HDF Performance on OpenStack

June 15, 2016

John Readey
The HDF Group
jreadey@hdfgroup.org

Mission

- Investigate performance of using HDF5 in a cloud environment
  - Measure performance using standard library
    - Compression
    - Chunk layouts
    - File aggregation
  - Investigate ways to harness cloud specific capabilities:
    - Elastic Compute – create compute instances on demand
    - Object Storage – utilize object store for persistent storage
  - Comparison to use of other frameworks like Hadoop or Spark
- Determine future work that would enable HDF5 to perform better in the cloud

Evaluation Criteria

- Evaluation Criteria
  - Performance – how fast can a typical science problem be computed
  - Storage – How much storage is needed for the dataset
  - Usability – how easy is it to perform tasks typical of science analytics
  - Scalability -- How effectively can multiple cores be used
  - Cost – Cost metrics (storage+compute) for various solutions

Plan of Investigation

- Select test dataset
  - NCEP3 – (720,1440) gridded data – 7980 files - 130 GB uncompressed
- Choose a science problem
  - Calculate min/max/avg/stdev for a given dataset
- Select compute platform
  - OSDC Griffin – OpenStack, 300 nodes
- Investigate HDF5 performance
  - Phase 1: using one compute node
    - Vary chunk layout/compression filters
  - Phase 2: using using multiple nodes
  - Phase 3: client/server with HDF Server
- Plan for Future work

Hardware

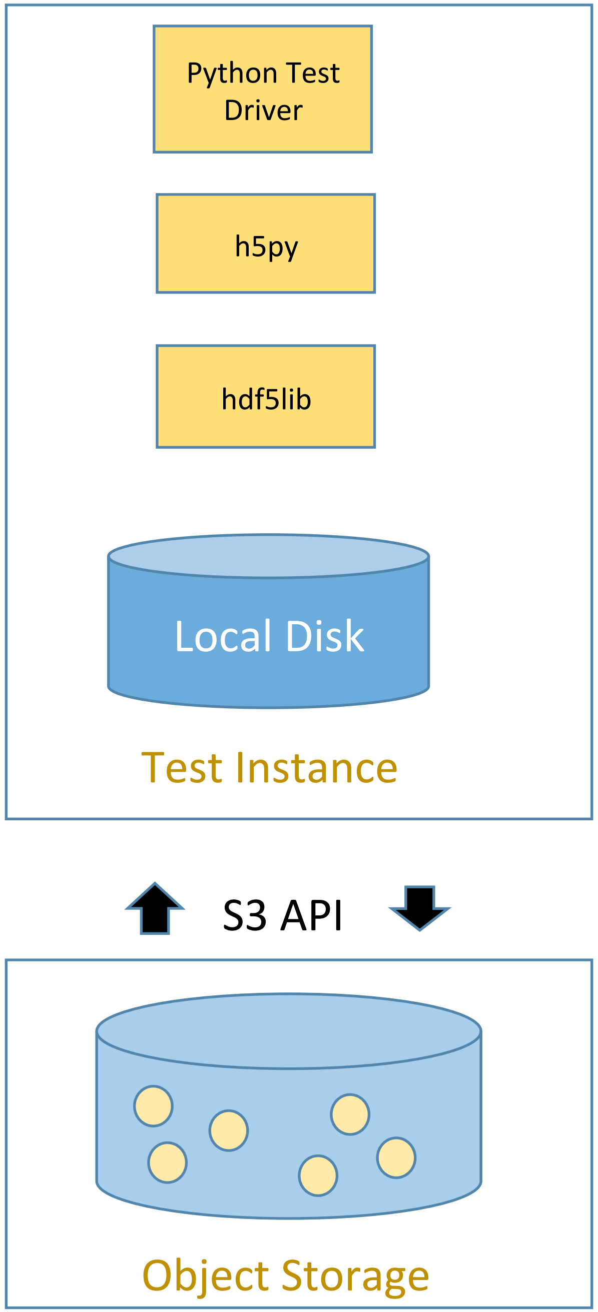
- Using Open Science Data Cloud Griffin cluster
- Xeon systems with 1-16 cores
- 300 compute nodes
- 10Gb Ethernet
- Ephemeral local POSIX file system
- Shared persistent storage (Ceph object store, S3 API)

Software

- HDF5 library v1.8.15
- Compression libraries: MAFISC/GZIP/BLOSC
- Operating system: Ubuntu Linux
- Linux development tools
- HDF5 tools: h5dump, h5repack, etc.
- Python 3
- Python packages: h5py, NumPy, ipyparallel, PyTables
- HDF Server: https://github.com/HDFGroup/h5serv
- H5pyd: https://github.com/HDFGroup/hpd

OpenStack at OSDC in Brief

- Instances can be created either programmatically or via web console
- Compute Instances initialized from snapshot or image file
- Many different instance configurations available
  - RAM 2GB – 100GB
  - Disk 10GB – 2TB
- Onboard disk is ephemeral! – will be lost when the instance is shut down

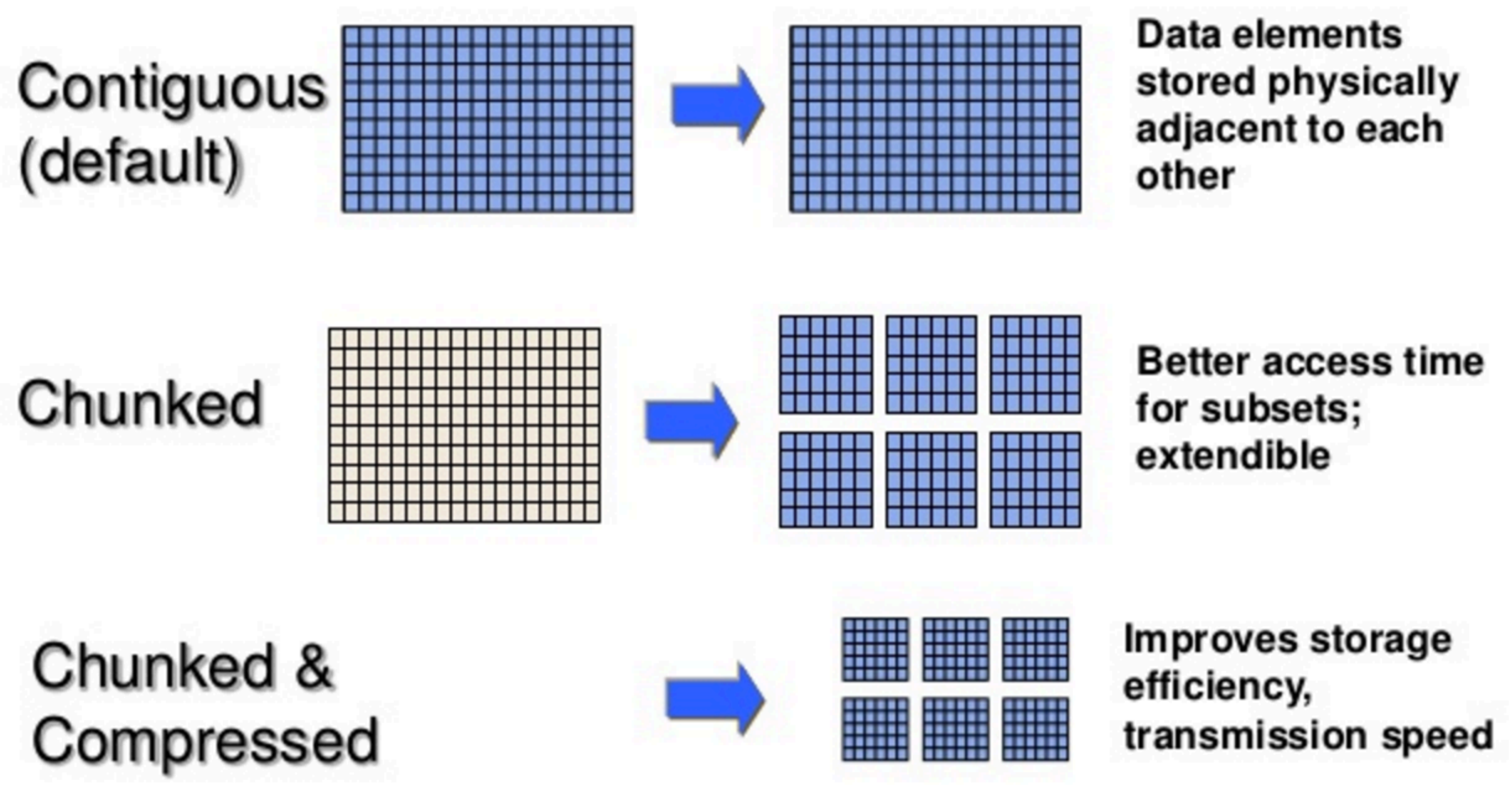


- Instance is created
- Data files copied from object storage to instance
- Test driver is run
- Results and performance measurements stored in Object Store
- Instance is shut down

HDF5 Chunking and compression

- Chunking is one of the storage layouts for HDF5 datasets
- HDF5 dataset's byte stream is broken up in *chunks* and stored at various locations in the file
- Chunks are of equal size in dataset's dataspace but may not be of equal byte size in the file
- HDF5 filtering works on chunks only
- Filters for compression/decompression, scaling, checksum calculation, etc.

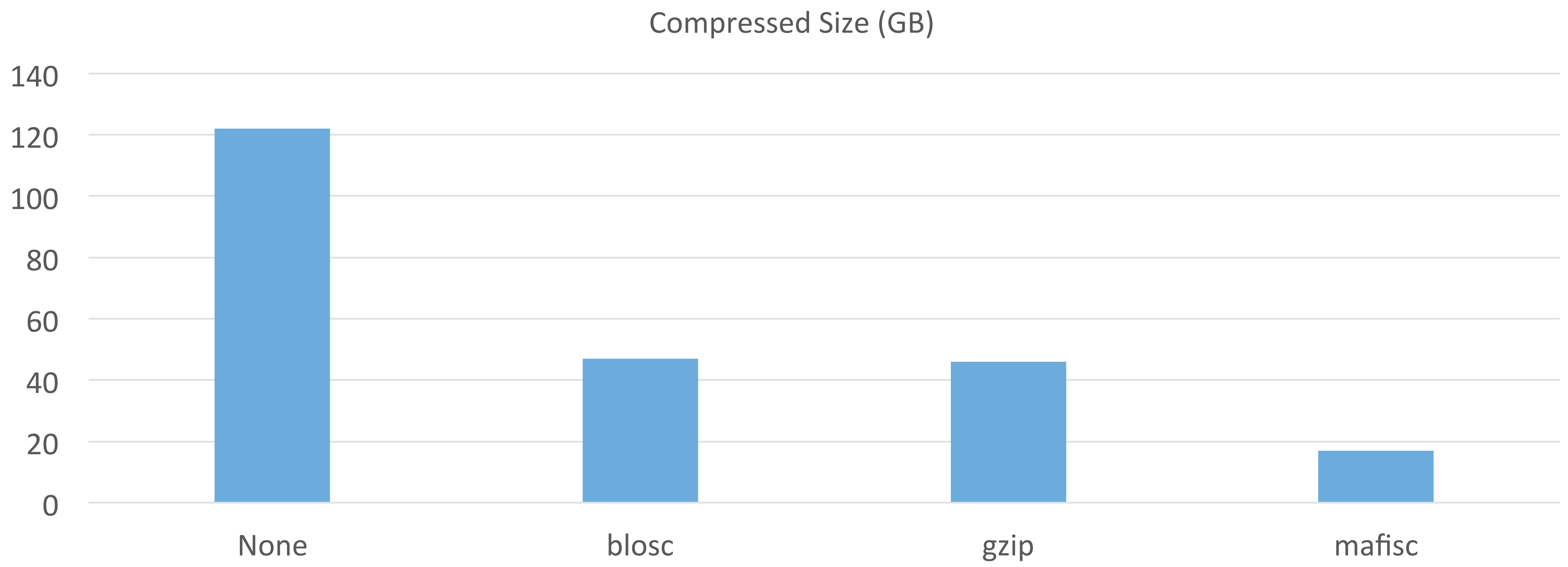
HDF5 Chunking and compression



Determining chunk layouts

- Two different chunking algorithms:
  - Unidata's *optimal* chunking formula for 3D datasets
  - h5py formula
- Three different chunk sizes chosen for the collated NCEP data set:
  - *Synoptic map*: 1×72×144
  - *Data rod*: 7850×1×1
  - *Data cube*: 25×20×20
- Best layout depends on how what the applications access pattern is

Results – Compression Size



MAFISC performed best, but is a lossey compressor.
Blosc and gzip have reduction of ~60%

Results – Runtime

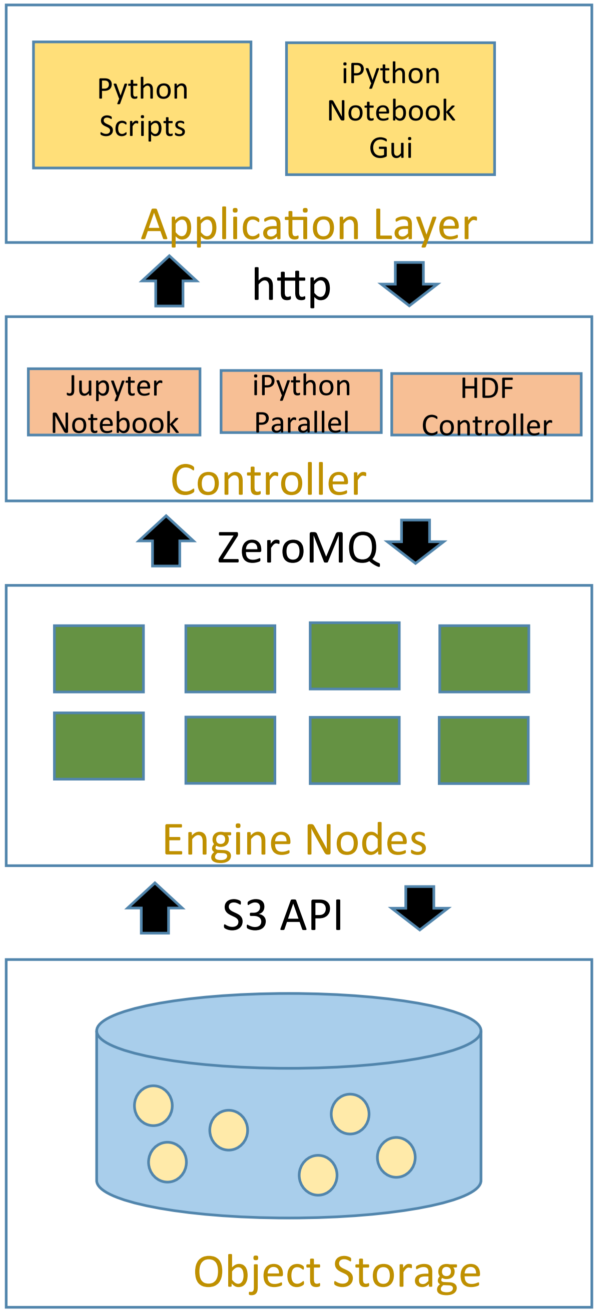
- Load from S3: ~60m
- Runtime:
  - No Compression/no chunking: 11.8m
  - (1x72x144) chunk layout/gzip: 5.2m
  - (25x20x20) chunk layout/gzip: 68.6m
  - (7850x1x1) chunk layout/gzip: 15.4d
- Full results at: https://github.com/HDFGroup/datacontainer/blob/master/results.txt

Phase 2 – Utilizing multiple nodes

- One advantage of cloud environments is on-demand compute, the ability to instantiate and provision compute nodes programmatically
- Frameworks like Hadoop or Spark harness the power of multiple compute nodes to get work done faster
- How easy would it be to utilize multiple instances with OpenStack and the standard HDF5 library?

Cluster Challenge

- Other systems (e.g. Hadoop) support clusters out of the box
- HDF5 does not…
- … So create "on-demand" cluster
  - Wrote code to launch VM's programmatically
  - Connect using ZeroMQ
  - Run with parallel Python
  - Modify test driver to support parallel Python
  - Wrote Python module to distribute data across engines

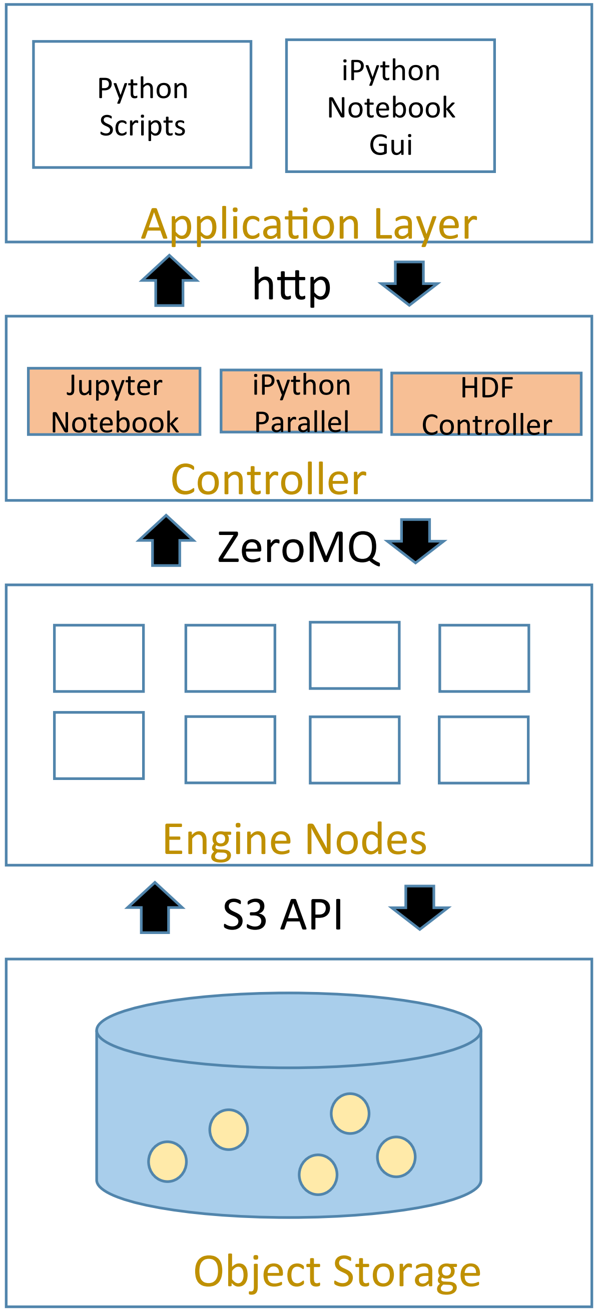


- Users connect to web-based Jupyter Notebook
- Run code via REPL or submit scripts
- Plot results using Matplotlib or other plotting package

- Controller Runs on VM & listens for client requests
- Runs Notebook kernels
- Spins up Engines as needed
- Dispatches work to engines (via iPyParallel/0MQ)

- Engine VMs create on demand by controller
- Each VM reads a partition of data from object store
- Code to be run pushed by controller
- Output returned to controller or saved to local store

- Object Store contains HDF5-based Data Collections (CIMP5/ CORDEX/NCEP3)
- Data collection storage size range from 100GBs to PBs
- Object size in MBs to GBs (can be tuned)
- Meta data maps time/geographic region to objects
- HDF5 compression/chunking reduces space required



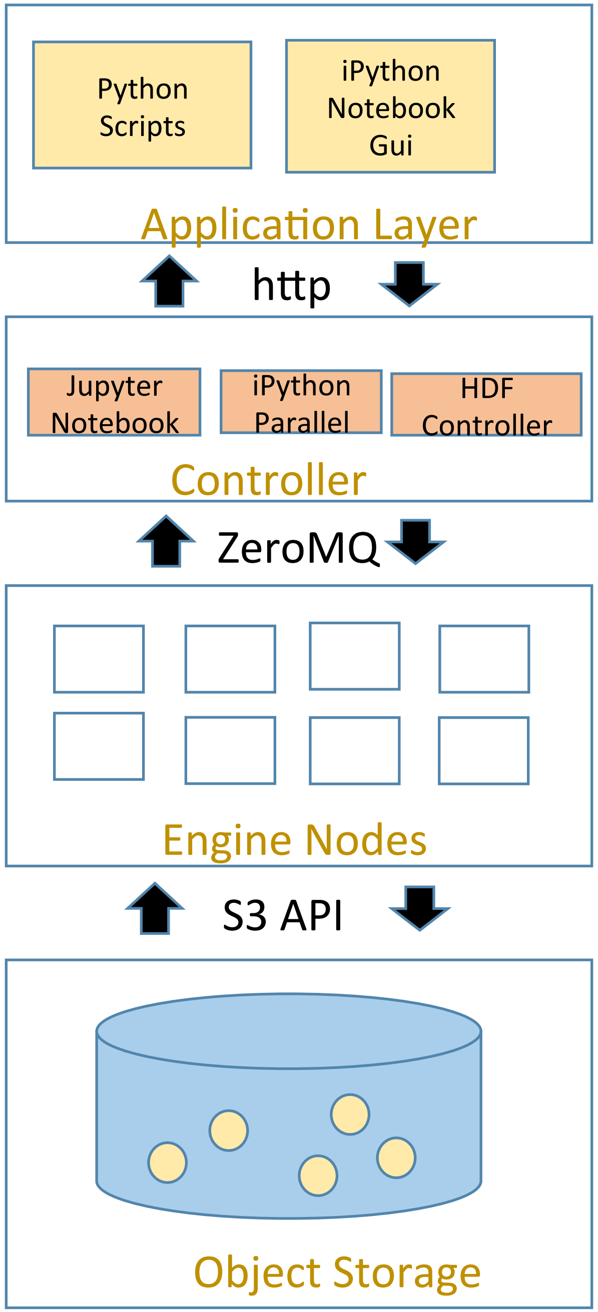
Application Life Cycle 1 – no users connected

- Controller listening for new clients
- Jupyter Hub listening for new notebook sessions

- No engines running

- S3 Data has been imported (Public-readable)

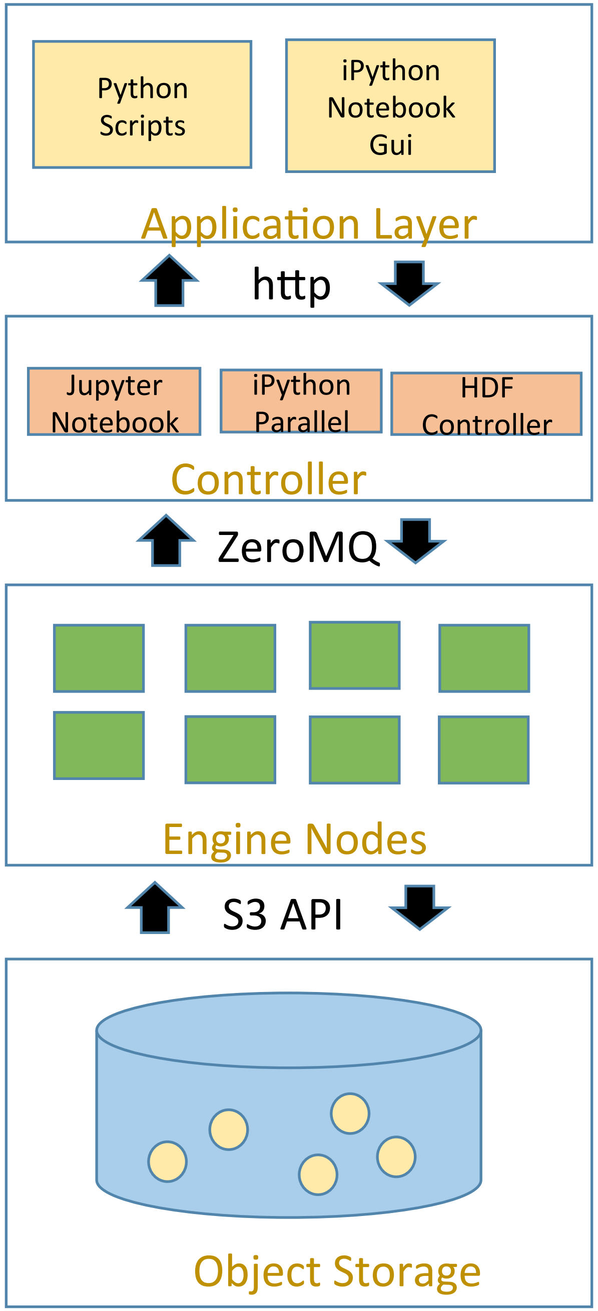
Application Life Cycle 2 – Use launches notebook session



- Jupyter Hub launches session
- Controller gets client request from notebook

- No engines running

- No S3 transfers



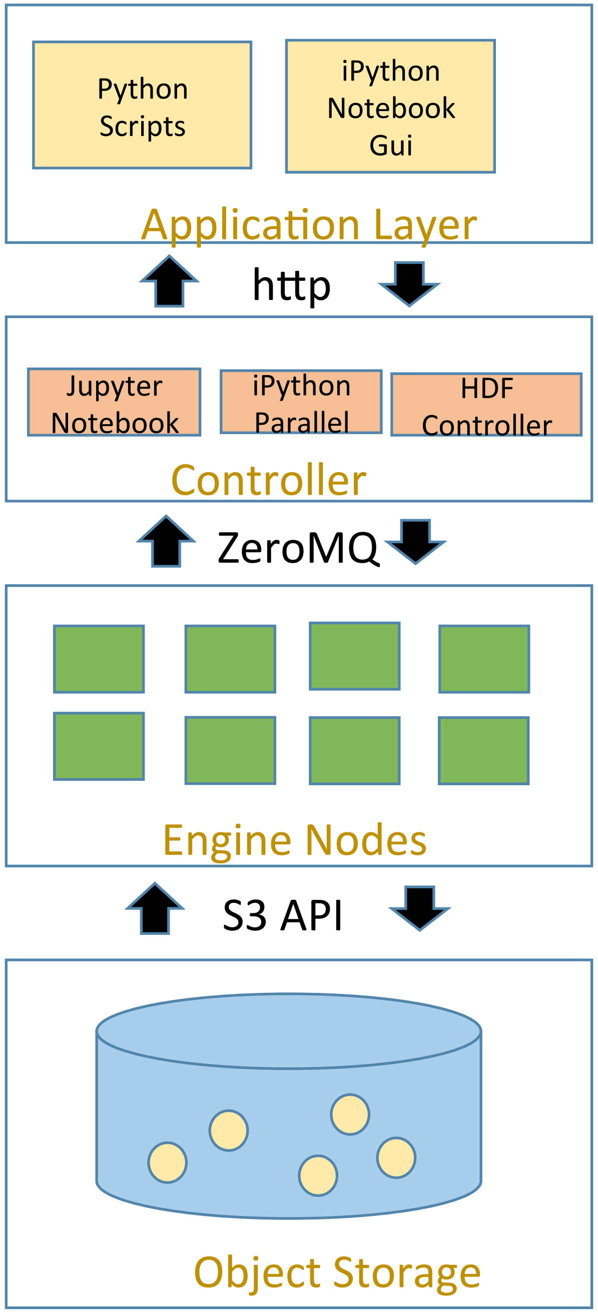
Application Life Cycle 3 – Use loads data collection

e.g. hdfcontroller.load('NCEP3') # user doesn't need to know S3 keys, just data collection label and any subsetting info (time or geo-region)

- Controller gets data request from notebook
- Determines optimum type and # of engines
- Launches Engines
- Tells engines to fetch data objects from S3

- Engines start
- Load data partition
- Signals to controller that data is ready

- Transfer data to engines



Application Life Cycle 4 – Data analytics

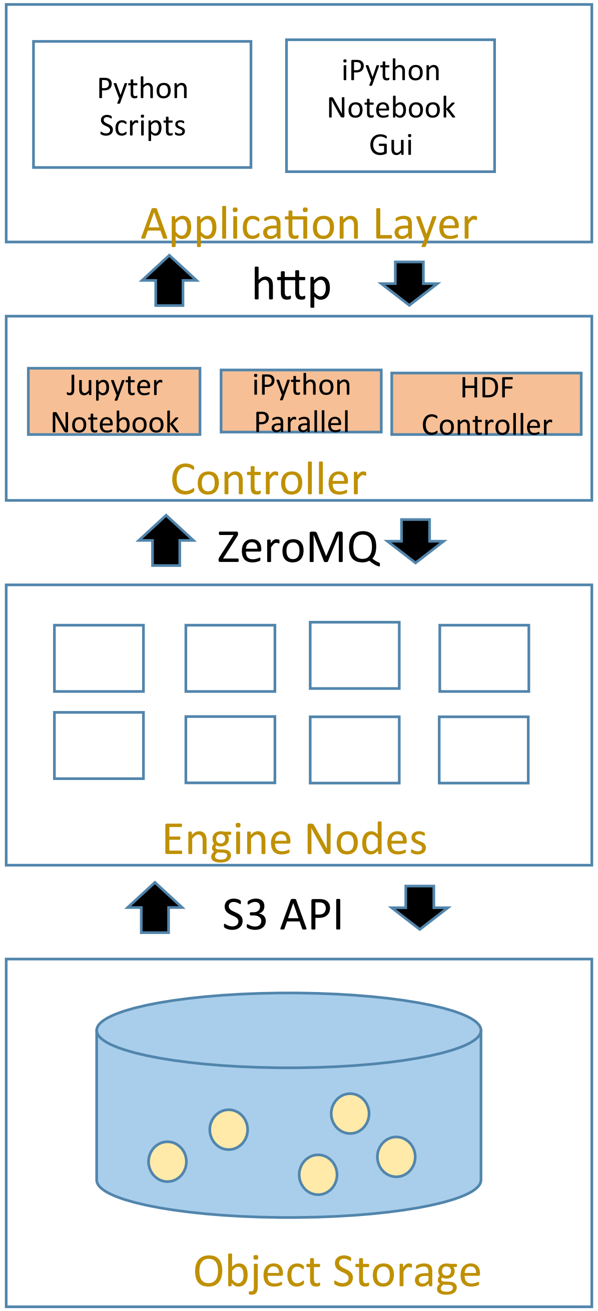
E.g.: get values at geolocation
Repeat cycle of query/analyze/plot as desired

- Controller gets user request from Client
- Dispatches across all engines
- Waits for responses
- Returns aggregated result to client

- Engines process requests
- Data is local to VM (SSD or RAM)

- No activity

Application Life Cycle 5 – no user ends session

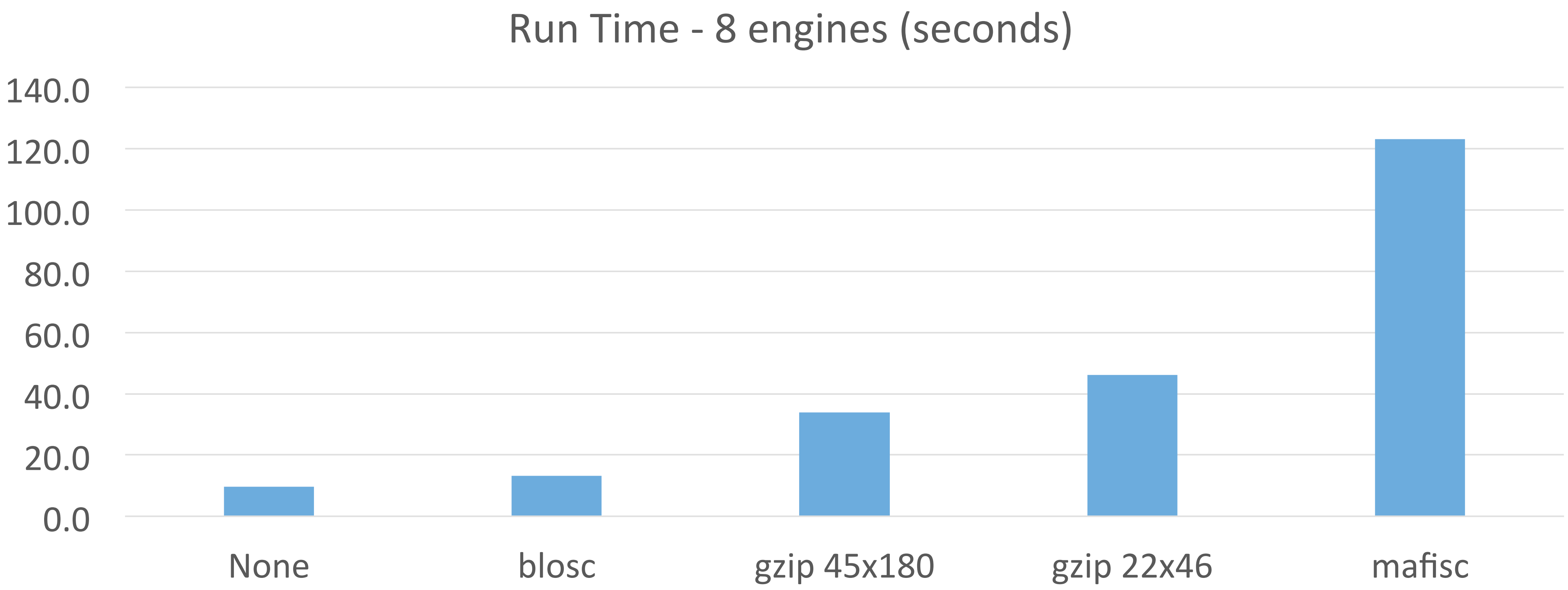


- Controller terminates Engines
- Continues listening for new notebook sessions

- Engines shutdown
- Any data stored locally is lost!

- No S3 activity

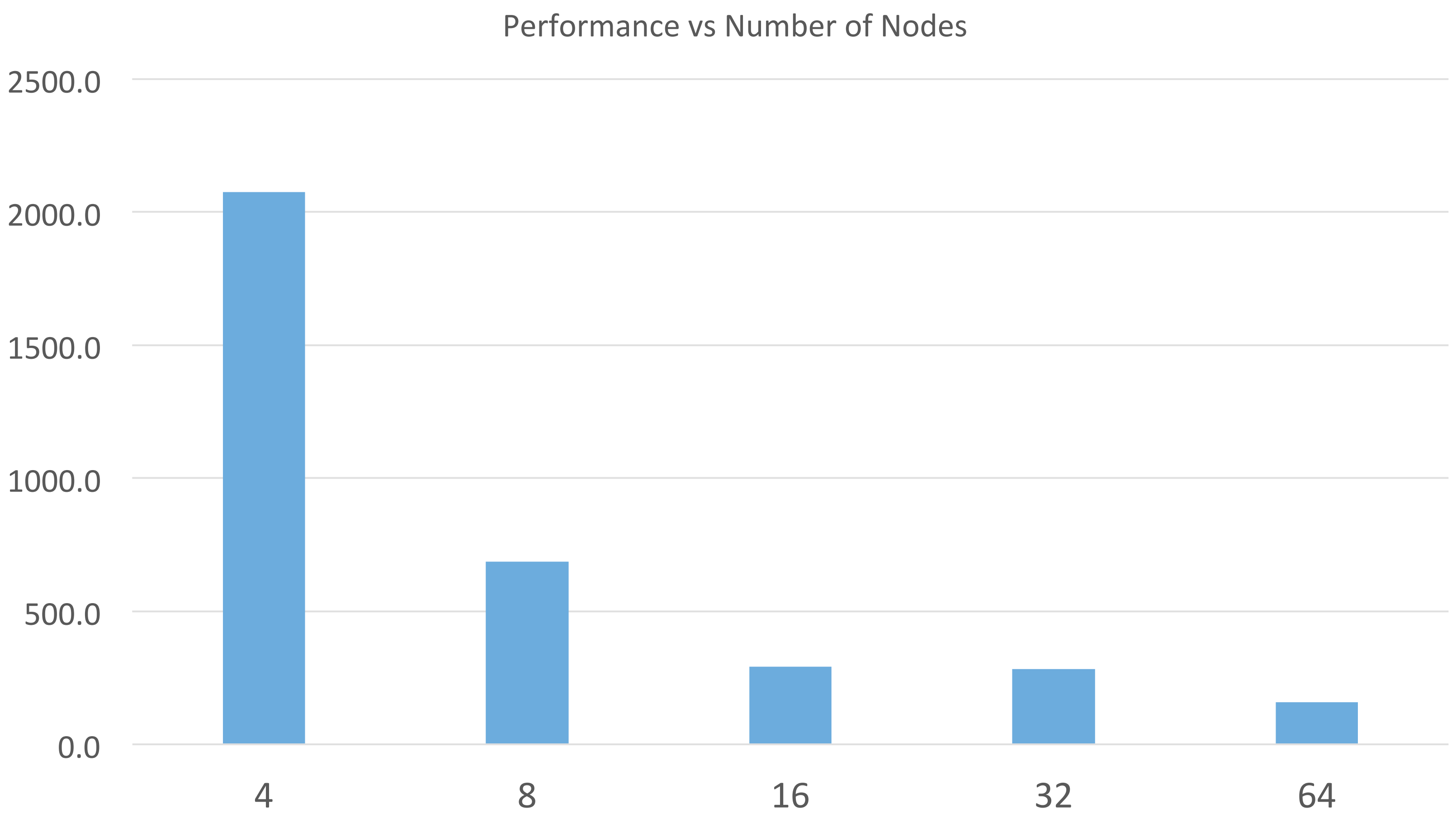
Results – Performance w/ 8 node



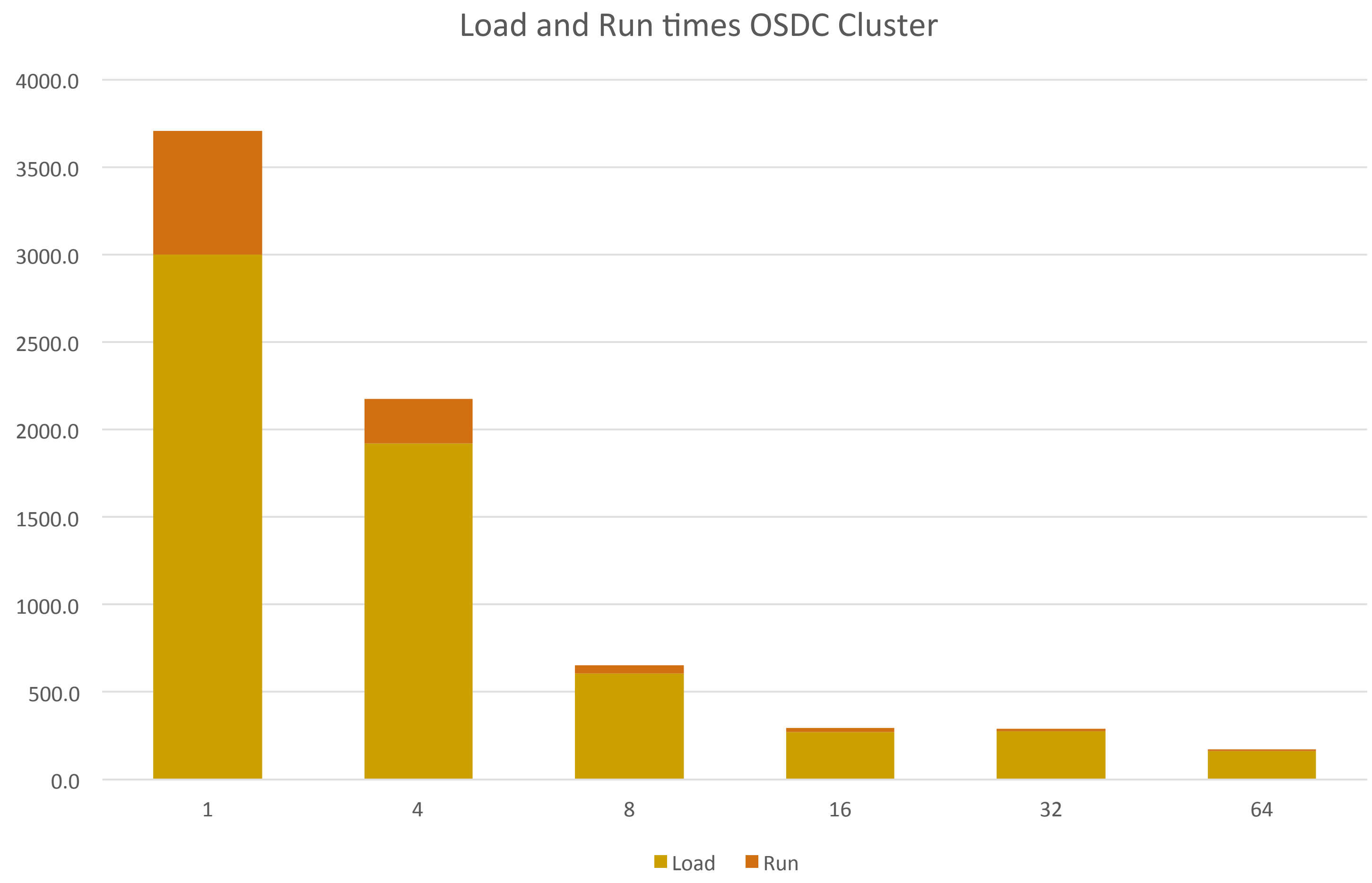
BLOSC has best performance
for compressed format

Todo: chunked but not
compressed dataset

Runtime – by number of nodes – no compression



Percent of time spent loading data goes up as number of nodes increases

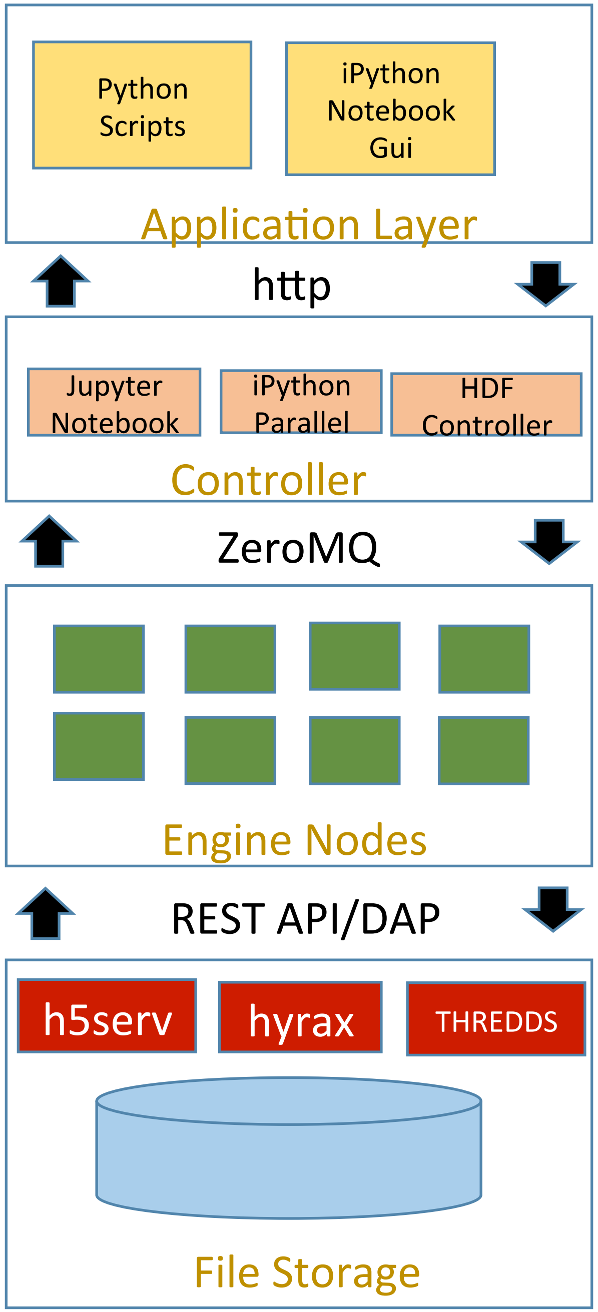


Conclusions - Phase II

- HDF5 with simple cluster solution (ZeroMQ/IPyParallel) provided:
    - Excellent performance – super linear with number of nodes
    - Did not require expansion or conversion of data (as with Hadoop, etc)
    - Enables scientist to use standard tools/apis for analysis
- Existing cluster solution didn't work well with large files (>10GB)
- Cluster launch time and data loading can dominate actual compute time

Methodology – Phase III

- Aggregate NCEP data files to 7850x720x1440 data cube
  - One file, ~100GB
- Setup large VM with file and server (h5serv, hyrax, or THREDDS)
- Parallel nodes access data via requests to server
- Adapt test script to use server interface
- Measure performance with different:
  - Servers
  - Chunk layout
  - Number of nodes



- Users connect to web-based Jupyter Notebook
- Run code via REPL or submit scripts
- Plot results using Matplotlib or other plotting package

- Controller Runs on VM & listens for client requests
- Runs Notebook kernels
- Spins up Engines as needed
- Dispatches work to engines (via iPyParallel/0MQ)

- Engine VMs create on demand by controller
- Each VM submits request to data server
- Code to be run pushed by controller
- Output returned to controller or saved to local store

- File Storage on Server node contains aggregated data
- Data collection storage size ~100GBs
- Meta data maps time/geographic region to objects
- HDF5 compression/chunking reduces space required

Results – Server Access

- Test runs with one node (compute summaries over time slices)

Chunk/Compression	Local	Hyrax	THREDDS	h5serv
None	148.6	3297.1	961.2	885.8
1x72x44/gzip9	317.6	8575.8	1264.1	?
25x20x20/gzip9	4131.0	13946.5	6936.8	?

Conclusions Phase III

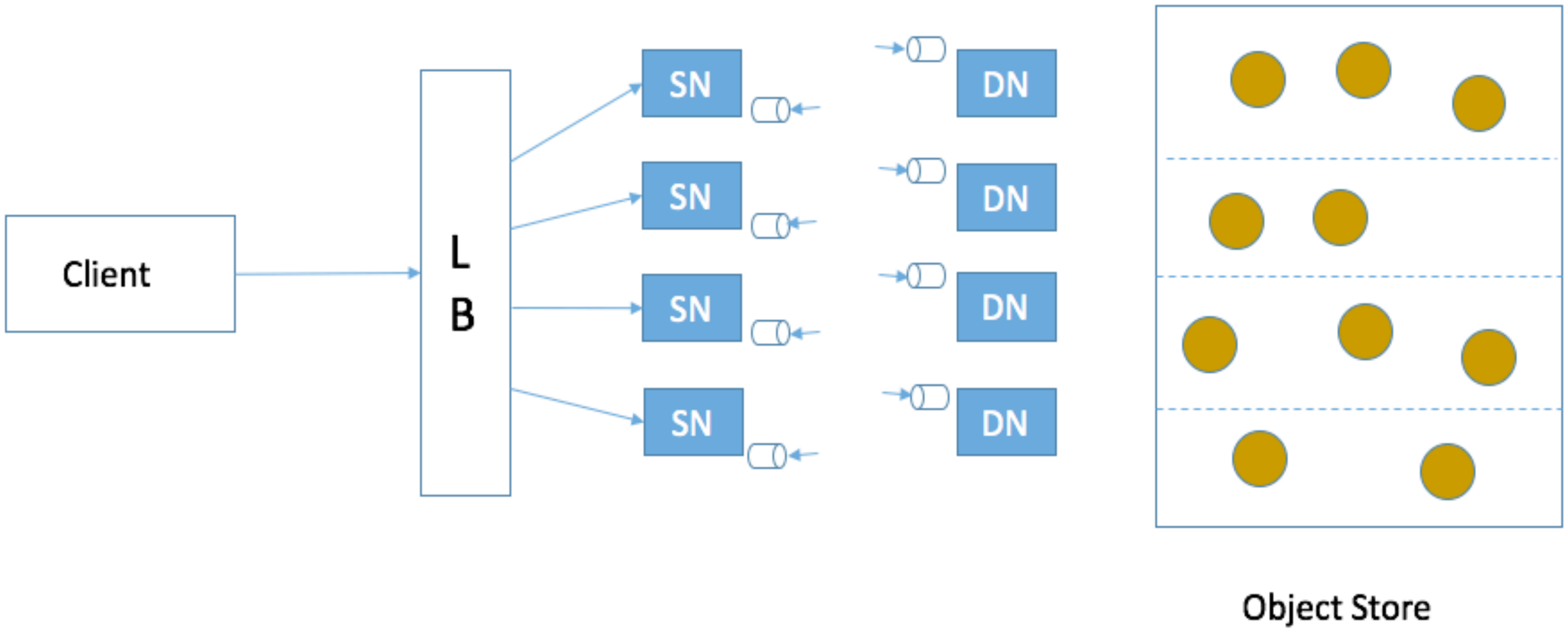
- Remote data access entails a performance penalty
- Allocation of a large instance running continuously required
  - Data on server will be lost if instance goes down
- Aggregate performance levels out with large number of clients
  - Server processes/network io become bottleneck

Future Directions – HDF Scalable Data Service

- Scalable Data Service for HDF5
  - Designed for public or private clouds
  - Uses Object Storage for persistent data
  - ”share-nothing” architecture
  - Support any number of clients
  - Cost-effective
  - Efficient access to HDF5 objects without opening file
  - Client SDK’s for C/Fortran/Python enable existing applications to be used with the service
  - REST API compatible with current HDF Server (reference implementation)

HSDS Architecture

- Service Nodes (SN) handle client requests
- Data Nodes (DN) partition object store
- Both SN and DN clusters can scale based on demand
- HDF Objects (links, datasets, chunks, etc.) stored as objects



Separation of Storage and Compute Costs

- Storage
  - AWS S3 can support any size storage at affordable costs (~$0.03/GB/month)
  - AWS has built in redundancy, so no need for backups, etc.
- Compute
  - If no active users, there is minimal compute costs (~$50/month)
  - Service nodes can scale up in response to load (costs proportional to usage)

Open Questions

- S3 storage
  - Optimal object store key mapping/object sizes
  - Compression/chunking to minimize cost/increase performance
- Cost profile (for AWS)
  - Steady state costs – S3 storage/controller VM
  - VM instance hours * number of engines
  - S3 requests?
- Best engines characteristics
  - Instance type - Need enough local storage. SSD is better than rotating
  - vCPUs? One thread per VM?
  - Optimal # of engines for a given data collection
- Security
  - ZeroMQ doesn't have any!
  - Run in VPC per user?
- How would AWS implementation perform compared to OpenStack?
- Compare using Docker Containers rather than VMs as engine (faster spin up time)
- Validation of transformed results