hatchet Documentation

Release 2023.1.1

LLNL Developers

Sep 20, 2023

USER DOCS

Hatchet is a Python-based library that allows Pandas dataframes to be indexed by structured tree and graph data. It is intended for analyzing performance data that has a hierarchy (for example, serial or parallel profiles that represent calling context trees, call graphs, nested regions' timers, etc.). Hatchet implements various operations to analyze a single hierarchical data set or compare multiple data sets, and its API facilitates analyzing such data programmatically.

You can get hatchet from its GitHub repository:

```
$ git clone https://github.com/llnl/hatchet.git
```

or install it using pip:

```
$ pip install llnl-hatchet
```

For Lawrence Livermore National Laboratory users, we recommend using the hatchet installation directly. For more information, see *Using Hatchet on LLNL Systems*.

If you are new to hatchet and want to start using it, see *Getting Started*, or refer to the full *User Guide* below.

CHAPTER

ONE

GETTING STARTED

1.1 Prerequisites

Hatchet has the following minimum requirements, which must be installed before Hatchet is run:

1. Python 2 (2.7) or 3 (3.5 - 3.8)
2. matplotlib
3. pydot
4. numpy, and
5. pandas

Hatchet is available on GitHub.

1.2 Installation

You can get hatchet from its GitHub repository using this command:

```
$ git clone https://github.com/llnl/hatchet.git
```

This will create a directory called `hatchet`.

1.2.1 Install and Build Hatchet

To build hatchet and update your PYTHONPATH, run the following shell script from the hatchet root directory:

```
$ source ./install.sh
```

Note: The `source` keyword is required to update your PYTHONPATH environment variable. It is not necessary if you have already manually added the hatchet directory to your PYTHONPATH.

Alternatively, you can install hatchet using pip:

```
$ pip install llnl-hatchet
```

1.2.2 Check Installation

After installing hatchet, you should be able to import hatchet when running the Python interpreter in interactive mode:

```
$ python
Python 3.7.4 (default, Jul 11 2019, 01:08:00)
[Clang 10.0.1 (clang-1001.0.46.4)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Typing `import hatchet` at the prompt should succeed without any error messages:

```
>>> import hatchet
>>>
```

1.3 Supported data formats

Currently, hatchet supports the following data formats as input:

- HPCToolkit database: This is generated by using `hpcprof-mpi` to post-process the raw measurements directory output by HPCToolkit.
- Caliper Cali file: This is the format in which caliper outputs raw performance data by default.
- Caliper Json-split file: This is generated by either running cali-query on the raw caliper data or by enabling the mpireport service when using caliper.
- DOT format: This is generated by using gprof2dot on `gprof` or `callgrind` output.
- String literal: Hatchet can read as input a list of dictionaries that represents a graph.
- List: Hatchet can also read a list of lists that represents a graph.

For more details on the different input file formats, refer to the *User Guide*.

CHAPTER

TWO

USING HATCHET ON LLNL SYSTEMS

Hatchet installations are available on both Intel and IBM systems at Lawrence Livermore National Laboratory.

To use one of these global installations, add the following to your Python script or Jupyter notebook. This code allows you to use hatchet and its dependencies.

Listing 1: Starter commands to find hatchet and its dependencies

```
import sys
import platform
import datetime as dt
from IPython.display import HTML, display

input_deploy_dir_str = "/usr/gapps/spot/live/"
machine = platform.uname().machine

sys.path.append(input_deploy_dir_str + "/hatchet-venv/" + machine + "/lib/python3.7/site-
↪packages")
sys.path.append(input_deploy_dir_str + "/hatchet/" + machine)
sys.path.append(input_deploy_dir_str + "/spotdb")

import hatchet
import spotdb
```

The following Python script loads a single SPOT/cali file into hatchet using Hatchet's `from_spotdb()`. This returns a list of hatchet `GraphFrames`, and we use `pop()` to access the single `GraphFrame` in the list.

Listing 2: Python script to load a single SPOT file into hatchet

```
import sys
import platform
import datetime as dt
from IPython.display import HTML, display

input_deploy_dir_str = "/usr/gapps/spot/live/"
machine = platform.uname().machine

sys.path.append(input_deploy_dir_str + "/hatchet-venv/" + machine + "/lib/python3.7/site-
↪packages")
sys.path.append(input_deploy_dir_str + "/hatchet/" + machine)
sys.path.append(input_deploy_dir_str + "/spotdb")
```

(continues on next page)

(continued from previous page)

```
import hatchet
import spotdb

input_db_uri_str = "./mpi"
input_run_ids_str = "c5UcO9xwAUKNVVFg1_0.cali"

db = spotdb.connect(input_db_uri_str)
runs = input_run_ids_str.split(',')

gfs = hatchet.GraphFrame.from_spotdb(db, runs)
gf = gfs.pop()

launchdate = dt.datetime.fromtimestamp(int(gf.metadata["launchdate"]))
jobsize = int(gf.metadata.get("jobsize", 1))

print("launchdate: {}, jobsize: {}".format(launchdate, jobsize))
print(gf.tree())
display(HTML(gf.dataframe.to_html()))
```

The following Python script loads multiple SPOT/cali files (most likely contained in the same directory) into hatchet using Hatchet's `from_spotdb()`. The files are specified as a single string, and commas delineate each file. The result is a list of hatchet `GraphFrames`, one for each file.

Listing 3: Python script to load multiple SPOT files into hatchet

```
import sys
import platform
import datetime as dt
from IPython.display import HTML, display

input_deploy_dir_str = "/usr/gapps/spot/live/"
machine = platform.uname().machine

sys.path.append(input_deploy_dir_str + "/hatchet-venv/" + machine + "/lib/python3.7/site-
↪packages")
sys.path.append(input_deploy_dir_str + "/hatchet/" + machine)
sys.path.append(input_deploy_dir_str + "/spotdb")

import hatchet
import spotdb

input_db_uri_str = "./mpi"
input_run_ids_str = "./mpi/cQ-CGJlYj-uFT2yv-_0.cali,./mpi/cQ-CGJlYj-uFT2yv-_1.cali,./mpi/
↪cQ-CGJlYj-uFT2yv-_2.cali"

db = spotdb.connect(input_db_uri_str)
runs = input_run_ids_str.split(',')

gfs = hatchet.GraphFrame.from_spotdb(db, runs)

for idx, gf in enumerate(gfs):
    launchdate = dt.datetime.fromtimestamp(int(gf.metadata["launchdate"]))
```

(continues on next page)

(continued from previous page)

```
    jobsize = int(gf.metadata.get("jobsize", 1))
    print("launchdate: {}, jobsize: {}".format(launchdate, jobsize))
    print(gf.tree())
    display(HTML(gf.dataframe.to_html()))
```

CHAPTER

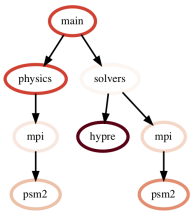
THREE

USER GUIDE

Hatchet is a Python tool that simplifies the process of analyzing hierarchical performance data such as calling context trees. Hatchet uses pandas dataframes to store the data on each node of the hierarchy and keeps the graph relationships between the nodes in a different data structure that is kept consistent with the dataframe.

3.1 Data structures in hatchet

Hatchet's primary data structure is a `GraphFrame`, which combines a structured index in the form of a graph with a pandas dataframe. The images on the right show the two objects in a `GraphFrame` – a `Graph` object (the index), and a `DataFrame object` storing the metrics associated with each node.



Graphframe stores the performance data that is read in from an HPCToolkit database, Caliper Json or Cali file, or gprof/callgrind DOT file. Typically, the raw input data is in the form of a tree. However, since subsequent operations on the tree can lead to new edges being created which can turn the tree into a graph, we store the input data as a directed graph. The graphframe consists of a graph object that stores the edge relationships between nodes and a dataframe that stores different metrics (numerical data) and categorical data associated with each node.

node	name	nid	node	time	time (inc)
main	main	0	main	40.0	200.0
physics	physics	1	physics	40.0	60.0
mpi	mpi	2	mpi	5.0	20.0
psm2	psm2	3	psm2	15.0	15.0
solvers	solvers	4	solvers	0.0	100.0
hypre	hypre	5	hypre	65.0	65.0
mpi	mpi	6	mpi	10.0	35.0
psm2	psm2	7	psm2	25.0	25.0

Graph: The graph can be connected or disconnected (multiple roots) and each node in the graph can have one or more parents and children. The node stores its frame, which can be defined by the reader. The call path is derived by appending the frames from the root to a given node.

Dataframe: The dataframe holds all the numerical and categorical data associated with each node. Since typically the call tree data is per process, a multiindex composed of the node and MPI rank is used to index into the dataframe.

3.2 Reading in a dataset

One can use one of several static methods defined in the GraphFrame class to read in an input dataset using hatchet. For example, if a user has an HPCToolkit database directory that they want to analyze, they can use the `from_hpctoolkit` method:

```
import hatchet as ht

if __name__ == "__main__":
    dirname = "hatchet/tests/data/hpctoolkit-cpi-database"
    gf = ht.GraphFrame.from_hpctoolkit(dirname)
```

Similarly if the input file is a split-JSON output by Caliper, they can use the `from_caliper` method:

```
import hatchet as ht

if __name__ == "__main__":
    filename = ("hatchet/tests/data/caliper-lulesh-json/lulesh-sample-annotation-profile.
↪json")
    gf = ht.GraphFrame.from_caliper(filename)
```

Examples of reading in other file formats can be found in *Analysis Examples*.

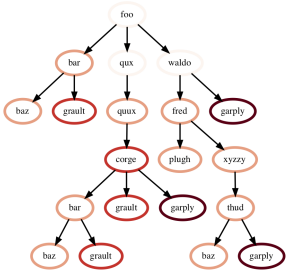
3.3 Visualizing the data

```
0.000 foo
├─ 5.000 bar
│  ├─ 5.000 baz
│  └─ 10.000 grault
├─ 0.000 qux
│  └─ 5.000 quux
│     └─ 10.000 corge
│        ├─ 5.000 bar
│        │  ├─ 5.000 baz
│        │  └─ 10.000 grault
│        ├─ 10.000 grault
│        └─ 15.000 garply
└─ 0.000 waldo
   ├─ 5.000 fred
   │  ├─ 5.000 plugh
   │  └─ 5.000 xyzzy
   │     └─ 5.000 thud
   │        ├─ 5.000 baz
   │        └─ 15.000 garply
   └─ 15.000 garply
```

When the graph represented by the input dataset is small, the user may be interested in visualizing it in entirety or a portion of it. Hatchet provides several mechanisms to visualize the graph in hatchet. One can use the `tree()` function to convert the graph into a string that can be printed on standard output:

```
print(gf.tree())
```

One can also use the `to_dot()` function to output the tree as a string in the Graphviz' DOT format. This can be written to a file and then used to display a tree using the `dot` or `neato` program.



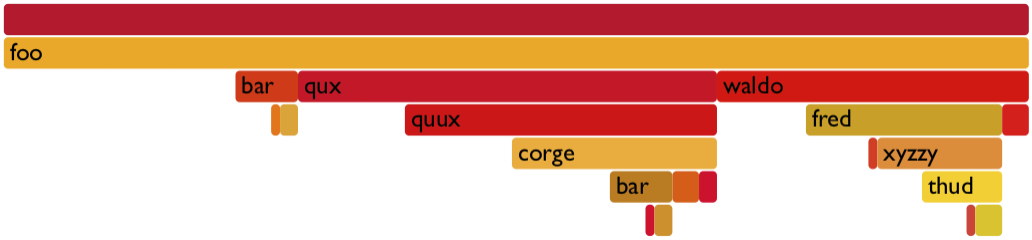

```
with open("test.dot", "w") as dot_file:
    dot_file.write(gf.to_dot())
```

```
$ dot -Tpdf test.dot > test.pdf
```

One can also use the `to_flamegraph` function to output the tree as a string in the folded stack format required by flamegraph. This file can then be used to create a flamegraph using `flamegraph.pl`.

```
with open("test.txt", "w") as folded_stack:
    folded_stack.write(gf.to_flamegraph())
```

```
$ ./flamegraph.pl test.txt > test.svg
```



One can also print the contents of the dataframe to standard output:

```
pd.set_option("display.width", 1200)
pd.set_option("display.max_colwidth", 20)
pd.set_option("display.max_rows", None)

print(gf.dataframe)
```

If there are many processes or threads in the dataframe, one can also print a cross section of the dataframe, say the values for rank 0, like this:

```
print(gf.dataframe.xs(0, level="rank"))
```

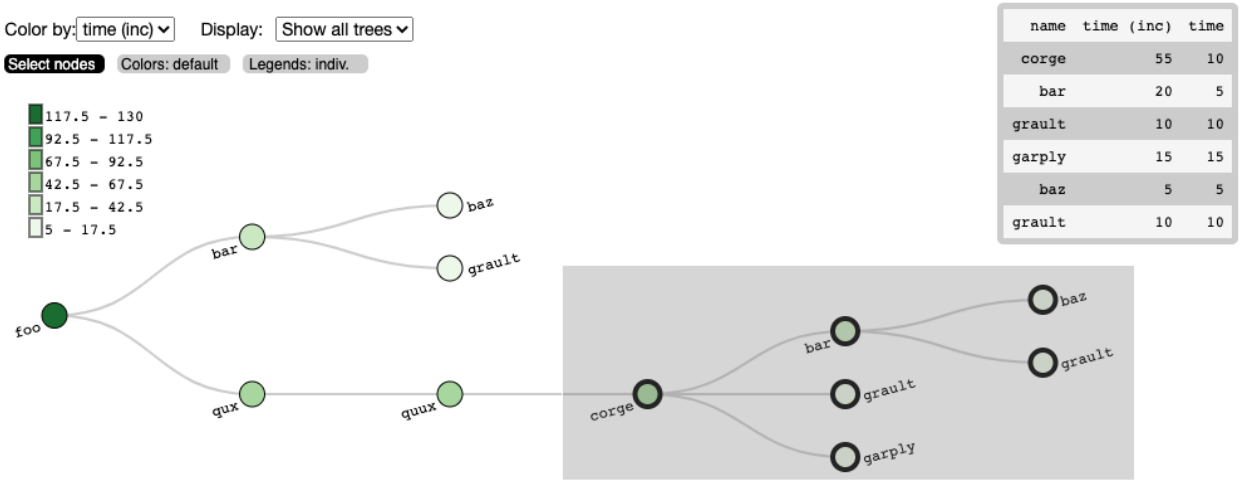
One can also view the graph in Hatchet's interactive visualization for Jupyter. In the Jupyter visualization shown below, users can explore their data by using their mouse to select and hide nodes. For those nodes selected, a table in the the upper right will display the metadata for the node(s) selected. The interactive visualization capability is still in the research stage, and is under development to improve and extend its capabilities. Currently, this feature is available for the literal graph/tree format, which is specified as a list of dictionaries. More on the literal format can be seen here.

```
roundtrip_path = "hatchet/external/roundtrip/"
%load_ext roundtrip
```

(continues on next page)

(continued from previous page)

```
literal_graph = [ ... ]
%loadVisualization roundtrip_path literal_graph
```



Once the user has explored their data, the interactive visualization outputs the corresponding call path query of the selected nodes.

```
%fetchData myQuery
print(myQuery)  # displays [{"name": "corge"}, "*"] for the selection above
```

This query can then be integrated into future workflows to automate the filtering of the data by the desired query in a Python script. For the selection above, we save the resulting query as a string and pass it to Hatchet's `filter()` function to filter the input literal graph. An example code snippet is shown below, with the resulting filtered graph shown on the right.

```
myQuery = [{"name": "corge"}, "*"]
gf = ht.GraphFrame.from_literal(literal_graph)
filter_gf = gf.filter(myQuery)
```

An example notebook of the interactive visualization can be found in the *docs/examples/tutorials* directory.

3.4 Dataframe operations

node	name	nid	node	time	time (inc)
main	main	0	main	40.0	200.0
physics	physics	1	physics	40.0	60.0
mpi	mpi	2	mpi	5.0	20.0
psm2	psm2	3	psm2	15.0	15.0
solvers	solvers	4	solvers	0.0	100.0
hypre	hypre	5	hypre	65.0	65.0
mpi	mpi	6	mpi	10.0	35.0
psm2	psm2	7	psm2	25.0	25.0

filter: `filter` takes a user-supplied function or query object and applies that to all rows in the DataFrame. The resulting Series or DataFrame is used to filter the DataFrame to only return rows that are true. The returned GraphFrame preserves the original graph provided as input to the filter operation.

```
filtered_gf = gf.filter(lambda x: x['time'] > 10.0)
```

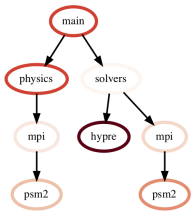
The images on the right show a DataFrame before and after a filter operation.

	name	nid	node	time	time (inc)
node					
main	main	0	main	40.0	200.0
physics	physics	1	physics	40.0	60.0
psm2	psm2	3	psm2	15.0	15.0
hypre	hypre	5	hypre	65.0	65.0
psm2	psm2	7	psm2	25.0	25.0

An alternative way to filter the DataFrame is to supply a query path in the form of a query object. A query object is a list of *abstract graph nodes* that specifies a call path pattern to search for in the GraphFrame. An *abstract graph node* is made up of two parts:

- A wildcard that specifies the number of real nodes to match to the abstract node. This is represented as either a string with value "." (match one node), "*" (match zero or more nodes), or "+" (match one or more nodes) or an integer (match exactly that number of nodes). By default, the wildcard is "." (or 1).
- A filter that is used to determine whether a real node matches the abstract node. In the high-level API, this is represented as a Python dictionary keyed on column names from the DataFrame. By default, the filter is an "always true" filter (represented as an empty dictionary).

The query object is represented as a Python list of abstract nodes. To specify both parts of an abstract node, use a tuple with the first element being the wildcard and the second element being the filter. To use a default value for either the wildcard or the filter, simply provide the other part of the abstract node on its own (no need for a tuple). The user **must** provide at least one of the parts of the above definition of an abstract node.



The query language example below looks for all paths that match first a single node with name *solvers*, followed by 0 or more nodes with an inclusive time greater than 10, followed by a single node with name that starts with *p* and ends in an integer and has an inclusive time greater than or equal to 10. When the query is used to filter and squash the the graph shown on the right, the returned GraphFrame contains the nodes shown in the table on the right.

node	name	nid	node	time	time (inc)
solvers	solvers	4	solvers	0.0	100.0
mpi	mpi	6	mpi	10.0	35.0
psm2	psm2	7	psm2	25.0	25.0

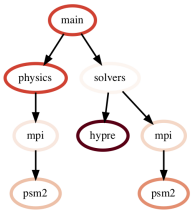
Filter is one of the operations that leads to the graph object and DataFrame object becoming inconsistent. After a filter operation, there are nodes in the graph that do not return any rows when used to index into the DataFrame. Typically, the user will perform a squash on the GraphFrame after a filter operation to make the graph and DataFrame objects consistent again. This can be done either by manually calling the `squash` function on the new GraphFrame or by setting the `squash` parameter of the `filter` function to `True`.

```
query = [
    {"name": "solvers"},
    ("*", {"time (inc)": "> 10"}),
    {"name": "p[a-z]+[0-9]", "time (inc)": ">= 10"}
]

filtered_gf = gf.filter(query)
```

drop_index_levels: When there is per-MPI process or per-thread data in the DataFrame, a user might be interested in aggregating the data in some fashion to analyze the graph at a coarser granularity. This function allows the user to drop the additional index columns in the hierarchical index by specifying an aggregation function. Essentially, this performs a `groupby` and `aggregate` operation on the DataFrame. The user-supplied function is used to perform the aggregation over all MPI processes or threads at the per-node granularity.

```
gf.drop_index_levels(function=np.max)
```

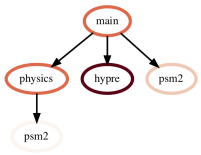
update_inclusive_columns: When a graph is rewired (i.e., the parent-child connections are modified), all the columns in the DataFrame that store inclusive values of a metric become inaccurate. This function performs a post-order traversal of the graph to update all columns that store inclusive metrics in the DataFrame for each node.



3.5 Graph operations

traverse: A generator function that performs a pre-order traversal of the graph and generates a sequence of all nodes in the graph in that order.

squash: The `squash` operation is typically performed by the user after a `filter` operation on the DataFrame. The squash operation removes nodes from the graph that were previously removed from the DataFrame due to a filter operation. When one or more nodes on a path are removed from the graph, the nearest remaining ancestor is connected by an edge to the nearest remaining child on the path. All call paths in the graph are re-wired in this manner.



A squash operation creates a new DataFrame in addition to the new graph. The new DataFrame contains all rows from the original DataFrame, but its index points to nodes in the new graph. Additionally, a squash operation will make the values in all columns containing inclusive metrics inaccurate, since the parent-child relationships have changed. Hence, the squash operation also calls `update_inclusive_columns` to make all inclusive columns in the DataFrame accurate again.

```
filtered_gf = gf.filter(lambda x: x['time'] > 10.0)
squashed_gf = filtered_gf.squash()
```

equal: The == operation checks whether two graphs have the same nodes and edge connectivity when traversing from their roots. If they are equivalent, it returns true, otherwise it returns false.

union: The `union` function takes two graphs and creates a unified graph, preserving all edges structure of the original graphs, and merging nodes with identical context. When Hatchet performs binary operations on two GraphFrames with unequal graphs, a union is performed beforehand to ensure that the graphs are structurally equivalent. This ensures that operands to element-wise operations like add and subtract, can be aligned by their respective nodes.

3.6 GraphFrame operations

copy: The `copy` operation returns a shallow copy of a GraphFrame. It creates a new GraphFrame with a copy of the original GraphFrame's DataFrame, but the same graph. As mentioned earlier, graphs in Hatchet use immutable semantics, and they are copied only when they need to be restructured. This property allows us to reuse graphs from GraphFrame to GraphFrame if the operations performed on the GraphFrame do not mutate the graph.

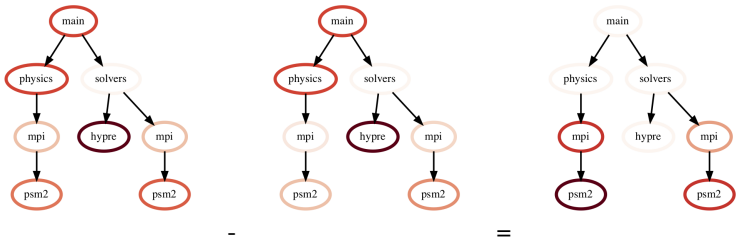
deepcopy: The `deepcopy` operation returns a deep copy of a GraphFrame. It is similar to `copy`, but returns a new GraphFrame with a copy of the original GraphFrame's DataFrame and a copy of the original GraphFrame's graph.

unify: `unify` operates on GraphFrames, and calls union on the two graphs, and then reindexes the DataFrames in both GraphFrames to be indexed by the nodes in the unified graph. Binary operations on GraphFrames call unify which in turn calls union on the respective graphs.

add: Assuming the graphs in two GraphFrames are equal, the `add (+)` operation computes the element-wise sum of two DataFrames. In the case where the two graphs are not identical, `unify` (described above) is applied first to create a unified graph before performing the sum. The DataFrames are copied and reindexed by the combined graph, and the add operation returns new GraphFrame with the result of adding these DataFrames. Hatchet also provides an in-place version of the add operator: +=.

subtract: The subtract operation is similar to the add operation in that it requires the two graphs to be identical. It applies `union` and reindexes DataFrames if necessary. Once the graphs are unified, the subtract operation computes the element-wise difference between the two DataFrames. The subtract operation returns a new GraphFrame, or it modifies one of the GraphFrames in place in the case of the in-place subtraction (-=).

```
gf1 = ht.GraphFrame.from_literal( ... )
gf2 = ht.GraphFrame.from_literal( ... )
gf2 -= gf1
```



tree: The `tree` operation returns the graphframe's graph structure as a string that can be printed to the console. By default, the tree uses the `name` of each node and the associated `time` metric as the string representation. This operation uses automatic color by default, but True or False can be used to force override.

CHAPTER

FOUR

QUERY LANGUAGE

As of version 1.2.0, Hatchet has a filtering query language that allows users to filter GraphFrames based on caller-callee relationships between nodes in the Graph. This query language contains two APIs: a high-level API that is expressed using built-in Python data types (e.g., lists, dictionaries, strings) and a low-level API that is expressed using Python callables.

Regardless of API, queries in Hatchet represent abstract paths, or path patterns, within the Graph being filtered. When filtering on a query, Hatchet will identify all paths in the Graph that match the query. Then, it will return a new GraphFrame object containing only the nodes contained in the matched paths. A query is represented as a list of *abstract graph nodes*. Each *abstract graph node* is made of two parts:

- A wildcard that specifies the number of real nodes to match to the abstract node
- A filter that is used to determine whether a real node matches the abstract node

The primary differences between the two APIs are the representation of filters, how wildcards and filters are combined into *abstract graph nodes*, and how *abstract graph nodes* are combined into a full query.

The following sections will describe the specifications for queries in both APIs and provide examples of how to use the query language.

4.1 High-Level API

The high-level API for Hatchet's query language is designed to allow users to quickly write simple queries. It has a simple syntax based on built-in Python data types (e.g., lists, dictionaries, strings). The following subsections will describe each component of high-level queries. After creating a query, it can be used to filter a GraphFrame by passing it to the `GraphFrame.filter` function as follows:

```
query = <QUERY GOES HERE>
filtered_gf = gf.filter(query)
```

4.1.1 Wildcards

Wildcards in the high-level API are specified by one of four possible values:

- The string `"."`, which means "match 1 node"
- The string `"*"`, which means "match 0 or more nodes"
- The string `"+"`, which means "match 1 or more nodes"
- An integer, which means "match exactly that number of nodes" (integer 1 is equivalent to `"."`)

4.1.2 Filters

Filters in the high-level API are specified by Python dictionaries. These dictionaries are keyed on the names of *node attributes*. These attributes' names are the same as the column names from the DataFrame associated with the GraphFrame being filtered (which can be obtained with `gf.dataframe`). There are also two special attribute names:

- *depth*, which filters on the depth of the node in the Graph
- *node_id*, which filters on the node's unique identifier within the GraphFrame

The values in a high-level API filter dictionary define the conditions that must be passed to pass the filter. Their data types depend on the data type of the corresponding attribute. The table below describes what value data types are valid for different attribute data types.

Attribute Data Type	Example Attributes	Valid Filter Value Types	Description of Condition
Real (integer or float)	*time* *time (inc)*	Real (integer or float)	Attribute value exactly equals filter value
		String starting with comparison operator	Attribute value must pass comparison described in filter value
String	*name*	Regex String (see Python re module for details)	Attribute must match filter value (passed to re.match)

The values in a high-level API filter dictionary can also be iterables (e.g., lists, tuples) of the valid values defined in the table above.

In the high-level API, all conditions (key-value pairs, including conditions contained in a list value) in a filter must pass for the a real node to match the corresponding *abstract graph node*.

4.1.3 Abstract Graph Nodes

In the high-level API, *abstract graph nodes* are represented by Python tuples containing a single wildcard and a single filter. Alternatively, an *abstract graph node* can be represented by only a single . When only providing a wildcard or a filter (and not both), the default is used for the other component. The defaults are as follows:

- Wildcard: `"."` (match 1 node)
- Filter: an "always-true" filter (any node passes this filter)

4.1.4 Full Queries

In the high-level API, a query is represented as a Python list of *abstract graph nodes*. In general, the following code can be used as a template to build a low-level query.

```
query = [
    (wildcard1, query1),
    (wildcard2, query2),
    (wildcard3, query3)
]
filtered_gf = gf.filter(query)
```

4.2 Low-Level API

The low-level API for Hatchet's query language is designed to allow users to perform more complex queries. It's syntax is based on Python callables (e.g., functions, lambdas). The following subsections will describe each component of low-level queries. Like high-level queries, low-level queries can be used to filter a GraphFrame by passing it to the `GraphFrame.filter` function as follows:

```
query = <QUERY GOES HERE>
filtered_gf = gf.filter(query)
```

4.2.1 Wildcards

Wildcards in the low-level API are the exact same as wildcards in the high-level API. The following values are currently allowed for wildcards:

- The string `"."`, which means "match 1 node"
- The string `"*"`, which means "match 0 or more nodes"
- The string `"+"`, which means "match 1 or more nodes"
- An integer, which means "match exactly that number of nodes" (integer 1 is equivalent to `"."`)

4.2.2 Filters

The biggest difference between the high-level and low-level APIs are how filters are represented. In the low-level API, filters are represented by Python callables. These callables should take one argument representing a node in the graph and should return a boolean stating whether or not the node satisfies the filter. The type of the argument to the callable depends on whether the `GraphFrame.drop_index_levels` function was previously called. If this function was called, the type of the argument will be a `pandas.Series`. This `Series` will be the row representing a node in the internal `pandas.DataFrame`. If the `GraphFrame.drop_index_levels` function was not called, the type of the argument will be a `pandas.DataFrame`. This `DataFrame` will contain the rows of the internal `pandas.DataFrame` representing a node. Multiple rows are returned in this case because the internal `DataFrame` will contain one row for every thread and function call.

For example, if you want to match nodes with an exclusive time (represented by "time" column) greater than 2 and an inclusive time (represented by "time (inc)" column) greater than 5, you could use the following filter. This filter assumes you have already called the `GraphFrame.drop_index_levels` function.

```
filter = lambda row: row["time"] > 2 and row["time (inc)"] > 5
```

4.2.3 Abstract Graph Nodes

To build *abstract graph nodes* in the low-level API, you will first need to import Hatchet's `QueryMatcher` class. This can be done with the following import.

```
from hatchet import QueryMatcher
```

The `QueryMatcher` class has two functions that can be used to build *abstract graph nodes*. The first function is `QueryMatcher.match`, which resets the query and constructs a new *abstract graph node* as the root of the query. The second function is `QueryMatcher.rel`, which constructs a new *abstract graph node* and appends it to the query. Both of these functions take two arguments: a wildcard and a low-level filter. If either the filter or wildcard are not provided, the default will be used. The defaults are as follows:

- Wildcard: "." (match 1 node)
- Filter: an "always-true" filter (any node passes this filter)

Both of these functions also return a reference to the `self` parameter of the `QueryMatcher` object. This allows `QueryMatcher.match` and `QueryMatcher.rel` to be chained together.

4.2.4 Full Queries

Full queries in the low-level API are built by making sucessive calls to the `QueryMatcher.match` and `QueryMatcher.rel` functions. In general, the following code can be used as a template to build a low-level query.

```
from hatchet import QueryMatcher

query = QueryMatcher().match(wildcard1, filter1)
    .rel(wildcard2, filter2)
    .rel(wildcard3, filter3)
filtered_gf = gf.filter(query)
```

4.3 Compound Queries

Compound queries is currently a development feature.

Compound queries allow users to apply some operation on the results of one or more queries. Currently, the following compound queries are available directly from `hatchet.query`:

- `AndQuery` and `IntersectionQuery`
- `OrQuery` and `UnionQuery`
- `XorQuery` and `SymDifferenceQuery`

Additionally, the compound query feature provides the following abstract base classes that can be used by users to implement their own compound queries:

- `AbstractQuery`
- `NaryQuery`

The following subsections will describe each of these compound query classes.

4.3.1 AbstractQuery

`AbstractQuery` is an interface (i.e., abstract base class with no implementation) that defines the basic requirements for a query in the Hatchet query language. All query types, including user-created compound queries, must inherit from this class.

4.3.2 NaryQuery

`NaryQuery` is an abstract base class that inherits from `AbstractQuery`. It defines the basic functionality and requirements for compound queries that perform one or more subqueries, collect the results of the subqueries, and performs some subclass defined operation to merge the results into a single result. Queries that inherit from `NaryQuery` must implment the `_perform_nary_op` function, which takes a list of results and should perform some operation on it.

4.3.3 AndQuery

The `AndQuery` class can be used to perform two or more subqueries and compute the intersection of all the returned lists of matched nodes. To create an `AndQuery`, simply create your subqueries (which can be high-level, low-level, or compound), and pass them to the `AndQuery` constructor. The following code can be used as a template for creating an `AndQuery`.

```
from hatchet.query import AndQuery

query1 = <QUERY GOES HERE>
query2 = <QUERY GOES HERE>
query3 = <QUERY GOES HERE>
and_query = AndQuery(query1, query2, query3)
filtered_gf = gf.filter(and_query)
```

`IntersectionQuery` is also provided as an alias (i.e., renaming) of `AndQuery`. The two can be used interchangably.

4.3.4 OrQuery

The `OrQuery` class can be used to perform two or more subqueries and compute the union of all the returned lists of matched nodes. To create an `OrQuery`, simply create your subqueries (which can be high-level, low-level, or compound), and pass them to the `OrQuery` constructor. The following code can be used as a template for creating an `OrQuery`.

```
from hatchet.query import OrQuery

query1 = <QUERY GOES HERE>
query2 = <QUERY GOES HERE>
query3 = <QUERY GOES HERE>
or_query = OrQuery(query1, query2, query3)
filtered_gf = gf.filter(or_query)
```

`UnionQuery` is also provided as an alias (i.e., renaming) of `OrQuery`. The two can be used interchangably.

4.3.5 XorQuery

The `XorQuery` class can be used to perform two or more subqueries and compute the symmetric difference (set theory equivalent to XOR) of all the returned lists of matched nodes. To create an `XorQuery`, simply create your subqueries (which can be high-level, low-level, or compound), and pass them to the `XorQuery` constructor. The following code can be used as a template for creating an `XorQuery`.

```
from hatchet.query import XorQuery

query1 = <QUERY GOES HERE>
```

(continues on next page)

(continued from previous page)

```
query2 = <QUERY GOES HERE>
query3 = <QUERY GOES HERE>
xor_query = XorQuery(query1, query2, query3)
filtered_gf = gf.filter(xor_query)
```

`SymDifferenceQuery` is also provided as an alias (i.e., renaming) of `XorQuery`. The two can be used interchangably.

CHAPTER

FIVE

GENERATING PROFILING DATASETS

5.1 HPCToolkit

HPCToolkit can be installed using Spack or manually. Instructions to build HPCToolkit manually can be found at http://hpctoolkit.org/software-instructions.html.

You can see a basic example of how to use HPCToolkit and generate performance data below.

```
$ mpirun -np <num_ranks> hpcrun <hpcrun_args> ./program.exe <program_args>
```

This command generates a "measurements" directory. Hatchet cannot read this natively and requires another step to generate a "database" directory using `hpcprof-mpi` as described below.

```
$ hpcstruct ./program.exe
$ mpirun -np 1 hpcprof-mpi --metric-db=yes -S ./program.exe.struct -I <path_to_src>
↪<measurements-directory>
```

The first command generates a struct file for the executable `program.exe`. This is provided as one of the arguments in the second command along with pointers to the source code and the generated measurements directory. You must add the `--metric-db=yes` option to `hpcprof-mpi` to generate the database directory in the format recognizable by hatchet.

You can specify the events you want to record as arguments to `hpcrun`. For example: `-e CPUTIME@5000` or `-e PAPI_TOT_CYC@5000000 -e PAPI_TOT_INS -e PAPI_L2_TCM -e PAPI_BR_INS`.

If you want to record data only for the main thread (0) and not for other helper threads, you can set this environment variable: `export HPCRUN_IGNORE_THREAD=1,2,...`

More information information about HPCToolkit can be found at HPCToolkit's documentation page.

5.2 Caliper

Caliper can be installed using Spack or manually from its GitHub repository. Instructions to build Caliper manually can be found in its documentation.

To record performance profiles using Caliper, you need to include `cali.h` and call the `cali_init()` function in your source code. You also need to link the Caliper library in your executable or load it using `LD_PRELOAD`. Information about basic Caliper usage can be found in the Caliper documentation.

To generate profiling data, you can use Caliper's built-in profiling configurations customized for Hatchet: `hatchet-region-profile` or `hatchet-sample-profile`. The former generates a profile based on user annotations in the code while the latter generates a call path profile (similar to HPCToolkit's output). If you

want to use one of the built-in configurations, you should set the `CALI_CONFIG` environment variable (e.g. `CALI_CONFIG=hatchet-sample-profile`).

Alternatively, you can use a custom Caliper .config file (default: caliper.config). If you create your own .config file, you can set the CALI_CONFIG_FILE environment variable to point to it. Two sample caliper.config files are presented below. Other example configuration files can be found in the Caliper GitHub repository.

```
CALI_SERVICES_ENABLE=aggregate,event,mpi,mpireport,timestamp
CALI_EVENT_TRIGGER=annotation,function,loop,mpi.function
CALI_TIMER_SNAPSHOT_DURATION=true
CALI_AGGREGATE_KEY=prop:nested,mpi.rank
CALI_MPI_WHITELIST=MPI_Send,MPI_Recv,MPI_Isend,MPI_Irecv,MPI_Wait,MPI_Waitall,MPI_Bcast,
↪MPI_Reduce,MPI_Allreduce,MPI_Barrier
CALI_MPIREPORT_CONFIG="SELECT annotation,function,loop,mpi.function,mpi.rank,sum(sum
↪#time.duration),inclusive_sum(sum#time.duration) group by mpi.rank,prop:nested format␣
↪json-split"
CALI_MPIREPORT_FILENAME="lulesh-annotation-profile.json"
```

```
CALI_SERVICES_ENABLE=aggregate,callpath,mpi,mpireport,sampler,symbollookup,timestamp
CALI_SYMBOLLOOKUP_LOOKUP_MODULE=true
CALI_TIMER_SNAPSHOT_DURATION=true
CALI_CALIPER_FLUSH_ON_EXIT=false
CALI_SAMPLER_FREQUENCY=200
CALI_CALLPATH_SKIP_FRAMES=4
CALI_AGGREGATE_KEY=callpath.address,cali.sampler.pc,mpi.rank
CALI_MPIREPORT_CONFIG="select source.function#callpath.address,sourceloc#cali.sampler.pc,
↪mpi.rank,sum(sum#time.duration),sum(count),module#cali.sampler.pc group by source.
↪function#callpath.address,sourceloc#cali.sampler.pc,mpi.rank,module#cali.sampler.pc␣
↪format json-split"
CALI_MPIREPORT_FILENAME="cpi-sample-callpathprofile.json"
```

You can read more about Caliper services in the Caliper documentation. Hatchet can read two Caliper outputs: the native .cali files and the split-JSON format (.json files).

5.3 TAU

TAU can be installed using Spack or manually via instructions in its install guide.

You can instrument and/or sample your program using TAU. To instrument your program, you can compile it with `tau_cc.sh` or `tau_cxx.sh` like any other compiler. To sample your program, you can run it with `tau_exec`.

Below, you can find the required environment variables to sample your program and get call path data using TAU. You can both instrument and sample your program using these environment variables and `tau_exec` after compiling your program with `tau_cc/cxx.sh`.

```
TAU_PROFILE=1
TAU_CALLPATH=1
TAU_SAMPLING=1
TAU_CALLPATH_DEPTH=100
TAU_EBS_UNWIND=1
(optional) TAU_METRICS=<TAU/PAPI_metrics>
(optional) PROFILEDIR=<directore_name_for_profile_data>
```

After setting these environment variables, you can run your program as:

```
$ mpirun -np <num_ranks> tau_exec -T mpi,openmp -ebs ./program.exe <program_args>
```

More information about using TAU can be found in its user guide.

5.4 timemory

Timemory can be installed using Spack or manually as suggested in its documentation.

Timemory can perform both runtime instrumentation and binary rewriting, but recommends using binary rewriting for distributed memory parallelism. To use binary rewriting, you need to first generate an instrumented executable and then run that instrumented executable as below.

```
$ timemory-run <timemory-run_options> -o <instrumented_executable> --mpi -- <executable>
$ mpirun -np <num_ranks> ./<instrumented_executable>
```

More information about how to use timemory can be found at https://timemory.readthedocs.io/en/develop/index.html.

5.5 pyinstrument

Hatchet can read pyinstrument JSON files which can be generated

by using its Python API or using the command line:

Command line

```
$ pyinstrument -r json -o <output.json> ./program.py
```

Python API

```
from pyinstrument import Profiler
from pyinstrument.renderers import JSONRenderer
profiler = Profiler()

profiler.start()
# do some work
profiler.stop()

print(JSONRenderer().render(profiler.last_session))
```

CHAPTER

SIX

ANALYSIS EXAMPLES

6.1 Reading different file formats

Hatchet can read in a variety of data file formats into a GraphFrame. Below, we show examples of reading in different data formats.

6.1.1 Read in an HPCToolkit database

A database directory is generated by using `hpcprof-mpi` to post-process the raw measurements directory output by HPCToolkit. To analyze an HPCToolkit database, the `from_hpctoolkit` method can be used.

```
#!/usr/bin/env python
#
# Copyright 2017-2023 Lawrence Livermore National Security, LLC and other
# Hatchet Project Developers. See the top-level LICENSE file for details.
#
# SPDX-License-Identifier: MIT

import hatchet as ht


if __name__ == "__main__":
    # Path to HPCToolkit database directory.
    dirname = "../../../hatchet/tests/data/hpctoolkit-cpi-database"

    # Use hatchet's ``from_hpctoolkit`` API to read in the HPCToolkit database.
    # The result is stored into Hatchet's GraphFrame.
    gf = ht.GraphFrame.from_hpctoolkit(dirname)

    # Printout the DataFrame component of the GraphFrame.
    print(gf.dataframe)

    # Printout the graph component of the GraphFrame.
    # Use "time (inc)" as the metric column to be displayed
    print(gf.tree(metric_column="time (inc)"))
```

6.1.2 Read in a Caliper cali file

Caliper's default raw performance data output is the cali. The cali format can be read by `cali-query`, which transforms the raw data into JSON format.

```
#!/usr/bin/env python
#
# Copyright 2017-2023 Lawrence Livermore National Security, LLC and other
# Hatchet Project Developers. See the top-level LICENSE file for details.
#
# SPDX-License-Identifier: MIT

import hatchet as ht


if __name__ == "__main__":
    # Path to caliper cali file.
    cali_file = (
        "../../../hatchet/tests/data/caliper-lulesh-cali/lulesh-annotation-profile.cali"
    )

    # Setup desired cali query.
    grouping_attribute = "function"
    default_metric = "sum(sum#time.duration),inclusive_sum(sum#time.duration)"
    query = "select function,%s group by %s format json-split" % (
        default_metric,
        grouping_attribute,
    )

    # Use hatchet's ``from_caliper`` API with the path to the cali file and the
    # query. This API will internally run ``cali-query`` on this file to
    # produce a json-split stream. The result is stored into Hatchet's
    # GraphFrame.
    gf = ht.GraphFrame.from_caliper(cali_file, query)

    # Printout the DataFrame component of the GraphFrame.
    print(gf.dataframe)

    # Printout the graph component of the GraphFrame.
    # Use "time (inc)" as the metric column to be displayed
    print(gf.tree(metric_column="time (inc)"))
```

6.1.3 Read in a Caliper JSON stream or file

Caliper's json-split format writes a JSON file with separate fields for Caliper records and metadata. The json-split format is generated by either running `cali-query` on the raw Caliper data or by enabling the `mpireport` service when using Caliper.

JSON Stream

```
#!/usr/bin/env python
#
# Copyright 2017-2023 Lawrence Livermore National Security, LLC and other
# Hatchet Project Developers. See the top-level LICENSE file for details.
#
# SPDX-License-Identifier: MIT

import subprocess
import hatchet as ht


if __name__ == "__main__":
    # Path to caliper cali file.
    cali_file = (
        "../../../hatchet/tests/data/caliper-lulesh-cali/lulesh-annotation-profile.cali"
    )

    # Setup desired cali query.
    cali_query = "cali-query"
    grouping_attribute = "function"
    default_metric = "sum(sum#time.duration),inclusive_sum(sum#time.duration)"
    query = "select function,%s group by %s format json-split" % (
        default_metric,
        grouping_attribute,
    )

    # Use ``cali-query`` here to produce the json-split stream.
    cali_json = subprocess.Popen(
        [cali_query, "-q", query, cali_file], stdout=subprocess.PIPE
    )

    # Use hatchet's ``from_caliper`` API with the resulting json-split.
    # The result is stored into Hatchet's GraphFrame.
    gf = ht.GraphFrame.from_caliper(cali_json.stdout)

    # Printout the DataFrame component of the GraphFrame.
    print(gf.dataframe)

    # Printout the graph component of the GraphFrame.
    # Use "time (inc)" as the metric column to be displayed
    print(gf.tree(metric_column="time (inc)"))
```

JSON File

```
#!/usr/bin/env python
#
# Copyright 2017-2023 Lawrence Livermore National Security, LLC and other
# Hatchet Project Developers. See the top-level LICENSE file for details.
#
# SPDX-License-Identifier: MIT

import hatchet as ht


if __name__ == "__main__":
    # Path to caliper json-split file.
    json_file = "../../../hatchet/tests/data/caliper-cpi-json/cpi-callpath-profile.json"

    # Use hatchet's ``from_caliper`` API with the resulting json-split.
    # The result is stored into Hatchet's GraphFrame.
    gf = ht.GraphFrame.from_caliper(json_file)

    # Printout the DataFrame component of the GraphFrame.
    print(gf.dataframe)

    # Printout the graph component of the GraphFrame.
    # Because no metric parameter is specified, ``time`` is used by default.
    print(gf.tree())
```

6.1.4 Read in a DOT file

The DOT file format is generated by using `gprof2dot` on `gprof` or `callgrind` output.

```
#!/usr/bin/env python
#
# Copyright 2017-2023 Lawrence Livermore National Security, LLC and other
# Hatchet Project Developers. See the top-level LICENSE file for details.
#
# SPDX-License-Identifier: MIT

import hatchet as ht


if __name__ == "__main__":
    # Path to DOT file.
    dot_file = "../../../hatchet/tests/data/gprof2dot-cpi/callgrind.dot.64042.0.1"

    # Use hatchet's ``from_gprof_dot`` API to read in the DOT file. The result
    # is stored into Hatchet's GraphFrame.
    gf = ht.GraphFrame.from_gprof_dot(dot_file)

    # Printout the DataFrame component of the GraphFrame.
    print(gf.dataframe)
```

(continues on next page)

(continued from previous page)

```
    # Printout the graph component of the GraphFrame.
    # Because no metric parameter is specified, ``time`` is used by default.
    print(gf.tree())
```

6.1.5 Read in a DAG literal

The literal format is a list of dictionaries representing a graph with nodes and metrics.

```
#!/usr/bin/env python
# -*- encoding: utf-8 -*-
#
# Copyright 2017-2023 Lawrence Livermore National Security, LLC and other
# Hatchet Project Developers. See the top-level LICENSE file for details.
#
# SPDX-License-Identifier: MIT

import hatchet as ht


if __name__ == "__main__":
    # Define a literal GraphFrame using a list of dicts.
    gf = ht.GraphFrame.from_literal(
        [
            {
                "frame": {"name": "foo"},
                "metrics": {"time (inc)": 135.0, "time": 0.0},
                "children": [
                    {
                        "frame": {"name": "bar"},
                        "metrics": {"time (inc)": 20.0, "time": 5.0},
                        "children": [
                            {
                                "frame": {"name": "baz"},
                                "metrics": {"time (inc)": 5.0, "time": 5.0},
                            },
                            {
                                "frame": {"name": "grault"},
                                "metrics": {"time (inc)": 10.0, "time": 10.0},
                            },
                        ],
                    },
                    {
                        "frame": {"name": "qux"},
                        "metrics": {"time (inc)": 60.0, "time": 0.0},
                        "children": [
                            {
                                "frame": {"name": "quux"},
                                "metrics": {"time (inc)": 60.0, "time": 5.0},
                                "children": [
                                    {
                                        "frame": {"name": "corge"},
```

(continues on next page)

(continued from previous page)


```
                                "metrics": {"time (inc)": 55.0, "time": 10.0},
                                "children": [
                                    {
                                        "frame": {"name": "bar"},
                                        "metrics": {
                                            "time (inc)": 20.0,
                                            "time": 5.0,
                                        },
                                        "children": [
                                            {
                                                "frame": {"name": "baz"},
                                                "metrics": {
                                                    "time (inc)": 5.0,
                                                    "time": 5.0,
                                                },
                                            },
                                            {
                                                "frame": {"name": "grault"},
                                                "metrics": {
                                                    "time (inc)": 10.0,
                                                    "time": 10.0,
                                                },
                                            },
                                        ],
                                    },
                                    {
                                        "frame": {"name": "grault"},
                                        "metrics": {
                                            "time (inc)": 10.0,
                                            "time": 10.0,
                                        },
                                    },
                                    {
                                        "frame": {"name": "garply"},
                                        "metrics": {
                                            "time (inc)": 15.0,
                                            "time": 15.0,
                                        },
                                    },
                                ],
                            }
                        ],
                    }
                ],
            },
            {
                "frame": {"name": "waldo"},
                "metrics": {"time (inc)": 55.0, "time": 0.0},
                "children": [
                    {
                        "frame": {"name": "fred"},
                        "metrics": {"time (inc)": 40.0, "time": 5.0},
```


(continues on next page)

(continued from previous page)


```
                            "children": [
                                {
                                    "frame": {"name": "plugh"},
                                    "metrics": {"time (inc)": 5.0, "time": 5.0},
                                },
                                {
                                    "frame": {"name": "xyzzy"},
                                    "metrics": {"time (inc)": 30.0, "time": 5.0},
                                    "children": [
                                        {
                                            "frame": {"name": "thud"},
                                            "metrics": {
                                                "time (inc)": 25.0,
                                                "time": 5.0,
                                            },
                                            "children": [
                                                {
                                                    "frame": {"name": "baz"},
                                                    "metrics": {
                                                        "time (inc)": 5.0,
                                                        "time": 5.0,
                                                    },
                                                },
                                                {
                                                    "frame": {"name": "garply"},
                                                    "metrics": {
                                                        "time (inc)": 15.0,
                                                        "time": 15.0,
                                                    },
                                                },
                                            ],
                                        }
                                    ],
                                },
                            ],
                        },
                        {
                            "frame": {"name": "garply"},
                            "metrics": {"time (inc)": 15.0, "time": 15.0},
                        },
                    ],
                },
            ],
        },
        {
            "frame": {"name": " (hoge)"},
            "metrics": {"time (inc)": 30.0, "time": 0.0},
            "children": [
                {
                    "frame": {"name": "( (piyo)"},
                    "metrics": {"time (inc)": 15.0, "time": 5.0},
                    "children": [
```


(continues on next page)

(continued from previous page)

```
                            {
                                "frame": {"name": " (fuga)"},
                                "metrics": {"time (inc)": 5.0, "time": 5.0},
                            },
                            {
                                "frame": {"name": " (hogera)"},
                                "metrics": {"time (inc)": 5.0, "time": 5.0},
                            },
                        ],
                    },
                    {
                        "frame": {"name": " (hogehoge)"},
                        "metrics": {"time (inc)": 15.0, "time": 15.0},
                    },
                ],
            },
        ]
    )

    # Printout the DataFrame component of the GraphFrame.
    print(gf.dataframe)

    # Printout the graph component of the GraphFrame.
    # Because no metric parameter is specified, ``time`` is used by default.
    print(gf.tree(metric_column=["time (inc)", "time"]))
```

6.2 Basic Examples

6.2.1 Applying scalar operations to attributes

Individual numeric columns in the dataframe can be scaled or offset by a constant using the native pandas operations. We make a copy of the original graphframe, and modify the dataframe directly. In this example, we offset the `time` column by -2 and scale it by 1/1e7, storing the result in a new column in the dataframe called `scaled time`.

```
gf = ht.GraphFrame.from_hpctoolkit('kripke')
gf.drop_index_levels()

offset = 1e7
gf.dataframe['scaled time'] = (gf.dataframe['time'] / offset) - 2
sorted_df = gf.dataframe.sort_values(by=['scaled time'], ascending=False)
print(sorted_df)
```

node	nid	time	time (inc)	scaled time
{'name': 'Kernel_3d_DGZ::scattering', 'type': 'function'}	60	7.669936e+07	7.896253e+07	5.669936
{'file': '<unknown file> [kripke]', 'line': '0', 'type': 'statement'}	79	6.030476e+07	6.030476e+07	4.030476
{'name': 'Kernel_3d_DGZ::LTimes', 'type': 'function'}	30	5.010439e+07	5.240528e+07	3.010439
{'file': '<unknown file> [kripke]', 'line': '0', 'type': 'statement'}	48	5.005872e+07	5.005872e+07	3.005872
{'name': 'Kernel_3d_DGZ::LPlusTimes', 'type': 'function'}	115	4.947707e+07	5.104498e+07	2.947707
{'file': '<unknown file> [kripke]', 'line': '0', 'type': 'statement'}	139	4.943149e+07	4.943149e+07	2.943149

6.2.2 Generating a flat profile

We can generate a flat profile in hatchet by using the `groupby` functionality in pandas. The flat profile can be based on any categorical column (e.g., function name, load module, file name). We can transform the tree or graph generated by a profiler into a flat profile by specifying the column on which to apply the `groupby` operation and the function to use for aggregation.

In the example below, we apply a pandas `groupby` operation on the `name` column. The time spent in each function is computed using `sum` to aggregate rows in a group. We then display the resulting DataFrame sorted by time.

```
# Read in Kripke HPCToolkit database.
gf = ht.GraphFrame.from_hpctoolkit('kripke')

# Drop all index levels in the DataFrame except ``node``.
gf.drop_index_levels()

# Group DataFrame by ``name`` column, compute sum of all rows in each
# group. This shows the aggregated time spent in each function.
grouped = gf.dataframe.groupby('name').sum()

# Sort DataFrame by ``time`` column in descending order.
sorted_df = grouped.sort_values(by=['time'],
                                ascending=False)

# Display resulting DataFrame.
print(sorted_df)
```

6.2.3 Identifying load imbalance

Hatchet makes it extremely easy to study load imbalance across processes or threads at the per-node granularity (call site or function level). A typical metric to measure imbalance is to look at the ratio of the maximum and average time spent in a code region across all processes.

In this example, we ran LULESH across 512 cores, and are interested in understanding the imbalance across processes. We first perform a `drop_index_levels` operation on the GraphFrame in two different ways: (1) by providing mean as a function in one case, and (2) max as the function to another copy of the DataFrame. This generates two DataFrames, one containing the average time spent in each node, and the other containing the maximum time spent in each node by

name	nid	time	time (inc)
<unknown file> [kripke]:0	17234	1.825282e+08	1.825282e+08
Kernel_3d_DGZ::scattering	60	7.669936e+07	7.896253e+07
Kernel_3d_DGZ::LTimes	30	5.010439e+07	5.240528e+07
Kernel_3d_DGZ::LPlusTimes	115	4.947707e+07	5.104498e+07
Kernel_3d_DGZ::sweep	981	5.018862e+06	5.018862e+06
memset.S:99	3773	3.168982e+06	3.168982e+06
memset.S:101	3970	2.120895e+06	2.120895e+06
Grid_Data::particleEdit	1201	1.131266e+06	1.249157e+06
<unknown file> [libpsm2.so.2.1]:0	324763	9.733415e+05	9.733415e+05
memset.S:98	3767	6.197776e+05	6.197776e+05

Fig. 1: Figure 1: Resulting DataFrame after performing a groupby on the `name` column in this HPCToolkit dataset (only showing a handful of rows for brevity). The DataFrame is sorted in descending order by the `time` column to show the function name with the biggest execution time.

any process. If we divide the corresponding columns of the two DataFrames and look at the nodes with the highest value of the max-to-average ratio, we can identify the nodes with highest imbalance.

```
# Read in LULESH Caliper dataset.
gf1 = ht.GraphFrame.from_caliper('lulesh-512cores')

# Create a copy of the GraphFrame.
gf2 = gf1.copy()

# Drop all index levels in gf1's DataFrame except ``node``, computing the
# average time spent in each node.
gf1.drop_index_levels(function=np.mean)

# Drop all index levels in a copy of gf1's DataFrame except ``node``, this
# time computing the max time spent in each node.
gf2.drop_index_levels(function=np.max)

# Compute the imbalance by dividing the ``time`` column in the max DataFrame
# (i.e., gf2) by the average DataFrame (i.e., gf1). This creates a new column
# called ``imbalance`` in gf1's DataFrame.
gf1.dataframe['imbalance'] = gf2.dataframe['time'].div(gf1.dataframe['time'])

# Sort DataFrame by ``imbalance`` column in descending order.
sorted_df = gf1.dataframe.sort_values(by=['imbalance'], ascending=False)

# Display resulting DataFrame.
print(sorted_df)
```

node	name	nid	time	time (inc)	imbalance
LagrangeNodal	LagrangeNodal	3.0	2.242594e+06	2.593621e+07	2.494720
main	main	0.0	1.106013e+05	5.357208e+07	2.161845
CalcForceForNodes	CalcForceForNodes	4.0	1.033639e+06	2.369361e+07	2.142526
CalcQForElems	CalcQForElems	16.0	3.351894e+06	6.649351e+06	2.037651
CalcEnergyForElems	CalcEnergyForElems	22.0	1.571996e+06	2.807323e+06	2.013174
CalcPressureForElems	CalcPressureForElems	23.0	1.235327e+06	1.235327e+06	2.005437

Fig. 2: Figure 2: Resulting DataFrame showing the imbalance in this Caliper dataset (only showing a handful of rows for brevity). The DataFrame is sorted in descending order by the new `imbalance` column calculated by dividing the max/average time of each function. The function with the highest level of imbalance within a node is `LagrangeNodal` with an imbalance of 2.49.

6.2.4 Comparing multiple executions

An important task in parallel performance analysis is comparing the performance of an application on two different thread counts or process counts. The `filter`, `squash`, and `subtract` operations provided by the Hatchet API can be extremely powerful in comparing profiling datasets from two executions.

In the example below, we ran LULESH at two core counts: 1 core and 27 cores, and wanted to identify the performance changes as one scales on a node. We subtract the GraphFrame at 27 cores from the GraphFrame at 1 core (after dropping the additional index levels), and sort the resulting GraphFrame by execution time.

```
# Read in LULESH Caliper dataset at 1 core.
gf1 = ht.GraphFrame.from_caliper('lulesh-1core.json')

# Read in LULESH Caliper dataset at 27 cores.
gf2 = ht.GraphFrame.from_caliper('lulesh-27cores.json')

# Drop all index levels in gf2's DataFrame except ``node``.
gf2.drop_index_levels()

# Subtract the GraphFrame at 27 cores from the GraphFrame at 1 core, and
# store result in a new GraphFrame.
gf3 = gf2 - gf1

# Sort resulting DataFrame by ``time`` column in descending order.
sorted_df = gf3.dataframe.sort_values(by=['time'], ascending=False)

# Display resulting DataFrame.
print(sorted_df)
```

node	name	nid	time	time (inc)
TimeIncrement	TimeIncrement	25.0	8.505048e+06	8.505048e+06
CalcQForElems	CalcQForElems	16.0	4.455672e+06	5.189453e+06
CalcHourglassControlForElems	CalcHourglassControlForElems	7.0	3.888798e+06	4.755817e+06
LagrangeNodal	LagrangeNodal	3.0	1.986046e+06	8.828475e+06
CalcForceForNodes	CalcForceForNodes	4.0	1.017857e+06	6.842429e+06

Fig. 3: Figure 3: Resulting DataFrame showing the performance differences when running LULESH at 1 core vs. 27 cores (only showing a handful of rows for brevity). The DataFrame sorts the function names in descending order by the `time` column. The `TimeIncrement` has the largest difference in execution time of 8.5e6 as the code scales from 1 to 27 cores.

6.2.5 Filtering by library

Sometimes, users are interested in analyzing how a particular library, such as PetSc or MPI, is used by their application and how the time spent in the library changes as we scale to a larger number of processes.

In this next example, we compare two datasets generated from executions at different numbers of MPI processes. We read in two datasets of LULESH at 27 and 512 MPI processes, respectively, and filter them both on the `name` column by matching the names against `^MPI`. After the filtering operation, we `squash` the DataFrames to generate GraphFrames that just contain the MPI calls from the original datasets. We can now subtract the squashed datasets to identify the biggest offenders.

```
# Read in LULESH Caliper dataset at 27 cores.
gf1 = GraphFrame.from_caliper('lulesh-27cores')

# Drop all index levels in DataFrame except ``node``.
gf1.drop_index_levels()

# Filter GraphFrame by names that start with ``MPI``. This only filters the #
# DataFrame. The Graph and DataFrame are now out of sync.
filtered_gf1 = gf1.filter(lambda x: x['name'].startswith('MPI'))

# Squash GraphFrame, the nodes in the Graph now match what's in the
# DataFrame.
squashed_gf1 = filtered_gf1.squash()

# Read in LULESH Caliper dataset at 512 cores, drop all index levels except
# ``node``, filter and squash the GraphFrame, leaving only nodes that start
# with ``MPI``.
gf2 = GraphFrame.from_caliper('lulesh-512cores')
gf2.drop_index_levels()
filtered_gf2 = gf2.filter(lambda x: x['name'].startswith('MPI'))
squashed_gf2 = filtered_gf2.squash()

# Subtract the two GraphFrames, store the result in a new GraphFrame.
diff_gf = squashed_gf2 - squashed_gf1

# Sort resulting DataFrame by ``time`` column in descending order.
sorted_df = diff_gf.dataframe.sort_values(by=['time'], ascending=False)

# Display resulting DataFrame.
print(sorted_df)
```

	node	time (inc)	name	nid	time
node					
MPI_Allreduce	MPI_Allreduce	2.072371e+06	MPI_Allreduce	3	2.072371e+06
MPI_Finalize	MPI_Finalize	4.042198e+04	MPI_Finalize	0	4.042198e+04
MPI_Isend	MPI_Isend	1.753768e+04	MPI_Isend	15	1.753768e+04
MPI_Isend	MPI_Isend	7.718737e+03	MPI_Isend	13	7.718737e+03
MPI_Isend	MPI_Isend	7.542969e+03	MPI_Isend	7	7.542969e+03
MPI_Waitall	MPI_Waitall	4.573508e+03	MPI_Waitall	5	4.573508e+03
MPI_Barrier	MPI_Barrier	4.240952e+03	MPI_Barrier	12	4.240952e+03

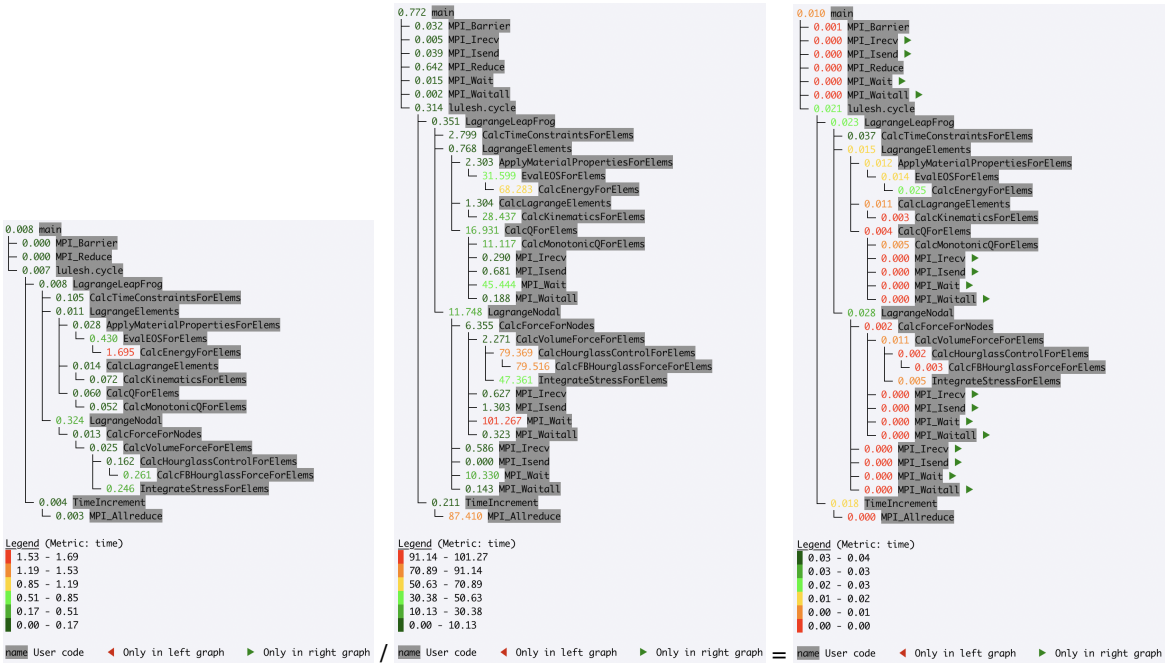
Fig. 4: Figure 4: Resulting DataFrame showing the MPI performance differences when running LULESH at 27 cores vs. 512 cores. The DataFrame sorts the MPI functions in descending order by the `time` column. In this example, the `MPI_Allreduce` function sees the largest increase in time scaling from 27 to 512 cores.

6.3 Scaling Performance Examples

6.3.1 Analyzing strong scaling performance

Hatchet can be used for a strong scaling analysis of applications. In this example, we compare the performance of LULESH running on 1 and 64 cores. By executing a simple `divide` of the two datasets in Hatchet, we can quickly pinpoint bottleneck functions. In the resulting graph, we invert the color scheme, so that functions that did not scale well (i.e., have a low speedup) are colored in red.

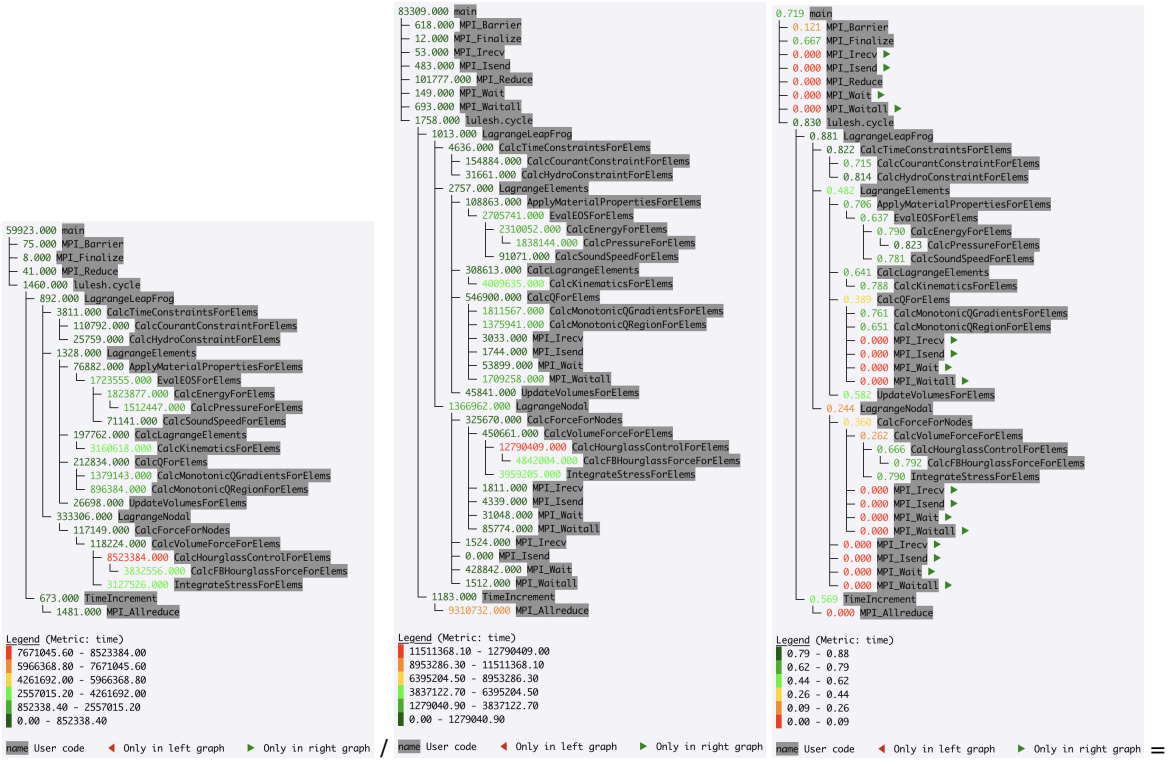
```
gf_1core = ht.GraphFrame.from_caliper('lulesh*-1core.json')
gf_64cores = ht.GraphFrame.from_caliper('lulesh*-64cores.json')

gf_64cores["time"] *= 64

gf_strong_scale = gf_1core / gf_64cores
```



6.3.2 Analyzing weak scaling performance

Hatchet can be used for comparing parallel scaling performance of applications. In this example, we compare the performance of LULESH running on 1 and 27 cores. By executing a simple `divide` of the two datasets in Hatchet, we can quickly identify which function calls did or did not scale well. In the resulting graph, we invert the color scheme, so that functions that did not scale well (i.e., have a low speedup) are colored in red.

```
gf_1core = ht.GraphFrame.from_caliper('lulesh*-1core.json')
gf_27cores = ht.GraphFrame.from_caliper('lulesh*-27cores.json')

gf_weak_scale = gf_1core / gf_27cores
```



6.3.3 Identifying scaling bottlenecks

Hatchet can also be used to analyze data in a weak or strong scaling performance study. In this example, we ran LULESH from 1 to 512 cores on third powers of some numbers. We read in all the datasets into Hatchet, and for each dataset, we use a few lines of code to filter the regions where the code spends most of the time. We then use the pandas' pivot and plot operations to generate a stacked bar chart that shows how the time spent in different regions of LULESH changes as the code scales.

```
# Grab all LULESH Caliper datasets, store in a sorted list.
datasets = glob.glob('lulesh*.json')
datasets.sort()

# For each dataset, create a new GraphFrame, and drop all index levels,
# except ``node``. Insert filtered graphframe into a list.
dataframes = []
for dataset in datasets:
    gf = ht.GraphFrame.from_caliper(dataset)
    gf.drop_index_levels()

    # Grab the number of processes from the file name, store this as a new
    # column in the DataFrame.
    num_pes = re.match('(.*)-(\d+)(.*)', dataset).group(2)
    gf.dataframe['pes'] = num_pes

    # Filter the GraphFrame keeping only those rows with ``time`` greater
    # than 1e6.
    filtered_gf = gf.filter(lambda x: x['time'] > 1e6)
```

(continues on next page)

(continued from previous page)

```
    # Insert the filtered GraphFrame into a list.
    dataframes.append(filtered_gf.dataframe)

# Concatenate all DataFrames into a single DataFrame called ``result``.
result = pd.concat(dataframes)

# Reshape the Dataframe, such that ``pes`` is an index column, ``name``
# fields are the new column names, and the values for each cell is the
# ``time`` fields.
pivot_df = result.pivot(index='pes', columns='name', values='time')

# Make a stacked bar chart using the data in the pivot table above.
pivot_df.loc[:,:].plot.bar(stacked=True, figsize=(10,7))
```

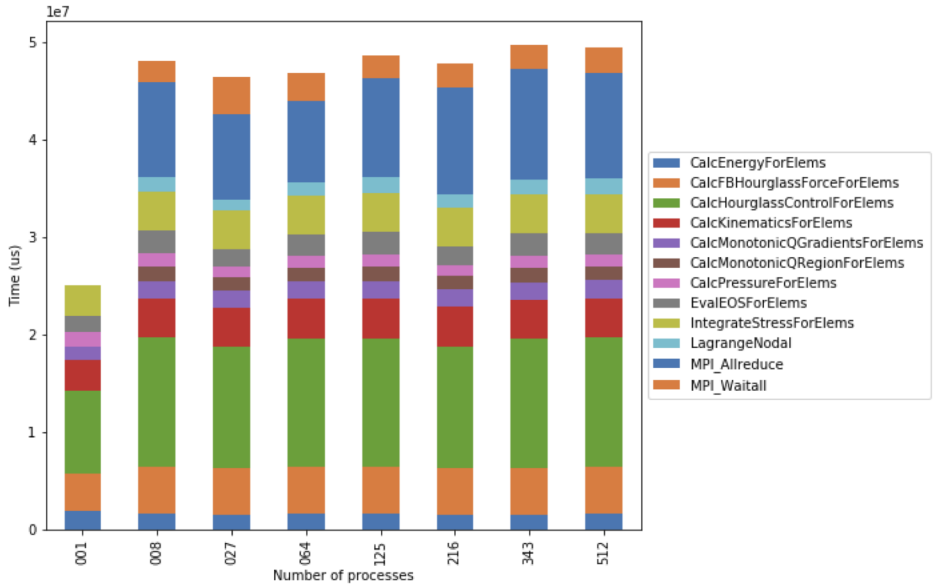


Fig. 5: Figure 5: Resulting stacked bar chart showing the time spent in different functions in LULESH as the code scales from 1 up to 512 processes. In this example, the `CalcHourglassControlForElems` function increases in runtime moving from 1 to 8 processes, then stays constant.

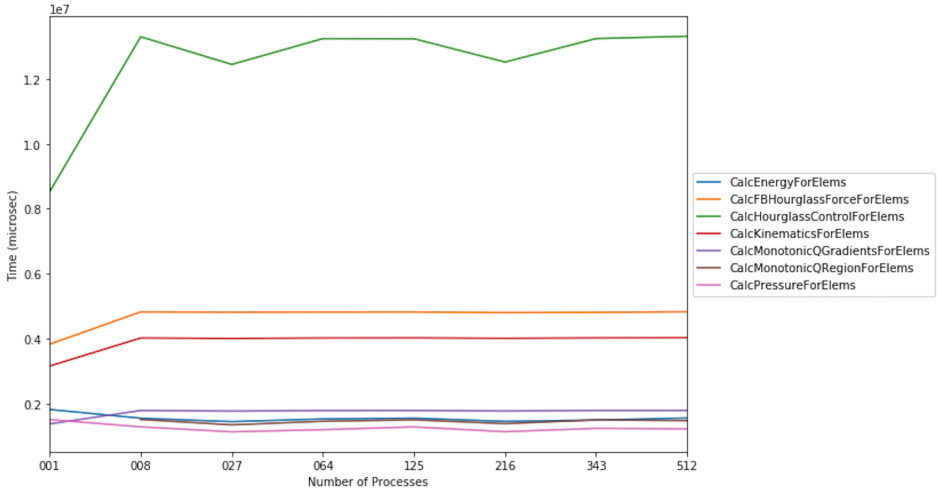
We use the same LULESH scaling datasets above to filter for time-consuming functions that start with the string `Calc`. This data is used to produce a line chart showing the performance of each function as the number of processes is increased. One of the functions (`CalcMonotonicQRegionForElems`) does not occur until the number of processes is greater than 1.

```
datasets = glob.glob('lulesh*.json')
datasets.sort()

dataframes = []
for dataset in datasets:
    gf = ht.GraphFrame.from_caliper(dataset)
    gf.drop_index_levels()

    num_pes = re.match('(.*)-(\d+)(.*)', dataset).group(2)
```

(continues on next page)

(continued from previous page)

```
    gf.dataframe['pes'] = num_pes
    filtered_gf = gf.filter(lambda x: x["time"] > 1e6 and x["name"].startswith('Calc'))
    dataframes.append(filtered_gf.dataframe)

result = pd.concat(dataframes)
pivot_df = result.pivot(index='pes', columns='name', values='time')
pivot_df.loc[:,:].plot.line(figsize=(10, 7))
```



CHAPTER

SEVEN

BASIC TUTORIAL: HATCHET 101

This tutorial introduces how to use hatchet, including basics about:

- Installing hatchet
- Using the pandas API
- Using the hatchet API

7.1 Installing Hatchet and Tutorial Setup

You can install hatchet using pip:

```
$ pip install llnl-hatchet
```

After installing hatchet, you can import hatchet when running the Python interpreter in interactive mode:

```
$ python
Python 3.7.7 (default, Mar 14 2020, 02:39:01)
[Clang 10.0.1 (clang-1001.0.46.4)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Typing `import hatchet` at the prompt should succeed without any error messages:

```
>>> import hatchet as ht
>>>
```

You are good to go!

The Hatchet repository includes stand-alone Python-based Jupyter notebook examples based on this tutorial. You can find them in the hatchet GitHub repository. You can get a local copy of the repository using `git`:

```
$ git clone https://github.com/llnl/hatchet.git
```

You will find the tutorial notebooks in your local hatchet repository under `docs/examples/tutorial/`.

7.2 Introduction

You can read in a dataset into Hatchet for analysis by using one of several `from_` static methods. For example, you can read in a Caliper JSON file as follows:

```
>>> import hatchet as ht
>>> caliper_file = 'lulesh-annotation-profile-1core.json'
>>> gf = ht.GraphFrame.from_caliper(caliper_file)
>>>
```

At this point, your input file (profile) has been loaded into Hatchet's data structure, known as a GraphFrame. Hatchet's GraphFrame contains a pandas DataFrame and a corresponding graph.

The DataFrame component of Hatchet's GraphFrame contains the metrics and other non-numeric data associated with each node in the dataset. You can print the dataframe by typing:

```
>>> print(gf.dataframe)
```

This should produce output like this:

node	rank	time (inc)	time	nid	name
{'name': 'main', 'type': 'region'}	0	27339729.0	59923.0	0	main
{'name': 'MPI_Barrier', 'type': 'region'}	0	75.0	75.0	27	MPI_Barrier
{'name': 'MPI_Finalize', 'type': 'region'}	0	8.0	8.0	21	MPI_Finalize
{'name': 'MPI_Reduce', 'type': 'region'}	0	41.0	41.0	20	MPI_Reduce
{'name': 'lulesh.cycle', 'type': 'region'}	0	27279682.0	1460.0	1	lulesh.cycle
{'name': 'LagrangeLeapFrog', 'type': 'region'}	0	27276068.0	892.0	2	LagrangeLeapFrog
{'name': 'CalcTimeConstraintsForElems', 'type':...	0	140362.0	3811.0	11	CalcTimeConstraintsForElems
{'name': 'CalcCourantConstraintForElems', 'type...	0	110792.0	110792.0	12	CalcCourantConstraintForElems
{'name': 'CalcHydroConstraintForElems', 'type':...	0	25759.0	25759.0	13	CalcHydroConstraintForElems
{'name': 'LagrangeElements', 'type': 'region'}	0	11082669.0	1328.0	9	LagrangeElements
{'name': 'ApplyMaterialPropertiesForElems', 'ty...	0	5207902.0	76882.0	23	ApplyMaterialPropertiesForElems
{'name': 'EvalEOSForElems', 'type': 'region'}	0	5131020.0	1723555.0	24	EvalEOSForElems
{'name': 'CalcEnergyForElems', 'type': 'region'}	0	3336324.0	1823877.0	25	CalcEnergyForElems
{'name': 'CalcPressureForElems', 'type': 'region'}	0	1512447.0	1512447.0	26	CalcPressureForElems
{'name': 'CalcSoundSpeedForElems', 'type': 'reg...	0	71141.0	71141.0	28	CalcSoundSpeedForElems
{'name': 'CalcLagrangeElements', 'type': 'region'}	0	3358380.0	197762.0	14	CalcLagrangeElements
{'name': 'CalcKinematicsForElems', 'type': 'reg...	0	3160618.0	3160618.0	15	CalcKinematicsForElems
{'name': 'CalcQForElems', 'type': 'region'}	0	2488361.0	212834.0	18	CalcQForElems
{'name': 'CalcMonotonicQGradientsForElems', 'ty...	0	1379143.0	1379143.0	19	CalcMonotonicQGradientsForElems
{'name': 'CalcMonotonicQRegionForElems', 'type'...	0	896384.0	896384.0	22	CalcMonotonicQRegionForElems
{'name': 'UpdateVolumesForElems', 'type': 'regi...	0	26698.0	26698.0	10	UpdateVolumesForElems
{'name': 'LagrangeNodal', 'type': 'region'}	0	16052145.0	333306.0	3	LagrangeNodal
{'name': 'CalcForceForNodes', 'type': 'region'}	0	15718839.0	117149.0	4	CalcForceForNodes
{'name': 'CalcVolumeForceForElems', 'type': 're...	0	15601690.0	118224.0	5	CalcVolumeForceForElems
{'name': 'CalcHourglassControlForElems', 'type'...	0	12355940.0	8523384.0	7	CalcHourglassControlForElems
{'name': 'CalcFBHourglassForceForElems', 'type'...	0	3832556.0	3832556.0	8	CalcFBHourglassForceForElems
{'name': 'IntegrateStressForElems', 'type': 're...	0	3127526.0	3127526.0	6	IntegrateStressForElems
{'name': 'TimeIncrement', 'type': 'region'}	0	2154.0	673.0	16	TimeIncrement
{'name': 'MPI_Allreduce', 'type': 'region'}	0	1481.0	1481.0	17	MPI_Allreduce

The Graph component of Hatchet's GraphFrame stores the connections between parents and children. You can print the graph using hatchet's tree printing functionality:

```
>>> print(gf.tree())
```

This will print a graphical version of the tree to the terminal:

```
59923.000 main
├─ 75.000 MPI_Barrier
├─ 8.000 MPI_Finalize
├─ 41.000 MPI_Reduce
└─ 1460.000 lulesh.cycle
   ├─ 892.000 LagrangeLeapFrog
   │  ├─ 3811.000 CalcTimeConstraintsForElems
   │  │  ├─ 110792.000 CalcCourantConstraintForElems
   │  │  └─ 25759.000 CalcHydroConstraintForElems
   │  ├─ 1328.000 LagrangeElements
   │  │  ├─ 76882.000 ApplyMaterialPropertiesForElems
   │  │  │  └─ 1723555.000 EvalEOSForElems
   │  │  │     ├─ 1823877.000 CalcEnergyForElems
   │  │  │     │  └─ 1512447.000 CalcPressureForElems
   │  │  │     └─ 71141.000 CalcSoundSpeedForElems
   │  │  ├─ 197762.000 CalcLagrangeElements
   │  │  │  └─ 3160618.000 CalcKinematicsForElems
   │  │  ├─ 212834.000 CalcQForElems
   │  │  │  ├─ 1379143.000 CalcMonotonicQGradientsForElems
   │  │  │  └─ 896384.000 CalcMonotonicQRegionForElems
   │  │  └─ 26698.000 UpdateVolumesForElems
   │  └─ 333306.000 LagrangeNodal
   │     └─ 117149.000 CalcForceForNodes
   │        └─ 118224.000 CalcVolumeForceForElems
   │           ├─ 8523384.000 CalcHourglassControlForElems
   │           │  └─ 3832556.000 CalcFBHourglassForceForElems
   │           └─ 3127526.000 IntegrateStressForElems
   └─ 673.000 TimeIncrement
      └─ 1481.000 MPI_Allreduce

Legend (Metric: time)
7671046.40 - 8523384.00
5966371.20 - 7671046.40
4261696.00 - 5966371.20
2557020.80 - 4261696.00
852345.60 - 2557020.80
8.00 - 852345.60

name User code    ◄ Only in left graph    ► Only in right graph
```

7.3 Analyzing the DataFrame using pandas

The `DataFrame` is one of two components that makeup the `GraphFrame` in hatchet. The pandas `DataFrame` stores the performance metrics and other non-numeric data for all nodes in the graph.

You can apply any pandas operations to the dataframe in hatchet. Note that modifying the dataframe in hatchet outside of the hatchet API is not recommended because operations that modify the dataframe can make the dataframe and graph inconsistent.

By default, the rows in the dataframe are sorted in traversal order. Sorting the rows by a different column can be done as follows:

```
>>> sorted_df = gf.dataframe.sort_values(by=['time'], ascending=False)
```

Individual numeric columns in the dataframe can be scaled or offset by a constant using native pandas operations. In the following example, we add a new column called `scale` to the existing dataframe, and print the dataframe sorted by this new column from lowest to highest:

```
>>> gf.dataframe['scale'] = gf.dataframe['time'] * 4
>>> sorted_df = gf.dataframe.sort_values(by=['scale'], ascending=True)
```

node	rank	time (inc)	time	nid	name
{'name': 'CalcHourglassControlForElems', 'type'...	0	12355940.0	8523384.0	7	CalcHourglassControlForElems
{'name': 'CalcFBHourglassForceForElems', 'type'...	0	3832556.0	3832556.0	8	CalcFBHourglassForceForElems
{'name': 'CalcKinematicsForElems', 'type': 'reg...	0	3160618.0	3160618.0	15	CalcKinematicsForElems
{'name': 'IntegrateStressForElems', 'type': 're...	0	3127526.0	3127526.0	6	IntegrateStressForElems
{'name': 'CalcEnergyForElems', 'type': 'region'}	0	3336324.0	1823877.0	25	CalcEnergyForElems
{'name': 'EvalEOSForElems', 'type': 'region'}	0	5131020.0	1723555.0	24	EvalEOSForElems
{'name': 'CalcPressureForElems', 'type': 'region'}	0	1512447.0	1512447.0	26	CalcPressureForElems
{'name': 'CalcMonotonicQGradientsForElems', 'ty...	0	1379143.0	1379143.0	19	CalcMonotonicQGradientsForElems
{'name': 'CalcMonotonicQRegionForElems', 'type'...	0	896384.0	896384.0	22	CalcMonotonicQRegionForElems
{'name': 'LagrangeNodal', 'type': 'region'}	0	16052145.0	333306.0	3	LagrangeNodal
{'name': 'CalcQForElems', 'type': 'region'}	0	2488361.0	212834.0	18	CalcQForElems
{'name': 'CalcLagrangeElements', 'type': 'region'}	0	3358380.0	197762.0	14	CalcLagrangeElements
{'name': 'CalcVolumeForceForElems', 'type': 're...	0	15601690.0	118224.0	5	CalcVolumeForceForElems
{'name': 'CalcForceForNodes', 'type': 'region'}	0	15718839.0	117149.0	4	CalcForceForNodes
{'name': 'CalcCourantConstraintForElems', 'type...	0	110792.0	110792.0	12	CalcCourantConstraintForElems
{'name': 'ApplyMaterialPropertiesForElems', 'ty...	0	5207902.0	76882.0	23	ApplyMaterialPropertiesForElems
{'name': 'CalcSoundSpeedForElems', 'type': 'reg...	0	71141.0	71141.0	28	CalcSoundSpeedForElems
{'name': 'main', 'type': 'region'}	0	27339729.0	59923.0	0	main
{'name': 'UpdateVolumesForElems', 'type': 'regi...	0	26698.0	26698.0	10	UpdateVolumesForElems
{'name': 'CalcHydroConstraintForElems', 'type':...	0	25759.0	25759.0	13	CalcHydroConstraintForElems
{'name': 'CalcTimeConstraintsForElems', 'type':...	0	140362.0	3811.0	11	CalcTimeConstraintsForElems
{'name': 'MPI_Allreduce', 'type': 'region'}	0	1481.0	1481.0	17	MPI_Allreduce
{'name': 'lulesh.cycle', 'type': 'region'}	0	27279682.0	1460.0	1	lulesh.cycle
{'name': 'LagrangeElements', 'type': 'region'}	0	11082669.0	1328.0	9	LagrangeElements
{'name': 'LagrangeLeapFrog', 'type': 'region'}	0	27276068.0	892.0	2	LagrangeLeapFrog
{'name': 'TimeIncrement', 'type': 'region'}	0	2154.0	673.0	16	TimeIncrement
{'name': 'MPI_Barrier', 'type': 'region'}	0	75.0	75.0	27	MPI_Barrier
{'name': 'MPI_Reduce', 'type': 'region'}	0	41.0	41.0	20	MPI_Reduce
{'name': 'MPI_Finalize', 'type': 'region'}	0	8.0	8.0	21	MPI_Finalize

node	rank	time (inc)	time	nid	name	scale
{'name': 'MPI_Finalize', 'type': 'region'}	0	8.0	8.0	21	MPI_Finalize	32.0
{'name': 'MPI_Reduce', 'type': 'region'}	0	41.0	41.0	20	MPI_Reduce	164.0
{'name': 'MPI_Barrier', 'type': 'region'}	0	75.0	75.0	27	MPI_Barrier	300.0
{'name': 'TimeIncrement', 'type': 'region'}	0	2154.0	673.0	16	TimeIncrement	2692.0
{'name': 'LagrangeLeapFrog', 'type': 'region'}	0	27276068.0	892.0	2	LagrangeLeapFrog	3568.0
{'name': 'LagrangeElements', 'type': 'region'}	0	11082669.0	1328.0	9	LagrangeElements	5312.0
{'name': 'lulesh.cycle', 'type': 'region'}	0	27279682.0	1460.0	1	lulesh.cycle	5840.0
{'name': 'MPI_Allreduce', 'type': 'region'}	0	1481.0	1481.0	17	MPI_Allreduce	5924.0
{'name': 'CalcTimeConstraintsForElems', 'type':...	0	140362.0	3811.0	11	CalcTimeConstraintsForElems	15244.0
{'name': 'CalcHydroConstraintForElems', 'type':...	0	25759.0	25759.0	13	CalcHydroConstraintForElems	103036.0
{'name': 'UpdateVolumesForElems', 'type': 'regi...	0	26698.0	26698.0	10	UpdateVolumesForElems	106792.0
{'name': 'main', 'type': 'region'}	0	27339729.0	59923.0	0	main	239692.0
{'name': 'CalcSoundSpeedForElems', 'type': 'reg...	0	71141.0	71141.0	28	CalcSoundSpeedForElems	284564.0
{'name': 'ApplyMaterialPropertiesForElems', 'ty...	0	5207902.0	76882.0	23	ApplyMaterialPropertiesForElems	307528.0
{'name': 'CalcCourantConstraintForElems', 'type...	0	110792.0	110792.0	12	CalcCourantConstraintForElems	443168.0
{'name': 'CalcForceForNodes', 'type': 'region'}	0	15718839.0	117149.0	4	CalcForceForNodes	468596.0
{'name': 'CalcVolumeForceForElems', 'type': 're...	0	15601690.0	118224.0	5	CalcVolumeForceForElems	472896.0
{'name': 'CalcLagrangeElements', 'type': 'region'}	0	3358380.0	197762.0	14	CalcLagrangeElements	791048.0
{'name': 'CalcQForElems', 'type': 'region'}	0	2488361.0	212834.0	18	CalcQForElems	851336.0
{'name': 'LagrangeNodal', 'type': 'region'}	0	16052145.0	333306.0	3	LagrangeNodal	1333224.0
{'name': 'CalcMonotonicQRegionForElems', 'type'...	0	896384.0	896384.0	22	CalcMonotonicQRegionForElems	3585536.0
{'name': 'CalcMonotonicQGradientsForElems', 'ty...	0	1379143.0	1379143.0	19	CalcMonotonicQGradientsForElems	5516572.0
{'name': 'CalcPressureForElems', 'type': 'region'}	0	1512447.0	1512447.0	26	CalcPressureForElems	6049788.0
{'name': 'EvalEOSForElems', 'type': 'region'}	0	5131020.0	1723555.0	24	EvalEOSForElems	6894220.0
{'name': 'CalcEnergyForElems', 'type': 'region'}	0	3336324.0	1823877.0	25	CalcEnergyForElems	7295508.0
{'name': 'IntegrateStressForElems', 'type': 're...	0	3127526.0	3127526.0	6	IntegrateStressForElems	12510104.0
{'name': 'CalcKinematicsForElems', 'type': 'reg...	0	3160618.0	3160618.0	15	CalcKinematicsForElems	12642472.0
{'name': 'CalcFBHourglassForceForElems', 'type'...	0	3832556.0	3832556.0	8	CalcFBHourglassForceForElems	15330224.0
{'name': 'CalcHourglassControlForElems', 'type'...	0	12355940.0	8523384.0	7	CalcHourglassControlForElems	34093536.0

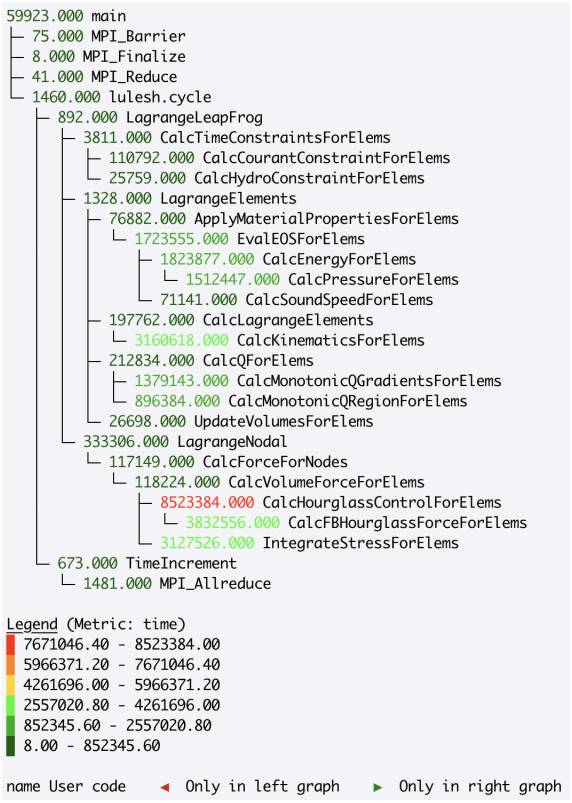
7.4 Analyzing the Graph via printing

Hatchet provides several methods of visualizing graphs. In this section, we show how a user can use the `tree()` method to convert the graph to a string that can be displayed to standard output. This function has several different parameters that can alter the output. To look at all the available parameters, you can look at the docstrings as follows:

```
>>> help(gf.tree)

Help on method tree in module hatchet.graphframe:

tree(metric_column='time', precision=3, name_column='name', expand_name=False,
context_column='file', rank=0, thread=0, depth=10000, highlight_name=False,
invert_colormap=False) method of hatchet.graphframe.GraphFrame instance
    Format this graphframe as a tree and return the resulting string.
```

To print the graph output:

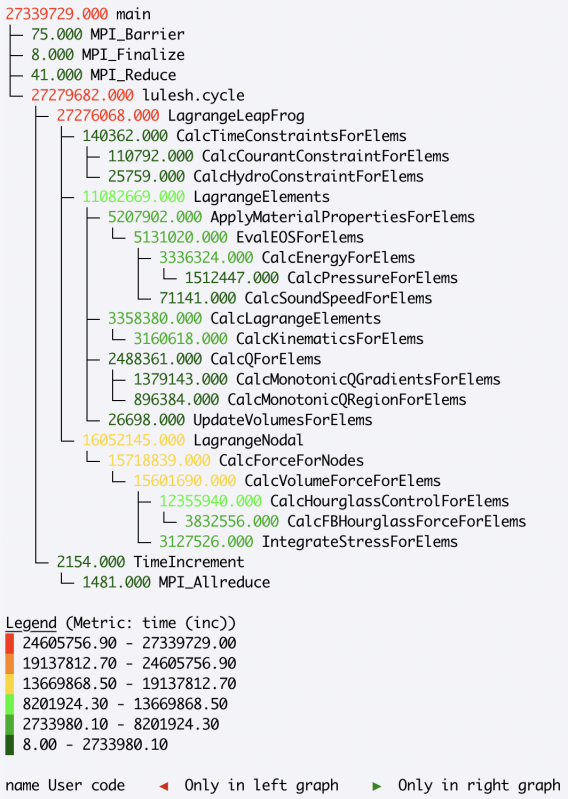
```
>>> gf.tree()
```



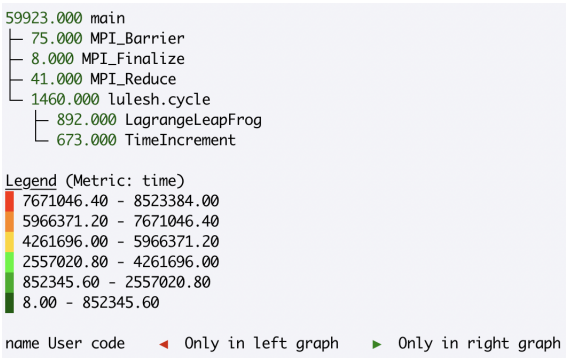
By default, the graph printout displays next to each node values in the `time` column of the dataframe. To display another column, change the argument to the `metric_column=` parameter:

```
>>> gf.tree(metric_column='time (inc)')
```

To view a subset of the nodes in the graph, a user can change the `depth=` value to indicate how many levels of the tree to display. By default, all levels in the tree are displayed. In the following example, we only ask to display the first three levels of the tree, where the root is the first level:




```
>>> gf.tree(depth=3)
```



By default, the `tree()` method uses a red-green colormap, whereby nodes with high metric values are colored red, while nodes with low metric values are colored green. In some use cases, a user may want to reverse the colormap to draw attention to certain nodes, such as performing a division of two graphframes to compute speedup:

```
>>> gf.tree(invert_colormap=True)
```

For a dataset that contains rank- and/or thread-level data, the tree visualization shows the metrics for rank 0 and thread 0 by default. To look at the metrics for a different rank or thread, a user can change the `rank=` or `thread=` parameters:

```
>>> gf.tree(rank=4)
```

```
59923.000 main
├─ 75.000 MPI_Barrier
├─ 8.000 MPI_Finalize
├─ 41.000 MPI_Reduce
└─ 1460.000 lulesh.cycle
   ├─ 892.000 LagrangeLeapFrog
   │  ├─ 3811.000 CalcTimeConstraintsForElems
   │  │  ├─ 110792.000 CalcCourantConstraintForElems
   │  │  └─ 25759.000 CalcHydroConstraintForElems
   │  ├─ 1328.000 LagrangeElements
   │  │  ├─ 76882.000 ApplyMaterialPropertiesForElems
   │  │  │  └─ 1723555.000 EvalEOSForElems
   │  │  │     ├─ 1823877.000 CalcEnergyForElems
   │  │  │     │  └─ 1512447.000 CalcPressureForElems
   │  │  │     └─ 71141.000 CalcSoundSpeedForElems
   │  │  ├─ 197762.000 CalcLagrangeElements
   │  │  │  └─ 3160618.000 CalcKinematicsForElems
   │  │  ├─ 212834.000 CalcQForElems
   │  │  │  ├─ 1379143.000 CalcMonotonicQGradientsForElems
   │  │  │  └─ 896384.000 CalcMonotonicQRegionForElems
   │  │  └─ 26698.000 UpdateVolumesForElems
   │  └─ 333306.000 LagrangeNodal
   │     └─ 117149.000 CalcForceForNodes
   │        └─ 118224.000 CalcVolumeForceForElems
   │           ├─ 8523384.000 CalcHourglassControlForElems
   │           │  └─ 3832556.000 CalcFBHourglassForceForElems
   │           └─ 3127526.000 IntegrateStressForElems
   └─ 673.000 TimeIncrement
      └─ 1481.000 MPI_Allreduce

Legend (Metric: time)
 7671046.40 - 8523384.00
 5966371.20 - 7671046.40
 4261696.00 - 5966371.20
 2557020.80 - 4261696.00
 852345.60 - 2557020.80
 8.00 - 852345.60

name User code    ◂ Only in left graph    ▸ Only in right graph
```

7.5 Analyzing the GraphFrame

Depending on the input data file, the DataFrame may be initialized with one or multiple index levels. In hatchet, the only required index level is `node`, but some readers may also set `rank` and `thread` as additional index levels. The index is a feature of pandas that is used to uniquely identify each row in the Dataframe.

We can query the column names of the index levels as follows:

```
>>> print(gf.dataframe.index.names)
```

This will show the column names of the index levels in a list:

```
['node', 'rank']
```

For this dataset, we see that there are two index columns: `node` and `rank`. Since hatchet requires (at least) `node` to be an index level, we can drop the extra `rank` index level, which will aggregate the data over all MPI ranks at the per-node granularity.

```
>>> gf.drop_index_levels()
>>> print(gf.dataframe)
```

This will aggregate over all MPI ranks and drop all index levels (except `node`).

Now let's imagine we want to focus our analysis on a particular set of nodes. We can filter the GraphFrame by some user-supplied function, which will reduce the number of rows in the DataFrame as well as the number of nodes in the graph. For this example, let's say we are only interested in nodes that start with the name `MPI_`.

```
>>> filt_func = lambda x: x['name'].startswith('MPI_')
>>> filter_gf = gf.filter(filt_func, squash=True)
>>> print(filter_gf.dataframe)
```

node	time (inc)	time	nid	name
{'name': 'main', 'type': 'region'}	27339729.0	59923.0	0	main
{'name': 'MPI_Barrier', 'type': 'region'}	75.0	75.0	27	MPI_Barrier
{'name': 'MPI_Finalize', 'type': 'region'}	8.0	8.0	21	MPI_Finalize
{'name': 'MPI_Reduce', 'type': 'region'}	41.0	41.0	20	MPI_Reduce
{'name': 'lulesh.cycle', 'type': 'region'}	27279682.0	1460.0	1	lulesh.cycle
{'name': 'LagrangeLeapFrog', 'type': 'region'}	27276068.0	892.0	2	LagrangeLeapFrog
{'name': 'CalcTimeConstraintsForElems', 'type':...	140362.0	3811.0	11	CalcTimeConstraintsForElems
{'name': 'CalcCourantConstraintForElems', 'type...	110792.0	110792.0	12	CalcCourantConstraintForElems
{'name': 'CalcHydroConstraintForElems', 'type':...	25759.0	25759.0	13	CalcHydroConstraintForElems
{'name': 'LagrangeElements', 'type': 'region'}	11082669.0	1328.0	9	LagrangeElements
{'name': 'ApplyMaterialPropertiesForElems', 'ty...	5207902.0	76882.0	23	ApplyMaterialPropertiesForElems
{'name': 'EvalEOSForElems', 'type': 'region'}	5131020.0	1723555.0	24	EvalEOSForElems
{'name': 'CalcEnergyForElems', 'type': 'region'}	3336324.0	1823877.0	25	CalcEnergyForElems
{'name': 'CalcPressureForElems', 'type': 'region'}	1512447.0	1512447.0	26	CalcPressureForElems
{'name': 'CalcSoundSpeedForElems', 'type': 'reg...	71141.0	71141.0	28	CalcSoundSpeedForElems
{'name': 'CalcLagrangeElements', 'type': 'region'}	3358380.0	197762.0	14	CalcLagrangeElements
{'name': 'CalcKinematicsForElems', 'type': 'reg...	3160618.0	3160618.0	15	CalcKinematicsForElems
{'name': 'CalcQForElems', 'type': 'region'}	2488361.0	212834.0	18	CalcQForElems
{'name': 'CalcMonotonicQGradientsForElems', 'ty...	1379143.0	1379143.0	19	CalcMonotonicQGradientsForElems
{'name': 'CalcMonotonicQRegionForElems', 'type'...	896384.0	896384.0	22	CalcMonotonicQRegionForElems
{'name': 'UpdateVolumesForElems', 'type': 'regi...	26698.0	26698.0	10	UpdateVolumesForElems
{'name': 'LagrangeNodal', 'type': 'region'}	16052145.0	333306.0	3	LagrangeNodal
{'name': 'CalcForceForNodes', 'type': 'region'}	15718839.0	117149.0	4	CalcForceForNodes
{'name': 'CalcVolumeForceForElems', 'type': 're...	15601690.0	118224.0	5	CalcVolumeForceForElems
{'name': 'CalcHourglassControlForElems', 'type'...	12355940.0	8523384.0	7	CalcHourglassControlForElems
{'name': 'CalcFBHourglassForceForElems', 'type'...	3832556.0	3832556.0	8	CalcFBHourglassForceForElems
{'name': 'IntegrateStressForElems', 'type': 're...	3127526.0	3127526.0	6	IntegrateStressForElems
{'name': 'TimeIncrement', 'type': 'region'}	2154.0	673.0	16	TimeIncrement
{'name': 'MPI_Allreduce', 'type': 'region'}	1481.0	1481.0	17	MPI_Allreduce

This will show a dataframe only containing those nodes that start with `MPI_`:

node	time (inc)	time	nid	name
{'name': 'MPI_Barrier', 'type': 'region'}	75.0	75.0	27	MPI_Barrier
{'name': 'MPI_Finalize', 'type': 'region'}	8.0	8.0	21	MPI_Finalize
{'name': 'MPI_Reduce', 'type': 'region'}	41.0	41.0	20	MPI_Reduce
{'name': 'MPI_Allreduce', 'type': 'region'}	1481.0	1481.0	17	MPI_Allreduce

By default, `filter` will make the graph consistent with the dataframe, so the dataframe and the graph contain the same number of nodes. That is, we specify `squash=True`, so the graph and the dataframe are inconsistent. When we print out the tree, we see that it has the same nodes as the filtered dataframe:

```
1481.000 MPI_Allreduce
75.000 MPI_Barrier
8.000 MPI_Finalize
41.000 MPI_Reduce
```

7.6 Analyzing Multiple GraphFrames

With hatchet, we can perform mathematical operators on multiple GraphFrames. This is useful for comparing the performance of functions at increasing concurrency or computing speedup of two different implementations of the same function, for example.

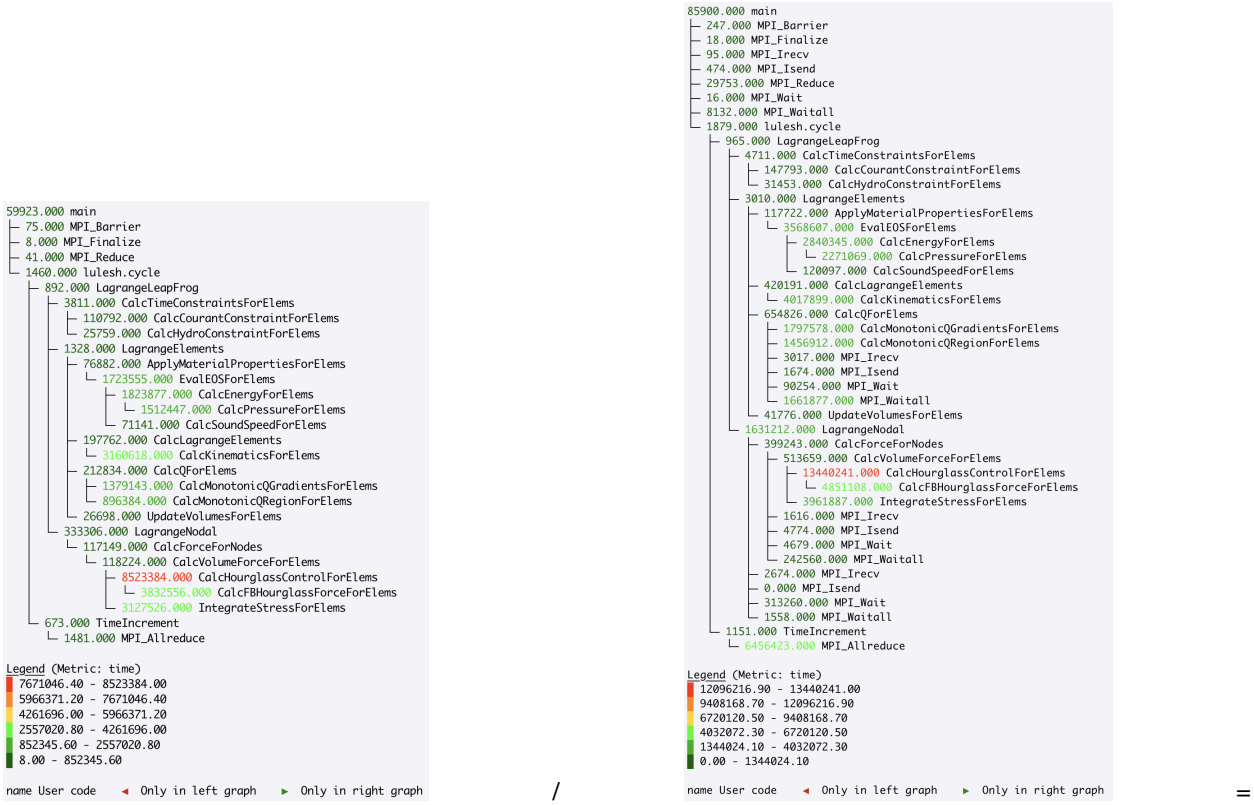
In the example below, we have two LULESH profiles collected at 1 and 64 cores using Caliper. The graphs of these two profiles are slightly different in structure. Due to the scale of the 64 core LULESH run, its profile contains additional MPI-related functions than the 1 core run. With hatchet, we can operate on profiles with different graph structures by first unifying the graphs, and the resulting graph annotates the nodes to indicate which graph the node originated from.

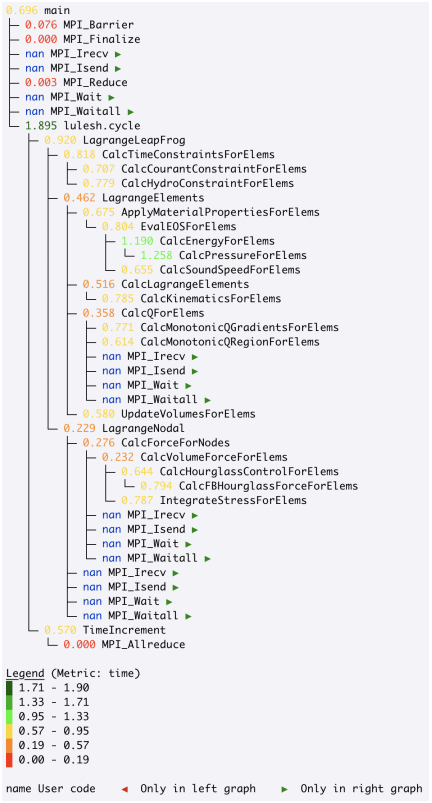
By dividing the profiles, we can analyze how the functions scale at higher concurrencies. Before performing the division operator, we drop the extra `rank` index level in both profiles, which aggregates the data over all MPI ranks at the per-

node granularity. When printing the tree, we specify `invert_colormap=True`, so that nodes with good speedup (i.e., low values) are colored green, while nodes with poor speedup (i.e., high values) are colored red. By default, nodes with low values are colored green, while high values are colored red.

Additionally, because the 64 core profile contained more nodes than the 1 core profile, the resulting tree is annotated with green triangles pointing to the right, indicating that these nodes originally came from the *right* tree (when thinking of gf3 = gf/gf2). In hatchet, those nodes contained in only one of the two trees are initialized with a value of nan, and are colored in blue.

```
>>> caliper_file_1core = 'lulesh-annotation-profile-1core.json'
>>> caliper_file_64cores = 'lulesh-annotation-profile-64cores.json'
>>> gf = ht.GraphFrame.from_caliper(caliper_file_1core)
>>> gf2 = ht.GraphFrame.from_caliper(caliper_file_64cores)
>>> gf.drop_index_levels()
>>> gf2.drop_index_levels()
>>> gf3 = gf/gf2
>>> gf3.tree(invert_colormap=True)
```





If you encounter bugs while using hatchet, you can report them by opening an issue on GitHub.

If you are referencing hatchet in a publication, please cite the following paper:

- Abhinav Bhatele, Stephanie Brink, and Todd Gamblin. Hatchet: Pruning the Overgrowth in Parallel Profiles. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '19). ACM, New York, NY, USA. DOI

CHAPTER

EIGHT

DEVELOPER GUIDE

8.1 Contributing to Hatchet

If you want to contribute a new data reader, feature, or bugfix to Hatchet, please read below. This guide discusses the contributing workflow used in the Hatchet project, and the granularity of pull requests (PRs).

8.1.1 Branches

The main branch in Hatchet that has the latest contributions is named `develop`. All pull requests should start from `develop` and target `develop`.

There is a branch for each minor release series. Release branches originate from `develop` and have tags for each revision release in the series.

8.1.2 Continuous Integration

Hatchet uses GitHub Actions for Continuous Integration testing. This means that every time you submit a pull request, a series of tests are run to make sure you didn't accidentally introduce any bugs into Hatchet. Your PR will not be accepted until it passes all of these tests.

Currently, we perform 2 types of tests:

Unit tests

Unit tests ensure that Hatchet's core API is working as expected. If you add a new data reader or new functionality to the Hatchet API, you should add unit tests that provide adequate coverage for your code. You should also check that your changes pass all unit tests. You can do this by typing:

```
$ pytest
```

Style tests

Hatchet uses Flake8 to test for PEP 8 compliance. You can check for compliance using:

```
$ flake8
```

Hatchet also uses python Black for code formatting. Format your files using:

```
$ black -t <target_python_version> <src>
```

8.1.3 Contributing Workflow

Hatchet is being actively developed, so the `develop` branch in Hatchet has new pull requests being merged often. The recommended way to contribute a pull request is to fork the Hatchet repo in your own space (if you already have a fork, make sure is it up-to-date), and then create a new branch off of `develop`.

We prefer that commits pertaining to different components of Hatchet (specific readers, the core graphframe API, query language, vis tools, etc.) prefix the component name in the commit message (for example `<component>: descriptive message`.

GitHub provides a detailed tutorial on creating pull requests.

CHAPTER

NINE

PUBLICATIONS AND PRESENTATIONS

9.1 Publications

- Stephanie Brink, Ian Lumsden, Connor Scully-Allison, Katy Williams, Olga Pearce, Todd Gamblin, Michela Taufer, Katherine Isaacs, Abhinav Bhatele. Usability and Performance Improvements in Hatchet. Presented at the ProTools 2020 Workshop, held in conjunction with the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '20), held virtually.
- Abhinav Bhatele, Stephanie Brink, and Todd Gamblin. Hatchet: Pruning the Overgrowth in Parallel Profiles. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '19), Denver, CO.

9.2 Posters

- Ian Lumsden. Graph-Based Profiling Analysis using Hatchet. Presented at SC '20. Slides | Video Presentation

9.3 Tutorials

- Automated Performance Analysis with Caliper, SPOT, and Hatchet, ECP Annual Meeting, April 12, 2021. `pdf` | YouTube
- Performance Analysis using Hatchet, LLNL, July 29/31, 2020.

CHAPTER

TEN

HATCHET PACKAGE

10.1 Subpackages

10.1.1 hatchet.cython_modules package

Subpackages

hatchet.cython_modules.libs package

Submodules

hatchet.cython_modules.libs.graphframe_modules module

hatchet.cython_modules.libs.graphframe_modules.**fast_not_isin**(*arr1*, *arr2*, *arr1_len*, *arr2_len*)

Check if elements in arr1 exist in arr2.

A fast check to see if each element in arr1 exists in arr2. This returns a vector mask equivalent to what the operation ~df.isin(...) would return.

Parameters

- **arr1** (*unsigned long long [][]*) – The array of values we are searching for.
- **arr2** (*unsigned long long [][]*) – The sorted array of values we are searching in.

Returns

A boolean mask over arr1 indicating whether each element is or is not in
the function. True indicates that an element is not in arr2.

Return type
(bool [])

hatchet.cython_modules.libs.graphframe_modules.**insert_one_for_self_nodes**(*snio_len*, *self_missing_node*, *snio_indices*)

Adds a '1' where rows are in self but not in other.

hatchet.cython_modules.libs.reader_modules module

hatchet.cython_modules.libs.reader_modules.**subtract_exclusive_metric_vals**(*nid*, *parent_nid*, *metrics*, *num_stmt_nodes*, *stride*)

Module contents

Submodules

hatchet.cython_modules.graphframe_modules module

hatchet.cython_modules.reader_modules module

Module contents

10.1.2 hatchet.external package

Subpackages

hatchet.external.roundtrip package

Subpackages

hatchet.external.roundtrip.roundtrip package

Submodules

hatchet.external.roundtrip.roundtrip.manager module

hatchet.external.roundtrip.roundtrip.version module

Module contents

Submodules

hatchet.external.roundtrip.setup module

Module contents

Submodules

hatchet.external.console module

class hatchet.external.console.**ConsoleRenderer**(*unicode=False*, *color=False*)

Bases: object

colors_disabled = <hatchet.external.console.ConsoleRenderer.colors_disabled object>

class colors_enabled

Bases: object

bg_white_255 = '\x1b[48;5;246m'

blue = '\x1b[34m'

colormap = []

cyan = '\x1b[36m'

dark_gray_255 = '\x1b[38;5;232m'

end = '\x1b[0m'

faint = '\x1b[2m'

left = '\x1b[38;5;160m'

right = '\x1b[38;5;28m'

render(*roots*, *dataframe*, ***kwargs*)

render_frame(*node*, *dataframe*, *indent=''*, *child_indent=''*)

render_legend()

render_preamble()

Module contents

10.1.3 hatchet.query package

Submodules

hatchet.query.compat module

class hatchet.query.compat.**AbstractQuery**

Bases: ABC

Base class for all 'old-style' queries.

abstract apply(*gf*)

class hatchet.query.compat.**AndQuery**(**args*)

Bases: *NaryQuery*

Compound query that returns the intersection of the results of the subqueries.

class hatchet.query.compat.**CypherQuery**(*cypher_query*)

Bases: *QueryMatcher*

Processes and applies Strinb-based queries to GraphFrames.

hatchet.query.compat.**IntersectionQuery**

alias of *AndQuery*

class hatchet.query.compat.**NaryQuery**(**args*)

Bases: *AbstractQuery*

Base class for all compound queries that act on and merged N separate subqueries.

apply(*gf*)

Applies the query to the specified GraphFrame.

Parameters
gf (GraphFrame) – the GraphFramme on which to apply the query

Results:
(list): A list of nodes representing the result of the query

class hatchet.query.compat.**NotQuery**(**args*)

Bases: *NaryQuery*

Compound query that returns all nodes in the GraphFrame that are not returned from the subquery.

class hatchet.query.compat.**OrQuery**(**args*)

Bases: *NaryQuery*

Compound query that returns the union of the results of the subqueries

class hatchet.query.compat.**QueryMatcher**(*query=None*)

Bases: *AbstractQuery*

Processes and applies base syntax queries and Object-based queries to GraphFrames.

apply(*gf*)

Apply the query to a GraphFrame.

Parameters
gf (GraphFrame) – the GraphFrame on which to apply the query

Returns
A list representing the set of nodes from paths that match this query

Return type
(list)

match(*wildcard_spec='.', filter_func=<function QueryMatcher.<lambda>>*)

Start a query with a root node described by the arguments.

Parameters

- **wildcard_spec** (*str, optional*) – the wildcard status of the node
- **filter_func** (*Callable, optional*) – a callable acceepting only a row from a pandas DataFrame that is used to filter this node in the query

Returns
the instance of the class that called this function

Return type
(*QueryMatcher*)

rel(*wildcard_spec='.', filter_func=<function QueryMatcher.<lambda>>*)

Add another edge and node to the query.

Parameters

- **wildcard_spec** (*str, optional*) – the wildcard status of the node

- **filter_func** (*Callable, optional*) – a callable acceepting only a row from a pandas DataFrame that is used to filter this node in the query

Returns
the instance of the class that called this function

Return type
(*QueryMatcher*)

hatchet.query.compat.**SymDifferenceQuery**

alias of *XorQuery*

hatchet.query.compat.**UnionQuery**

alias of *OrQuery*

class hatchet.query.compat.**XorQuery**(**args*)

Bases: *NaryQuery*

Compound query that returns the symmetric difference (i.e., set-based XOR) of the results of the subqueries

hatchet.query.compat.**parse_cypher_query**(*cypher_query*)

Parse all types of String-based queries, including multi-queries that leverage the curly brace delimiters.

Parameters
cypher_query (*str*) – the String-based query to be parsed

Returns
a Hatchet query for this String-based query

Return type
(*CypherQuery*)

hatchet.query.compound module

class hatchet.query.compound.**CompoundQuery**(**queries*)

Bases: `object`

Base class for all types of compound queries.

class hatchet.query.compound.**ConjunctionQuery**(**queries*)

Bases: *CompoundQuery*

A compound query that combines the results of its subqueries using set conjunction.

class hatchet.query.compound.**DisjunctionQuery**(**queries*)

Bases: *CompoundQuery*

A compound query that combines the results of its subqueries using set disjunction.

class hatchet.query.compound.**ExclusiveDisjunctionQuery**(**queries*)

Bases: *CompoundQuery*

A compound query that combines the results of its subqueries using exclusive set disjunction.

class hatchet.query.compound.**NegationQuery**(**queries*)

Bases: *CompoundQuery*

A compound query that inverts/negates the result of its single subquery.

hatchet.query.engine module

class hatchet.query.engine.**QueryEngine**

Bases: object

Class for applying queries to GraphFrames.

apply(*query*, *graph*, *dframe*)

Apply the query to a GraphFrame.

Parameters

- **query** (Query *or* CompoundQuery) – the query being applied
- **graph** (Graph) – the Graph to which the query is being applied
- **dframe** (*pandas.DataFrame*) – the DataFrame associated with the graph

Returns
A list representing the set of nodes from paths that match the query

Return type
(list)

reset_cache()

Resets the cache in the QueryEngine.

hatchet.query.errors module

exception hatchet.query.errors.**BadNumberNaryQueryArgs**

Bases: Exception

Raised when a query filter does not have a valid syntax

exception hatchet.query.errors.**InvalidQueryFilter**

Bases: Exception

Raised when a query filter does not have a valid syntax

exception hatchet.query.errors.**InvalidQueryPath**

Bases: Exception

Raised when a query does not have the correct syntax

exception hatchet.query.errors.**MultiIndexModeMismatch**

Bases: Exception

Raised when an ObjectQuery or StringQuery object is set to use multi-indexed data, but no multi-indexed data is provided

exception hatchet.query.errors.**RedundantQueryFilterWarning**

Bases: Warning

Warned when a query filter does nothing or is redundant

hatchet.query.object_dialect module

class hatchet.query.object_dialect.**ObjectQuery**(*query*, *multi_index_mode='off'*)

Bases: *Query*

Class for representing and parsing queries using the Object-based dialect.

hatchet.query.query module

class hatchet.query.query.**Query**

Bases: object

Class for representing and building Hatchet Call Path Queries

match(*quantifier='.'*, *predicate=<function Query.<lambda>>*)

Start a query with a root node described by the arguments.

Parameters

- **quantifier** (*".", "*", "+", or int, optional*) – the quantifier for this node (tells how many graph nodes to match)
- **predicate** (*Callable, optional*) – the predicate for this node (used to determine whether a graph node matches this query node)

Returns
returns self so that this method can be chained with subsequent calls to "rel"/"relation"

Return type
(*Query*)

rel(*quantifier='.'*, *predicate=<function Query.<lambda>>*)

Add a new node to the end of the query.

Parameters

- **quantifier** (*".", "*", "+", or int, optional*) – the quantifier for this node (tells how many graph nodes to match)
- **predicate** (*Callable, optional*) – the predicate for this node (used to determine whether a graph node matches this query node)

Returns
returns self so that this method can be chained with subsequent calls to "rel"/"relation"

Return type
(*Query*)

relation(*quantifer='.'*, *predicate=<function Query.<lambda>>*)

Alias to Query.rel. Add a new node to the end of the query.

Parameters

- **quantifier** (*".", "*", "+", or int, optional*) – the quantifier for this node (tells how many graph nodes to match)
- **predicate** (*Callable, optional*) – the predicate for this node (used to determine whether a graph node matches this query node)

Returns
returns self so that this method can be chained with subsequent calls to "rel"/"relation"

Return type
(*Query*)

hatchet.query.string_dialect module

class hatchet.query.string_dialect.**StringQuery**(*cypher_query*, *multi_index_mode='off'*)

Bases: *Query*

Class for representing and parsing queries using the String-based dialect.

hatchet.query.string_dialect.**cname**(*obj*)

Utility function to get the name of the rule represented by the input

hatchet.query.string_dialect.**filter_check_types**(*type_check*, *df_row*, *filt_lambda*)

Utility function used in String-based predicates
to make sure the node data used in the actual boolean predicate is of the correct type.

Parameters

- **type_check** (*str*) – a string containing a boolean Python expression used to validate node data typing
- **df_row** (*pandas.Series or pandas.DataFrame*) – the row (or sub-DataFrame) representing the data for the current node being tested
- **filt_lambda** (*Callable*) – the lambda used to actually confirm whether the node satisfies the predicate

Returns
True if the node satisfies the predicate. False otherwise

Return type
(bool)

hatchet.query.string_dialect.**parse_string_dialect**(*query_str*, *multi_index_mode='off'*)

Parse all types of String-based queries, including multi-queries that leverage the curly brace delimiters.

Parameters
query_str (*str*) – the String-based query to be parsed

Returns
A Hatchet query object representing the String-based query

Return type
(*Query* or *CompoundQuery*)

Module contents

hatchet.query.**combine_via_conjunction**(*query0*, *query1*)

hatchet.query.**combine_via_disjunction**(*query0*, *query1*)

hatchet.query.**combine_via_exclusive_disjunction**(*query0*, *query1*)

hatchet.query.**is_hatchet_query**(*query_obj*)

hatchet.query.**negate_query**(*query*)

10.1.4 hatchet.readers package

Submodules

hatchet.readers.caliper_native_reader module

class hatchet.readers.caliper_native_reader.**CaliperNativeReader**(*filename_or_caliperreader*, *native*, *string_attributes*)

Bases: object

Read in a native *.cali* file using Caliper's python reader.

create_graph(*ctx='path'*)

read()

Read the caliper records to extract the calling context tree.

read_metrics(*ctx='path'*)

hatchet.readers.caliper_reader module

class hatchet.readers.caliper_reader.**CaliperReader**(*filename_or_stream*, *query=''*)

Bases: object

Read in a Caliper file (*cali* or split JSON) or file-like object.

create_graph()

read()

Read the caliper JSON file to extract the calling context tree.

read_json_sections()

hatchet.readers.cprofile_reader module

class hatchet.readers.cprofile_reader.**CProfileReader**(*filename*)

Bases: object

create_graph()

Performs the creation of our node graph

read()

class hatchet.readers.cprofile_reader.**NameData**

Bases: object

Faux Enum for python

FILE = 0

FNCNAME = 2

LINE = 1

class hatchet.readers.cprofile_reader.**StatData**

Bases: object

Faux Enum for python

EXCTIME = 2

INCTIME = 3

NATIVECALLS = 1

NUMCALLS = 0

SRCNODE = 4

hatchet.readers.cprofile_reader.**print_incomptable_msg**(*stats_file*)

Function which makes the syntax cleaner in Profiler.write_to_file().

hatchet.readers.dataframe_reader module

class hatchet.readers.dataframe_reader.**DataframeReader**(*filename*)

Bases: ABC

Abstract Base Class for reading in checkpointing files.

read(***kwargs*)

exception hatchet.readers.dataframe_reader.**InvalidDataFrameIndex**

Bases: Exception

Raised when the DataFrame index is of an invalid type.

hatchet.readers.gprof_dot_reader module

class hatchet.readers.gprof_dot_reader.**GprofDotReader**(*filename*)

Bases: object

Read in gprof/callgrind output in dot format generated by gprof2dot.

create_graph()

Read the DOT files to create a graph.

read()

Read the DOT file generated by gprof2dot to create a graphframe. The DOT file contains a call graph.

hatchet.readers.hdf5_reader module

class hatchet.readers.hdf5_reader.**HDF5Reader**(*filename*)

Bases: *DataframeReader*

hatchet.readers.hpctoolkit_reader module

class hatchet.readers.hpctoolkit_reader.**HPCToolkitReader**(*dir_name*)

Bases: object

Read in the various sections of an HPCToolkit experiment.xml file and metric-db files.

count_cpu_threads_per_rank()

create_node_dict(*nid*, *hnode*, *name*, *node_type*, *src_file*, *line*, *module*)

Create a dict with all the node attributes.

fill_tables()

Read certain sections of the experiment.xml file to create dicts of load modules, src_files, procedure_names, and metric_names.

parse_xml_children(*xml_node*, *hnode*)

Parses all children of an XML node.

parse_xml_node(*xml_node*, *parent_nid*, *parent_line*, *hparent*)

Parses an XML node and its children recursively.

read()

Read the experiment.xml file to extract the calling context tree and create a dataframe out of it. Then merge the two dataframes to create the final dataframe.

Returns
new GraphFrame with HPCToolkit data.

Return type
(*GraphFrame*)

read_all_metricdb_files()

Read all the metric-db files and create a dataframe with num_nodes X num_metricdb_files rows and num_metrics columns. Three additional columns store the node id, MPI process rank, and thread id (if applicable).

hatchet.readers.hpctoolkit_reader.**init_shared_array**(*buf_*)

Initialize shared array.

hatchet.readers.hpctoolkit_reader.**read_metricdb_file**(*args*)

Read a single metricdb file into a 1D array.

hatchet.readers.json_reader module

class hatchet.readers.json_reader.**JsonReader**(*json_spec*)

Bases: object

Create a GraphFrame from a json string of the following format.

Returns
graphframe containing data from dictionaries

Return type
(*GraphFrame*)

read()

hatchet.readers.literal_reader module

class hatchet.readers.literal_reader.**LiteralReader**(*graph_dict*)

Bases: object

Create a GraphFrame from a list of dictionaries.

TODO: calculate inclusive metrics automatically.

Example:

```
dag_ldict = [
    {
        "frame": {"name": "A", "type": "function"},
        "metrics": {"time (inc)": 30.0, "time": 0.0},
        "children": [
            {
                "frame": {"name": "B",  "type": "function"},
                "metrics": {"time (inc)": 11.0, "time": 5.0},
                "children": [
                    {
                        "frame": {"name": "C", "type": "function"},
                        "metrics": {"time (inc)": 6.0, "time": 5.0},
                        "children": [
                            {
                                "frame": {"name": "D", "type": "function"},
                                "metrics": {"time (inc)": 1.0, "time": 1.0},
                            }
                        ],
                    }
                ],
            },
            {
                "frame": {"name": "E", "type": "function"},
                "metrics": {"time (inc)": 19.0, "time": 10.0},
                "children": [
                    {
                        "frame": {"name": "H", "type": "function"},
                        "metrics": {"time (inc)": 9.0, "time": 9.0}
                    }
                ],
            },
        ],
    }
]
```

Returns
graphframe containing data from dictionaries

Return type
(*GraphFrame*)

parse_node_literal(*frame_to_node_dict*, *node_dicts*, *child_dict*, *hparent*, *seen_nids*)

Create node_dict for one node and then call the function recursively on all children.

read()

hatchet.readers.pyinstrument_reader module

class hatchet.readers.pyinstrument_reader.**PyinstrumentReader**(*filename*)

Bases: object

create_graph()

read()

hatchet.readers.spotdb_reader module

class hatchet.readers.spotdb_reader.**SpotDBReader**(*db_key*, *list_of_ids=None*, *default_metric='Total time (inc)'*)

Bases: object

Import multiple runs as graph frames from a SpotDB instance

read()

Read given runs from SpotDB

Returns

List of GraphFrames, one for each entry that was found

class hatchet.readers.spotdb_reader.**SpotDatasetReader**(*regionprofile*, *metadata*, *attr_info*)

Bases: object

Reads a (single-run) dataset from SpotDB

create_graph()

Create the graph. Fills in df_data and metric_columns.

read(*default_metric='Total time (inc)'*)

Create GraphFrame for the given Spot dataset.

hatchet.readers.tau_reader module

class hatchet.readers.tau_reader.**TAUReader**(*dirname*)

Bases: object

Read in a profile generated using TAU.

create_graph()

create_node_dict(*node*, *columns*, *metric_values*, *name*, *filename*, *module*, *start_line*, *end_line*, *rank*, *thread*)

read()

Read the TAU profile file to extract the calling context tree.

hatchet.readers.timemory_reader module

class hatchet.readers.timemory_reader.**TimemoryReader**(*input*, *select=None*, ***_kwargs*)

Bases: object

Read in timemory JSON output

create_graph()

Create graph and dataframe

read()

Read timemory json.

Module contents

10.1.5 hatchet.util package

Submodules

hatchet.util.colormaps module

class hatchet.util.colormaps.**ColorMaps**

Bases: object

BrBG = ['\x1b[38;5;94m', '\x1b[38;5;179m', '\x1b[38;5;222m', '\x1b[38;5;116m', '\x1b[38;5;37m', '\x1b[38;5;23m']

PRGn = ['\x1b[38;5;90m', '\x1b[38;5;140m', '\x1b[38;5;183m', '\x1b[38;5;151m', '\x1b[38;5;70m', '\x1b[38;5;22m']

PiYG = ['\x1b[38;5;162m', '\x1b[38;5;176m', '\x1b[38;5;219m', '\x1b[38;5;149m', '\x1b[38;5;70m', '\x1b[38;5;22m']

PuOr = ['\x1b[38;5;130m', '\x1b[38;5;208m', '\x1b[38;5;220m', '\x1b[38;5;189m', '\x1b[38;5;104m', '\x1b[38;5;57m']

RdBu = ['\x1b[38;5;124m', '\x1b[38;5;209m', '\x1b[38;5;224m', '\x1b[38;5;153m', '\x1b[38;5;75m', '\x1b[38;5;25m']

RdGy = ['\x1b[38;5;124m', '\x1b[38;5;209m', '\x1b[38;5;223m', '\x1b[38;5;251m', '\x1b[38;5;244m', '\x1b[38;5;238m']

RdYlBu = ['\x1b[38;5;196m', '\x1b[38;5;208m', '\x1b[38;5;220m', '\x1b[38;5;153m', '\x1b[38;5;68m', '\x1b[38;5;24m']

RdYlGn = ['\x1b[38;5;196m', '\x1b[38;5;208m', '\x1b[38;5;220m', '\x1b[38;5;46m', '\x1b[38;5;34m', '\x1b[38;5;22m']

Spectral = ['\x1b[38;5;196m', '\x1b[38;5;208m', '\x1b[38;5;220m', '\x1b[38;5;191m', '\x1b[38;5;114m', '\x1b[38;5;26m']

get_colors(*colormap*, *invert_colormap*)

Returns a list of colors based on the colormap and invert_colormap arguments.

hatchet.util.config module

hatchet.util.deprecated module

hatchet.util.deprecated.**deprecated_params**(***old_to_new*)

hatchet.util.deprecated.**rename_kwargs**(*fname*, *old_to_new*, *kwargs*)

hatchet.util.dot module

hatchet.util.dot.**to_dot**(*hnode*, *dataframe*, *metric*, *name*, *rank*, *thread*, *threshold*, *visited*)

Write to graphviz dot format.

hatchet.util.dot.**trees_to_dot**(*roots*, *dataframe*, *metric*, *name*, *rank*, *thread*, *threshold*)

Calls to_dot in turn for each tree in the graph/forest.

hatchet.util.executable module

hatchet.util.executable.**which**(*executable*)

Finds an *executable* in the user's PATH like command-line which.

Parameters

executable (*str*) – executable to search for

hatchet.util.profiler module

class hatchet.util.profiler.**Profiler**

Bases: object

Wrapper class around cProfile. Exports a pstats file to be read by the pstats reader.

reset()

Description: Resets the profilier.

start()

Description: Place before the block of code to be profiled.

stop()

Description: Place at the end of the block of code being profiled.

write_to_file(*filename=''*, *add_pstats_files=[]*)

Description: Write the pstats object to a binary file to be read in by an appropriate source.

hatchet.util.profiler.**print_incomptable_msg**(*stats_file*)

Function which makes the syntax cleaner in Profiler.write_to_file().

hatchet.util.timer module

class hatchet.util.timer.**Timer**

Bases: object

Simple phase timer with a context manager.

end_phase()

phase(*name*)

start_phase(*phase*)

Module contents

10.1.6 hatchet.vis package

Subpackages

hatchet.vis.external package

Module contents

Submodules

hatchet.vis.loader module

hatchet.vis.static_fixer module

Module contents

10.1.7 hatchet.writers package

Submodules

hatchet.writers.dataframe_writer module

class hatchet.writers.dataframe_writer.**DataframeWriter**(*filename*)

Bases: ABC

write(*gf*, ***kwargs*)

exception hatchet.writers.dataframe_writer.**InvalidDataFrameIndex**

Bases: Exception

Raised when the DataFrame index is of an invalid type.

hatchet.writers.hdf5_writer module

class hatchet.writers.hdf5_writer.**HDF5Writer**(*filename*)

Bases: *DataframeWriter*

Module contents

10.2 Submodules

10.3 hatchet.frame module

class hatchet.frame.**Frame**(*attrs=None*, ***kwargs*)

Bases: object

The frame index for a node. The node only stores its frame.

Parameters
attrs (*dict*) – dictionary of attributes and values

copy()

get(*name*, *default=None*)

property tuple_repr

Make a tuple of attributes and values based on reader.

values(*names*)

Return a tuple of attribute values from this Frame.

10.4 hatchet.graph module

class hatchet.graph.**Graph**(*roots*)

Bases: object

A possibly multi-rooted tree or graph from one input dataset.

copy(*old_to_new=None*)

Create and return a copy of this graph.

Parameters
old_to_new (*dict, optional*) – if provided, this dictionary will be populated with mappings from old node -> new node

enumerate_depth()

enumerate_traverse()

find_merges()

Find nodes that have the same parent and frame.

Find nodes that have the same parent and duplicate frame, and return a mapping from nodes that should be eliminated to nodes they should be merged into.

Returns
dictionary from nodes to their merge targets

Return type
(dict)

`static from_lists(*roots)`

Convenience method to invoke Node.from_lists() on each root value.

`is_tree()`

True if this graph is a tree, false otherwise.

`merge_nodes(merges)`

Merge some nodes in a graph into others.

`merges` is a dictionary keyed by old nodes, with values equal to the nodes that they need to be merged into. Old nodes' parents and children are connected to the new node.

Parameters
merges (*dict*) – dictionary from source nodes -> targets

`node_order_traverse(order='pre', attrs=None, visited=None)`

Preorder traversal of all roots of this Graph, sorting by "node order" column.

Parameters
attrs (*list or str, optional*) – If provided, extract these fields from nodes while traversing and yield them. See `traverse()` for details.

Only preorder traversal is currently supported.

`normalize()`

`traverse(order='pre', attrs=None, visited=None)`

Preorder traversal of all roots of this Graph.

Parameters
attrs (*list or str, optional*) – If provided, extract these fields from nodes while traversing and yield them. See `traverse()` for details.

Only preorder traversal is currently supported.

`union(other, old_to_new=None)`

Create the union of self and other and return it as a new Graph.

This creates a new graph and does not modify self or other. The new Graph has entirely new nodes.

Parameters

- **other** (Graph) – another Graph
- **old_to_new** (*dict, optional*) – if provided, this dictionary will be populated with mappings from old node -> new node

Returns
new Graph containing all nodes and edges from self and other

Return type
(*Graph*)

`hatchet.graph.index_by(attr, objects)`

Put objects into lists based on the value of an attribute.

Returns
dictionary of lists of objects, keyed by attribute value

Return type
(dict)

10.5 hatchet.graphframe module

exception hatchet.graphframe.**EmptyFilter**

Bases: Exception

Raised when a filter would otherwise return an empty GraphFrame.

class hatchet.graphframe.**GraphFrame**(*graph*, *dataframe*, *exc_metrics=None*, *inc_metrics=None*, *default_metric='time'*, *metadata={}*)

Bases: object

An input dataset is read into an object of this type, which includes a graph and a dataframe.

add(*other*)

Returns the column-wise sum of two graphframes as a new graphframe.

This graphframe is the union of self's and other's graphs, and does not modify self or other.

Returns
new graphframe

Return type
(*GraphFrame*)

copy()

Return a partially shallow copy of the graphframe.

This copies the DataFrame object, but the data is comprised of references. The Graph is shared between self and the new GraphFrame.

Parameters
self (GraphFrame) – Object to make a copy of.

Returns

Copy of self
graph (graph): Reference to self's graph dataframe (DataFrame): Pandas "non-deep" copy of dataframe exc_metrics (list): Copy of self's exc_metrics inc_metrics (list): Copy of self's inc_metrics default_metric (str): N/A metadata (dict): Copy of self's metadata

Return type
other (*GraphFrame*)

deepcopy()

Return a deep copy of the graphframe.

Parameters
self (GraphFrame) – Object to make a copy of.

Returns

Copy of self
graph (graph): Deep copy of self's graph dataframe (DataFrame): Pandas "deep" copy with node objects updated to match graph from "node_clone" exc_metrics (list): Copy of

self's exc_metrics inc_metrics (list): Copy of self's inc_metrics default_metric (str): N/A metadata (dict): Copy of self's metadata

Return type
other (*GraphFrame*)

div(*other*)

Returns the column-wise float division of two graphframes as a new graphframe.

This graphframe is the union of self's and other's graphs, and does not modify self or other.

Returns
new graphframe

Return type
(*GraphFrame*)

drop_index_levels(*function=<function mean>*)

Drop all index levels but *node*.

filter(*filter_obj*, *squash=True*, *update_inc_cols=True*, *num_procs=2*, *rec_limit=1000*, *multi_index_mode='off'*)

Filter the dataframe using a user-supplied function.

Note: Operates in parallel on user-supplied lambda functions.

Parameters

- **filter_obj** (*callable, list, or* QueryMatcher) – the filter to apply to the GraphFrame.
- **squash** (*boolean, optional*) – if True, automatically call squash for the user.
- **update_inc_cols** (*boolean, optional*) – if True, update inclusive columns when performing squash.
- **rec_limit** – set Python recursion limit, increase if running into recursion depth errors) (default: 1000).

static from_caliper(*filename_or_stream*, *query=None*)

Read in a Caliper .cali or .json file.

Parameters

- **filename_or_stream** (*str or file-like*) – name of a Caliper output file in *.cali* or JSON-split format, or an open file object to read one
- **query** (*str*) – cali-query in CalQL format

static from_caliperreader(*filename_or_caliperreader*, *native=False*, *string_attributes=[]*)

Read in a native Caliper *cali* file using Caliper's python reader.

Parameters

- **filename_or_caliperreader** (*str or* CaliperReader) – name of a Caliper output file in *.cali* format, or a CaliperReader object
- **native** (*bool*) – use native or user-readable metric names (default)
- **string_attributes** (*str or list, optional*) – Adds existing string attributes from within the caliper file to the dataframe

static from_cprofile(*filename*)

Read in a pstats/prof file generated using python's cProfile.

static from_gprof_dot(*filename*)

Read in a DOT file generated by gprof2dot.

static from_hdf(*filename*, ***kwargs*)

static from_hpctoolkit(*dirname*)

Read an HPCToolkit database directory into a new GraphFrame.

Parameters
dirname (*str*) – parent directory of an HPCToolkit experiment.xml file

Returns
new GraphFrame containing HPCToolkit profile data

Return type
(*GraphFrame*)

static from_json(*json_spec*, ***kwargs*)

static from_lists(**lists*)

Make a simple GraphFrame from lists.

This creates a Graph from lists (see `Graph.from_lists()`) and uses it as the index for a new GraphFrame. Every node in the new graph has exclusive time of 1 and inclusive time is computed automatically.

static from_literal(*graph_dict*)

Create a GraphFrame from a list of dictionaries.

static from_pyinstrument(*filename*)

Read in a JSON file generated using Pyinstrument.

static from_spotdb(*db_key*, *list_of_ids=None*)

Read multiple graph frames from a SpotDB instance

Parameters

- **db_key** (*str or SpotDB object*) – locator for SpotDB instance This can be a SpotDB object directly, or a locator for a spot database, which is a string with either:
  - A directory for .cali files,
  - A .sqlite file name
  - A SQL database URL (e.g., "mysql://hostname/db")
- **list_of_ids** – The list of run IDs to read from the database. If this is None, returns all runs.

Returns
A list of graphframes, one for each requested run that was found

static from_tau(*dirname*)

Read in a profile generated using TAU.

static from_timemory(*input=None*, *select=None*, ***_kwargs*)

Read in timemory data.

Links:
https://github.com/NERSC/timemory https://timemory.readthedocs.io

Parameters

- **input** (*str or file-stream or dict or None*) – Valid argument types are:

  1. Filename for a timemory JSON tree file
  2. Open file stream to one of these files
  3. Dictionary from timemory JSON tree

  Currently, timemory supports two JSON layouts: flat and tree. The former is a 1D-array representation of the hierarchy which represents the hierarchy via indentation schemes in the labels and is not compatible with hatchet. The latter is a hierarchical representation of the data and is the required JSON layout when using hatchet. Timemory JSON tree files typically have the extension ".tree.json".

  If input is None, this assumes that timemory has been recording data within the application that is using hatchet. In this situation, this method will attempt to import the data directly from timemory.

  At the time of this writing, the direct data import will:

  1. Stop any currently collecting components
  2. Aggregate child thread data of the calling thread
  3. Clear all data on the child threads
  4. Aggregate the data from any MPI and/or UPC++ ranks.

  Thus, if MPI or UPC++ is used, every rank must call this routine. The zeroth rank will have the aggregation and all the other non-zero ranks will only have the rank-specific data.

  Whether or not the per-thread and per-rank data itself is combined is controlled by the *collapse_threads* and *collapse_processes* attributes in the *timemory.settings* submodule.

  In the C++ API, it is possible for only #1 to be applied and data can be obtained for an individual thread and/or rank without aggregation. This is not currently available to Python, however, it can be made available upon request via a GitHub Issue.

- **select** (*list of str*) – A list of strings which match the component enumeration names, e.g. ["cpu_clock"].
- **per_thread** (*boolean*) – Ensures that when applying filters to the graphframe, frames with identical name/file/line/etc. info but from different threads are not combined
- **per_rank** (*boolean*) – Ensures that when applying filters to the graphframe, frames with identical name/file/line/etc. info but from different ranks are not combined

generate_exclusive_columns(*inc_metrics=None*)

Generates exclusive metrics from available inclusive metrics. :param inc_metrics: Instead of generating the exclusive time for each inclusive metric, it is possible to specify those metrics manually. Defaults to None. :type inc_metrics: str, list, optional

Currently, this function determines which metrics to generate by looking for one of two things:

1. An inclusive metric ending in "(inc)" that does not have an exclusive metric with the same name (minus "(inc)")
2. An inclusive metric not ending in "(inc)"

The metrics that are generated will have one of two name formats:

1. If the corresponding inclusive metric's name ends in "(inc)", the exclusive metric will have the same name, minus "(inc)"

2. If the corresponding inclusive metric's name does not end in "(inc)", the exclusive metric will have the same name as the inclusive metric, followed by a "(exc)" suffix

groupby_aggregate(*groupby_function*, *agg_function*)

Groupby-aggregate dataframe and reindex the Graph.

Reindex the graph to match the groupby-aggregated dataframe.

Update the frame attributes to contain those columns in the dataframe index.

Parameters

- **self** (*graphframe*) – self's graphframe
- **groupby_function** – groupby function on dataframe
- **agg_function** – aggregate function on dataframe

Returns
new graphframe with reindexed graph and groupby-aggregated dataframe

Return type
(*GraphFrame*)

mul(*other*)

Returns the column-wise float multiplication of two graphframes as a new graphframe.

This graphframe is the union of self's and other's graphs, and does not modify self or other.

Returns
new graphframe

Return type
(*GraphFrame*)

show_metric_columns()

Returns a list of dataframe column labels.

squash(*update_inc_cols=True*)

Rewrite the Graph to include only nodes present in the DataFrame's rows.

This can be used to simplify the Graph, or to normalize Graph indexes between two GraphFrames.

Parameters
update_inc_cols (*boolean, optional*) – if True, update inclusive columns.

sub(*other*)

Returns the column-wise difference of two graphframes as a new graphframe.

This graphframe is the union of self's and other's graphs, and does not modify self or other.

Returns
new graphframe

Return type
(*GraphFrame*)

subgraph_sum(*columns*, *out_columns=None*, *function=<function GraphFrame.<lambda>>*)

Compute sum of elements in subgraphs.

For each row in the graph, `out_columns` will contain the element-wise sum of all values in `columns` for that row's node and all of its descendants.

This algorithm is worst-case quadratic in the size of the graph, so we try to call `subtree_sum` if we can. In general, there is not a particularly efficient algorithm known for subgraph sums, so this does about as well as we know how.

Parameters

- **columns** (*list of str*) – names of columns to sum (default: all columns)
- **out_columns** (*list of str*) – names of columns to store results (default: in place)
- **function** (*callable*) – associative operator used to sum elements, sum of an all-NA series is NaN (default: sum(min_count=1))

subtree_sum(*columns*, *out_columns=None*, *function=<function GraphFrame.<lambda>>*)

Compute sum of elements in subtrees. Valid only for trees.

For each row in the graph, `out_columns` will contain the element-wise sum of all values in `columns` for that row's node and all of its descendants.

This algorithm will multiply count nodes with in-degree higher than one – i.e., it is only correct for trees. Prefer using `subgraph_sum` (which calls `subtree_sum` if it can), unless you have a good reason not to.

Parameters

- **columns** (*list of str*) – names of columns to sum (default: all columns)
- **out_columns** (*list of str*) – names of columns to store results (default: in place)
- **function** (*callable*) – associative operator used to sum elements, sum of an all-NA series is NaN (default: sum(min_count=1))

to_dict()

to_dot(*metric=None*, *name='name'*, *rank=0*, *thread=0*, *threshold=0.0*)

Write the graph in the graphviz dot format: https://www.graphviz.org/doc/info/lang.html

to_flamegraph(*metric=None*, *name='name'*, *rank=0*, *thread=0*, *threshold=0.0*)

Write the graph in the folded stack output required by FlameGraph http://www.brendangregg.com/flamegraphs.html

to_hdf(*filename*, *key='hatchet_graphframe'*, ***kwargs*)

to_json()

to_literal(*name='name'*, *rank=0*, *thread=0*, *cat_columns=[]*)

Format this graph as a list of dictionaries for Roundtrip visualizations.

tree(*metric_column=None*, *annotation_column=None*, *precision=3*, *name_column='name'*, *expand_name=False*, *context_column='file'*, *rank=0*, *thread=0*, *depth=10000*, *highlight_name=False*, *colormap='RdYlGn'*, *invert_colormap=False*, *colormap_annotations=None*, *render_header=True*, *min_value=None*, *max_value=None*)

Visualize the Hatchet graphframe as a tree

Parameters

- **metric_column** (*str, list, optional*) – Columns to use the metrics from. Defaults to None.
- **annotation_column** (*str, optional*) – Column to use as an annotation. Defaults to None.
- **precision** (*int, optional*) – Precision of shown numbers. Defaults to 3.
- **name_column** (*str, optional*) – Column of the node name. Defaults to "name".

- **expand_name** (*bool, optional*) – Limits the lenght of the node name. Defaults to False.
- **context_column** (*str, optional*) – Shows the file this function was called in (Available with HPCToolkit). Defaults to "file".
- **rank** (*int, optional*) – Specifies the rank to take the data from. Defaults to 0.
- **thread** (*int, optional*) – Specifies the thread to take the data from. Defaults to 0.
- **depth** (*int, optional*) – Sets the maximum depth of the tree. Defaults to 10000.
- **highlight_name** (*bool, optional*) – Highlights the names of the nodes. Defaults to False.
- **colormap** (*str, optional*) – Specifies a colormap to use. Defaults to "RdYlGn".
- **invert_colormap** (*bool, optional*) – Reverts the chosen colormap. Defaults to False.
- **colormap_annotations** (*str, list, dict, optional*) – Either provide the name of a colormap, a list of colors to use or a dictionary which maps the used annotations to a color. Defaults to None.
- **render_header** (*bool, optional*) – Shows the Preamble. Defaults to True.
- **min_value** (*int, optional*) – Overwrites the min value for the coloring legend. Defaults to None.
- **max_value** (*int, optional*) – Overwrites the max value for the coloring legend. Defaults to None.

Returns
String representation of the tree, ready to print

Return type
str

unify(*other*)

Returns a unified graphframe.

Ensure self and other have the same graph and same node IDs. This may change the node IDs in the dataframe.

Update the graphs in the graphframe if they differ.

update_inclusive_columns()

Update inclusive columns (typically after operations that rewire the graph.

exception hatchet.graphframe.**InvalidFilter**

Bases: Exception

Raised when an invalid argument is passed to the filter function.

hatchet.graphframe.**parallel_apply**(*filter_function*, *dataframe*, *queue*)

A function called in parallel, which does a pandas apply on part of a dataframe and returns the results via multiprocessing queue function.

10.6 hatchet.node module

exception hatchet.node.**MultiplePathError**

Bases: Exception

Raised when a node is asked for a single path but has multiple.

class hatchet.node.**Node**(*frame_obj*, *parent=None*, *hnid=-1*, *depth=-1*)

Bases: object

A node in the graph. The node only stores its frame.

add_child(*node*)

Adds a child to this node's list of children.

add_parent(*node*)

Adds a parent to this node's list of parents.

copy()

Copy this node without preserving parents or children.

dag_equal(*other*, *vs=None*, *vo=None*)

Check if DAG rooted at self has the same structure as that rooted at other.

classmethod from_lists(*lists*)

Construct a hierarchy of nodes from recursive lists.

For example, this will construct a simple tree:

```
Node.from_lists(
    ["a",
        ["b", "d", "e"],
        ["c", "f", "g"],
    ]
)
```

```
    a
   / \
  b   c
 / |   | \
d  e   f  g
```

And this will construct a simple diamond DAG:

```
d = Node(Frame(name="d"))
Node.from_lists(
    ["a",
        ["b", d],
        ["c", d]
    ]
)
```

```
  a
 / \
b   c
```

(continues on next page)

(continued from previous page)

```
\ /
 d
```

In the above examples, the 'a' represents a Node with its *frame == Frame(name="a")*.

node_order_traverse(*order='pre'*, *attrs=None*, *visited=None*)

Traverse the tree depth-first and yield each node, sorting children by "node order".

Parameters

- **order** (*str*) – "pre" or "post" for preorder or postorder (default: pre)
- **attrs** (*list or str, optional*) – if provided, extract these fields from nodes while traversing and yield them
- **visited** (*dict, optional*) – dictionary in which each visited node's in-degree will be stored

path(*attrs=None*)

Path to this node from root. Raises if there are multiple paths.

This is useful for trees (where each node only has one path), as it just gets the only element from `self.paths`. This will fail with a MultiplePathError if there is more than one path to this node.

paths()

List of tuples, one for each path from this node to any root.

Paths are tuples of node objects.

traverse(*order='pre'*, *attrs=None*, *visited=None*)

Traverse the tree depth-first and yield each node.

Parameters

- **order** (*str*) – "pre" or "post" for preorder or postorder (default: pre)
- **attrs** (*list or str, optional*) – if provided, extract these fields from nodes while traversing and yield them
- **visited** (*dict, optional*) – dictionary in which each visited node's in-degree will be stored

hatchet.node.**node_traversal_order**(*node*)

Deterministic key function for sorting nodes by specified "node order" (which gets assigned to _hatchet_nid) in traversals.

hatchet.node.**traversal_order**(*node*)

Deterministic key function for sorting nodes in traversals.

10.7 hatchet.version module

10.8 Module contents

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

h

INDEX

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X