Map Reduce Programming Guide

Fabric Engine Version 1.2.0-beta

Table of Contents

Chapter 1. Introduction

Fabric has traditionally provided access to parallelism through use of of its dependency graph to structure computation and data. However, the dependency graph has several shortcomings, especially when used for server-side computation:

- Constructing dependency graphs takes a lot of code. Even the simplest dependency graph requires the creation and connection of several different objects.
- Since the data in the dependency graph is persistent, it tends to incur a heavy memory overhead.
- Dependency graphs cannot be created or modified from within KL itself.

To solve these problems, Fabric provides a separate model of parallel programming called "Map-Reduce" that is inspired by models of parallelism provided by functional programming languages and commercial MapReduce frameworks for large-scale computing. The map-reduce functionality is available both within KL and from the host language such as JavaScript or Python. It is simple to use, provides highly-parallel performance and incurs a minimal memory overhead.

1.1. Background

Traditionally, the map-reduce paradigm provides a simple way of performing parallel operations on large sets of data. The input to map-reduce is a large array of data whose elements are of a common type, and the output is a single value of another type. The output is produced from the input by performing the following steps:

- For each input element `Xi` in the input array `[x1,x2,...xn]`, the value `mi = map(xi)` is computed. This can be done for the input elements in parallel.
- The result `R = reduce([M1,M2,...,Mn])` is computed by combining the results of the map operations.

The canonical example of a map-reduce operation is to count the number of occurrences of a given word in a large set of documents (for simplicity, strings). Then the `map(Xi)` operation counts the number of occurrences of the word in the string `Xi`, and the `reduce([M1,M2,...,Mn])` operation simply sums the results of all the `Mi`.

There are several problems with map-reduce in its simplest form:

- The individual `Xi` themselves might be values that need to be computed and/or that take a lot of memory. As such, it makes sense to retrieve (or compute) the value `Xi` right before computing `map(Xi)` and then to immediately throw away the value of `Xi`.
- Instead of computing `reduce([M1,M2,...,Mn])` once all the `Mi` are computed, we use less memory and make better use of parallelism by accumulating the result. First we initialize `R` to a default value, and then as each `Mi` is computed we compute `R = reduce(R, Mi)`. Once this is done for all the `Mi`, we return the resulting `R`. We must use a mutex to guarantee that `reduce(R, Mi)` is only ever executed by one thread at a time.

Fabric addresses these issues as well as others to provide a more general framework for parallel computation inspired by the traditional map-reduce case.

1.2. Concepts

Map-reduce in Fabric has a more generic and powerful implementation that arises from a core set of concepts and operations. The traditional map-reduce case is then just a specific case of what can be done with the Fabric map-reduce framework. All of the concepts and operations are available both within KL programs and are accessible using the host language interface to the Fabric core.

1.2.1. Producers

The first concept is the notion of a *producer*. A producer is a first-class object whose methods can be used to produce scalar and vector values; these values can be any of the core types in Fabric, and registered (user-defined) types, or even more producers.

There are two classes of producers in Fabric, described below.

1.2.1.1. Value Producers

A *value producer* is a producer that can produce a scalar value; it has the method `produce()` that returns the value, as well as the method `flush()` that flushes any cached values (see the 'Caches' section below).

1.2.1.2. Array Producers

An *array producer* is a producer that can produce an array (vector) of elements; it has five methods:

- `getCount()`: returns the number of elements in the array
- `produce(index)`: returns the element of the array with index `index`
- `produce(startIndex, count)`: returns the subarray of `count` elements of the array starting with index `startIndex`
- `produce()`: returns the array of all elements
- `flush()`: flushes any cached values (see the 'Caches' section below).

The indices are zero-based, as in KL. All the the array elements have the same type; in a strong sense, an array producer is an object that can be used to populate a variable-length KL array.

Note that an array producer specifies how to produce the elements of an array without actually producing them. This means that you can create an array produce for an array with billions of elements and it takes no more memory than an array producer for one element; it's only when the elements are produced that the results may be stored, depending on how the results are used.

The `produce()` and `produce(startIndex, count)` methods produce the individual elements of the array in parallel.

1.2.2. Types of Producers

For each class (value or array) of producer, there are four basic types of producers that produce values in different ways.

1.2.2.1. Constant Producers

A constant producer has fixed values that are produced. It does not need to execute any code to produce its values, and the values it produces are specified when the constant producer is created.

1.2.2.2. Generators

A generator is a producer that calls a function to produce its value. In the case of an array producer, the function that is called can optionally receive the index within the array and the total number of elements of the array; these can be used to calculate the element to generate. Both value and array producers can optionally take a "shared value" which can be

used to pass things like shared parameters to the generator. The shared value is itself the result of calling `produce()` on a value producer, which means that it can itself potentially be a calculated value.

1.2.2.3. Maps

A map is a producer that takes as input a value of one type and produces from it a value of another, potentially different, type. An example of a simple map might be one that takes a string as input and produces the length of the string as output. As with generators, an array map can optionally take the index of the element being produced as well as the count of the array, and both value and array maps can take a shared value.

1.2.2.4. Transforms

A transform is a producer that modifies the value of another producer. The same behaviour could be accomplished using a map that uses the same type for input and output, but using a transform instead will require less memory and generally result in slightly better runtime performance. An example of a simple transform might be one that normalizes a vector. As with generators and maps, an array transform can optionally take the index of the element being produced as well as the count of the array, and both value and array transforms can take a shared value.

1.2.3. Reduce

The reduce operation is the bridge between array producers and value producers. A reduce operation takes an array producer and a function as input and returns a value producer as output. The reduce operation works by calling the function for each element produced by the array producer; this function is then used to progressively produce the result of reduce operation as a value producer. There are two guarantees for the function:

- The function is called exactly once for each element of the input array producer; and
- The function is called by only one thread at a time so that no manual synchronization is necessary.

However, the order in which the elements of the array producer is undefined, and as such algorithms cannot depend on this order.

As with generators, maps and transforms, the reduce function can optionally take the index of element of the input array producer as well as its total count, and you can optionally pass a shared value.

A simple example of a reduce operation would be to sum an array of values. The array producer would produce the individual values, and the reduce function would simply add each value to the result.

1.2.4. Caches

A cache is a producer that simply caches the value of another producer. Caches are a simple solution for situations where the same results would be computed multiple times. As an example, if multiple producers all used the same shared value producer, it would probably make sense to put a value cache in front of the value producer so that it's not recomputed every time it's used.

All producers support a method called `flush()` that recursively flushes any caches. So, for example, if you have a reduce operation that uses a shared value that is cached, calling `flush()` on the reduce operation will flush the connected shared value cache.

1.2.5. Composing Producers and Reduce Operations

The power of Fabric's map-reduce model is found through the use of composition. Maps, transforms and reduce operations all take other producers as input and allow modification of the results. As well, generators, maps, transforms and

reduce operations all optionally take shared values that are the results of the `produce()` operation of a value producer; this value produce can in turn be a complex, composed operation such as a reduce operation on a map.

As an example, suppose you had a large set of documents and you wanted to count the occurrence of the longest word that occurs across all the documents; assume for the example that there is a unique longest word. You need to first figure out what this word is and then count it. The compositional model would be:

```
reduce(
  input: map(
    input: constArray(documents),
    function: countWord,
    sharedValue: reduce(
      input: map(
        input: constArray(documents),
        function: findLongestWord
        ),
      function: pickLongestWord
    ),
  function: sumValue
  )
```

Then the functions would look something like:

```
operator findLongestWord(String document, io String longestWord) {
  // Set longestWord to the longest word in document
}

operator pickLongestWord(String word, io String longestWord) {
  if (word.length > longestWord.length)
    longestWord = word;
}

operator countWord(String document, io Size count, String word) {
  // Set count to the number of occurrences of word in document
}

operator sumValue(Size count, io Size totalCount) {
  totalCount += count;
}
```


Chapter 2. KL Implementation

As mentioned previously, the map-reduce model is available both directly in KL and directly from JavaScript or Python. Using the map-reduce model directly from within KL allows for simple multithreading directly from within KL itself, without the need to create structures within the host language.

2.1. KL Constant Producers

In KL, a constant value producer is created using the `createConstValue(value)` call. The value parameter can be any typed KL r-value, eg. a constant value, the value of a variable, or the result of a function call. The result of the value is a value of type `ValueProducer<ValueType>`, where `ValueType` is the type of `value`. For example, the KL code:

```
operator entry() {
  ValueProducer<Scalar> vp = createConstValue(1.4142);
  report(vp.produce());
}
```

produces the result:

```
1.4142
```

A constant array producer is created using the `createConstArray(array)` function. Its single parameter array must be an expression that resolves to a fixed-length, variable-length or sliced array. The return value is of type `ArrayProducer<ElementType>`, where `ElementType` is the type of the array elements. For example, the KL code:

```
operator entry() {
  String a[];
  a.push('zero'); a.push('one'); a.push('two');
  ArrayProducer<String> ap = createConstArray(a);
  report(ap.produce());
}
```

produces the result:

```
["zero","one","two"]
```

2.2. KL Generators

A value generator in KL is created using the `createValueGenerator(functionName)`. `functionName` must be the name of a KL function available in the same KL module that has the signature operator `functionName(io ValueType value);` the result is a value of type `ValueProducer<ValueType>`. For example, the KL code:

```
operator gen(io String value[]) {
  value.push("Hello, world!");
}

operator entry() {
  ValueProducer<String[]>vp = createValueGenerator(gen);
  report(vp.produce());
```

```
}
```

produces the output:

```
["Hello, world!"]
```

An array generator in KL is created using the `createArrayGenerator(countValueProducer, functionName)`. The `countValueProducer` parameter must be a value producer that produces a value of type `Size`, ie. a value of type `ValueProducer<Size>`. If you are generating a fixed-size array of size 16, for instance, you can pass the result of `createConstValue(16)`, but you can also pass a more complex value producer such as a value generator. The functionName is the name of a function in the same module with one of the following prototypes:

- `operator functionName(io ElementType element)`
- `operator functionName(io ElementType element, Index index)`
- `operator functionName(io ElementType element, Index index, Size count)`

In cases where the index parameter is present, the index of the element within the array is passed to the generator function; similarly, when the count parameter is present the total number of elements in the array is passed. This can be useful for figuring out what value to generate.

An example of an array generator:

```
operator gen(
  io Scalar v,
  Index i,
  Size n
  )
{
  // Produces n uniform values on the interval [0,1], including 0 and 1 themselves
  v = Scalar(i) / Scalar(n-1);
}

operator entry() {
  ArrayProducer<Scalar> ap = createArrayGenerator(createConstValue(Size(10)), gen);
  report(ap.produce());
}
```

produces:

```
[0,0.1111111,0.2222222,0.3333333,0.4444444,0.5555556,0.6666667,0.7777778,0.8888889,1]
```

In both cases, you can optionally specify a shared value producer as the last parameter to the `create...Generator()` call. When a shared value producer is provided, the function receives an additional parameter whose type is the value type of the shared value producer. So, for value generators, the prototype of the operator becomes:

```
operator gen(io ValueType value, SharedType sharedValue) {
  ...
}
```

and for array generators the prototype of the operator becomes:

```
operator gen(io ElementType element, Index index, Size count, SharedType sharedValue) {

  ...
}
```


Note that when using a shared value with an array generator you must include the index and count parameters in the operator even if they are unused.

2.3. KL Maps

A value map in KL is created using the `createValueMap(inputValueProducer, functionName)`. `inputValueProducer` must be a value producer, and `functionName` must be the name of a KL function available in the same KL module that has the signature operator `functionName(InputType input, io OutputType output);` the type `InputType` must be the same as the value type of the input value producer. The result of the `createValueMap` call is a value of type `ValueProducer<OutputType>`. For example, the KL code:

```
operator map(String input, io Size output) {
  output = input.length;
}

operator entry() {
  ValueProducer<Size> vp = createValueMap(createConstValue("Hello, world!"), map);
  report(vp.produce());
}
```

produces the output:

```
13
```

An array map in KL is created using the `createArrayMap(inputArrayProducer, functionName)`. The `inputArrayProducer` parameter must be an array producer. Assuming that the element type of `inputArrayProducer` is `InputType`, `functionName` is the name of a function in the same module with one of the following prototypes:

- `operator functionName(InputType input, io OutputType output)`
- `operator functionName(InputType input, io OutputType output, Index index)`
- `operator functionName(InputType input, io OutputType output, Index index, Size count)`

In cases where the `index` parameter is present, the index of the element within the array is passed to the generator function; similarly, when the count parameter is present the total number of elements in the array is passed. The result of the `createArrayMap` call is a value of type `ArrayProducer<OutputType>`.

An example of an array map:

```
operator map(String input, io Size output) {
  output = input.length;
}

operator entry() {
  String a[]; a.push("one"); a.push("two"); a.push("three");
  ArrayProducer<String> iap = createConstArray(a);
  report(iap.produce());
  ArrayProducer<Size> oap = createArrayMap(iap, map);
  report(oap.produce());
}
```

produces:

```
["one","two","three"]
[3,3,5]
```

In both cases, you can optionally specify a shared value producer as the last parameter to the `create...Map()` call. When a shared value producer is provided, the function receives an additional parameter whose type is the value type of the shared value producer. So, for value maps, the prototype of the operator becomes:

```
operator gen(InputType input, io OutputType output, SharedType sharedValue) {
  ...
}
```

and for array maps the prototype of the operator becomes:

```
operator gen(InputType input, io OutputType output, Index index, Size count, SharedType
sharedValue) {
  ...
}
```

Note that when using a shared value with an array map you must include the `index` and `count` parameters in the operator even if they are unused.

2.4. KL Transforms

A value transform in KL is created using the `createValueTransform(inputValueProducer, functionName)`. `inputValueProducer` must be a value producer, and `functionName` must be the name of a KL function available in the same KL module that has the signature operator `functionName(io ValueType value)`; the type `ValueType` must be the same as the value type of the input value producer. The result of the `createValueMap` call is a value of type `ValueProducer<ValueType>`. For example, the KL code:

```
operator transform(io Scalar value) {
  value = sqrt(value);
}

operator entry() {
  ValueProducer<Scalar> vp = createValueTransform(createConstValue(Scalar(3.14)), transform);
  report(vp.produce());
}
```

produces the output:

```
1.772004
```

An array transform in KL is created using the `createArrayTransform(inputArrayProducer, functionName)`. The `inputArrayProducer` parameter must be an array producer. Assuming that the element type of `inputArrayProducer` is `ElementType`, `functionName` is the name of a function in the same module with one of the following prototypes:

- `operator functionName(io ElementType element)`
- `operator functionName(io ElementType element, Index index)`
- `operator functionName(io ElementType element, Index index, Size count)`

In cases where the `index` parameter is present, the index of the element within the array is passed to the generator function; similarly, when the `count` parameter is present the total number of elements in the array is passed. The result of the `createArrayTransform` call is a value of type `ArrayProducer<ElementType>`.

An example of an array transform:

```
operator transform(io Scalar value) {
  value = sqrt(value);
}

operator entry() {
  Scalar ia[]; ia.push(3.14); ia.push(2.71); ia.push(10.0); ia.push(87.32);
  ArrayProducer<Scalar> iap = createConstArray(ia);
  report(iap.produce());
  ArrayProducer<Scalar> oap = createArrayTransform(iap, transform);
  report(oap.produce());
}
```

produces:

```
[3.14,2.71,10,87.32]
[1.772004,1.646208,3.162278,9.344517]
```

In both cases, you can optionally specify a shared value producer as the last parameter to the `create...Transform()` call. When a shared value producer is provided, the function receives an additional parameter whose type is the value type of the shared value producer. So, for value maps, the prototype of the operator becomes:

```
operator gen(io ValueType value, SharedType sharedValue) {
  ...
}
```

and for array maps the prototype of the operator becomes:

```
operator gen(io ElementType element, Index index, Size count, SharedType sharedValue) {
  ...
}
```

Note that when using a shared value with an array map you must include the `index` and `count` parameters in the operator even if they are unused.

2.5. KL Reduce Operations

To create a reduce operation in KL, use the `createReduce(inputArrayProducer, functionName)` call. `inputArrayProducer` must be an array producer; assume its element type is `InputType`. `functionName` must be the name of a function in the same module with one of the prototypes:

- `operator functionName(InputType input, io OutputType output)`
- `operator functionName(InputType input, io OutputType output, Index index)`
- `operator functionName(InputType input, io OutputType output, Index index, Size count)`

The result of the `createReduce` call is a value producer with value type `OutputType`, ie. a result of type `ValueProducer<OutputType>`. An example of using a reduce operation in KL:

```
operator generate(io Size value, Index index) {
  value = index + 1;
}
```

```
operator reduce(Size input, io Size output) {
  output += input;
}

operator entry() {
  // Report the sum 1+2+...+99+100
  ValueProducer<Size> vp = createReduce(
    createArrayGenerator(
      createConstValue(Size(100)),
      generate
    ),
    reduce
  );
  report(vp.produce());
}
```

This produces the result:

```
5050
```

You can optionally specify a shared value producer as the last parameter to the `createReduce()` call. When a shared value producer is provided, the function receives an additional parameter whose type is the value type of the shared value producer. The prototype of the operator becomes:

```
operator reduce(InputType input, io OutputType output, Index index, Size count, SharedType
sharedValue) {
  ...
}
```

Note that when using a shared value with an array map you must include the `index` and `count` parameters in the operator even if they are unused.

2.6. KL Caches

A value cache is created in KL using the `createValueCache(inputValueProducer)` call. The element type of the resulting value producer is the same as that of `inputValueProducer`. Example usage of a value cache in KL:

```
operator gen(io String output) {
  report("  Running generator!");
  output = "Hello";
}

operator entry() {
  // test caching ValueGenerator
  ValueProducer<String> vp1 = createValueCache(createValueGenerator(gen));
  report("vp1 = " + vp1);

  report("Should run generator");
  report("vp1.produce() = " + vp1.produce());

  report("Should not run generator (use cache)");
  report("vp1.produce() = " + vp1.produce());

  vp1.flush();
  report("Should run generator");
  report("vp1.produce() = " + vp1.produce());
}
```

resulting in:


```
vp1 = ValueProducer<String>
Should run generator
  Running generator!
vp1.produce() = Hello
Should not run generator (use cache)
vp1.produce() = Hello
Should run generator
  Running generator!
vp1.produce() = Hello
```

Similarly, an array cache is created using the `createArrayProducer(inputArrayProducer)` call. The resulting array producer has the same element type as `inputArrayProducer`. Example usage:

```
operator generator(
  io Integer value
  )
{
  report("  Running generator");
  value = 42;
}

operator entry() {
  // Generator caching
  ValueProducer<Size> cvg = createConstValue(Size(10));
  ArrayProducer<Integer> gen = createArrayCache(
    createArrayGenerator(cvg, generator)
    );

  report(gen);
  report(" gen.getCount() = " + gen.getCount());

  report("Should run generator 10x");
  for (Index i=0; i<10; ++i)
    report(" gen.produce() = " + gen.produce(i));

  report("Should not run generator (cached)");
  for (Index i=0; i<10; ++i)
    report(" gen.produce() = " + gen.produce(i));

  gen.flush();
  report("Should run generator 10x");
  for (Index i=0; i<10; ++i)
    report(" gen.produce() = " + gen.produce(i));
}
```

resulting in:

```
ArrayProducer<Integer>
 gen.getCount() = 10
Should run generator 10x
  Running generator
 gen.produce() = 42
  Running generator
 gen.produce() = 42
  Running generator
 gen.produce() = 42
  Running generator
 gen.produce() = 42
  Running generator
 gen.produce() = 42
  Running generator
 gen.produce() = 42
  Running generator
 gen.produce() = 42
  Running generator
```

```
 gen.produce() = 42
  Running generator
 gen.produce() = 42
  Running generator
 gen.produce() = 42
Should not run generator (cached)
 gen.produce() = 42
 gen.produce() = 42
 gen.produce() = 42
 gen.produce() = 42
 gen.produce() = 42
 gen.produce() = 42
 gen.produce() = 42
 gen.produce() = 42
 gen.produce() = 42
 gen.produce() = 42
Should run generator 10x
  Running generator
 gen.produce() = 42
  Running generator
  Running transform
 tr.produce() = 2
  Running transform
 tr.produce() = 4
  Running transform
 tr.produce() = 6
  Running transform
 tr.produce() = 8
  Running transform
 tr.produce() = 10
  Running transform
 tr.produce() = 12
  Running transform
 tr.produce() = 14
  Running transform
 tr.produce() = 16
  Running transform
 tr.produce() = 18
```


Chapter 3. Scripting Language Implementations (JavaScript/Python)

All of the map-reduce functionality that is available directly in KL is also available from JavaScript and Python. Syntax between JavaScript and Python is nearly identical and therefore most examples will be given in JavaScript, unless there the corresponding Python implementation would be unclear.

3.1. Creating a `client` object

Creating a `client` object does differ between JavaScript and Python so we will describe it here once and then leave it out of further examples.

In JavaScript:

```
var client = require('Fabric').createClient();
```

And in Python:

```
import fabric
client = fabric.createClient()
```

The client object is the base from which the rest of the Map Reduce objects are created.

3.2. Compilation of KL Code

In order to provide the KL code used for operators for map-reduce operations, a generic KL compilation framework has been provided.

For simple examples, functions for compiling operators directly have been provided:

- `client.KLC.createValueGeneratorOperator`
- `client.KLC.createValueMapOperator`
- `client.KLC.createValueTransformOperator`
- `client.KLC.createArrayGeneratorOperator`
- `client.KLC.createArrayMapOperator`
- `client.KLC.createArrayTransformOperator`
- `client.KLC.createReduceOperator`

Each of these functions take the parameters `sourceName`, `sourceCode`, `operatorName`. `sourceName` is the name of the source file to display when a compilation error occurs, `sourceCode` is the actual KL source code, and `operatorName` is the name of the operator that should result from the call. For example:

```
   > var reduce = client.KLC.createReduceOperator('reduce.kl', 'operator reduce(Size input, io
Size output) { output += input; }', 'reduce');
   > console.log(reduce.toJSON());
```

results in:

```
{ kind: 'ReduceOperator',
  entryName: 'reduce',
  ast:
   [ { '+': 'Operator',
       returnExprType: '',
       body: [Object],
       friendlyName: 'reduce',
       entryName: 'reduce',
       params: [Object] } ] }
```

which is a JSON representation of the operator. This operator can then be passed to a call to `client.MR.createReduce(...)` as the operator parameter.

For more complex examples, it will make more sense to include one or more KL 'source files' into a single compilation unit and to then resolve the operators within. This is done using the `client.KLC.createCompilation()`, `compilation.addSourceFile(sourceName, sourceCode)`, `compilation.run()` and `executable.resolve...Operator()` functions. For example:

```
> var compilation = fabricClient.KLC.createCompilation();
> compilation.addSource('generate.kl', 'operator generate(io Size value, Index index) { value = index + 1; }');
> compilation.addSource('reduce.kl', 'operator reduce(Size input, io Size output) { output += input; }');
> var executable = compilation.run();
> var reduce = executable.resolveReduceOperator('reduce');
> reduce.toJSON();
```

results in:

```
{ kind: 'ReduceOperator',
  entryName: 'reduce',
  ast:
   [ { '+': 'Operator',
       returnExprType: '',
       body: [Object],
       friendlyName: 'generate',
       entryName: 'generate',
       params: [Object] },
     { '+': 'Operator',
       returnExprType: '',
       body: [Object],
       friendlyName: 'reduce',
       entryName: 'reduce',
       params: [Object] } ] }
```

Note that the AST in the JSON representation in this case is larger; this is because the two "source files" we combined in a single compilation unit. Similarly, we could call:

```
> var generate = executable.resolveArrayGeneratorOperator('generate');
> generate.toJSON();
```

which results in:

```
{ kind: 'ArrayGeneratorOperator',
  entryName: 'generate',
  ast:
   [ { '+': 'Operator',
```



```
          returnExprType: '',
          body: [Object],
          friendlyName: 'generate',
          entryName: 'generate',
          params: [Object] },
        { '+': 'Operator',
          returnExprType: '',
          body: [Object],
          friendlyName: 'reduce',
          entryName: 'reduce',
          params: [Object] } ] }
```

3.3. Asynchronous Production

In addition to the usual `produce()` methods on value and array producers, the JavaScript and Python bindings also support `produceAsync(callback)` methods. `produceAsync(callback)` causes the produce operation to happen asynchronously (ie. in the background); when the operation is complete, the function callback is called, passing the result of the `produce()` operation as the first parameter. This is very useful for servicing computation in the background without blocking the main thread.

For example, in JavaScript:

```
var cv = client.MR.createConstValue('Size', 42);
cv.produceAsync(function (result) {
  console.log("Result is " + result);
  fabricClient.close();
});
console.log("Waiting for result...");
```

And in Python, since the syntax is slightly different:

```
cv = client.MR.createConstValue('Size', 42);
def callback(result):
  print 'Result is ' + result
  client.close()
cv.produceAsync(callback)
print 'Waiting for result...'
```

will both produce:

```
Waiting for result...
Result is 42
```

Python programmers may ask how it is that the Python example doesn't simply exit immediately when it hits the end of file. This is because the Fabric Python module waits for all its clients to close before allowing the script to exit. If more control is required over when the clients will exit, the user can use the Fabric client's `running()` method to both test whether or not the client object is still running, as well as to service any pending actions.

```
cv = client.MR.createConstValue('Size', 42);
def callback(result):
  print 'Result is ' + result
  client.close()
cv.produceAsync(callback)
print 'Waiting for result...'
while client.running():
  # perform some other actions
  pass
print 'Program exiting...'
```


3.4. Using a Single Fabric Client for Multiple Computations

Unlike the dependency graph, it is safe to use a single Fabric client for multiple computations, as long as you create separate producers and reduce operations for each computation. In this way, the operators can be shared between the computations. A simple Node.js HTTP server to generate the Fibonacci numbers might look like:

```
fabric = require('Fabric');
http = require('http');
fs = require('fs');

var fabricClient = fabric.createClient();

var fibonacciGeneratorOperator = fabricClient.KLC.createValueGeneratorOperator(
  "fibonacci-mr.kl",
  fs.readFileSync("fibonacci-mr.kl", "utf8"),
  "fibonacci"
);

http.createServer(function (req, res) {
  var fibonacciGenerator = fabricClient.MR.createValueGenerator(
    fibonacciGeneratorOperator,
    fabricClient.MR.createConstValue('Integer', 40)
  );
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end("\n"+fibonacciGenerator.produce()+"\n");
}).listen(1337, "127.0.0.1");
```

3.5. Constant Producers

In JavaScript and Python, a constant value producer is created using the `client.MR.createConstValue(typeName, value)` call. The `typeName` parameter is the name of the type of the value (eg. `Size`), and the `value` parameter is the value to use. The value is interpreted as the specified type; if no conversion is possible an error results. For example, the JavaScript code:

```
client.MR.createConstValue(
  'Scalar',
  1.4142
).produce();
```

produces the result:

```
1.414199948310852
```

(since floating-point numbers are 64-bit in JavaScript)

A constant array producer is created using the `client.MR.createConstArray(elementTypeName, array)` function. The `elementTypeName` parameter is the name of the element type of the resulting array producer, and the `array` parameter must be an array of values that can be casted to the specified element type. For example, the JavaScript code:

```
client.MR.createConstArray(
  'Scalar',
  [1.4142, 3.14159, 2.718281]
```



```
).produce();
```

produces the result:

```
[ 1.414199948310852,
  3.141590118408203,
  2.718281030654907 ]
```

3.6. Generators

A value generator in JavaScript or Python is created using the `client.MR.createValueGenerator(valueGeneratorOperator)`. `valueGeneratorOperator` must be a value generator operator returned from `client.KLC.createValueGeneratorOperator()` or `executable.resolveValueGeneratorOperator()`. For example, the JavaScript code:

```
client.MR.createValueGenerator(
  client.KLC.createValueGeneratorOperator(
    "generate.kl",
    "operator generate(io Scalar x) { x = 5.9; }",
    "generate"
  )
).produce();
```

produces the output:

```
5.900000095367432
```

An array generator in KL is created using the `client.MR.createArrayGenerator(countValueGenerator, arrayGeneratorOperator)`. The `countValueProducer` parameter must be a value producer that produces a value of type `Size`. `arrayGeneratorOperator` must be an array generator operator returned from `client.KLC.createArrayGeneratorOperator()` or `executable.resolveArrayGeneratorOperator()`. An example of an array generator:

```
client.MR.createArrayGenerator(
  client.MR.createConstValue('Size', 1000000),
  client.KLC.createArrayGeneratorOperator(
    "generate.kl",
    "operator generate(io Scalar x, Index index) { x = 3.141 * index; }",
    "generate"
  )
).produce(9876, 16);
```

produces:

```
[ 31020.515625,
  31023.658203125,
  31026.798828125,
  31029.939453125,
  31033.080078125,
  31036.220703125,
  31039.36328125,
  31042.50390625,
  31045.64453125,
  31048.78515625,
  31051.92578125,
```

```
    31055.06640625,
    31058.208984375,
    31061.349609375,
    31064.490234375,
    31067.630859375 ]
```

In both cases, you can optionally specify a shared value producer as the last parameter to the `client.MR.create...Generator()` call. When a shared value producer is provided, the operator receives an additional parameter whose type is the value type of the shared value producer, as in KL.

3.7. Maps

A value map in JavaScript or Python is created using the `client.MR.createValueMap(inputValueProducer, valueMapOperator)`. `inputValueProducer` must be a value producer, and `valueMapOperator` must be a value map operator returned from `client.KLC.createValueMapOperator()` or `executable.resolveValueMapOperator()`. For example, the JavaScript code:

```
client.MR.createValueMap(
  client.MR.createConstValue(
    "String",
    "Hello, world!"
  ),
  client.KLC.createValueMapOperator(
    "map.kl",
    "operator map(String input, io Size output) { output = input.length; }",
    "map"
  )
).produce();
```

produces the output:

```
13
```

An array map in JavaScript or Python is created using the `client.MR.createArrayMap(inputArrayProducer, arrayMapOperator)`. An example of an array map:

```
client.MR.createArrayMap(
  client.MR.createConstArray(
    "String",
    ["Hello, world!", "Goodbye, cruel world!"]
  ),
  client.KLC.createArrayMapOperator(
    "map.kl",
    "operator map(String input, io Size output) { output = input.length; }",
    "map"
  )
).produce();
```

produces:

```
[ 13, 21 ]
```

As in KL, you can optionally specify a shared value producer as the last parameter to the `client.MR.create...Map()` call. When a shared value producer is provided, the operator receives an additional parameter whose type is the value type of the shared value producer, as in KL.

3.8. Transforms

A value transform in JavaScript or Python is created using the `client.MR.createValueTransform(inputValueProducer, valueTransformOperator)`. The type of the first parameter of the operator must be the value type of the input producer. For example:

```
client.MR.createValueTransform(
  client.MR.createConstValue(
    "String",
    "Peter"
  ),
  client.KLC.createValueTransformOperator(
    "trans.kl",
    "operator trans(io String value) { value = 'Hello, ' + value + '!'; }",
    "trans"
  )
).produce();
```

produces the output:

```
'Hello, Peter!'
```

An array transform in JavaScript or Python is created using the `client.MR.createArrayTransform(inputArrayProducer, arrayTransformOperator)`. The type of the first parameter of the operator must be the element type of the input producer. For example:

```
client.MR.createArrayTransform(
  client.MR.createConstArray(
    "String",
    ["Peter","Fred","Joe"]
  ),
  client.KLC.createArrayTransformOperator(
    "trans.kl",
    "operator trans(io String value) { value = 'Hello, ' + value + '!'; }",
    "trans"
  )
).produce();
```

produces:

```
[ 'Hello, Peter!',
  'Hello, Fred!',
  'Hello, Joe!' ]
```

As in KL, you can optionally specify a shared value producer as the last parameter to the `client.MR.create...Transform()` call. When a shared value producer is provided, the operator receives an additional parameter whose type is the value type of the shared value producer, as in KL.

3.9. Reduce

To create a reduce operation in JavaScript or Python, use the `client.MR.createReduce(inputArrayProducer, reduceOperator)` call. `inputArrayProducer` must be an array producer; assume its element type is `InputType`. `reduceOperator` must be a reduce operator with one of prototypes:

- `operator operatorName(InputType input, io OutputType output)`
- `operator operatorName(InputType input, io OutputType output, Index index)`
- `operator operatorName(InputType input, io OutputType output, Index index, Size count)`

The result of the `client.MR.createReduce(...)` call is a value producer with value type `OutputType`. For example:

```
client.MR.createReduce(
  client.MR.createConstArray(
    "String",
    ["Peter","Fred","Joe"]
  ),
  client.KLC.createReduceOperator(
    "reduce.kl",
    "operator reduce(String input, io Size output) { output += input.length; }",
    "reduce"
  )
).produce();
client.close();
```

produces the result:

```
12
```

You can optionally specify a shared value producer as the last parameter to the `client.MR.createReduce()` call. When a shared value producer is provided, the function receives an additional parameter whose type is the value type of the shared value producer. The prototype of the operator becomes:

```
operator reduce(InputType input, io OutputType output, Index index, Size count, SharedType
sharedValue) {
  ...
}
```

Note that when using a shared value with an array map you must include the `index` and `count` parameters in the operator even if they are unused.

3.10. Caches

To create a value cache in JavaScript or Python, use the `client.MR.createValueCache(inputValueProducer)` call. The value type of the cache is the same as the value type of `inputValueProducer`. Example usage:

```
var gen =
  client.MR.createValueCache(
    client.MR.createValueGenerator(
      client.KLC.createValueGeneratorOperator(
        "xfo.kl", "operator xfo(io Integer value) { report('Running generator'); value =
121; }", "xfo"
        )
      )
    );

console.log("Should run generator");
console.log(gen.produce());
```



```
    console.log("Should not run generator (cached)");
    console.log(gen.produce());

    gen.flush();
    console.log("Should run generator");
    console.log(gen.produce());

    client.close();
```

results in:

```
    Should run generator
    [FABRIC] [MT] Running generator
    121
    Should not run generator (cached)
    121
    Should run generator
    [FABRIC] [MT] Running generator
    121
```

To create an array cache in JavaScript or Python, use the `client.MR.createArrayCache(inputArrayProducer)` call. The element type of the resulting array provider is the same as the element type of `inputArrayProducer`. Example usage:

```
    cv = client.MR.createConstValue("Size", 10);

    ago = client.KLC.createArrayGeneratorOperator("foo.kl", "operator foo(io Scalar output, Index
index) { report('Running generator'); output = sqrt(Scalar(index)); }", "foo");
    ag = client.MR.createArrayGenerator(cv, ago);

    var cache = client.MR.createArrayCache(ag);

    var count = ag.getCount();
    console.log("cache.getCount() = " + count);

    console.log("Should run generator");
    for (var i=0; i<count; ++i)
      console.log("cache.produce("+i+") = "+cache.produce(i));

    console.log("Should not run generator (cached)");
    for (var i=0; i<count; ++i)
      console.log("cache.produce("+i+") = "+cache.produce(i));

    cache.flush();
    console.log("Should run generator");
    for (var i=0; i<count; ++i)
      console.log("cache.produce("+i+") = "+cache.produce(i));

    client.close();
```

results in:

```
    cache.getCount() = 10
    Should run generator
    [FABRIC] [MT] Running generator
    cache.produce(0) = 0
    [FABRIC] [MT] Running generator
    cache.produce(1) = 1
    [FABRIC] [MT] Running generator
    cache.produce(2) = 1.414213538169861
    [FABRIC] [MT] Running generator
    cache.produce(3) = 1.732050776481628
    [FABRIC] [MT] Running generator
    cache.produce(4) = 2
```

```
[FABRIC] [MT] Running generator
cache.produce(5) = 2.2360680103302
[FABRIC] [MT] Running generator
cache.produce(6) = 2.449489831924438
[FABRIC] [MT] Running generator
cache.produce(7) = 2.645751237869263
[FABRIC] [MT] Running generator
cache.produce(8) = 2.828427076339722
[FABRIC] [MT] Running generator
cache.produce(9) = 3
Should not run generator (cached)
cache.produce(0) = 0
cache.produce(1) = 1
cache.produce(2) = 1.414213538169861
cache.produce(3) = 1.732050776481628
cache.produce(4) = 2
cache.produce(5) = 2.2360680103302
cache.produce(6) = 2.449489831924438
cache.produce(7) = 2.645751237869263
cache.produce(8) = 2.828427076339722
cache.produce(9) = 3
Should run generator
[FABRIC] [MT] Running generator
cache.produce(0) = 0
[FABRIC] [MT] Running generator
cache.produce(1) = 1
[FABRIC] [MT] Running generator
cache.produce(2) = 1.414213538169861
[FABRIC] [MT] Running generator
cache.produce(3) = 1.732050776481628
[FABRIC] [MT] Running generator
cache.produce(4) = 2
[FABRIC] [MT] Running generator
cache.produce(5) = 2.2360680103302
[FABRIC] [MT] Running generator
cache.produce(6) = 2.449489831924438
[FABRIC] [MT] Running generator
cache.produce(7) = 2.645751237869263
[FABRIC] [MT] Running generator
cache.produce(8) = 2.828427076339722
[FABRIC] [MT] Running generator
cache.produce(9) = 3
```

