Dask and Parquet

Dr. Scott Gorlin
Harvard University
Fall 2018

Agenda

- Salted Graphs
- Data Containers
- Parquet
- Dask
- Luigi Targets
- Dask and Luigi
- Final Project

Salted Graphs

A Hash by Any Other Name

> *I hate updating the `__version__` manually. Why not use `inspect.getsource(MyTask.run)`?*
>
> — *A student*

Automatic Hashing

An ***automatic hash*** is a Really Good Idea

```
def auto_version(SomeTask):
    SomeTask.__version__ = inspect.getsource(
        SomeTask.run)

@auto_version
class MyTask(Task):
    def run(self):
        ...
```

Automatic Hashing

An ***automatic hash*** is a Really Good Idea

… which is hard to implement!

This schema is too sensitive. It has poor recall (comments will change the version) and is not sensitive to changes elsewhere

```
def auto_version(SomeTask):
    SomeTask.__version__ = inspect.getsource(
        SomeTask.run)

@auto_version
class MyTask(Task):
    def run(self):
        output = some_func(self.input())
```

```
def some_func(data):
    # This code changes!
    # __version__ doesn't know
```

Automatic Hashing

An ***automatic hash*** is a Really Good Idea

There is no principled way of inspecting ***all*** the relevant code!

If you can afford to rerun the data pipeline on every code release, you should not be persisting the output.

```
def auto_version(SomeTask):
    SomeTask.__version__ = "{}-{}".format(
        SomeTask.__name__,
        pkg.__version__ # Commit/git tag
    )

@auto_version
class MyTask(Task):
    def run(self):
        output = some_func(self.input())
```

```
def some_func(data):
    # Now we're covered...
    # IF this code is in pkg!
```

You know best

Manually specifying the task version is like manually specifying a release version.

It is tempting to automate it, but you will find yourself debugging code *just* to make your version scheme work.

It is much easier to inject your knowledge directly.

The Signature Input

If your task is simple enough, you may have luck with a ***signature input***.

This input should be rich enough to expose any nuance, feature, or bug.

You call the task on the input, and hash the output!

```
class AutoVersion:
    def __get__(self, task, owner):
        return hash(
            task.process(task.SIGNATURE_INPUT))
```

```
class AddTask(Task):
    __version__ = AutoVersion()
    SIGNATURE_INPUT = 0
    value = IntParameter()

    def process(self, data):
        return add(data, self.value)

    def run(self):
        with self.output().open('w') as f:
            f.write(self.process(self.input()))
```

The Point of Requirement

Luigi Utils has two cool compositional tools:

```
# Copies Parameters
@inherits(TaskA)
class TaskB:
    ...

# Copies and requires
@requires(TaskB)
class TaskC(Task):
    ...
```

… which seemingly work well for basic parameter sharing

The Point of Requirement

Luigi Utils has two cool compositional tools:

```
# Copies Parameters
@inherits(TaskA)
class TaskB:
    param = Parmater()

# Copies and requires
@requires(TaskB, TaskD, ...)
class TaskC(Task):
    ...
```

Although, they force inheritance of unnecessary parameters

What if `param` is not meaningful for `TaskC`?

What if it wants to use a different value?

You must change `requires()` to modify any requirement

The Point of Requirement

Our approach is more extensible

```
class TaskA(Task):
    requires = Requires()

    # Type completion for forks
    in1 = Requirement(Task1)
    in2 = Requirement(Task2)

class TaskA2(TaskA):
    # Auto inherits in1, in2
    in3 = Requirement(Task3)
```

```
class ComplexRequirement(Requirement):
    def __init__(self, factory):
        self.factory = factory

    def __get__(self, task, owner):
        return self.factory(task)

class TaskC(Task):
    data = ComplexRequirement(
        lambda task: task.clone(
            OtherTask, param=translate[task.param])
    )
```

The Point of Requirement

Our approach is more extensible

```
class NeighborhoodTask(Task):
    neighbors = NeighorhoodRequirements(Task1)

class RollingAggregation(Task):
    history = HistoricalRequirements(Task2, days=7)
```

```
class NeighborhoodRequirements(Requirement):
    def __get__(self, task, owner):
        neighbors = {}
        for yn, yo in {'n':1, 'c':0, 's':-1}:
            for xn, xo in {'e':1, 'c':0, 'w':-1}:
                neighbors[yn+xn] = task.clone(
                    self.task_class,
                    x=task.x + xo,
                    y=task.y + yo,
                )
        return neighbors
```

NB: I’m ignoring a few details like nested requirements, but hopefully the point is clear

Data Containers

Row and Column Stores

Row Storage

Any csv:

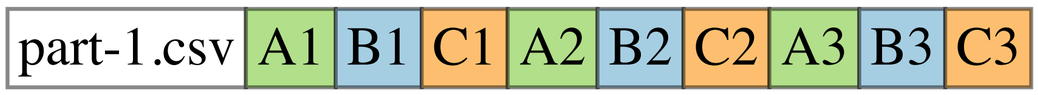
Column Storage

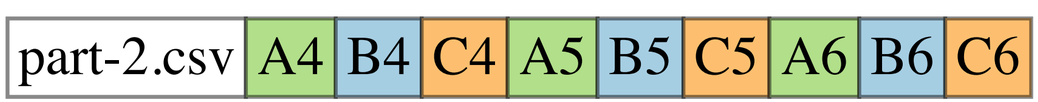
Parquet *et al*:

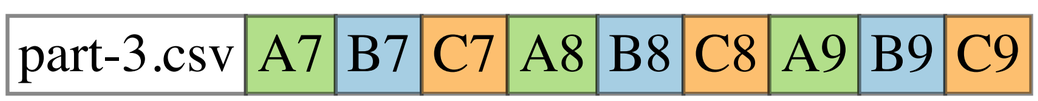
Both systems are designed for ***tabular, structured data***. The schema may be explicit or implicit, but is assumed fixed or slowly changing.

Row and Column Stores

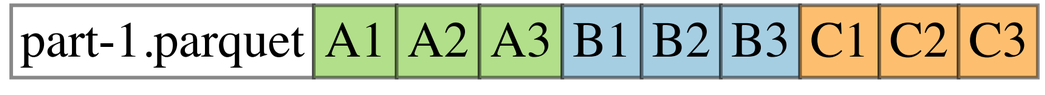
Row Storage

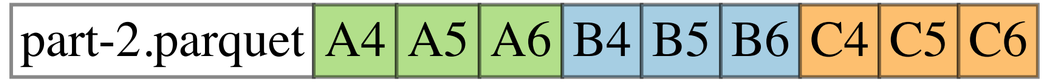


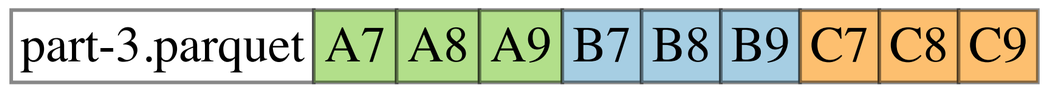




Column Storage







Both systems can be ***sharded*** across multiple files, creating row groups or stripes

Key-Value Stores

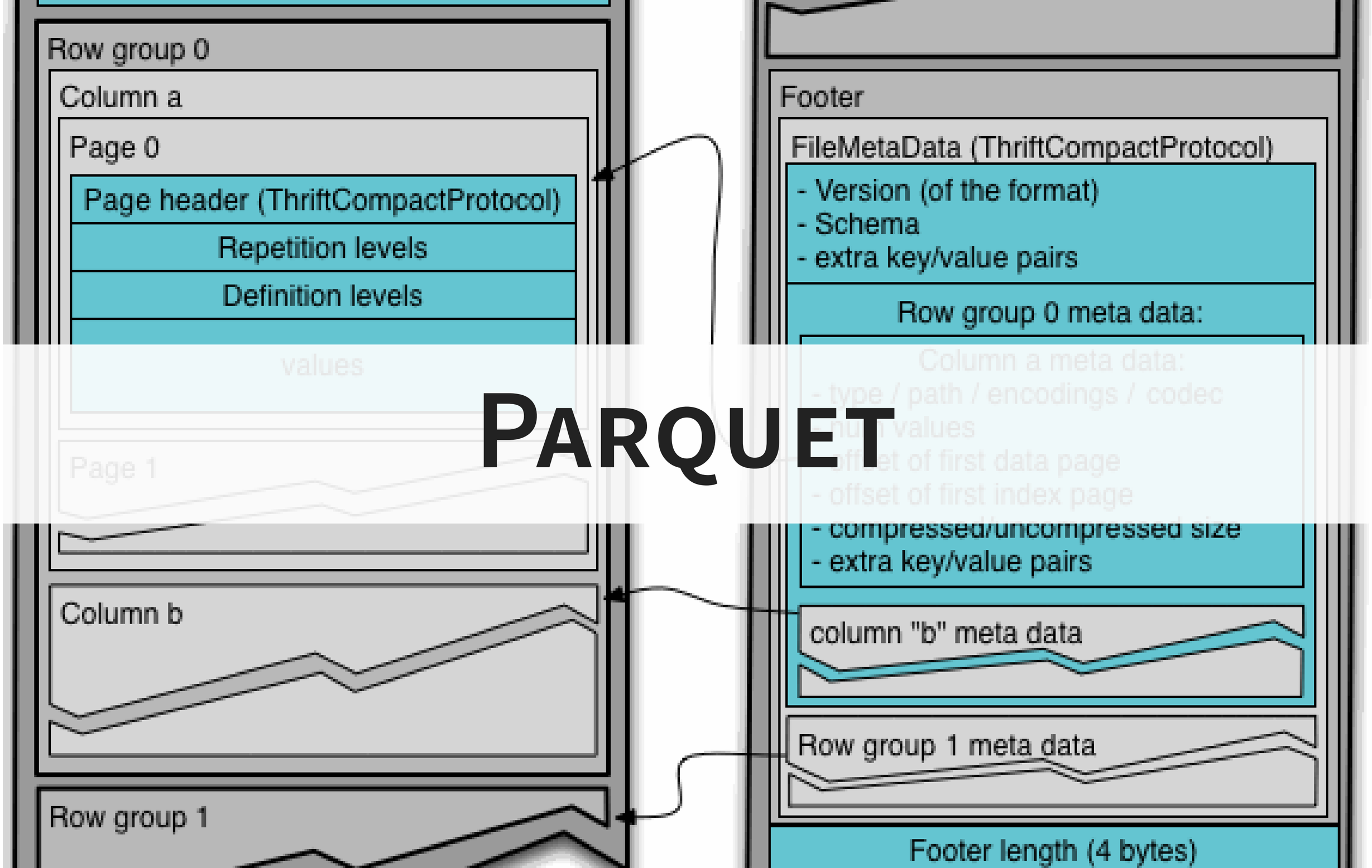
Something completely different

Offer fast lookup/scan on key, but not other values

Values may be arrays, unstructured, dicts, …

Key ranges may be partitioned

Key	Value
A1	B=B1; C=C1
A2	B=B2; C=C2
A3	B=B3; C=C3
A4	C=C3; D=D4


Row group 0
Column a
Page 0
Page header (ThriftCompactProtocol)
Repetition levels
Definition levels
values
Page 1
Column b
Row group 1
Footer
FileMetaData (ThriftCompactProtocol)
- Version (of the format)
- Schema
- extra key/value pairs
Row group 0 meta data:
Column a meta data:
- type / path / encodings / codec
- num values
- offset of first data page
- offset of first index page
- compressed/uncompressed size
- extra key/value pairs
column "b" meta data
Row group 1 meta data
Footer length (4 bytes)

PARQUET

Column stats

Parquet stores statistics about each row group: min/max vals, n unique vals, …

… as well as *which row groups/parts* exist, optionally

Column Stats

In each partfile

```
dataset/
    part.0.parquet # header stores metadata/stats
```

Dask reads these fine, but must glob to know parts and read every header for divisions. Annoyingly slow.

Explicit Metadata

```
dataset/
    _common_metadata # Explicit schema
    _metadata  # Summary file
    part.0.parquet
```

Much faster to get schema/stats - just read 1 file!

Spark does not save summary metadata by default!

```
df.write
.option("parquet.enable.summary-metadata", "true")
.parquet("{{ output.path }}")
```

Predicate Pushdown

Parts of a query (the predicates, eg `WHERE`) can be pushed to the data engine

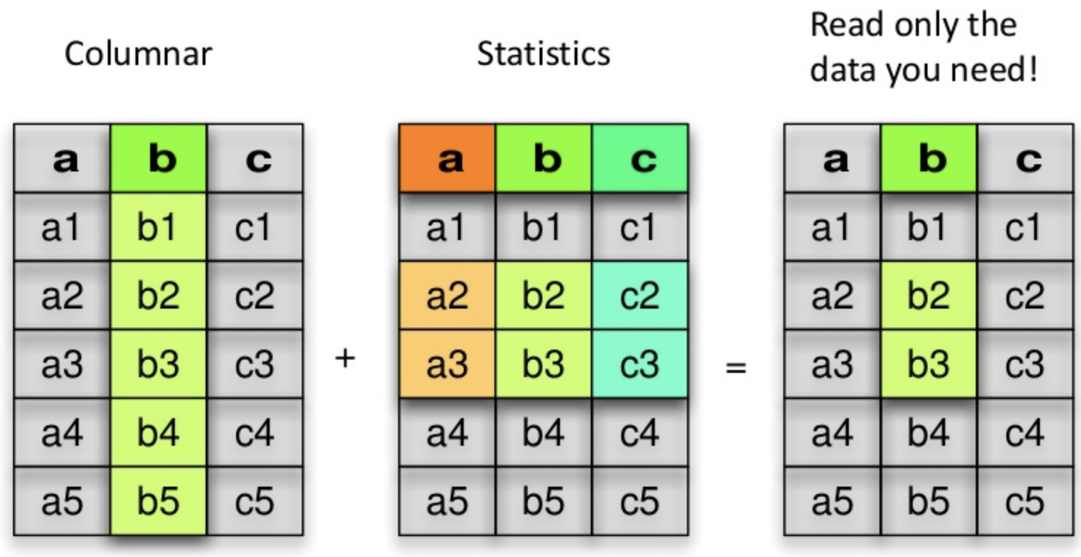
This can dramatically impact performance by *skipping some data reads entirely*

This plays the same role as an index in an RDBMS; it is not as effective, but does work for distributed reads

Predicate Pushdown

```
select b from table where a2 <= table.a <= a3
```

```
read_parquet(
    path,
    # Projection pushdown
    columns=['b'],

    # Predicate pushdown
    filters=[
        # NB: doesn't guarantee, only filters group
        ('a', '>=', a2),
        ('a', '<=', a3)
    ],
)
```



The Columnar Roadmap

Predicate Pushdown (approx!)

Predicate pushdown is not always exact - it may be used to ***exclude*** chunks, but you may still get rows that don't match your criteria.

Use them as an optimization technique, not a query

```
read_parquet(path, filters=[('a', '=', 5)]).query("a == 5")
```

A Tale of Two (Parquet) Engines

FastParquet

- Designed for dask
- Col stats and predicate pushdowns
- Default ***dask*** engine

- Dask contributors only - code quality may be lower

PyArrow

- Official package
- Default ***pandas*** engine

- No filters/pushdowns!

Dask

Refresher - The Basics

```python
def some_df_function(dataframe):
    """Expects/returns a dataframe or series"""
    return dataframe['a'] + dataframe['b']

def some_scalar_function(row):
    """Operates elementwise or on a row series"""
    return row + 1
```

```python
ddf = dd.read_parquet(...)
ddf.groupby('a').mean()  # Pandas API
ddf.map_partitions(some_df_function, meta=float)
ddf.apply(some_scalar_function)
```

Refresher - The Basics

Avoid compute!

```
df = ddf.compute()
df['new_column'] = 1
dd.from_pandas(df, chunksize=10).to_parquet(...)
```

It's ***only*** necessary when the ***graph*** is dependent on a computed value, eg `ddf.compute().T`, where the schema can't be known until the rows are computed.

Avoid repartition!

```
# This is taking too long! Let me rebalance
ddf.repartition(divisions)
```

Most performance issues and bugs in dask are due to missing divisions or wrong metadata. Instead of repartitioning or persisting and reading back, fix the metadata issue.

Predicates

Dask exposes predicate pushdown in two ways:

Parquet

```
read_parquet(
    path,

    # Predicate pushdown
    filters=[
        ('a', '>=', a2),
        ('a', '<=', a3)
    ],
)
```

The Index

```
>>> ddf = dd.read_parquet('data/hashed')
Dask DataFrame Structure:
                learn project
npartitions=5
0d733444       object  object
33174d90          ...     ...
...               ...     ...
d9a774d1          ...     ...
ffc21800          ...     ...
Dask Name: read-parquet, 5 tasks
```

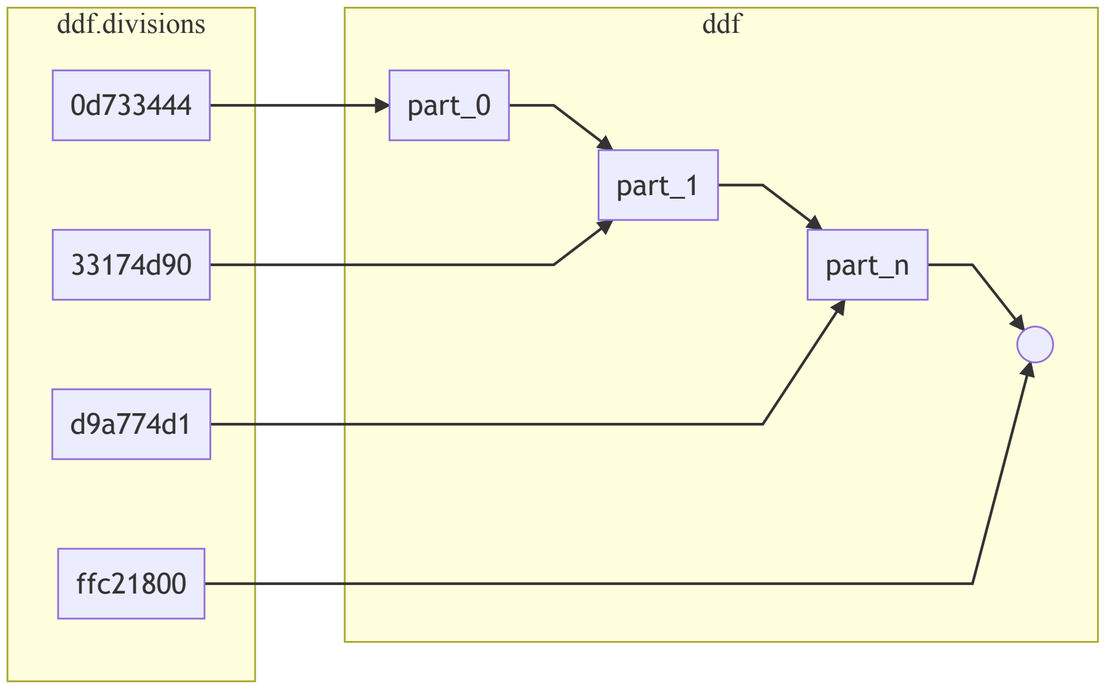
Predicates

Recall `df.divisions` are the index bounds of each dask partition

```
ddf.divisions[i]
<= ddf.get_partition(i).index
< ddf.divisions[i+1]
```

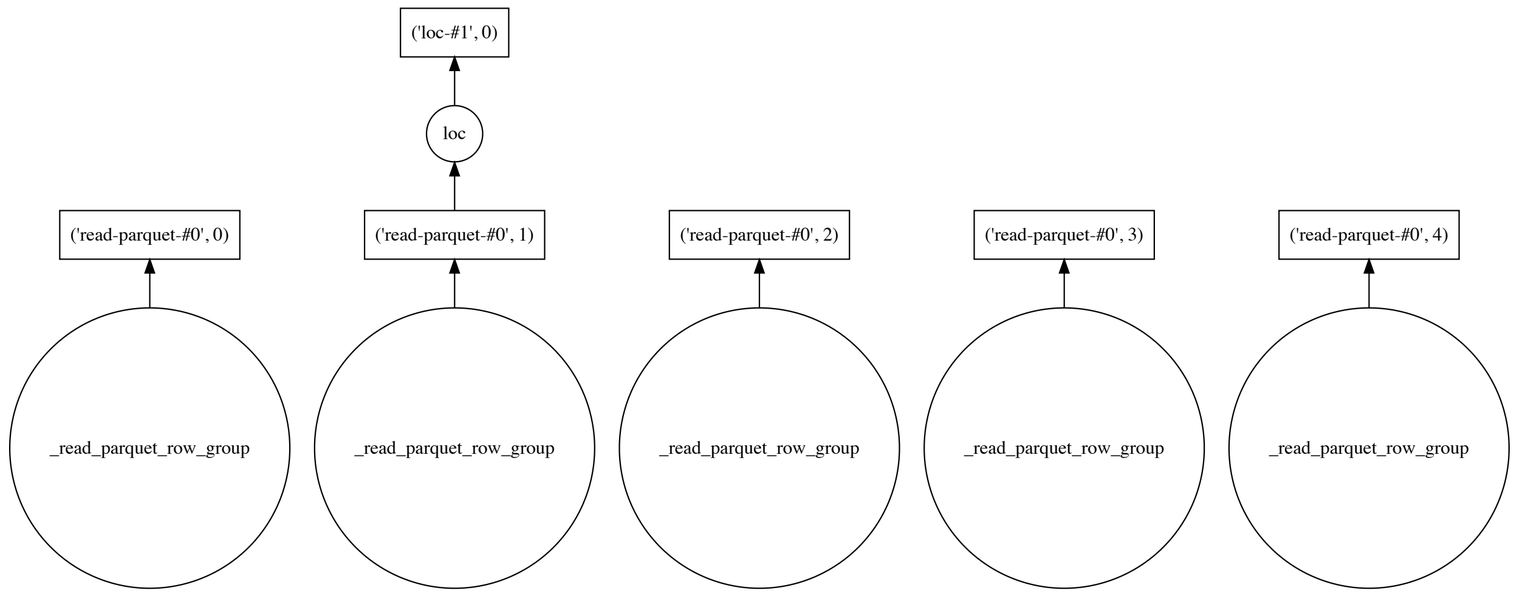
Beware the known design flaw: `ddf.divisions[-1]` is the ***maximum value*** of the index in `ddf.get_partition(-1)`

See #3384 and PR-3430



Predicates

```
>>> df.loc['57019809']
Dask DataFrame Structure:
                learn project
npartitions=1
57019809       object  object
57019809          ...     ...
Dask Name: loc, 6 tasks
```



Most dask operations are partition- and index-aware, and will operate intelligently.

Be sure that `divisions` are set!

Predicates

Dask does ***not*** use any other parquet metadata. Any advanced filtering, partition magic, etc must be pushed into read_parquet or make use of the dataframe divisions. Or use Spark.

Dask DataFrames as Key/Value

The performance boost using index divisions is massive, and makes us think differently about a `dask.DataFrame`

Basic aggregations are fine, but if your task involves a `WHERE`, `GROUP BY`, etc, a sorted index column is critical for performance

This makes the `DataFrame` more like a Key/Value store!

Use the Index

Indexed DataFrames

Pandas

```
>>> ddf.head()
                    learn    ...       semester
hashed_id
0d733444   I want to ...     ...      2018-fall
19fd3c63   Understand...     ...      2018-fall
```

```
>>> ddf.head().reset_index().set_index(['semester',
'hashed_id'])
                      learn        project
semester  hashed_id
2018-fall 0d733444   I want ...   The project...
```

Design your index to be searchable, and write functions on the surrogate index!

Surrogate Index

```
def get_key(row):
    return '{}-{}'.format(row.semester, row.name)

>>> ddf.assign(
        key=ddf[['semester']].apply(
            get_key, axis=1, meta=object)
    ).set_index('key')
```

```
Dask DataFrame Structure:
                        learn project semester
npartitions=5
fall_2018-0d733444  object  object   object
fall_2018-33174d90     ...     ...      ...
...                    ...     ...      ...
fall_2018-d9a774d1     ...     ...      ...
fall_2018-ffc21800     ...     ...      ...
Dask Name: sort_index, 35 tasks
```

Indexed DataFrames

```
Dask DataFrame Structure:
                     learn project semester
npartitions=5
fall_2018-0d733444  object  object   object
fall_2018-33174d90     ...     ...      ...
...                    ...     ...      ...
fall_2018-d9a774d1     ...     ...      ...
fall_2018-ffc21800     ...     ...      ...
Dask Name: sort_index, 35 tasks
```

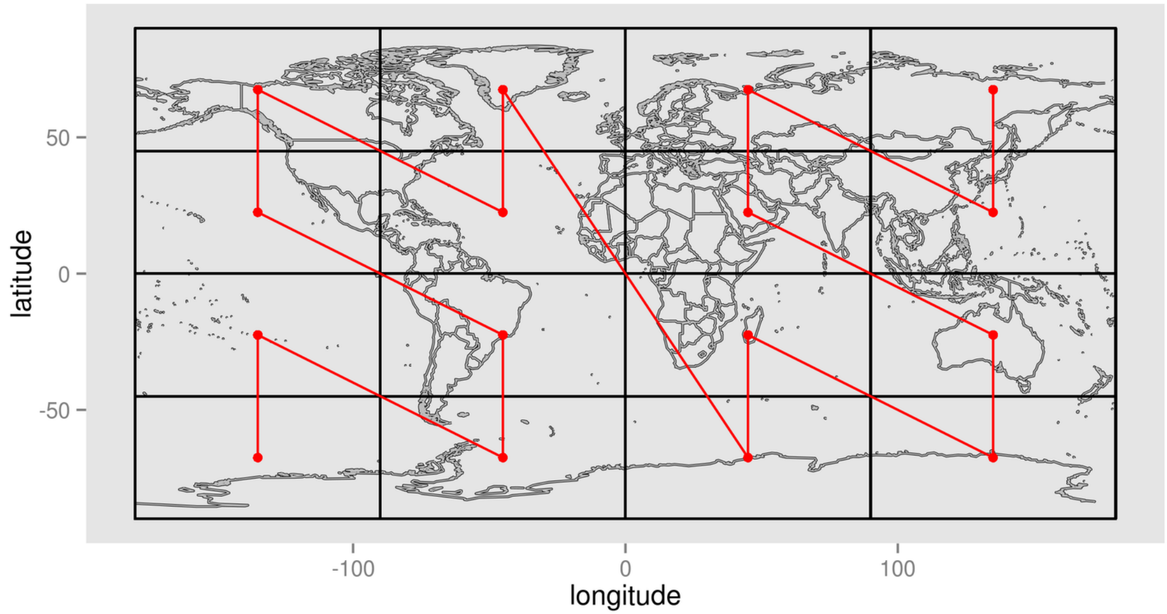
```
def query_semester(ddf, semester):
    return ddf.loc[semester+'-':semester+'-g']

def query_student(ddf, hashed_id, semester=None):
    if semester:
        return ddf.loc[
            '{}-{}'.format(hashed_id, semester)
        ]
    return dd.concat([
        query_student(ddf, hashed_id,
            semester=s)
        for s in all_semesters
    ])
```

Surrogate keys take some mental adjustment. You need to design the store up front for what you will query on, or rely on full table scans

Fancy Indexing

Every point on earth can be mapped along a 1-d, fractal, space filling curve

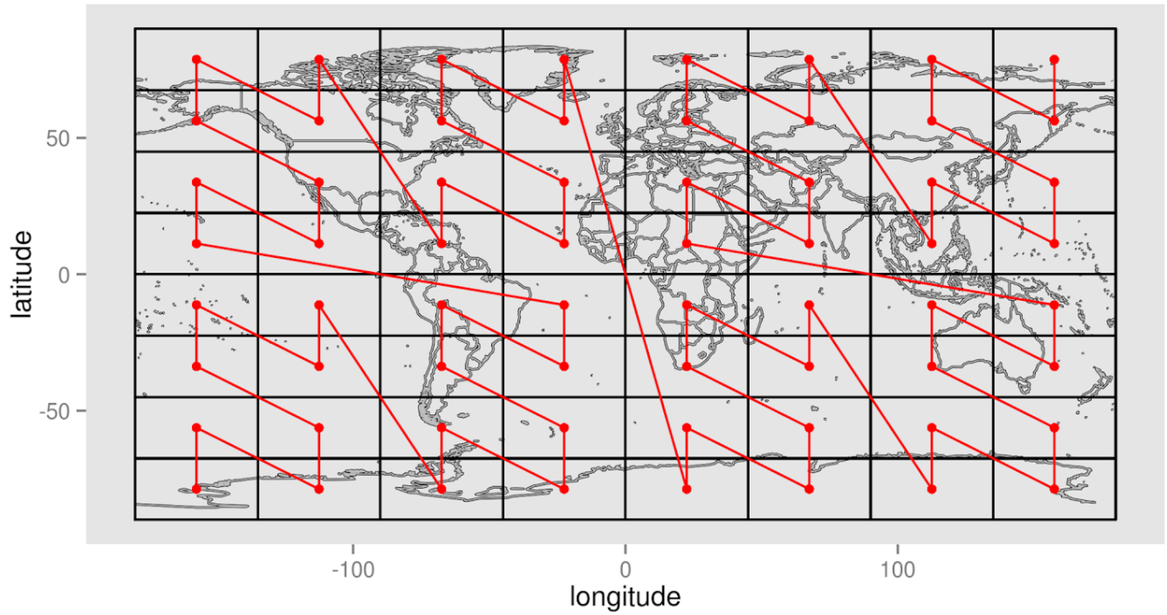
Querying a region of space is accomplished via integer range scans on this curve



Fancy Indexing

Every point on earth can be mapped along a 1-d, fractal, space filling curve

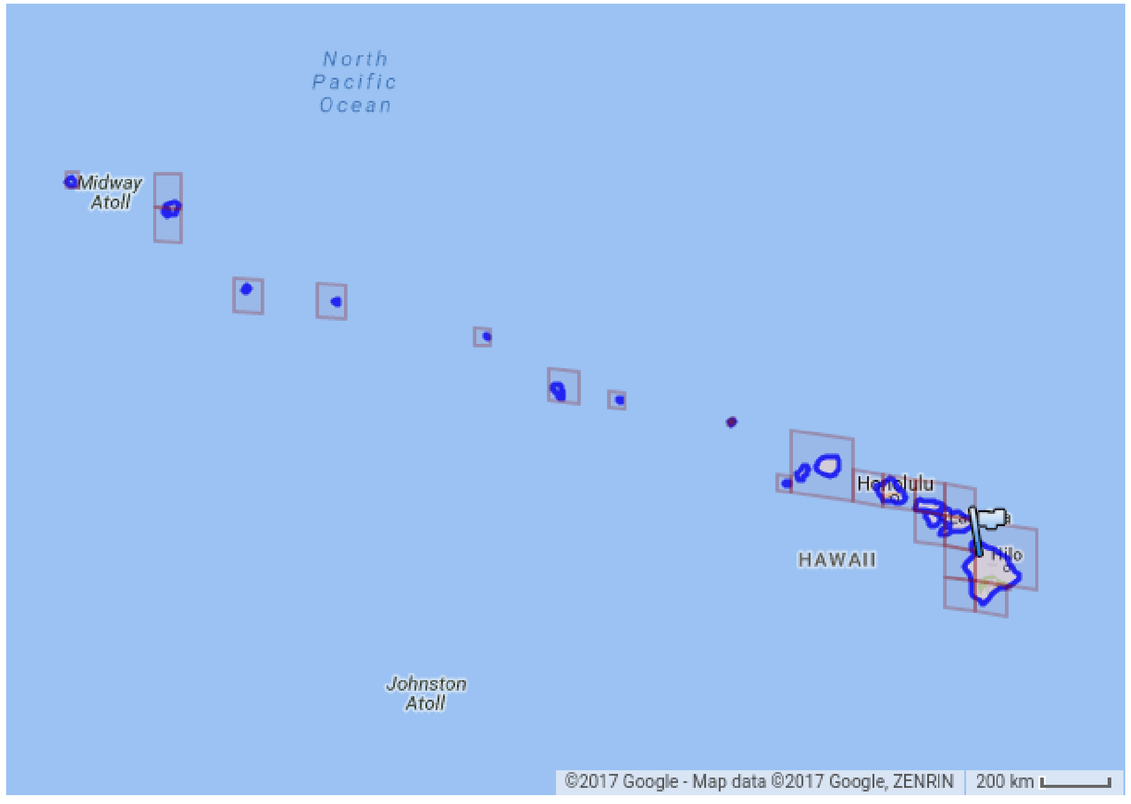
Querying a region of space is accomplished via integer range scans on this curve



Fancy Indexing

Shapes to Slices

```
# A geospatial integer-indexed ddf
# slc eg == slice(102013231, 1231321421, 1)
# the bounds of some higher level cell which covers
# part of the shape
local = dd.concat([
    ddf.loc[slc]
    for slc in get_covering_slices(shape)
])
```



Partitions

Implicit

```
ddf.to_parquet('data/flat')
```

```
flat/
    _common_metadata
    _metadata
    part.0.parquet
    ...
    part.4.parquet
```

Explicit

```
ddf.to_parquet('data/partitioned', partition_on=['semester'])
```

```
partioned/
    _common_metadata
    _metadata
    semester=fall_2018/
        part.0.parquet
        ...
        part.4.parquet
```

Partitions

Implicit

```
>>> ddf.read_parquet('data/flat')
Dask DataFrame Structure:
                     learn project
semester
npartitions=5
fall_2018-0d733444  object  object
object
fall_2018-33174d90     ...     ...
...
...                    ...     ...
...
fall_2018-d9a774d1     ...     ...
...
fall_2018-ffc21800     ...     ...
```

Explicit

```
>>> dd.read_parquet('data/partitioned')
Dask DataFrame Structure:
                     learn project         semester
npartitions=5
fall_2018-0d733444  object  object  category[known]
fall_2018-33174d90     ...     ...              ...
...                    ...     ...              ...
fall_2018-d9a774d1     ...     ...              ...
fall_2018-ffc21800     ...     ...              ...
Dask Name: read-parquet, 5 tasks
```

Partitions

But, Dask doesn't handle all partition schemes well!

```
>>> ddf.to_parquet('data/by_grad')
learn                grad
key
fall_2018-0d733444   True
fall_2018-19fd3c63   False
```

```
by_grad/
    _metadata
    grad=True/
        part.0.parquet
        ....
```

```
>>> dd.read_parquet('data/by_grad')
Dask DataFrame Structure:
                  learn project semester                grad
npartitions=9
                 object  object   object  category[known]
...                 ...     ...      ...                ...
Dask Name: read-parquet, 9 tasks
```

Note we lose `divisions` because the index is no longer globally sortable

Partitions

Dask only handles explicit partitions well if the partition schemes preserve global sortability of the index. Spark *et al* do a better job.

Explicit predicate pushdowns should be ok, just not operations on the index

See related discussion on Github #3384 and PR-3430


Luigi Targets

Targets

No No

```
class MyTask(Task):
    def run(self):
        with open(self.output().path, 'w') as f:
            f.write()
```

Better

```
class MyTask(Task):
    def run(self):
        with self.output().open('w') as f:
            f.write()
```

Allows the target to encapsulate file format, atomicity, etc

Remote Targets

… and to abstract the *file sytem*:

```
class FileSystemTarget:
    fs = None

    def open(self):
        return self.fs.open()
```

```
class LocalTarget(FileSystemTarget):
    fs = LocalFileSystem()

class S3Target(FileSystemTarget):
    fs = S3FileSystem()

class RemoteTarget(FileSystemTarget):
    fs = RemoteFileSystem()  # (S)FTP
```

You task doesn’t care where the data is!

(as long as you avoid using `target.path`)

Remote Configuration

Of course, we need to configure new systems

```
from luigi.contrib.ftp import RemoteTarget
t = RemoteTarget(
    # Is the home directory config or code??
    '/home/c/s/cscie29/hashed.xlsx',
    # What about the host?
    'fas.harvard.edu',
    sftp=True,
    # These are definitely secrets
    username="cscie29ro", password="password",
)
t.exists()  # Checks using SFTP
```

Remote Configuration

One possibility:

```
def get_config_suite(prefix):
    n = len(prefix) + 1
    kw = {
        k[n:].lower(): v
        for k, v in os.environ.items()
        if k.startswith(prefix + '_')
    }
    return kw
```

```
>>> os.environ['CSCI_FTP_HOST'] = 'fas.harvard.edu'
...
>>> get_config_suite('CSCI_FTP')
{
 'host': 'fas.harvard.edu',
 'username': 'cscie29ro',
 'password': 'password'
}
```

```
cfg = get_config_suite(prefix)
host = cfg.pop('host')
RemoteTarget(path, host, **cfg)
```

Dask and Luigi

Targets

Parquet (and other dask collections) look for folders, not files

They implement their own file protocols, and don’t want to use luigi’s

Targets

```
join = os.path.join
class BaseDaskTarget(Target):
    def exists(self):
        return os.path.isfile(join(self.path, '_SUCCESS'))

    def write(self, collection, *args, **kwargs):
        self._write(collection, *args, **kwargs)
        with open(join(self.path, '_SUCCESS')) as f:
            # Create flag success file
            pass

    def read(self, *args, **kwargs):
        raise NotImplementedError()
```

```
class ParquetTarget(BaseDaskTarget):
    def _write(self, collection, **kw):
        collection.to_parquet(
            self.path, **kwargs)
    def read(self, **kw):
        return dd.read_parquet(
            self.path, **kwargs)
```

Targets

The previous implementation is thematic, not exact

Local dask targets could use a temporary directory which is moved (though `.exists()` still prefers a flag file)

Dask has interfaces to S3 and HDFS, so you should avoid `os.*` and rely on internal dask equivalents (eg `s3fs`), which exist, but are beyond the scope of this course.

Final Project

Start thinking

Anything you can clearly relate to course material:

- ***Contribute*** to a popular OSS project
- ***Solve a problem from work*** and write up a case study
- ***Write a problem set or lecture*** on an exciting topic
- ***Write a tool*** you wish you had for this course
- ***Find an answer*** to $a^n + b^n = c^n$ for $n > 2$

Proposals (non-binding) will be due soon

Expected effort approximately 10-20 hours