Published in final edited form as:

Proc SPIE Int Soc Opt Eng. 2017 ; 2017: . doi:10.1117/12.2254371.

DAX - The Next Generation: Towards One Million Processes on Commodity Hardware

Stephen M. Damon^a, Brian D. Boyd^b, Andrew J. Plassard^a, Warren Taylor^b, and Bennett A. Landman^{a,b}

^aElectrical Engineering, Vanderbilt University, 2301 Vanderbilt Place, Nashville, TN USA 37235

^bPsychiatry, Vanderbilt University, 2301 Vanderbilt Place, Nashville, TN USA 37235

Abstract

Large scale image processing demands a standardized way of not only storage but also a method for job distribution and scheduling. The eXtensible Neuroimaging Archive Toolkit (XNAT) is one of several platforms that seeks to solve the storage issues. Distributed Automation for XNAT (DAX) is a job control and distribution manager. Recent massive data projects have revealed several bottlenecks for projects with >100,000 assessors (i.e., data processing pipelines in XNAT). In order to address these concerns, we have developed a new API, which exposes a direct connection to the database rather than REST API calls to accomplish the generation of assessors. This method, consistent with XNAT, keeps a full history for auditing purposes. Additionally, we have optimized DAX to keep track of processing status on disk (called DISKQ) rather than on XNAT, which greatly reduces load on XNAT by vastly dropping the number of API calls. Finally, we have integrated DAX into a Docker container with the idea of using it as a Docker controller to launch Docker containers of image processing pipelines. Using our new API, we reduced the time to create 1,000 assessors (a sub-cohort of our case project) from 65040 seconds to 229 seconds (a decrease of over 270 fold). DISKQ, using pyXnat, allows launching of 400 jobs in under 10 seconds which previously took 2,000 seconds. Together these updates position DAX to support projects with hundreds of thousands of scans and to run them in a time-efficient manner.

Keywords

XNAT; High Performance Computing; Pipelines

1. INTRODUCTION

Large scale imaging studies present a new and unique challenge in the field of neuroimaging. Standalone, online, and near-line storage no longer provide an efficient

Correspondence to: Stephen M. Damon.

DAX is an open project available on GitHub (<https://github.com/VUIIS/dax>) as well as the PostgreSQL API (https://github.com/VUIIS/psql_xnat_tools/). DISKQ was developed in a branch inside of dax called diskq, and currently merged into the master. The PostgreSQL interface is in the head of the psql_xnat_tools repo and ready for use. Version 0.4.0 of DAX is currently being used for production and has hardened for the previous 2 years. All spiders alluded to in this work are available on NITRC (<https://www.nitrc.org/projects/masimatlab>). Finally, all documentation, including all spiders is available at (http://dax.readthedocs.io/en/spider_release/).

manner to store acquired and processed data. One tendency is that data often ends up becoming a spaghetti mess and it is hard to find exactly what pipeline was run and where the “final” results are stored. XNAT[1] is a mature neuro-informatics platform which allows for streamlined and efficient archiving for numerous imaging modalities. XNAT is built on top of the XSD language which allows for simple configuration, customization, and deployment solving many common organized data storage issues.

Integral to the success of any image processing study is a way to ensure that pipelines run in not only a timely manner, but that they always run the exact same version (i.e., that they preserve algorithm provenance). Without this information, the fundamental image processing may change and one cannot guarantee that results are consistent. DAX[2] provides a way for users to capture an image processing pipeline into a “spider,” which handles not only the software requirements but also provenance for linked image processing pipelines. DAX’s integration with XNAT has solved many problems at Vanderbilt University not only in terms of storage and archiving, but also processing and a streamlined way to view and report results.

Briefly, DAX communicates with XNAT to determine which sessions have scan(s) that match requirements for a processing pipeline to run. If the requirements exist, an assessor is created. From that point forward (as the queue permits), spiders are launched to complete the work specified by the assessors. Note that spiders contain all necessary version controlled software. Upon completion, results are transferred back to XNAT and the processing and assessor are marked complete.

We have recently generated 37,000 imaging assessors for a cohort of patients with optic nerve disease. This massive amount of data heavily loaded our systems. We decided to make fundamental architectural changes so that our code base could scale to projects only limited by storage space. Herein, we present two approaches for such a solution. First, we demonstrate and evaluate a new API solely written in python which exposes a direct connection to the database via psycopg2 and does not use the REST API wrapped by pyXnat[3]. Secondly, we evaluate a new way of storing provenance and hierarchy of job submission using local disks to store the queue rather than on XNAT via the REST API. Finally, we integrated these improvements into Docker[4] containers for simple distribution.

2. METHODS

In order to write a new API, we installed a clean XNAT 1.6.5 on a build server running Ubuntu 14.04 LTS and PostgreSQL 9.1. The system has 12 cores and 16 GB of RAM. The software installed on this system mimics our current production system. However, all tests were run on a non-loaded system to assess “ideal” conditions. All logging was turned on for PostgreSQL. We then used pyXnat to generate a single assessor and sifted through the logs to find all relevant statements for generating the assessor. These were then methodically transformed into an API which could be used to generate the exact same calls.

All subjects (XNAT’s definition for a person) and sessions (XNAT’s definition for an imaging experiment – single time point) were pre-generated using pyXnat. Subject and

session generation time has not been an issue for pyXnat. Additionally, subjects and sessions are most often created from our users via archiving data coming from a dcm4chee PACS (<http://dcm4chee.org>).

The previously mentioned optic neuritis cohort was used as a case study to assess performance. Specifically, 10, 100, 1000, and 10,000 assessors were timed using the “time” system call in Ubuntu. All tests included time to establish a connection to XNAT as is necessary to use the API. Checks were implemented in both pyXnat and the new API to ensure that the assessor did not already exist, which also mimics our current production system though this adds a bit of overhead to the process.

DISKQ focuses on reducing the number of REST API calls. The previous implementation in DAX (version 0.4.0) first created an assessor and then a secondary process would run to re-initialize the assessor and generate the command line call to run the spider. This was not computationally expensive, but it was time expensive given the overhead of communication. Since each initialize took a fair amount of time (on the order of several minutes) this was removed. The command-line call is rather initialized during the primary process (i.e., when the assessor was created). Thus, overhead was vastly improved for the ability to launch jobs quickly. Figure 3 explains how DAX works.

A Task is considered to be an assessor in DAX terminology. Section A of Figure 3 describes “dax_build”. This process is responsible for creating new assessors in XNAT. The first thing that is considered is if the task will ever run. This decision depends on the spider that is running and checks all the requested inputs which can be NIFTI files, and or complete assessors previous to this one in provenance. If the task doesn’t have any of the requirements and won’t ever have them (e.g., there will never be a T1-weighted scan), then the task is set to a status of NO_DATA and will never be checked again. However, if the Task has all of the inputs and is waiting for a process to finish such as the conversion from DICOM to NIFTI, the status will be set to NEED_INPUTS. Finally, if all dependencies are met, the status will be set to JOB_RUNNING.

Section B depicts the process of dax_launch which is responsible for launching jobs on the grid. All Tasks with a status of JOB_RUNNING are aggregated and the queue limit is checked on the grid to ensure jobs aren’t launched over the group limit. If the queue is not full, it will launch all jobs in queue up to the queue limit (or all of them if the number of Tasks is less than the queue limit). If the queue is full, the jobs will wait until the queue has decreased and then launch.

Section C depicts the process of dax_update_tasks. This process is responsible for checking if jobs are still running on the grid. Each task is checked by grid job ID to see what the job status is. If the job is still running, the Task is ignored until the next time dax_update_tasks runs. If the job has finished, the exit code is checked to see if the job on the grid completed with a zero exit status. If it is, the status is set to READY_TO_COMPLETE. If the exit status is anything other than 0 the status is set to JOB_FAILED.

Section D depicts the process of dax_upload. This process is responsible for uploading the completed job back to the Task in XNAT. After upload, if the status was set to

READY_TO_COMPLETE, the Task status on XNAT is set to COMPLETE. Otherwise, it remains as JOB_FAILED. These four processes comprise the entire DAX infrastructure.

Dockers were generated using a standard Ubuntu 14.04LTS container. A DockerFile was generated and configured to install required development tools (including libxml2-dev, libxslt1-dev, python-dev, wget, zip, unzip, and zlib1g-dev) as well as all updates. Git was also installed since it was not part of the default install in the container. The install was accomplished by including a line to run the standard python install “python setup.py install” since Dockers run as a root user by default. The pre-configured Dockerfile is available on the DAX Github page: (https://github.com/VUIIS/dax_docker/blob/master/dax/dax_0_4_0/Dockerfile)

3. RESULTS

Table 1 displays the results for generating assessors in quantities of 10, 100, 1000, and 10,000 for both PostgreSQL and pyXnat. Also shown is the fractional increase of time saved by bypassing the REST API layer. As can be seen from the results, pyXnat was nonlinear and the results for Postgres were nearly linear. Results were then plotted in Figure 1 to show this relationship and how time is a function of the number of assessors for both pyXnat and Postgres.

Using linear regression, an equation of $y=110x-1600$ was found for assessors generated using pyxnat and $y=0.18X-3.7$ for assessors generated using the new API. This revealed times of 2 days and 2 hours using the new API and 1273 days using pyxnat to generate 1 million assessors.

4. DISCUSSION

The new API substantially out-performed pyXnat by a factor of 7 fold. The benefits were largely realized due to the removal of a layer in between the database (i.e., the REST API was bypassed). Additionally, it was observed the longer that the new API ran, performance decreased slightly. Profiling the new API elucidate this issue revealed that experiment ID's are generated as a sequence, so the database could not simply call select nextval or a UUID called from Java. Rather, all ID's are selected from the experiment and experiment history (the audit trail) tables, joined, and filtered starting with the first looking for the maximum ID. Then, 1 is added to this value and the experiment ID is generated. This process is repeated for every new ID generated (subjects, experiments, and assessors). Naturally, as the table grows, the join takes longer and longer. Ideally the API could keep track of this, but it would be harder in an asynchronous way of processing unless the host mac address is taken into account.

DISKQ produced a sharp decrease in the amount of time needed for launching a job on the grid. The benefits were largely realized by reducing API calls to XNAT. With both the DISKQ and API changes, DAX utilizes the REST API to ensure that database corruption is not an issue. One crucial issue that must be resolved is a way for assessor to be deleted using the new API in the event of a broken pipe. This becomes an issue because commits to the database must be made serially and not in one batch because the audit trail is checked

throughout the process of creating an assessor. XNAT maintains a history of every change that is made via web interface or API. This history is also required if generating assessors through the database. Specifically, the user who modified the workflow event (i.e., making the assessor) is logged as well as the time and the related workflow ID. This cascades down through the image_assessor_data table (the parent table), deriveddata (since this is an assessment of an experiment, and finally proc_genprocdata our custom datatype which stores memory used, the node it ran on, wall-time used, and finally the job id). Together these compose a full audit of the generation of the assessor.

Docker integration was straight forward. This powerful method was great for isolating a sandbox for new users to test out tools without touching their current system. The hope is to migrate all 100+ processing pipelines into their own Docker container for simple use and distribution. These Dockers would then be spun up by DAX to launch image processing pipelines in an always “clean” environment so users have a bit more peace of mind as to what their data ran on. This is integral for longitudinal studies where versions of software are important (such as FreeSurfer[5], MATLAB, MIPAV, SPM[6], FSL[7], and JIST[8–10]).

Acknowledgments

This research was supported by NSF CAREER 1452485, NIH grants 5R21EY024036 and R01EB017230. This research was conducted with the support from Intramural Research Program, National Institute on Aging, NIH. This study was in part using the resources of the Advanced Computing Center for Research and Education (ACCRE) at Vanderbilt University, Nashville, TN. This project was supported in part by ViSE/VICTR VR3029 and the National Center for Research Resources, Grant UL1 RR024975-01, and is now at the National Center for Advancing Translational Sciences, Grant 2 UL1 TR000445-06.

REFERENCES

1. Marcus D, Olsen T, Ramaratnam M, et al. XNAT: a software framework for managing neuroimaging laboratory data.
2. Harrigan RL, Yvernault BC, Boyd BD, et al. Vanderbilt University Institute of Imaging Science Center for Computational Imaging XNAT: A multimodal data archive and processing environment. *NeuroImage*. 2016; 124:1097–1101. [PubMed: 25988229]
3. Schwartz Y, Barbot A, Thyreau B, et al. PyXNAT: XNAT in python. 2013 arXiv preprint arXiv: 1301.6952.
4. Merkel D. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*. 2014; 2014(239):232.
5. Fischl B. FreeSurfer. *Neuroimage*. 2012; 62(2):774–781. [PubMed: 22248573]
6. Ashburner J. SPM: a history. *Neuroimage*. 2012; 62(2):791–800. [PubMed: 22023741]
7. Smith, S., Woolrich, M., Behrens, T. *Fmrib software library*. Oxford, England:
8. Lucas BC, Bogovic JA, Carass A. The Java Image Science Toolkit (JIST) for rapid prototyping and publishing of neuroimaging software. *Neuroinformatics*. 2010; 8(1):5–17. [PubMed: 20077162]
9. Bao S, Damon SM, Landman BA. Performance management of high performance computing for medical image processing in Amazon Web Services. 97890Q-97890Q-8.
10. Li B, Bryan F, Landman BA. Next Generation of the Java Image Science Toolkit (JIST): Visualization and Validation. *Insight J*. 2012; 2012:1–16.

Figure 1.

Workflow of the DAX infrastructure. Yellow indicates data coming down from a server or web frontend. Green indicates a request or data coming in to a server. Blue indicates an XNAT to database transaction. Black indicates data coming down from a single web frontend to a local workstation. Note that users are only able to connect to one of the web frontends. The secondary frontend is for API requests only and gets heavily bogged down. The primary frontend is fast for user only access.

Figure 3.

Lifecycle of a DAX Task. Please see text for an extended explanation of the workflow.

```
1  #
2  # This sets up a base dax docker image
3  #
4  FROM ubuntu:14.04
5  # Install updates, pip and dax
6  RUN \
7  apt-get update && \
8  apt-get -y install python-pip && \
9  apt-get -y install libxml2-dev libxslt1-dev python-dev zlib1g-dev && \
10 apt-get install wget zip unzip && \
11 wget https://github.com/VUIIS/dax/archive/v0.4.0.zip && \
12 unzip v0.4.0.zip && cd dax-0.4.0 && \
13 python setup.py install
14 #Set to bash
15 CMD ["bash"]
```

Figure 4.

Example Dockerfile of how DAX is installed into a clean Ubuntu 14.04 LTS container.

Figure 4.
Plot of log number of assessors as a function of log time.

Table 1

Results with pyXnat and the new API generating assessors

Number of Assessors	Time for pyXnat (s)	Time for new API (s)	Fractional increase
10	60	3	20
100	7020	26	270
1000	65040	229	284
10000	1092101	2715	402