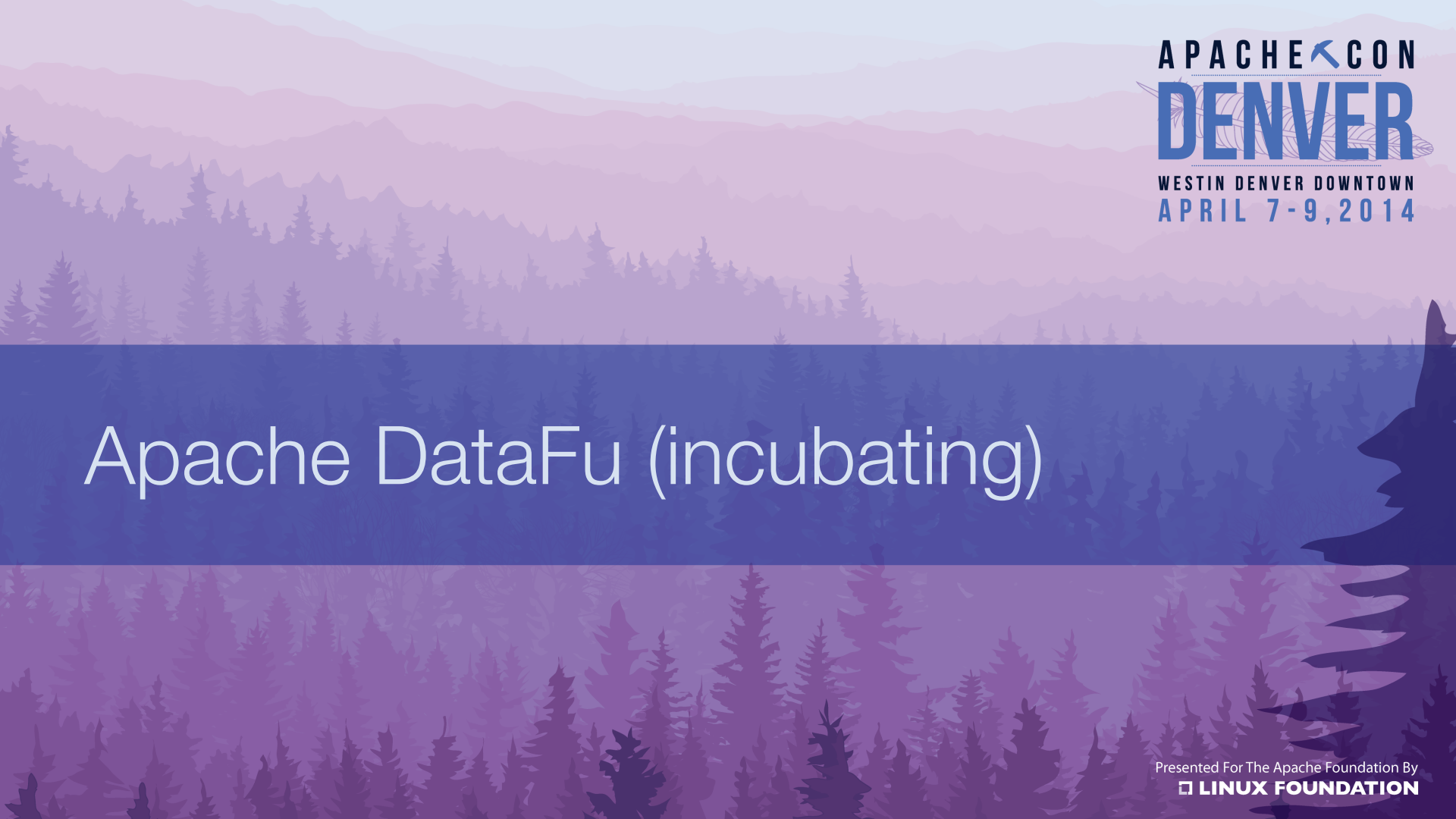


Apache DataFu (incubating)

Presented For The Apache Foundation By
LINUX FOUNDATION

APACHECON
DENVER
WESTIN DENVER DOWNTOWN
APRIL 7-9, 2014

William Vaughan

Staff Software Engineer, LinkedIn

www.linkedin.com/in/williamgvaughan

Presented For The Apache Foundation By
LINUX FOUNDATION

Apache DataFu

- Apache DataFu is a collection of libraries for working with large-scale data in Hadoop.
- Currently consists of two libraries:
    - DataFu Pig – a collection of Pig UDFs
    - DataFu Hourglass – incremental processing
- Incubating

History

- LinkedIn had a number of teams who had developed generally useful UDFs
- Problems:
    - No centralized library
    - No automated testing
- Solutions:
    - Unit tests (PigUnit)
    - Code coverage (Cobertura)
- Initially open-sourced 2011; 1.0 September, 2013

What it's all about

- Making it easier to work with large scale data
- Well-documented, well-tested code

- Easy to contribute
  - Extensive documentation
  - Getting started guide
  - i.e. for DataFu Pig – it should be easy to add a UDF, add a test, ship it

DataFu community

- People who use Hadoop for working with data
- Used extensively at LinkedIn
- Included in Cloudera's CDH
- Included in Apache Bigtop

APACHECON
DENVER
WESTIN DENVER DOWNTOWN
APRIL 7-9, 2014

DataFu - Pig

Presented For The Apache Foundation By
LINUX FOUNDATION

DataFu Pig

- A collection of UDFs for data analysis covering:
  - Statistics
  - Bag Operations
  - Set Operations
  - Sessions
  - Sampling
  - General Utility
  - And more..

Coalesce

- A common case: replace null values with a default

```
data = FOREACH data GENERATE (val IS NOT NULL ? val : 0) as result;
```

- To return the first non-null value

```
data = FOREACH data GENERATE (val1 IS NOT NULL ? val1 :
                              (val2 IS NOT NULL ? val2 :
                              (val3 IS NOT NULL ? val3 :
                                 NULL))) as result;
```

Coalesce

- Using Coalesce to set a default of zero

```
data = FOREACH data GENERATE Coalesce(val,0) as result;
```

- It returns the first non-null value

```
data = FOREACH data GENERATE Coalesce(val1,val2,val3) as result;
```

Compute session statistics

- Suppose we have a website, and we want to see how long members spend browsing it
- We also want to know who are the most engaged
- Raw data is the click stream

```
pv = LOAD 'pageviews.csv' USING PigStorage(',')
     AS (memberId:int, time:long, url:chararray);
```

Compute session statistics

- First, what is a session?
  - Session = sustained user activity
  - Session ends after 10 minutes of no activity

```
DEFINE Sessionize  datafu.pig.sessions.Sessionize('10m');
```

- Session expects ISO-formatted time

```
DEFINE UnixToISO
org.apache.pig.piggybank.evaluation.datetime.convert.UnixToISO();
```

Compute session statistics

- Sessionize appends a sessionId to each tuple
- All tuples in the same session get the same sessionId

```
pv_sessionized = FOREACH (GROUP pv BY memberId) {
  ordered = ORDER pv BY isoTime;
  GENERATE FLATTEN(Sessionize(ordered))
           AS (isoTime, time, memberId, sessionId);
};

pv_sessionized = FOREACH pv_sessionized GENERATE
                 sessionId, memberId, time;
```

Compute session statistics

- Statistics:

```
DEFINE Median   datafu.pig.stats.StreamingMedian();
DEFINE Quantile datafu.pig.stats.StreamingQuantile('0.90','0.95');
DEFINE VAR      datafu.pig.stats.VAR();
```

- You have your choice between streaming (approximate) and exact calculations (slower, require sorted input)

Compute session statistics

- Computer the session length in minutes

```
session_times =
  FOREACH (GROUP pv_sessionized BY (sessionId,memberId))
           GENERATE group.sessionId as sessionId,
                    group.memberId as memberId,
                    (MAX(pv_sessionized.time) -
                     MIN(pv_sessionized.time))
                     / 1000.0 / 60.0 as session_length;
```

Compute session statistics

- Compute the statistics

```
session_stats = FOREACH (GROUP session_times ALL) {
  GENERATE
    AVG(ordered.session_length) as avg_session,
    SQRT(VAR(ordered.session_length)) as std_dev_session,
    Median(ordered.session_length) as median_session,
    Quantile(ordered.session_length) as quantiles_session;
};
```

Compute session statistics

- Find the most engaged users

```
long_sessions =
  filter session_times by
    session_length >
    session_stats.quantiles_session.quantile_0_95;

very_engaged_users =
  DISTINCT (FOREACH long_sessions GENERATE memberId);
```

Pig Bags

- Pig represents collections as a bag
- In PigLatin, the ways in which you can manipulate a bag are limited
- Working with an inner bag (inside a nested block) can be difficult

DataFu Pig Bags

- DataFu provides a number of operations to let you transform bags
  - AppendToBag – add a tuple to the end of a bag
  - PrependToBag – add a tuple to the front of a bag
  - BagConcat – combine two (or more) bags into one
  - BagSplit – split one bag into multiples

DataFu Pig Bags

- It also provides UDFs that let you operate on bags similar to how you might with relations
  - BagGroup – group operation on a bag
  - CountEach – count how many times a tuple appears
  - BagLeftOuterJoin – join tuples in bags by key

Counting Events

- Let's consider a system where a user is recommended items of certain categories and can act to accept or reject these recommendations

```
impressions = LOAD '$impressions' AS (user_id:int, item_id:int,
   timestamp:long);
accepts = LOAD '$accepts' AS (user_id:int, item_id:int, timestamp:long);
rejects = LOAD '$rejects' AS (user_id:int, item_id:int, timestamp:long);
```

Counting Events

- We want to know, for each user, how many times an item was shown, accepted and rejected

```
features: {
  user_id:int,
  items:{(
    item_id:int,
    impression_count:int,
    accept_count:int,
    reject_count:int)}
}
```

Counting Events

One approach…

```
-- First cogroup
features_grouped = COGROUP
  impressions BY (user_id, item_id),
  accepts BY (user_id, item_id),
  rejects BY (user_id, item_id);
-- Then count
features_counted = FOREACH features_grouped GENERATE
  FLATTEN(group) as (user_id, item_id),
  COUNT_STAR(impressions) as impression_count,
  COUNT_STAR(accepts) as accept_count,
  COUNT_STAR(rejects) as reject_count;
-- Then group again
features = FOREACH (GROUP features_counted BY user_id) GENERATE
  group as user_id,
  features_counted.(item_id, impression_count, accept_count, reject_count)
    as items;
```

Counting Events

- But it seems wasteful to have to group twice
- Even big data can get reasonably small once you start slicing and dicing it
- Want to consider one user at a time – that should be small enough to fit into memory

Counting Events

- Another approach: Only group once
- Bag manipulation UDFs to avoid the extra mapreduce job

```
DEFINE CountEach          datafu.pig.bags.CountEach('flatten');
DEFINE BagLeftOuterJoin   datafu.pig.bags.BagLeftOuterJoin();
DEFINE Coalesce           datafu.pig.util.Coalesce();
```

- CountEach – counts how many times a tuple appears in a bag
- BagLeftOuterJoin – performs a left outer join across multiple bags

Counting Events

A DataFu approach…

```
features_grouped = COGROUP impressions BY user_id, accepts BY user_id,
  rejects BY user_id;

features_counted = FOREACH features_grouped GENERATE
  group as user_id,
  CountEach(impressions.item_id) as impressions,
  CountEach(accepts.item_id) as accepts,
  CountEach(rejects.item_id) as rejects;

features_joined = FOREACH features_counted GENERATE
  user_id,
  BagLeftOuterJoin(
    impressions, 'item_id',
    accepts, 'item_id',
    rejects, 'item_id'
  ) as items;
```

Counting Events

- Revisit Coalesce to give default values

```
features = FOREACH features_joined {
  projected = FOREACH items GENERATE
    impressions::item_id as item_id,
    impressions::count as impression_count,
    Coalesce(accepts::count, 0) as accept_count,
    Coalesce(rejects::count, 0) as reject_count;
  GENERATE user_id, projected as items;
}
```

Sampling

- Suppose we only wanted to run our script on a sample of the previous input data

```
impressions = LOAD '$impressions' AS (user_id:int, item_id:int,
   item_category:int, timestamp:long);
accepts = LOAD '$accepts' AS (user_id:int, item_id:int, timestamp:long);
rejects = LOAD '$rejects' AS (user_id:int, item_id:int, timestamp:long);
```

- We have a problem, because the cogroup is only going to work if we have the same key (user_id) in each relation

Sampling

- DataFu provides SampleByKey

```
DEFINE SampleByKey datafu.pig.sampling.SampleByKey('a_salt','0.01');

impressions = FILTER impressions BY SampleByKey('user_id');
accepts = FILTER impressions BY SampleByKey('user_id');
rejects = FILTER rejects BY SampleByKey('user_id');
features = FILTER features BY SampleByKey('user_id');
```

Left outer joins

- Suppose we had three relations:

```
input1 = LOAD 'input1' using PigStorage(',') AS (key:INT,val:INT);
input2 = LOAD 'input2' using PigStorage(',') AS (key:INT,val:INT);
input3 = LOAD 'input3' using PigStorage(',') AS (key:INT,val:INT);
```

- And we wanted to do a left outer join on all three:

```
joined = JOIN input1 BY key LEFT,
              input2 BY key,
              input3 BY key;
```

- Unfortunately, this is not legal PigLatin

Left outer joins

- Instead, you need to join twice:

```
data1 = JOIN input1 BY key LEFT, input2 BY key;
data2 = JOIN data1 BY input1::key LEFT, input3 BY key;
```

- This approach requires two MapReduce jobs, making it inefficient, as well as inelegant

Left outer joins

- There is always cogroup:

```
data1 = COGROUP input1 BY key, input2 BY key, input3 BY key;
data2 = FOREACH data1 GENERATE
  FLATTEN(input1), -- left join on this
  FLATTEN((IsEmpty(input2) ? TOBAG(TOTUPLE((int)null,(int)null)) : input2))
      as (input2::key,input2::val),
  FLATTEN((IsEmpty(input3) ? TOBAG(TOTUPLE((int)null,(int)null)) : input3))
      as (input3::key,input3::val);
```

- But, it's cumbersome and error-prone

Left outer joins

- So, we have EmptyBagToNullFields

```
data1 = COGROUP input1 BY key, input2 BY key, input3 BY key;
data2 = FOREACH data1 GENERATE
  FLATTEN(input1), -- left join on this
  FLATTEN(EmptyBagToNullFields(input2)),
  FLATTEN(EmptyBagToNullFields(input3));
```

- Cleaner, easier to use

Left outer joins

- Can turn it into a macro

```
DEFINE left_outer_join(relation1, key1, relation2, key2, relation3, key3)
returns joined {
  cogrouped = COGROUP
    $relation1 BY $key1, $relation2 BY $key2, $relation3 BY $key3;
  $joined = FOREACH cogrouped GENERATE
    FLATTEN($relation1),
    FLATTEN(EmptyBagToNullFields($relation2)),
    FLATTEN(EmptyBagToNullFields($relation3));
}
```

```
features = left_outer_join(input1, val1, input2, val2, input3, val3);
```

Schema and aliases

- A common (bad) practice in Pig is to use positional notation to reference fields
- Hard to maintain
    - Script is tightly coupled to order of fields in input
    - Inserting a field in the beginning breaks things downstream
- UDFs can have this same problem
    - Especially problematic because code is separated, so the dependency is not obvious

Schema and aliases

- Suppose we are calculating monthly mortgage payments for various interest rates

```
mortgage = load 'mortgage.csv' using PigStorage('|')
as (principal:double,
    num_payments:int,
    interest_rates: bag {tuple(interest_rate:double)});
```

Schema and aliases

- So, we write a UDF to compute the payments
- First, we need to get the input parameters:

```
@Override
public DataBag exec(Tuple input) throws IOException
{
  Double principal = (Double)input.get(0);
  Integer numPayments = (Integer)input.get(1);
  DataBag interestRates = (DataBag)input.get(2);

  // ...
```

Schema and aliases

- Then do some computation:

```
DataBag output = bagFactory.newDefaultBag();

for (Tuple interestTuple : interestRates) {
  Double interest = (Double)interestTuple.get(0);

  double monthlyPayment = computeMonthlyPayment(principal, numPayments,
                                                interest);

  output.add(tupleFactory.newTuple(monthlyPayment));
}
```

Schema and aliases

- The UDF then gets applied

```
payments = FOREACH mortgage GENERATE MortgagePayment($0,$1,$2);
```

- Or, a bit more understandably

```
payments = FOREACH mortgage GENERATE
  MortgagePayment(principal,num_payments,interest_rates);
```

Schema and aliases

- Later, the data is changes, and a field is prepended to tuples in the interest_rates bag

```
mortgage = load 'mortgage.csv' using PigStorage('|')
as (principal:double,
    num_payments:int,
    interest_rates: bag {tuple(wow_change:double,interest_rate:double)});
```

- The script happily continues to work, and the output data begins to flow downstream, causing serious errors, later

Schema and aliases

- Write the UDF to fetch arguments by name using the schema
- AliasableEvalFunc can help

```
Double principal = getDouble(input,"principal");
Integer numPayments = getInteger(input,"num_payments");
DataBag interestRates = getBag(input,"interest_rates");
```

```
for (Tuple interestTuple : interestRates) {
  Double interest = getDouble(interestTuple,
    getPrefixedAliasName("interest_rates", "interest_rate"));
  // compute monthly payment...
}
```

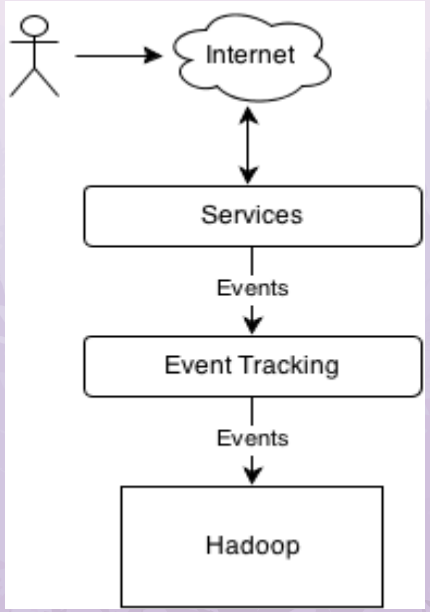
Other awesome things

- New and coming things
  - Functions for calculating entropy
  - OpenNLP wrappers
  - New and improved Sampling UDFs
  - Additional Bag UDFs
  - InHashSet
  - More…


APACHECON
DENVER
WESTIN DENVER DOWNTOWN
APRIL 7-9, 2014
DataFu - Hourglass
Presented For The Apache Foundation By
LINUX FOUNDATION

Event Collection

- Typically online websites have instrumented services that collect events
- Events stored in an offline system (such as Hadoop) for later analysis
- Using events, can build dashboards with metrics such as:
  - # of page views over last month
  - # of active users over last month
- Metrics derived from events can also be useful in recommendation pipelines
  - e.g. impression discounting

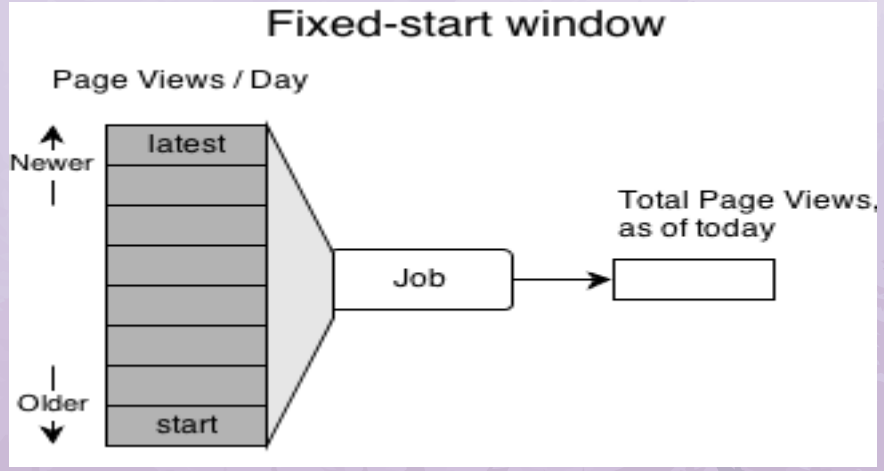


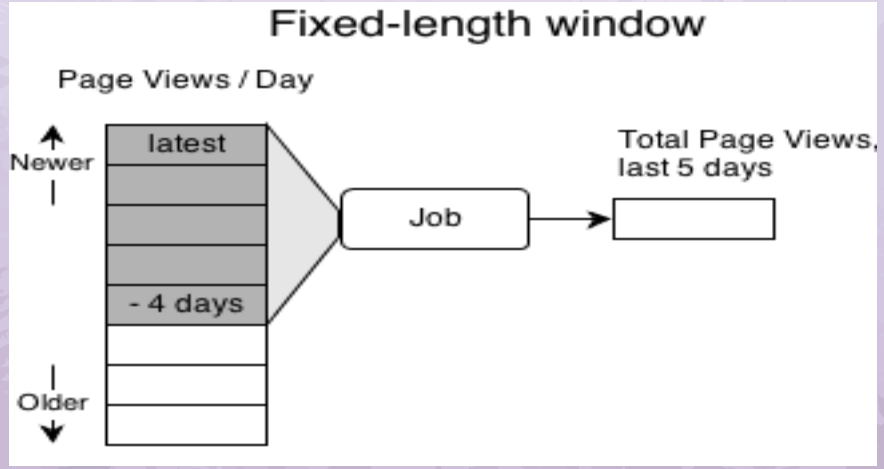
Event Storage

- Events can be categorized into topics, for example:
  - page view
  - user login
  - ad impression/click
- Store events by topic and by day:
  - /data/page_view/daily/2013/10/08
  - /data/page_view/daily/2013/10/09
- Hourglass allows you to perform computation over specific time windows of data stored in this format

Computation Over Time Windows

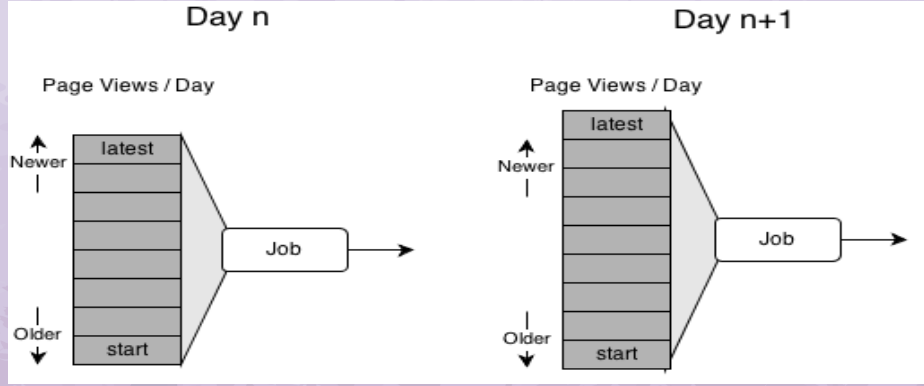
- In practice, many of computations over time windows use either:





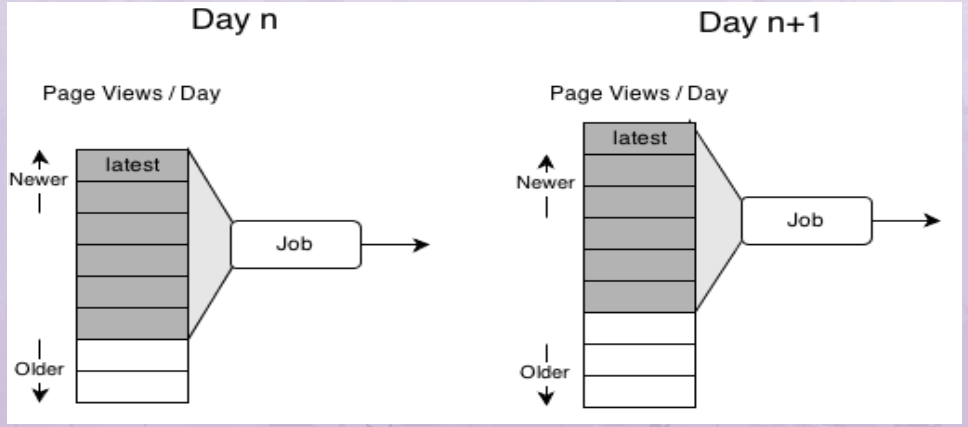
Recognizing Inefficiencies

- But, frequently jobs re-compute these daily
- From one day to next, input changes little
- Fixed-start window includes one new day:



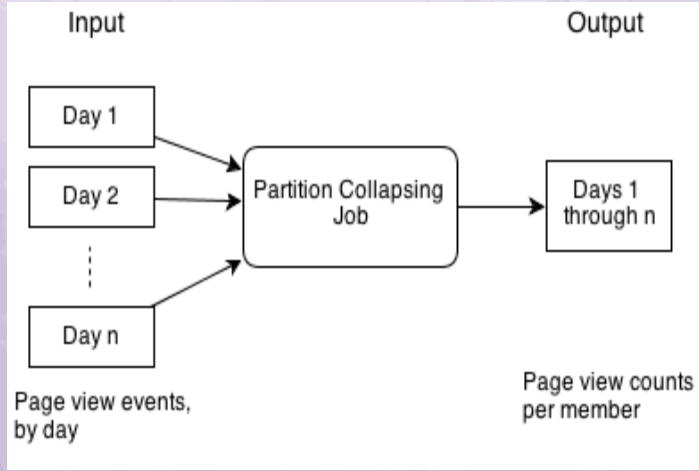
Recognizing Inefficiencies

- Fixed-length window includes one new day, minus oldest day



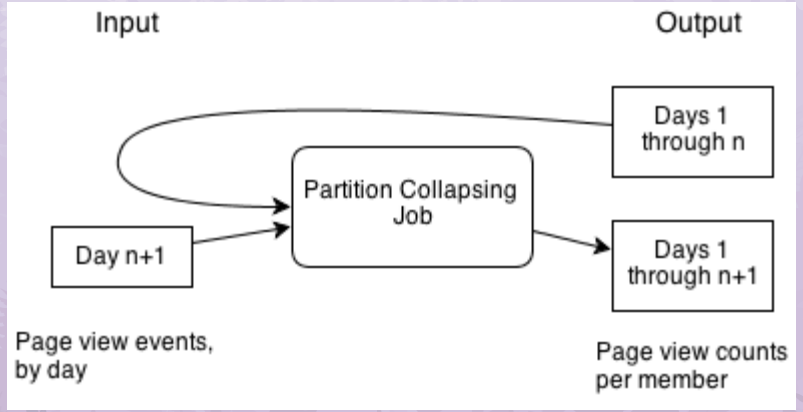
Improving Fixed-Start Computations

- Suppose we must compute page view counts per member
- The job consumes all days of available input, producing one output.
- We call this a *partition-collapsing* job.
- But, if the job runs tomorrow it has to reprocess the same data.



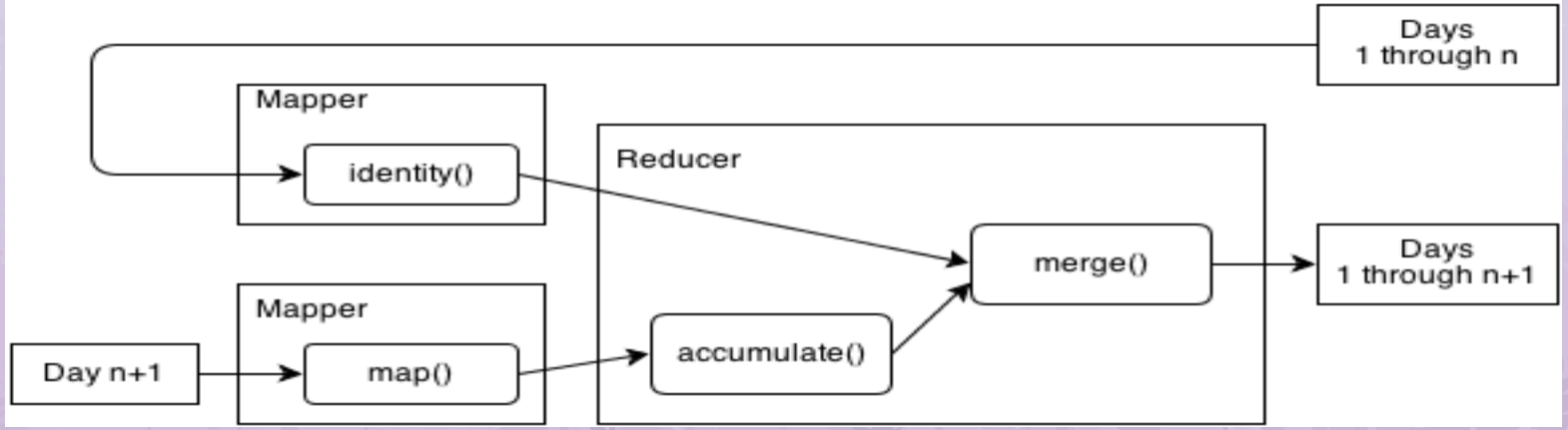
Improving Fixed-Start Computations

- Solution: Merge new data with previous output
- We can do this because this is an arithmetic operation
- Hourglass provides a partition-collapsing job that supports output reuse.



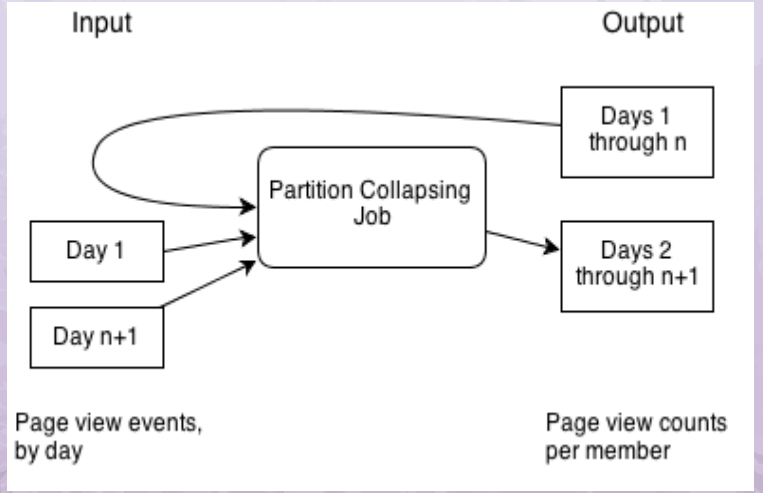
Partition-Collapsing Job Architecture (Fixed-Start)

- When applied to a fixed-start window computation:



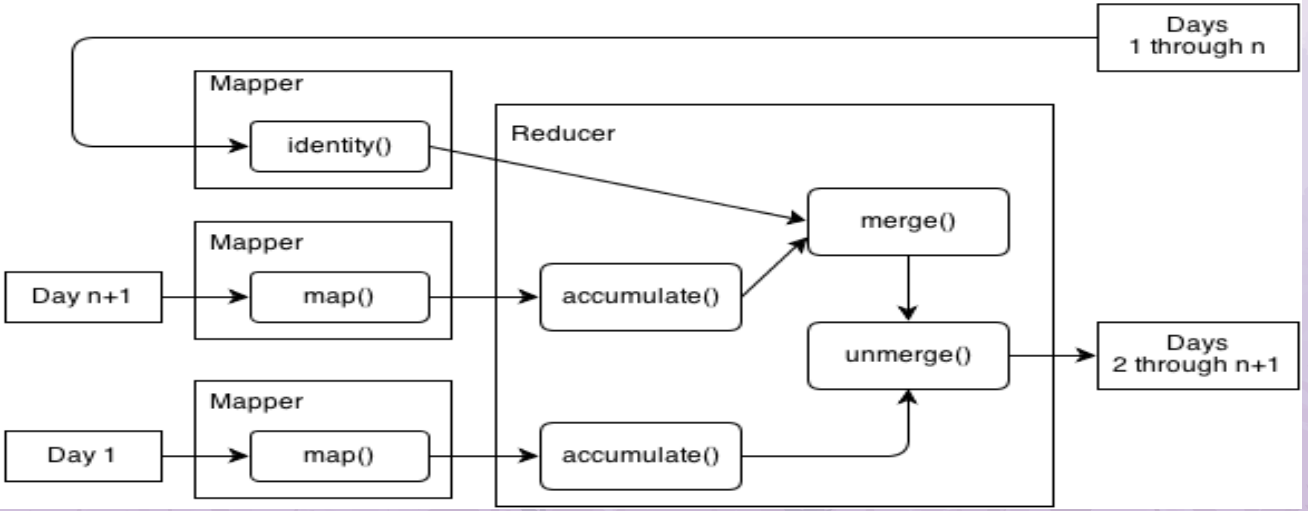
Improving Fixed-Length Computations

- For a fixed-length job, can reuse output using a similar trick:
  - Add new day to previous output
  - Subtract old day from result
- We can subtract the old day since this is arithmetic



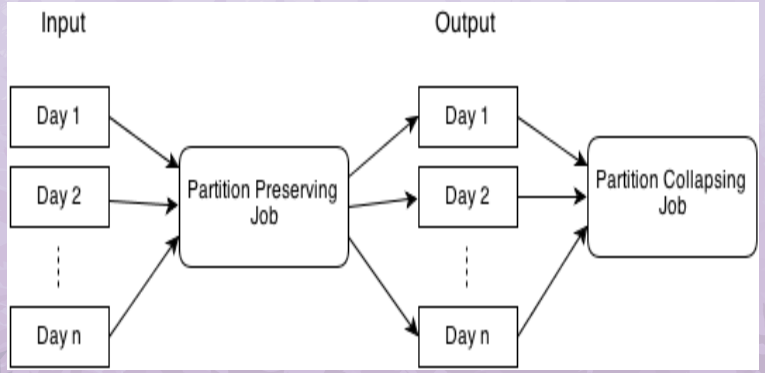
Partition-Collapsing Job Architecture (Fixed-Length)

- When applied to a fixed-length window computation:

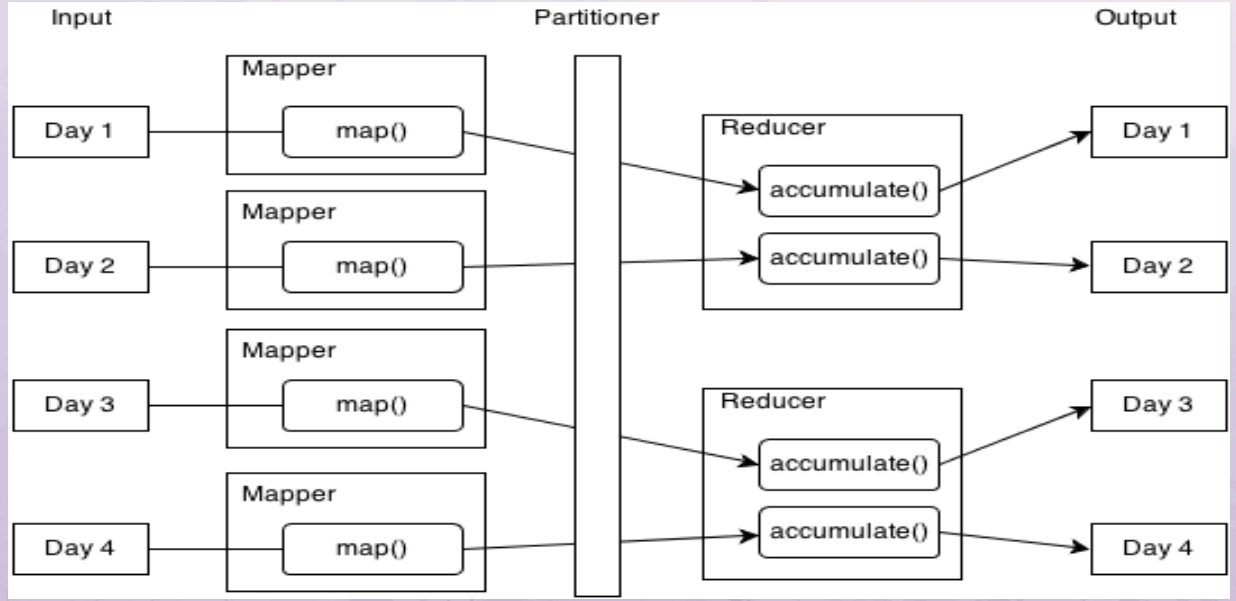


Improving Fixed-Length Computations

- But, for some operations, cannot subtract old data
  - example: max(), min()
- Cannot reuse previous output, so how to reduce computation?
- Solution: partition-preserving job
  - Partitioned input data,
  - partitioned output data
- aggregate the data in advance



Partition-Preserving Job Architecture



MapReduce in Hourglass

- MapReduce is a fairly general programming model
- Hourglass requires:
  - reduce() must output (key,value) pair
  - reduce() must produce at most one value
  - reduce() implemented by an accumulator
- Hourglass provides all the MapReduce boilerplate for you for these types of jobs

Summary

- Two types of jobs:
  - **Partition-preserving:** consume partitioned input data, produce partitioned output data
  - **Partition-collapsing:** consume partitioned input data, produce single output

Summary

- You provide:
  - Input: time range, input paths
  - Implement: map(), accumulate()
  - Optional: merge(), unmerge()
- Hourglass provides the rest to make it easier to implement jobs that incrementally process

Questions?

http://datafu.incubator.apache.org/