JuliaDB

May 4, 2017

Contents

Part I

Introduction

Chapter 1

JuliaDB.jl

1.1 Overview

The JuliaDB package provides a distributed table data structure where some of the columns form a sorted index. This structure is equivalent to an N-dimensional sparse array, and follows the array API to the extent possible. While a table data structure provided by JuliaDB can be used for any kind of array data, it is highly efficient at the storage and querying of data sets whose indices have a natural sorted order, such as time-series data.

The JuliaDB package provides functionality for ingesting data from a variety of data sources, and provides full integration with the rest of the Julia language ecosystem for performing analytics directly on data stored within a JuliaDB table using many Julia processes.

1.2 Installation

```
Pkg.clone("https://github.com/JuliaComputing/JuliaDB.jl.git")
```

1.3 Introduction

The data structure (called `DTable`) provided by this package maps tuples of indices to data values. Hence, it is similar to a hash table mapping tuples to values, but with a few key differences. First, the index tuples are stored columnwise, with one vector per index position: there is a vector of first indices, a vector of second indices, and so on. The index vectors are expected to be homogeneous to allow more efficient storage. Second, the indices must have a total order, and are stored lexicographically sorted (first by the first index, then by the second index, and so on, left-to-right). While the indices must have totally-ordered types, the data values can be anything. Finally, for purposes of many operations an `DTable` acts like an N-dimensional array of its data values, where the number of dimensions is the number of index columns. A `DTable` implements a distributed memory version of the `IndexedTable` data structure provided by the `IndexedTables.jl` package and re-exported by JuliaDB.

Part II

Tutorial

Chapter 2

Using JuliaDB

2.1 Construction of an IndexedTable

The `IndexedTable` constructor accepts a series of vectors. The last vector contains the data values, and the first N vectors contain the indices for each of the N dimensions. As an example, let's construct an array of the high temperatures for three days in two cities:

```
julia> using IndexedTables, JuliaDB

julia> hitemps = IndexedTable([fill("New York",3); fill("Boston",3)],
                              repmat(Date(2016,7,6):Date(2016,7,8), 2),
                              [91,89,91,95,83,76])
──────────────────────────┬──
"Boston"    2016-07-06 │ 95
"Boston"    2016-07-07 │ 83
"Boston"    2016-07-08 │ 76
"New York"  2016-07-06 │ 91
"New York"  2016-07-07 │ 89
"New York"  2016-07-08 │ 91
```

Notice that the data was sorted first by city name, then date, giving a different order than we initially provided. On construction, `IndexedTable` takes ownership of the columns and sorts them in place (the original vectors are modified).

2.2 Conversion of a Local IndexedTable to a distributed JuliaDB Table

One can convert an existing `IndexedTable` to a JuliaDB `DTable` through the use of the `distribute` function.

```
julia> dhitemps = distribute(hitemps, 2)
DTable with 6 rows in 2 chunks:

──────────────────────────┬──
"Boston"    2016-07-06 │ 95
"Boston"    2016-07-07 │ 83
"Boston"    2016-07-08 │ 76
"New York"  2016-07-06 │ 91
"New York"  2016-07-07 │ 89
...
```

The first argument provided to `distribute` is an existing `IndexedTable` and the second argument describes how the indexed table should be distributed amongst worker processes. If the second argument is a scalar of value n, then the `IndexedTable` will be split into n approximately equal chunks across the worker processes. If the second argument is a vector of n integers, then the distributed table with n separate chunks with each chunk having the number of rows present in each element of that vector.

2.3 Importing data

Reading from CSV files

Importing data from column-based sources is straightforward. JuliaDB currently provides two distinct methods for importing data: `loadfiles` and `ingest`. Both functions load the contents of one or more CSV files in a given directory and return a `DTable` of the loaded data. The `ingest` function has the additional property of transforming the data into an efficient internal storage format, and saving both the original data and associated JuliaDB metadata to disk in a provided output directory.

The argument signature and help for `ingest` is the following:

`JuliaDB.ingest` – Function.

```
ingest(files::Union{AbstractVector,String}, outputdir::AbstractString; <options>...)
```

ingests data from CSV files into JuliaDB. Stores the metadata and index in a directory `outputdir`. Creates `outputdir` if it doesn't exist.

Arguments:

- `delim::Char`: the delimiter to use to read the text file with data. defaults to `,`
- `indexcols::AbstractArray`: columns that are meant to act as the index for the table. Defaults to all but the last column. If `datacols` is set, defaults to all columns other than the data columns. If `indexcols` is an empty vector, an implicit index of itegers `1:n` is added to the data.
- `datacols::AbstractArray`: columns that are meant to act as the data for the table. Defaults to the last column. If `indexcols` is set, defaults to all columns other than the index columns.
- `agg::Function`: aggregation function to use to combine data points with the same index. Defaults to nothing which leaves the data unaggregated (see `aggregate` to aggregate post-loading)). table.)
- `presorted::Bool`: specifies if each CSV file is internally already sorted according to the specified index column. This will avoid a re-sorting.
- `tomemory::Bool`: Load data to memory after ingesting instead of mmapping. Defaults to false.
- The rest of the keyword arguments will be passed on to `TextParse.csvread` which is used by this function to load data from individual files.

See also `loadfiles` and `save`

`source`

The argument signature and help for `loadfiles` is the following:

`JuliaDB.loadfiles` – Function.

```
loadfiles(files::Union{AbstractVector,String}, delim = ','; <options>)
```

Load a collection of CSV `files` into a DTable, where `files` is either a vector of file paths, or the path of a directory containing files to load.

Arguments:

- usecache::Bool: use cached metadata from previous loads while loading the files. Set this to false if you are changing other options.

All other arguments options are the same as those listed in ingest.

See also ingest.

source

As stated above in the help text, each function has a set of optional input arguments that are specific to that particular function, as well as the ability to pass a set of trailing input arguments that are subsequently passed on to TextParse.csvread.

An in-place variant of the ingest! function will append data from new files on to an existing DTable stored in a defined outputdir. The help string for the in-place version of ingest! is the following:

JuliaDB.ingest! – Function.

```
ingest!(files::Union{AbstractVector,String}, outputdir::AbstractString; <options>...)
```

ingest data from files and append into data stored in outputdir. Creates outputdir if it doesn't exist. Arguments are the same as those to ingest. The index range of data in the new files should not overlap with files previously ingested.

See also ingest

source

Saving and Loading existing JuliaDB DTables

Saving an existing DTable can be accomplished through the use of the save function. The save function has the following help string:

JuliaDB.save – Function.

```
save(t::DTable, outputdir::AbstractString)
```

Saves a DTable to disk. This function blocks till all files data has been computed and saved. Saved data can be loaded with load.

See also ingest, load

source

Loading a previously saved DTable from disk can be accomplished through use of the load function. The load function has the following help string:

JuliaDB.load – Function.

```
load(dir::AbstractString; tomemory)
```

Load a saved DTable from dir directory. Data can be saved using ingest or save functions. If tomemory option is true, then data is loaded into memory rather than mmapped.

See also ingest, save

source

2.4 Indexing

Most lookup and filtering operations on `DTable` are done via indexing. Our `dhitemps` array behaves like a 2-d array of integers, accepting two indices:

```
julia> dhitemps["Boston", Date(2016,7,8)]
76
```

If the given indices exactly match the element types of the index columns, then the result is a scalar. In other cases, a new `DTable` is returned, giving data for all matching locations:

```
julia> dhitemps["Boston", :]
DTable with 1 chunks:

──────────────────────┬──
"Boston"  2016-07-06 │ 95
"Boston"  2016-07-07 │ 83
"Boston"  2016-07-08 │ 76
...
```

2.5 Permuting dimensions

As with other multi-dimensional arrays, dimensions can be permuted to change the sort order. With `DTable` the interpretation of this operation is especially natural: simply imagine passing the index columns to the constructor in a different order, and repeating the sorting process:

```
julia> permutedims(dhitemps, [2, 1])
DTable with 2 chunks:

──────────────────────┬──
2016-07-06  "Boston"   │ 95
2016-07-06  "New York" │ 91
2016-07-07  "Boston"   │ 83
2016-07-07  "New York" │ 89
2016-07-08  "Boston"   │ 76
...
```

Now the data is sorted first by date. In some cases such dimension permutations are needed for performance. The leftmost column is esssentially the primary key -- indexing is fastest in this dimension.

2.6 Select and aggregate

In some cases one wants to consider a subset of dimensions, for example when producing a simplified summary of data. This can be done by passing dimension (column) numbers (or names, as symbols) to `select`:

```
julia> select(dhitemps, 2)
DTable with 2 chunks:

──────────┬──
2016-07-06 │ 95
2016-07-06 │ 91
2016-07-07 │ 83
```

```
2016-07-07 | 89
2016-07-08 | 76
...
```

In this case, the result has multiple values for some indices, and so does not fully behave like a normal array anymore. Operations that might leave the array in such a state accept the keyword argument agg, a function to use to combine all values associated with the same indices:

```
julia> select(dhitemps, 2, agg=max)
DTable with 2 chunks:

──────────┼────
2016-07-06 | 95
2016-07-07 | 89
2016-07-08 | 91
...
```

The aggregation operation can also be done by itself, in-place, using the function `aggregate!`.

`select` also supports filtering columns with arbitrary predicates, by passing `column=>predicate` pairs:

```
julia> select(dhitemps, 2=>Dates.isfriday)
DTable with 2 chunks:

──────────────────────┼────
"Boston"    2016-07-08 | 76
"New York"  2016-07-08 | 91
...
```

2.7 Converting dimensions

A location in the coordinate space of an array often has multiple possible descriptions. This is especially common when describing data at different levels of detail. For example, a point in time can be expressed at the level of seconds, minutes, or hours. In our toy temperature dataset, we might want to look at monthly instead of daily highs.

This can be accomplished using the `convertdim` function. It accepts a DTable, a dimension number to convert, a function or dictionary to apply to indices in that dimension, and an aggregation function (the aggregation function is needed in case the mapping is many-to-one). The following call therefore gives monthly high temperatures:

```
julia> convertdim(dhitemps, 2, Dates.month, agg=max)
DTable with 2 chunks:

────────────┼────
"Boston"    7 | 95
"New York"  7 | 91
...
```

2.8 Named columns

`DTable` and `IndexedTable` are built on a simpler data structure called `Columns` that groups a set of vectors together. This structure is used to store the index part of an `IndexedTable`, and a `IndexedTable` can be constructed by passing one of these objects directly. `Columns` allows names to be associated with its constituent vectors. Together, these features allow `IndexedTable` and `DTable` arrays with named dimensions:

```
julia> hitemps = IndexedTable(Columns(city = [fill("New York",3); fill("Boston",3)],
                                      date = repmat(Date(2016,7,6):Date(2016,7,8), 2)),
                                      [91,89,91,95,83,76])
city        date       │
────────────────────────┼──
"Boston"    2016-07-06 │ 95
"Boston"    2016-07-07 │ 83
"Boston"    2016-07-08 │ 76
"New York"  2016-07-06 │ 91
"New York"  2016-07-07 │ 89
"New York"  2016-07-08 │ 91

julia> dhitemps = distribute(hitemps,2)
DTable with 6 rows in 2 chunks:

city        date       │
────────────────────────┼──
"Boston"    2016-07-06 │ 95
"Boston"    2016-07-07 │ 83
"Boston"    2016-07-08 │ 76
"New York"  2016-07-06 │ 91
"New York"  2016-07-07 │ 89
...
```

Now dimensions (e.g. in `select` operations) can be identified by symbol (e.g. `:city`) as well as integer index.

A `Columns` object itself behaves like a vector, and so can be used to represent the data part of a `DTable`. This provides one possible way to store multiple columns of data:

```
julia> t = IndexedTable(Columns(x = rand(4), y = rand(4)),
                        Columns(observation = rand(1:2,4), confidence = rand(4)))
x         y         │ observation  confidence
────────────────────┼──────────────────────────
0.666683  0.193852  │ 2            0.707125
0.668276  0.136898  │ 2            0.519529
0.811008  0.511275  │ 2            0.0158134
0.977066  0.895341  │ 2            0.57636

julia> dt = distribute(t, 2)
DTable with 4 rows in 2 chunks:

x         y         │ observation  confidence
────────────────────┼──────────────────────────
0.666683  0.193852  │ 2            0.707125
0.668276  0.136898  │ 2            0.519529
0.811008  0.511275  │ 2            0.0158134
0.977066  0.895341  │ 2            0.57636
```

In this case the data elements are structs with fields `observation` and `confidence`, and can be used as follows:

```
julia> filter(d->d.confidence > 0.5, dt)
DTable with 2 chunks:

x         y         │ observation  confidence
────────────────────┼──────────────────────────
```

```
0.666683  0.193852 | 2            0.707125
0.668276  0.136898 | 2            0.519529
0.977066  0.895341 | 2            0.57636
...
```

Part III

API Reference

Chapter 3

API documentation

3.1 Compute and gather

Operations in JuliaDB are out-of-core in nature. They return `DTable` objects which can contain parts that are not yet evaluated. `compute` and `gather` are ways to force evaluation.

`Dagger.compute` – Method.

```
compute(t::DTable, allowoverlap=true)
```

Computes any delayed-evaluations in the `DTable`. The computed data is left on the worker processes. Subsequent operations on the results will reuse the chunks.

If `allowoverlap` is false then the computed data is resorted to have no chunks with overlapping index ranges if necessary.

If you expect the result of some operation to be used more than once, it's better to compute it once and then use it many times.

See also `gather`.

> **Warning**
>
> `compute(t)` requires at least as much memory as the size of the result of the computing `t`. If the result is expected to be big, try `compute(save(t, "output_dir"))` instead. See `save` for more.

`source`

`Dagger.gather` – Method.

```
gather(t::DTable)
```

Gets distributed data in a DTable `t` and merges it into IndexedTable object

> **Warning**
>
> `gather(t)` requires at least as much memory as the size of the result of the computing `t`. If the result is expected to be big, try `compute(save(t, "output_dir"))` instead. See `save` for more. This data can be loaded later using `load`.

`source`

3.2 Indexing

`Base.getindex` – Method.

```
t[idx...]
```

Returns a `DTable` containing only the elements of `t` where the given indices (`idx`) match. If `idx` has the same type as the index tuple of the `t`, then this is considered a scalar indexing (indexing of a single value). In this case the value itself is looked up and returned.

source

3.3 Queries

`Base.Sort.select` – Method.

```
select(t::DTable, conditions::Pair...)
```

Filter based on index columns. Conditions are accepted as column-function pairs.

Example: `select(t, 1 => x->x>10, 3 => x->x!=10 ...)`

source

`Base.Sort.select` – Method.

```
select(t::DTable, which...; agg)
```

Returns a new DTable where only a subset of the index columns (specified by `which`) are kept.

The agg keyword argument is a function which specifies how entries with equal indices should be aggregated. If agg is unspecified, then the repeating indices are kept in the output, you can then aggregate using `aggregate`

source

`IndexedTables.aggregate` – Method.

```
aggregate(f, t::DTable)
```

Combines adjacent rows with equal indices using the given 2-argument reduction function `f`.

source

`IndexedTables.aggregate_vec` – Method.

```
aggregate_vec(f::Function, x::DTable)
```

Combine adjacent rows with equal indices using a function from vector to scalar, e.g. `mean`.

source

`Base.filter` – Method.

```
filter(f, t::DTable)
```

Filters `t` removing rows for which `f` is false. `f` is passed only the data and not the index.

source

`IndexedTables.convertdim` – Method.

```
convertdim(x::DTable, d::DimName, xlate; agg::Function, name)
```

Apply function or dictionary `xlate` to each index in the specified dimension. If the mapping is many-to-one, `agg` is used to aggregate the results. `name` optionally specifies a name for the new dimension. `xlate` must be a monotonically increasing function.

See also `reducedim` and `aggregate`

source

`Base.reducedim` – Method.

```
reducedim(f, t::DTable, dims)
```

Remove `dims` dimensions from `t`, aggregate any rows with equal indices using 2-argument function `f`.

See also `reducedim_vec`, `select` and `aggregate`.

source

`IndexedTables.reducedim_vec` – Method.

```
reducedim_vec(f::Function, t::DTable, dims)
```

Like `reducedim`, except uses a function mapping a vector of values to a scalar instead of a 2-argument scalar function.

See also `reducedim` and `aggregate_vec`.

source

3.4 Joins

`IndexedTables.naturaljoin` – Method.

```
naturaljoin(left::DTable, right::DTable, [op])
```

Returns a new `DTable` containing only rows where the indices are present both in `left` AND `right` tables. The data columns are concatenated.

source

`IndexedTables.leftjoin` – Method.

```
leftjoin(left::DTable, right::DTable, [op::Function])
```

Keeps only rows with indices in `left`. If rows of the same index are present in `right`, then they are combined using `op`. `op` by default picks the value from `right`.

source

`IndexedTables.asofjoin` – Method.

```
asofjoin(left::DTable, right::DTable)
```

Keeps the indices of `left` but uses the value from `right` corresponding to highest index less than or equal to that of left.

source

`Base.merge` – Method.

```
merge(left::DTable, right::DTable; agg)
```

Merges `left` and `right` combining rows with matching indices using `agg`. By default `agg` picks the value from `right`.

source