mapreduce

Programming technique for analyzing data sets that do not fit in memory

collapse all in page

Syntax

```
outds = mapreduce(ds,mapfun,reducefun)
outds = mapreduce(ds,mapfun,reducefun,mr)
outds = mapreduce(__,Name,Value)
```

Description

example

`outds = mapreduce(ds,mapfun,reducefun)` applies map function `mapfun` to input datastore `ds`, and then passes the values associated with each unique key to reduce function `reducefun`. The output datastore is a `KeyValueDatastore` object that points to `.mat` files in the current folder.

`outds = mapreduce(ds,mapfun,reducefun,mr)` optionally specifies the run-time configuration settings for `mapreduce`. The `mr` input is the result of a call to the `mapreducer` function. Typically, this argument is used with Parallel Computing Toolbox™, MATLAB® Distributed Computing Server™, or MATLAB Compiler™. For more information, see Speed Up and Deploy MapReduce Using Other Products.

`outds = mapreduce(__,Name,Value)` specifies additional options with one or more `Name,Value` pair arguments using any of the previous syntaxes. For example, you can specify `'OutputFolder'` followed by a character vector specifying a path to the output folder.

Examples

collapse all

Count Flights by Airline

Try This Example

Use `mapreduce` to count the number of flights made by each unique airline carrier in a data set.

Create a datastore using the `airlinesmall.csv` data set. This 12-megabyte data set contains 29 columns of flight information for several airline carriers, including arrival and departure times. In this example, select `UniqueCarrier`(airline name) as the variable of interest. Specify the `'TreatAsMissing'` name-value pair so that the datastore treats `'NA'` values as missing and replaces them with `NaN` values.

```
ds = tabularTextDatastore('airlinesmall.csv', 'TreatAsMissing', 'NA');
ds.SelectedVariableNames = 'UniqueCarrier';
ds.SelectedFormats = '%C';
```

Preview the data.

```
preview(ds)
```

```
ans=8×1 table
    UniqueCarrier
    _____________

        PS
        PS
        PS
        PS
        PS
        PS
        PS
        PS
```

Run `mapreduce` on the data. The map and reduce functions count the number of instances of each airline carrier name in each chunk of data, then combine those intermediate counts into a final count. This method leverages the intermediate sorting by unique key performed by `mapreduce`. The functions `countMapper` and `countReducer` are included at the end of this script.

```
outds = mapreduce(ds, @countMapper, @countReducer);
```

```
********************************
*      MAPREDUCE PROGRESS      *
********************************
Map   0% Reduce   0%
Map  16% Reduce   0%
Map  32% Reduce   0%
Map  48% Reduce   0%
Map  65% Reduce   0%
Map  81% Reduce   0%
Map  97% Reduce   0%
Map 100% Reduce   0%
Map 100% Reduce  10%
Map 100% Reduce  21%
Map 100% Reduce  31%
Map 100% Reduce  41%
Map 100% Reduce  52%
Map 100% Reduce  62%
Map 100% Reduce  72%
Map 100% Reduce  83%
Map 100% Reduce  93%
Map 100% Reduce 100%
```

```
readall(outds)
```

```
ans=29×2 table
      Key         Value
    ________    _______

    'AA'        [14930]
    'AS'        [ 2910]
    'CO'        [ 8138]
    'DL'        [16578]
    'EA'        [  920]
    'HP'        [ 3660]
    'ML (1)'    [   69]
    'NW'        [10349]
    'PA (1)'    [  318]
    'PI'        [  871]
    'PS'        [   83]
    'TW'        [ 3805]
    'UA'        [13286]
    'US'        [13997]
    'WN'        [15931]
    'AQ'        [  154]
```

The map function `countMapper` leverages the fact that the data is categorical. The `countcats` and `categories` functions are used on each chunk of the input data to generate key/value pairs of the airline name and associated count.

```
function countMapper(data, info, intermKV)
% Counts unique airline carrier names in each chunk.
a = data.UniqueCarrier;
```

```
c = num2cell(countcats(a));
keys = categories(a);
addmulti(intermKV, keys, c)
end
```

The reduce function `countReducer` reads in the intermediate data produced by the map function and adds together all of the counts to produce a single final count for each airline carrier.

```
function countReducer(key, intermValIter, outKV)
% Combines counts from all chunks to produce final counts.
count = 0;
while hasnext(intermValIter)
    data = getnext(intermValIter);
    count = count + data;
end
add(outKV, key, count)
end
```

Input Arguments

collapse all

`ds` — Input datastore
datastore object

Input datastore, specified as a datastore object. Use the `datastore` function to create a datastore object from your data set.

`mapfun` — Function handle to map function
function handle

Function handle to map function. `mapfun` receives chunks from input datastore `ds`, and then uses the `add` and `addmulti` functions to add key-value pairs to an intermediate `KeyValueStore` object. The number of calls to the map function by `mapreduce` is equal to the number of chunks in the `datastore` (the number of chunks is determined by the `ReadSize` property of the datastore).

The inputs to the map function are `data`, `info`, and `intermKVStore`, which `mapreduce` automatically creates and passes to the map function:

- The `data` and `info` inputs are the result of a call to the `read` function of `datastore`, which `mapreduce` executes automatically before each call to the map function.
- `intermKVStore` is the name of the intermediate `KeyValueStore` object to which the map function needs to add key-value pairs. If none of the calls to the map function add key-value pairs to `intermKVStore`, then `mapreduce` does not call the reduce function and the output datastore is empty.

  An example of a template for the map function is

  ```
  function myMapper(data, info, intermKVStore)

  %do a calculation with the data chunk

  add(intermKVStore, key, value)

  end
  ```

  Example: `@myMapper`

  Data Types: `function_handle`

`reducefun` — Function handle to reduce function
function handle

Function handle to reduce function. `mapreduce` calls `reducefun` once for each unique key added to the intermediate `KeyValueStore` by the map function. In each call, `mapreduce` passes the values associated with the active key to `reducefun` as a `ValueIterator` object.

The `reducefun` function loops through the values for each key using the `hasnext` and `getnext` functions. Then, after performing some calculation(s), it writes key-value pairs to the final output.

The inputs to the reduce function are `intermKey`, `intermValIter`, and `outKVStore`, which `mapreduce` automatically creates and passes to the reduce function:

- `intermKey` is the active key from the intermediate `KeyValueStore` object. Each call to the reduce function by `mapreduce` specifies a new unique key from the keys in the intermediate `KeyValueStore` object.
- `intermValIter` is the `ValueIterator` associated with the active key, `intermKey`. This `ValueIterator` object contains all of the values associated with the active key. Scroll through the values using the `hasnext` and `getnext` functions.
- `outKVStore` is the name for the final `KeyValueStore` object to which the reduce function needs to add key-value pairs. `mapreduce` takes the output key-value pairs from `outKVStore` and returns them in the output datastore, `outds`, which is a `KeyValueDatastore` object by default. If none of the calls to the reduce function add final key-value pairs to `outKVStore`, then the output datastore is empty.

  An example of a template for the reduce function is

```
function myReducer(intermKey, intermValIter, outKVStore)

while hasnext(intermValIter)

    X = getnext(intermValIter);

    %do a calculation with the current value, X

end

add(outKVStore, key, value)

end
```

Example: `@myReducer`

Data Types: `function_handle`

`mr` — Execution environment
MapReducer object

Execution environment, specified as a MapReducer object. `mr` is the result of a call to the `mapreducer` function. The default `mr` argument is a call to `gcmr`, which uses the default global execution environment for `mapreduce` (in MATLAB the default is `mapreducer(0)`, which returns a `SerialMapReducer` object).

Note

This setting specifies the execution environment for `mapreduce` and is not necessary to run `mapreduce` on your local computer. For more information, see Speed Up and Deploy MapReduce Using Other Products.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `outds = mapreduce(ds, @mapfun, @reducefun, 'Display', 'off', 'OutputFolder', 'C:\Users\username\Desktop')`

collapse all

`'OutputType'` — Type of datastore output
`'Binary'` (default) | `'TabularText'`

Type of datastore output, specified as `'Binary'` or `'TabularText'`. The default setting of `'Binary'` returns a `KeyValueDatastore` output datastore that points to binary (`.mat` or `.seq`) files in the output folder. The `'TabularText'` option returns a `TabularTextDatastore` output datastore that points to `.txt` files in the output folder.

The table provides the details for each of the output types.

`'OutputType'`	Type of datastore output	Datastore points to files of type	Values that the Reduce function can add	Keys that the R can add
`'Binary'` (default)	`KeyValueDatastore`	`.mat` (or `.seq` when running against Hadoop®).	Any valid MATLAB object.	Character vector scalars that are n complex, logical
`'TabularText'`	`TabularTextDatastore`	`.txt`	Character vectors, or numeric scalars that are not `NaN`, complex, logical, or sparse.	Character vector scalars that are n complex, logical

Data Types: `char` | `string`

`'OutputFolder'` — Destination folder of `mapreduce` output
`pwd` (default) | file path

Destination folder for `mapreduce` output, specified as a file path. The default output folder is the current folder, `pwd`. You can specify a different path with a fully qualified path or with a path relative to the current folder.

Example: `mapreduce(..., 'OutputFolder', 'MyOutputFolder\Results')` specifies a file path relative to the current folder for the output.

Data Types: `char` | `string`

`'Display'` — Toggle for command line progress output
`'on'` (default) | `'off'`

Toggle for command line progress output, specified as `'on'` or `'off'`. The default is `'on'`, so that `mapreduce` displays progress information in the command window during the map and reduce phases of execution.

Data Types: `char` | `string`

Output Arguments

collapse all

`outds` — Output datastore
`KeyValueDatastore` (default) | `TabularTextDatastore`

Output datastore, returned as a `KeyValueDatastore` or `TabularTextDatastore` object. By default, `outds` is a `KeyValueDatastore` object that points to `.mat`files in the current folder. Use the `Name,Value` pair arguments for `'OutputType'` and `'OutputFolder'` to return a `TabularTextDatastore` object or change the location of the output files, respectively.

`mapreduce` does not sort the key-value pairs in `outds`. Their order may differ when using other products with `mapreduce`.

To view the contents of `outds`, use the `preview`, `read`, or `readall` functions of `datastore`.

Tips

- Debugging your `mapreduce` algorithms to examine how key-value pairs move through the different phases is always useful. To examine the movement of data, set breakpoints in your

map and reduce functions. The breakpoints stop execution of `mapreduce`, allowing you to examine the current status of relevant variables, like the `KeyValueStore` or `ValueIterator`. For more information, see Debug MapReduce Algorithms.

- Some recommendations to optimize `mapreduce` performance on any platform are:
  - Minimize the number of calls to the map function. The easiest approach is to increase the value of the `ReadSize` property of the input datastore. The result is that `mapreduce` passes larger chunks of data to the map function, and the datastore depletes with fewer reads.
  - Decrease the amount of intermediate data sent between map and reduce functions. One approach is to use `unique` inside a map function to combine similar keys. See Compute Mean by Group Using MapReduce for an example of this technique.

See Also

`KeyValueStore` | `ValueIterator` | `datastore` | `gcmr` | `mapreducer` | `tall`

Topics

- Compute Mean Value with MapReduce
- Compute Summary Statistics by Group Using MapReduce
- Using MapReduce to Fit a Logistic Regression Model
- Getting Started with MapReduce
- Speed Up and Deploy MapReduce Using Other Products
- Build Effective Algorithms with MapReduce