BIG DATA

Batch layer

INTRODUCTION

- In the last chapters you learned
  - how to form a data model for your dataset
  - how to store your data in the batch layer in a scalable way
- In this chapter
  - you'll learn how to compute arbitrary functions on that data.
  - We'll start by introducing some motivating examples to illustrate the concepts of computation on the batch layer.
  - Then you'll learn in detail how to compute indexes of the master dataset that the application layer will use to complete queries.
  - You'll examine the trade-offs between recomputation algorithms and incremental algorithms
  - You'll see what it means for the batch layer to be scalable
  - Then you'll learn about MapReduce

MOTIVATING EXAMPLES

- Number of pageviews over time

```
function pageviewsOverTime(masterDataset, url, startHour, endHour) {
    pageviews = 0
    for(record in masterDataset) {
        if(record.url == url &&
           record.time >= startHour &&
           record.time <= endHour) {
           pageviews += 1
           }
    }
    return pageviews
}
```

MOTIVATING EXAMPLES

- Gender inference

Normalizes all names associated with the person

Averages each name's probability of being male

Returns the gender with the highest likelihood

```
function genderInference(masterDataset, personId) {
    names = new Set()
    for(record in masterDataset) {
        if(record.personId == personId) {
            names.add(normalizeName(record.name))
        }
    }
    maleProbSum = 0.0
    for(name in names) {
        maleProbSum += maleProbabilityOfName(name)
    }
    maleProb = maleProbSum / names.size()
    if(maleProb > 0.5) {
        return "male"
    } else {
        return "female"
    }
}
```

MOTIVATING EXAMPLES

- Influence score

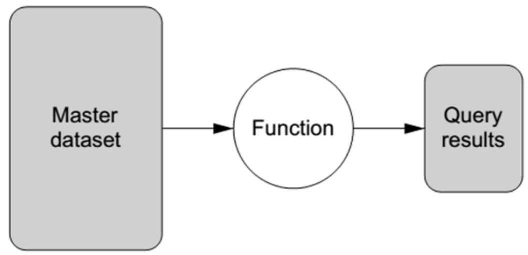
```
function influence_score(masterDataset, personId) {
    influence = new Map()
    for(record in masterDataset) {
        curr = influence.get(record.responderId) || new Map(default=0)
        curr[record.sourceId] += 1

       influence.set(record.sourceId, curr)
    }

    score = 0
    for(entry in influence) {
        if(topKey(entry.value) == personId) {
            score += 1
        }
    }
    return score
}
```

Computes amount of influence between all pairs of people

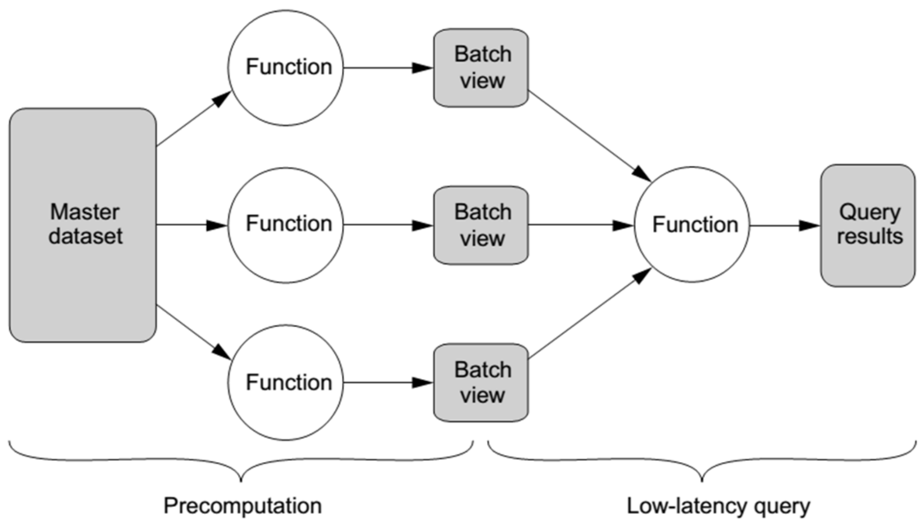
Counts the number of people for whom personId is the top influencer

COMPUTING ON THE BATCH LAYER

- A naive strategy



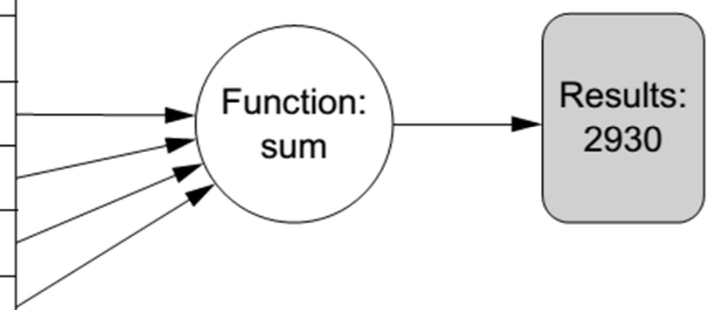
- Instead, you can precompute intermediate results



COMPUTING ON THE BATCH LAYER

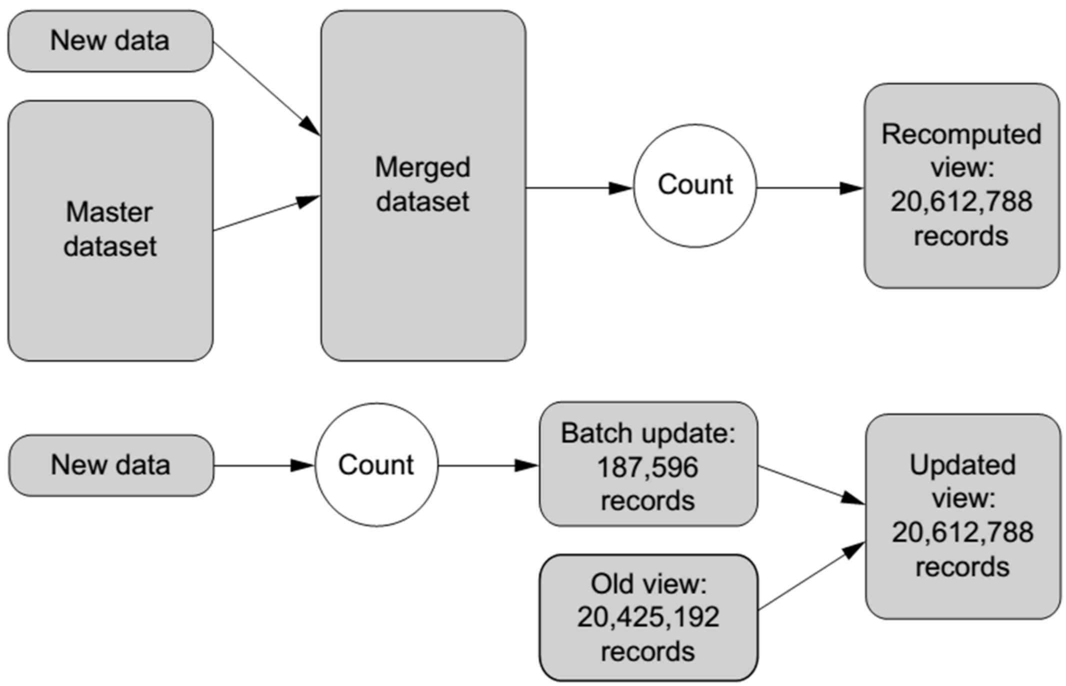
- Precompute intermediate results for pageviews example

URL	Hour	# Pageviews
foo.com/blog	2012/12/08 15:00	876
foo.com/blog	2012/12/08 16:00	987
foo.com/blog	2012/12/08 17:00	762
foo.com/blog	2012/12/08 18:00	413
foo.com/blog	2012/12/08 19:00	1098
foo.com/blog	2012/12/08 20:00	657
foo.com/blog	2012/12/08 21:00	101



COMPUTING ON THE BATCH LAYER

- Recomputation algorithms vs. incremental algorithms



key trade-offs between the two approaches are performance, human-fault tolerance, and the generality of the algorithm

COMPUTING ON THE BATCH LAYER

- Recomputation algorithms vs. incremental algorithms
  - Performance has two aspects
    - the amount of resources required to update a batch view with new data
      - incremental algorithm almost always uses significantly less resources
    - the size of the batch views produced
      - the size of the batch view for incremental algorithm can be significantly larger

URL	# Unique visitors
foo.com	2217
foo.com/blog	1899
foo.com/about	524
foo.com/careers	413
foo.com/faq	1212
...	...

Recomputation batch view

URL	# Unique visitors	Visitor IDs
foo.com	2217	1,4,5,7,10,12,14,....
foo.com/blog	1899	2,3,5,17,22,23,27,...
foo.com/about	524	3,6,7,19,24,42,51,...
foo.com/careers	413	12,17,19,29,40,42,...
foo.com/faq	1212	8,10,21,37,39,46,55,...
...	...	...

Incremental batch view

Figure 6.7 **A comparison between a recomputation view and an incremental view for determining the number of unique visitors per URL**

COMPUTING ON THE BATCH LAYER

- Recomputation algorithms vs. incremental algorithms
  - Human-fault tolerance
    - recomputation algorithms are inherently human-fault tolerant
    - human mistakes can cause serious problems in incremental algorithms
  - Generality of the algorithms
    - incremental algorithm can generate prohibitively large batch views
      - storage cost can be reduced at the price of making the algorithm approximate instead of exact
    - incremental algorithms shift complexity to on-the-fly computations
      - Example: improving semantic normalization in gender inference example

COMPUTING ON THE BATCH LAYER

- Recomputation algorithms vs. incremental algorithms
  - Choosing a style of algorithm

	Recomputation algorithms	Incremental algorithms
Performance	Requires computational effort to process the entire master dataset	Requires less computational resources but may generate much larger batch views
Human-fault tolerance	Extremely tolerant of human errors because the batch views are continually rebuilt	Doesn't facilitate repairing errors in the batch views; repairs are ad hoc and may require estimates
Generality	Complexity of the algorithm is addressed during precomputation, resulting in simple batch views and low-latency, on-the-fly processing	Requires special tailoring; may shift complexity to on-the-fly query processing
Conclusion	Essential to supporting a robust data-processing system	Can increase the efficiency of your system, but only as a supplement to recomputation algorithms

SCALABILITY IN THE BATCH LAYER

- Scalability definition:
  - the ability of a system to maintain performance under increased load by adding more resources
- More important scalability is linear scalability
  - maintaining performance by adding resources in proportion to the increased load
- MapReduce is linearly scalable
  - should the size of your master dataset double, then twice the number of servers will be able to build the batch views with the same latency

MAPREDUCE: A PARADIGM FOR BIG DATA COMPUTING

- Expresses computations in terms of map and reduce functions that manipulate key/value pairs
- The canonical example: word count

```
function word_count_map(sentence) {
  for(word in sentence.split(" ")) {
      emit(word, 1)
  }
}
```

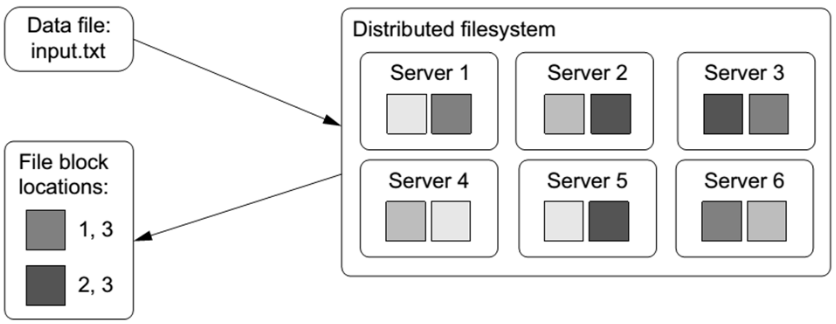
```
function word_count_reduce(word, values) {
  sum = 0
  for(val in values) {
      sum += val
  }
  emit(word, sum)
}
```

MAPREDUCE: A PARADIGM FOR BIG DATA COMPUTING

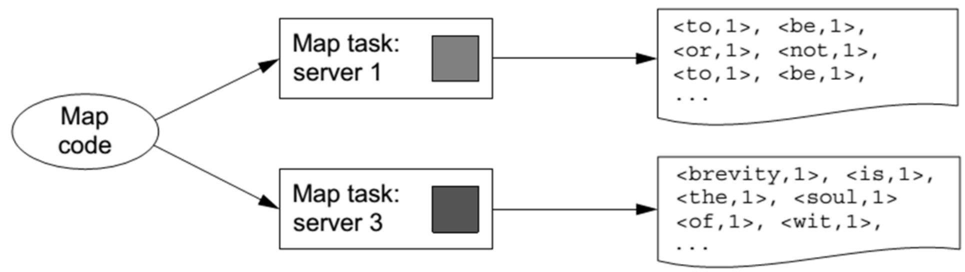
- Scalability
  - programs written in terms of MapReduce are inherently scalable
  - MapReduce automatically parallelizes the computation across a cluster of machines regardless of input size.
    - All the details of concurrency, transferring data between machines, and execution planning are abstracted by the framework

MAPREDUCE: A PARADIGM FOR BIG DATA COMPUTING

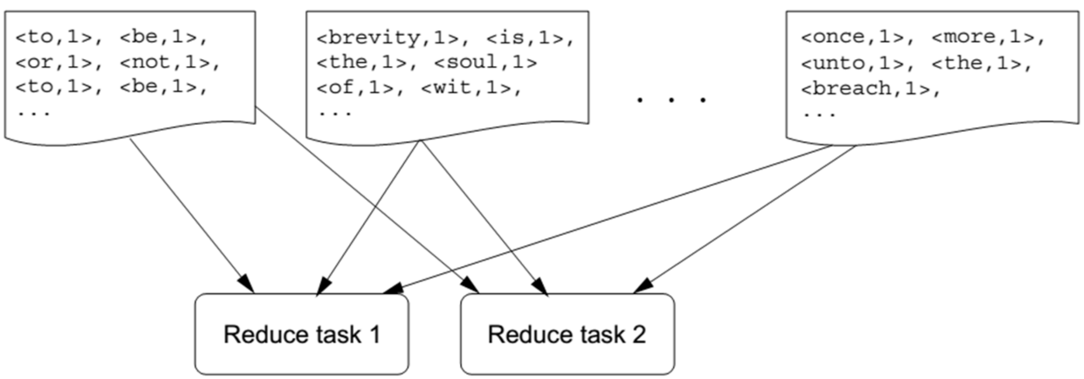
- Scalability



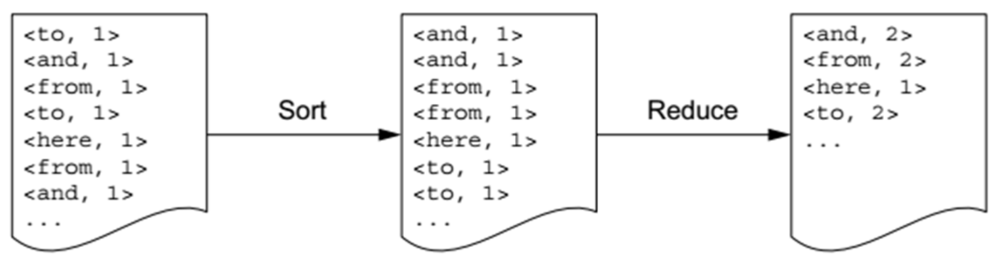
Before a MapReduce program begins processing data, it first determines the block locations within the distributed filesystem.



1. Code is sent to the servers hosting the input files to limit network traffic across the cluster.
2. The map tasks generate intermediate key/value pairs that will be redirected to reduce tasks.



During the shuffle phase, all of the key/value pairs generated by the map tasks are distributed among the reduce tasks. In this process, all of the pairs with the same key are sent to the same reducer.



MAPREDUCE: A PARADIGM FOR BIG DATA COMPUTING

- Fault-tolerance
    - MapReduce computations are also fault tolerant
    - MapReduce watches for errors and automatically retries that portion of the computation on another node
        - MapReduce requires that your map and reduce functions be deterministic
    - An entire application will fail only if a task fails more than a configured number of times—typically four

MAPREDUCE: A PARADIGM FOR BIG DATA COMPUTING

- Generality of MapReduce
  - MapReduce computational model is expressive enough to compute almost any functions on your data
  - Pageviews over time

```
function map(record) {
    key = [record.url, toHour(record.timestamp)]
    emit(key, 1)
}

function reduce(key, vals) {
    emit(new HourPageviews(key[0], key[1], sum(vals)))
}
```

MAPREDUCE: A PARADIGM FOR BIG DATA COMPUTING

- Generality of MapReduce
  - Gender inference

```
function map(record) {
    emit(record.userid, normalizeName(record.name))
}

function reduce(userid, vals) {
    allNames = new Set()
    for(normalizedName in vals) {
        allNames.add(normalizedName)
    }
    maleProbSum = 0.0
    for(name in allNames) {
        maleProbSum += maleProbabilityOfName(name)
    }
    maleProb = maleProbSum / allNames.size()
    if(maleProb > 0.5) {
        gender = "male"
    } else {
        gender = "female"
    }
    emit(new InferredGender(userid, gender))
}
```

Semantic normalization occurs during the mapping stage.

A set is used to remove any potential duplicates.

Averages the probabilities of being male.

Returns the most likely gender.

MAPREDUCE: A PARADIGM FOR BIG DATA COMPUTING

- Generality of MapReduce
  - Influence score

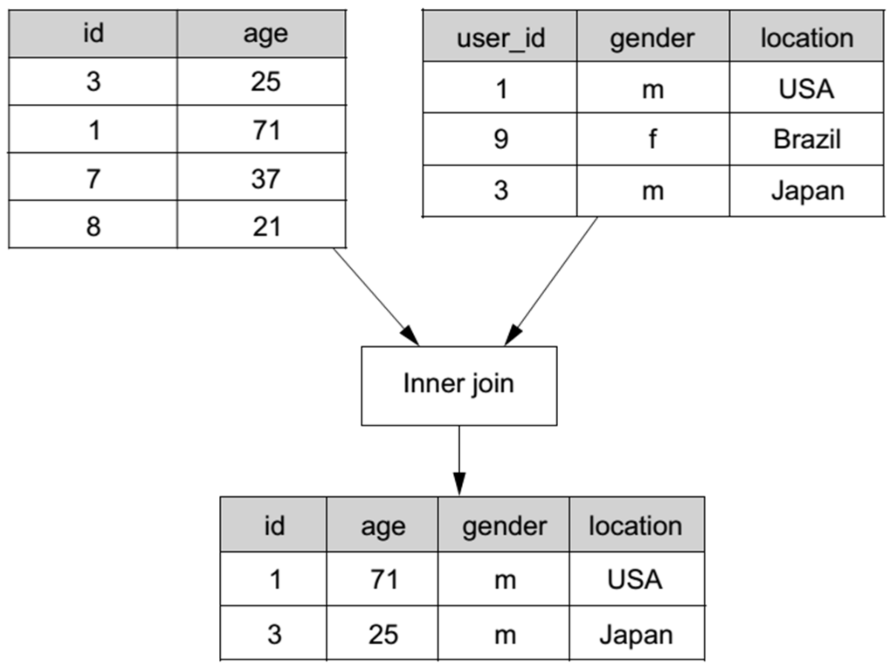
```
function map1(record) {
    emit(record.responderId, record.sourceId)
}

function reduce1(userid, sourceIds) {
    influence = new Map(default=0)
    for(sourceId in sourceIds) {
        influence[sourceId] += 1
    }
    emit(topKey(influence))
}

function map2(record) {
    emit(record, 1)
}

function reduce2(influencer, vals) {
    emit(new InfluenceScore(influencer, sum(vals)))
}
```

The first job determines the top influencer for each user.

The top influencer data is then used to determine the number of people each user influences.

MAPREDUCE: A PARADIGM FOR BIG DATA COMPUTING

- Low-level nature of MapReduce
  - Multistep computations are unnatural
    - Intermediate output of chained MapReduce jobs should be manually stored and cleaned
  - Joins are very complicated to implement manually



MAPREDUCE: A PARADIGM FOR BIG DATA COMPUTING

- Low-level nature of MapReduce
  - Joins are very complicated to implement manually
    - you need to read two independent datasets in a single MapReduce job

Use the source directory the record came from to determine if the record is on the left or right side of the join.

Set the MapReduce key to be the id or user_id, respectively. This will cause all records of those ids on either side of the join to get to the same reduce invocation. If you were joining on multiple keys at once, you'd put a collection as the MapReduce key.

The values you care to put in the output record are put into a list here. Later they'll be concatenated with records from the other side of the join to produce the output.

```
function join_map(sourcedir, record) {
  if(sourcedir=="/data/age") {
    emit(record.id, {"side" = "l"
, "values" = [record.age]})
  } else {
    emit(record.user_id,
         {"side" = "r",
          "values" = [record.gender, record.location]})
  }
}

function join_reduce(id, records) {
  side_l = []
  side_r = []

  for(record : records) {
    values = record.get("values")
    if(record.get("side") == "l") {
      side_l.add(values)
    } else {
      side_r.add(values)
    }
  }

  for(l : side_l) {
    for(r : side_r) {
      emit(concat([id], l, r), null)
    }
  }
}
```

When reducing, first split records from either side of the join into "left" and "right" lists.

To achieve the semantics of joining, concatenate every record on each side of the join with every record on the other side of the join.

The id is added to the concatenated values to produce the final result. Note that because MapReduce always operates in terms of key/value pairs, in this case you emit the result as the key and set the value to null. You could also do it the other way around

Imagine joining on multiple fields, with five sides to the join, with some sides as outer joins and some as inner joins

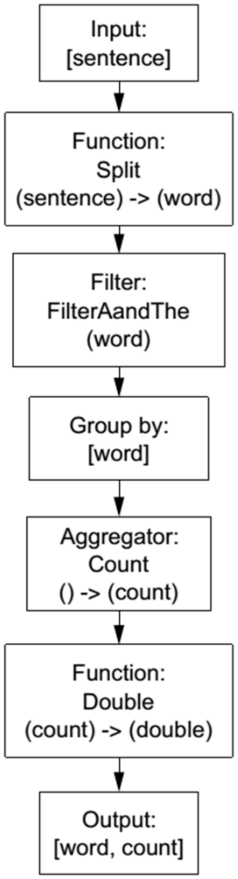
MAPREDUCE: A PARADIGM FOR BIG DATA COMPUTING

- Low-level nature of MapReduce
  - Logical and physical execution tightly coupled
    - Extended word-count example
      - Works, but it mixes together multiple tasks into the same function
      - Good programming practice involves separating independent functionality
      - Modularizing creates more MapReduce jobs, making the computation hugely inefficient

```
EXCLUDE_WORDS = Set("a", "the")

function map(sentence) {
  for(word : sentence) {
    if(not EXCLUDE_WORDS.contains(word)) {
      emit(word, 1)
    }
  }
}

function reduce(word, amounts) {
  result = 0
  for(amt : amounts) {
    result += amt
  }

  emit(result * 2)
}
```

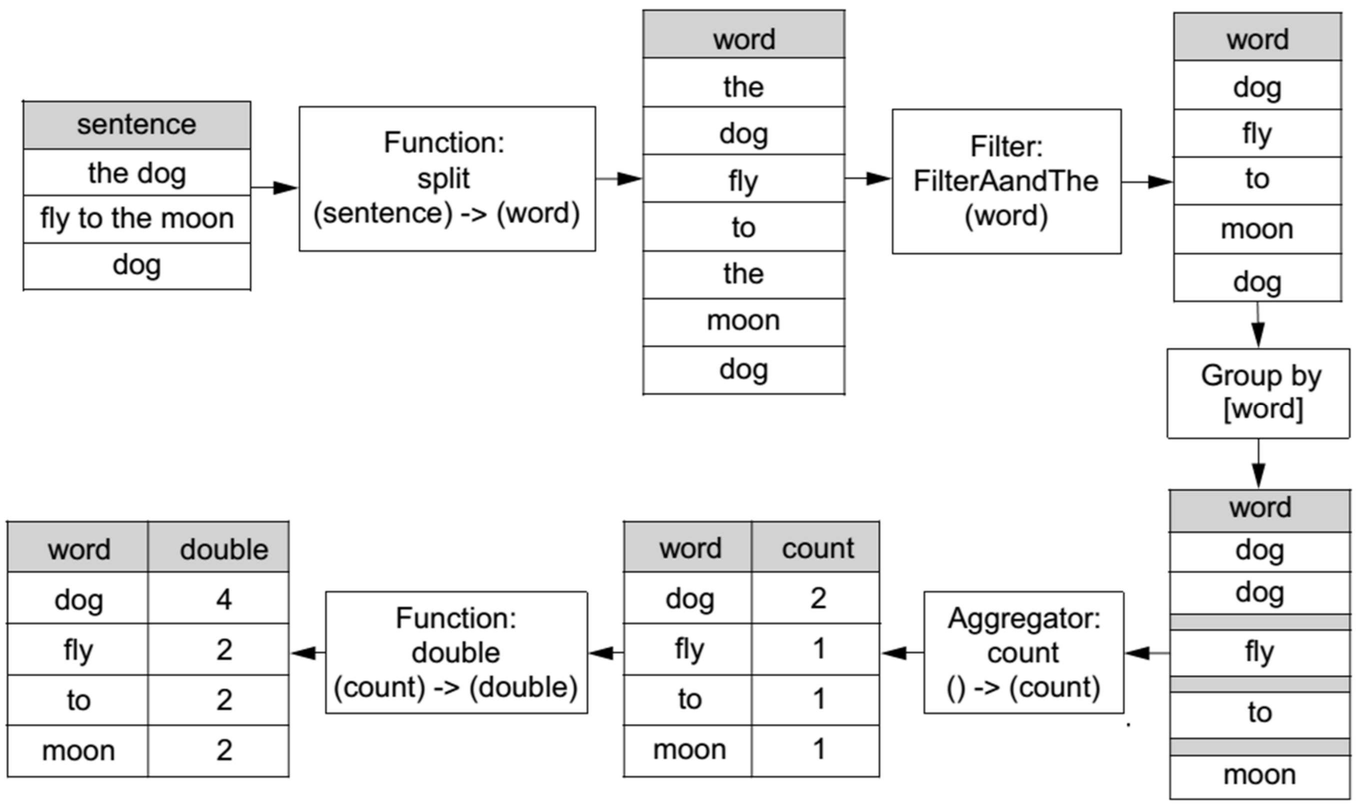
PIPE DIAGRAMS

- Concepts
  - The idea is to think of processing in terms of
    - Tuples
    - Functions
    - Filters
    - Aggregators
    - joins
    - merges



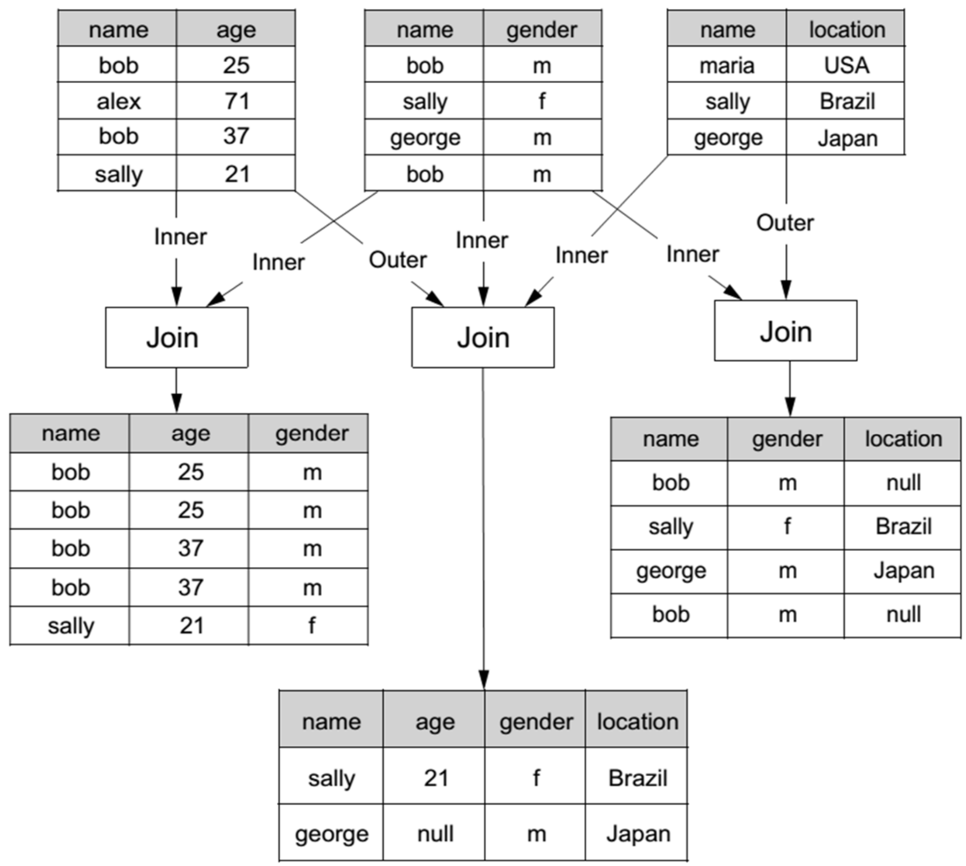
PIPE DIAGRAMS

- Concepts



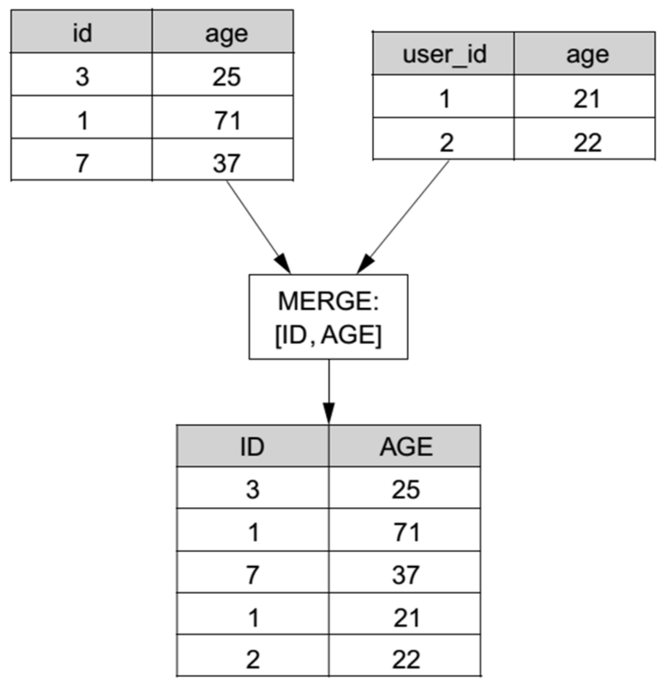
PIPE DIAGRAMS

- Concepts



PIPE DIAGRAMS

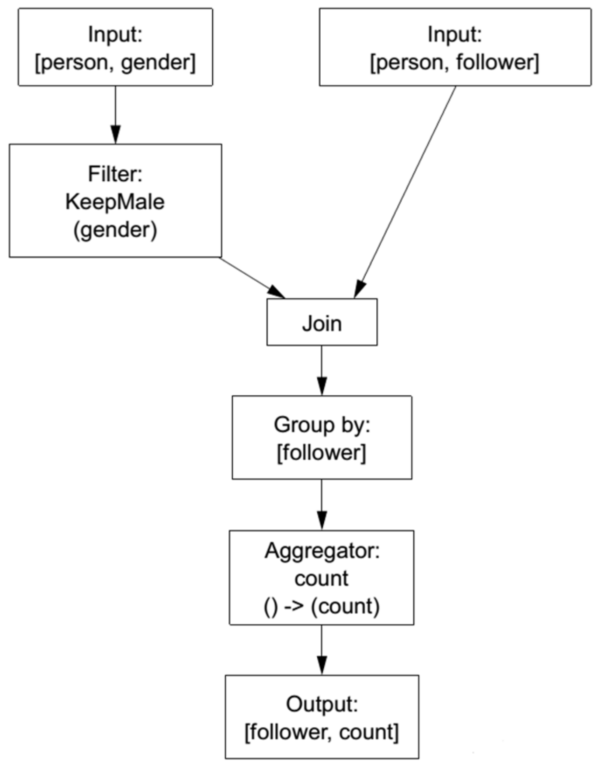
- Concepts



merge operation requires all tuple sets to have the same number of fields and specifies new names for the tuples

PIPE DIAGRAMS

- Concepts



compute the number of males each person follows

PIPE DIAGRAMS

- Executing pipe diagrams via MapReduce
  - Pipe diagrams can be compiled to a series of MapReduce jobs
    - Functions and filters
      - look at one record at a time
      - can be run either in a map step or in a reduce step following a join or aggregation
    - Group by
      - easily translated to MapReduce via the key emitted in the map step
    - Aggregators
      - looks at all tuples for a group
      - happens in the reduce step

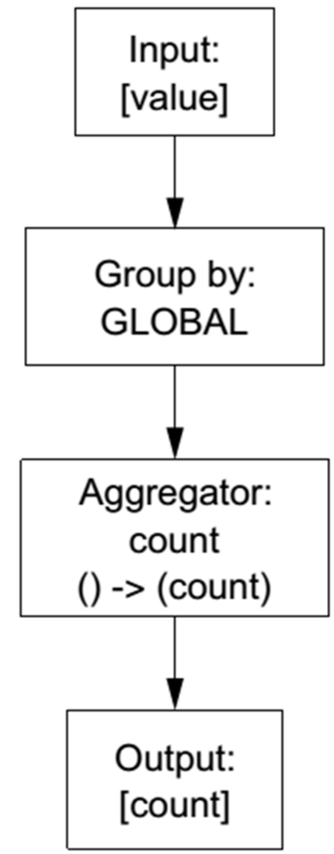
PIPE DIAGRAMS

- Executing pipe diagrams via MapReduce
  - Pipe diagrams can be compiled to a series of MapReduce jobs
    - Join
      - You've already seen the basics of implementing joins
      - require some code in the map step and some code in the reduce step
    - Merge
      - just means the same code will run on multiple sets of data

a smart compiler will pack as many operations into the same map or reduce step as possible

PIPE DIAGRAMS

- Combiner aggregators
  - example
    - compute the count of all the records
    - every tuple should go into the same group
    - the aggregator should run on every single tuple in dataset.
    - Normally
      - that every tuple would go to the same machine
      - then the aggregator code would run on that machine
    - This isn't scalable
    - can be executed a lot more efficiently
      - compute partial counts
      - send the partial counts to a single machine to produce global count

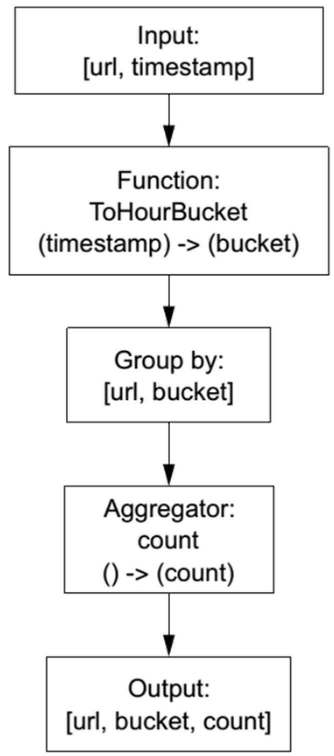


PIPE DIAGRAMS

- Combiner aggregators
    - All combiner aggregators work this way
        - doing a partial aggregation first
        - then combining the partial results to get the desired result.
    - Not every aggregator can be expressed this way
    - When it’s possible you get huge performance and scalability boosts

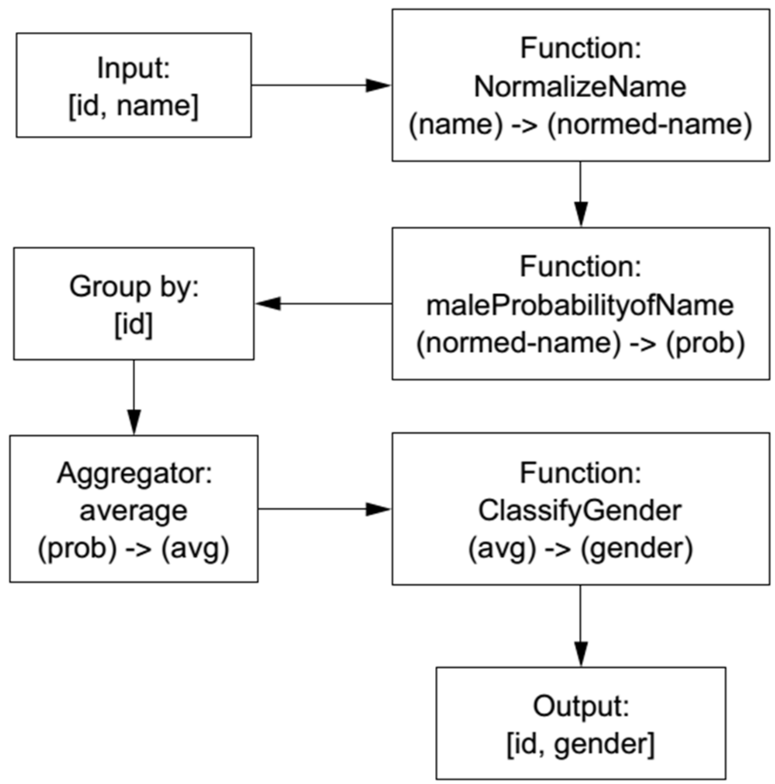
PIPE DIAGRAMS

- Examples
  - Pageviews over time



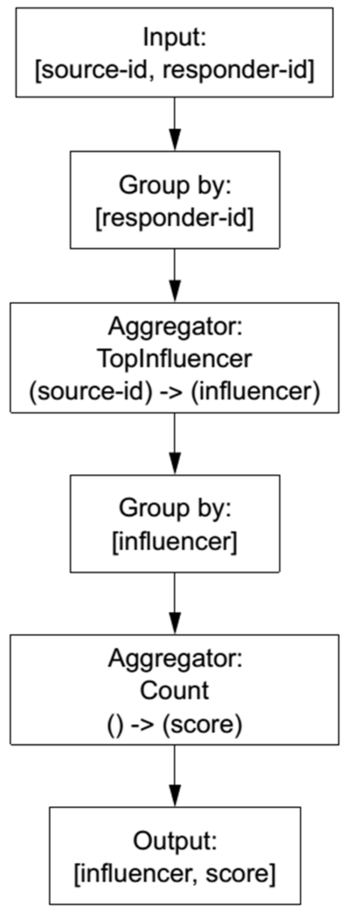
PIPE DIAGRAMS

- Examples
  - Gender inference



PIPE DIAGRAMS

- Examples
  - Influence score



ILLUSTRATION

- Jcascalog
  - a fairly direct mapping of pipe diagrams
  - enables a whole range of abstraction and composition techniques that just aren’t possible with other tools
  - enables programming techniques that allow you to write very concise, very elegant code

AN ILLUSTRATIVE EXAMPLE

- Word count:

```
List SENTENCE = Arrays.asList(
  Arrays.asList("Four score and seven years ago our fathers"),
  Arrays.asList("brought forth on this continent a new nation"),
  Arrays.asList("conceived in Liberty and dedicated to"),
  Arrays.asList("the proposition that all men are created equal"),
  ...
```

```
Api.execute(new StdoutTap(),
            new Subquery("?word", "?count")
              .predicate(SENTENCE, "?sentence")
              .predicate(new Split(), "?sentence").out("?word")
              .predicate(new Count(), "?count"));
```

Queries output to be written to the console

Specifies the output types returned by the query

Reads each sentence from the input

Tokenizes each sentence into separate words

Determines the count for each word

```
public static class Split extends CascalogFunction {
  public void operate(FlowProcess process, FunctionCall call) {
    String sentence = call.getArguments().getString(0);
    for (String word: sentence.split(" ")) {
      call.getOutputCollector().add(new Tuple(word));
    }
  }
}
```

Partitions a sentence into words

Emits each word in its own tuple

COMMON PITFALLS OF DATA-PROCESSING TOOLS

- Complexity in code
  - Essential complexity
  - Accidental complexity
    - Minimize this to have code that easier to maintain
    - Two sources:
      - Custom languages
      - Poorly composable abstractions

AN INTRODUCTION TO JCASCALOG

- The JCascalog data model
  - the same as that of the pipe diagrams
  - manipulates and transforms tuples
    - A set of tuples shares a schema
  - When executing a query
    - represents the initial data as tuples
    - transforms input into other tuple sets at each stage
  - Punctuation:
    - ? for non-nullable ! For nullable !! for nullable in outer joisn
  - Examples dataset:

AGE	
?person	?age
"alice"	28
"bob"	33
"chris"	40
"david"	25

GENDER	
?person	?gender
"alice"	"f"
"bob"	"m"
"chris"	"m"
"emily"	"f"

FOLLOWS	
?person	?follows
"alice"	“david"
"alice"	“bob"
"bob"	"david"
"emily"	"gary"

INTEGER
?num
-1
0
1
2

AN INTRODUCTION TO JCASCALOG

- The structure of a JCascalog query
  - Consist of
    - a destination tap
    - a subquery that defines the actual computation

The destination tap

The output fields

Predicates that define the desired output

```
Api.execute(new StdoutTap(),
            new Subquery("?person")
              .predicate(AGE, "?person", "?age")
              .predicate(new LT(), "?age", 30));
```

AN INTRODUCTION TO JCASCALOG

- The structure of a JCascalog query
  - predicates can be categorized into four main types:
    - Function predicate
      - specifies a relationship between a set of input fields and a set of output fields
    - Filter predicate
      - specifies a constraint on a set of input fields and removes all un matched tuples
    - Aggregator predicate
      - a function on a group of tuples
    - generator predicate
      - simply a finite set of tuples.
      - can either be
        - A source of data like an in-memory data structure or file on HDFS
        - Result from another subquery

AN INTRODUCTION TO JCASCALOG

- The structure of a JCascalog query
  - Predicate examples:

Type	Example	Description
Generator	`.predicate(SENTENCE, "?sentence")`	A generator that creates tuples from the `SENTENCE` dataset, with each tuple consisting of a single field called `?sentence`.
Function	`.predicate(new Multiply(), 2, "?x").out("?z")`	This function doubles the value of `?x` and stores the result as `?z`.
Filter	`.predicate(new LT(), "?y", 50)`	This filter removes all tuples unless the value of `?y` is less than 50.

AN INTRODUCTION TO JCASCALOG

- The structure of a JCascalog query
  - Predicates share a common structure
    - first argument is the predicate operation
    - remaining arguments are parameters for that operation
    - labels for the outputs are specified using the out method
  - provide extremely rich semantics

Type	Example	Description
Function as filter	`.predicate(new Plus(), 2, "?x").out(6)`	Although `Plus()` is a function, this predicate filters all tuples where the value of `?x` ≠ `4`.
Compound filter	`.predicate(new Multiply(), 2, "?a").out("?z")` `.predicate(new Multiply(), 3, "?b").out("?z")`	In concert, these predicates filter all tuples where `2(?a)` ≠ `3(?b)`.

AN INTRODUCTION TO JCASCALOG

- Querying multiple datasets
  - Joins are expressed explicitly in SQL
  - Joins in JCascalog are implicit based on the variable names

Language	Query	Description

SQL	`SELECT AGE.person, AGE.age, GENDER.gender`
	`FROM AGE`
	`INNER JOIN GENDER`
	`ON AGE.person = GENDER.person`
		This clause explicitly defines the join condition.

JCascalog	`new Subquery("?person", "?age", "?gender")`
	`   .predicate(AGE, "?person", "?age")`
	`   .predicate(GENDER, "?person", "?gender);`
		By specifying `?person` as a field name for both datasets, JCascalog does an implicit join using the shared name.

AN INTRODUCTION TO JCASCALOG

- Querying multiple datasets
  - Outer joins

<table>
<tr><th>Join type</th><th>Query</th><th>Results</th></tr>
<tr><td>Left outer join</td><td><pre>new Subquery("?person", "?age", "!!gender")
   .predicate(AGE, "?person", "?age")
   .predicate(GENDER, "?person", "!!gender");</pre></td><td>

?name	?age	?gender
"bob"	33	"m"
"chris"	40	"m"
"david"	25	null
"jim"	32	null

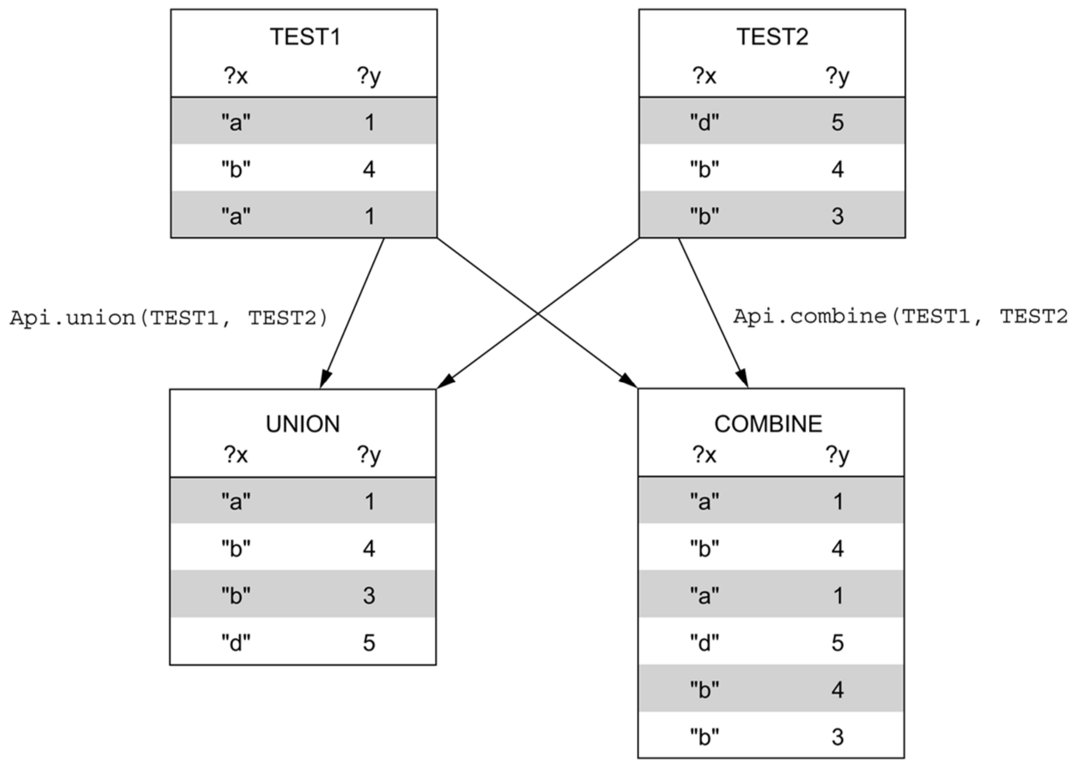
</td></tr>
<tr><td>Full outer join</td><td><pre>new Subquery("?person", "!!age", "!!gender")
   .predicate(AGE, "?person", "!!age")
   .predicate(GENDER, "?person", "!!gender");</pre></td><td>

?name	?age	?gender
"alice"	null	"f"
"bob"	33	"m"
"chris"	40	"m"
"david"	25	null
"emily"	null	"f"
"jim"	32	null

</td></tr>
</table>

AN INTRODUCTION TO JCASCALOG

- Querying multiple datasets
  - combine and union



AN INTRODUCTION TO JCASCALOG

- Grouping and aggregators
    - grouping is implicit based on the desired query output

The output field names define all potential groupings.

The underscore informs JCascalog to ignore this field.

```
new Subquery("?person", "?count")
  .predicate(FOLLOWS, "?person", "_")
  .predicate(new Count(), "?count");
```

When executing the aggregator, the output fields imply tuples should be grouped by ?person.

This query will group tuples by ?gender.

Before the aggregator, the AGE and GENDER datasets are joined.

```
new Subquery("?gender", "?count")
  .predicate(GENDER, "?person", "?gender")
  .predicate(AGE, "?person", "?age")
  .predicate(new LT(), "?age", 30)
  .predicate(new Count(), "?count");
```

Tuples are then filtered on ?age.

Even though the ?person and ?age fields were used in earlier predicates, they are discarded by the aggregator because they aren't included in the specified output.

AN INTRODUCTION TO JCASCALOG

- Stepping though an example query

Generators for the test datasets

Pre-aggregator function and filter

Multiple aggregators

Post-aggregator predicates

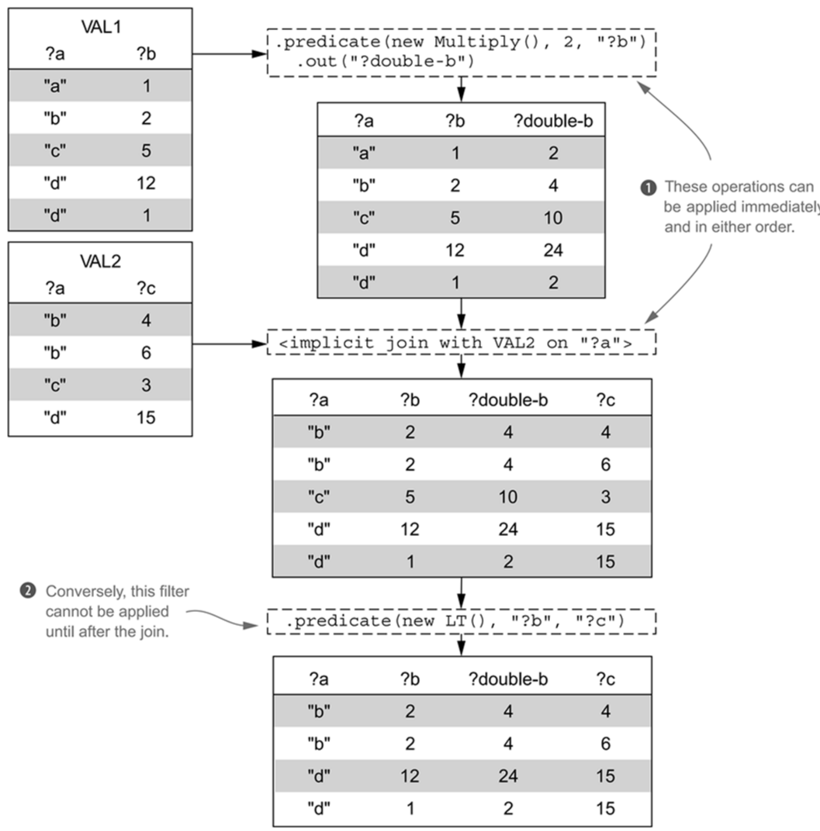
```
new Subquery("?a", "?avg")
  .predicate(VAL1, "?a", "?b")
  .predicate(VAL2, "?a", "?c")
  .predicate(new Multiply(), 2, "?b").out("?double-b")
  .predicate(new LT(), "?b", "?c")
  .predicate(new Count(), "?count")
  .predicate(new Sum(), "?double-b").out("?sum")
  .predicate(new Div(), "?sum", "?count").out("?avg")
  .predicate(new Multiply(), 2, "?avg").out("?double-avg")
  .predicate(new LT(), "?double-avg", 50);
```

VAL1	
?a	?b
"a"	1
"b"	2
"c"	5
"d"	12
"d"	1

VAL2	
?a	?c
"b"	4
"b"	6
"c"	3
"d"	15

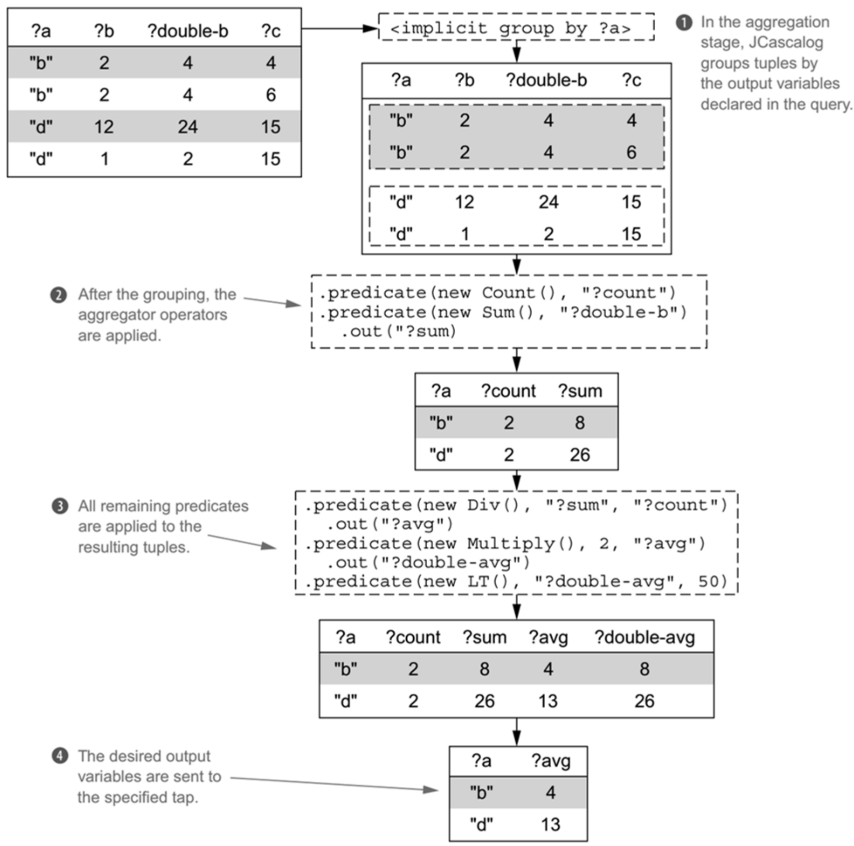
AN INTRODUCTION TO JCASCALOG

- Stepping though an example query



AN INTRODUCTION TO JCASCALOG

- Stepping though an example query



AN INTRODUCTION TO JCASCALOG

- Custom predicate operations
  - done by implementing the appropriate interfaces
  - FILTERS

```
public static class GreaterThanTenFilter extends CascalogFilter {
  public boolean isKeep(FlowProcess process, FilterCall call) {
    return call.getArguments().getInteger(0) > 10;
  }
}
```

Obtains the first element of the input tuple and treats the value as an integer

AN INTRODUCTION TO JCASCALOG

- Custom predicate operations
  - FUNCTIONS
    - emits zero or more tuples as output

Obtains the value from the input tuple

```
public static class IncrementFunction extends CascalogFunction {
  public void operate(FlowProcess process, FunctionCall call) {
    int v = call.getArguments().getInteger(0);
    call.getOutputCollector().add(new Tuple(v + 1));
  }
}
```

Emits a new tuple with the incremented value

?a	?b
"a"	1
"b"	4
"a"	1

```
.predicate(new IncrementFunction(), "?b")
  .out("?c")
```

?a	?b	?c
"a"	1	2
"b"	4	5
"a"	1	2

AN INTRODUCTION TO JCASCALOG

- Custom predicate operations
  - FUNCTIONS
    - can act as a filter if it emits zero tuples

Regards input value as a string

```
public static class TryParseInteger extends CascalogFunction {
  public void operate(FlowProcess process, FunctionCall call) {
    String s = call.getArguments().getString(0);
    try {
      int i = Integer.parseInt(s);
      call.getOutputCollector().add(new Tuple(i));
    }
    catch(NumberFormatException e) {}
  }
}
```

Emits value as integer if parsing succeeds

Emits nothing if parsing fails

?a	?b
"aaa"	1
"2"	4
"3"	1

```
.predicate(new TryParseInteger(), "?a")
  .out("?c")
```

?a	?b	?c
"2"	4	2
"3"	1	3

AN INTRODUCTION TO JCASCALOG

- Custom predicate operations
  - FUNCTIONS
    - each output tuple is appended to its own copy of the input arguments

```
public static class Split extends CascalogFunction {
  public void operate(FlowProcess process, FunctionCall call) {
    String sentence = call.getArguments().getString(0);
    for(String word: sentence.split(" ")) {
      call.getOutputCollector().add(new Tuple(word));
    }
  }
}
```

Emits each word as a separate tuple

For simplicity, splits into words using a single whitespace

?s
"the big dog"
"data"

```
.predicate(new Split(), "?s").out("?w")
```

?s	?w
"the big dog"	"the"
"the big dog"	"big"
"the big dog"	"dog"
"data"	"data"

AN INTRODUCTION TO JCASCALOG

- Custom predicate operations
  - AGGREGATORS : three different types
    - First: aggregator
      - looks at one tuple at a time for each tuple in a group
      - adjusts some internal state for each observed tuple
      - can be chained in a query
        - computing multiple aggregations at the same time for the same group

Initializes the aggregator internal state

Called for each tuple; updates the internal state to store the running sum

```
public static class SumAggregator extends CascalogAggregator {
  public void start(FlowProcess process, AggregatorCall call) {
    call.setContext(0);
  }

  public void aggregate(FlowProcess process, AggregatorCall call) {
    int total = (Integer) call.getContext();
    call.setContext(total + call.getArguments().getInteger(0));
  }

  public void complete(FlowProcess process, AggregatorCall call) {
    int total = (Integer) call.getContext();
    call.getOutputCollector().add(new Tuple(total));
  }
}
```

Once all tuples are processed, emits a tuple with the final result

AN INTRODUCTION TO JCASCALOG

- Custom predicate operations
  - AGGREGATORS : three different types
    - Second: buffer
      - receives an iterator to the entire set of tuples for a group
      - easier to write than aggregators
      - can not be chained in a query
      - can't be used along with any other aggregator type

```
public static class SumBuffer extends CascalogBuffer {
  public void operate(FlowProcess process, BufferCall call) {
    Iterator<TupleEntry> it = call.getArgumentsIterator();
    int total = 0;
    while(it.hasNext()) {
      TupleEntry t = it.next();
      total+=t.getInteger(0);
    }
    call.getOutputCollector().add(new Tuple(total));
  }
}
```

The tuple set is accessible via an iterator.

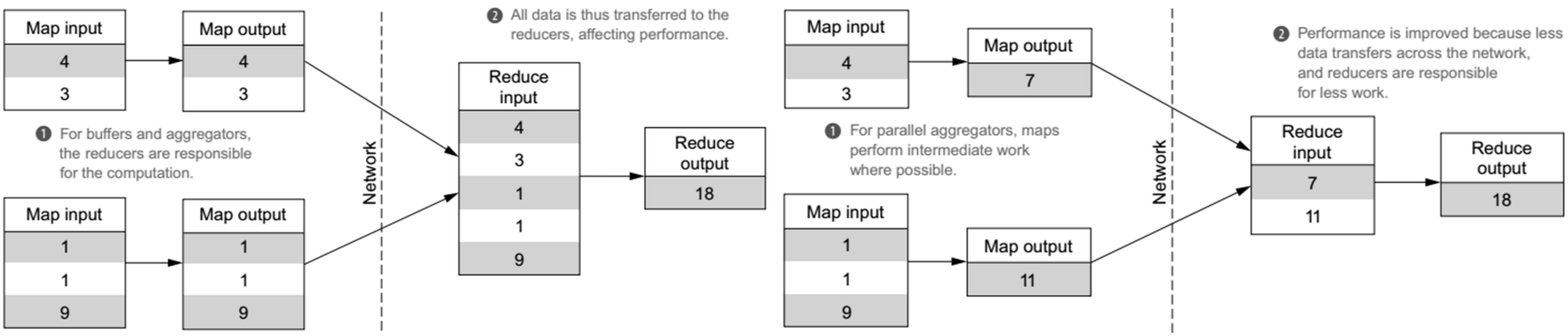
A single function iterates over all tuples and emits the output tuple.

AN INTRODUCTION TO JCASCALOG

- Custom predicate operations
  - AGGREGATORS : three different types
    - Third: parallel aggregators
      - analogous to combiner aggregators
      - performs an aggregation incrementally by doing partial aggregations in the map tasks

AN INTRODUCTION TO JCASCALOG

- Custom predicate operations
  - AGGREGATORS : three different types
    - Third: parallel aggregators



AN INTRODUCTION TO JCASCALOG

- Custom predicate operations
  - AGGREGATORS : three different types
    - Third: parallel aggregators
      - you must implement two functions:
        - init: maps the arguments from a single tuple to a partial aggregation for that tuple
        - combine: specifies how to combine two partial aggregations into a single aggregation value
      - can be chained with other parallel aggregators or regular aggregators
        - But act like regular aggregators when chaining with regular aggregators

For sum, the partial aggregation is just the value in the argument.

```
public static class SumParallel implements ParallelAgg {
  public void prepare(FlowProcess process, OperationCall call) {}

  public List<Object> init(List<Object> input) {
    return input;
  }

  public List<Object> combine(List<Object> input1,
    List<Object> input2) {
    int val1 = (Integer) input1.get(0);
    int val2 = (Integer) input2.get(0);
    return Arrays.asList((Object) (val1 + val2));
  }
}
```

To combine two partial aggregations, simply sum the values.

AN INTRODUCTION TO JCASCALOG

- Composition
  - Combining subqueries
    - they can be addressed as data sources for other subqueries

The first subquery determines all people that follow more than 2 others.

Considers only the follower, not the source

```
Subquery manyFollows = new Subquery("?person")
                           .predicate(FOLLOWS, "?person", "_")
                           .predicate(new Count(), "?count")
                           .predicate(new GT(), "?count", 2);

Api.execute(new StdoutTap(),
            new Subquery("?person1", "?person2")
              .predicate(manyFollows, "?person1")
              .predicate(manyFollows, "?person2")
              .predicate(FOLLOWS, "?person1", "?person2"));
```

Counts the number of people each user follows, and keeps those with count greater than 2

Uses the results of the first subquery as the source for this subquery

Only keeps records where follower and source are both present in the result of the first subquery

find all the records in FOLLOWS dataset where each person in the record follows more than two people

AN INTRODUCTION TO JCASCALOG

- Composition
  - Combining subqueries
    - Subqueries are lazy
      - nothing is computed until Api.execute is called

```
Subquery wordCount = new Subquery("?word", "?count")
                         .predicate(SENTENCE, "?sentence")
                         .predicate(new Split(), "?sentence").out("?word")
                         .predicate(new Count(), "?count");

Api.execute(new StdoutTap(),
            new Subquery("?count", "?num-words")
              .predicate(wordCount, "_", "?count")
              .predicate(new Count(), "?num-words"));
```

The basic word count subquery

The second subquery only requires the count for each word.

Determines the number of words for each count value

finding the number of words that exist for each computed word count

AN INTRODUCTION TO JCASCALOG

- Composition
  - Dynamically created subqueries

```
{"buyer": 123, "seller": 456, "amt": 50, "timestamp": 1322401523}
{"buyer": 1009, "seller": 12, "amt": 987, "timestamp": 1341401523}
{"buyer": 2, "seller": 98, "amt": 12, "timestamp": 1343401523}
```

A regular Java function dynamically generates the subquery.

Generates a tap from the provided HDFS path

Calls the custom JSON parsing function

```
public static Subquery parseTransactionData(String path) {
  return new Subquery("?buyer", "?seller", "?amt", "?timestamp")
    .predicate(Api.hfsTextline(path), "?line")
    .predicate(new ParseTransactionRecord(), "?line")
    .out("?buyer", "?seller", "?amt", "?timestamp");
}
```

The subquery needs a Cascalog function to perform the actual parsing.

An external library converts the JSON to a map.

The desired map values are translated into a single tuple.

```
public static class ParseTransactionRecord extends CascalogFunction {
  public void operate(FlowProcess process, FunctionCall call) {
    String line = call.getArguments().getString(0);
    Map parsed = (Map) JSONValue.parse(line);
    call.getOutputCollector().add(new Tuple(parsed.get("buyer"),
                                            parsed.get("seller"),
                                            parsed.get("amt"),
                                            parsed.get("timestamp")));
  }
}
```

```
public static Subquery buyerNumTransactions(String path) {
  return new Subquery("?buyer", "?count")
    .predicate(parseTransactionData(path), "?buyer", "_", "_", "_")
    .predicate(new Count(), "?count");
}
```

Disregards all fields but the buyer

AN INTRODUCTION TO JCASCALOG

- Composition
  - Dynamically created subqueries
    - Dynamic predicates in sub-query
    - find all chains of retweets of a certain length

```
public static Subquery chainsLength3(Object pairs) {
  return new Subquery("?a", "?b", "?c")
    .predicate(pairs, "?a", "?b")
    .predicate(pairs, "?b", "?c");
}
public static Subquery chainsLength4(Object pairs) {
  return new Subquery("?a", "?b", "?c", "?d")
    .predicate(pairs, "?a", "?b")
    .predicate(pairs, "?b", "?c")
    .predicate(pairs, "?c", "?d");
}
```

AN INTRODUCTION TO JCASCALOG

- Composition
  - Dynamically created subqueries
    - Dynamic predicates in sub-query
    - find all chains of retweets of a certain length

```
public static Subquery chainsLengthN(Object pairs, int n) {
  List<String> genVars = new ArrayList<String>();
  for(int i=0; i<n; i++) {
    genVars.add(Api.genNullableVar());
  }

  Subquery ret = new Subquery(genVars);
  for(int i=0; i<n-1; i++) {
    ret = ret.predicate(pairs, genVars.get(i), genVars.get(i+1));
  }
  return ret;
}
```

Generates unique nullable output variables

Loops to define the required number of joins

AN INTRODUCTION TO JCASCALOG

- Composition
  - Dynamically created subqueries
    - draw a random sample of N elements from a dataset of unknown size
      1. Generate a random number for every element.
      2. Find the N elements with the smallest random numbers.

```
public static Subquery fixedRandomSample(Object data, int n) {
  List<String> inputVars = new ArrayList<String>();
  List<String> outputVars = new ArrayList<String>();
  for(int i=0; i < Api.numOutFields(data); i++) {
    inputVars.add(Api.genNullableVar());
    outputVars.add(Api.genNullableVar());
  }

  String randVar = Api.genNullableVar();
  return new Subquery(outputVars)
    .predicate(data, inputVars)
    .predicate(new RandLong(), randVar)
    .predicate(Option.SORT, randVar)
    .predicate(new Limit(n), inputVars).out(outputVars);
}
```

Introspects the input dataset to determine the correct number of input and output fields

Generates separate fields for input and output variables

Creates a separate field to hold the random values

Uses the JCascalog RandLong function to append each input tuple with a random value

Performs secondary sorting on the random values

Uses the Limit aggregator to find *N* random tuples from the dataset

AN INTRODUCTION TO JCASCALOG

- Composition
  - Predicate macros
    - is an operation that JCascalog expands to another set of predicates
    - can create powerful abstractions by composing predicates together

```
PredicateMacroTemplate Average =
  PredicateMacroTemplate.build("?val")
    .out("?avg")
    .predicate(new Count(), "?count")
    .predicate(new Sum(), "?val").out("?sum")
    .predicate(new Div(), "?sum", "?count").out("?avg");
```

Uses a template to define a predicate macro with one input variable

The predicate macro returns a single output.

Average expands to three predicates: count, sum, and div.

Temp variables store the results of the aggregation.

Divides aggregate results to compute final output

AN INTRODUCTION TO JCASCALOG

- Composition
  - Predicate macros
    - is an operation that JCascalog expands to another set of predicates
    - can create powerful abstractions by composing predicates together

```
new Subquery("?result")
  .predicate(INTEGER, "?n")
  .predicate(Average, "?n).out("?result");
```

Example source code using the Average predicate macro.

↓

```
new Subquery("?result")
  .predicate(INTEGER, "?n")
  .predicate(new Count(), "?count_gen1")
  .predicate(new Sum(), "?n).out("?sum_gen2")
  .predicate(new Div(), "?sum_gen2", "?count_gen1")
  .out("?result");
```

Behind the scenes, JCascalog expands the macro into its constituent predicates using unique field names so as not to conflict with the surrounding subquery.

AN INTRODUCTION TO JCASCALOG

- Composition
  - Predicate macros
    - Compute the number of distinct values for a given set of variables
      - Templates only support fixed sets of input and output variables
    - Macros with flexible number of input and output variables:

```
public static class DistinctCountAgg extends CascalogAggregator {
  static class State {
    int count = 0;
    Tuple last = null;
  }

  public void start(FlowProcess process, AggregatorCall call) {
    call.setContext(new State());
  }

  public void aggregate(FlowProcess process, AggregatorCall call) {
    State s = (State) call.getContext();
    Tuple t = call.getArguments().getTupleCopy();
    if(s.last==null || !s.last.equals(t)) {
      s.count++;
    }
    s.last = t;
  }

  public void complete(FlowProcess process, AggregatorCall call) {
    State s = (State) call.getContext();
    call.getOutputCollector().add(new Tuple(s.count));
  }
}
```

Internal state to track the current count and the previously seen tuple

For each group, initializes the tracking state

When processing a tuple, retrieves the current state

Increases the distinct count only if the current tuple differs from the previous one

Always updates the last seen tuple in the state

When all tuples of the group have been processed, emits the distinct count

AN INTRODUCTION TO JCASCALOG

- Composition
  - Predicate macros
    - Compute the number of distinct values for a given set of variables

```
public static Subquery distinctCountManual() {
  return new Subquery("?distinct-followers-count")
    .predicate(FOLLOWS, "?person", "_")
    .predicate(Option.SORT, "?person")
    .predicate(new DistinctCountAgg(), "?person")
    .out("?distinct-followers-count");
```

Sorts the tuple by ?person field

The input and output fields are determined when the macro is used within a subquery.

```
public static class DistinctCount implements PredicateMacro {
  public List<Predicate> getPredicates(Fields inFields,
                                       Fields outFields) {
    List<Predicate> ret = new ArrayList<Predicate>();
    ret.add(new Predicate(Option.SORT, inFields));
    ret.add(new Predicate(new DistinctCountAgg(),
                          inFields,
                          outFields));
    return ret;
  }
}
```

Groups are sorted by the provided input fields.

For this macro, the distinct count emits a single field, but the general macro form supports multiple outputs.

AN INTRODUCTION TO JCASCALOG

- Composition
  - Dynamically created predicate macros

```
new Subquery("?x", "?y", "?z")
  .predicate(TRIPLETS, "?a", "?b", "?c")
  .predicate(new IncrementFunction(), "?a").out("?x")
  .predicate(new IncrementFunction(), "?b").out("?y")
  .predicate(new IncrementFunction(), "?c").out("?z");
```

Reads a dataset containing triples of numbers

Returns a new triplet where each field is incremented

it's a simple query, but there's considerable repetition

AN INTRODUCTION TO JCASCALOG

- Composition
  - Dynamically created predicate macros

```
new Subquery("?x", "?y", "?z")
  .predicate(TRIPLETS, "?a", "?b", "?c")
  .predicate(new Each(new IncrementFunction()), "?a", "?b", "?c")
  .out("?x", "?y", "?z");
```

```
public static class Each implements PredicateMacro {
  Object _op;

  public Each(Object op) {
    _op = op;
  }

  public List<Predicate> getPredicates(Fields inFields,
                                       Fields outFields) {
    List<Predicate> ret = new ArrayList<Predicate>();
    for(int i=0; i<inFields.size(); i++) {
      Object in = inFields.get(i);
      Object out = outFields.get(i);
      ret.add(new Predicate(_op,
                            Arrays.asList(in),
                            Arrays.asList(out)));
    }
    return ret;
  }
}
```

Each is parameterized with the predicate operation to use.

The predicate macro creates a predicate for each given input/output field pair.