PiterPy

TECHNICAL CONFERENCE FOR HARDCORE PYTHON DEVELOPERS

RedisGears

SAINT PETERSBURG
2019 NOVEMBER 1

Meir Shpilraien (@Meir_Shpilraien)
Redislabs, *Software Architect*

PiterPy

Agenda

- What is Redis?
- What is RedisGears and how it works?
- Streaming processing with python and RedisGears
- Redis(Gears) and Python integration
- RedisAI and RedisGears
- Demos, Demos, and more Demos

PiterPy

What is Redis?

Redis is an **open source** (BSD licensed), **in-memory data structure** store, used as a database, cache and message broker. It supports data structures such as **strings, hashes, lists, sets, sorted sets** with range queries, **bitmaps, hyperloglogs, geospatial** indexes with radius queries and streams. Redis has built-in **replication**, **Lua scripting, LRU eviction, transactions** and different levels of **on-disk persistence**, and provides high availability via Redis Sentinel and automatic partitioning with Redis Cluster.

PiterPy

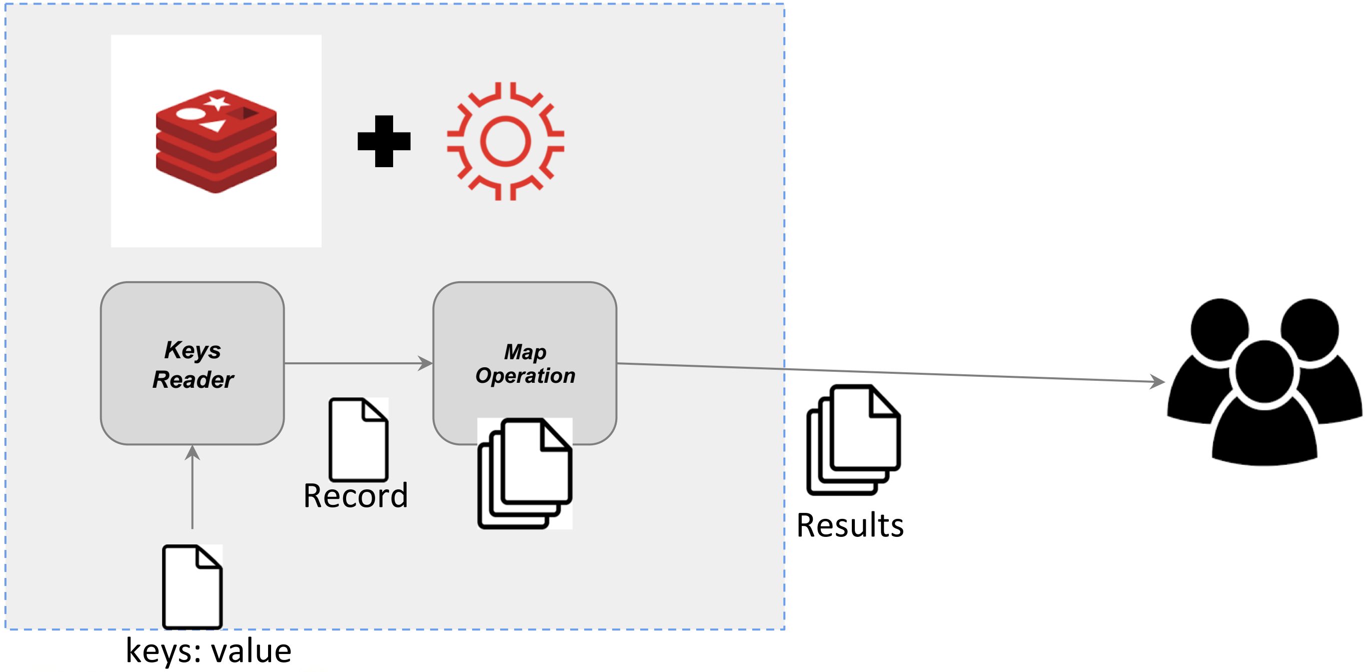
What is RedisGears?

RedisGears is a **Serverless** engine for **multi-model and cluster operations** in Redis, supporting both **event driven** as well as **batch operations**

- Almost always agnostic from redis topology (stand alone, cluster, enterprise)
- Built in coordinator for cluster support
- Built in map/reduce operations
- **Support full embedded Python** and C api
- Built as a Redis module

PiterPy

Using RedisGears - Mapping Example



PiterPy

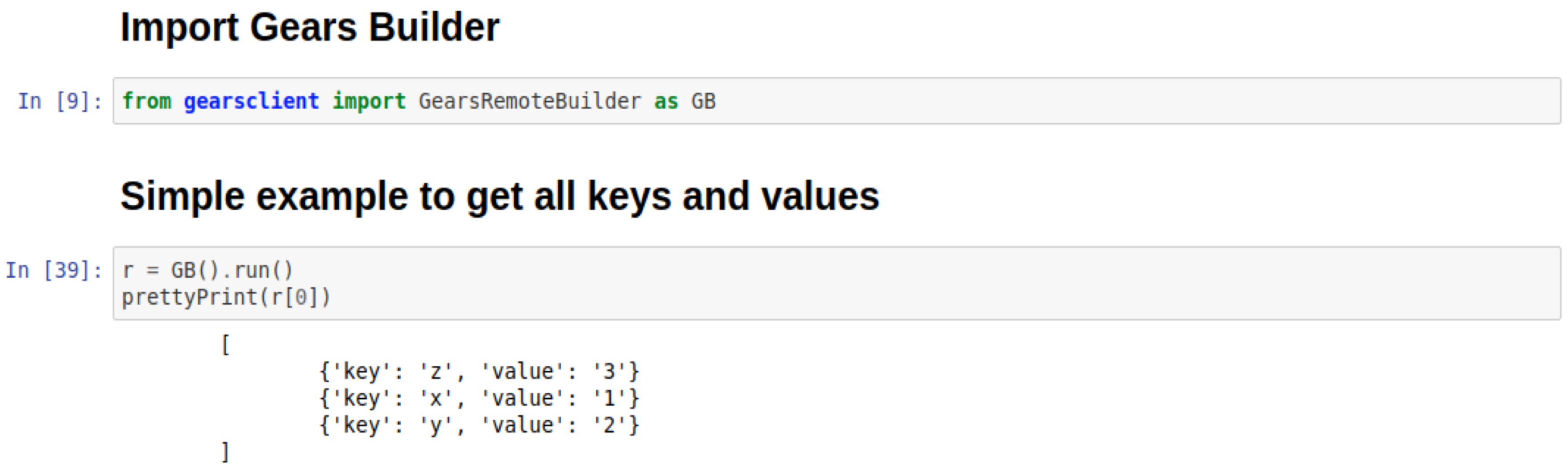
Using RedisGears

RedisGears allow defining a **pipe of operations**

- Returning value from one operation pass to the operation that follows it in the pipe
- Last operation returning the result to the user
- First operation is called ‘reader’ - responsible for providing data
  - **Keys reader** - read keys from Redis
  - **Stream reader** - read streams from Redis
  - **Python reader** - allow to user to write his own readers in python
- Data units that pass through the pipe are called **Records**

PiterPy

Using RedisGears



Import Gears Builder

```
In [9]: from gearsclient import GearsRemoteBuilder as GB
```

Simple example to get all keys and values

```
In [39]: r = GB().run()
         prettyPrint(r[0])
```

```
[
        {'key': 'z', 'value': '3'}
        {'key': 'x', 'value': '1'}
        {'key': 'y', 'value': '2'}
]
```

PiterPy

Using RedisGears

Simple Examples

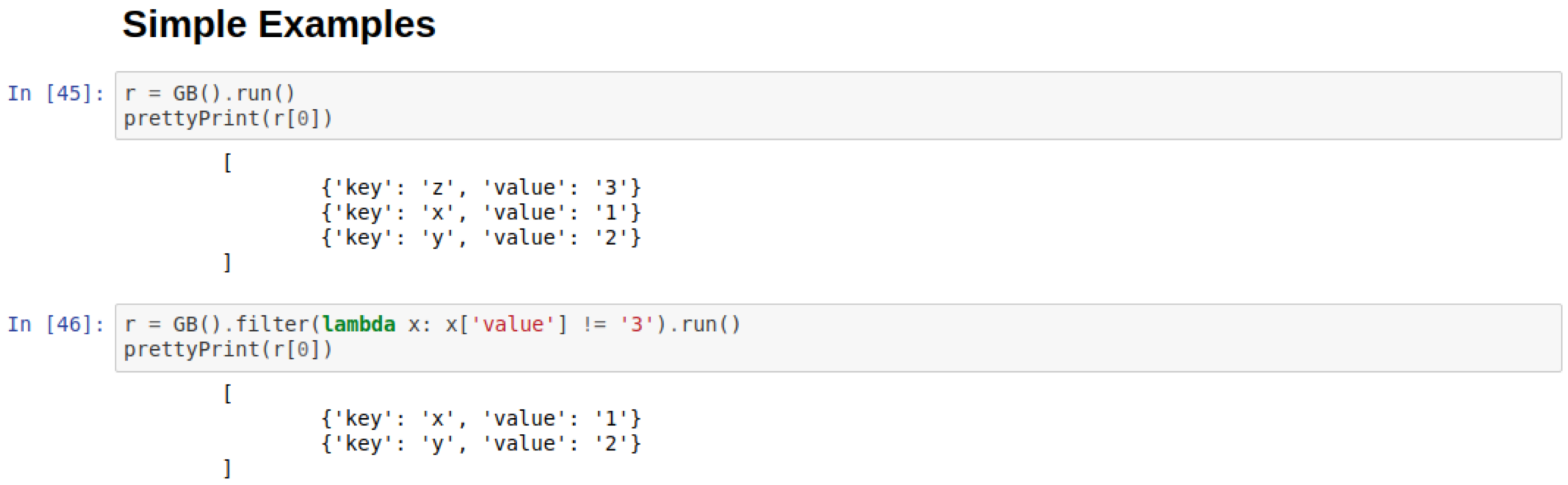
```
In [39]: r = GB().run()
         prettyPrint(r[0])

        [
                {'key': 'z', 'value': '3'}
                {'key': 'x', 'value': '1'}
                {'key': 'y', 'value': '2'}
        ]
```

```
In [40]: r = GB().map(lambda x: x['key']).run()
         prettyPrint(r[0])

        [
                z
                x
                y
        ]
```

```
In [41]: r = GB().map(lambda x: x['value']).run()
         prettyPrint(r[0])

        [
                3
                1
                2
        ]
```

PiterPy

Using RedisGears



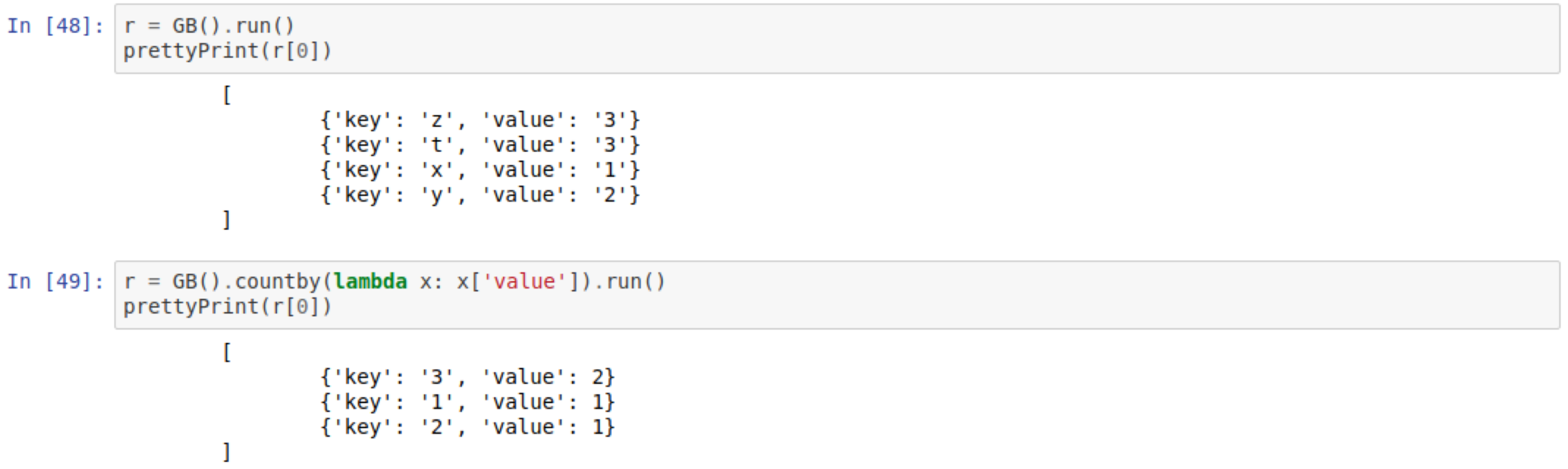
Simple Examples

```
In [45]: r = GB().run()
         prettyPrint(r[0])

        [
                {'key': 'z', 'value': '3'}
                {'key': 'x', 'value': '1'}
                {'key': 'y', 'value': '2'}
        ]

In [46]: r = GB().filter(lambda x: x['value'] != '3').run()
         prettyPrint(r[0])

        [
                {'key': 'x', 'value': '1'}
                {'key': 'y', 'value': '2'}
        ]
```

PiterPy

Using RedisGears

Simple Examples



```
In [48]: r = GB().run()
         prettyPrint(r[0])

                [
                        {'key': 'z', 'value': '3'}
                        {'key': 't', 'value': '3'}
                        {'key': 'x', 'value': '1'}
                        {'key': 'y', 'value': '2'}
                ]

In [49]: r = GB().countby(lambda x: x['value']).run()
         prettyPrint(r[0])

                [
                        {'key': '3', 'value': 2}
                        {'key': '1', 'value': 1}
                        {'key': '2', 'value': 1}
                ]
```

PiterPy

Imdb Example

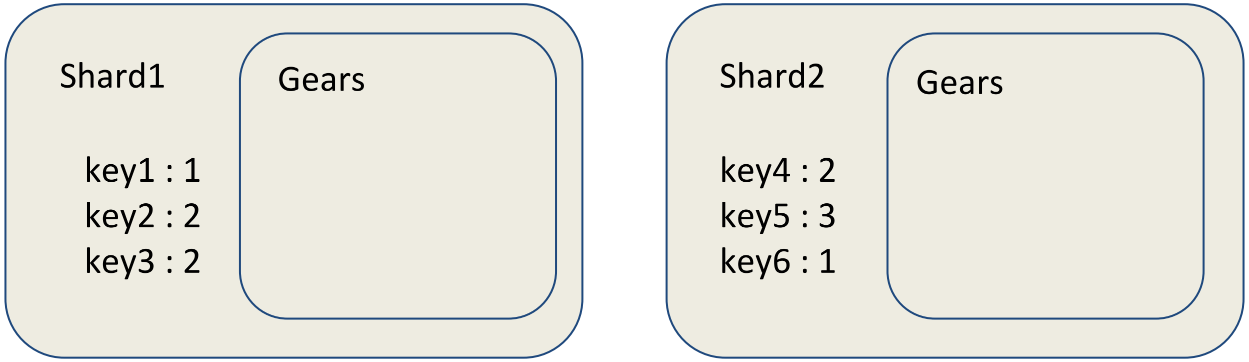
```
# create the pipe builder
builder = GB('KeysOnlyReader')

# get from each hash the genres field
builder.map(lambda x:execute('hget', x, 'genres'))

# filter those who do not have genres
builder.filter(lambda x: x is not None)

# split genres by comma
builder.flatmap(lambda x: x.split(','))

# count for each genre the number of times it appears
builder.countby()

# start the execution
builder.run()
```

How does it Work on Cluster

- RedisGears handles the distribution on the execution to all the nodes in the cluster
- Local operation runs in parallel on the shards (map, filter, ...)
- Accumulate operations (groupby, countby, ...) requires the data to be reshuffled such that records belong to the same group will be located on the same shard.
  - Each shard perform the reduce function locally and continue the execution
- On done, the results returns to the shard that start the execution (the initiator) and it returns the data back to redis

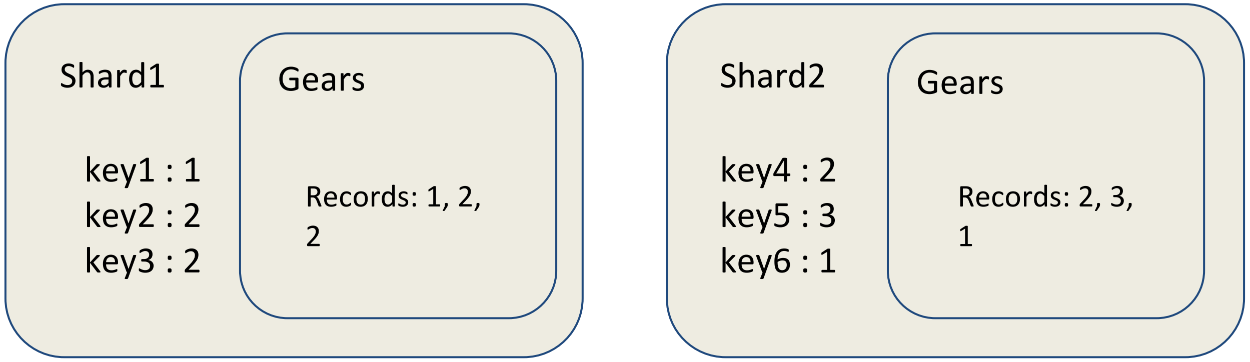
How does it Work on Cluster

- Count distinct values example



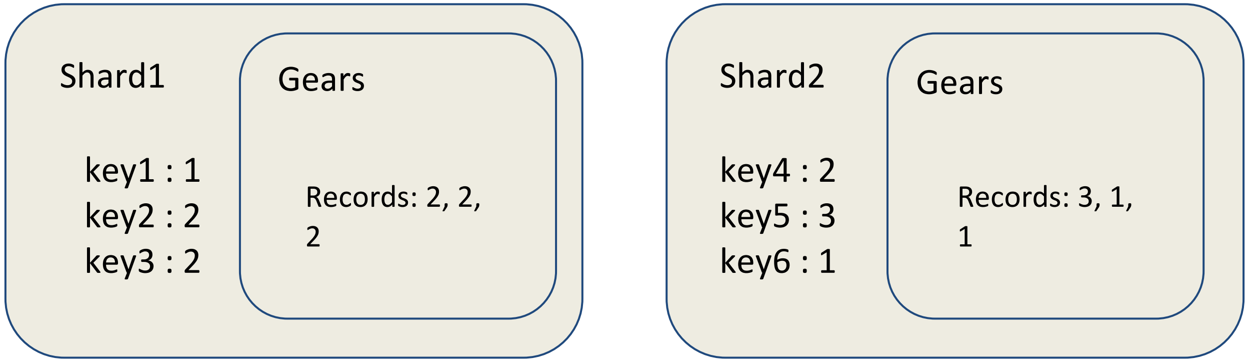
How does it Work on Cluster

- Values are extracted from redis



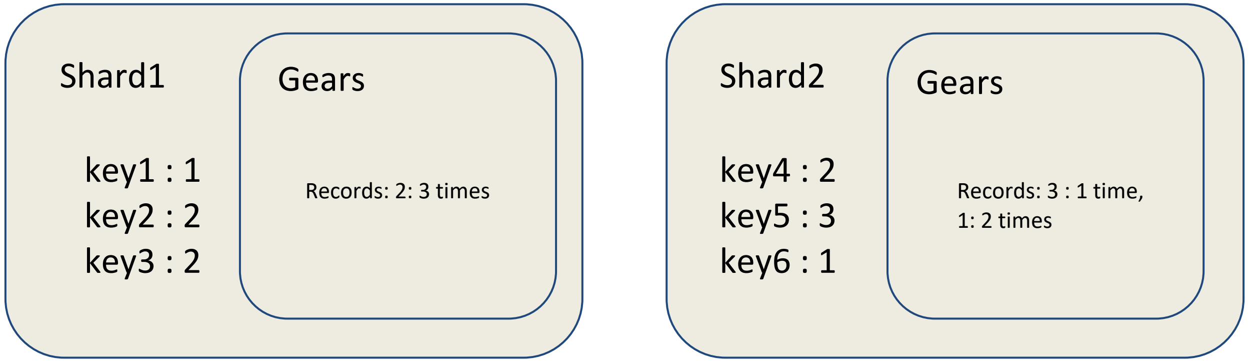
How does it Work on Cluster

- Data is reshaffeled such that records from same group will be located on same shard



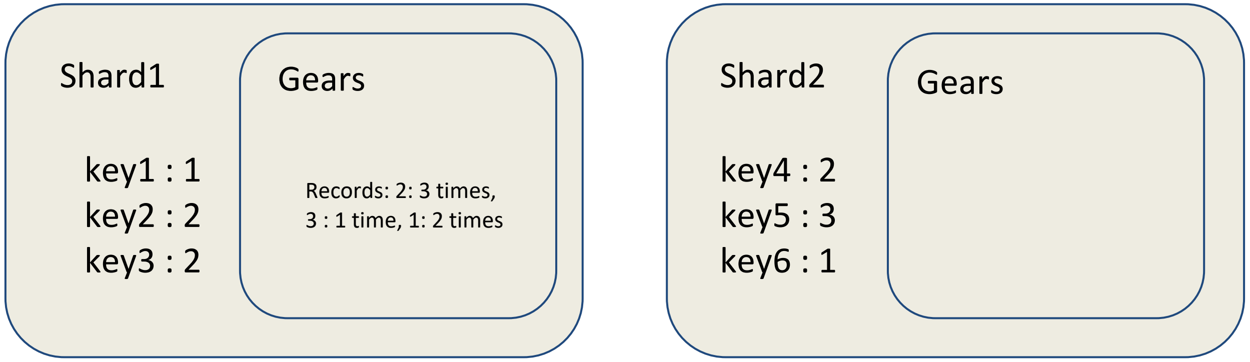
How does it Work on Cluster

- Count distinct is performed on each shard separately



How does it Work on Cluster

- Collect the data

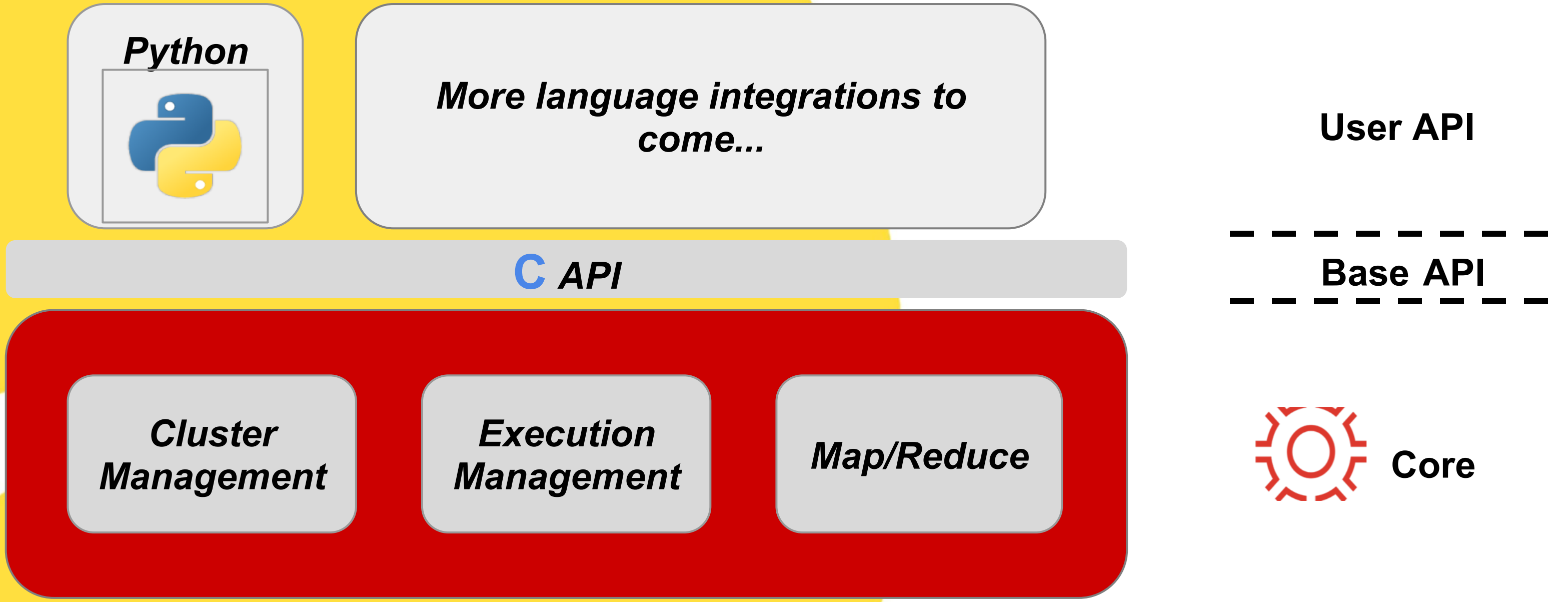


Stream Processing with RedisGears

- RedisGears expose a Streaming API which allows triggers executions on events
  - Redis Streams events
  - Redis Keys events
- The following will maintain a set of all the keys in redis

```
# create the builder
builder = GB()

# filter events on key:'all_keys'
builder.filter(lambda x: x['key'] != 'all_keys')

# add the keys to 'all_keys' set
builder.map(lambda x:execute('sadd', 'all_keys', x['key']))

# register the execution on key space notification
builder.register()
```

RedisGears Architecture



PiterPy

RedisGears and Python

- RedisGears expose a C level api which can be used by anyone.
- RedisGears runs an embedded python interpreter that uses the C level api to interact with RedisGears, such interaction allow the python interpreter to perform:
    - MapReduce Operations
    - Streaming Processing
- RedisGears take care of cluster management and distribute operations

Embedded Python Pros and Cons

Pros:

- Fast - Direct memory access to redis internal objects
- Less memory usage
  - No need to copy the date to another process
  - Instead of starting multiple interpreters we create sub-interpreters
- Easier to show and control memory allocation in redis info report

Embedded Python Pros and Cons

Cons:

- Different clients share the same interpreter
- It is not possible to run python code from 2 client simultaneously (will be solved in future python releases)
- A bug in the interpreter might cause redis to crash - Less secure

PiterPy

Python Sub-Interpreters

Client 1

```
# declare a global counter
global Counter

# Count how many keys there are in redis
GB().foreach(lambda x: Counter+=1).run()
```

Client 2

```
# declare a global counter
global Counter

# Count how many keys contains 'foo'
GB().filter(lambda x:
x['key'].contains('foo')).foreach(lambda x:
Counter+=1).run()
```

PiterPy

Python Sub-Interpreters

A **sub-interpreter** is a (almost) totally **separate environment** for the execution of Python code. The Python C API makes it possible to create a new sub-interpreter using `Py_NewInterpreter`, destroy it using `Py_EndInterpreter` and switch between sub-interpreters using `PyThreadState_Swap`. RedisGears invokes these internally and maintains the association between the user's call to `RG.PYEXECUTE` and its respective sub-interpreter.

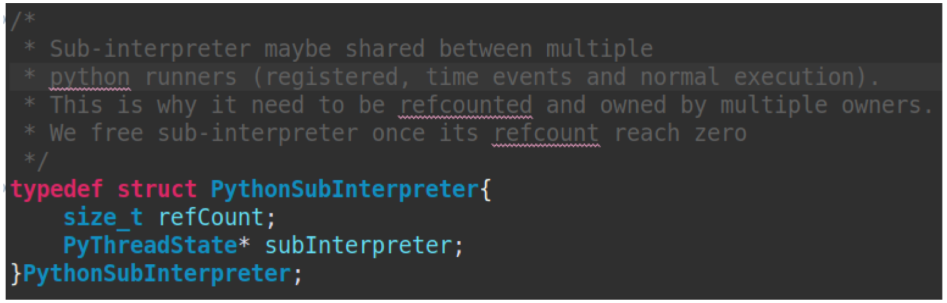
Python Sub-Interpreters

```
# declare a global counter
global Counter

# Count how many keys contains ‘foo’
GB().filter(lambda x: x[‘key’].contains(‘foo’)).foreach(lambda x:
Counter+=1).register()
```

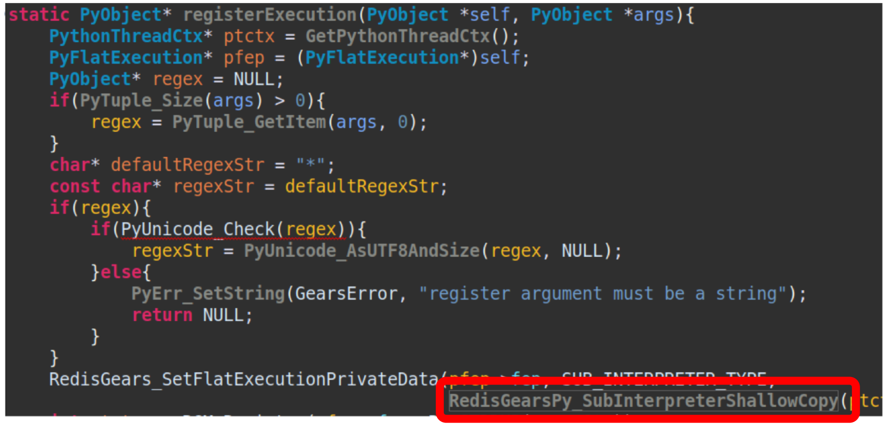
PiterPy

Python Sub-Interpreters

When `RG.PYEXECUTE` is called, a **new sub-interpreter is created** to execute the provided script. That sub-interpreter is also **"inherited" by all subsequent operations** - i.e. executions, registrations and timeEvents, that the script creates. Because there may be **multiple owner for the sub-interpreter**, RedisGears keeps an **internal reference count** for each one so it can be safely freed.



```c
/*
 * Sub-interpreter maybe shared between multiple
 * python runners (registered, time events and normal execution).
 * This is why it need to be refcounted and owned by multiple owners.
 * We free sub-interpreter once its refcount reach zero
 */
typedef struct PythonSubInterpreter{
    size_t refCount;
    PyThreadState* subInterpreter;
}PythonSubInterpreter;
```

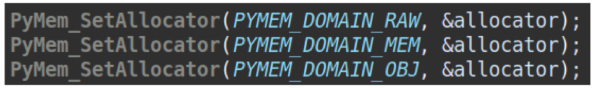


```c
static PyObject* registerExecution(PyObject *self, PyObject *args){
    PythonThreadCtx* ptctx = GetPythonThreadCtx();
    PyFlatExecution* pfep = (PyFlatExecution*)self;
    PyObject* regex = NULL;
    if(PyTuple_Size(args) > 0){
        regex = PyTuple_GetItem(args, 0);
    }
    char* defaultRegexStr = "*";
    const char* regexStr = defaultRegexStr;
    if(regex){
        if(PyUnicode_Check(regex)){
            regexStr = PyUnicode_AsUTF8AndSize(regex, NULL);
        }else{
            PyErr_SetString(GearsError, "register argument must be a string");
            return NULL;
        }
    }
    RedisGears_SetFlatExecutionPrivateData(pfep->fep, SUB_INTERPRETER_TYPE,
                                           RedisGearsPy_SubInterpreterShallowCopy(ptc
```

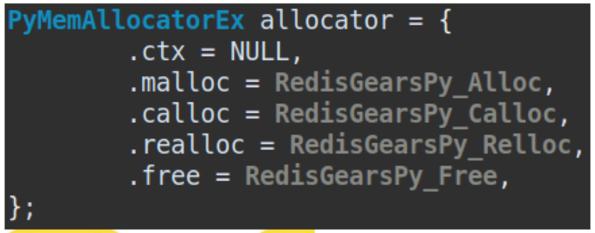
PiterPy

Python Interpreter with Redis Allocator

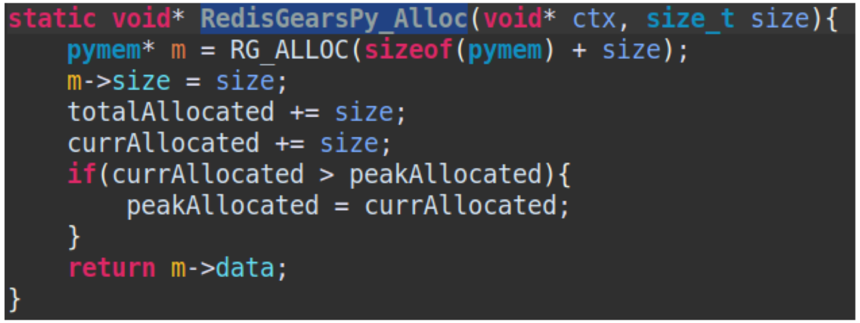
- The python interpreter allows setting custom memory allocators
- We used this ability to allow the python interpreter to use the Redis memory allocator
  - The python memory usage is showed in redis ‘info memory’ command
  - We can control and limit the amount of memory used by the interpreter
  - Future plans: control the amount of memory used by sub-interpreter, i.e - limit used memory by a single client



```c
PyMem_SetAllocator(PYMEM_DOMAIN_RAW, &allocator);
PyMem_SetAllocator(PYMEM_DOMAIN_MEM, &allocator);
PyMem_SetAllocator(PYMEM_DOMAIN_OBJ, &allocator);
```



```c
PyMemAllocatorEx allocator = {
        .ctx = NULL,
        .malloc = RedisGearsPy_Alloc,
        .calloc = RedisGearsPy_Calloc,
        .realloc = RedisGearsPy_Relloc,
        .free = RedisGearsPy_Free,
};
```



```c
static void* RedisGearsPy_Alloc(void* ctx, size_t size){
    pymem* m = RG_ALLOC(sizeof(pymem) + size);
    m->size = size;
    totalAllocated += size;
    currAllocated += size;
    if(currAllocated > peakAllocated){
        peakAllocated = currAllocated;
    }
    return m->data;
}
```

Python and Clustering

- Cluster operations require serialize and deserialize python object between redis shards
  - It also require serializing python functions between the shards
- For such serialization RedisGears make use of [CloudPickle]
- When execution is created, all the python function listed by the execution is distributed to all the shards
- When execution is running, records might sent from one shard to another (for example during groupby) using CloudPickle serialization

PiterPy

Python and Clustering

- Some objects can not be serialized:
  - Native (C implemented) objects like tensors, numpy matrix, ...
- Those objects will need to be transformed to a serialized object before sent to another shard
- Currently its the user responsibility to transform them otherwise the execution will failed.

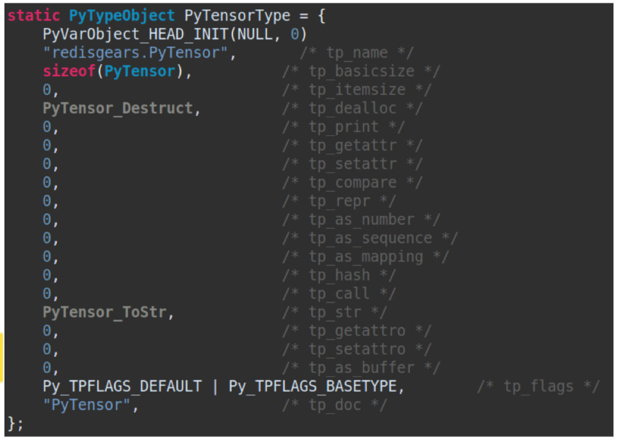
RedisAI

- RedisAI:
  - A new Redis module that allow running AI models directly on redis
  - Expose Tensors and Models as Redis Data Types
  - Expose C api for other modules to use it directly (direct function call)
  - Create tensore
  - Run models
- RedisAI comes in handy when your data already located on redis
  - Get it out of Redis to Run a tensorflow model is time consuming
  - Instead we can run the tensetflow model directly on redis
- Use Cases:
  - Stream Data classification like: Image processing, Sound recognition
  - Fraud Detection

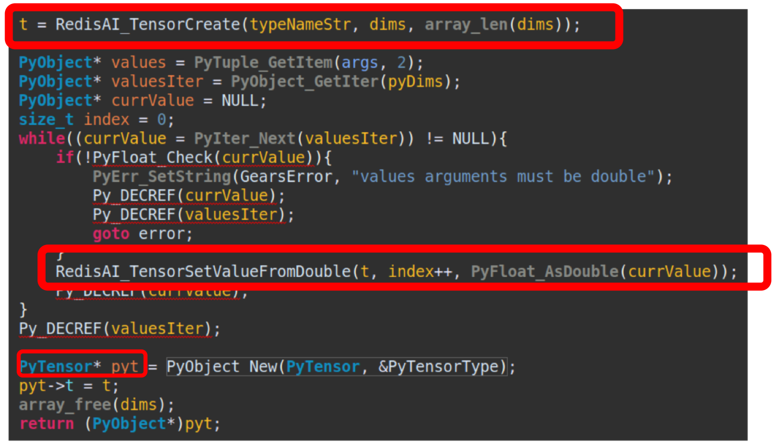
PiterPy

RedisGears & RedisAI

- RedisAI expose a direct C api that can be used by other redis modules
- RedisGears can use the C api to expose AI capabilities via the python interpreter
    - PyTensor and PyGraphRunner are two Native objects expose to the Gears python interpreter and allow the user to run AI model via the gear script



```c
static PyTypeObject PyTensorType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    "redisgears.PyTensor",          /* tp_name */
    sizeof(PyTensor),              /* tp_basicsize */
    0,                             /* tp_itemsize */
    PyTensor_Destruct,             /* tp_dealloc */
    0,                             /* tp_print */
    0,                             /* tp_getattr */
    0,                             /* tp_setattr */
    0,                             /* tp_compare */
    0,                             /* tp_repr */
    0,                             /* tp_as_number */
    0,                             /* tp_as_sequence */
    0,                             /* tp_as_mapping */
    0,                             /* tp_hash */
    0,                             /* tp_call */
    PyTensor_ToStr,                /* tp_str */
    0,                             /* tp_getattro */
    0,                             /* tp_setattro */
    0,                             /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE,        /* tp_flags */
    "PyTensor",                    /* tp_doc */
};
```



```c
t = RedisAI_TensorCreate(typeNameStr, dims, array_len(dims));

PyObject* values = PyTuple_GetItem(args, 2);
PyObject* valuesIter = PyObject_GetIter(pyDims);
PyObject* currValue = NULL;
size_t index = 0;
while((currValue = PyIter_Next(valuesIter)) != NULL){
    if(!PyFloat_Check(currValue)){
        PyErr_SetString(GearsError, "values arguments must be double");
        Py_DECREF(currValue);
        Py_DECREF(valuesIter);
        goto error;
    }
    RedisAI_TensorSetValueFromDouble(t, index++, PyFloat_AsDouble(currValue));
    Py_DECREF(currValue);
}
Py_DECREF(valuesIter);

PyTensor* pyt = PyObject_New(PyTensor, &PyTensorType);
pyt->t = t;
array_free(dims);
return (PyObject*)pyt;
```

PiterPy

RedisGears & RedisAI

PiterPy

Links

https://redis.io/
https://oss.redislabs.com/redisgears/
https://oss.redislabs.com/redisai/
https://github.com/RedisGears/RedisGears
https://github.com/RedisGears/AnimalRecognitionDemo
https://github.com/RedisGears/EdgeRealtimeVideoAnalytics
https://github.com/RedisGears/redisgears-py

PiterPy

Thanks You