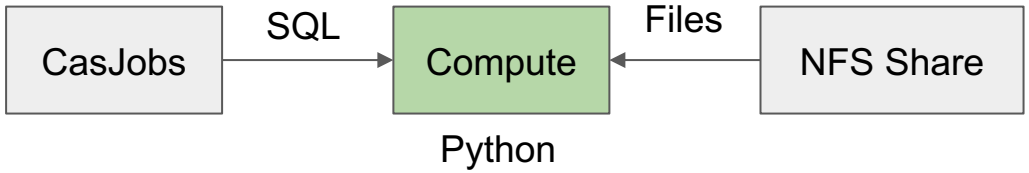
Use of the Spark Distributed Computing Engine to Enhance SciServer Capabilities

SciServer on Big Data

- Data-proximate computing
- Interface duality
  - SQL for search and retrieval
  - Python (et al) codes for analysis on results
- Typical workflow
  - Search for interesting objects based on summary data/statistics using SQL
  - Retrieve table, perhaps up to millions of results
  - Retrieve related raw data (e.g. image/spectrum) for some objects
  - Further analysis in a contained compute environment



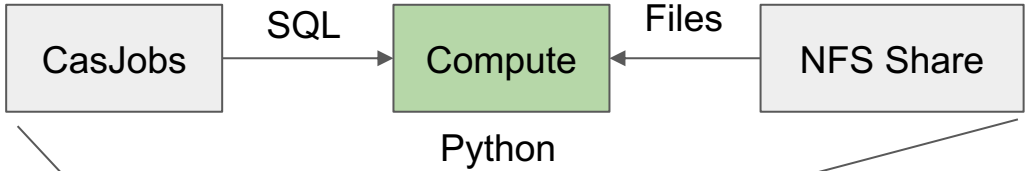
Limitations

- Mode works really well for:
  - “Needle in the haystack” problems
  - Simple full-sky aggregations
  - Summary statistics
- Less well for:
  - Some machine learning applications
    - Where we almost want to download the whole catalog
  - Queries based custom “derivative” values
    - Typically database stores derived values, e.g. age/abundance not spectrum
  - When the “needle” gets very large
    - We might need to work in batches, things get slow
  - Reprocessing/ETL

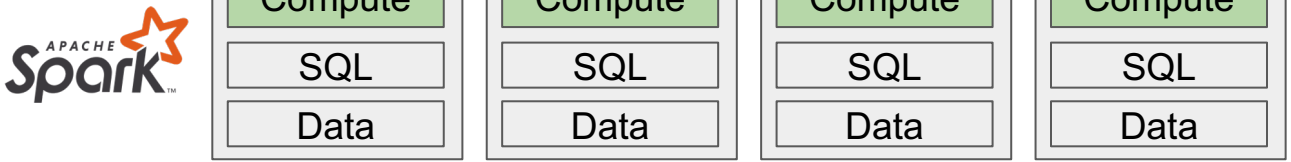
Spark

- Spark collapses the SQL/code duality
- Parallelizes it, and places it on top of the data
- All with simple and resilient interfaces (and multi-language)

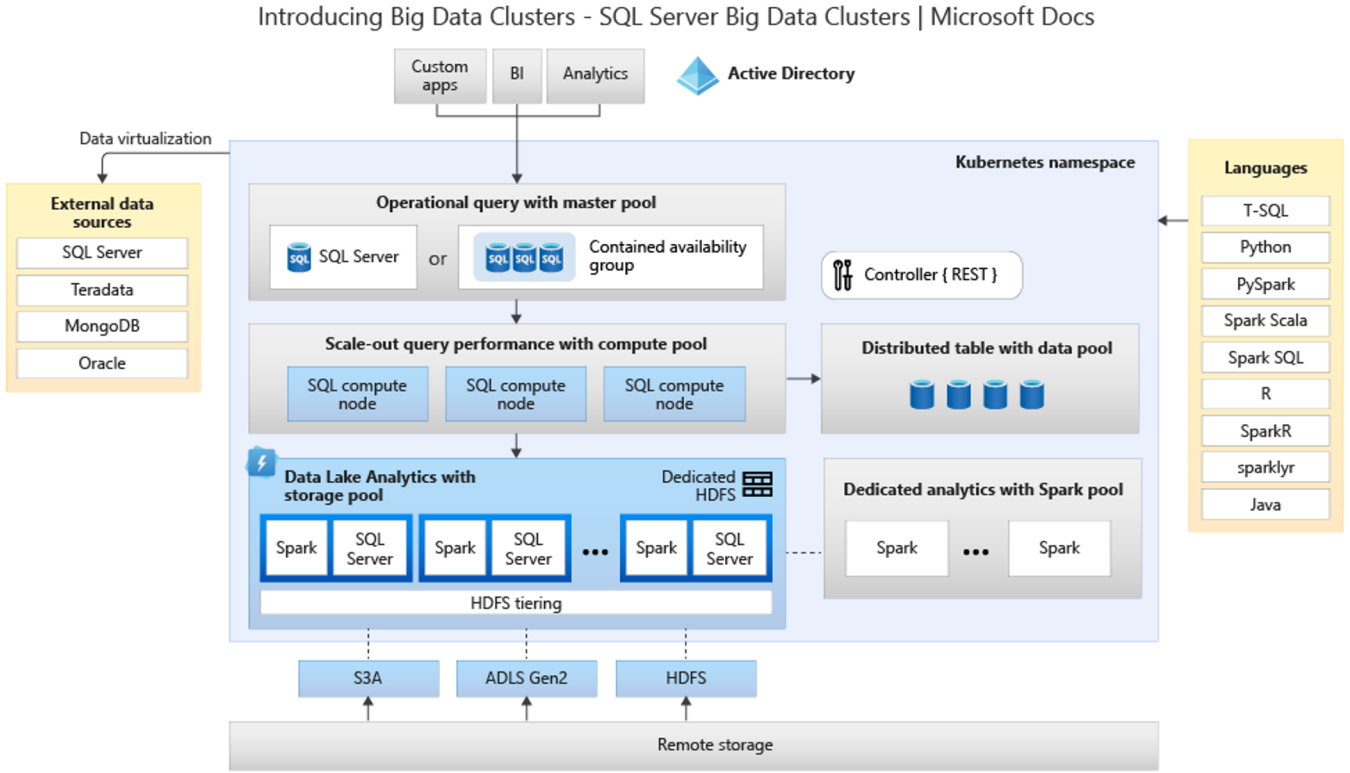
This:



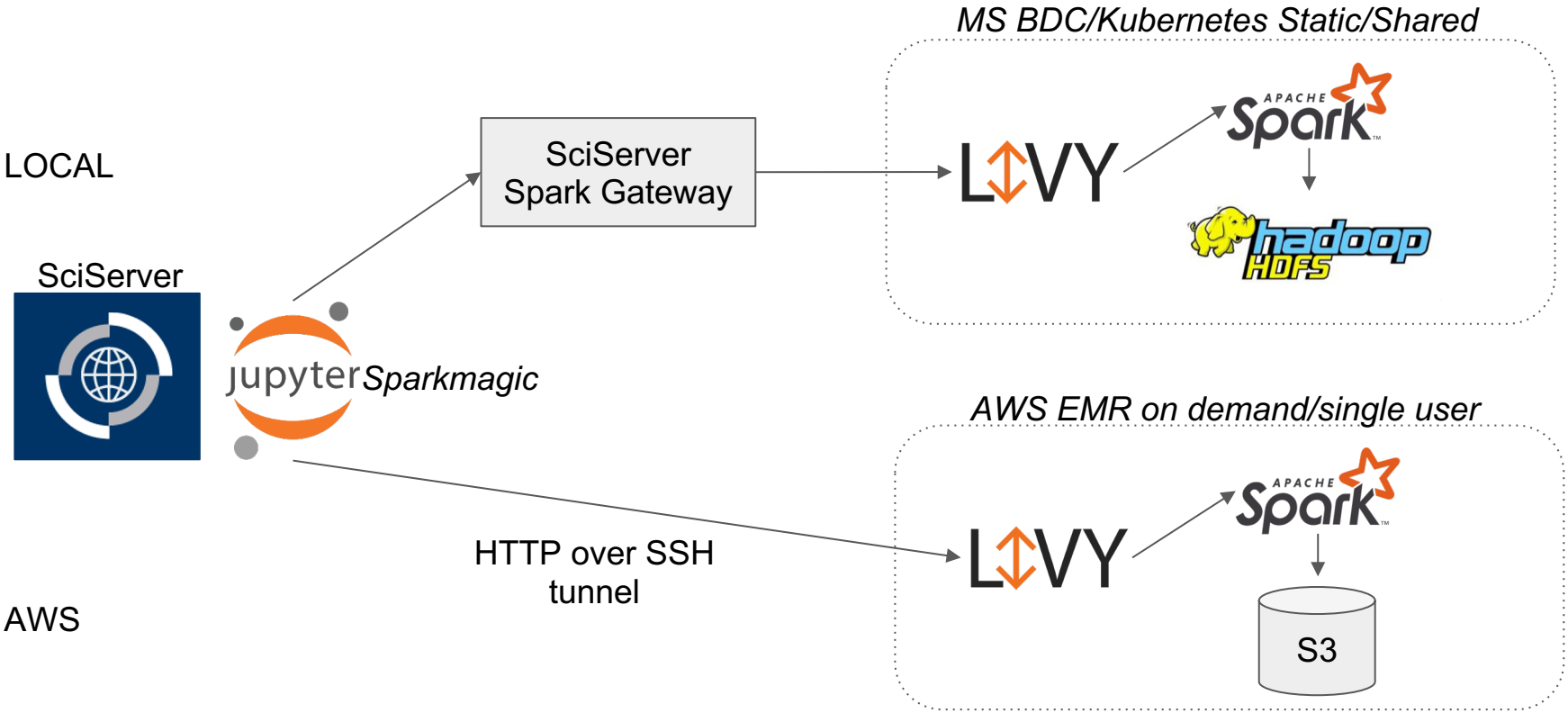
Becomes:



Microsoft Big Data Clusters (BDC)



SciServer + Spark



Case Study: SDSS MaNGA

- IFU data, 2-d spatial coordinate + spectral cubes
- Make available a variety of raw/processed products
- A PostgreSQL implementation can handle array type data, but Spark may simplify queries and improve wall-clock time

Case Study: SDSS MaNGA

- Data ingest:
  - Using Spark, read in FITS files from SAS
  - Conversion function: Python + Astropy -> Spark DataFrame type
  - Write parquet files to HDFS
  - This system is suitable for large scale data (re-)processing, easy to distribute code over data and scale out, no requirement on specific file formats (though some caveats for “non-native” formats)

Case Study: SDSS MaNGA

- Benchmark Queries:
  - 3 factors: Simplicity, Wall-time, Scalability
- Traditional DB:
  - Design can be complex, more targeted, tradeoff for use-cases
- Spark:
  - More general - store everything, achieve performance via parallelism and clever storage formats
  - Trade-off is fast queries suffer scheduling overhead, lack of index, etc.

Ex Test Query: Select all galaxies with an H-alpha flux value > 5 in more than 20% of their good spaxels

Postres

```
SELECT anon_1.mangadatadb_cube_mangaid, anon_1.mangadatadb_cube_plate, concat(anon_1.mangadatadb_cube_plate, '-', anon_1.mangadatadb_ifudesign_name) AS plateifu, anon_1.mangadatadb_ifudesign_name FROM (SELECT mangadatadb.cube.mangaid AS mangadatadb_cube_mangaid, mangadatadb.cube.plate AS mangadatadb_cube_plate, concat(mangadatadb.cube.plate, '-', mangadatadb.ifudesign.name) AS plateifu, mangadatadb.ifudesign.name AS mangadatadb_ifudesign_name, mangadapdb.cleanspaxelprop7.emline_gflux_ha_6564 AS mangadapdb_cleanspaxelprop7_emline_gflux_ha_6564, mangadapdb.cleanspaxelprop7.x AS mangadapdb_cleanspaxelprop7_x, mangadapdb.cleanspaxelprop7.y AS mangadapdb_cleanspaxelprop7_y FROM mangadatadb.cube JOIN mangadatadb.ifudesign ON mangadatadb.ifudesign.pk = mangadatadb.cube.ifudesign_pk JOIN mangadapdb.file ON mangadatadb.cube.pk = mangadapdb.file.cube_pk JOIN mangadapdb.cleanspaxelprop7 ON mangadapdb.file.pk = mangadapdb.cleanspaxelprop7.file_pk JOIN mangadatadb.pipeline_info AS drpalias ON drpalias.pk = mangadatadb.cube.pipeline_info_pk JOIN mangadatadb.pipeline_info AS dapalias ON dapalias.pk = mangadapdb.file.pipeline_info_pk JOIN (SELECT mangadapdb.cleanspaxelprop7.file_pk AS binfile, count(mangadapdb.cleanspaxelprop7.pk) AS goodcount FROM mangadapdb.cleanspaxelprop7 WHERE mangadapdb.cleanspaxelprop7.binid_binned_spectra != -1 AND mangadapdb.cleanspaxelprop7.binid_stellar_continua != -1 AND mangadapdb.cleanspaxelprop7.binid_spectral_indices != -1 AND mangadapdb.cleanspaxelprop7.binid_em_line_moments != -1 AND mangadapdb.cleanspaxelprop7.binid_em_line_models != -1 GROUP BY mangadapdb.cleanspaxelprop7.file_pk) AS bingood ON bingood.binfile = mangadapdb.cleanspaxelprop7.file_pk JOIN (SELECT mangadapdb.cleanspaxelprop7.file_pk AS valfile, count(mangadapdb.cleanspaxelprop7.pk) AS valcount FROM mangadapdb.cleanspaxelprop7 WHERE mangadapdb.cleanspaxelprop7.emline_gflux_ha_6564 > 5 GROUP BY mangadapdb.cleanspaxelprop7.file_pk) AS goodhacount ON goodhacount.valfile = mangadapdb.cleanspaxelprop7.file_pk WHERE drpalias.pk = 32 AND dapalias.pk = 34 AND goodhacount.valcount >= 0.2 * bingood.goodcount) AS anon_1 GROUP BY anon_1.mangadatadb_cube_mangaid, anon_1.mangadatadb_cube_plate, concat(anon_1.mangadatadb_cube_plate, '-', anon_1.mangadatadb_ifudesign_name), anon_1.mangadatadb_ifudesign_name
```

Wall: 33min

Spark

```
In [14]: with Timer():
             # get total counts of number of good spaxels, grouped by plateifu
             tc = maps.filter(good_spaxels).groupby('plateifu').count().withColumnRenamed('count', 'totalc')
             # get counts of number of good spaxels with H-alpha > 5, grouped by plateifu
             fc = maps.filter(good_spaxels).filter(maps['emline_gflux_ha_6564'] > 5).groupby('plateifu').count().withColumnRenamed
         ('count', 'filterc')
             # join the tables and filter where
             tmp = tc.join(fc, 'plateifu')
             tmp.filter(tmp.filterc >= 0.2 * tmp.totalc).count()
```

Wall: ~7 seconds

Case Study: SDSS MaNGA

- Spark not always faster
  - Overheads in managing sessions could make it quite a bit slower for short queries
  - Can’t beat highly optimized DB that fit in memory of single node
- Mixed mode (size dependent)
  - SQL for some use-cases, e.g. object metadata
  - Spark for others, e.g. custom codes, ML, complex queries
  - General idea behind MS BDC

	PSQL	Spark
Q1	<5s	<5s
Q2	33min	<10s
Q3	3min	<10s

Case Study: ZTF local/AWS

- Obtained parquet files from our UW colleagues (thanks!)
- > 1B objects, light curves in up-to 3 bands of varying number of epochs
- ~3TB compressed
- Compare local Spark cluster with AWS EMR+S3

Case Study: ZTF local/AWS

- Task: Clustering on lightcurve + object features. Unsupervised analog to [1]
  - Involves SQL for subselect, cleaning
  - Custom codes for feature extraction (difficult/impossible with standard SQL)
  - Heavy computations for light-curve period folding (esp when considering 1B+ objects)

In [16]:
```
spark.sql(f'''
SELECT 'r-only' cat, count(*) n FROM ztf_nobs WHERE nobs_r >= {min_obs}
UNION SELECT 'g+r' cat,count(*) n FROM ztf_nobs WHERE nobs_g >= {min_obs} AND nobs_r >= {min_obs}
UNION SELECT 'g+r+i' cat,count(*) n FROM ztf_nobs WHERE nobs_g >= {min_obs} AND nobs_r >= {min_obs} AND nobs_i >= {min_obs}
ORDER BY n DESC
''').toPandas().set_index('cat')/ztf_N
```

```
    featurized_rdd = spark.sql(f'SELECT * FROM {table} WHERE stripe = {stripe}') \
        .rdd \
        .map(filter_lc_by_flags) \
        .filter(lambda r: r['nobs_r'] > min_obs) \
        .map(lambda r: (r['ps1_objid'], r)) \
        .mapValues(featurize_row) \
        .map(to_feature_cols)
    spark.createDataFrame(featurized_rdd, schema=schema, verifySchema=False) \
        .write.mode('overwrite').parquet(feature_stripe)
```

[1] Van Rostel, Jan et al. “The ZTF Source Classification Project I. Methods and Infrastructure”

Case Study: ZTF local/AWS

- Local vs Remote
  - Local:
    - Recycled ~10 year old hardware for experiments
    - Limited scalability, takes some effort to add hardware
    - Critically, we already have an operational data center
  - Cloud (AWS):
    - State-of-the art hardware
    - Essentially unlimited scaling
    - Costly (in your face - but easy to understand)

Feature generation:

AWS EMR:
- 40 x c5a.4xlarge [16 core/ 32GB]
- Wall time: ~16 hrs
- Cost: ~$360 (ec2 spot/emr/s3)

Local:
- 14 x [24 core / 48 GB]
- Wall time: ~100 hrs
- Cost: ~$15 *

** assuming 14 nodes at $10,000 each with 5 year lifetime, constant use. Ele: 1kw @ $0.09/hr per node No management/running costs considered. Large grain of salt required!*

Case Study: ZTF local/AWS

- Features close to > typical node memory
- Interested in DBSCAN over Kmeans, but more difficult to distribute - not included in Spark ML library
- Even after PCA-based dimensionality reduction, single-thread neighbor search expensive (tree generation)
  - Long haul feature generation on Spark
  - Transfer ~40GB of PCA-features to SciServer compute
  - DBSCAN using scikit-learn - *exceeds jobs timeouts*

Conclusions

- SciServer does data-proximate big-data - Adding Spark can turn big -> BIG
- Standard SQL databases have some limitations
  - Often designed with specific use-case focus
  - Arbitrary functions against data non-trivial
  - Search and retrieve / summary
- With Spark
  - Less structured - throw cores at the problem
  - Highly scalable - throw cores at the problem
  - Save some cores with clever storage formats (like parquet)
  - Generality - run SQL and arbitrary code - don’t have to know all the questions
  - Use as system for scaling data (re-)processing/ETL

Conclusions

- Cloud
  - Costs - on demand
  - Scalability - more control over bringing down walltime
- Local
  - Can be cost-effective - older hardware mixed hardware
  - Difficult to scale

Conclusions

- Interfaces
  - So far, our experiments were a mix of SQL and Python code from Jupyter notebooks
  - SQL from CasJobs-like interface might be desirable
    - Overhead in scheduling spark sessions
    - Experimenting with Trino (forked from PrestoDB)