_server.web_workers N_{core} is defined in DAS config file, see das.core_workers N_{api} is defined in DAS config file, see das.api_workers

For example, usig the following configuration parameters

```
[das]
multitask = True          # enable multitasking for DAS core (threading)
core_workers = 10         # number of DAS core workers who contact data-providers
api_workers = 2           # number of API workers who run simultaneously
thread_weights = 'dbs:3','phedex:3' # thread weight for given services
das.services = dbs,phedex,dashboard,monitor,runregistry,sitedb2,tier0,conddb,google_maps,postalcode,

[web_server]              # DAS web server configruation parameters
thread_pool = 30          # number of threads for CherryPy
web_workers = 10          # Number of DAS web server workers who handle user requests
dbs_daemon = True         # Run DBSDaemon (boolean)
onhold_daemon = True      # Run onhold daemon for queries which put on hold after hot threshold
```

we get the DAS server running with 151 threads

$$N_{main} = 7,\ N_{CP} = 30,\ N_{DAS} = 114$$

where N_{DAS} has the following breakdown

$$N_{web} = 20,\ N_{core} = 60,\ N_{api} = 34$$

here we calculated N_{api} as following: we have 13 services, each of them uses 2 API workers (as specified in das configuration), but dbs and phedex data-services are weighed with weight 3, therefore the total number of dbs and phedex workers is 6, respectively. To sum up the numbers we have: 11 services with 2 API workers plus 6 workers for dbs and 6 workers for phedex.

Debugging DAS server

There is nice way to get a snapshot of current activity of DAS server by sending SIGUSR1 signal to DAS server, e.g. upon executing *kill -SIGUSR1 <PID>* command you'll get the following output in DAS log

```
# Thread: DASAbstractService:dbs:PluginTaskManager(4706848768)
File: "/opt/local/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/threading.py", line
  self.__bootstrap_inner()
File: "/opt/local/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/threading.py", line
  self.run()
File: "/Users/vk/CMS/GIT/github/DAS/src/python/DAS/utils/task_manager.py", line 39, in run
  task = self._tasks.get()
File: "/opt/local/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/Queue.py", line 168,
  self.not_empty.wait()
File: "/opt/local/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/threading.py", line
  waiter.acquire()
....
.... and similar output for all other DAS threads
....
```

1.6.3 DAS records

DAS data objects

DAS needs to deal with a variety of different data object representations. Data from providers may have both different formats (eg XML, JSON), and different ways of storing hierarchical information. The structure of response data is not known to DAS a-priori. Therefore it needs to treat them as data objects. Here we define what it means for DAS and provide examples of DAS data objects or DAS records.

There are basically two types of data objects: flat and hierarchical ones.

- Flat

```
{"dataset":"abc", size:1, "location":"CERN"}
```

- Hierarchical

```
{"dataset": {"name":"abc", size:1, "location":"CERN"}}
```

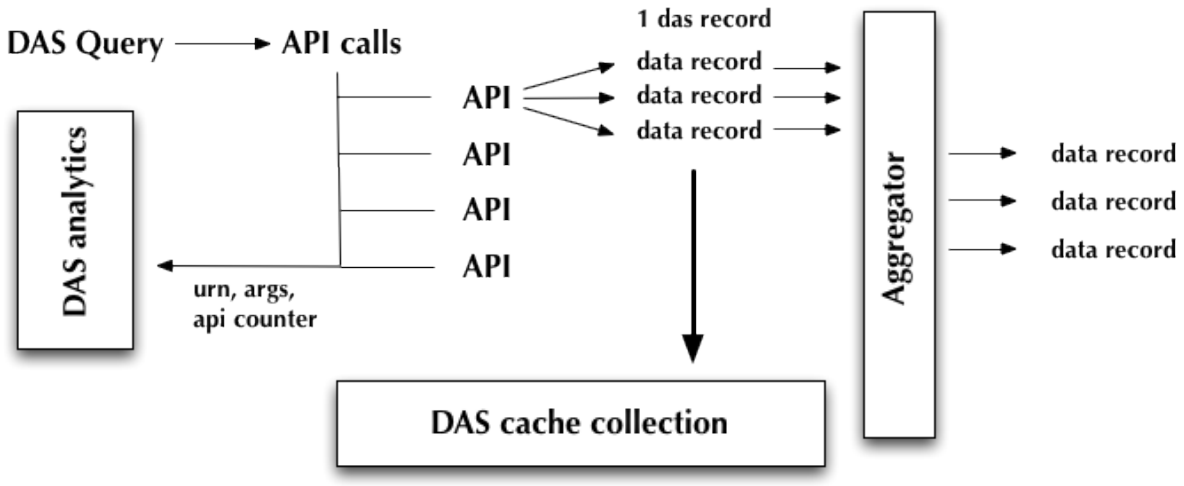
The first has the disadvantage of not being able to easily tell what this object represents, whereas this is not the case for the latter. It is clear in that case that the object principally represents a *dataset*. However, all good things come with a cost; the hierarchical structures are much more expensive to parse, both in python and MongoDB. In DAS we store objects in hierarchical structures, but try to minimize the nesting depth. This allows us to talk about the key/attributes of the object in a more natural way, and simplifies their aggregation. For instance, if two different data-providers serve information about files and file objects containing the *name* attribute, it will be trivial to merge the objects based on the *name* value.

DAS records represent meta-data in [JSON] data format. This is a lightweight, near universal format which can represent complex, nested structures via values, dictionaries, lists. JSON is a native JavaScript data format (JavaScript Object Notation), and is represented by the dictionary type in python. A DAS record is just a collection of data supplied by data-services which participate in DAS. DAS wraps each data-service record with auxiliary meta-data such as internal ID, references to other DAS records, and a DAS header. The DAS header contains information about underlying API calls made to data-provider. For example:

```
{"query": "{\"fields\": null, \"spec\": {\"block.name\": \"/abc\"}}",
 "_id": "4b6c8919e2194e1669000002",
 "qhash": "aa8bcf183d916ea3befbdfbcbf40940a",
 "das": {"status": "ok", "qhash": "aa8bcf183d916ea3befbdfbcbf40940a",
         "ctime": [0.55365610122680664, 0.54806804656982422],
         "url": ["http://a.v.com/api", "http://c.d.com/api"],
```

```
        "timestamp": 1265404185.2611251,
        "lookup_keys": ["block.name", "file.name", "block.name"],
        "system": ["combined"],
        "services": [{"combined":["dbs", "phedex"]}],
        "record": 0,
        "api": ["blockReplicas", "listBlocks"],
        "expire": 1265407785.2611251}
}
```

DAS workflow produce the following set of records



- data records, which contains data coming out from data-services, every data record contains a pointer to das record
- das records, which contains information how we retrieve data
- analytics records, which contains information about API calls

1.6.4 DAS Analytics DB

DAS Analytics DB keeps information about user queries placed to DAS. Such information is used for pre-fetching strategies and further analysis of user queries.

Analytics DB records

query records

A query record is produced each time a user sends a query to DAS, either by the CLI or the web interface. The record is created when DASCore::adjust_query is called (providing add_to_analytics is enabled):

- dasquery, input DAS query
- dhash, md5 hash of dasquery
- mongoquery, DAS query using MongoDB syntax
- qhash, md5 hash of mongoquery

- time, timestamp query was made

It is possible for these records to be identical except for the timestamp. The time parameter is wanted for analytics, but could alternatively be stored as an array attached to an otherwise unique record, if necessary in future.

Here is an example of query record

```
{u'_id': ObjectId('4b4f4caee2194e72ae000001'),
 u'dasquery': u'site = T1_CH_CERN',
 u'dhash': u'7f0a8d3f0e44f35b72f504fcb77482b7',
 u'mongoquery': u'{"fields": null, "spec": {"site.name": "T1_CH_CERN"}}',
 u'qhash': u'5e0dbc2a8e523e0ca401a42a8868f139',
 u'time': 123456789.0}
```

api records

Non-persistent API records are used to keep track of API calls within DAS. Each record contains:

- apicall, a dictionary of API parameters
  - api, name of the data-service API
  - api_params, a dictionary of API input parameters
  - expire, expiration timestamp for API call
  - system, name of data-service
  - url, the URL of the invoked data-service, please note it may or may not be a full URL to the relevant api (which may be part of the query parameters).

Here is an example of API record

```
{u'_id': ObjectId('4b4f4cb0e2194e72b2000003'),
 u'apicall': {u'api': u'CMStoSiteName',
              u'api_params': {u'name': u'T1_CH_CERN'},
              u'expire': 1263531376.068213,
              u'system': u'sitedb',
              u'url': u'https://cmsweb.cern.ch/sitedb/json/index/CMStoSiteName'}}
```

These records are used by DAS to check availability of meta-data provided by this API call. So if user placed a request which can be covered by any API records whose parameters are superset of input query, we look-up this by using API records.

These records are deleted from analytics DB once apicall.expire passes.

api counter records

DAS keeps track of API calls. For that we have a separate persistent records with the following keys

- api, a dictionary of API name and its input parameters
- counter, an incremental counter for this API call
- qhash, corresponding md5 mongoquery hash
- system, a data-service name

```
{u'_id': ObjectId('4b4f4cb0e2194e72b2000000'),
 u'api': {u'name': u'CMStoSiteName', u'params': {u'name': u'T1_CH_CERN'}},
 u'counter': 1,
 u'qhash': u'5e0dbc2a8e523e0ca401a42a8868f139',
 u'system': u'sitedb'}
```

1.6.5 DAS raw cache

The DAS raw cache holds all *raw* data-service output, converted into uniform DAS records.

DAS cache records

The DAS cache keeps metadata for the data-services in a form of DAS record. Each data-service output is converted into a DAS record according to *DAS mapping*.

data records

Each data record contains data-service meta-data. The structure of the data service response is a-priori unknown to DAS. Since DAS operates with JSON almost any data structure can be stored into DAS, e.g. dictionaries, lists, strings, numerals, etc. The only fields DAS appends to the record are:

- das, contains DAS expiration timestamp
- das_id, refers to query2apis data record
- primary_key, provide a primary key to the stored data

For example, here is a data record from SiteDB

```
{u'_id': ObjectId('4b4f4cb0e2194e72b2000002'),
 u'das': {u'expire': 1263531376.062233},
 u'das_id': u'4b4f4cb0e2194e72b2000001',
 u'primary_key': u'site.name',
 u'site': {u'name': u'T1_CH_CERN', u'sitename': u'CERN'}}
```

query2apis records

This type of DAS record contains information about underlying API calls made by DAS upon provided user query. It contains the following keys

- das, a dictionary of DAS operations
  - api, a list of API calls made by DAS upon provided user query
  - ctime, a call time spent for every API call
  - expire, a shortest expiration time stamp among all API calls
  - lookup_keys, a DAS look-up key for provided user query
  - qhash, a md5 hash of input query (in MongoDB syntax)
  - status, a status request field
  - system, a corresponding list of data-service names
  - timestamp, a timestamp of last API call

  - url, a list of URLs for API calls
  - version, reserved for future use

- query, an input user query in a form of MongoDB syntax

Here is an example query2apis record for the following user input *site=T1_CH_CERN*

```
{u'_id': ObjectId('4b4f4cb0e2194e72b2000001'),
 u'das': {u'api': [u'CMStoSiteName',
                   u'CMStoSiteName',
                   u'CMStoSAMName',
                   u'CMSNametoAdmins',
                   u'CMSNametoSE',
                   u'SEtoCMSName',
                   u'CMSNametoCE',
                   u'nodes',
                   u'blockReplicas'],
          u'ctime': [1.190140962600708,
                     1.190140962600708,
                     0.71966314315795898,
                     0.72777295112609863,
                     0.7784569263458252,
                     0.75019693374633789,
                     0.74393796920776367,
                     0.28762698173522949,
                     0.30852007865905762],
          u'expire': 1263489980.1307981,
          u'lookup_keys': [u'site.name'],
          u'qhash': u'5e0dbc2a8e523e0ca401a42a8868f139',
          u'status': u'ok',
          u'system': [u'sitedb', u'sitedb', u'sitedb', u'sitedb',
                      u'sitedb', u'sitedb', u'sitedb', u'phedex', u'phedex'],
          u'timestamp': 1263488176.062233,
          u'url': [u'https://cmsweb.cern.ch/sitedb/json/index/CMStoSiteName',
                   u'https://cmsweb.cern.ch/sitedb/json/index/CMStoSiteName',
                   u'https://cmsweb.cern.ch/sitedb/json/index/CMStoSAMName',
                   u'https://cmsweb.cern.ch/sitedb/json/index/CMSNametoAdmins',
                   u'https://cmsweb.cern.ch/sitedb/json/index/CMSNametoSE',
                   u'https://cmsweb.cern.ch/sitedb/json/index/SEtoCMSName',
                   u'https://cmsweb.cern.ch/sitedb/json/index/CMSNametoCE',
                   u'http://cmsweb.cern.ch/phedex/datasvc/xml/prod/nodes',
                   u'http://cmsweb.cern.ch/phedex/datasvc/xml/prod/blockReplicas'],
          u'version': u''},
 u'query': u'{"fields": null, "spec": {"site.name": "T1_CH_CERN"}}'}
```

1.6.6 DAS merge cache

The DAS merge cache is used to keep merged (aggregated) information from multiple data-service responses. For example, if service A and service B return documents

```
{'service':'A', 'foo':1, 'boo':[1,2,3]}
{'service':'B', 'foo':1, 'data': {'test':1}}
```

the DAS will merge those documents based on the common key, *foo* and resulting merged (aggregated) document will be of the following form:

```
{'service':['A','B'], 'foo':1, 'boo':[1,2,3], 'data':{'test':1}}
```

But of course DAS provides much more then just merging the document content. Below are more concrete example of merged CMS records.

DAS merge records

DAS merge records represent aggregated results made by DAS upon user input query. Each query contains

- das, an expiration timestamp, based on shortest expire timestamps of corresponding *data records*
- das_id, a list of corresponding _id's of *data records* used for this aggregation
- an aggregated data-service part, e.g. site.

```
{u'_id': ObjectId('4b4f4cb4e2194e72b2000033'),
 u'das': {u'expire': 1263489980.1307981},
 u'das_id': [u'4b4f4cb0e2194e72b2000001',
             u'4b4f4cb0e2194e72b2000001',
             u'4b4f4cb0e2194e72b2000001',
             u'4b4f4cb0e2194e72b2000001',
             u'4b4f4cb0e2194e72b2000001',
             u'4b4f4cb0e2194e72b2000001',
             u'4b4f4cb0e2194e72b2000001',
             u'4b4f4cb0e2194e72b2000001',
             u'4b4f4cb0e2194e72b2000001',
             u'4b4f4cb0e2194e72b2000001',
             u'4b4f4cb0e2194e72b2000001',
             u'4b4f4cb0e2194e72b2000001',
             u'4b4f4cb0e2194e72b2000001',
             u'4b4f4cb0e2194e72b2000001',
             u'4b4f4cb0e2194e72b2000001',
             u'4b4f4cb0e2194e72b2000001',
             u'4b4f4cb0e2194e72b2000001',
             u'4b4f4cb0e2194e72b2000001',
             u'4b4f4cb0e2194e72b2000001',
             u'4b4f4cb0e2194e72b2000001',
             u'4b4f4cb0e2194e72b2000001',
             u'4b4f4cb0e2194e72b2000001',
             u'4b4f4cb0e2194e72b2000001',
             u'4b4f4cb0e2194e72b2000001',
             u'4b4f4cb0e2194e72b2000001',
             u'4b4f4cb0e2194e72b2000001',
             u'4b4f4cb0e2194e72b2000001',
             u'4b4f4cb0e2194e72b2000001',
             u'4b4f4cb0e2194e72b2000001',
             u'4b4f4cb0e2194e72b2000001',
             u'4b4f4cb0e2194e72b2000001',
             u'4b4f4cb0e2194e72b2000001',
             u'4b4f4cb0e2194e72b2000001',
             u'4b4f4cb0e2194e72b2000001'],
 u'site': [{u'ce': u'ce126.cern.ch', u'name': u'T1_CH_CERN'},
           {u'ce': u'ce201.cern.ch', u'name': u'T1_CH_CERN'},
           {u'ce': u'ce131.cern.ch', u'name': u'T1_CH_CERN'},
           {u'ce': u'ce103.cern.ch', u'name': u'T1_CH_CERN'},
           {u'ce': u'ce128.cern.ch', u'name': u'T1_CH_CERN'},
           {u'admin': {u'email': u'Josh.Bendavid@cern.ch', u'forename': u'Josh',
```

```
            u'surname': u'Bendavid', u'title': u'Data Manager'},
 u'name': u'T1_CH_CERN'},
{u'admin': {u'email': u'gowdy@cern.ch', u'forename': u'Stephen',
            u'surname': u'Gowdy',u'title': u'T0 Operator'},
 u'name': u'T1_CH_CERN'},
{u'admin': {u'email': u'gowdy@cern.ch', u'forename': u'Stephen',
            u'surname': u'Gowdy', u'title': u'Site Executive'},
 u'name': u'T1_CH_CERN'},
{u'admin': {u'email': u'gowdy@cern.ch', u'forename': u'Stephen',
            u'surname': u'Gowdy', u'title': u'Data Manager'},
 u'name': u'T1_CH_CERN'},
{u'admin': {u'email': u'gowdy@cern.ch', u'forename': u'Stephen',
            u'surname': u'Gowdy', u'title': u'Site Admin'},
 u'name': u'T1_CH_CERN'},
{u'admin': {u'email': u'dmason@fnal.gov', u'forename': u'David',
            u'surname': u'Mason', u'title': u'Data Manager'},
 u'name': u'T1_CH_CERN'},
{u'ce': u'ce132.cern.ch', u'name': u'T1_CH_CERN'},
{u'ce': u'ce130.cern.ch', u'name': u'T1_CH_CERN'},
{u'ce': u'ce127.cern.ch', u'name': u'T1_CH_CERN'},
{u'name': u'T1_CH_CERN', u'samname': u'CERN-PROD'},
{u'name': u'T1_CH_CERN', u'sitename': u'CERN'},
{u'admin': {u'email': u'Victor.Zhiltsov@cern.ch', u'forename': u'Victor',
            u'surname': u'Zhiltsov', u'title': u'Data Manager'},
 u'name': u'T1_CH_CERN'},
{u'admin': {u'email': u'Peter.Kreuzer@cern.ch', u'forename': u'Peter',
            u'surname': u'Kreuzer', u'title': u'Site Admin'},
 u'name': u'T1_CH_CERN'},
{u'ce': u'ce125.cern.ch', u'name': u'T1_CH_CERN'},
{u'ce': u'ce112.cern.ch', u'name': u'T1_CH_CERN'},
{u'ce': u'ce129.cern.ch', u'name': u'T1_CH_CERN'},
{u'ce': u'ce133.cern.ch', u'name': u'T1_CH_CERN'},
{u'ce': u'ce202.cern.ch', u'name': u'T1_CH_CERN'},
{u'ce': u'ce106.cern.ch', u'name': u'T1_CH_CERN'},
{u'ce': u'ce105.cern.ch', u'name': u'T1_CH_CERN'},
{u'ce': u'ce111.cern.ch', u'name': u'T1_CH_CERN'},
{u'ce': u'ce104.cern.ch', u'name': u'T1_CH_CERN'},
{u'ce': u'ce113.cern.ch', u'name': u'T1_CH_CERN'},
{u'ce': u'ce107.cern.ch', u'name': u'T1_CH_CERN'},
{u'ce': u'ce114.cern.ch', u'name': u'T1_CH_CERN'},
{u'ce': u'ce124.cern.ch', u'name': u'T1_CH_CERN'},
{u'admin': {u'email': u'Peter.Kreuzer@cern.ch', u'forename': u'Peter',
            u'surname': u'Kreuzer', u'title': u'Site Executive'},
 u'name': u'T1_CH_CERN'},
{u'admin': {u'email': u'Christoph.Paus@cern.ch', u'forename': u'Christoph',
            u'surname': u'Paus', u'title': u'Data Manager'},
 u'name': u'T1_CH_CERN'},
{u'admin': {u'email': u'ceballos@cern.ch', u'forename': u'Guillelmo',
            u'surname': u'Gomez-Ceballos', u'title': u'Data Manager'},
 u'name': u'T1_CH_CERN'}]}
```


1.6.7 DAS logging DB

The DAS logging DB holds information about all requests made to DAS. All records are stored in a capped collection of MongoDB. Capped collections are fixed sized collections that have a very high performance auto-LRU age-out feature (age out is based on insertion order) and maintain insertion order for the objects in the collection.

Logging DB records


```
{"args": {"query": "site=T1_CH_CERN"},
 "qhash": "7c8ba62a07ff2820c217ae3b51686383", "ip": "127.0.0.1",
 "hostname": "", "port": 65238,
 "headers": {"Remote-Addr": "127.0.0.1", "Accept-Encoding": "identity",
             "Host": "localhost:8211", "Accept": "application/json",
             "User-Agent": "Python-urllib/2.6", "Connection": "close"},
 "timestamp": 1263488174.364929,
 "path": "/status", "_id": "4b4f4caee2194e72ae000003", "method": "GET"}
```


1.6.8 DAS analytics

DAS analytics subsystem provides a task scheduler for running analytic or maintenance tasks at regular intervals on the DAS server. It does not need to run on the same machine as other DAS components, but does require access to the MongoDB datastore.

analytics_controller

The main class of the analytics system is analytics_controller. This is invoked with `das_analytics [options] [config file(s)]`. All options can be specified either in a configuration file (executed as python source, so global variables in the file become configuration options) or specified on the command line (`--help` will list the available options). Options on the command line take precedence. Jobs can only be specified in a command line. Upon start, the server blocks until killed (rather than daemonising - this does not appear to agree with the python multiprocessing module).

The controller runs a task scheduler, which handles the execution of tasks, and a web server, which provides the principal information and control interface for the analytics system.

The controller creates a mongo capped collection in which to store the results of the tasks it runs, and the log messages generated by tasks and the analytics system itself. This defaults to 64MiB in size.

scheduler

The scheduler owns a pool of worker processes, and submits jobs from an internal queue to the pool at their submission time. The jobs then run and on completion will either be automatically rescheduled based on their requested interval, can request a custom resubmission time and can also create new child tasks. Jobs that fail are automatically retried a small number of times before being abandoned. A minimum submission interval is enforced to try and prevent badly configured jobs performing a DoS attack on the server. The scheduler runs in a separate thread.

webserver

The analytics webserver is run automatically on starting the server. By default, it serves from http://host:8213/analytics/. It does not include any security itself, and contains control functions (add/remove task, alter configuration, etc) which should not be generally available, so should be accessible only by SSH forwarding or similar.

The webserver allows one to: * View the current schedule, tasks currently executing and change the resubmission time. * View the raw output of recently run tasks, and the configuration of each task. * Reports showing better presented outputs from tasks or the system in general. * Control the server, either changing the system configuration or adding new tasks. * View a small amount of online help.

tasks

An analytics task is a python class which will be run at regular intervals by the analytics server. Tasks receive a set of standard arguments at init in addition to those specified when the job was configured, then are called with no arguments to run. They should then return a dictionary containing any information they wish to make visible on the webserver, along with special arguments such as a requested resubmission time or new tasks they wish to spawn.

Tasks are specified in a python configuration file like:

```
Task(name, classname, interval, **kwargs)
```

where *kwargs* include any task-specific arguments necessary, *classname* is the name of a class in DAS.analytics.tasks and *name* is a display identifier for the task.

Additionally, you can supply special kwargs *only_once*, *only_before*, *max_runs* to limit the task running. Time-like arguments are given as seconds in the unix time convention.

When the task is called, it receives a kwargs dictionary containing

logger a logger with methods info, debug, warning, error, critical as per the logging package (but not identical, since it is a custom class that passes log messages through a pipe back to the controller process)

DAS a DASCore instance (currently one-per-process is created, in future this may change to a single global instance to avoid connection overhead, rationale for one-per-job is that a separate logger can be supplued in each case)

name name of this task

index the number of times this task has run

interval interval of the task in seconds

plus any *kwargs* given when the task was defined. The task should use the supplied logger to record any log messages (which will then be stored along with the task result.

The actual task should be carried out in the *__call__* method of the class object, which should finally return a dictionary containing any useful information, and any special information the task wishes to pass back to the scheduler.

The following special values are understood in the return dictionary:

resubmit set to False to disable resubmission

next set to a (GMT) time to be resubmitted then instead of after the normal interval

new_tasks set to a list of dictionaries to create those tasks. Generally, child tasks should be created with 'only_before' set to the next anticipated run of this task, since no interaction between parents and children in currently supported

Upon startup, the server indexes all classes in DAS.analytics.tasks searching for task classes, but it is recommended that tasks be placed in a file of the same name. The indexer will look up the following attributes if they exist.

- task_title: display name for the task (default task.__class__.__name__)
- task_info: longer text description (default task.__doc__)
- **task_options: list of dictionaries, each containing string attributes name, default, type, help** used by the web interface to make adding tasks easier
- hide: don't index this task

standalone tasks

Tasks can be run independently of the analytics scheduler with the script *das_analytics_task*. This will run the task as normal and report if new tasks were requested or resubmission altered, but obviously no new tasks will actually be spawned. Useful for testing or running one-off jobs. Example, to run the test task:

```
das_analytics_task -c Test -o "{'message': 'hello world'}"
```

cli interface

A CLI script, *das_analytics_cli* is provided which allows some of the webserver control options to be triggered without a web browser. This still makes HTTP calls so requires the web server to be running and reachable.

reports

Reports are classes that are intended to provide richer views of the output of either tasks, analytics in general or DAS in general. A report consists of a class in DAS.analytics.reports (optionally inheriting from DAS.analytics.utils.Report), and one or more cheetah templates used for presenting information.

Code in a report class is executed by the web thread handling the request, and so should involve a minimum of actual computation. If computation is required for a report, a task should be created which runs intermittently performing the computation and returning the result, which can then be retrieved by the report from the analytics mongodb collection.

The __call__ method of the report should return a two-tuple of the name of the template desired and a dictionary required by cheetah to format the template. The __call__ method receives any extra parameters to the report URL as kwargs.

Reports are indexed similarly to tasks, with the following parameters understood.

- report_group: For grouping reports (default "general")
- report_title: Display name (default report.__class__.__name__)
- report_info: Longer information (default report.__doc__)
- hide: Don't index this class.

By convention, the templates should be called analytics_report_<reportname>.tmpl

Templates wishing to generate plots can generate URLs to a plotfairy instance, if the location of one has been supplied to analytics. See existing examples.

implemented tasks

Test

message Some text to print

Designed to test the scheduler, this will print a message each time it runs then randomly prevent resubmission, submit extra copies of itself, raise exceptions, etc etc.

QueryRunner

query A query in text or mongo dictionary format (passed to DASCore::call)

This is a simple query issuer that runs at a fixed interval, making a call to DAS. This will guarantee the data is in the cache at call time, but not that it is renewed (so it may expire 30 seconds later).

QueryMaintainer

query A query in mongo *storage* format (ie, `spec: [{'key':..., 'value':...}]`). This should be a simple, single-argument query with no modifiers, eg `{'fields': None, 'spec': [{'key':'dataset.name', 'value':'xyz'}]}`

preempt Time before data expiry it should be renewed. Default is 5 minutes. If the analytics server is busy (all workers filled), jobs may not run at submission time.

QueryMaintainer performs a (currently non-atomic) update of the requested query each time it runs, looks up the earliest expiry time of the resulting data, and reschedules itself to run next *preempt* seconds before this time.

The update is currently performed by calling *remove_from_cache* and then *call*, but this should be replaced by an atomic update when possible.

It may also be worth producing an analogous class for individual API calls, since some data services will have different expiry times and replacing everything after the first expiry may not be optimum behaviour.

HotspotBase

fraction proportion of items to maintain, interpreted different depending on *mode*

mode item selection metric, currently supports

calls select the items representing the top *fraction* of all calls made

keys select the top *fraction* of all items, sorted by number of calls made

fixed select the first *fraction* items, sorted by number of calls made (in this mode *fraction* should be > 1)

period period for consideration, default 1 month

allowed_gap length of gap in summary documents that can be ignored (should be << *period* and < *interval*)

HotspotBase is a base class that should not be instantiated directly. It provides a framework for analysis where short periods of time are analysed, to produce some sort of item->count mapping, and then these summary documents are averaged over a much longer period to determine the most popular items in this period according to some metric, which can then be used to inform pre-fetch strategies.

Users should implement:

```
generate_task(self, item, count, epoch_start, epoch_end)

    Return a new task dictionary for the selected item, eg
    a QueryMaintainer task if queries are the items being selected

make_one_summary(self, start, finish)

    Return an item->count mapping for the start and end times specified
```

and may also implement:

```
report(self)

    Return a dictionary of extra keys to go in the return value

preselect_items(self, items)

    Remove unwanted keys (eg, those containing wildcards if you want to
    not consider them), then return items

mutate_items(self, items)

    Perform any merging of selected keys, eg determining which queries
    are supersets of each other, then return items
```

ValueHotspot

key A DAS key that we want to examine the values of, eg *dataset.name*

preempt Passed to spawned *QueryMaintainer* tasks

allow_wildcarding Whether to include values containing wildcards

find_supersets Attempt to find superset queries of selected queries **experimental**

ValueHotspot considers the values given to a particular DAS key (no-condition queries are ignored). Each time it runs, it performs a moving average over the past *period* (default 1 month, from *HotspotBase*) to find the most requested *fraction* of values.

For each of these values, a QueryMaintainer task is spawned, with *only_before* set to the next run of the *ValueHotspot* instance.

Hence, the most popular queries for a given DAS key are kept in the cache.

KeyHotspot

child_interval Interval for spawned tasks

child_preempt Passed to spawned tasks as *preempt*

KeyHotspot is intended for very infrequent running (eg daily or less), to determine which DAS keys are currently being most used.

For each of the most used keys, a ValueHotspot instance is spawned to run until the next run of this instance, which will in turn spawn QueryMaintainers to keep the most popular values for that query in the cache.

QueryAnalyzer

Generates summary information about the types of queries that the DAS server has executed for the query report.

LifetimeMonitor

NotImplemented. Was intended to pull the data from random queries, and monitor over what period of time it actually changed for the purposes of dynamically setting data lifetimes rather than using default values.

1.6.9 Code documentation

DAS Analytics classes

Analytics cli module

Analytics config module

Analytics controller module

Analytics schedule module

Analytics task module

Analytics utils module

Analytics web module

Analytics task module

Analytics HotspotBase module

Base class for scheduling analyzers

class DAS.analytics.tasks.hotspot_base.**HotspotBase**(***kwargs*)

This is a base-class for periodically-running analyzers that want to examine the moving average of some key->counter map, and pick the top few for further attention.

DASQueries are extracted from analytics DB. The selected items are passed to generate_task callback implemented in subclasses. It look-up DAS query expiration timestamp and if necessary calls DAS to get it (along with results of the query).

generate_task(*item*, *count*, *epoch_start*, *epoch_end*)

For the given selected key, generate an appropriate task dictionary as understood by taskscheduler.

Should be a generator or return an iterable

get_all_items(*summaries*)

Merge the summary dictionaries.

get_summaries(*epoch_start*, *epoch_end*)

Fetch all the available pre-computed summaries and determine if any need to be constructed at this time.

make_one_summary(*start*, *finish*)

Actually make a summary of item->count pairs for the specified time range. Subclasses need to implement this for the analysis in question.

make_summary(*start*, *finish*)

Split the summarise requests into interval-sized chunks and decide if they're necessary at all.

mutate_items(*items*)

This is a last part of selection chain.

Optionally, mutate the selected items. A subclass wishing to merge together keys should do so here.

preselect_items(*items*)
This is a part of selection chain.

Optionally, preselect the items for consideration. A subclass wishing to exclude certain key types could do so here (but could also do so in make_one_summary).

This is a good place to implement clustering algorithm for selected items. For example, if several queries are selected, we may analyze who has more weight and only pass those for task generation step.

report()
Generate some extra keys to go in the job report, if desired.

select_items(*items*)
Take a mapping of item->count pairs and determine which are "hot" based on the selected mode.

Analytics KeyHotspot module

KeyHotspot analyzer

class DAS.analytics.tasks.key_hotspot.**KeyHotspot**(***kwargs*)
This analyzer runs infrequently (~daily) to determine based on a moving average of the whole period which keys (not their values) are most requested, and uses this to start several child ValueHotspot instances to monitor those keys, using the HotspotBase metrics.

generate_task(*item*, *count*, *epoch_start*, *epoch_end*)
Generate task

make_one_summary(*start*, *finish*)
Actually make the summary

Analytics KeyLearning module

Analytics QueryMaintainer module

Analytics QueryRunner module

Analytics ValueHotspot task

DAS Core classes

DAS aggregators

DAS analytics module

DAS core module

DAS key learning module

DAS mapping module

DAS mongo cache module

DAS parser

DAS parser cache module

DAS QL module

DAS Query module

DAS robot module

DAS SON manipulator

DAS services

DAS operates using a provided list of data-services and their definitions. The nature of these services is unimportant provided that they provide a some form of API which DAS can call and aggregate the data returned. Each service needs to be registered in DAS by providing an appropriate configuration file and (optionally) a service handler class..

CMS services

Each CMS data-service is represented by a mapping and, optionally, by a plugin class. The data-service map contains description of the data-service, e.g. URL, URN, expiry timeout as well as API and notations maps.

- the API map relates DAS keys and API input parameters. It contains the following items:
  - *api*, name of the API

  - *params*, a list of API input parameters together with regex patterns accepted to check the format of or identify ambiguous values.
  - *record* represents DAS record. Each record has
    - *daskeys*, a list of DAS maps; each map relates keys in the user query to the appropriate DAS representation
      - *key*, a DAS key used in DAS queries, e.g. *block*
      - *map*, a DAS record representation of the *key*, e.g. *block.name*
      - *pattern*, a regex pattern for DAS key
    - *das2api*, is a map between DAS key representations and API input parameters
      - *api_param*, an API input parameter, e.g. *se*
      - *das_key*, a DAS key it represents, e.g. *site.se*
      - *pattern*, a regex pattern for *api_param*
- Notation map represents a mapping between data-service output and DAS records. It is optional.

Please use these links *API map* and *API notation* for concrete examples.

DAS abstract service

DAS generic service

DAS map reader module

DAS tools

DAS provides a useful set of tools.

DAS config tools

DAS config generator

class DAS.tools.create_das_config.**ConfigOptionParser**
option parser

get_opt()
Returns parse list of options

DAS.tools.create_das_config.**main**()
Main function

DAS admin tools

DAS stress test module

DAS bench tools

DAS benchmark tool

class DAS.tools.das_bench.**NClientsOptionParser**
client option parser

get_opt()
Returns parse list of options

class DAS.tools.das_bench.**UrlRequest**(*method*, **args*, ***kwargs*)
URL requestor class which supports all RESTful request methods. It is based on urllib2.Request class and overwrite request method. Usage: UrlRequest(method, url=url, data=data), where method is GET, POST, PUT, DELETE.

get_method()
Return request method

DAS.tools.das_bench.**avg_std**(*input_file*)
Calculate average and standard deviation

DAS.tools.das_bench.**gen_passwd**(*length=8*, *chars='ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0*
Random string generator, code based on http://code.activestate.com/recipes/59873-random-password-generation/

DAS.tools.das_bench.**main**()
Main routine

DAS.tools.das_bench.**make_plot**(*xxx*, *yyy*, *std=None*, *name='das_cache.pdf'*, *xlabel='Number of clients'*, *ylabel='Time/request (sec)'*, *yscale=None*, *title=''*)
Make standard plot time vs nclients using matplotlib

DAS.tools.das_bench.**natcasecmp**(*aaa*, *bbb*)
Natural string comparison, ignores case.

DAS.tools.das_bench.**natcmp**(*aaa*, *bbb*)
Natural string comparison, case sensitive.

DAS.tools.das_bench.**natsort**(*seq*, *icmp=<function natcmp at 0x361baa0>*)
In-place natural string sort.

DAS.tools.das_bench.**natsort_key**(*sss*)
Used internally to get a tuple by which s is sorted.

DAS.tools.das_bench.**natsorted**(*seq*, *icmp=<function natcmp at 0x361baa0>*)
Returns a copy of seq, sorted by natural string sort.

DAS.tools.das_bench.**random_index**(*bound*)
Generate random number for given upper bound

DAS.tools.das_bench.**runjob**(*nclients*, *host*, *method*, *params*, *headers*, *idx*, *limit*, *debug=0*, *logname='spammer'*, *dasquery=None*)
Run spammer for provided number of parralel clients, host name and method (API). The output data are stored into lognameN.log, where logname is an optional parameter with default as spammer.

DAS.tools.das_bench.**spammer**(*stream*, *host*, *method*, *params*, *headers*, *write_lock*, *debug=0*)
start new process for each request

DAS.tools.das_bench.**try_int**(*sss*)
Convert to integer if possible.

DAS.tools.das_bench.**urlrequest**(*stream*, *url*, *headers*, *write_lock*, *debug=0*)
URL request function

DAS CLI tool

DAS client

DAS command line tool

class DAS.tools.das_client.**DASOptionParser**
DAS cache client option parser

get_opt()
Returns parse list of options

class DAS.tools.das_client.**HTTPSClientAuthHandler**(*key=None*, *cert=None*, *level=0*)
Simple HTTPS client authentication class based on provided key/ca information

get_connection(*host*, *timeout=300*)
Connection method

https_open(*req*)
Open request method

DAS.tools.das_client.**convert_time**(*val*)
Convert given timestamp into human readable format

DAS.tools.das_client.**fullpath**(*path*)
Expand path to full path

DAS.tools.das_client.**get_data**(*host*, *query*, *idx*, *limit*, *debug*, *threshold=300*, *ckey=None*, *cert=None*, *das_headers=True*)
Contact DAS server and retrieve data for given DAS query

DAS.tools.das_client.**get_value**(*data*, *filters*, *base=10*)
Filter data from a row for given list of filters

DAS.tools.das_client.**main**()
Main function

DAS.tools.das_client.**prim_value**(*row*)
Extract primary key value from DAS record

DAS.tools.das_client.**print_summary**(*rec*)
Print summary record information on stdout

DAS.tools.das_client.**size_format**(*uinput*, *ibase=0*)
Format file size utility, it converts file size into KB, MB, GB, TB, PB units

DAS.tools.das_client.**unique_filter**(*rows*)
Unique filter drop duplicate rows.

DAS map reduce tools

DAS mapping db tools

DAS robot tools

DAS utilities

DAS provides a useful set of utilities.

CERN SSO authentication utils

CERN SSO toolkit. Provides get_data method which allow to get data begind CERN SSO protected site.

class DAS.utils.cern_sso_auth.**HTTPSClientAuthHandler**(*key=None*, *cert=None*, *level=0*)
Simple HTTPS client authentication class based on provided key/ca information

get_connection(*host*, *timeout=300*)
Connection method

https_open(*req*)
Open request method

DAS.utils.cern_sso_auth.**get_data**(*url*, *key*, *cert*, *debug=0*)
Main routine to get data from data service behind CERN SSO. Return file-like descriptor object (similar to open).

DAS.utils.cern_sso_auth.**timestamp**()
Construct timestamp used by Shibboleth

DAS config utils

Config utilities

DAS.utils.das_config.**das_configfile**()
Return DAS configuration file name $DAS_ROOT/etc/das.cfg

DAS.utils.das_config.**das_readconfig**(*debug=False*)
Return DAS configuration

DAS.utils.das_config.**das_readconfig_helper**(*debug=False*)
Read DAS configuration file and store DAS parameters into returning dictionary.

DAS.utils.das_config.**read_configparser**(*dasconfig*)
Read DAS configuration

DAS.utils.das_config.**read_wmcore**(*filename*)
Read DAS python configuration file and store DAS parameters into returning dictionary.

DAS.utils.das_config.**wmcore_config**(*filename*)
Return WMCore config object for given file name

DAS.utils.das_config.**write_configparser**(*dasconfig*, *use_default*)
Write DAS configuration file

DAS DB utils

DAS option module

DAS option's class

class DAS.utils.das_option.**DASOption**(*section*, *name*, *itype='string'*, *default=None*, *validator=None*, *destination=None*, *description=''*)
Class representing a single DAS option, independent of storage mechanism. Must define at least a section and name.

In configparser

[section] name = value

In WMCORE config

config.section.name = value

The type parameter forces conversion as appropriate. The default parameter allows the option to not be specified in the config file. If default is not set then not providing this key will throw an exception. The validator parameter should be a single-argument function which returns true if the value supplied is appropriate. The destination argument is for values which shouldn't end up in config[section][name] = value but rather config[destination] = value Description is provided as a future option for some sort of automatic documentation.

get_from_configparser(*config*)
Extract a value from a configparser object.

get_from_wmcore(*config*)
As per get_from_configparser.

write_to_configparser(*config*, *use_default=False*)
Write the current value to a configparser option. This assumes it has already been read or the values in self.value somehow changed, otherwise if use_default is set only those values with defaults are set.

write_to_wmcore(*config*, *use_default=False*)
As per write_to_configparser.

DAS timer module

DAS json wrapper

JSON wrapper around different JSON python implementations. We use simplejson (json), cjson and yajl JSON implementation.

NOTE: different JSON implementation handle floats in different way Here are few examples

..doctest:

```
r1={"ts":time.time()}
print r1
{'ts': 1374255843.891289}
```

Python json:

..doctest:

```
print json.dumps(r1), json.loads(json.dumps(r1))
{"ts": 1374255843.891289} {u'ts': 1374255843.891289}
```

CJSON:

..doctest:: print cjson.encode(r1), cjson.decode(cjson.encode(r1)) {"ts": 1374255843.89} {'ts': 1374255843.89}

YAJL:

..doctest:

```
print yajl.dumps(r1), yajl.loads(yajl.dumps(r1))
{"ts":1.37426e+09} {u'ts': 1374260000.0}
```

Therefore when records contains timestamp it is ADVISED to round it to integer. Then json/cjson implementations will agreee on input/output, while yajl will still differ (for that reason we can't use yajl).

class DAS.utils.jsonwrapper.**JSONDecoder**(***kwargs*)
JSONDecoder wrapper

decode(*istring*)
Decode JSON method

raw_decode(*istring*)
Decode given string

class DAS.utils.jsonwrapper.**JSONEncoder**(***kwargs*)
JSONEncoder wrapper

encode(*idict*)
Decode JSON method

iterencode(*idict*)
Encode input dict

DAS.utils.jsonwrapper.**dump**(*doc*, *source*)
Use json.dump for back-ward compatibility, since cjson doesn't provide this method. The dump method works on file-descriptor objects.

DAS.utils.jsonwrapper.**dumps**(*idict*, ***kwargs*)
Based on default MODULE invoke appropriate JSON encoding API call

DAS.utils.jsonwrapper.**load**(*source*)
Use json.load for back-ward compatibility, since cjson doesn't provide this method. The load method works on file-descriptor objects.

DAS.utils.jsonwrapper.**loads**(*idict*, ***kwargs*)
Based on default MODULE invoke appropriate JSON decoding API call

DAS logger

General purpose DAS logger class. PrintManager class is based on the following work http://stackoverflow.com/questions/245304/how-do-i-get-the-name-of-a-function-or-method-from-within-a-python-function-or-m http://stackoverflow.com/questions/251464/how-to-get-the-function-name-as-string-in-python

class DAS.utils.logger.**NullHandler**(*level=0*)
Do nothing logger

emit(*record*)
This method does nothing.

handle(*record*)
This method does nothing.

class DAS.utils.logger.**PrintManager**(*name='function'*, *verbose=0*)
PrintManager class

debug(*msg*)
print debug messages

error(*msg*)
print warning messages

info(*msg*)
print info messages

warning(*msg*)
print warning messages

DAS.utils.logger.**funcname**()
Extract caller name from a stack

DAS.utils.logger.**print_msg**(*msg*, *cls*, *prefix=''*)
Print message in a form cls::caller msg, suitable for class usage

DAS.utils.logger.**set_cherrypy_logger**(*hdlr*, *level*)
set up logging for CherryPy

DAS pycurl manager

DAS query utils

DAS query utils.

DAS.utils.query_utils.**compare_dicts**(*input_dict*, *exist_dict*)
Helper function for compare_specs. It compares key/val pairs of Mongo dict conditions, e.g. {'$gt':10}. Return true if exist_dict is superset of input_dict

DAS.utils.query_utils.**compare_specs**(*input_query*, *exist_query*)
Function to compare set of fields and specs of two input mongo queries. Return True if results of exist_query are superset of resulst for input_query.

DAS.utils.query_utils.**compare_str**(*query1*, *query2*)
Function to compare string from specs of query. Return True if query2 is supperset of query1. query1&query2 is the string in the pattern:

```
([a-zA-Z0-9_\-\#/*\.])*
```

* is the sign indicates that a sub string of *:

```
([a-zA-Z0-9_\-\#/*\.])*
```

case 1. if query2 is flat query(w/out *) then query1 must be the same flat one

case 2. if query1 is start/end w/ * then query2 must start/end with *

case 3. if query2 is start/end w/out * then query1 must start/end with query2[0]/query[-1]

case 4. query1&query2 both include *
Way to perform a comparision is spliting:

query1 into X0*X1*X2*X3 query2 into Xa*Xb*Xc

foreach X in (Xa, Xb, Xc):

case 5. X is '':

continue

special case:

when X0 & Xa are '' or when X3 & Xc are '' we already cover it in case 2

case 6. X not in query1 then return False

case 7. X in query1 begin at index:

case 7-1. X is the first X not '' we looked up in query1.(Xa) last_idx = index ; continue

case 7-2. X is not the first:

try to find the smallest Xb > Xa if and Only if we could find a sequence:

satisfy Xc > Xb > Xa, otherwise return False '=' will happen when X0 = Xa then we could return True

DAS.utils.query_utils.**convert2pattern**(*query*)
In MongoDB patterns are specified via regular expression. Convert input query condition into regular expression patterns. Return new MongoDB compiled w/ regex query and query w/ debug info.

DAS.utils.query_utils.**decode_mongo_query**(*query*)
Decode query from storage format into mongo format.

DAS.utils.query_utils.**encode_mongo_query**(*query*)
Encode mongo query into storage format. MongoDB does not allow storage of dict with keys containing "." or MongoDB operators, e.g. $lt. So we convert input mongo query spec into list of dicts whose "key"/"value" are mongo query spec key/values. For example

```
spec:{"block.name":"aaa"}

converted into

spec:[{"key":"block.name", "value":'"aaa"'}]
```

Conversion is done using JSON dumps method.

DAS regex expressions

Regular expression patterns

DAS.utils.regex.**word_chars**(*word*, *equal=True*)
Creates a pattern of given word as series of its characters, e.g. for given word dataset I'll get '^d$|^da$|^dat$|^data$|^datas$|^datase$|^dataset$' which can be used later in regular expressions

DAS task manager

DAS threadpool utils

URL utils

Generic utils

DAS web modules

DAS autocomplete module

DAS cms representation module

Set of DAS codes

DAS web status codes

DAS.web.das_codes.**decode_code**(*code*)
Return human readable string for provided code ID

DAS.web.das_codes.**web_code**(*error*)
Return DAS WEB code for provided error string

DAS das representation module

DAS server

DAS test data-service server

DAS web server

DAS web manager

DAS dbs daemon module

DAS web tools

DAS web utils

1.6.10 DAS performance benchmarks

Performance benchmarks

CMS data used for Benchmarks

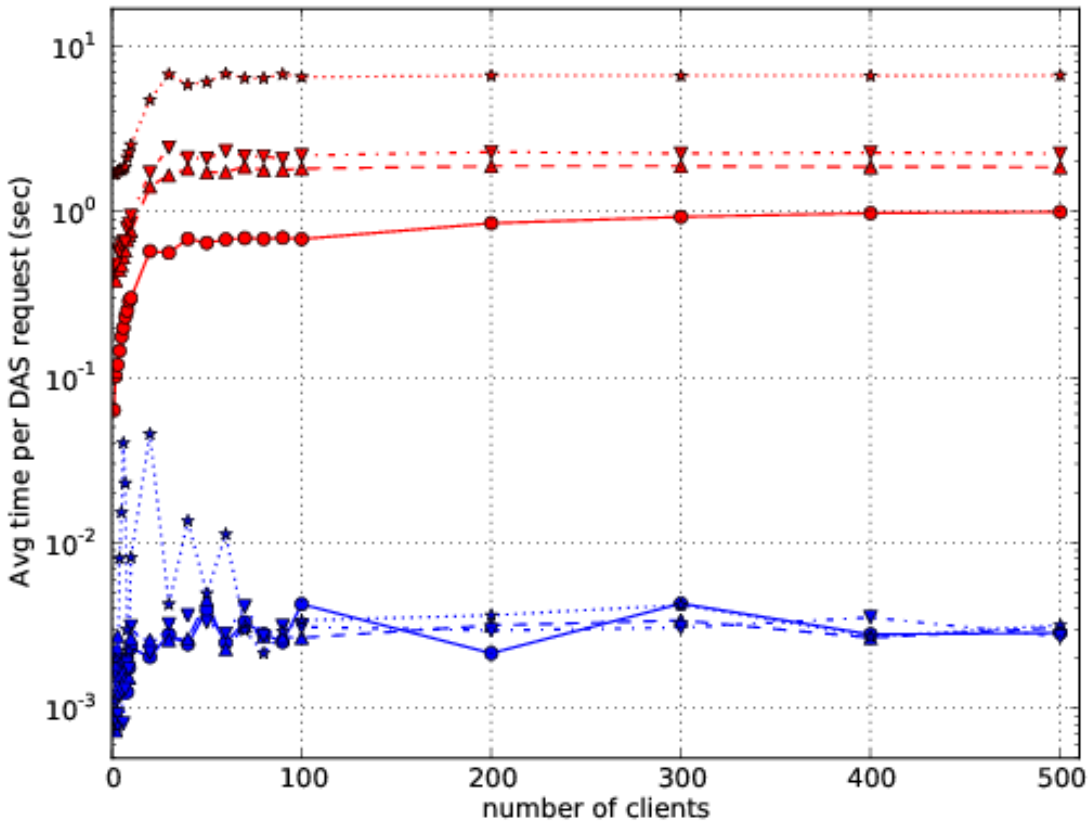
We used ~50K datasets from CMS DBS system and ~450K block records from both DBS and Phedex CMS systems. All of them were populated into DAS cache up-front, since I was only interested in read tests (DAS have an ability to populate the cache). The tests consist of three different types of queries:

- all clients use fixed value for DAS queries, e.g. dataset=/a/b/c or block=/a/b/c#123
- all clients use fixed pattern for DAS queries, e.g. dataset=/a* or block=/a*
- all clients use random patterns, e.g. dataset=/a*, dataset=/Z* or block=/a*, block=/Z*

Once the query has been placed into DAS cache server we retrieve only first record out of the result set and ship it back to the client. The respond time is measured as the total time DAS server spends for a particular query.

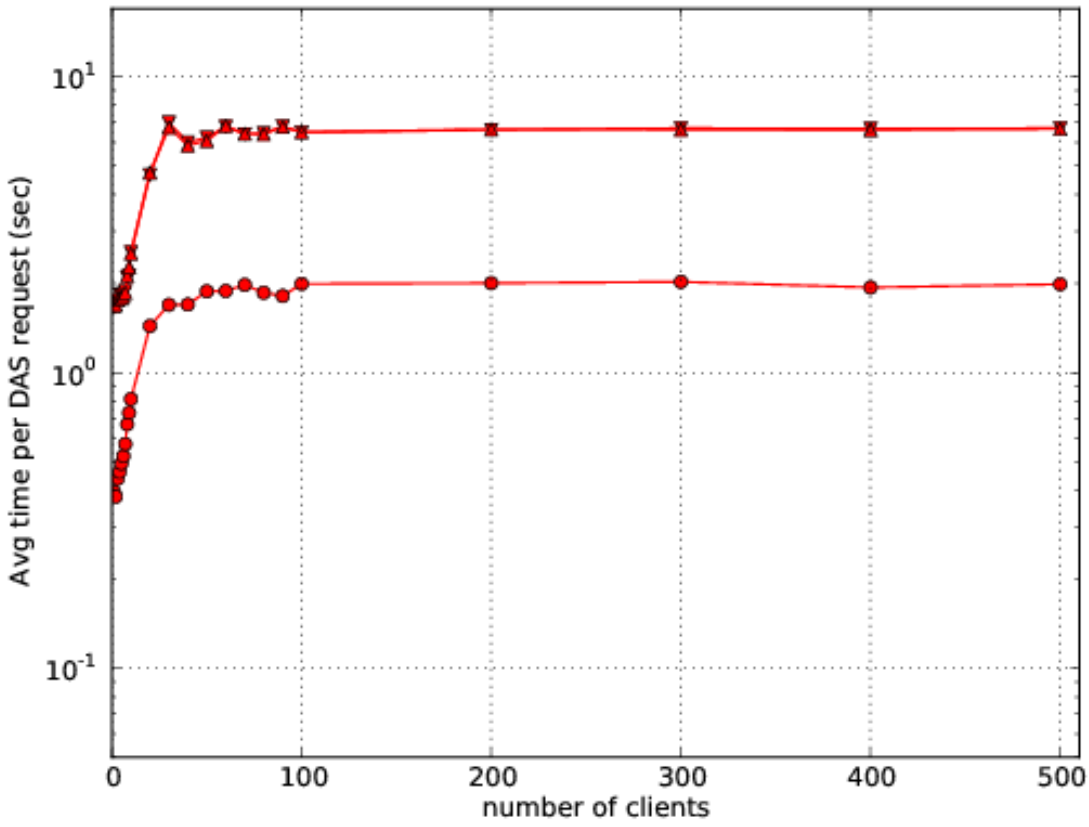
Benchmark results

First, we tested our CherryPy server and verified that it can sustain a load of 500 parallel clients at the level of 10^-5 sec. Then we populated MongoDB with 50k dataset and 500k block records from DBS and Phedex CMS systems. We performed the read test of MongoDB and DAS using 1-500 parallel clients with current set of CMS datasets and block meta-data, 50K and 450K, respectively. Then we populated MongoDB with 10x and 100x of statistics and repeat the tests. The plot showing below represents comparison of DAS (red lines) versus MongoDB (blue lines) read tests for 50k (circles), 500k (down triangles), 5M (up triangles) and 50M (stars):



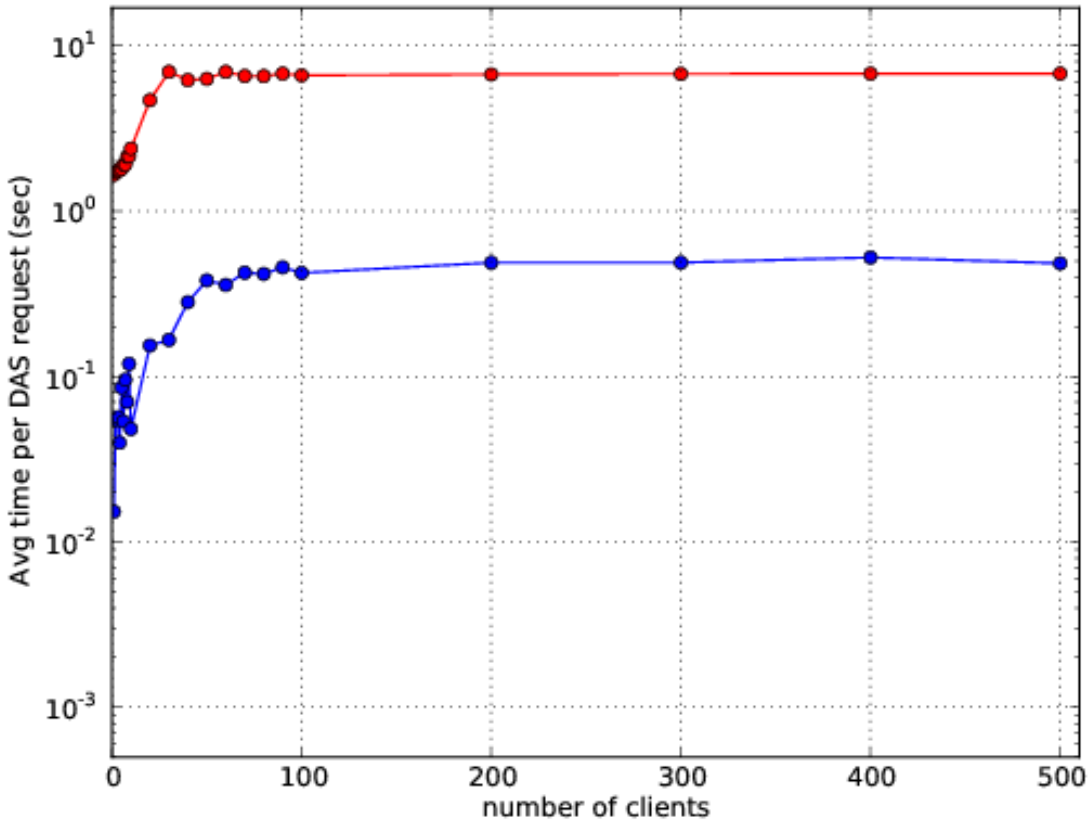
We found these results very satisfactory. As was expected MongoDB can easily sustain such load at the level of few mili-seconds. The DAS numbers also seems reasonable since DAS workflow is much more complicated. It includes DAS parsing, query analysis, analytics, etc. The most important, the DAS performance seems to be driven by MongoDB back-end and has constant scale factor which can be tuned later.

Next we performed three tests discussed above with 10x of block meta-data statistics.


Avg time per DAS request (sec)
10^1
10^0
10^{-1}
0
100
200
300
400
500
number of clients

The curve with circles points represents test #1, i.e. fixed key-value, while top/down triangles represents pattern value and random pattern value, tests #2 and #3, respectively. As can be seen pattern tests are differ by the order of magnitude from fixed key-value, but almost identical among each other.

Finally, we tested DAS/MongoDB with random queries and random data access, by asking to return me a single record from entire collection (not only the first one as shown above). For that purpose we generated a random index and used idx/limit for MongoDB queries. Here is the results



The blue line shows MongoDB performance, while red shows the DAS. This time the difference between DAS and MongoDB is only one order of magnitude differ with respect to first shown plot and driven by DAS workflow.

Benchmarks tool

The DAS provides a benchmarking tool, das_bench

```
das_bench --help

Usage: das_bench [options]

Examples:
  Benchmark Keyword Search:
  das_bench --url=https://cmsweb-testbed.cern.ch/das/kws_async --nclients=20 --dasquery="/DoubleMu/A
```

```
  Benchmark DAS Homepage (repeating each request 10 times):
  src/python/DAS/tools/das_bench.py --url=https://das-test-dbs2.cern.ch/das --nclients=30 --dasquery=

Options:
  -h, --help            show this help message and exit
  -v DEBUG, --verbose=DEBUG
                        verbose output
  --url=URL             specify URL to test, e.g.
                        http://localhost:8211/rest/test
  --accept=ACCEPT       specify URL Accept header, default application/json
  --idx-bound=IDX       specify index bound, by default it is 0
  --logname=LOGNAME     specify log name prefix where results of N client
                        test will be stored
  --nclients=NCLIENTS   specify max number of clients
  --minclients=NCLIENTS specify min number of clients, default 1
  --repeat=REPEAT       repeat each benchmark multiple times
  --dasquery=DASQUERY   specify DAS query to test, e.g. dataset
  --output=FILENAME     specify name of output file for matplotlib output,
                        default is results.pdf, can also be file.png etc
```

which can be used to benchmark DAS.

Performance of individual components (profiling)

DAS has been profiled on vocms67:

- 8 core, Intel Xeon CPU @ 2.33GHz, cache size 6144 KB
- 16 GB of RAM
- Kernel 2.6.18-164.11.1.el5 #1 SMP Wed Jan 20 12:36:24 CET 2010 x86_64 x86_64 x86_64 GNU/Linux

The DAS benchmarking is performed using the following query

```
das_cli --query="block" --no-output
```

Latest results are shown below:

```
...
INFO   0x9e91ea8> DASMongocache::update_cache, ['dbs'] yield 387137 rows
...
INFO   0x9e91ea8> DASMongocache::update_cache, ['phedex'] yield 189901 rows
...
INFO   0x9e91ea8> DASMongocache::merge_records, merging 577038 records

DAS execution time (phedex) 106.446726799 sec
DAS execution time (dbs) 72.2084658146 sec
DAS execution time (merge) 62.8879590034 sec
DAS execution time 241.767010927 sec, Wed, 20 Jan 2010 15:54:33 GMT
```

The largest contributors to execution time are

```
  das_cli --query="block" --verbose=1 --profile --no-output

  Mon Jan 18 22:43:27 2010    profile.dat

       54420138 function calls (54256630 primitive calls) in 247.423 CPU seconds
```

```
 Ordered by: internal time

 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
 560649   78.018    0.000   78.018    0.000 {method 'recv' of '_socket.socket' objects}
    992   23.301    0.023   23.301    0.023 {pymongo._cbson._insert_message}
1627587   19.484    0.000   19.484    0.000 {DAS.extensions.das_speed_utils._dict_handler}
    467   17.295    0.037   21.042    0.045 {pymongo._cbson._to_dicts}
  23773   16.945    0.001   16.945    0.001 {built-in method feed}
 576626   12.101    0.000   77.974    0.000 /data/projects/das/COMP/DAS/src/python/DAS/utils/utils.py
1627587    9.383    0.000   28.867    0.000 /data/projects/das/COMP/DAS/src/python/DAS/utils/utils.py
 969267    7.716    0.000   10.656    0.000 /data/projects/das/slc5_amd64_gcc434/external/py2-pymongo
 392636    4.499    0.000   47.851    0.000 /data/projects/das/COMP/DAS/src/python/DAS/utils/utils.py
      1    3.877    3.877   72.942   72.942 /data/projects/das/COMP/DAS/src/python/DAS/core/das_mongo
1153242    3.644    0.000    5.443    0.000 /data/projects/das/COMP/DAS/src/python/DAS/utils/utils.py
 576626    3.042    0.000   89.345    0.000 /data/projects/das/COMP/DAS/src/python/DAS/core/das_mongo
 576715    3.002    0.000    8.917    0.000 /data/projects/das/slc5_amd64_gcc434/external/py2-pymongo
 969267    2.798    0.000   17.809    0.000 /data/projects/das/slc5_amd64_gcc434/external/py2-pymongo
 ......
```

1.7 Release notes

1.7.1 Release 2.X.Y series

This release series is targeted to DAS production stability and quality.

- 2.5.X
  - Change behavior of DAS CLI to show bytes as is and upon user request its representation in certain base, e.g. –base=2 will show MiB notations, etc.
  - Support DBS3 instances with slashes, e.g. prod/global, int/global
  - Extend list of supported DBS3 instances, in addition to global and phys0[1-3], I added instances with slashes, e.g. int/global. If instances does not have slash it is considered to come from prod DBS URL.
- 2.4.X
  - Re-factor RequestManager code to use internal store instead of MongoDB one
  - Re-evaluate racing conditions:
    - the clean-up worker should use lock to safely wipe out expired records
    - the remove_expired method should only care about given dasquery records
    - add additional delta when check expiration timestamp, it is required to protect code from situation when records can be wiped out during request operation
  - Fixed issues: 4098, 4097, 4090, 4095, 4093, 4089, 4085, 4082, 4081, 4079, 4077
  - Wrapped dasply parser call into spawn function, this fix intermittent problem with dasply under high load
  - Added spawn_manager along with its unit test
  - Re-factor code to use individual DB connections instead of sharing one in various places. This work address DAS instability issue reported in #4024.
  - Added cookie support in DAS client
  - Added support of DAS client version, it is configured via config.web_server.check_clients option

        - check DAS client version in DAS server
        - issue warning status and associative message for the client about version mismatch
    - Added code to re-initiate DAS request in check_pid, issue #4060
    - Prepared server/client for distributed deployment on cmsweb
- 2.3.X
    - Fix issue 4077 (don't yield record from empty MCM response)
    - Work on DAS deployment procedure
        - two set of DAS maps (one for production URLs and another for testbed)
    - Separate DAS web server with KWS one
        - run KWS on dedicated port (8214) with single-threaded DASCore
    - Assign type for records in mapping db; DASMapping code re-factoring to reduce latency of the queries
    - Fix query default assignment (issue #4055)
    - Provide ability to look-up config files used by ReqMgr for dataset creation (issue #4045)
        - Add config look-up for given dataset which bypass MCM data-service
    - Add clean-up worker for DAS caches; fixed #4050
    - Additional work on stability of DAS server under high-load (issue #4024)
        - add exhaust option to all MongoDB find calls
        - add index for mapping db
        - add DAS clean-up daemon, issue #4050
        - reduce pagination usage
        - increase MongoDB pool size, turn on auto_start_request and safe option for MongoClient
        - step away from db connection singleton
    - Add outputdataset API for ReqMgr data-service (issue #4043)
    - Fix python3 warnings
- 2.2.X
    - Fixed MCM prepid issue, switch to produces rest API
    - Merge and integrated KWS search
- 2.1.X
    - Make das-headers mandatory in das_client.py and keep –das-headers option and das_headers input parameter in get_data API for backward compatibility
    - Replaced all has_key dict calls with key in dict statement (work towards python3 standard)
    - Add check_services call to DAS core to check status of services
    - Pass write_concern flag to MongoClient, by default it is off
    - Fixed #4032
    - Re-factor core/web code to propagate error record back to end-user and setup error status code in this case
    - Throw error records when urlfetch_getdata fails

  - Move set_misses into write_to_cache
  - Made adjustments to DBS3 data-service based on recent changes of DBS3 APIs
- 2.0.X
  - Add services attribute to das part of data record, it shows which DAS services were used, while system attribute used to show which CMS systems were used to produce data record(s)
  - Turn off dbs_phedex, it producing too much load, instead use individual services
  - Re-evaluate lifetime of records in DAS cache: the clean-up should be done either for qhash/das.expire pair (less then current tstamp) or for records which live in cache long enough, via das.exire<tstamp-rec_ttl
  - Introduce dasdb.record_ttl configuration parameter int das config
  - Fix issue4023
  - Changes to allow DAS run with DBS2/DBS3 in a mix mode
  - Extend download LFN link to download web page, issue 4022
  - Add Status link to DAS header and let users to see status of DAS queue
  - Re-factor DASMapping code, see ticket 4021
  - Add support for mcm dataset=/a/b/c query; first it looks-up information from ReqMgr to get its info for given dataset, then it parse ReqMgr info and extracts PrepID and passes it to MCM data-service.
  - Add MCM links on dataset summary page when information is provided by reqmgr data-service (MC datasets)
  - Add code to support MCM (PREP) data-service (issue 3449), user can look-up mcm info by using the following query: mcm prepid=<PREP-ID>
  - Remove timestamp attribute from passed dict to md5hash function, it is required due to dynamic nature of timestamp which leads to modification of the hash of the record
  - Add new stress tool, see bin/das_stress_tool
  - Round timestamp for map records as well as for dasheader due to inconsistent behavior of json parsers, see note in jsonwrapper module
  - Fix issue4017: add hash to all DAS map records; add verification of hash into DASMapping check_maps method
  - Fix issue4016: add aux-record called arecord; arecord contains count of corresponding map record, map record type and a system. Adjust DASMapping check_maps method to perform full check of DAS maps by comparing count field from aux-record with actual number of maps in DAS mapping DB
  - Apply common set of indexes for both cache/merge collection to properly get/merge records
  - Allow runs DBS3 API to yield individual records
  - Support block tier=GEN-SIM date between [20120223, 20120224] query via blocksummaries DBS3 API
  - Switch from block_names to block_name as input parameter for blocksummaries DBS3 API; handle correctly incorrect values for dates in DBS3 blocksummaries API
  - Fix issues 4014, 4013, 4009
  - Add lumi4block_run and dataset4block DBS3 APIs
  - fix run input parameter for all DBS3 APIs
  - Add runsummaries API

1.7.2 Older releases

Release 1.X.Y series

- Fix wildcards to provide more informative messages in text mode
- Fix issues: 3997, 3975
- Replace phedex_tier_pattern with phedex_node_pattern
- Get rid of empty_record, query. Instead, introduce das.record with different codes. Codes are defined in utils/utils.py record_codes function. Add mongodb index on codes; modified queries to look-up das/data-records using new das.record field
- Fix issue with ply_query parameter
- Add extra slash to avoid one round trip
- Work on support new run parameter w/ DBS3 APIs, now DAS is capable to use run-range/run-list queries into DBS3
- Use json.dumps to printout JSON dict to stdout
- 1.11.X
  - Add support for block,run,lumi dataset=/a/b/c queries
  - Add plistlib python module w/ None modifications to handle DAS XML output
  - Add list of attributes for config output
  - Add summary4block_run API
  - Highlight unknown global tags in web UI
  - Re-factor the code: add insert_query_records which scan input DAS query and insert query records into DAS cache, then it yields list of acknowledged data-services which used by call API for data retrieval
  - Extend incache API to work with query or data records by providing query_record flag with default value of False (check data records)
  - Take care of potential failure of PLY parser. Use few trials on given input and then give-up
  - Fix bug in task manager when I mix-up return type of spawn function which cause task fails under race conditions
  - Add support for summary dataset=/a/b/c query without run conditions
  - Add support for run range in DBS2 summary dataset/run query
  - Add expand_lumis helper function into das aggregators which flatten lumi lists, e.g. [[1,3], [5,7]] into [1,2,3,5,6,7]. This allows correctly count number of lumis in DAS records
  - Implement support for comp-ops queries, e.g. find run, lumi for given dataset and optional run range find file, lumi for given dataset and optional run range find file, lumi, run for given dataset and optional run range this work is done via new urlfetch_getdata module
- 1.10.X
  - Add urlfetch_pycurl module to fetch content from multiple urls
  - Use custom db_monitor which check MongoDB connection as well as periodically reload DAS maps
  - Add preliminary support for file block=/a/b/c#123 runs site query (must have urlfetch proxy)
  - Allow user to get DBS file into regardless of its status, ticket 3992

  - Add indexes for file.name,dataset.name.block.name and run.run_number in DAS cache collection to prevent error on sorting entities
  - Add support for block dataset run in/between [1,2] query, ticket 3974
  - Apply file.name index to allow MongoDB to sort the files, ticket 3988 this is required in rare case when number of files is very large and MongoDB give up on sorting without the index. I may apply similar index on block as well since their number in dataset can be large as well.
  - Add constrain on block name for lumi block=/a/b/c#123 queries, ticket 3977
  - Add pyurlfetch client
  - Add proxy_getdata to request data from external urlproxy server, ticket 3986; should be used to fetch data concurrently
  - Add support for file dataset=/a/b/c run in [1,2,3] site=T2_CH_CERN, ticket 3982 (requires external urlproxy server, see 3986)
  - Split fakeDatasetSummary into fakeDatasetPattern and fakeDatasetSummary to support look-up of valid datasets for given pattern and any dataset info for givan dataset path; ticket 3990
  - Add draft code to accommodate file dataset=/a/b/c run in [1,2,3] site=X query (still under development)
  - Add url_proxy module which can work with pyurlfecth or Go proxy server
  - Add get_proxy, proxy_getdata and implementation (still experimental) of proxy usage within DBS3 module
  - Re-wrote update_query_record API; update ctime for query records
  - Separte insertion of query and data records
  - Remove analytics calls from abstract service, current analytics implementation require full re-design, it does not make any good so far
  - Add distinguishing message in ticket issue title for no apis/no results errors
  - Add fakeFiles4BlockRun API to cover file block=/a/b/c#123 run in [1,2,3] queries required by CMSSW Integration Builds (IB).
  - Fix file block=/a/b/c#123 query (DBS should contribute to it)
  - Add dataset pattern constratins for all DBS/DBS3 queries
  - Remove listLFNs since listFiles cover the use case to look-up file for a given dataset
  - Add filelumis4dataset API to support file,lumi dataset=/a/b/c queries
  - Add support for run IN [1,2,3] queries, this will be allowed in DBS/DBS3, CondDB, RunRegistry dataservices
  - Upgrade to Prototype.js 1.7
  - Remove lumi API from CondDB mapping; add lumi API to RunRegistry mapping; clean-up RunRegistry code and remove v2 APIs, the v3 is default now
  - Re-factor Vidmantas code: move wild-card errors into separate template; sanitize template parameters; clean-up code
  - Add das_exceptions module, move all Wild-card excepion into this module
  - Imrove web UI links with box_attention for submitting DAS tickets, ticket #3969

- 1.9.X
  - Fix ticket #3967 (preserve DAS records order while removing duplicates)

  - Fix ticket #3966 (strip-off zero in das filters)
  - Add JS function to handle Event (hide DAS keys window) via ESC
  - Resolve double counting issue, ticket #3965
  - Add Show DAS keys description to web UI
  - Wrap combined_site4dataset API call into try/except block and show exception on web UI. This will help to catch transient missing values from combined data-service for site dataset=/a/b/c queries.
  - Add DASKEY EQUAL VALUE VALUE error condition to DAS PLY parser to cover the case when user cut-and-paste some value and it has empty space, e.g. dataset=/a/b/c om
  - Always use upper() for DBS status since it is stored in upper-case in DBS DB
  - Add function to print DAS summary records
  - Add DAS SERVER BUSY message to web server, ticket #3945
  - Read prim_key from mapping DB rather then lookup_keys in das_mongocache module (with fallback to lookup_keys)
  - Fix verbose printout for pycurl_manager module
  - Add support for summary dataset=/a/b/c run=123, ticket #3960
  - Re-factor das_client to be used in other python application; change return type from str to json in get_data API; add das-headers flag to explicitly ask for DAS headers, by default drop DAS headers
  - Re-factor dasmongocache code to support multiple APIs responses for single DAS key
  - Add api=das_core to dasheader when we first register query record
  - Extend DAS aggregator utility to support multiple APIs repsonse for single DAS key
  - Add db_monitor threads to DASMapping/DASMongocache classes
  - Switch from explicit show|hide links to dynamic show/hide which switch via ToggleTag JS function
  - Adjust web UI with Eric's suggestions to show service names in color boxes; remove DAS color map line in result output
  - Revert to base 10 in size_format
  - Add update_filters method to DASQuery class to allow upgrade its filters with spec keys; this is useful on web UI, when end-user specifies a filter and we need to show primary key of the record
  - Wrote check_filters function to test applied filters in a given query and invoke it within nresults method, ticket #3958
  - Collapse lumi list from DBS3, ticket #3954
  - Remove dbs url/instances from DAS configuration and read this information directly from DAS maps; fixed #3955

- 1.8.X
  - Add support of lumi block=/a/b/c#123 and block file=/path/f.root queries both in DBS and DBS3
  - Do not check field keys in a query, e.g. allow to get partial results
  - Fix plain web view when using DAS filters
  - Extend DAS support for file dataset=/a/b/c run between [1,2] queries
  - Keep links around even if data service reports the error
  - Catch error in combined data-service and report them to UI

  - Protect qxml_parser from stream errors
  - Convert regex strings into raw strings
  - Separate curl cache into get/post instances to avoid racing condition for cached curl objects
  - Convert das timestamp into presentation datetime format
  - Queue type can be specified via qtype parameter in web section of DAS configuration file
  - Extend task_manager to support PriorityQueue
  - Revert default to cjson instead of yajl module, since later contains a bug which incorrectly rounds off large numbers; there is also an outstanding issue with potential memory leak
  - Remove dataset summary look-up information for dataset pattern queries to match DBS2 behavior and reduce DAS/DBS latency, see 9254ae2..86138bd
  - Replace range with xrange since later returns generator rather than list
  - Add capability to dump DAS status stack by sending SIGQUIT signal to DAS server, e.g. upon the following call *kill -3 <PID>* DAS server will dump into its logs the current snapshot of all its threads
  - Apply Vidmantas wildcard patch to improve usage of dataset patterns on web UI
  - Fix Phedex checksum parsing
  - Switch to new PyMongo driver, version 2.4
    * change Connection to MongoClient
    * remove safe=True for all insert/update/remove operation on mongo db collection, since it is default with MongoClient
  - DAS CLI changes:
    * Add exit codes
    * Add –retry option which allows user to decide if s/he wants to proceed with request when DAS server is busy; retry follows log^5 function
    * Set init waiting time to 2 sec and max to 20 sec; use cycle for sleep time, e.g. when we reach the max drop to init waiting time and start cycle again. This behavior reduce overall waiting time for end-users
  - Fix issue with DBS3 global instance look-up
  - Switch to HTML5 doctype
  - New schema for DAS maps
    * re-factor code to handle new schema
    * change all maps/cms_maps according to new schema
    * add new documentation for new schame, see mappings.rst
  - Add support to look-up INVALID files in DBS2/DBS3
  - Enable dbs_phedex combined engine
  - Add new thread module to deal with threads in DAS
  - Switch from low-level thread.start_new_thread to new DAS thread module, assign each thread a name
  - Properly handle MongoDB connection errors and print out nice output about their failure (thread name, time stamps, etc.)

- 1.7.X

  - Switch from PRODUCTION to VALID dataset access type in DBS3
  - Adjust das_core and das_mongocache to optionally use dasquery.hashes
    - hashes can be assigned at run-time for pattern queries, e.g. dataset=/*abc*
    - hashes can be used to look-up data once this field is filled up
  - Let DBSDaemon optionally write dataset hashes, this can be used to enhance dataset pattern look-up in DAS cache, see ticket #3932
  - Add hashes data member and property to DASQuery class
  - Work on DBS3 APIs
  - Fix issue with forward/backward calls in a browser which cause existing page to use ajaxCheckPid. I added reload call which enforces browser to load page content with actual data
    - revisit ajaxCheckPid and check_pid functions. Removed ahash, simplify check_pid, use reload at the end of the request/check_pid handshake
  - Add fakeDataset4Site DBS2 API to look-up datasets for a given site, ticket #3084
    - DBS3 will provide new API for that
  - Change DAS configuration to accept web_service.services who lists local DAS service, e.g. dbs_phedex, dbs_lumi
  - Modify dbs_phedex service to initialize via DAS maps
  - Add lumi_service into combined module
  - Introduced services mapping key
  - Adjust combined map file to use services mapping key
  - Switch to pycurl HTTP manager, which shows significant performance boost
  - Work on pycurl_manager to make it complaint with httplib counterpart
- 1.6.X
  - Add new logging flag to enable/disable logging DAS DB requests into logging db (new flag is dasdb.logging and its values either True or False)
  - Change pymongo.objectid to bson.objectid, pymongo.code to bson.code since pymongo structure has been changed (since 2.2.1 pymongo version)
  - Introduce new dataset populator tool which should fetch all DBS datasets and keep them alive in DAS cache (not yet enabled)
  - Move DAS into github.com/dmwm organization
  - Extend das_dateformat to accept full timestamp (isoformat); provide set of unit tests for das_dateformat; fix web UI to accept date in full isoformat (user will need to provide quotes around timestamp, e.g. '20120101 01:01:01'); fixes #3931
  - Set verbose mode only when parserdb option is enabled
- 1.5.X
  - Add SERVICES into global scope to allow cross service usage, e.g. site look-up for DBS dataset records
  - Add site look-up for user based datasets, ticket #3432
  - Revisit onhold daemon and cache requests flaw
    - Start onhold daemon within init call (ensure MongoDB connection)

        - Check DAS cache first for CLI requests regardless if pid presence in a request
        - Put requests on hold only if user exceeds its threshold and server is busy, otherwise pass it through
    - Set DAS times, ticket #3758
    - Convert RR times into DAS date format (isoformat)
    - Fix ticket #3796
- 1.4.X
    - Move code to github
    - Fix bug in testing for numbers, SiteDB now contains unicode entries
    - Add HTTP links into record UI representation
    - Call clean-up method upon request/cache web methods.
    - Add htlKeyDescription, gtKey into RunRegistry, ticket #3735
    - Improve no result message, ticket #3724
    - Update error message with HTTPError thrown by data-provider, ticket #3718
    - Fix das_client to proper handle DAS filters, ticket #3706
    - Change Error to External service error message, see ticket #3697
    - Skip reqmgr API call if user provide dataset pattern, ticket #3691
    - Enable cache threshold reading via SiteDB group authorization
    - Add support for block dataset=/bla run=123 query, ticket #3688
    - Fix tickets #3636, #3639
- 1.3.X
    - Add new method for SiteDB2 which returns api data from DAS cache
    - Add parse_dn function to get user info from user DN
    - Add new threshold function which parse user DN and return threshold (it consults sitedb and look-up user role, if role is DASSuperUser it assigns new threshold)
    - Add suport_hot_threshold config parameter to specify hot threshold for super users
    - Extend check_pid to use argument hash (resolve issue with compeing queries who can use different filters)
    - Do not rely on Referrer settings, ticket #3563
    - Fix tickets #3555, #3556
    - Fix plain view, ticket #3509
    - Fix xml/json/plain requests via direct URL call
    - Clean-up web server and checkargs
    - Add sort filer to web UI
    - Add sort filter, users will be able to use it as following file dataset=/a/b/c | sort file.size, file dataset=/a/b/c | sort file.size- The default order is ascending. To reverse it, user will need to add minus sign at the end of the sort key, e.g. file.size-
    - Re-factor code to support multiple filters. They now part of DASQuery object. All filters are stored as a dict, e.g. {'grep': <filter list>, 'unique': 1, 'sort': 'file.size}

  - Add sitedb links for site/user DAS queries
  - Re-factor code which serves JS/CSS/YUI files; reduce number of client/server round-trips to load those files on a page
  - fix ddict internal loop bug
  - add representation of dict/list values for given key attributes, e.g. user will be able to select block.replica and see list of dicts on web page
- 1.2.X
  - Pass instance parameter into das_duplicates template, ticket #3338
  - Add qhash into data records (simplify their look-up in mongocache manager)
  - Simplify query submission for web interface (removed obsolete code from web server)
  - Fix issue with sum coroutines (handle None values)
  - Avoid unnecessary updates for DAS meta-records
  - Made das core status code more explicit
  - Remove ensure_index from parser.db since it's capped collection
  - Made QLManager being a singleton
  - Add safe=True for all inserts into das.cache/merge collection to avoid late records arrival in busy multi-threaded environment
  - Add trailing slash for condDB URL (to avoid redirection)
  - Show data-service name in error message
  - Show dataset status field
  - Add support to pass array of values into DAS filter, ticket #3350 but so far array needs to consist of single element (still need to fix PLY)
  - Update TFC API rules (fix its regex in phedex mapping)
  - Init site.name with node.name when appropriate
  - Fill admin info in new SiteDB when user look-up the site
  - Switch to new SiteDB
  - Switch to new REST RunRegistry API
  - Remove dbs instance from phedex subscription URL and only allow DBS global link, ticket #3284
  - Fix issue with invalid query while doing sort in tableview (ticket #3281) discard qhash from the tableview presentation layer
  - Implement onhold request queue. This will be used to slow down users who sequentially abuse DAS server. See ticket #3145 for details.
  - Add qhash into DASquery __str__
  - Fix issue with downloading config from gridfs, ticket 3245
  - Fix DBS run in query with wide run range, use gte/lte operators instead
  - Fix issue with recursive calls while retrieve dict keys
  - Eliminate duplicates in plain view, ticket 3222
  - Fix fakeFiles4DatasetRunLumis API call and check its required parameters

  - Fix plain view with filter usage, ticket #3216
  - Add support for dataset group=X site=T3_XX_XXXX or dataset group=X site=a.b.com queries via blockreplicas Phedex API, ticket #3209
  - Fix IP look-up for das_stats, ticket #3208
  - Provide match between various SiteDB2 APIs in order to build combined record
  - Remove ts field and its index from das.cache collection, it is only needed for das.merge
  - Work on integration with new SiteDB, ticket #2514
  - Switch to qhash look-up procedure, ticket #3153
  - Fix DBS summary info, ticket #3146
  - Do not reflect request headers, ticket #3147
  - Fix DBSDaemon to work with https for DBS3
  - Add ability to DAS CLI to show duplicates in records, ticket #3120
  - Parse Phedex checksum and split its value into adler32/checksum, ticket #3119, 3120
  - Remove from dataset look-up for a given file constrain to look-up only VALID datasets, when user provide a file I need to look-up dataset and provide its status, ticket #3123
  - Resolved issue with duplicates of competing, but similar queries at web UI.
  - Changed task manager to accept given pid for tasks.
  - Generated pid at web layer; check status of input query in a cache and find similar one (if found check status of similar request and generate results upon its completion); moved check_pid code from web server into its one template; adjusted ajaxCheckPid call to accept external method parameter (such that I can use different methods, e.g. check_pid and check_similar_pid)
  - Fixed several issues with handling StringIO (delivered by pycurl)

- 1.1.X
  - Extend not equal filter to support patterns, ticket #3078
  - Reduce number of DAS threads by half (the default values for workers was too high)
  - Name all TaskManagers to simplify their debugging
  - Configure number of TaskManager for DASCore/DASAbstractService via das configuration file
  - Fix issue with data look-up from different DBS instances (introduce instance in das part of the record), ticket #3058
  - Switch to generic DASQuery interface. A new class is used as a placeholder for all DAS queries. Code has been refactored to accept new DASQuery interface
  - Revisited analytics code based on Gordon submission: code-refactoring; new tasks (QueryMaitainer, QueryRunner, AnalyticsClenup, etc); code alignment with DAS core reorganization, ticket #1974
  - Fix issue with XML parser when data stream does not come from data-service, e.g. data-service through HTTP error and DAS data layer creates HTTP JSON record
  - Fix bug in db_monitor who should check if DB connection is alive and reset DB cursor, ticket #2986
  - Changes for new analytics (das_singleton, etc.)
  - Add new tool, das_stats.py, which dumps DAS statistics from DAS logdb
  - Add tooltip template and tooltips for block/dataset/replica presence; ticket #2946

  - Move creation of logdb from web server into mongocache (mongodb layer); created new DASLogdb class which will responsible for logdb; add insert/deletion records into logdb; change record in logdb to carry type (e.g. web, cache, merge) and date (in a form of yyyymmdd) for better querying
  - add gen_counter function to count number of records in generator and yield back records themselves
  - add support for != operator in DAS filters and precise match of value in array, via filter=[X] syntax, ticket #2884
  - match nresults with get_from_cache method, i.e. apply similar techniques for different types of DAS queries, w/ filters, aggregators, etc.
  - properly encode/decode DAS queries with value patterns
  - fix issue with system keyword
  - allow usage of combined dbs_phedex service regardless of DBS, now works with both DBS2 and DBS3
  - Fix unique filter usage in das client, add additions to convert timestamp/size into human readable format, ticket #2792
  - Retire DASLogger in favor of new PrintManager
  - code re-factoring to address duplicates issue; ticket #2848
  - add dataset/block/replica presence, according to ticket #2858; made changes to maps
- 1.0.X
  - add support for release file=lfn query, ticket #2837
  - add creation_time/modification_time/created_by/modified_by into DBS maps, ticket #2843
  - fix duplicates when applying filters/aggregators to the query, tickets #2802, #2803
  - fix issue with MongoDB 2.x index lookup (error: cannot index parallel arrays).
  - test DAS with MongoDB 2.0.1
  - remove IP lookup in phedex plugin, ticket #2788
  - require 3 slashes for dataset/block pattern while using fileReplicas API, ticket #2789
  - switch DBS3 URL to official one on cmsweb; add dbs3 map into cms_maps
  - migrate from http to https for all Phedex URLs; ticket 2755
  - switch default format for DAS CLI; ticket 2734
  - add support for 'file dataset=/a/b/c run=1 lumi=80' queries both in DBS2/DBS3, ticket #2602
  - prohibit queries with ambiguos value for certain key, ticket #2657
  - protect filter look-up when DAS cache is filled with error record, ticket #2655
  - fix makepy to accept DBS instance; ticket #2646
  - fix data type conversion in C-extension, ticket #2594
  - fix duplicates shown in using DAS CLI, ticket #2593
  - add Phedex subscription link, fixes #2588
  - initial support for new SiteDB implementation
  - change the behavior of compare_spec to only compare specs with the same key content, otherwise it leads to wrong results when one query followed by another with additional key, e.g. file dataset=abc followed by file dataset=abc site=X. This lead compare_spec to identify later query as subset of former one, but cache has not had site in records, ticket #2497

  - add new data retrieval manager based on pycurl library; partial resolution for ticket #2480
  - fix plain format for das CLI while using aggregators, ticket 2447
  - add dataset name to block queries
  - add DAS timestamp to all records; add link to TC; fixes #2429, #2392
  - re-factor das web server, and put DAS records representation on web UI into separate layer. Create abstract representation class and current CMS representation. See ticket 1975.

Release 0.9.X series

- 0.9.X
  - change RunRegistry URL
  - fix issue with showing DAS error records when data-service is down, see ticket #2230
  - add DBS prod local instances, ticket 2200
  - fix issue with empty record set, see tickets #2174, 2183, 2184
  - upon user request highlight in bold search values; dim off other links; adjust CSS and das_row template, ticket #2080
  - add support for key/cert in DAS map records, fixes #2068
  - move DotDict into stand-alone module, fixes #2047
  - fix block child/parent relationship, tickets 2066, 2067
  - integrate DAS with FileMover, add Download links to FM for file records, ticket #2060
  - add filter/aggragator builder, fixes #978
  - remove several run attributes from DBS2 output, since this information belong to CondDB and is not present in DBS3 output
  - add das_diff utility to check merged records for inconsistencies. This is done during merge step. The keys to compare are configurable via presentation map. So far I enable block/file/run keys and check for inconsistencies in size/nfiles/nevents in them
  - replace ajax XHR recursive calls with pattern matching and onSuccess/onException in ajaxCheckPid/check_pid bundle
  - walk through every exception in a code and use print_exc as a default method to print out exception message. Adjust all exception to PEP 3110 syntax
  - code clean-up
  - replace traceback with custom print_exc function which prints all exceptions in the following format: msg, timestamp, exp_type, exc_msg, file_location
  - remove extra cherrypy logging, clean-up DAS server logs

Release 0.8.X series

- 0.8.X

  - resolve double requests issue, ticket #1881, see discussion on HN https://hypernews.cern.ch/HyperNews/CMS/get/webInterfaces/708.html
  - Adjust RequestManager to store timestamp and handle stale requests
  - Make DBSDaemon be aware of different DBS instances, ticket #1857
  - fix getdata to assign proper timestamp in case of mis-behaved data-services ticket #1841
  - add dbs_daemon configuration into DAS config, which handles DBS parameters for DBSDaemon (useful for testing DBS2/DBS3)
  - add TFC Phedex API
  - add HTTP Expires handling into getdata
  - made a new module utils/url_utils.py to keep url related functions in one place; remove duplicate getdata implementation in combined/dbs_phedex module
  - add dbs_daemon whose task to fetch all DBS dataset; this info is stored into separte collection and can be used for autocompletion mode
  - improve autocompletion
  - work on scalability of DAS web server, ticket #1791

Release 0.7.X series

towards making DAS easy to use for end-users.

- 0.7.X
  - ticket #1727, issue with index/sort while geting records from the cache
  - revisit how to retrieve unique records from DAS cache
  - add DAS query builder into autocomplete
  - extend refex to support free-text based queries
  - add DBS status keyword to allow to select dataset with different statuses in DBS, the default status is VALID, ticket #1608
  - add datatype to select different type of data, e.g. MC, data, calib, etc.
  - if possible get IP address of SE and create appropriate link to ip service
  - calculate run duration from RR output
  - add conddb map into cms_maps
  - add initial support for search without DAS keywords
  - apply unique filter permanently for output results
  - add help cards to front web page to help users get use with DAS syntax
  - work on CondDB APIs
  - fix issue with IE
  - turn off multitask for analytics services
  - add query examples into front-page
  - get file present fraction for site view (users want to know if dataset is completed on a site or not)

  - fix PLY to accept y|n as a value, can be used to check openness of the block
  - add create_indexes into das_db module to allow consistenly create/ensure indexes in DAS code

Release 0.6.X series

DAS web server; add multitasking support.

- 0.6.5
  - handle auto-connection recovery for DBSPhedexService
  - fix site/se hyperlinks
- 0.6.4
  - create new DBSPhedexService to answer the dataset/site quesitions. it uses internal MongoDB to collect info from DBS3/Phedex data-services and map-reduce operation to extract desired info.
- 0.6.3
  - support system parameter in DAS queries, e.g. block block=/a/b/c#123 system=phedex
  - add condition_keys into DAS records, this will assure that look-up conditions will be applied properly. For instance, user1 requested dataset site=abc release=1 and user2 requested dataset site=abc. The results of user1 should not be shown in user2 queries since it is superset of previous query. Therefore each cache look-up is supplemented by condition_keys
  - add suport for the following queries: dataset release=CMSSW_4_2_0 site=cmssrm.fnal.gov dataset release=CMSSW_4_2_0 site=T1_US_FNAL
  - add new combined DAS plugin to allow combined queries across different data services. For instance, user can request to find all datasets at given Tier site for a given release. To accomplish this request I need to query both DBS/Phedex. Provided plugin just do that.
  - add new method/tempalte to get file py snippets
  - re-factor code which provide table view for DAS web UI
  - add new phedex URN to lookup files for a given dataset/site
  - put instance as separate key into mongo query (it's ignored everywhere except DBS)
  - work on web UI (remove view code/yaml), put dbs instances, remember user settings for view/instance on a page
  - add physics group to DBS2 queries
  - add support to look-up of sites for a given dataset/block
  - allow to use pattern in filters, e.g. block.replica.site=*T1*
  - add filters values into short record view
  - add links to Release, Children, Parents, Configs into dataset record info
  - add support to look-up release for a given dataset
  - add support to look-up cofiguration files for given dataset
  - add fakeConfig, fakeRelease4Dataset APIs in DBS2
  - add support for CondDB
  - add hyperlinks to DAS record content (support only name, se, run_number), ticket #1313

  - adjust das configuration to use single server (remove cache_server bits)
  - switch to single server, ticket #1125
    - remove web/das_web.py, web/das_cache.py
  - switch to MongoDB 1.8.0
- 0.6.2
  - das config supports new parameters queue_limit, number_of_workers)
  - add server busy feature (check queue size vs nworkers, reject requests above threashold), ticket #1315
  - show results of agg. functions for key.size in human readable format, e.g. GB
  - simplify DASCacheMgr
  - fix unique filter #1290
  - add missing fakeRun4File API to allow look-up run for a given file, fixes #1285
  - remove 'in' from supported list of operator, users advised to use 'between' operator
  - DBS3 support added, ticket #949
  - fix #1278
  - fix #1032; re-structure the code to create individual per data-srv query records instead of a single one. Now, each request creates 1 das query record plus one query record per data-srv. This allows to assign different expire timestamp for data-srv's and achieve desired scalability for data-service API calls.
  - re-wrote task_manager using threads, due to problems with multiprocessing modules
  - re-wrote cache method for DAS web servers to use new task_manager
  - adjust das_client to use new type of PID returned by task_manager upon request. The PID is a hash of passed args plus time stamp
  - bump to new version to easy distinguish code evolution
- 0.6.1
  - replace gevent with multiprocessing module
  - add task_manager which uses multiprocessing module and provides the same API as gevent
- 0.6.0
  - code refactoring to move from implicit data look-up to explicit one. The 0.5.X series retieved all data from multiple sources based on query constrains, e.g. dataset=/a/b/c query cause to get datasets, files, block which match the constraint. While new code makes precise matching between query and API and retrieve only selected data, in a case above it will retrieve only dataset, but not files. To get files users must explicitly specify it in a query, e.g. file dataset=/a/b/c
  - constrain PLY to reject ambiguos queries with more then one condition, without specifying selection key, e.g. dataset=/a/b/c site=T1 is not allowed anymore and proper exception will be thrown. User must specify what they want to select, dataset, block, site.
  - protect aggregator functions from NULL results
  - new multiprocessing pool class
  - use gevent (if present, see http://www.gevent.org/) to handle data retrieval concurently
  - switch to YAJL JSON parser
  - add error_expire to control how long expire records live in cache, fixes #1240

  - fix monitor plugin to handle connection errors

Release 0.5.X series

performed stress tests and code audit DAS servers.

- 0.5.11
  - change RunRegistry API
  - fix showing result string in web UI when using aggregators
  - bug fix for das_client with sparse records
  - add new das_web_srv, a single DAS web server (not enabled though)
  - fix das_top template to use TRACE rather then savannah
- 0.5.10
  - add DAS cache server time into the web page, fixes #941
  - remove obsolete yuijson code from DAS web server
  - use DASLogger in workers (instead of DummyLogger) when verbosity level is on. This allows to get proper printouts in debug mode.
  - fix bug in compare_specs, where it was not capable to identify that str value can be equal to unicode value (add unittest for that).
  - classified logger messages, move a lot of info into debug
  - change adjust_params in abstract interface to accept API as well
  - adjust DBS2 plugin to use adjust_params for specific APIs, e.g. listPrimaryDatasets, to accept other parameters, fix #934
  - add new DAS keyword, parent, and allow parent look-up for dataset/file via appropriate DBS2 APIs
  - extend usage of records DAS keyword to the following cases
    - look-up all records in DAS cache and apply conditions, e.g. records | grep file.size>1, file.size<10
    - look-up all records in DAS cache regardless of their content (good/bad records), do not apply das.empty_record condition to passed empty spec
  - Fix filter->spec overwrite, ticket #958
  - Add cache_cleaner into cache server, its task is periodically clean-up expired records in das.cache, das.merge, analytics.db
  - Fix bug in expire_timestamp
  - Remove loose query condition which leads to pattern look-up (ticket #960)
  - Fix but in das_ply to handle correctly date
    - add new date regex
    - split t_DATE into t_DATE, t_DATE_STR
  - add support for fake queries in DBS plugin to fake non-existing DBS API via DBS-QL
  - remove details from DSB listFiles
  - add adjust_params to phedex plugin

- adjust parameters in phedex map, blockReplicas can be invoked with passed dataset
- update cms_maps with fake DBS2 APIs
- add DAS_DB_KEYWORDS (records, queries, popular)
- add abstract support to query DAS (popular) queries, a concrete implementation will be added later
- fix #998
- fix SiteDB maps
- fix host parameter in das_cache_client
- remove sys.exit in das_admin to allow combination of multiple options together
- fix compare_specs to address a bug when query with value A is considered as similar to next query with value A*
- fix get_status to wait for completion of DAS core workflow
- fix merge insert problem when records exceed MongoDB BSON limit (4MB), put those records into GridFS
- fix nresults to return correct number of found results when applying a filter, e.g. monitor | grep monitor.node=T3_US_UCLA
- replace listProcessedDatasets with fakeDatasetSummary, since it's better suits dataset queries. DBS3 will provide proper API to look-up dataset out of provided dataset path, release, tier, primary_dataset.
- fix listLFNs to supply file as primary key
- comment out pass_api call to prevent from non-merge situation, must revisit the code
    * fix issue with missing merge step when das record disapper from cache
- bug fix to prevent from null string in number of events
- increase expire time stamp for dashboard, due to problem described in 1032 ticket. I need to revisit code and make das record/service rather then combined one to utilize cache better. Meanwhile align expire timestamp wrt to DBS/Phedex
- add DBS support to look-up file via provided run (so far using fake API)
- use fakseDataset4Run instead of fakeFile4Run, since it's much faster. Users will be able to find dataset for a given run and then find files for a given dataset
- fix issue with JSON'ifying HTTP error dict
- replace DAS error placement from savannah to TRAC
- add new special keyword, instance, to allow query results from local DBS instances. The keyword itself it neutral and can be applied to any system. Add new abstract method url_instance in abstract_service which can be used by sub-systems to add actual logic how to adjust sub-system URL to specific instance needs.
- replace connection_monitor with dascore_monitor to better handle connection/DASCore absense due to loosing connection to MongoDB
- propagate parser error to user, adjust both DAS cache/web servers
- fix queries with date clause, ticket #1112
- add filter view to show filtered data in plain/text, ticket #959
- add first implementation of tabular representation, ticket #979, based on YUI DataSource table with dynamic JSON/AJAX table feeder

  - add jsonstreamer
  - add cache method to web server (part of future merge between cache/web servers)
  - add das_client which talks to web server; on a web server side made usage of multiprocessing module to handle client requests. Each request spawns a new process.
  - visualize record's system by colors on web UI, ticket #977
  - add child/parent look-up for dataset/files
  - work on DAS PLY/web UI to make errors messages more clear, especially adjust to handle DBS-QL queries
  - added dbsql_vs_dasql template which guides how to construct DAS QL expressions for given DBS QL ones
  - fix concurrency problem/query race conditions in DAS core
  - remove fakeListFile4Site from DBS maps since DBS3 does not cover this use case
  - modified das_client to allow other tools use it as API
  - fix DBS/phedex maps to match dashes/underscores in SE patterns
  - add adjust_params into SiteDB to allow to use patterns in a way SiteDB does it (no asterisks)
  - disable expert interface
  - update analytics in DAS core when we found a match
- 0.5.9
  - fix issue with <,> operators and numeric valus in filters
  - add tier into DBS listProcessedDatasets API as input parameter, so user can query as "dataset primary_dataset=ZJetToEE_Pt* tier=*GEN*"
  - DBS2 API provides typos in their output, e.g. primary_datatset, processed_datatset, add those typos into DAS map to make those attributes queriable.
  - Add lumi into DBS map, as well as its presentation UI keys
- 0.5.8
  - Finish work to make presentation layer more interactive, ticket #880
    - create hyperlinks for primary DAS keys
    - round numbers for number of events, etc.
    - present file/block size in GB notations
  - add new "link" key into presentation to indicate that given key should be used for hyperlinks
  - add reverse look-up from presentation key into DAS key
  - add cache for presentation keys in DAS mapping class
  - update DAS chep paper, it is accepted as CMS Note CR-2010/230
  - fix issue with similar queries, e.g. dataset=/a/b/c is the same as dataset dataset=/a/b/c
  - improve presentation layer and add links
    - replace link from boolean to a list of record in presentation YML file

        * the link key in presentation now refers to list of records, where each record is a dict of name/query. The name is shown on a web UI under the Links:, whiel query represents DAS query to get this value, for example {"name":"Files", "query":"file dataset=%s"}
    - fix issue with counting results in a cache
    - make dataset query look-up close to DD view, fixes #821
    - add YAJL (Yet Another JSON Library) as experimental JSON module, see http://lloyd.github.com/yajl/ and its python binding.
    - add keylearning and autocompletion, ticket #50
    - add parse_filter, parse_filters functions to parse input list of filters, they used by core/mongocache to yield/count results when filters are passed DAS-QL. This addresses several Oli use cases when multiple filters will be passed to DAS query, e.g. file dataset=/a/b/c | grep file.size>1, file.size<100
    - add special DAS key records, which can be used to look-up records regarless of condition/filter content, e.g. the DAS query site=T1_CH_CERN only shows site records, while other info can be pulled to DAS. So to look-up all records for given condition user can use records site=T1_CH_CERN
    - remove obsolete code from das_parser.py
- 0.5.7
    - Fix dbport/dbhost vs uri bug for das expert interface
    - Created new self-contained unit test framework to test CMS data-services
        * add new DASTestDataService class which represents DAS test integration web server
        * provide unit test against DAS test data web service
        * add new configuration for DASTestDataService server
        * perform queries against local DAS test data service, all queries can be persistent and adjusted in unittest
        * add fake dbs/phedex/sitedb/ip/zip services into DASTestDataService
    - remove all handlers before initialization of DASLogger
    - add NullHandler
    - add collection parameter to DAS core get_from_cache method
    - add unit test for web.utils
    - add delete_db_collection to mapping/analytics classes
    - remove obsolete templates, e.g. das_admin, mapreduce.
    - sanitize DAS templates, #545
    - Fix issues with showing records while applying DAS filters, #853
    - Move opensearch into das_opensearch.tmpl
    - Fix dbs/presentation maps
    - Add size_format function
    - Updated performance plot
    - make presentation layer more friendly, fixes #848, #879, #880
    - add new configuration parameter status_update, which allow to tune up DAS web server AJAX status update message (in msec)

  - re-factor DAS web server code (eliminate unnecessary AJAX calls; implement new pagination via server calls, rather JS; make form and all view methods to be internal; added check_data method; redesign AJAX status method)
  - Make admin tool be transparent to Ipython
  - Add new functions/unit tests for date conversion, e.g. to_seconds, next_day, prev_day
  - fix date issue with dashboard/runregistry services, fixes #888. Now user will be able to retrieve information for a certain date
- 0.5.6
  - add usable analytics system; this consists of a daemon (analytics_controller) which schedules tasks (which might spawn other tasks), several worker processes which actually perform these tasks and a cherrypy server which provides some information and control of the analytics tasks
  - the initial set of tasks are
    - Test - prints spam and spawns more copies of itself, as might be expected
    - QueryRunner - duplicates DAS Robot, issues a fixed query at regular intervals
    - QueryMaintainer - given a query, looks up expiry times for all associated records and reschedules itself shortly before expiry to force an update
    - ValueHotspot - identifies the most used values for a given key, and spawns QueryMaintainers to keep them in the cache until the next analysis
    - KeyHotspot - identifies the most used query keys, and spawns ValueHotspot instances to keep their most popular values maintained in the cache
  - provides a cli utility, das_analytics_task allowing one-off tasks to be run without starting the analytics server
  - fix apicall records in analytics_db so that for a given set of all parameters except expiry, there is only one record
  - fix genkey function to properly compare dictionaries with different insert histories but identical content
  - alter analyticsdb query records to store an array of call times rather than one record per query, with a configurable history time
  - append "/" to $base to avoid /das?query patterns
  - Updates for analytics server, add JSON methods, add help section to web page
  - Analytics CLI
  - Add ability to learn data-service output keys, fixes #424
  - Add new class DASQuery
  - Add analytics server pid into analytics configuration
  - Prepend python to all shell scripts to avoid permission problem
  - fix dbs blockpath map
  - add new presentation layouts for various services
  - increase ajaxStatus lookup time
  - fix issue with date, in the case when date was specified as a range, e.g. date last 24h, the merge records incorrectly record the date value
- 0.5.5

  - fix map-reduce parsing using DAS PLY
  - introduce das_mapreduces() function which look-up MR functions in das.mapreduce collection
  - fixes for Tier0,DBS3 services
  - fix core when no services is available, it returns an empty result set
  - fix DAS parser cache to properly store MongoDB queries. By default MongoDB does not allow usage of $ sign in dictionary keys, since it is used in MongoQL. To fix the issue we encode the query as dict of key/value/operator and decode it back upon retrieval.
  - fix DAS PLY to support value assignment in filters, e.g. block | grep site=T1
  - Fixes for Dashboard, RunRegistry services
  - Eliminate DAS_PYTHONPATH, automatically detect DAS code location
  - Drop off ez_setup in favor python distutils, re-wrote setup.py to use only distutils
  - add opensearch plugin
  - fix issue with DAS PLY shift/reduce conflict (issue with COMMA/list_for_filter)
  - add to DAS PLY special keys, date and system, to allow queries like run date last 24h, jobsummary date last 24h. Prevent queires like run last 24h since it leads to ambuguous conditions.
  - add support for GridFS; parse2gridfs generator pass docs whose size less then MongoDB limit (4MB) or store doc into GridFS. In later case the doc in DAS workflow is replaced with gridfs pointer (issue #611)
  - add new method to DAS cache server to get data from GridFS for provided file id
  - fix DAS son manipulator to support gridfs_id
  - fix das_config to explicitly use DAS_CONFIG environment
  - fix bug with expire timestamp update from analytics
  - add support for “test” and “clean” action in setup.py; remove das_test in favor standard python setup.py test
  - add weighted producer into queryspammer toolkit; this allows to mimic real time behavior of most popular queries and ability to invoke DAS robots for them (up-coming)
  - fix #52, now both min and max das aggregators return _id of the record
  - return None as db instances when MongoDB is down
  - add avg/median functions to result object; modified result object to hold result and rec counter; add helper das function to associate with each aggreagators, e.g. das_min
  - drop dbhost/dbport in favor of dburi, which can be a list of MongoDB uris (to be used for connection with MongoDB replica sets)
  - replace host/port to URI for MongoDB specs, this will allow to specify replication sets in DAS config
  - use bson.son import SON to be compatible with newer version of pymongo
  - use col.count() vs col.find().count(), since former is O(1) operation wrt O(N) in later case

- 0.5.3 - 0.5.4 series
  - Clean-up %post and do not package docs over there
  - All names in bin are adjusted to one schema: das_<task>.
  - All scripts in bin are changed to use /bin/sh or /bin/bash and use ${1+”$@”} instead of “$@”
  - bin area has been clean-up, e.g. das_doc, dassh is removed, etc.

- Remove runsum_keys in runsum_service.py since it is obsolete code
- Fix issue w/ root.close() for runsum_service.py (parser function)
- Remove session from plotfairy
- Remove encode4admin
- Add urllib.quote(param) for das_services.tmpl and das_tables.tmpl
- fix #446
- das_jsontable.tmpl is removed since it's obsolete and no one is using it.
- Remove das_help.tmpl and /das/help since it is obsolete
- Remove das_admin.py since it is obsolete
- Reviewed decorator in web/tools.py and commented out unused decorators, exposexml, exposeplist. I want to keep them around upon they become relevant for DAS long terms.
- Fix issue with wrap2das methods and made them internal.
- Add checkargs decorator to validate input parameters for das_web
- Change socket_queue_size to 100
- Set engine.autoreload_on=False, request.show_tracebacks=False. Verified that server runs in production mode by default.
- Add parameters validation for das_web/das_expert.
- fix #493, allow relocation of PLY parsertab.py
- fix #494, allow usage of HTTP Expires if data-services provide that
- change eval(x) into eval(x, { "__builtins__": None }, {}) for those cases when fail to use json.load(x). Some data-service are not fully compliant and the issue with them need to be resolved at their end.
- Use singleton class for Connection to reduce number of ESTABLISHED connections seeing on server. For details see http://groups.google.com/group/mongodb-user/browse_thread/thread/67d77a62059568d7# https://svnweb.cern.ch/trac/CMSDMWM/ticket/529
- use isinstance instead of types.typeXXX
- make generic cern_sso_auth.py to authenticate with CERN SSO system
- make das_map to accept external map dir parameter which specify locations of DAS maps
- fix queryspammer to handle generators; add weights
- unify DAS configuration via das_option
- Remove das docs from RPM, will run it stand-alone elsewhere
- Move checkargs into DAS.web.utils; reuse this decorator for all DAS servers to sanitize input arguments; added new unit test for it
- Introduce DAS server codes, they resides in DAS.web.das_codes
- Change DAS server behavior to return HTTPError. The passed message contains DAS server error code.
- fix #525, #542.
- fix issue with counting of empty records, #455
- Handle the case when MongoDB is down. Both DAS servers can handle now outage of MongoDB either at start-up or during their operations. Adjust code to use a single mongodb host/port across all databases, fix #566

  - Remove from all unit test hardcoded value for mongodb host/port, instead use those from DAS configuration file
  - Use calendar.timegm instead of time.mktime to correctly convert timestamp into sec since epoch; protect expire timestamp overwrite if exires timestamp is less then local time
  - Add empty_record=0 into DAS records, to allow consistent look-up
  - Added DAS_PYTHONROOT, DAS_TMPLROOT, DAS_IMAGESROOT, DAS_CSSROOT, DAS_JSROOT to allow DAS code relocation
- 0.5.0 till 0.5.2
  - based on Gordon series of patches the following changes has been implemented
    - new analytics package, which keeps track of all input queries
    - new DAS PLY parser/lexer to confirm DAS QL
    - added new queryspammer tool
  - added spammer into DAS cache client, to perform benchmarking of DAS cache server
  - added a few method to DAS cache server for perfomance measurements of bare CherryPy, CherryPy+MongoDB, CherryPy+MongoDB+DAS
  - remove white/back list in favor of explicit configuration of DAS services via DAS configuration systems (both das.cfg and das_cms.py)
  - added index on das.expire
  - fixed issue with SON manipulator (conversion to str for das_id, cache_id)
  - enable checks for DAS key value patterns
  - added URN's to query record
  - added empty records into DAS merge to prevent cases when no results aggregated for user request
    - empty records are filtered by web interface
    - values for empty records are adjusted to avoid presence of special $ key, e.g. we cannot store to MongoDB records with {'$in': [1,2]}
  - new das_bench tool
  - fixed regex expression for DAS QL pattern, see http://groups.google.com/group/mongodb-user/browse_thread/thread/8507223a70de7d51
  - various speed-up enhancements (missing indexes, empty records, regex bug, etc.)
  - added new RunRegistry CMS data-service
  - updated DAS documentation (proof-reading, DAS QL section, etc.)
  - remove src/python/ply to avoid overlap with system defaul ply and added src/python/parser to keep parsertab.py around

Release 0.4.X series

map is represented data-service URI (URL, input parameters, API, etc.).

- 0.4.13 till 0.4.18
  - adjustment to CMS environment and SLA requirements

  - ability to read both cfg and CMS python configuration files
  - replacement of Admin to Expert interface and new authentication scheme via DN (user certificates) passed by front-end
  - new mongodb admin.dns collection
  - add PID to cherrypy das_server configuration
- 0.4.12
  - added unique filter
  - change value of verbose/debug options in all cli tools to be 0, instead of None, since it's type suppose to be int
  - add new example section to web FAQ
  - re-define logger/logformat in debug mode; the logger is used StreamHandler in this mode, while logformat doesn't use time stamp. This is usefull for DAS CLI mode, when –verbose=1 flag is used.
  - add "word1 word2" pattern to t_WORD for das_lexer, it's going to be used by searching keywords in cmsswconfig service and can be potentially used elsewhere to support multiple keywords per single DAS key
  - fix bug with apicall which should preceed update_cache
  - add simple enc/dec schema for DAS admin authentication
  - add logger configuration into das.cfg
  - separate logger streams into das.log, das_web.log and das_cache.log
  - das_lexer supports floats
  - Add ability for filter to select specific values, e.g. run=123 | grep PD=MinBias right now only equal condition is working, in future may extend into support of other operators
  - add CMSSW release indexer
- 0.4.11
  - adjust abstract data-service and mongocache to use DAS compliant header if it is supplied by DAS compliant API, e.g. Tier0.
  - added cmsswconfigs data-service
  - work on xml_parser to make it recursive. Now it can handle nested children.
  - Fix problem with multiple look-up keys/API, by using api:lookup_keys dict. This had impact on storage of this information within das part of the record. Adjust code to handle it properly
  - added map for Tier0 monitoring data-service
  - fix problem with id references for web interface
  - fix problem with None passed into spec during parsing step
- 0.4.10
  - added new mapping for Phedex APIs
  - work on aggregator to allow merged records to have reference to their parent records in DAS cache, name them as cache_id
  - improve DAS admin interface:
    * show and hide various tasks

        * DAS tasks (query db, clean db, das queries)
        * Add digest authentication to admin interface, based on cherrypy.tools.digest_auth
    - allow to use multiple aggregators at the same time, e.g. site=T1_* | count(site.id), sum(site.id), avg(site.id)
    - enable aggregators in DAS core
    - migrated from CVS to SVN/GIT
    - added AJAX interface for DAS query look-up in admin interface
    - bug fix in core to get status of similar queries
    - validate web pages against XHTML 1.0, using http://validator.w3.org/check
- V0.4.9
    - update admin interface (added query info)
    - integrate DAS lexer in to DAS parser
    - add new class DASLexer, which is based on [PLY]
    - remove >, <, >=, <= operators from a list of supported ones, since they don't make sense when we map input DAS query into underlying APIs. The API usually only support = and range operators. Those operators are supported by MongoDB back-end, but we need more information how to support them via DAS <-> API callback
    - work on DAS parser to improve error catching of unsupported keywords and operators
    - split apart query insertion into DAS cache from record insertion to ensure that every query is inserted. The separation is required since record insertion is a generator which may not run if result set is empty
    - synchronized expire timestamp in DAS cache/merge/analytics db's
- V0.4.8
    - fix pagination
    - display DAS key for all records on the web to avoid overlap w/ records coming out from multiple data-providers (better visibility)
    - protect DASCacheMgr with queue_limit configurable via das.cfg
    - found that multiprocess is unrealiable (crash on MacOSX w/ python version from macports); some processes become zombies. Therefore switch to ThreadPool for DAS cache POST requests
    - added ThreadPool
    - work on DBS2 maps
    - make monitoring_worker function instead of have it inside of DASCacheMgr
    - re-factor DASCacheMgr, now it only contains a queue
    - switch to use <major>.<minor>.<release> notations for DAS version
    - switch to use dot notation in versions, the setup.py/ez_tools.py substitute underscore with dash while making a tar ball
- V04_00_07
    - re-factor DAS configuration system
    - switch to pymongo 1.5.2
    - switch to MongoDB 1.4

  - added admin web interface; it shows db info, DAS config, individual databases and provide ability to look-up records in any collection

- V04_00_06
  - added support for proximity results
  - resolve issue with single das keyword provided in an input query
  - dynamically load of DAS plugins using __import__ instead of eval(klass)
  - first appearance of analytics code
  - fix issue with data object look-up
  - switch to new DAS QL parser
- V04_00_05
  - re-wrote DAS QL parser
  - move to stand-alone web server (remove WebTools dependency)
  - adjust web UI
- V04_00_04
  - choose to use flat-namespace for DAS QL keys in DAS queries
  - added aggregator functions, such as sum/count, etc. as coroutines
  - added "grep" filer for DAS QL
  - extended dotdict class with _set/_get methods
  - re-wrote C-extension for dict_helper
  - added wild_card parameter into maps to handle data-service with specific wild_card characters, e.g. *, %, etc.
  - added ability to handle data-service HTTPErrors. The error records are recorded into both DAS cache and DAS merge collection. They will be propagated to DAS web server where admin view can be created to view them
- V04_00_02, V04_00_03
  - bug fix releases
- V04_00_01
  - minor tweaks to make CMS rpms
  - modifications for init scripts to be able to run in stand-alone mode
- V04_00_00 - incorporate all necessary changes for plug-and-play - modifications for stand-alone mode

Release V03 series

Major change in this release was a separation of DAS cache into independent cache and merge DB collection. The das.cache collection stores *raw* API results, while das.merge keeps only merged records.

- V03_00_04
  - minor changes to documentation structure

- V03_00_03
  - added DAS doc server
  - added sphinx support as primary DAS documentation system
- V03_00_02
  - work on DAS cli tools
- V03_00_01
  - bug fixes
- V03_00_00
  - separate DAS cache into das.cache and das.merge collections

Release V02 series

This release series is based on MongoDB. After a long evaluation of different technologies, we made a choice in favor of MongoDB.

- added support for map/reduce
- switch to pipes syntax in QL for aggregation function support
- switch DAS QL to free keyword based syntax

Release V01 series

Evalutaion series. During this release cycle we played with the following technologies:

- Memcached
- CouchDB
- custom file-based cache

At that time DAS QL was based on DBS-QL syntax. During this release series we added DAS cache/web servers; made CLI interface.

1.8 References

CHAPTER 2

Indices and tables

- *genindex*
- *modindex*
- *search*

Bibliography

[CMS] http://cms.web.cern.ch/cms/

[LHC] http://en.wikipedia.org/wiki/Lhc

[DAS] https://github.com/dmwm/das/

[Python] http://www.python.org/, version 2.6

[CPF] http://www.cherrypy.org/, version 3.1.2

[YUI] http://developer.yahoo.com/yui/, version 2

[Gen] http://www.dabeaz.com/generators/

[YAML] http://en.wikipedia.org/wiki/Yaml, http://pyyaml.org/wiki/PyYAMLDocumentation, version PyYAML-3.08

[PYLINT] http://pypi.python.org/pypi/pylint

[Mongodb] http://www.mongodb.org, version 1.4.1

[MongodbOverview] http://www.paperplanes.de/2010/2/25/notes_on_mongodb.html

[Pymongo] http://api.mongodb.org/python/, version 1.6

[Cheetah] http://www.cheetahtemplate.org/, version 2.4.0

[Cherrypy] http://www.cherrypy.org/, version 3.1.2

[Sphinx] http://sphinx.pocoo.org/, version 0.6.4

[IPython] http://ipython.scipy.org/, version 0.10

[Memcached] http://memcached.org

[Couchdb] http://couchdb.apache.org

[JSON] http://en.wikipedia.org/wiki/JSON

[CJSON] http://pypi.python.org/pypi/python-cjson

[PLY] http://www.dabeaz.com/ply/

[Gevent] http://www.gevent.org/

[YAJL] http://lloyd.github.com/yajl/

[CURL] http://curl.haxx.se/download.html

[PyCurl] http://pycurl.sourceforge.net/

[PyYAML] http://pyyaml.org/

[VENV] https://raw.github.com/pypa/virtualenv/master/virtualenv.py

[GIT] http://git-scm.com/

d