Disco Documentation

Release

Disco Project

December 16, 2013

Contents

CHAPTER 1

Background

1.1 What is Disco?

Disco is an implementation of *mapreduce* for distributed computing. Disco supports parallel computations over large data sets, stored on an unreliable cluster of computers, as in the original framework created by Google. This makes it a perfect tool for analyzing and processing large data sets, without having to worry about difficult technicalities related to distribution such as communication protocols, load balancing, locking, job scheduling, and fault tolerance, which are handled by Disco.

Disco can be used for a variety data mining tasks: large-scale analytics, building probabilistic models, and full-text indexing the Web, just to name a few examples.

1.1.1 Batteries included

The Disco core is written in Erlang, a functional language that is designed for building robust fault-tolerant distributed applications. Users of Disco typically write jobs in Python, which makes it possible to express even complex algorithms with very little code.

For instance, the following fully working example computes word frequencies in a large text:

```
from disco.core import Job, result_iterator

def map(line, params):
    for word in line.split():
        yield word, 1

def reduce(iter, params):
    from disco.util import kvgroup
    for word, counts in kvgroup(sorted(iter)):
        yield word, sum(counts)

if __name__ == '__main__':
    job = Job().run(input=["http://discoproject.org/media/text/chekhov.txt"],
                    map=map,
                    reduce=reduce)
    for word, count in result_iterator(job.wait(show=True)):
        print(word, count)
```

Disco is designed to integrate easily in larger applications, such as Web services, so that computationally demanding tasks can be delegated to a cluster independently from the core application. Disco provides an extremely compact Python API – typically only two functions are needed – as well as a REST-style Web API for job control and a easy-to-use Web interface for status monitoring.

Disco also exposes a simple worker protocol, allowing jobs to be written in any language that implements the protocol.

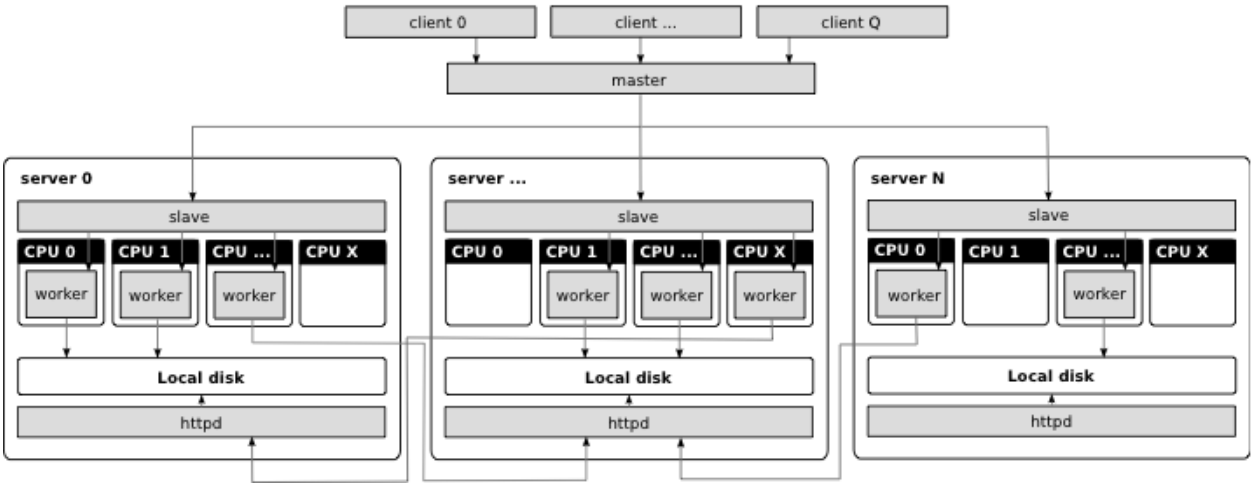
1.1.2 Distributed computing made easy

Disco is a good match for a cluster of commodity Linux servers. New nodes can be added to the system on the fly, by a single click on the Web interface. If a server crashes, active jobs are automatically re-routed to other servers without any interruptions. Together with an automatic provisioning mechanism, such as Fully Automatic Installation, even a large cluster can be maintained with only a minimal amount of manual work. As a proof of concept, Nokia Research Center in Palo Alto maintains an 800-core cluster running Disco using this setup.

1.1.3 Main features

- Proven to scale to hundreds of CPUs and tens of thousands of simultaneous tasks.
- Used to process datasets in the scale of tens of terabytes.
- Extremely simple to use: A typical tasks consists of two functions written in Python and two calls to the Disco API.
- Tasks can be specified in any other language as well, by implementing the Disco worker protocol.
- Input data can be in any format, even binary data such as images. The data can be located on any source that is accesible by HTTP or it can distributed to local disks.
- Fault-tolerant: Server crashes don't interrupt jobs. New servers can be added to the system on the fly.
- Flexible: In addition to the core map and reduce functions, a combiner function, a partition function and an input reader can be provided by the user.
- Easy to integrate to larger applications using the standard Disco module and the Web APIs.
- Comes with a built-in distributed storage system (*Disco Distributed Filesystem*).

1.2 Technical Overview



Disco is based on the master-slave architecture.

- The *master* receives *jobs*, adds them to the job queue, and runs them in the cluster when nodes become available.
- *Client* programs submit jobs to the master.
- *Slaves* are started by the master on each node in the cluster. They spawn and monitor all processes that run on their respective nodes.
- *Workers* perform job *tasks*. The locations of their output results are sent to the master.
- Once files are stored in the Disco cluster, Disco tries to maintain *data locality* by scheduling tasks which use those files as input on the same nodes that the files are stored on. Disco runs an HTTP server on each node so that data can be accessed remotely, when a worker cannot be run on the same node that its input is located.

Users can limit the number of workers run in parallel on each node. Thus, users can easily designate the cluster to run as many tasks as there are available CPUs, disks, or any other number.

If high availability of the system is a concern, CPUs in the cluster can be partitioned amongst arbitrary many Disco masters. This way several Disco masters can co-exist, which eliminates the only single point of failure in the system.

1.3 Disco FAQ

Common Questions

- I tried to install Disco but it doesn't work. Why?
- How come `ssh localhost erl` doesn't use my normal `$PATH`?
- How do I profile programs in Disco?
- How do I debug programs in Disco?
- Do I always have to provide a function for map and reduce?
- How many maps can I have? Does a higher number of maps lead to better performance?
- I have one big data file, how do I run maps on it in parallel?
- How do I pass the output of one map-reduce phase to another?
- How do I print messages to the Web interface from Python?
- My input files are stored in CSV / XML / XYZ format. What is the easiest to use them in Disco?
- Why not Hadoop?
- How do I use Disco on Amazon EC2?

1.3.1 I tried to install Disco but it doesn't work. Why?

See *Troubleshooting Disco installation*. If the problem persists, contact Disco developers *on IRC or the mailing list*.

1.3.2 How come `ssh localhost erl` doesn't use my normal `$PATH`?

```
ssh localhost erl
```

is different from:

```
ssh localhost
erl
```

In general, interactive shells behave differently than non-interactive ones. For example, see the Bash Reference Manual.

1.3.3 How do I profile programs in Disco?

Disco can use the Profile module to profile map and reduce tasks written in Python. Enable profiling by setting `profile = True` in your `disco.job.Job`.

Here's a simple example:

```
"""
Try running this from the ``examples/faq/`` directory using:

disco run profile.ProfileJob http://example.com/data | xargs disco wait && xargs disco pstats -k
"""
from disco.job import Job

class ProfileJob(Job):
    profile = True

    @staticmethod
    def map(entry, params):
        yield entry.strip(), 1
```

See Also:

`disco.core.Disco.profile_stats()` for accessing profiling results from Python.

1.3.4 How do I debug programs in Disco?

Set up a single node Disco cluster locally on your laptop or desktop. It makes debugging a Disco job almost as easy as debugging any Python script.

1.3.5 Do I always have to provide a function for map and reduce?

No, you may specify either *map* or *reduce* or both. Many simple tasks can be solved with a single map function, without reduce.

It is somewhat less typical to specify only the reduce function. This case arises when you want to merge results from independent map jobs, or you want to join several input files without going through the map phase.

See also: *Data Flow in Disco Jobs*

1.3.6 How many maps can I have? Does a higher number of maps lead to better performance?

In theory there is no restriction. In practice, the number is of course limited by the available disk space (for input files) and the amount of RAM that is required by the Disco master. Disco includes a test case, in `tests/test_50k.py` that starts 50,000 map tasks in parallel. You should be able to add a few zeroes there without any trouble. If you perform any stress tests of your own, let us know about your findings!

Each map and reduce instance is allocated exclusive access to a CPU. This means that the number of parallel processes is limited by the number of available CPUs. If you have 50,000 map instances but only 50 CPUs, only 50 maps are run in parallel while 49,550 instances are either waiting in the job queue or marked as ready — assuming that no other jobs are running in the system at the same time and your input is split to at least 50,000 separate files.

The number of maps can never exceed the number of input files as Disco can't order many maps to process a single input file. In other words, to run K maps in parallel you need at least K input files. See *Pushing Chunked Data to DDFS* for more on splitting data stored in *Disco Distributed Filesystem*.

In general, the question about the expected speedup when increasing parallelism is a rather complicated one and it depends heavily on the task at hand. See Amdahl's Law for more information about the subject. However, unless your tasks are so light that the execution time is dominated by the overhead caused by Disco, you can expect to gain some speedup by adding more maps until the number of maps equals to the number of available CPUs.

1.3.7 I have one big data file, how do I run maps on it in parallel?

See *Pushing Chunked Data to DDFS*.

1.3.8 How do I pass the output of one map-reduce phase to another?

Many algorithms can be implemented cleanly as a sequence of *mapreduce jobs*. Chaining jobs together is also efficient, as the job's results are readily distributed and stored in Disco's internal format.

Here's an example that runs ten jobs in a sequence, using outputs from the previous job as the input for the next one. The code can also be found in `examples/faq/chain.py`. The job increments each value in the input by one:

```
from disco.job import Job
from disco.worker.classic.func import chain_reader

class FirstJob(Job):
    input = ['raw://0', 'raw://0']
```

```
        @staticmethod
        def map(line, params):
            yield int(line) + 1, ""

    class ChainJob(Job):
        map_reader = staticmethod(chain_reader)

        @staticmethod
        def map(key_value, params):
            yield int(key_value[0]) + 1, key_value[1]

    if __name__ == "__main__":
        # Jobs cannot belong to __main__ modules.  So, import this very
        # file to access the above classes.
        import chain
        last = chain.FirstJob().run()
        for i in range(9):
            last = chain.ChainJob().run(input=last.wait())
        print(last.name)
```

Assuming that the input files consists of zeroes, this example will produce a sequence of tens as the result.

1.3.9 How do I print messages to the Web interface from Python?

Use a normal Python **print** statement.

Note: This is meant for simple debugging, if you print messages too often, Disco will throttle your worker. The master limits the rate of messages coming from workers, to prevent it from being overwhelmed.

Internally, Disco wraps everything written to `sys.stdout` with appropriate markup for the Erlang worker process, which it communicates with via `sys.stderr`. See also *The Disco Worker Protocol*.

1.3.10 My input files are stored in CSV / XML / XYZ format. What is the easiest to use them in Disco?

See `disco.worker.classic.func.input_stream()`.

For CSV files you can also have a look at the csv module shipped in the Python standard library.

1.3.11 Why not Hadoop?

We see that platforms for distributed computing will be of such high importance in the future that it is crucial to have a wide variety of different approaches which produces healthy competition and co-evolution between the projects. In this respect, Hadoop and Disco can be seen as complementary projects, similar to Apache, Lighttpd and Nginx.

It is a matter of taste whether Erlang and Python are more suitable for the task than Java. We feel much more productive with Python than with Java. We also feel that Erlang is a perfect match for the Disco core that needs to handle tens of thousands of tasks in parallel.

Thanks to Erlang, the Disco core is remarkably compact. It is relatively easy to understand how the core works, and start experimenting with it or adapt it to new environments. Thanks to Python, it is easy to add new features around the core which ensures that Disco can respond quickly to real-world needs.

1.3.12 How do I use Disco on Amazon EC2?

In general, you can use the EC2 cluster as any other Disco cluster. However, if you want to access result files from your local machine, you need to set the `DISCO_PROXY` setting. This configures the master node as a proxy, since the computation nodes on EC2 are not directly accessible.

Hint: For instance, you could open an SSH tunnel to the master:

```
ssh MASTER -L 8989:localhost:8989
```

and set `DISCO_PROXY=http://localhost:8989`.

1.4 Release notes

1.4.1 Disco 0.4.5 (Mar 28, 2013)

Changes

- Disco documentation is now also at ReadTheDocs, along with documentation for DiscoDB.
- Mochiweb has been updated to fix compilation issues with Erlang 16B, which removed support for parameterized modules.
- Disco debian packages are no longer hosted on discoproject.org. Instead, Debian/Ubuntu users are encouraged to build their own packages for their particular Erlang/Python environment using the `make-discoproject-debian` script in the source tree. Please read the comments in the script.

Bugfixes

- Fix `ddfs xcat` display output, thanks to John Emhoff.
- Fix `disco jobdict` command (#341).
- Clarify the documentation in several places, thanks to feedback from Pavel Hančar, and fixes from John Emhoff.
- Fix a formatting bug in `disco.util:urljoin`.
- Fixed job deletion from UI when job has quotes in name, thanks to @nvdev on Github.
- Ensure that *known* garbage in DDFS is deleted immediately, without waiting for the safety timeout required for blobs and tags of indeterminate status.

1.4.2 Disco 0.4.4 (Dec 5, 2012)

New features

- The Python client library should now be Python3 compatible (version 3.2 or higher). As usual, the Python versions on the client and in the Disco cluster should match; mixed configurations are not supported. Since Python3 differentiates between string and unicode objects, Disco jobs will need to do the same. In particular, the default *map_reader* will provide `bytes` objects to the `map` function.
- Client and master version commands have been added to the `disco` command-line interface (issue #283). Currently, the client version command only works for Disco installed as a python egg.

- Installation support for NetBSD, thanks to Yamamoto Takashi.
- There is now a script to ease the creation of Disco debian packages, used to create the Debian packages provided from discoproject.org. Note that this script does *not* follow Debian packaging guidelines; use at your own risk!
- Small efficiency and logging improvements to DDFS.

Changes

- The `disco` and `ddfs` command-line scripts are now packaged as part of python-disco Debian package, so that they can be used on clients. Thanks to Daniel Graña.

Bugfixes

- `disco.ddfs.DDFS.pull()` should now obey DISCO_PROXY settings. Thanks to Daniel Graña.
- Intercept Python warning messages to sys.stderr, which break the Disco worker protocol. They are now logged as messages. Thanks to Daniel Graña.
- The HTTP header handling in the Disco client library is more case-resilient.

1.4.3 Disco 0.4.3 (Aug 22, 2012)

New features

- An extended Disco tutorial, thanks to Davin Potts.
- More documentation on using the proxy mode, and recovering from a master failure.
- More efficient (faster and using less memory) event_server, which should speed up UI responses for large jobs.
- Better fault-tolerance in re-replication, which should speed up node-removal. Node-removal of more than one node is now better tested and supported.
- Less unnecessary creation of garbage tags in DDFS, by avoiding creating new tag incarnations when their content has not changed. Since less garbage is created, GC will now complete more quickly.
- A "local-cluster" mode for DDFS, that simulates a multi-node DDFS cluster on a single machine. This is purely a developer feature for the purpose of improving DDFS testing, and cannot be used for running Disco jobs using DDFS. Thanks to Harry Nakos.

Changes

- Change the default partition function to use the key hash directly, instead of the string version of the key; this should address some unicode failures (#265). Thanks to quasiben and tmielika.
- Improved logging, especially to track re-replication progress.
- Major cleanup of Erlang codebase.

Bugfixes

- More fixes to DISCO_PROXY mode (#269). This mode is required for using DDFS in the "local cluster" mode.
- Fix a race when the UI tried to access information for a job that had been submitted but not yet unpacked (#304).

1.4.4 Disco 0.4.2 (Apr 26, 2012)

New features

- New fault-tolerant garbage collector and re-replicator (GC/RR).
- Allow scheduling of nodes for safe removal from DDFS (#201).
- Some useful GC statistics are now shown in the UI.

Changes

- Discodb and Discodex separated out into submodule repositories.
- Master/Erlang build switched to rebar, with source tree re-organized appropriately.
- Master logging switched to lager. Note that the format of the logs has changed as a result.
- Many dialyzer-related cleanups. Thanks to Kostis Sagonas.
- Cleanup of debian package build.

Bugfixes

- The new GC/RR closes #254, where a concurrent update to a tag was not handled at some points during GC.
- The new GC/RR also closes #256, where lost tag updates for re-replicated blobs caused later re-replication failures.
- Fix a case when the master node could run out of file descriptors when servicing an unexpectedly large number of jobpack requests from worker nodes (20d8fbe, 10a33b9, 0f7eaeb).
- Fixes to make DISCO_PROXY usable again (#269). Thanks to Dmitrijs Milajevs.
- Fix a crash due to an already started lock server (64096a3).
- Handle an existing disco user on package install (4f04e14). Thanks to Pedro Larroy.
- Fix a crash of ddfs_master due to timeouts in linked processes (#312).

1.4.5 Disco 0.4.1 (Sep 23rd 2011)

The official Disco repository is now at http://github.com/discoproject/disco

New features

- DiscoDB: *ddb_cursor_count()* added. `iterator.count()` is now faster.
- DiscoDB: Value lists are now stored in deltalists instead of lists during discodb construction, resulting to 50-75% smaller memory footprint in the many-values-per-key case.

Bugfixes

- Fix GC timeout issue (#268).
- Fix regression in Temp GC (09a1debb). Thanks to Jamie Brandon.
- Improved and fixed documentation. Thanks to Jens Rantil, stillinbeta and Luke Hoersten.
- Fix chunking. Thanks to Daniel Grana.
- Minor fixes in DiscoDB.
- Fix a bug in job pack extraction (e7b3b6).

1.4.6 Disco 0.4 (May 4th 2011)

New features

- *The Disco Worker Protocol* introduced to support custom *workers*, especially in languages besides Python (see ODisco for an OCaml worker now included in `contrib`).
- Complete overhaul of the Python `disco.worker` to support the new protocol. Most notably the worker is now completely self-contained - you do not have to install Python libraries on slave nodes anymore.
- *Job History* makes using the command-line less tedious. Several other enhancements to `disco` and `ddfs` command line tools.
- *Setting up Disco* is easier than ever. Updated Debian packaging and dependencies make *Installing Disco System-Wide* a breeze.
- More documentation, including a *DiscoDB Tutorial* using extended `disco.job.Job` classes.
- Throttling of messages coming from the worker, to prevent them from overwhelming the master without killing the process.
- Upgraded to mochiweb 2.0.
- Support for log rotation on the *master* via `DISCO_ROTATE_LOG`.
- *prefix* is now optional for jobs.
- Many Dialyzer-related improvements.
- Separate Debian branch containing rules to create Debian packages merged under `pkg`.
- Debian package for DiscoDB.
- *disco.worker.classic.external - Classic Disco External Interface* provides the task type on the command line, to allow a single binary to handle both map and reduce phases.

Bugfixes

- **DDFS:**
  - **important** Recreating a previously deleted tag with a token did not work correctly. The call returned without an error but the tag was not created.
  - Under some circumstances DDFS garbage collector deleted .partial files, causing PUT operations to fail (6deef33f).
- Redundant inputs using the `http://` scheme were not handled correctly (`disco://` scheme worked ok) (9fcc740d).

- Fix *eaddrinuse* errors caused by already running nodes (1eed58d08).
- Fix newlines in error messages in the web UI.
- The web UI no longer loses the filter when the events are refreshed.
- Several fixes in *node_mon*. It should handle unavailable nodes now more robustly.
- The OOB issue (#227) highlighted below became a non-issue as GC takes care of removing OOB results when the job is garbage collected.
- Fix the issue with the job starting even when the client got an error when submitting a new job.

Deprecated

- `disco.util.data_err()`, `disco.util.err()`, and `disco.util.msg()`, have all been deprecated in favor of using `raise` and `print` statements.
- Jobs without inputs i.e. generator maps: See the *raw://* protocol in `disco.core.Disco.new_job()`.
- *map_init* and *reduce_init* deprecated. Use *input_stream* or *reader* instead.
- *scheme_dfs* removed.
- Deprecated `DDFS_ROOT` setting, use `DDFS_DATA` instead.

1.4.7 Disco 0.3.2 (Dec 6th 2010)

Note: In contrast to earlier releases, in `0.3.2` purging a job does not delete OOB results of the job automatically. This is listed as issue #227 and will be fixed in the next release together with other changes in OOB handling. Meanwhile, you can use `disco.ddfs.DDFS.delete()` to delete OOB results if needed.

New features

- Built-in support for chunking large inputs (see *Tutorial* and `disco.ddfs.DDFS.chunk()`).
- List of blacklisted nodes is persistent over restarts.
- Disconnected nodes are now highlighted in the web UI.
- Explicit hostname (`tag://host/tag`) is now allowed in tag urls.
- **Some commonly used functions added to `disco.func`:**
  - `disco.func.gzip_line_reader()`
  - `disco.func.sum_combiner()`
  - `disco.func.sum_reduce()`
- Job owner shown in the web UI (can be overridden with the `DISCO_JOB_OWNER` setting).
- `DISCO_WORKER_MAX_MEM` setting can be used to limit the maximum amount of memory that can be used by a worker process.
- ***Disco Distributed Filesystem*:**
  - Tags can now contain arbitrary user-defined attributes (see *DDFS APIs* and `disco.ddfs.DDFS.setattr()` and `disco.ddfs.DDFS.getattr()`).
  - Basic token-based permission control for tags (see *DDFS APIs*).

  - Improved REST API (see *DDFS APIs*).
  - `DDFS_PARANOID_DELETE` setting allows an external program to be used to delete or verify obsolete files (see `disco.settings`).
- Functions are now allowed in arguments of partial job functions.
- Improved documentation, and a new document *Administering Disco*.

Bugfixes

- Several bugfixes in DDFS garbage collection.
- Tasks may be marked successful before results are persisted to disk (#208).
- Improved error handling for badly dying tasks (#162).
- Allow dots in DDFS paths (#196).
- Improved handling of out of memory conditions (#168, #200).
- Fix blocking `net_adm:names` in `node_mon` (#216).
- Fix a badmatch error on unknown jobname (#81).
- Fixed error handling if sort fails.
- Tutorial example fixed.
- HTTP error message made more informative.

1.4.8 Disco 0.3.1 (Sep 1st 2010)

Note: This release fixes a serious bug in how partition files are handled under certain error conditions. The bug has existed since Disco 0.1.

If a node becomes unavailable, for instance due to network congestion, master restarts the tasks that were running on the failed node on other nodes. However, it is possible that old tasks continue running on the failed node, producing results as usual. This can lead to duplicate entries being written to result files.

Note that not all task failures are suspectible to this bug. If the task itself fails, which is the most typical error scenario, Disco ensures that results are still valid. Only if your job events have contained messages like `Node unavailable` or `Connection lost to the node`, it is possible that results are invalid and you should re-run the suspected jobs with Disco 0.3.1 or newer.

This bug also revealed a similar issue with jobs that save their results to DDFS with `save=True` (available since Disco 0.3). It is possible that duplicate tasks create duplicate entries in the result tag. This is easy to detect and fix afterwards by listing urls in the tag and ensuring that there are no duplicates. A script is provided at `util/fix-jobtag` that can be used to check and fix suspected tags.

New features

- **Improved robustness and scalability:**
  - Jobs are now immortal by default; they should never fail due to temporary errors unless a user-defined limit is reached.
  - New shuffle phase to optimize intermediate results for reduce.

  - Support for Varnish for `DISCO_PROXY`. In some cases, Varnish can be over three times faster than Lighttpd.

- ***Disco Distributed Filesystem*:**
  - Improved blob placement policy.
  - Atomic set updates (`update=1`).
  - Delayed commits (`delayed=1`), which gives a major performance boost without sacrificing data consistency.
  - Garbage collection is now scheme-agnostic (#189).
- **Major DiscoDB enhancements:**
  - Values are now compressed without sacrificing performance.
  - Constructor accepts unsorted key-value pairs.
  - Option (`unique_items=True`) to remove duplicates from inputs automatically.
  - `unique_values()` iterator.
- Alternative signature for reduce: Reduce can now `yield` key-value pairs (or return an iterator) instead of calling `out.add()` (see `disco.func.reduce2()`).
- Enhanced Java support added as a Git submodule under `contrib/java-ext` (Thanks to Ryan Maus).
- Disk space monitoring for DDFS added to the Web UI.
- Lots of enhancements to `disco` command line.
- New setting `DISCO_SORT_BUFFER_SIZE` to control memory usage of the external sort (see `disco.settings`).
- `disco.func.gzip_reader()` for reading gzipped inputs.
- Easier single-node installation with default localhost configuration.

Deprecated

- **Important!** The default reader function, `disco.func.map_line_reader()`, will be deprecated. The new default is to iterate over the object returned by *map_reader*. In practice, the default *map_reader* will still return an object that iterates over lines. However, it will not strip newline characters from the end of lines as the old `disco.func.map_line_reader()` does.

  Make sure that your jobs that rely on the default *map_reader* will handle newline characters correctly. You can do this easily by calling `string.strip()` for each line.

Backwards incompatible changes

- Installation script for Amazon EC2 removed (`aws/setup-instances.py`) and documentation updated accordingly (see *How do I use Disco on Amazon EC2?*). Disco still works in Amazon EC2 and other similar environments flawlessly but a more modern mechanism for easy deployments is needed.

Bugfixes

- **Critical** bug fixes to fix partition file handling and `save=True` behavior under temporary node failures (see a separate note above).

- Delayed commits in DDFS fix OOB slowness (#155)
- Fix unicode handling (#185, #190)
- In-memory sort disabled as it doesn't work well compressed inputs (#145)
- Fixed/improved replica handling (#170, #178, #176)
- Three bugfixes in DiscoDB querying and iterators (#181)
- Don't rate limit internal messages, to prevent bursts of messages crashing the job (#169)
- Random bytes in a message should not make json encoding fail (#161)
- `disco.core.Disco.wait()` should not throw an exception if master doesn't respond immediately (#183)
- Connections should not fail immediately if creating a connection fails (#179)
- Fixed an upload issue in `comm_pycurl.py` (#156)
- Disable HTTP keep-alive on master.
- Sort failing is not a fatal error.
- Partitioned only-reduce did not check the number of input partitions correctly.
- `DISCO_PROXY` did not work correctly if disco was run with a non-standard port.
- `node_mon` didn't handle all messages from nodes correctly, which lead its message queue to grow, leading to spurious `Node unavailable` messages.
- Fix mouse-over for showing active cores in the status page.

1.4.9 Disco 0.3 (May 26th 2010)

New features

- *Disco Distributed Filesystem* - distributed and replicated data storage for Disco.
- Discodex - distributed indices for efficient querying of data.
- DiscoDB - lightning fast and scalable mapping data structure.
- New internal data format, supporting compression and pickling of Python objects by default.
- Clarified the partitioning logic in Disco, see *Data Flow in Disco Jobs*.
- Integrated web server (Mochiweb) replaces Lighttpd, making installation easier and allows more fine-grained data flow control.
- Chunked data transfer and improved handling of network congestion.
- Support for partial job functions (Thanks to Jarno Seppänen)
- Unified interface for readers and input streams, writers deprecated. See `disco.core.Disco.new_job()`.
- New `save=True` parameter for `disco.core.Disco.new_job()` which persists job results in DDFS.
- New garbage collector deletes job data `DISCO_GC_AFTER` seconds after the job has finished (see `disco.settings`). Defaults to 100 years. Use `save=True`, if you want to keep the results permanently.
- Support for Out-of-band (OOB) results implemented using DDFS.
- `disco-worker` checks that there is enough disk space before it starts up.
- `discocli` - Command line interface for Disco

- `ddfscli` - Command line interface for DDFS
- Improved load balancing in scheduler.
- Integrated Disco proxy based on Lighttpd.
- Debian packaging: `disco-master` and `disco-node` do not conflict anymore, making it possible to run Disco locally from Debian packages.

Deprecated

These features will be removed in the coming releases:

- *object_reader* and *object_writer* - Disco supports now pickling by default.
- *map_writer* and *reduce_writer* (use output streams instead).
- *nr_reduces* (use *partitions*)
- *fun_map* and *input_files* (use *map* and *input*)

Backwards incompatible changes

- Experimental support for GlusterFS removed
- `homedisco` removed - use a local Disco instead
- Deprecated `chunked` parameter removed from `disco.core.Disco.new_job()`.
- If you have been using a custom output stream with the default writer, you need to specify the writer now explictly, or upgrade your output stream to support the *.out(k, v)‘* method which replaces writers in 0.3.

Bugfixes

- Jobs should disappear from list immediately after deleted (bug #43)
- Running jobs with empty input gives “Jobs status dead” (bug #92)
- Full disk may crash a job in *_safe_fileop()* (bug #120)
- Eventmonitor shows each job multiple times when tracking multiple jobs (bug #94)
- Change eventmonitor default output handle to sys.stderr (bug #83)
- Tell user what the spawn command was if the task fails right away (bug #113)
- Normalize pathnames on PYTHONPATH (bug #134)
- Timeouts were handled incorrectly in wait() (bug #96)
- Cast unicode urls to strings in comm_curl (bug #52)
- External sort handles objects in values correctly. Thanks to Tomaž Šolc for the patch!
- Scheduler didn’t handle node changes correctly - this solves the hanging jobs issue
- Several bug fixes in *comm_*.py*
- Duplicate nodes on the node config table crashed master
- Handle timeout correctly in fair_scheduler_job (if system is under heavy load)

1.4.10 Disco 0.2.4 (February 8th 2010)

New features

- New fair job scheduler which replaces the old FIFO queue. The scheduler is inspired by Hadoop's Fair Scheduler. Running multiple jobs in parallel is now supported properly.
- *Scheduler* option to control data locality and resource usage. See `disco.core.Disco.new_job()`.
- Support for custom input and output streams in tasks: See *map_input_stream*, *map_output_stream*, *reduce_input_stream* and *reduce_output_stream* in `disco.core.Disco.new_job()`.
- `disco.core.Disco.blacklist()` and `disco.core.Disco.whitelist()`.
- New test framework based on Python's unittest module.
- Improved exception handling.
- Improved IO performance thanks to larger IO buffers.
- Lots of internal changes.

Bugfixes

- Set `LC_ALL=C` for disco worker to ensure that external sort produces consistent results (bug #36, 7635c9a)
- Apply rate limit to all messages on stdout / stderr. (bug #21, db76c80)
- Fixed *flock* error handing for OS X (b06757e4)
- Documentation fixes (bug #34, #42 9cd9b6f1)

1.4.11 Disco 0.2.3 (September 9th 2009)

New features

- The `disco.settings` control script makes setting up and running Disco much easier than before.
- Console output of job events (screenshot). You can now follow progress of a job on the console instead of the web UI by setting `DISCO_EVENTS=1`. See `disco.core.Disco.events()` and `disco.core.Disco.wait()`.
- Automatic inference and distribution of dependent modules. See `disco.modutil`.
- *required_files* parameter added to `disco.core.Disco.new_job()`.
- Combining the previous two features, a new easier way to use external C libraries is provided, see *disco.worker.classic.external - Classic Disco External Interface*.
- Support for Python 2.6 and 2.7.
- Easier installation of a simple single-server cluster. Just run `disco master start` on the disco directory. The `DISCO_MASTER_PORT` setting is deprecated.
- Improved support for OS X. The `DISCO_SLAVE_OS` setting is deprecated.
- Debian packages upgraded to use Erlang 13B.
- Several improvements related to fault-tolerance of the system
- Serialize job parameters using more efficient and compact binary format.

- Improved support for GlusterFS (2.0.6 and newer).
- Support for the pre-0.1 `disco` module, `disco.job` call etc., removed.

Bugfixes

- **critical** External sort didn't work correctly with non-numeric keys (5ef88ad4)
- External sort didn't handle newlines correctly (61d6a597f)
- Regression fixed in `disco.core.Disco.jobspec()`; the function works now again (e5c20bbfec4)
- Filter fixed on the web UI (bug #4, e9c265b)
- Tracebacks are now shown correctly on the web UI (bug #3, ea26802ce)
- Fixed negative number of maps on the web UI (bug #28, 5b23327 and 3e079b7)
- The `comm_curl` module might return an insufficient number of bytes (761c28c4a)
- Temporary node failure (noconnection) shouldn't be a fatal error (bug #22, ad95935)
- *nr_maps* and *nr_reduces* limits were off by one (873d90a7)
- Fixed a Javascript bug on the config table (11bb933)
- Timeouts in starting a new worker shouldn't be fatal (f8dfcb94)
- The connection pool in `comm_httplib` didn't work correctly (bug #30, 5c9d7a88e9)
- Added timeouts to `comm_curl` to fix occasional issues with the connection getting stuck (2f79c698)
- All *IOErrors* and *CommExceptions* are now non-fatal (f1d4a127c)

1.4.12 Disco 0.2.2 (July 26th 2009)

New features

- Experimental support for POSIX-compatible distributed filesystems, in particular GlusterFS. Two modes are available: Disco can read input data from a distributed filesystem while preserving data locality (aka *inputfs*). Disco can also use a DFS for internal communication, replacing the need for node-specific web servers (aka *resultfs*).

Bugfixes

- `DISCO_PROXY` handles now out-of-band results correctly (commit b1c0f9911)
- *make-lighttpd-proxyconf.py* now ignores commented out lines in */etc/hosts* (bug #14, commit a1a93045d)
- Fixed missing PID file in the *disco-master* script. The */etc/init.d/disco-master* script in Debian packages now works correctly (commit 223c2eb01)
- Fixed a regression in *Makefile*. Config files were not copied to */etc/disco* (bug #13, commit c058e5d6)
- Increased *server.max-write-idle* setting in Lighttpd config. This prevents the http connection from disconnecting with long running, cpu-intensive reduce tasks (bug #12, commit 956617b0)

1.4.13 Disco 0.2.1 (May 26th 2009)

New features

- Support for redundant inputs: You can now specify many redundant addresses for an input file. Scheduler chooses the address which points at the node with the lowest load. If the address fails, other addresses are tried one by one until the task succeeds. See *inputs* in `disco.core.Disco.new_job()` for more information.
- Task profiling: See *How do I profile programs in Disco?*
- Implemented an efficient way to poll for results of many concurrent jobs. See `disco.core.Disco.results()`.
- Support for the Curl HTTP client library added. Curl is used by default if the `pycurl` module is available.
- Improved storing of intermediate results: Results are now spread to a directory hierarchy based on the md5 checkum of the job name.

Bugfixes

- Check for `ionice` before using it. (commit dacbbbf785)
- `required_modules` didn't handle submodules (PIL.Image etc.) correctly (commit a5b9fcd970)
- Missing file balls.png added. (bug #7, commit d5617a788)
- Missing and crashed nodes don't cause the job to fail (bug #2, commit 6a5e7f754b)
- Default value for nr_reduces now never exceeds 100 (bug #9, commit 5b9e6924)
- Fixed homedisco regression in 0.2. (bugs #5, #10, commit caf78f77356)

1.4.14 Disco 0.2 (April 7th 2009)

New features

- *Out-of-band results*: A mechanism to produce auxiliary results in map/reduce tasks.
- Map writers, reduce readers and writers (see `disco.core.Disco.new_job()`): Support for custom result formats and internal protocols.
- Support for arbitrary output types.
- Custom task initialization functions: See *map_init* and *reduce_init* in `disco.core.Disco.new_job()`.
- Jobs without inputs i.e. generator maps: See the *raw://* protocol in `disco.core.Disco.new_job()`.
- Reduces without maps for efficient join and merge operations: See *Do I always have to provide a function for map and reduce?*.

Bugfixes

(NB: bug IDs in 0.2 refer to the old bug tracking system)

- `chunked = false` mode produced incorrect input files for the reduce phase (commit db718eb6)
- Shell enabled for the disco master process (bug #7, commit 7944e4c8)
- Added warning about unknown parameters in `new_job()` (bug #8, commit db707e7d)

- Fix for sending invalid configuration data (bug #1, commit bea70dd4)
- Fixed missing `msg`, `err` and `data_err` functions (commit e99a406d)

1.5 Glossary

client The program which submits a *job* to the *master*.

blob An arbitrary file stored in *Disco Distributed Filesystem*.

See also *Blobs*.

data locality Performing computation over a set of data near where the data is located. Disco preserves *data locality* whenever possible, since transferring data over a network can be prohibitively expensive when operating on massive amounts of data.

See locality of reference.

DDFS See *Disco Distributed Filesystem*.

Erlang See Erlang.

garbage collection (GC) DDFS has a tag-based filesystem, which means that a given blob could be addressed via multiple tags. This means that blobs can only be deleted once the last reference to it is deleted. DDFS uses a garbage collection procedure to detect and delete such unreferenced data.

immutable See immutable object.

map The first phase of a *job*, in which *tasks* are usually scheduled on the same node where their input data is hosted, so that local computation can be performed.

Also refers to an individual task in this phase, which produces records that may be *partitioned*, and *reduced*. Generally there is one map task per input.

master Distributed core that takes care of managing *jobs*, garbage collection for *DDFS*, and other central processes.

See also *Technical Overview*.

job A set of map and/or reduce *tasks*, coordinated by the Disco *master*. When the master receives a `disco.job.JobPack`, it assigns a unique name for the job, and assigns the tasks to *workers* until they are all completed.

See also `disco.job`

job functions Job functions are the functions that the user can specify for a `disco.worker.classic.worker`. For example, `disco.worker.classic.func.map()`, `disco.worker.classic.func.reduce()`, `disco.worker.classic.func.combiner()`, and `disco.worker.classic.func.partition()` are job functions.

job dict The first field in a *job pack*, which contains parameters needed by the master for job execution.

See also *The Job Dict* and `disco.job.JobPack.jobdict`.

job home The working directory in which a *worker* is executed. The *master* creates the *job home* from a *job pack*, by unzipping the contents of its *jobhome* field.

See also *The Job Home* and `disco.job.JobPack.jobhome`.

job pack The packed contents sent to the master when submitting a new job. Includes the *job dict* and *job home*, among other things.

See also *The Job Pack* and `disco.job.JobPack`.

JSON JavaScript Object Notation.

See Introducing JSON.

mapreduce A paradigm and associated framework for distributed computing, which decouples application code from the core challenges of fault tolerance and data locality. The framework handles these issues so that *jobs* can focus on what is specific to their application.

See MapReduce.

partitioning The process of dividing output records into a set of labelled bins, much like *tags* in *DDFS*. Typically, the output of *map* is partitioned, and each *reduce* operates on a single partition.

pid A process identifier. In Disco this usually refers to the *worker pid*.

See process identifier.

reduce The last phase of a *job*, in which non-local computation is usually performed.

Also refers to an individual *task* in this phase, which usually has access to all values for a given key produced by the *map* phase. Grouping data for reduce is achieved via *partitioning*.

replica Multiple copies (or replicas) of blobs are stored on different cluster nodes so that blobs are still available inspite of a small number of nodes going down.

re-replication When a node goes down, the system tries to create additional replicas to replace copies that were lost at the loss of the node.

SSH Network protocol used by *Erlang* to start *slaves*.

See SSH.

slave The process started by the *Erlang* slave module.

See also *Technical Overview*.

stdin The standard input file descriptor. The *master* responds to the *worker* over *stdin*.

See standard streams.

stdout The standard output file descriptor. Initially redirected to *stderr* for a Disco *worker*.

See standard streams.

stderr The standard error file descriptor. The *worker* sends messages to the *master* over *stderr*.

See standard streams.

tag A labelled collection of data in *DDFS*.

See also *Tags*.

task A *task* is essentially a unit of work, provided to a *worker*. A Disco *job* is made of *map* and *reduce* tasks.

See also `disco.task`.

worker A *worker* is responsible for carrying out a *task*. A Disco *job* specifies the executable that is the worker. Workers are scheduled to run on the nodes, close to the data they are supposed to be processing.

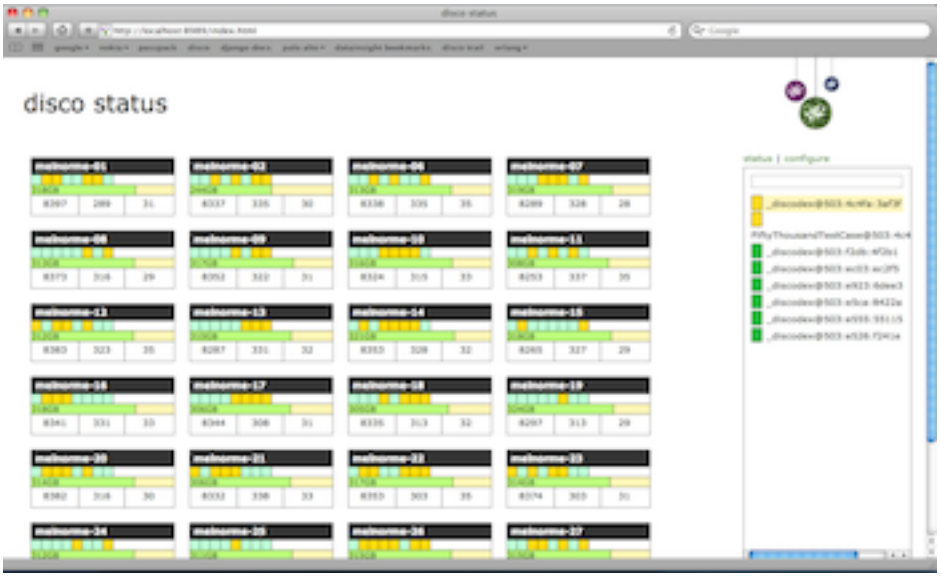
See Also:

`The Python Worker module`, and *The Disco Worker Protocol*.

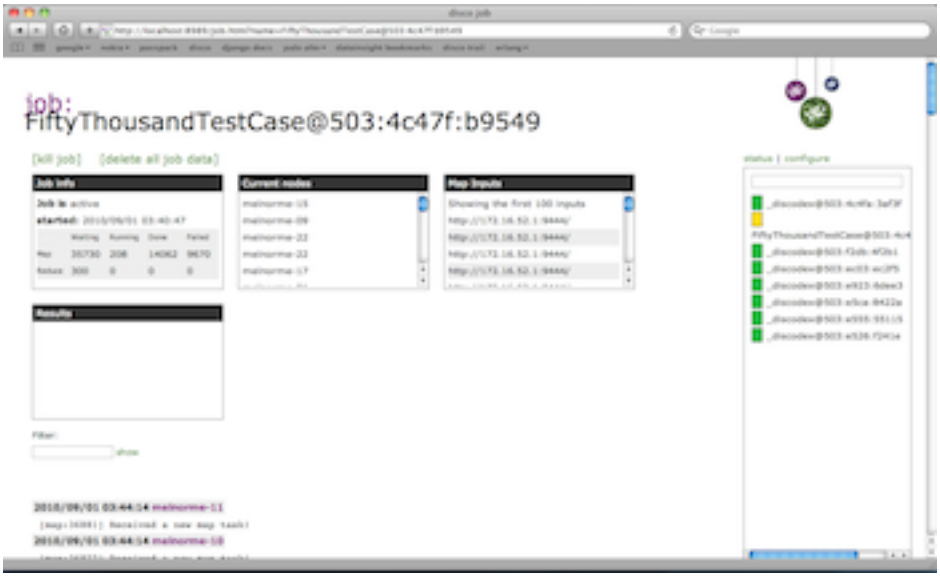
ZIP Archive/compression format, used e.g. for the *job home*.

See ZIP.

1.6 Screenshots



Main screen



Job status page

Console output of job events

CHAPTER 2

Getting started

2.1 Get Disco

2.1.1 Latest release

Get one of the official releases and follow the *installation instructions* after downloading the package.

2.1.2 Development branches

Clone a bleeding edge version from github:

```
git clone git://github.com/discoproject/disco.git
```

If compiling from source, you cannot use the zip or tar.gz packages generated by github, but must instead get the git repo using the above command.

2.1.3 Debian packages

If you use Debian or a Debian-based distribution such as Ubuntu, on the AMD64 architecture, you can just *apt-get install* Disco. Add the following line to your */etc/apt/sources.list*:

```
deb http://discoproject.org/debian /
```

After running *apt-get update*, you can install the *disco-master* package to your master node. If you run Disco in a cluster, you should install the *disco-node* package to other nodes. The *python-disco* package is required on all machines where Disco scripts are run.

After installation, see steps 4-6 in *Setting up Disco* that describe how to configure and test Disco.

Following distributions are supported

- *python-disco* works on Debian stable (Lenny) and newer
- *disco-master* and *disco-node* work on Debian testing (Squeeze) and newer
- All packages should work on recent Ubuntu releases

This means that Debian stable can be used to submit Disco jobs but not to run Disco master or nodes.

Warning: Our Debian packages are experimental! They may not play nicely with other packages or they may destroy your computer.

2.2 Setting up Disco

This document helps you to install Disco from source, either on a single server or a cluster of servers. This requires installation of some *Prerequisites*.

See Also:

Installing Disco System-Wide.

2.2.1 Background

You should have a quick look at *Technical Overview* before setting up the system, to get an idea what should go where and why. To make a long story short, Disco works as follows:

- Disco users start Disco jobs in Python scripts.
- Jobs requests are sent over HTTP to the master.
- Master is an *Erlang* process that receives requests over HTTP.
- Master launches *slaves* on each node over *SSH*.
- Slaves run Disco tasks in *worker* processes.

2.2.2 Prerequisites

You need at least one Linux/Unix server. Any distribution should work (including Mac OS X).

On each server the following are required:

- SSH daemon and client
- Erlang/OTP R14A or newer
- Python 2.6.6 or newer, or Python 3.2 or newer

The same version of Erlang and Python should be installed on all servers. The default version of Python on the clients from which Disco jobs are submitted should also match that on the servers.

Optionally, `DISCO_PROXY` needs one of

- Lighttpd 1.4.17 or newer
- Varnish 2.1.3 or newer

Due to issues with unicode in Python2's httplib library, we recommend installing the pycurl package. Disco will transparently use pycurl when available.

2.2.3 Install Disco

Short Version

```
git clone git://github.com/discoproject/disco.git DISCO_HOME
cd DISCO_HOME
make
bin/disco nodaemon
```

Hint: Its convenient to add the `disco` command to your path.

Long Version

Download *a recent version of Disco*.

Extract the package (if necessary) and `cd` into it. We will refer to this directory as `DISCO_HOME`.

Now compile Disco:

```
make
```

This is often the easiest and the least intrusive way to get started with Disco.

You should repeat the above command on all machines in your Disco cluster.

Note: Disco must be located at the same path on all the nodes.

To start the master and enter its Erlang shell, without redirecting the log to a file, run:

```
bin/disco nodaemon
```

To start the master as a daemon and log to a file, use:

```
bin/disco start
```

Hint: If Disco has started up properly, you should be able to see its processes running:

```
ps aux | grep beam.*disco
```

If you don't see any Disco processes, you may want to try *Troubleshooting Disco installation*.

2.2.4 Configure Authentication

Next we need to enable passwordless login via ssh to all servers in the Disco cluster. If you have only one machine, you need to enable passwordless login to `localhost` for the Disco user.

Run the following command as the Disco user, assuming that it doesn't have valid ssh-keys already:

```
ssh-keygen -N '' -f ~/.ssh/id_dsa
```

If you have one server (or shared home directories), say:

```
cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
```

Otherwise, repeat the following command for all the servers `nodeX` in the cluster:

```
ssh-copy-id nodeX
```

Now try to login to all servers in the cluster or `localhost`, if you have only one machine. You should not need to give a password nor answer to any questions after the first login attempt.

As the last step, if you run Disco on many machines, you need to make sure that all servers in the Disco cluster use the same Erlang cookie, which is used for authentication between Erlang nodes. Run the following command as the Disco user on the master server:

```
scp ~/.erlang.cookie nodeX:
```

Repeat the command for all the servers `nodeX`.

Warning: The Erlang cookie must be readable only to the disco user. If it isn't, run `chmod 400 ~/.erlang.cookie` on all the nodes.

2.2.5 Add nodes to Disco

At this point you should have Disco up and running. The final step, before testing the system, is to specify which servers are available for Disco. This is done via Disco's web interface.

Point your browser at `http://<DISCO_MASTER_HOST>:<DISCO_PORT>`, where `DISCO_MASTER_HOST` and `DISCO_PORT` should be replaced with their actual values. Normally you can use http://localhost:8989, if you run Disco locally or through an SSH tunnel.

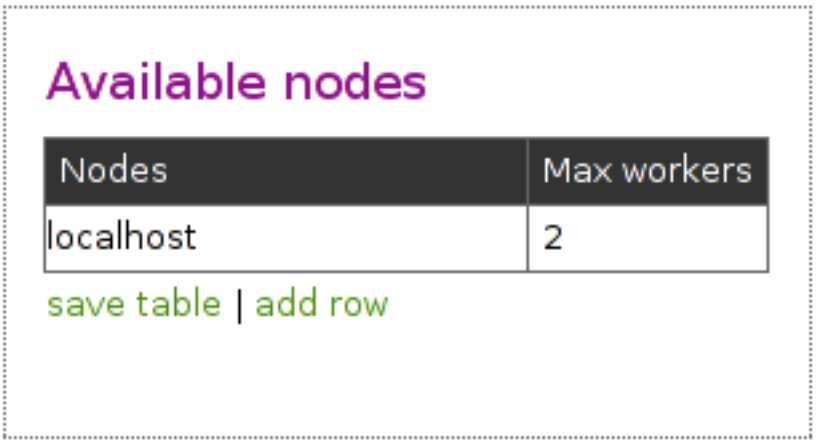
You should see the Disco main screen (see *a screenshot*). Click `configure` on the right side of the page. On the configuration page, click `add row` to add a new set of available nodes. Click the cells on the new empty row, and add hostname of an available server (or a *range of hostnames*) in the left cell, and the number of available cores (CPUs) on that server in the right cell. Once you have entered a value, click the cell again to save it.

Warning: Keep in mind that for more than one node, hostnames need to resolve globally (e.g. you should be relying on DNS to resolve hostnames, **not** */etc/hosts* on an individual machine).
Hostnames used by Disco are shortnames, and not fully qualified hostnames. DNS must be configured to correctly resolve the shortnames of the hostnames in the cluster.
Disco cannot currently use raw IP addresses for hostnames. Short DNS hostnames must be used to name cluster nodes.
A relatively common mistake is that `master` is just an alias for the loopback address, such as `localhost`, provided in */etc/hosts* on the master machine. In such cases, some nodes may **not** be able to resolve the master properly: they may all resolve to themselves (if they all have the same hosts file), nothing at all, or different machines (if they are configured differently).

You can add as many rows as needed to fully specify your cluster, which may have varying number of cores on different nodes. Click `save table` when you are done.

Add the `localhost`

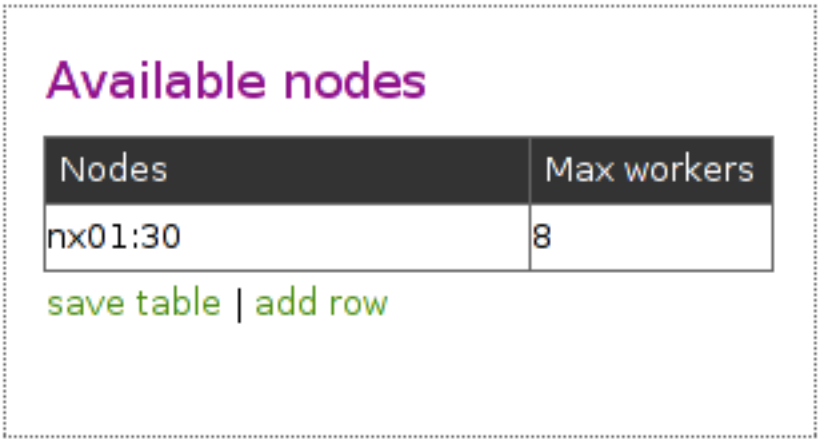
If you have only a single machine, the resulting table should look like this, assuming that you have two cores available for Disco:



Warning: It is not advised to use the *master* as a *slave* node in a serious Disco cluster.

Add multiple nodes in the same line

You can also specify multiple nodes on a single line, if the nodes are named with a common prefix, as here:



This table specifies that there are 30 nodes available in the cluster, from `nx01` to `nx30` and each node has 8 cores.

2.2.6 Test the System

Now Disco should be ready for use.

We can use the following simple Disco script that computes word frequencies in a text file to see that the system works correctly.

```
from disco.core import Job, result_iterator

def map(line, params):
    for word in line.split():
        yield word, 1

def reduce(iter, params):
    from disco.util import kvgroup
```

```
    for word, counts in kvgroup(sorted(iter)):
        yield word, sum(counts)

if __name__ == '__main__':
    job = Job().run(input=["http://discoproject.org/media/text/chekhov.txt"],
                    map=map,
                    reduce=reduce)
    for word, count in result_iterator(job.wait(show=True)):
        print(word, count)
```

Run the script as follows from `DISCO_HOME`:

```
python examples/util/count_words.py
```

Disco attempts to use the current hostname as `DISCO_MASTER_HOST`, if it is not defined in any settings file.

If you are runing Disco on multiple machines you must use the same version of Python for running Disco scripts as you use on the server side.

You can run the script on any machine that can access the master. The safest bet is to test the script on the master node itself.

If the machine where you run the script can access the master node but not other nodes in the cluster, you need to set `DISCO_PROXY`. The proxy address should be the same as the master's above. This makes Disco fetch results through the master node, instead of connecting to the nodes directly.

If the script produces some results, congratulations, you have a working Disco setup! If you are new to Disco, you might want to read the *Tutorial* next.

If the script fails, see the section about *Troubleshooting Disco installation*.

2.3 Installing Disco System-Wide

Note: Shortcut for Debian / Ubuntu users:

If you run Debian or some recent version of Ubuntu on the AMD64 architecture, you can use the `make-discoproject-debian` script in the source tree to build your own deb-packages. This will ensure that the packages are compatible with the software (particularly Python) versions in your cluster.

Alternatively, you may try out our *experimental* deb-packages which are available at the *Disco download page*.

If the packages installed properly, you should *Configure Authentication*.

2.3.1 Install From Source

Assuming you have already gotten Disco running out of the source directory, as described in *Install Disco*, to install system-wide, just run `make install` as root:

```
make install
```

This will build and install the Disco master to your system (see the `Makefile` for exact directory locations). You can specify `DESTDIR` and `prefix`, in compliance with GNU make.

On systems that are intended to function as Disco worker nodes only, you can use the `make install-node` target instead.

2.3.2 System Settings

`make install` installs a configuration file to `/etc/disco/settings.py` that is tuned for clusters, not a single machine.

By default, the settings assume that you have at least three nodes in your cluster, so DDFS can use three-way replication. If you have fewer nodes, you need to lower the number of replicas in `/etc/disco/settings.py`:

```
DDFS_TAG_MIN_REPLICAS=1
DDFS_TAG_REPLICAS=1
DDFS_BLOB_REPLICAS=1
```

Most likely you do not need to modify anything else in this file right now, but you can change the settings here, if the defaults are not suitable for your system.

See `disco.settings` for more information.

2.3.3 Creating a *disco* user

You can use any account for running Disco, however it may be convenient to create a separate *disco* user. Among other advantages, this allows setting resource utilization limits for the *disco* user (through `limits.conf` or similar mechanism).

Since Disco places no special requirements on the user, (except access to certain ports and the ability to execute and read its files), simply follow the guidelines of your system when it comes to creating new users.

2.3.4 Keeping Disco Running

You can easily integrate `disco` into your system's startup sequence. As an example, you can see how `disco-master.init` is implemented in Disco's `debian` packaging.

2.3.5 Configuring DDFS Storage

On the Disco nodes, DDFS creates by default a subdirectory named `vol0` under the `DDFS_DATA` directory to use for storage. If you have one or more dedicated disks or storage areas you wish to use instead, you can mount them under the directory specified by `DDFS_DATA` as subdirectories named `vol0`, `vol1` and so on.

2.4 Tutorial

This tutorial shows how to create and run a Disco job that counts words. To start with, you need nothing but a single text file. Let's call the file `bigfile.txt`. If you don't happen to have a suitable file on hand, you can download one from here.

2.4.1 1. Prepare input data

Disco can distribute computation only as well as data can be distributed. In general, we can push data to *Disco Distributed Filesystem*, which will take care of distributing and replicating it.

Note: Prior to Disco 0.3.2, this was done by splitting data manually, and then using `ddfs push` to push user-defined blobs. As of Disco 0.3.2, you can use `ddfs chunk` to automatically chunk and push size-limited chunks to DDFS. See *Pushing Chunked Data to DDFS*.

Lets chunk and push the data to a tag `data:bigtxt`:

```
ddfs chunk data:bigtxt ./bigfile.txt
```

We should have seen some output telling us that the chunk(s) have been created. We can also check where they are located:

```
ddfs blobs data:bigtxt
```

and make sure they contain what you think they do:

```
ddfs xcat data:bigtxt | less
```

Note: Chunks are stored in Disco's internal compressed format, thus we use `ddfs xcat` instead of `ddfs cat` to view them. `ddfs xcat` applies some `input_stream()` (by default, `chain_reader()`), whereas `ddfs cat` just dumps the raw bytes contained in the blobs.

If you used the file provided above, you should have only ended up with a single chunk. This is because the default chunk size is 64MB (compressed), and the `bigfile.txt` is only 12MB (uncompressed). You can try with a larger file to see that chunks are created as needed.

Hint: If you have unchunked data stored in DDFS that you would like to chunk, you can run a Disco job, to parallelize the chunking operation. Disco includes an example of how to do this, which should work unmodified for most use cases.

2.4.2 2. Write job functions

Next we need to write *map* and *reduce* functions to count words. Start your favorite text editor and create a file called `count_words.py`. First, let's write our map function:

```
def fun_map(line, params):
        for word in line.split():
                yield word, 1
```

Quite compact, eh? The map function takes two parameters, here they are called *line* and *params*. The first parameter contains an input entry, which is by default a line of text. An input entry can be anything though, since you can define a custom function that parses an input stream (see the parameter *map_reader* in the `Classic Worker`). The second parameter, *params*, can be any object that you specify, in case that you need some additional input for your functions.

For our example, we can happily process input line by line. The map function needs to return an iterator over of key-value pairs. Here we split a line into tokens using the builtin `string.split()`. Each token is output separately as a key, together with the value *1*.

Now, let's write the corresponding reduce function:

```
def fun_reduce(iter, params):
        from disco.util import kvgroup
        for word, counts in kvgroup(sorted(iter)):
                yield word, sum(counts)
```

The first parameter, *iter*, is an iterator over those keys and values produced by the map function, which belong to this reduce instance (see *partitioning*).

In this case, words are randomly assigned to different reduce instances. Again, this is something that can be changed (see `partition()` for more information). However, as long as all occurrences of the same word go to the same reduce, we can be sure that the final counts are correct.

The second parameter *params* is the same as in the map function.

We simply use `disco.util.kvgroup()` to pull out each word along with its counts, and sum the counts together, yielding the result. That's it. Now we have written map and reduce functions for counting words in parallel.

2.4.3 3. Run the job

Now the only thing missing is a command for running the job. There's a large number of parameters that you can use to specify your job, but only three of them are required for a simple job like ours.

In addition to starting the job, we want to print out the results as well. First, however, we have to wait until the job has finished. This is done with the `wait()` call, which returns results of the job once has it has finished. For convenience, the `wait()` method, as well as other methods related to a job, can be called through the `Job` object.

A function called `result_iterator()` takes a list of addresses to the result files, that is returned by `wait()`, and iterates through all key-value pairs in the results.

The following example from `examples/util/count_words.py` runs the job, and prints out the results:

```
from disco.core import Job, result_iterator

def map(line, params):
    for word in line.split():
        yield word, 1

def reduce(iter, params):
    from disco.util import kvgroup
    for word, counts in kvgroup(sorted(iter)):
        yield word, sum(counts)

if __name__ == '__main__':
    job = Job().run(input=["http://discoproject.org/media/text/chekhov.txt"],
                    map=map,
                    reduce=reduce)
    for word, count in result_iterator(job.wait(show=True)):
        print(word, count)
```

Note: This example could also be written by extending `disco.job.Job`. See, for example, *examples/util/wordcount.py*.

Now comes the moment of truth.

Run the script as follows:

```
python count_words.py
```

If everything goes well, you will see that the job executes. The inputs are read from the tag `data:bigtxt`, which was created earlier. Finally the output is printed. While the job is running, you can point your web browser at `http://localhost:8989` (or some other port where you run the Disco master) which lets you follow the progress of your job in real-time.

You can also set `DISCO_EVENTS` to see job events from your console:

```
DISCO_EVENTS=1 python count_words.py
```

In this case, the events were anyway printed to the console, since we specified `show=True`.

2.4.4 What next?

As you saw, creating a new Disco job is pretty straightforward. You could extend this simple example in any number of ways. For instance, by using the params object to include a list of stop words.

You could continue on with *Extended Tutorial* which is intended as a follow-on tutorial to this one.

If you pushed the data to *Disco Distributed Filesystem*, you could try changing the input to `tag://data:bigtxt`, and add `map_reader = disco.worker.classic.func.chain_reader`.

You could follow the *DiscoDB Tutorial*, to learn more about using `discodb` with Disco.

You could try using `sum_combiner()`, to make the job more efficient.

You can also experiment with custom partitioning and reader functions. They are written in the same way as map and reduce functions. Just see some examples in the `disco.worker.classic.func` module. After that, you could try *chaining jobs together*, so that output of the previous job becomes input for the next one.

The best way to learn is to pick a problem or algorithm that you know well, and implement it with Disco. After all, Disco was designed to be as simple as possible so you can concentrate on your own problems, not on the framework.

2.5 Extended Tutorial

This tutorial expands on the introductory *Tutorial* to expose the user to Disco's exported classes while solidifying the concepts of feeding input into and capturing output from Disco jobs. As a working example, this tutorial walks the user through implementing one approach for performing an inner_join operation on arbitrarily large datasets.

As a prerequisite, the reader is expected to have successfully completed the introductory *Tutorial* on a functional (happily configured and working) installation of Disco.

2.5.1 1. Background and sample input

Let's first prepare a sample input data set that's small enough and simple enough for us to follow and know what to expect on output. We will prepare two sets of input in csv format to be "joined" together using the first entry in each row as the key to match (join) on. Create a file named `set_A.csv` containing the following text:

```
1,"alpha"
2,"beta"
3,"gamma"
4,"delta"
5,"epsilon"
```

Create a second file named `set_B.csv` containing the following text:

```
1,"who"
2,"what"
3,"where"
4,"when"
5,"why"
6,"how"
```

When we inner_join these two datasets using the first entry in each row as its key, we would like to see output that looks something like this:

```
1,"alpha","who"
2,"beta","what"
3,"gamma","where"
4,"delta","when"
5,"epsilon","why"
```

Note that there is no line in the output for key=6 as seen in the input data of `set_B.csv` because it did not have a matched pair for that key in `set_A.csv`. Please also note that we would expect the output to be the same even if the order of the lines were scrambled in either of the two input data sets.

Note: If you're a big data fanatic and can't wait to get to a macho volume of input, *be patient*. Let's make sure we get everything working right and we understand what's happening with small data first before turning up the volume.

You should now have two files in your working directory named `set_A.csv` and `set_B.csv` which contain 5 and 6 lines, respectively, of text data.

2.5.2 2. Split input data into chunks

In the introductory *Tutorial*, we made use of a DDFS (*Disco Distributed Filesystem*) command, `ddfs chunk`, to split input data into chunks and copy it onto DDFS. To provide a more concrete sense of how to chunk input data, let's instead split our input data *before* we push it to DDFS. When we do push our already-split data to DDFS, we will tell DDFS to treat the distinct chunks as one.

As alluded to before, there are many strategies for performing efficient join operations inside MapReduce frameworks. Here we will take the approach of combining our two input data sets (A and B) into a single input stream. With a single input stream, it's easier to see how to split up the input, do work on it, then merge it back together. This approach doesn't necessarily harm performance but there are different strategies tuned for optimal performance depending upon the nature of your data. (Search the net for "`mapreduce join`" to see the wealth of competing strategies out there.)

Assuming a unix-like environment from here on, start by combining our two input files:

```
% cat set_A.csv set_B.csv > both_sets.csv
```

Next, we want to split our `both_sets.csv` file into chunks with 2 lines each. You can do this with a text editor yourself, by hand, or we can make use of the convenient unix utility `split` to do the job for us:

```
% split -l 2 both_sets.csv
```

Running `split` as above should create 6 files named `xaa` through `xaf`. You can quickly verify this by performing a count of the lines in each file and seeing that it adds up to 11:

```
% wc -l xa?
  2 xaa
  2 xab
  2 xac
  2 xad
  2 xae
  1 xaf
 11 total
```

Now that we've split the input data ourselves into 6 chunks, let's push our split data into DDFS and label it all with a single tag, `data:both_sets`, so that we can refer to all our chunks as one:

```
% ddfs push data:both_sets ./xa?
```

You can verify that all 11 lines made it into DDFS and are accessible via that single tag by asking to `cat` it back to the screen:

```
% ddfs cat data:both_sets
```

By splitting our input data into 6 chunks, we are now set up to perform 6 executions of our *map* function (which we have yet to implement). If you have a processor with 6 cores, you could conceivably perform all 6 map operations in parallel at the same time. If you have more than 6 cores either on one processor or across multiple processors available to Disco, you'll only be able to make use of, at most, 6 of them at one time during the map phase of a MapReduce job. In general: If you want more map operations to be running at the same time, make more chunks (smaller chunks). Taking it too far, if you make more chunks than you have cores, you won't get further speedup from parallelism.

You should now have the 11 lines of input csv-format data stored in DDFS in 6 chunks under the tag `data:both_sets`. While not necessarily the best approach for splitting and importing your largest datasets into DDFS, it may prove helpful to remember that you can chunk your data all at once *or* bring it in in pieces.

2.5.3 3. Write a job using a derived class

In the introductory *Tutorial*, we defined a *map* function and a *reduce* function, and then supplied them as parameters to `Job().run()`. But there's more fun to be had by deriving a new class from `Job`. Let's start by declaring our new class and saving it in a source file named `simple_innerjoin.py`:

```
class CsvInnerJoiner(Job):
    def map(self, row, params):
        # TODO
        pass

    def reduce(self, rows_iter, out, params):
        # TODO
        pass
```

Before we turn attention to implementing either of the *map* or *reduce* methods, we should consider our need, in this example, to read input that's in csv format. A convenient solution is to implement `map_reader()` in our class:

```
@staticmethod
def map_reader(fd, size, url, params):
    reader = csv.reader(fd, delimiter=',')
    for row in reader:
        yield row
```

This will allow us to implement `map()` to operate on one row's worth of input data at a time without needing to worry about raw input format.

Our strategy with our *map* and *reduce* methods will be to first sort all of the input data by their unique keys (which will put row 4 from `set_A.csv` right next to / in front of row 4 from `set_B.csv`), then merge consecutive rows having the same unique key. This puts most of the burden on our `reduce()` implementation, but we'll ease that a bit in a later pass. Since `map()` does not need to do much other than serve as a pass-through (quickly), modify our placeholder for `map()` to read:

```
def map(self, row, params):
    yield row[0], row[1:]
```

This will separate the unique key (in position 0) from all the other data on a row (assuming we want to re-use this for something more interesting than our fairly trivial input data set so far).

Now we ask `reduce()` to do the real work in its updated definition:

```
def reduce(self, rows_iter, out, params):
    from disco.util import kvgroup
    from itertools import chain
    for url_key, descriptors in kvgroup(sorted(rows_iter)):
        merged_descriptors = list(chain.from_iterable(descriptors))
        if len(merged_descriptors) > 1:
            out.add(url_key, merged_descriptors)
```

Again, as in *Tutorial*, we are using `disco.util.kvgroup()` to group together consecutive rows in our sorted input and hand them back as a group (iterable). Note our test to see if we have a matched pair or not is somewhat fragile and may not work for more general cases – we highlight this as an area for improvement for the reader to consider later.

Let's round out our `simple_innerjoin.py` tool by making it easy to supply names for input and output, while also making our output come out in csv format – adding to the bottom of `simple_innerjoin.py`:

```
if __name__ == '__main__':
    input_filename = "input.csv"
    output_filename = "output.csv"
    if len(sys.argv) > 1:
        input_filename = sys.argv[1]
        if len(sys.argv) > 2:
            output_filename = sys.argv[2]

    from simple_innerjoin import CsvInnerJoiner
    job = CsvInnerJoiner().run(input=[input_filename])

    with open(output_filename, 'w') as fp:
        writer = csv.writer(fp)
        for url_key, descriptors in result_iterator(job.wait(show=True)):
            writer.writerow([url_key] + descriptors)
```

Note: Notice the important nuance in our importing the `CsvInnerJoiner` class from our own source file. Ordinarily, if this script were run independently, we would not expect to need to import a class that's being defined in the same source file. Because Disco pickle's this source file (using its own `dPickle`) for the sake of distributing it to worker nodes, upon unpickling the definition of `CsvInnerJoiner` will no longer be visible in the local context. Try running with the "from ..." line commented out to see the resulting complaint from the Unpickler run by the workers. If anything, we should take this as a gentle reminder to be cognizant that we are preparing code to run in a distributed, parallel system and that we occasionally need to make some small adjustments for that environment.

In the prior *Tutorial*, all output flowed to the screen (stdout) but here we capture the output flowing from our job into a file in csv format. We chose to use the csv format throughout this *Extended Tutorial* for convenience but clearly other methods of redirecting output and formatting it to your own needs are possible in the same way.

2.5.4 4. Results and exploring partitions

We should now be set up to run our job with 6 input chunks corresponding to 6 invocations of our `map()` method and the output of those map runs will flow into 1 invocation of our `reduce()` method to then produce our final csv result file. Launching from the command-line:

```
% python simple_innerjoin.py data:both_sets output.csv
```

At this point, please check that the output found in the file `output.csv` matches what was expected. (Pedants can play further with formatting and quotation rules via the csv module, to taste.) If you instead encounter errors, please

double-check that your file faithfully matches the code outlined thus far and please double-check that you can still run the example from the introductory *Tutorial*.

Thus far we've been running parallel invocations of `map()` but not of `reduce()` – let's change that by requesting that the output from the map phase be divided into 2 partitions. Add the following line to the very top of our definition of the `CsvInnerJoiner` class, to look something like this:

```
class CsvInnerJoiner(Job):
    partitions = 2

    ...*truncated*...
```

Run the job again from the command-line and this time you may find that while the output might be correct, the output is no longer in sort-order. This is because we did not sort over all rows – only the rows handed to a particular invocation of `reduce()` were sorted, though we still get to see the output from parallel invocations of `reduce()` concatenated together in our single output csv file.

This helps highlight a problem we're going to have once we start throwing larger volumes of data at this Disco job: invoking `sorted()` requires a potentially large amount of memory. Thankfully Disco provides, as part of its framework, an easier solution to this common need for working with sorted results in the reduce step. At the top of our definition of the `CsvInnerJoiner` class, let's add the following line:

```
class CsvInnerJoiner(Job):
    partitions = 2
    sort = True

    ...*truncated*...
```

Simultaneously, we can remove the use of `sorted()` from the one line in our implementation of `reduce()` so that it now reads as:

```
def reduce(self, rows_iter, out, params):
    from disco.util import kvgroup
    from itertools import chain
    for url_key, descriptors in kvgroup(rows_iter):
        merged_descriptors = list(chain.from_iterable(descriptors))
        if len(merged_descriptors) > 1:
            out.add(url_key, merged_descriptors)
```

Now the work of sorting the results flowing from the mappers is done for us by the framework and that sort is performed across all mappers' results before being partitioned and handed as input to the reducers.

2.5.5 5. Big(ger) Data

Let's quickly generate a bigger input data set with which to work. The following one-liner can be modified to generate as little or as much sample data as you have patience / disk space to hold – modify the `1000000` near the end of the line to create as many rows of data as you like:

```
% python -c "import csv, sys, random; w = csv.writer(sys.stdout);
[w.writerow([i, int(999999*random.random())]) for i in range(1000000)]" > input1.csv
```

Run it twice (saving the first run's output in a different name from the second run's) to give yourself two sets of input data just as before. Then follow the steps from either this *Extended Tutorial* or the prior introductory *Tutorial* to chunk the input data and push it to DDFS in whatever manner you like. (Let's assume you tag your chunked input data as `data:bigger_sets` in DDFS.)

The only modification to `simple_innerjoin.py` that we suggest, depending upon how large your newly generated input data set is, is to increase the number of partitions to ratchet up the number of parallel runs of `reduce()`. Then go ahead and run your job in the same way:

```
% python simple_innerjoin.py data:bigger_sets bigger_output.csv
```

By monitoring the processes on the system(s) where you've configured Disco, you will hopefully be able to observe individual workers performing their map tasks and reduce tasks, the framework doing your sorting work for you in between, and how much cpu processing time is being used versus time spent waiting on disk or other resources. Having a larger dataset with a longer runtime makes observing these things much easier.

Note that you may quickly find your disk access speed to become a bottleneck and for this reason and others you should consider playing with the number of partitions as well as the number of input chunks (how many reducers and mappers, respectively) to find your system's optimal throughput for this job.

As a variation on the above, remember that our `simple_innerjoin.py` script has the capability to read its input data from a local file instead of DDFS – try running again with a local file supplied as the location of the input (instead of `data:bigger_sets`). Did you get an error message with "Invalid tag (403)"? If so, you need to ensure Disco recognizes that you are supplying a filename and not the name of a tag. Did you get an error message with "IOError: [Errno 2] No such file or directory"? If so, you either need to supply the full path to the file (not a relative path name) or that path may not be available to Disco everywhere (if so, a good reason to use DDFS again). Was your run faster or slower than using DDFS?

After playing with ever larger volumes of data and tweaking the controls that Disco provides, you'll quickly gain confidence in being able to throw any size job at Disco and knowing how to go about implementing a solution.

2.5.6 simple_innerjoin.py listing

Complete source all in one place:

```
from disco.core import Job, result_iterator
import csv, sys


class CsvInnerJoiner(Job):
    partitions = 2
    sort = True

    def map(self, row, params):
        yield row[0], row[1:]

    @staticmethod
    def map_reader(fd, size, url, params):
        reader = csv.reader(fd, delimiter=',')
        for row in reader:
            yield row

    def reduce(self, rows_iter, out, params):
        from disco.util import kvgroup
        from itertools import chain
        #for url_key, descriptors in kvgroup(sorted(rows_iter)):
        for url_key, descriptors in kvgroup(rows_iter):
            merged_descriptors = list(chain.from_iterable(descriptors))
            if len(merged_descriptors) > 1:
                out.add(url_key, merged_descriptors)


if __name__ == '__main__':
    input_filename = "input.csv"
    output_filename = "output.csv"
    if len(sys.argv) > 1:
```

```
    input_filename = sys.argv[1]
    if len(sys.argv) > 2:
        output_filename = sys.argv[2]

from simple_innerjoin import CsvInnerJoiner
job = CsvInnerJoiner().run(input=[input_filename])

with open(output_filename, 'w') as fp:
    writer = csv.writer(fp)
    for url_key, descriptors in result_iterator(job.wait(show=True)):
        writer.writerow([url_key] + descriptors)
```

2.5.7 What next?

A natural next step in experimenting with partitioning involves *chaining jobs together* since the number of partitioned outputs from one job becomes the number of chunked inputs for the next. As a baby step, you could move the `reduce()` method implemented above into a second, chained job and replace it in the first job with a do-nothing substitute like `disco.worker.classic.func.nop_reduce()`.

As already mentioned in the introductory *Tutorial*, the best way to learn is to pick a problem or algorithm that you know well, and implement it with Disco. After all, Disco was designed to be as simple as possible so you can concentrate on your own problems, not on the framework.

2.6 Troubleshooting Disco installation

Setting up Disco should tell you enough to get Disco up and running, but it may happen that Disco doesn't work properly right after installation. If you can't *run the word count example* successfully, the reason is usually a small misconfigured detail somewhere. This document tries to help you figure out what's going wrong.

Since Disco is a distributed system based on loosely coupled components, it is possible to debug the system by testing the components one by one. This document describes the troubleshooting process. It is intended to help you to get Disco working locally, on a single computer. After you have done this, distributing it should be rather straightforward: the same debugging techniques apply.

Note: It's assumed that you have already followed the steps in *Install Disco*.

First, ensure the following:

- The version of Erlang is the same throughout the cluster.
- The version of Disco is the same throughout the cluster, and installed in the same location.
- The 'python' executable or symbolic link points to the same version of Python across the cluster, and on the clients from which Disco jobs are submitted.

2.6.1 Make sure Disco is not running

If you have started Disco earlier, try to stop the *master* using `disco stop` (or `C-c` if you are running with `disco nodaemon`). If you cannot seem to stop Disco this way, kill the `beam` processes by hand,

---

Hint: You can use:

```
ps aux | grep beam.*disco
```

and:

```
kill PID
```

to hunt down and kill the *pids*, respectively.

2.6.2 Is the master starting?

Start Disco by saying:

```
disco nodaemon
```

If everything goes well, you should see a bunch of `=INFO REPORT=` messages printed to the screen. If you see any `=ERROR REPORT=` messages, something is wrong, and you should try to resolve the particular issue *Erlang* is reporting. These messages often reveal what went wrong during the startup.

If you see something like this:

```
application: disco
exited: {bad_return,{{disco_main,start,[normal,[]]},
        {'EXIT',["Specify ",scgi_port]}}}
```

Disco is trying to start up properly, but your Erlang installation probably doesn't work correctly and you should try to re-install it.

Note: If you started disco using `disco start`, you will have to check the logs in `DISCO_LOG_DIR` for such messages.

If you can't find the log file, the master didn't start at all. See if you can find master binaries in the `ebin` directory under `DISCO_MASTER_HOME`. If there are no files there, check for compilation errors when you *Install Disco*.

Hint: If you don't know what `DISCO_LOG_DIR` is (or any other `setting`), you can check with:

```
disco -v
```

If the master is running, you can proceed to the next step (you can double check with `ps` as in *Make sure Disco is not running*). If not, the master didn't start up properly.

2.6.3 Are there any nodes on the status page?

Now that we know that the master process is running, we should be able to configure the system. Open your web browser and go to http://localhost:8989/ (or whatever your `DISCO_MASTER_HOST` and `DISCO_PORT` are set to). The Disco status page should open.

Do you see any boxes with black title bars on the status page (like in this screenshot)? If not, add nodes to the system as instructed in *Add nodes to Disco*.

If adding nodes through the web interface fails, you can try editing the config file manually. For instance, if you replace `DISCO_ROOT` in the following command, it will create a configuration file with one node:

```
echo '[["localhost", "1"]]' > DISCO_ROOT/disco_4441.config
```

Hint: Remember to restart the master after editing the config file by hand.

Note: Note that as of version 0.3.1 of Disco, jobs can be submitted to Disco even if there are no nodes configured. Disco assumes that this configuration is a temporary state, and some nodes will be added. In the meantime, Disco retains the jobs, and will start or resume them once nodes are added to the configuration and become available.

Now is a good time to try to run a Disco *job*. Go ahead and retry the *installation test*. You should see the job appear on the Disco status page. If the job succeeds, it should appear with a green box on the job list. If it turns up red, we need to continue debugging.

2.6.4 Are slaves running?

In addition to the master process on the master node, *Erlang* runs a *slave* on each node in a Disco cluster.

Make sure that the slave is running:

```
ps aux | grep -o disco.*slave@
```

If is is running, you should see something like this:

```
disco_8989_master@discodev -sname disco_8989_slave@
disco.*slave@
```

If you get a similar output, go to Do workers run?. If not, read on.

Is SSH working?

The most common reason for the slave not starting up is a problem with *SSH*. Try the following command:

```
ssh localhost erl
```

If SSH asks for a password, or any other confirmation, you need to configure SSH properly as instructed in *authentication configuration*.

If SSH seems to work correctly, Erlang should be able to start a slave. Check that you get something similar when you do:

```
[user@somehost dir]$ disco debug
Erlang VERSION

Eshell VERSION (abort with ^G)
(testmaster@somehost)1> slave:start(localhost, "testnode").
{ok,testnode@localhost}
(testmaster@somehost)1> net_adm:ping(testnode@localhost).
pong
```

If Erlang doesn't return `{ok,_Node}` for the first expression, or if it returns `pang` for the second expression, there's probably something wrong either with your *authentication configuration*.

Note: Node names need to be consistent. If your master node is called `huey` and your remote node `dewey`, `dewey` must be able to connect to the master node named `huey`, and vice versa. Aliasing is not allowed.

2.6.5 Is your firewall configured correctly?

Disco requires a number of ports to be accessible to function properly.

- 22 - SSH
- 8990 - DDFS web API
- 8989 - Disco web interface/API. Must be unblocked on slaves and the master.
- 4369 - Erlang port mapper
- 30000 to 65535 - Communication between Erlang slaves

Note: Future versions of Disco may allow you to specify a port range for Erlang to use. However, the current version of Disco does not, so you must open up the entire port range.

2.6.6 Is your DNS configured correctly?

Disco uses short DNS names of cluster nodes in its configuration. Please ensure that short hostnames were entered in the *Add nodes to Disco* step, and that DNS resolves these short names correctly across all nodes in the cluster.

2.6.7 Do workers run?

The *master* is responsible for starting individual processes that execute the actual *map* and *reduce tasks*. Assuming that the master is running correctly, the problem might be in the *worker*.

See what happens with the following command:

```
ssh localhost "python DISCO_HOME/lib/disco/worker/classic/worker.py"
```

Where `DISCO_HOME` in this case must be the Disco source directory. It should start and send a message like this:

```
WORKER 32 {"version": "1.0", "pid": 13492}
```

If you get something else, you may have a problem with your `PATH` or Python installation.

2.6.8 Still no success?

If the problem persists, or you can't get one of the steps above working, do not despair! Report your problem to friendly Disco developers *on IRC or the mailing list*. Please mention in your report the steps you followed and the results you got.

2.7 Get Involved

2.7.1 Develop Disco

Get Disco from github. You can easily fork a repository of your own (just click "fork"). You can *set up a development environment* on a single machine or a cluster.

2.7.2 Mailing list / discussion groups

We have a Google group for Disco which serves as our mailing list. Subscribe here.

2.7.3 IRC

Join Disco discussions at our IRC channel *#discoproject* at Freenode. This is usually the fastest way to get answers to your questions and get in touch with Disco developers.

If you haven't used IRC before, see here how to get started.

2.7.4 Roadmap / Wiki

We are using the issues list as a roadmap for Disco. You can report bugs and wishlist features for Disco there. Also see the wiki for other miscellaneous information.

CHAPTER 3

Disco In Depth

3.1 Administering Disco

3.1.1 Monitoring the cluster

An overall view of the state of the cluster is provided on the status page, which is the default page for the web user interface, and is accessible by clicking `status` on the top right of any page. This shows a node status box for each node in the cluster.

A black title background in the top row of the status box indicates that the node is connected to the master, whereas a node that the master is unable to connect to will have a crimson title background. A blacklisted node will have a lavender background for the entire status box. Note that the master still attempts to connect to blacklisted nodes, which means that the blacklist status and the connection status for a node are independent.

The second row in the status box shows a box for each of the configured cores on the node, and shows the busy cores in yellow. The third row shows amount of available disk space in the DDFS filesystem on the node.

The fourth row shows some job statistics for the node. The leftmost number shows the number of tasks that successfully completed on the node. The middle number indicates the number of tasks that were restarted due to them experiencing recoverable failures. The rightmost number indicates the number of tasks that crashed due to non-recoverable errors. Since a crashing task also causes its job to fail, this number also indicates the number of jobs that failed due to tasks crashing on this node.

The job listing along the top right side of the page shows running jobs in yellow, completed jobs in green, and crashed or killed jobs in pink. Hovering over a running job highlights the cores that the tasks of the job are using on each node in light blue.

Below the job listing on the right is some information from the last DDFS garbage collection run. A table mentions the number of tags and blobs that were kept after the last GC run and their total sizes in bytes, along with similar information for deleted blobs and tags.

3.1.2 Using a proxy

A cluster is sometimes configured with a few head nodes visible to its users, but most of the worker nodes hidden behind a firewall. Disco can be configured to work in this environment using a HTTP proxy, provided the node running the Disco master is configured as one of the head nodes, and the port used by the proxy is accessible to the Disco users.

To use this mode, the Disco settings on the master (in the `/etc/disco/settings.py` file) should set `DISCO_PROXY_ENABLED` to `"True"`. Disco then starts a HTTP proxy specified by `DISCO_HTTPD` (defaulting to *lighttpd*) on port `DISCO_PROXY_PORT` (defaulting to `8999`) with a configuration that proxies HTTP access to the Disco worker nodes. Currently, Disco supports generating configurations for the *lighttpd* and *varnish* proxies. The Disco master needs to be restarted for any changes in these settings to take effect.

To use this proxy from the client side, i.e. in order to use the `disco` and `ddfs` commands, the users need to set `DISCO_PROXY` in their environment, or in their local `/etc/disco/settings.py` file. The value of this should be in the format `http://<proxy-host>:<proxy-port>`, with `<proxy-host>` and `<proxy-port>` specifying how the proxy is accessible to the user.

3.1.3 Blacklisting a Disco node

You may decide that you need to reduce the load on a node if it is not performing well or for some other reason. In this case, you can blacklist a node, which informs Disco that the node should not be used for running any tasks or storing any new DDFS data. However, Disco will still use the node for reading any DDFS data already stored on that node. Note that blacklisting a node will not trigger re-replication of data away from the node (however, see below on blacklisting a DDFS node).

Blacklisting a node is done on the configuration page, accessible by clicking `configure` on the top right side of any page. Type the name of the node in the text entry box under `Blacklisted nodes for Disco`, and press enter. The node should now show up above the text entry box in the list of blacklisted nodes. Any node in this list can be clicked to whitelist it.

The set of blacklisted nodes is persisted, so that it is not lost across a restart.

3.1.4 Adjusting Disco system settings

Disco provides a tunable parameter `max_failure_rate` to administrators to control the number of recoverable task failures a job can experience before Disco fails the entire job. This parameter can be set on the configuration page.

3.1.5 Removing nodes from Disco

You probably went through the process of adding a node to the cluster when you configured the Disco cluster in *Add nodes to Disco*. Removing a node from the Disco configuration is a very similar process, except that you need to click `remove` next to the node(s) you wish to remove, and then click `save table` when you are done.

3.1.6 Blacklisting a DDFS node

There are various ways of removing a node that hosts DDFS data from the Disco cluster, with differing implications for safety and data availability.

- The node could be physically removed from the cluster, but left in the Disco configuration. In this case, it counts as a failed node, which reduces by one the number of additional node failures that could be safely tolerated. *Garbage collection and Re-replication* will attempt the replication of any missing live blobs and tags.
- The node could be physically removed from the cluster as well as the Disco configuration. In this case, Disco treats the missing node as an unknown node instead of as a failed node, and the number of additional node failures tolerated by DDFS does not change. However, this voids safety, since Disco might allow more nodes hosting DDFS data to fail than is safe. For example, if the replication factor is set to 3, Disco might treat two additional node failures as safe, whereas actually those two nodes might be hosting the last two remaining replicas of a blob, the third replica being lost when the first node was removed from the configuration. If this

happens, *Garbage collection and Re-replication* will not suffice to replicate the missing blobs and tags, and some data might be permanently lost.

The drawback of both of these approaches is that there is no indication provided by Disco as to when, if ever, DDFS is in a consistent state again with respect to the data that was hosted on the removed node.

DDFS now allows scheduling the removal of a node from DDFS, by putting the node on a DDFS *blacklist*, which is specified using the text entry box labeled `Blacklisted nodes for DDFS`. This makes *Garbage collection and Re-replication* actively replicate data away from that node; that is, additional replicas are created to replace the blobs hosted on a blacklisted node, and when safe, references to the blobs on that node are removed from any referring tags. DDFS data on the blacklisted node is however not deleted.

In addition, DDFS now provides an indication when all the data and metadata that was hosted on that node has been re-replicated on other cluster nodes, so that that node can be safely removed from the Disco cluster (both physically, as well as from the configuration) with a guarantee that no data has been lost. The indication is provided by the node entry being highlighted in green in the blacklist. It may require several runs of *Garbage collection and Re-replication* to re-replicate data away from a node; since by default it runs once a day, several days may be needed before a DDFS blacklisted node becomes safe for removal.

3.1.7 Handling a master failure

Disco is currently a single master system, which means it has a single point of failure. This master controls both job scheduling as well as the *Disco Distributed Filesystem*. The failure of the master will result in the termination of currently running jobs and loss of access to the *Disco Distributed Filesystem*. However, it will not result in any loss of data in DDFS, since all metadata in DDFS is replicated, just like data. The only centralized static information is the Disco settings file on the master (specified by the `DISCO_SETTINGS_FILE`, which defaults to `/etc/disco/settings.py` for installation), and the Disco cluster configuration, maintained in the file specified by the `DISCO_MASTER_CONFIG` setting. You can examine all the settings for Disco using the `disco -v` command.

A failed Disco master can be replaced by installing the Disco master on a new machine (or even an existing Disco node, though this is not recommended for large or busy clusters). See *Installing Disco System-Wide* for details on installing the Disco master. On the replacement machine, you will need to copy the settings and configuration files from the original master into their expected locations. For this reason, it is a good idea to backup these files after any change. The `DISCO_SETTINGS_FILE` is manually modified, while the `DISCO_MASTER_CONFIG` file is managed by the Disco master. The config file is changed whenever nodes are added or removed as members of the cluster, or any blacklist.

3.2 Pushing Chunked Data to DDFS

```
ddfs chunk data:bigtxt /path/to/bigfile.txt
```

Hint: If the local `/path/to/bigfile.txt` is in the current directory, you must use `./bigfile.txt`, or another path containing `/` chars, in order to specify it. Otherwise `ddfs` will think you are specifying a *tag*.

The creation of chunks is record-aware; i.e. chunks will be created on record boundaries, and 'ddfs chunk' will not split a single record across separate chunks. The default record parser breaks records on line boundaries; you can specify your own record parser using the `reader` argument to the `ddfs.chunk` function, or the `-R` argument to `ddfs chunk`.

The chunked data in DDFS is stored in Disco's internal format, which means that when you read chunked data in your job, you will need to use the `disco.func.chain_reader`. Hence, as is typical, if your map tasks are reading chunked data, specify `map_reader=disco.func.chain_reader` in your job.

3.3 Contributing to Disco

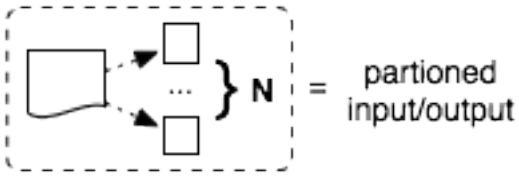
The easiest way to contribute to Disco is to use github. All of the core Disco developers use a fork of the official repository, from which their changes can be publicly tracked and pulled.

If you want to start contributing patches to disco, make sure you follow the *Disco coding conventions*.

3.4 Data Flow in Disco Jobs

Disco allows the chaining together of jobs containing *map* and/or *reduce* phases. *Map* and *Reduce* phases each have their own concepts of data flow. Through combinations of chained jobs, Disco supports a remarkable variety of data flows.

Understanding data flow in Disco requires understanding the core concept of *partitioning*. Map results in Disco can be either *partitioned* or *non-partitioned*. For partitioned output, the results Disco returns are index files, which contain the URLs of each individual output partition:



In the diagrams below, it should be clear when Disco is relying on either reading partitioned input or writing partitioned output.

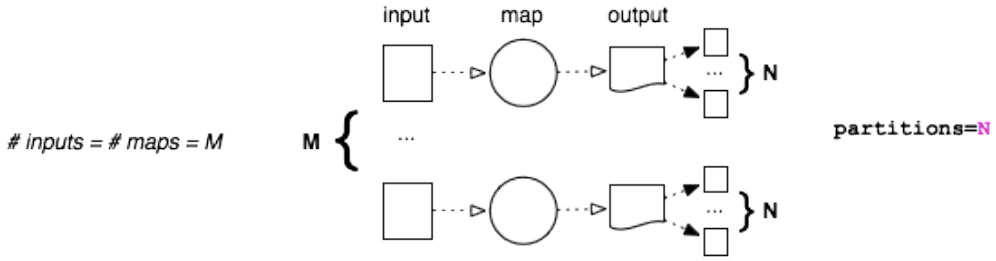
The overall data flow in a classic Disco job is controlled by four `disco.worker.classic.worker.Worker` parameters. The presence of `map` and `reduce`, determine the overall structure of the job (i.e. whether its *mapreduce*, *map-only*, or *reduce-only*). The `partitions` parameter determines whether or not the map output is partitioned, and the number of partitions. The `merge_partitions` parameter determines whether partitioned input to reduce is merged.

3.4.1 Map Flows

The two basic modes of operation for the map phase correspond directly to writing either partitioned or non-partitioned output.

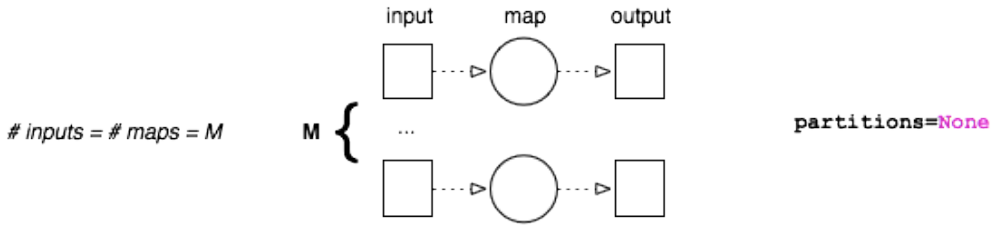
Partitioned Map

For partitioned map, each output is written to one of the partitions:



Non-Partitioned Map

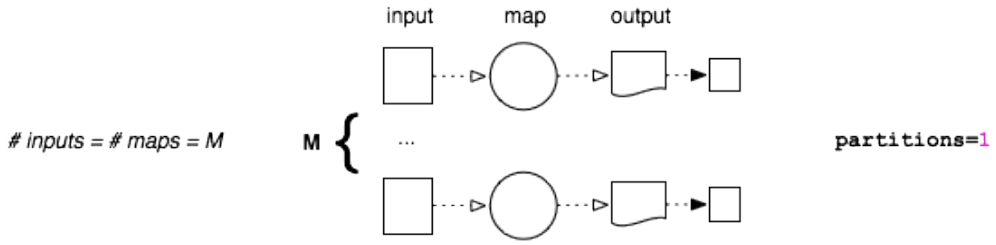
For non-partitioned map, *every* map task creates a single output. In other words, the input-output relationship is exactly *1 to 1*:



Single-Partition Map

Notice that for partitioned output with `N` partitions, **exactly** `N` files will be created *for each node*, *regardless of the number of maps*. If the map tasks run on `K` nodes, exactly `K * N` files will be created. Whereas for non-partitioned output with `M` inputs, exactly `M` output files will be created.

This is an important difference between partitioned output with 1 partition, and non-partitioned output:

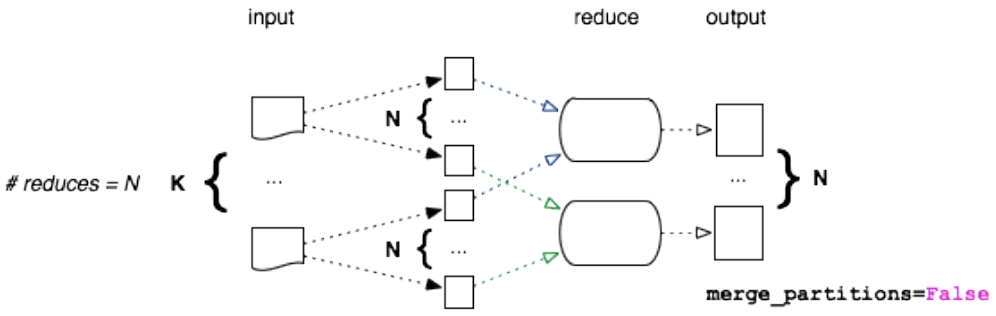


The default number of partitions for map is 1. This means that by default if you run `M` maps on `K` nodes, you end up with `K` files containing the results. In older versions of Disco, there were no partitions by default, so that jobs with a huge number of inputs produced a huge number of outputs. If `M >> K`, this is suboptimal for the reduce phase.

3.4.2 Reduce Flows

The basic modes of operation for the reduce phase correspond to partitioned/non-partitioned input (instead of output as in the map phase).

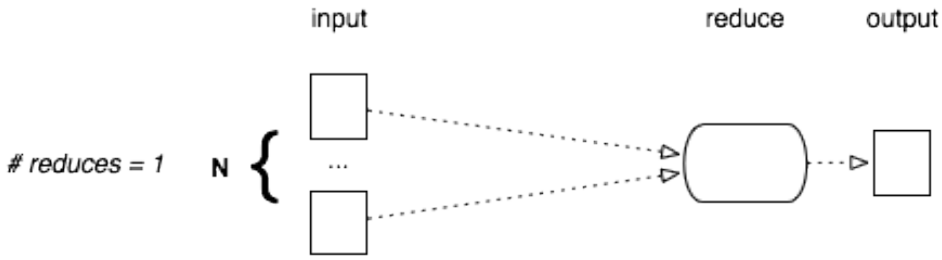
Normal Partitioned Reduce Flow



As you might expect, the default is to distribute the reduce phase.

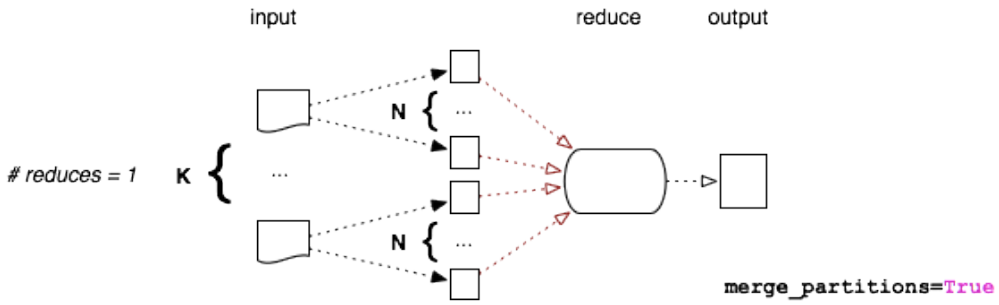
Non-Partitioned Reduce

For non-partitioned input, there can only ever be 1 reduce task:



The situation is slightly more complicated for partitioned input, as there is a choice to be made whether or not to merge the partitions, so that all results are handled by a single reduce:

Merge Partitioned Reduce



Or to use the normal, distributed reduce, in which there are `N` reduces for `N` partitions:

3.5 Disco Distributed Filesystem

Contents

3.5.1 Introduction

Disco Distributed Filesystem (DDFS) provides a distributed storage layer for Disco. DDFS is designed specifically to support use cases that are typical for Disco and *mapreduce* in general: Storage and processing of massive amounts of immutable data. This makes it very suitable for storing, for instance: log data, large binary objects (photos, videos, indices), or incrementally collected raw data such as web crawls.

In this sense, DDFS is complementary to traditional relational databases or distributed key-value stores, which often have difficulties in scaling to tera- or petabytes of bulk data. Although DDFS stands for Disco Distributed *filesystem*, it is not a general-purpose POSIX-compatible filesystem. Rather, it is a special purpose storage layer similar to the Google filesystem or related open-source projects such as Hadoop Distributed Filesystem (HDFS), MogileFS or Tabled.

DDFS is a low-level component in the Disco stack, taking care of data *distribution*, *replication*, *persistence*, *addressing* and *access*. It does not provide a sophisticated query facility in itself but it is **tightly integrated** with Disco *jobs*. Disco can store job results to DDFS, providing persistence for and easy access to processed data.

DDFS is a **tag-based** filesystem: Instead of having to organize data to directory hierarchies, you can tag sets of objects with arbitrary names and retrieve them later based on the given *tags*. For instance, tags can be used to timestamp different versions of data, or denote the source or owner of data. Tags can contain links to other tags, and data can be referred to by multiple tags; tags hence form a network or a directed **graph of metadata**. This provides a flexible way to **manage terabytes** of data assets. DDFS also provides a mechanism to store arbitrary attributes with the tags, for instance, to denote data type.

DDFS is **schema-free**, so you can use it to store arbitrary, non-normalized data. However, it is not suitable for storing data items that are very small (fewer than 4K) or that need to be updated often, such as user passwords or status indicators. You can store frequently changing data in a key-value store or a relational database. If you need to analyze this data with Disco, you can dump a snapshot of the full database to DDFS, for instance, to update your user models every night.

DDFS is **horizontally scalable**. New nodes can be added to the storage cluster on the fly, using the Disco web UI. All heavy IO on bulk data is distributed, so there are no bottlenecks limiting the amount of data that DDFS can handle. Only metadata is handled centrally, ensuring that it is kept **consistent** all the time.

DDFS is designed to operate on commodity hardware. **Fault-tolerance** and **high availability** are ensured by K-way replication of both data and metadata, so the system tolerates *K-1* simultaneous hardware failures without interruptions.

DDFS stores data and metadata on normal local filesystems, such as *ext3* or *xfs*, so even under a catastrophic failure data is recoverable using standard tools.

3.5.2 Concepts

DDFS operates on two concepts: *Blobs* and *Tags*.

Blobs

Blobs are arbitrary objects (files) that have been pushed to DDFS. They are distributed to storage nodes and stored on their local filesystems. Multiple copies or replicas are stored for each blob.

Tags

Tags contain metadata about blobs. Most importantly, a tag contains a list of URLs (one for each replica) that refer to blobs that have been assigned this tag. Tag may also contain links to other tags. It may also include user-defined metadata.

The next section describes the role of tags and blobs in more detail. It also shows how they relate to the five main tasks of DDFS, data *distribution*, *replication*, *persistence*, *addressing* and *access*.

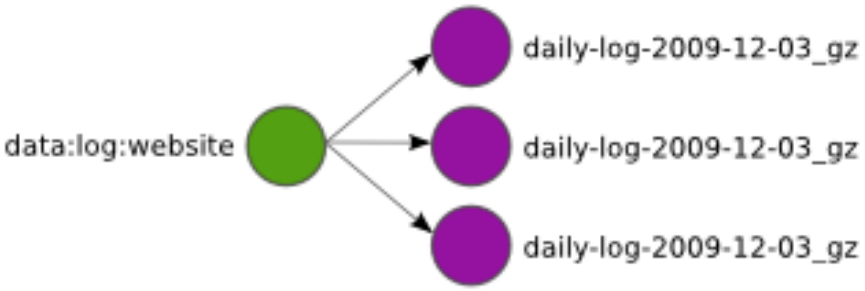
3.5.3 Overview

Consider that you have a log file containing data of a single day.

For DDFS, this is a blob. When you push the blob to DDFS using *DDFS APIs*, a DDFS client **distributes** the blob to K nodes.

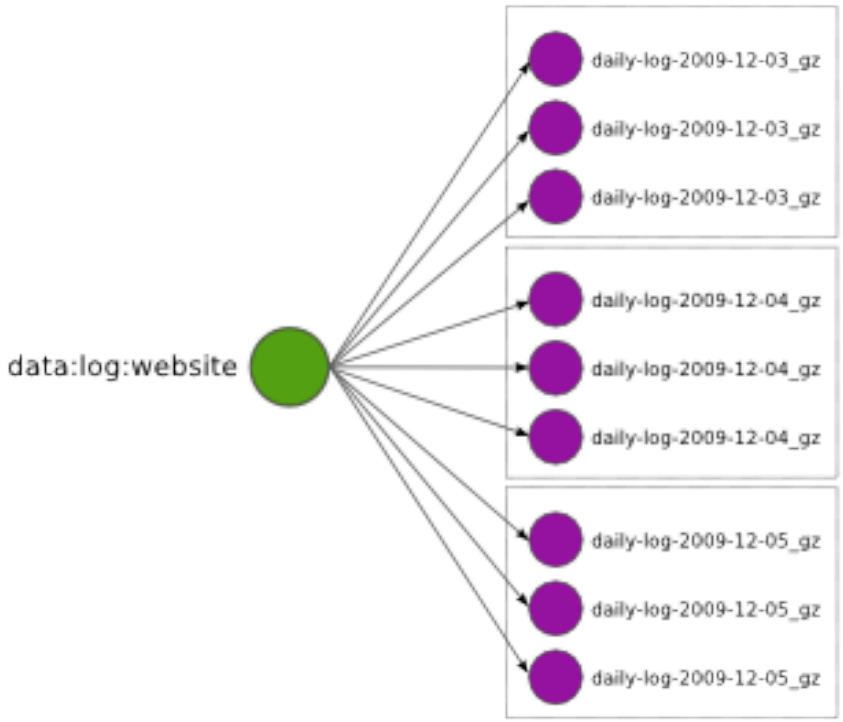
By default, K is 3, so you get three identical **replicas** of the blob. DDFS and Disco can utilize any of the replicas, in case some of them are unavailable due to disk or server failure. DDFS ensures that you will always have K replicas, even if disks fail, by re-replicating blobs if needed. This guarantees that your data is truly **persistent**.

Even persistent data is not very valuable if it cannot be accessed easily. The blobs distributed above are stored on three random nodes. To be able to use them efficiently, metadata storing **addresses** of the blobs is needed. DDFS uses tags for this purpose.

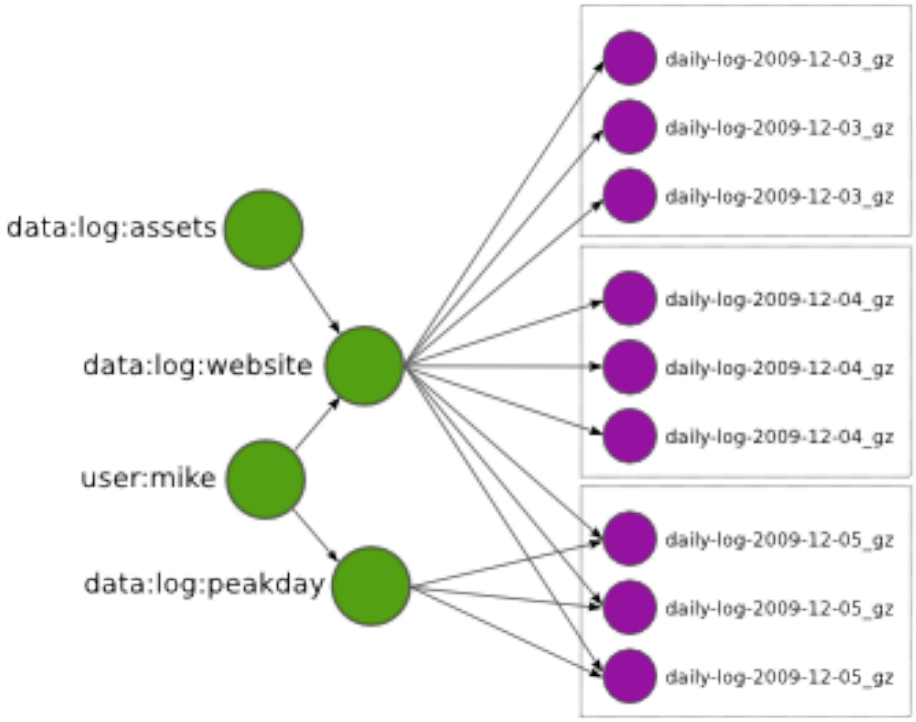


The green tag allows you to query data behind *data:log:website* using *DDFS APIs* and retrieve a tag object that contains URLs to the blobs. You can **access** the blobs using their URLs over HTTP as usual, or give the list to Disco to be used as inputs for a Map/Reduce job. Naturally metadata should not be lost under any circumstances, so tags are replicated and distributed to many nodes similarly to blobs.

Each blob *must* have at least one tag linking to it. Otherwise the blob is practically unaccessible or *orphaned*. Orphaned blobs are eventually deleted by the garbage collector. Correspondingly, if you want to delete a set of blobs from DDFS, you must delete all links (or tags) referencing the blobs which makes them orphaned and subject to eventual removal.



Eventually you want to add more daily logs (blobs) under the tag *data:log:website*. Each daily log is replicated separately, so the tag ends up containing many *replication sets*, that is, lists of URLs that pointing at replicas of a blob. Replications sets are represented by dotted boxes above.



DDFS allows tags to reference other tags. This is a very powerful feature which makes it possible to implement tag hierarchies and graphs. For instance, the tag *user:mike* above links to all tags owned by Mike. *DDFS APIs* provides functions to traverse the tag graph, so it is straightforward to retrieve all tags and blobs owned by Mike.

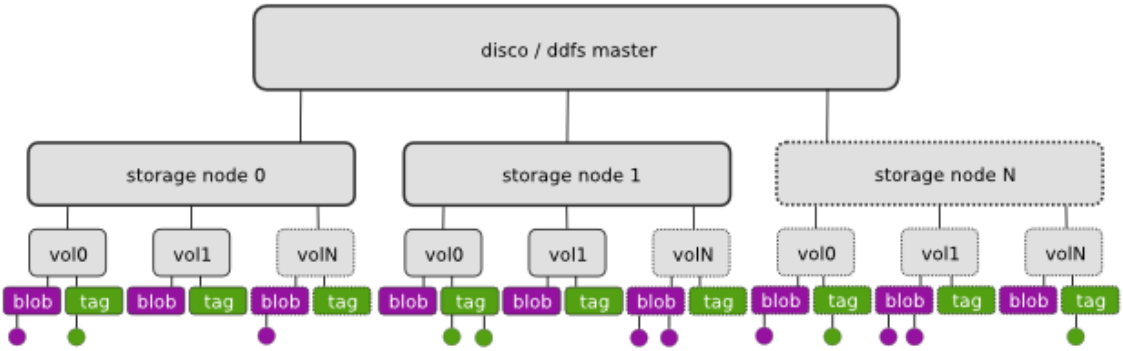
Tags may also reference overlapping sets of blobs, as in *data:log:peakday* above. This feature is useful if you want to provide many alternative views to the same data. DDFS is designed to scale to millions of tags, so you can use them without hesitation.

Tags also support a token-based authorization mechanism to control read and write access. If a write-token is specified for a tag, all operations that wish to modify the tag will need to provide this write-token. Without this token, any write operation will return an "unauthorized" error. Similarly, a read-token can be used to control accesses that read the tag. Read and write tokens can be independently specified.

When a token is specified for an operation that creates a new tag, that token becomes the new tag's read and write token. This allows the atomic creation of access-controlled tags.

In addition to being a container of metadata about blobs, a tag can also contain a limited number of user-defined attributes, each with a name and a string value.

Implementation



DDFS is embedded in Disco, hence the architecture diagram above closely resembles that of Disco (see *Technical Overview*). DDFS is currently coordinated by a single master node, similar to Disco itself. This choice was motivated by the ease of implementation and robustness, following experiences of the first version of the Google filesystem. As no data is stored on the master node, it is not a single point of failure with respect to data persistence. It mainly acts as a lock server, ensuring atomicity of metadata operations.

Each storage node contains a number of disks or volumes (*vol0..volN*), assigned to DDFS by mounting them under `DDFS_DATA/vol0` ... `DDFS_DATA/volN` (see `DDFS_DATA`). On each volume, DDFS creates two directories, `tag` and `blob`, for storing tags and blobs, respectively. DDFS monitors available disk space on each volume on regular intervals for load balancing. New blobs are stored to the least loaded volumes.

Each storage node maintains a cache of all tags stored on the node. When the master node receives a request accessing a yet unseen tag, it queries the storage nodes to find all replicas of the tag. Thanks to the cache, this operation is reasonably fast. Similarly, if the master node crashes and restarts, re-populating the master cache takes only some seconds.

All tag-related operations are handled by the master, to ensure their atomicity and consistency. The client may push new blobs to DDFS by first requesting a set of URLs for the desired number of replicas from the master. After receiving the URLs, the client can push the blobs individually to the designated URLs using HTTP PUT requests. After pushing all replicas successfully to storage nodes, the client can tag the blobs by making a tag request to the master with a list of URLs to the newly created blobs.

If the client fails to push all K replicas to storage nodes, it can request a new set of URLs from the master, excluding the failed nodes. This approach is enabled by default in the DDFS Python API. The client can also decide to accept only M replicas, where $M < K$, if this is sufficient for the application. If the master detects that a node has become unresponsive, it is automatically blacklisted and dropped from subsequent queries. Thanks to replicated data and metadata, this does not result in any data loss.

A regular garbage collection process makes sure that the required number of replicas is maintained, orphaned blobs are deleted and deleted tags are eventually removed from the system. The desired number of replicas is defined in the configuration file, see `disco.settings` for details.

Blobs can be accessed either over HTTP, using DDFS's built-in web server on each storage node, or directly on local disk. The latter feature is heavily utilized by Disco, which prefers to run tasks on the nodes where data is physically stored, to minimize network traffic.

The token-based authorization scheme is implemented using the basic access authentication scheme of HTTP, as described in RFC 2617.

Settings

DDFS can be configured using the normal Disco settings file. See `disco.settings`.

3.5.4 DDFS APIs

Python API

DDFS can be used either through the native Web API or with a Python module, `disco.ddfs`, which hides the Web API behind Python functions.

Web API

We assume below that the Disco master can be found at `http://disco:8989`. All responses by the Web API are encoded in JSON.

Add a new blob

GET `http://disco:8989/ddfs/new_blob/BLOB[?replicas=N&exclude=NODE1,NODE2..]`

Requests PUT URLs for a new blob. The blob is given a prefix `BLOB`. You can use the same prefix for any number of blobs, each call to `new_blob` generates a new version of the blob. `BLOB` must match the character class `[A-Za-z0-9_\-@:]+`.

Optionally, you can request URLs for `N` replicas. However, currently DDFS only guarantees that the minimum number of replicas (`DDFS_BLOB_REPLICAS`) specified in the settings file is maintained.

You can also specify a list of nodes, `NODE1` etc., to exclude from the returned list of URLs.

Returns a list of URLs on storage nodes where the blob can be pushed using HTTP PUT requests.

Add blobs to a tag

POST `http://disco:8989/ddfs/tag/TAG`

Appends a list of URLs or replication sets to a tag TAG. If TAG doesn't exist, it is created. `TAG` must match the character class `[A-Za-z0-9_\-@:]+` (same as with `BLOB` above).

The request body must be a JSON-encoded message of the form

```
[["http://node1/blob1", "http://node2/blob1"], ["http://node1/blob2"...]]
```

which lists the replication sets to be added to the tag.

Typically, this request is made after successfully pushing blobs to storage nodes. In this case, the list of URLs is the list received from storage nodes, in response to the HTTP PUT calls, and the request body typically looks like

```
[["disco://node1/blob1", "disco://node2/blob1"], ["disco://node1/blob2"...]]
```

Alternatively, you can specify

```
[["tag://sometag1"], ["tag://sometag2"]]
```

to add links to existing tags.

Returns a list of tag URLs.

Return a tag

GET `http://disco:8989/ddfs/tag/TAG`

Returns contents of the tag `TAG`. The returned object is a JSON-encoded dictionary. It contains the following items:

- `id` Versioned tag ID
- `version` Version of the tag object
- `last-modified` When the tag was last modified
- `urls` List of URLs to tags and/or blobs

Replace contents of a tag

PUT `http://disco:8989/ddfs/tag/TAG`

Similar to POST `tag` above but replaces the existing list of URLs instead of appending URLs to it. The request follows the same format as POST `tag`.

Delete a tag

DELETE `http://disco:8989/ddfs/tag/TAG`

Delete the tag `TAG`. Note that blobs referenced by the tag are removed only when **all** references to the blobs are removed. If several tags link to the blobs, deleting a single tag does not affect the blobs.

List all tags

GET `http://disco:8989/ddfs/tags[/PREFIX0/PREFIX1...]`

Returns all tags stored in DDFS. As the returned list of tags can be potentially really long, tags can be filtered by prefix.

Special syntactic sugar is provided for filtering hierarchically named tags, that is, tags with prefixes separated by colons. You can query a certain prefix by replacing colons with slashes in the URL. For instance, all tags starting with `data:log:website` can be found with

`http://disco:8989/ddfs/tags/data/log/website`

which is equal to

`http://disco:8989/ddfs/tags/data:log:website`

Set an attribute on a tag

PUT `http://disco:8989/ddfs/tag/TAG/ATTR`

Sets the ATTR attribute of the tag TAG to a value VAL, where VAL is the request body. If the attribute did not exist, it is created; if it did exist, its value is overwritten. ATTR must match the character class `[A-Za-z0-9_\-@:]+`, while VAL should be a UTF8 string.

Get a tag attribute

GET `http://disco:8989/ddfs/tag/TAG/ATTR`

Retrieves the value of the ATTR attribute of the tag TAG. The value is returned in the request body.

Delete a tag attribute

DELETE `http://disco:8989/ddfs/tag/TAG/ATTR`

Deletes the ATTR attribute of the tag TAG. No error is returned if the tag does not possess the attribute ATTR.

Token-based Authorization

A token for a tag operation is provided in an Authorization header field for the corresponding HTTP request. The userid for the HTTP basic credential is set to the string `token`, and the token is used as the value of the password. For example, the operation to retrieve the tag TAG protected by the read-token TOKEN will look like

GET `http://disco:8989/ddfs/tag/TAG` Authorization: `Basic dG9rZW46VE9LRU4=`

where "dG9rZW46VE9LRU4=" is the base64 encoding of "token:TOKEN".

Tokens are stored in tags as attributes in a separate `ddfs:` namespace; i.e. the read-token is stored as the `ddfs:read-token` attribute of the tag, while the write-token is the `ddfs:write-token` attribute. Hence, the above-described calls to get, set, and delete attributes can also be used to perform the corresponding operations on a tag's read and write tokens.

3.5.5 Internals

This section provides information about DDFS internals, supplementing comments in the source code. This discussion is mainly interesting to developers and advanced users of DDFS and Disco.

As one might gather from the sections above, metadata (tag) operations are the central core of DDFS, mainly due to their transactional nature. Another non-trivial part of DDFS is re-replication and garbage collection of tags and blobs. These issues are discussed in more detail below.

Blob operations

Operations on blobs are reasonably simple. The client is responsible for pushing data to storage nodes, using HTTP PUT requests. `new_blob` returns a list of URLs, based on the available disk space, to which the blob data can be PUT. A node receiving data via a PUT first creates a temporary !partial file into which the blob is received, and then renames the file into the blobname on successful completion.

Getting a blob is just a matter of making a normal HTTP GET request.

Tag operations

Tags are the only mutable data type in DDFS. Each tag update creates a new version of the tag; the latest version of the tag is used to get the current contents of the tag. Updating data in a distributed system is a non-trivial task. Classical solutions include centralized lock servers, various methods based on eventual consistency and consensus protocols such as Paxos. Currently DDFS takes the first centralized approach, which is straightforward to implement in a single-master architecture.

All operations manipulating a tag are serialized, although many distinct tags can be processed concurrently. Serialization is achieved by handling each tag in a separate *gen_server* process, in `ddfs/ddfs_tag.erl` (tag server). Tag servers are instantiated on demand basis, and killed after a period of inactivity. Together, tag servers implement the master cache.

To get a tag, tag server queries all storage nodes to find all versions of the tag (see `ddfs/ddfs_tag:get_tagdata()`). From the list of all available versions, it finds replicas of the latest tag version, chooses one of them randomly, and retrieves the tag data. It is not safe to get tag data if more than *K - 1* nodes are unavailable, as in this case not all versions of the tag might be available.

After the tag data is received, it is manipulated depending on the requested operation (GET, POST, PUT). After this, an updated version of the tag is committed to DDFS. This is a critical operation, to ensure consistency of metadata.

DDFS uses a modified 3-phase commit protocol to commit the tag data back to storage nodes. The transaction proceeds as follows:

1. Choose *K* destination nodes.
2. Send the tag data to the chosen nodes, using a temporary filename.
3. If the operation fails on a node, choose another node and retry.
4. If all nodes fail before *K* replicas are written, abort.
5. Once *K* temporary replicas are written successfully, make a call to rename temporary replicas to final replicas.
6. If rename on any of the nodes succeed, the transaction succeeds, otherwise aborts.

All message passing between the storage nodes and the master is limited by a timeout. Note that it is possible, under exceptional circumstances, that less than *K* replicas are written due to lack of rollback functionality in the last step. However, the client is informed about the number of replicas written, so it can safely reissue the tag request, if it notices an insufficient number of replicas. In any case, garbage collection process will recreate the missing replicas eventually.

Tag delete

Deleting a tag is a non-trivial operation. Obviously deleting just the newest version of the tag is insufficient, as this would merely resurface a previous version. Deleting all versions of the tag is not very robust, as it is very likely that a temporarily unavailable node might contain a version of the tag, which would resurface once the node becomes available again.

DDFS uses a special tag (metatag) `+deleted` (inaccessible to the user due to the plus sign), to list deleted tags. Each tag operation checks whether the requested tag exists on this list, to hide deleted tags from the user. Actual deletion is handled by garbage collector in `ddfs/ddfs_gc_main:process_deleted()`.

The deleted tag is kept on the `+deleted` list until all known versions of the tag have been garbage collected, and a sufficient quarantine period has passed since the last seen version, to ensure that all nodes which might be temporarily unavailable have been restarted.

Due to this mechanism, it is critical that no node stays unavailable for more than `?DELETED_TAG_EXPIRES` (see `ddfs/config.hrl`) days before restarting. The period is currently one month.

Garbage collection and Re-replication

A central background process implements garbage collection and re-replication, ensuring the consistency and persistence of data and metadata in DDFS. It takes care of the following tasks:

- Remove leftover !partial. files (from failed PUT operations).
- Remove orphaned tags (old versions and deleted tags).
- Remove orphaned blobs (blobs not referred by any tag).
- Recover lost replicas for non-orphaned blobs (from lost tag updates)
- Deleted old deleted tags from the `+deleted` metatag.
- Re-replicate blobs that do not have enough replicas.
- Update tags that contain blobs that were re-replicated, and/or re-replicate tags that don't have enough replicas.

Garbage collection and re-replication are documented at the beginning of `ddfs/ddfs_gc_main.erl`. They are performed only when the cluster is in a safe state with respect to *Fault tolerance*, i.e. there are fewer than *K* failed nodes in the cluster.

Fault tolerance

DDFS piggybacks on Disco on fault-tolerance. It relies on Disco's `node_mon.erl` to monitor availability of nodes, and to blacklist unavailable nodes.

Currently many operations are set to fail if more than *K - 1* nodes are down at the same time. Given *K*-way replication, this policy gives a good guarantee that the returned data is always consistent. However, in a large cluster (say, more than 100 nodes), it is quite possible to have more than two nodes down (with the default 3-way replication) at any point of time. Increasing *K* when the cluster grows is not a good option, as this would be wasteful and it would increase latencies unnecessarily.

One possible solution to this issue is to restrict node operations to a subset of nodes instead of all of them. This would mean that the *K - 1* limit of failed nodes is imposed on a fixed subset of nodes, which is a very reasonable assumption on a cluster of any size. The node space could be partitioned using a consistent hashing mechanism, which could be integrated to `ddfs/ddfs_tag.erl` without major changes in the overall architecture of DDFS.

3.6 DiscoDB Tutorial

This tutorial is a guide to using `DiscoDBs` in Disco.

See Also:

`discodb`.

3.6.1 Create a DiscoDB

First, let's modify the *word count example* to write its output to a `DiscoDB`:

```
"""
This example could be run and the results printed from the `examples/util` directory in Disco:

disco run wordcount_ddb.WordCount http://discoproject.org/media/text/chekhov.txt
"""
from disco.core import Job
from disco.util import kvgroup

from disco.worker.classic.func import discodb_stream

class WordCount(Job):
    reduce_output_stream = discodb_stream

    @staticmethod
    def map(line, params):
        for word in line.split():
            yield word, 1

    @staticmethod
    def reduce(iter, params):
        for word, counts in kvgroup(sorted(iter)):
            yield word, str(sum(counts))
```

Notice how all we needed to do was change the *reduce_output_stream* to `disco.worker.classic.func.discodb_stream()`, and turn the count into a `str`. Remember, DiscoDBs only store byte sequences as keys and values, its up to the user to serialize objects; in this case we just use str.

3.6.2 Query a DiscoDB

Next, lets write a job to query the `DiscoDBs`.

```
"""
This example could be run and the results printed from the `examples/util` directory in Disco:

python query_ddb.py <query> <input> ...
"""
import sys
```

```
from disco.core import Job, Disco, result_iterator

from disco.worker.classic.func import nop_map
from disco.schemes.scheme_discodb import input_stream

class Query(Job):
    map_input_stream = (input_stream, )
    map = staticmethod(nop_map)

    @staticmethod
    def map_reader(discodb, size, url, params):
        for k, vs in discodb.metaquery(params):
            yield k, list(vs)

if __name__ == '__main__':
    job = Query().run(input=sys.argv[2:], params=sys.argv[1])
    for k, vs in result_iterator(job.wait()):
        print('{0}\t{1}'.format(k, sum(int(v) for v in vs)))
```

Now let's try creating our word count db from scratch, and querying it:

```
$ cd disco/examples/util
$ disco run wordcount_ddb.WordCount http://discoproject.org/media/text/chekhov.txt
WordCount@515:66e88:d39
$ disco results @ | xargs python query_ddb.py 'word'
word    18
$ disco results @?WordCount | xargs python query_ddb.py 'this | word'
this | word     217
```

Hint: The special arguments `@` and `@?<string>` are replaced by the most recent job name and the most recent job with name matching `<string>`, respectively. See `discocli`.

There are a few things to note in this example. First of all, we use a `disco.worker.classic.func.nop_map()`, since we do all the real work in our *map_reader*. We use a builtin `disco.worker.classic.input_stream()`, to return a `DiscoDB` from the file on disk, and that's the object our *map_reader* gets as a handle.

Notice also how we turn *vs* into a list. This is because `discodb.DiscoDB.metaquery()` returns lazy objects, which cannot be pickled.

Finally, notice how we run the `disco.job.Job`. We make the *input* and *params* runtime parameters, since we are pulling them from the command line. When we iterate over the results, we deserialize our counts using `int`, and sum them together.

Make sure you understand why we sum the counts together, why there could possibly be more than one count in the result (even though we only had one global *reduce* in our word count job).

Hint: Look at the second query we executed on the command line.

3.7 The Job Pack

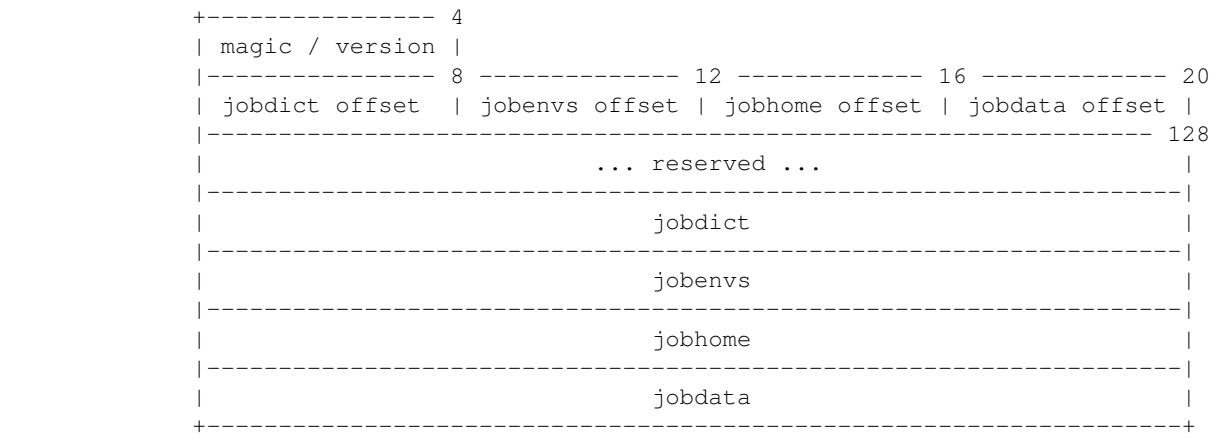
The *job pack* contains all the information needed for creating and running a Disco *job*.

The first time any *task* of a job executes on a Disco node, the job pack for the job is retrieved from the master, and *The Job Home* is unzipped into a job-specific directory.

See Also:

The Python `disco.job.JobPack` class.

File format:



3.7.1 The Job Dict

The *job dict* is a *JSON* dictionary.

`jobdict.`**`input`**
: A list of urls or a list of lists of urls. Each url is a string.

  Note: An inner list of urls gives replica urls for the same data. This lets you specify redundant versions of an input file. If a list of redundant inputs is specified, the scheduler chooses the input that is located on the node with the lowest load at the time of scheduling. Redundant inputs are tried one by one until the task succeeds. Redundant inputs require that `map?` is specified.

`jobdict.`**`worker`**
: The path to the *worker* binary, relative to the *job home*. The master will execute this binary after it unpacks it from *The Job Home*.

`jobdict.map?`
: Boolean telling whether or not this job should have a *map* phase.

`jobdict.reduce?`
: Boolean telling whether or not this job should have a *reduce* phase.

`jobdict.`**`nr_reduces`**
: Non-negative integer telling the master how many reduces to run.

  Warning: This attribute will soon be removed, as the number of reduces can be inferred in all cases.

`jobdict.`**`prefix`**
: String giving the prefix the master should use for assigning a unique job name.

  Note: Only characters in `[a-zA-Z0-9_]` are allowed in the prefix.

`jobdict.`**`scheduler`**
Dictionary of options for the job scheduler. Currently supports the following keys:

- *max_cores* - use at most this many cores (applies to both map and reduce). Default is `2**31`.
- *force_local* - always run task on the node where input data is located; never use HTTP to access data remotely.
- *force_remote* - never run task on the node where input data is located; always use HTTP to access data remotely.

New in version 0.2.4.

`jobdict.`**`owner`**
String name of the owner of the *job*.

3.7.2 Job Environment Variables

A *JSON* dictionary of environment variables (with string keys and string values). The master will set these in the environment before running the `jobdict.worker`.

3.7.3 The Job Home

The *job home* directory serialized into *ZIP* format. The master will unzip this before running the `jobdict.worker`. The *worker* is run with this directory as its working directory.

In addition to the *worker* executable, the *job home* can be populated with files that are needed at runtime by the worker. These could either be shared libraries, helper scripts, or parameter data.

Note: The .disco subdirectory of the *job home* is reserved by Disco.

The job home is shared by all tasks of the same job on the same node. That is, if the job requires two map task and two reduce task executions on a particular node, then the job home will be unpacked only once on that node, but the worker executable will be executed four times in the job home directory, and it is also possible for some of these executions to be concurrent. Thus, the worker should take care to use unique filenames as needed.

3.7.4 Additional Job Data

Arbitrary data included in the *job pack*, used by the *worker*. A running worker can access the job pack at the path specified by `jobfile` in the response to the *TASK* message.

3.7.5 Creating and submitting a Job Pack

The jobpack can be constructed and submitted using the `disco job` command.

3.8 Out-of-band results

New in version 0.2. In addition to standard input and output streams, map and reduce tasks can output results through an auxiliary channel called *out-of-band results* (OOB). In contrast to the standard output stream, which is sequential, OOB results can be accessed by unique keys.

Out-of-band results should not be used as a substitute for the normal output stream. Each OOB key-value pair is saved to an individual file which waste space when values are small and which are inefficient to random-access in bulk. Due to these limitations, OOB results are mainly suitable, e.g for outputting statistics and other metadata about the actual results.

To prevent rogue tasks from overwhelming nodes with a large number of OOB results, each is allowed to output 1000 results at maximum. Hitting this limit is often a sign that you should use the normal output stream for you results instead.

You can not use OOB results as a communication channel between concurrent tasks. Concurrent tasks need to be independent to preserve desirable fault-tolerance and scheduling characteristics of the map/reduce paradigm. However, in the reduce phase you can access OOB results produced in the preceding map phase. Similarly you can access OOB results produced by other finished jobs, given a job name.

You can retrieve OOB results outside tasks using the `disco.core.Disco.oob_list()` and `disco.core.Disco.oob_get()` functions.

3.9 Style Guide for Disco Code

This document contains the coding conventions for the Disco codebase.

3.9.1 Erlang

Unless otherwise specified, Erlang Programming Rules apply.

Also, the use of -spec annotations and Dialyzer is highly recommended before checking in commits.

3.9.2 Python

Unless otherwise specified, guidelines from **PEP 8** apply.

3.9.3 Documentation

- inline with code when possible

3.10 The Disco Worker Protocol

Note: This protocol is used by a *worker* in Disco. You don't need to write your own worker to run a *job*, you can usually use the standard `disco.worker`.

A *worker* is an executable that can run a task, given by the Disco *master*. The executable could either be a binary executable, or a script that launches other executables. In order to be usable as a Disco worker, an executable needs to implement the *The Disco Worker Protocol*. For instance, ODisco, which implements the Disco Worker Protocol, allows you to write Disco jobs in the O'Caml language.

There are two parts to using a binary executable as a Disco worker. The first part is the submission of the executable to the Disco master. The second part is the interaction of the executable with its execution environment, when running a *task*.

This document describes how the worker should communicate with Disco, while it is executing. For more on creating and submitting the *job pack*, see *The Job Pack*.

Note: The same executable is launched to perform *map* and *reduce* tasks.

A worker-initiated synchronous message-based protocol is used for this communication. That is, all communication is initiated by the worker by sending a message to Disco, and Disco in turn sends a reply to each message. A worker could pipeline several messages to Disco, and it will receive a response to each message in order.

The worker sends messages over *stderr* to Disco, and receives responses over *stdin*. Disco should respond within *600 seconds*: it is advised for the worker to timeout after waiting this long.

Note: Workers should not write anything to *stderr*, except messages formatted as described below. A worker's *stdout* is also initially redirected to stderr.

Note: In Python, `Workers` wrap all exceptions, and everything written to *stdout*, with an appropriate message to Disco (on *stderr*). For instance, you can raise a `disco.error.DataError` to abort the worker and try again on another host. For other types of failures, it is usually best to just let the worker catch the exception.

3.10.1 Message Format

Messages in the protocol, both from and to the worker, are in the format:

<*name*> 'SP' <*payload-len*> 'SP' <*payload*> '\n'

where 'SP' denotes a single space character, and <*name*> is one of:

DONE
ERROR
FAIL
FATAL
INPUT
INPUT_ERR
MSG
OK
OUTPUT
PING
RETRY
TASK
WAIT
WORKER

<*payload-len*> is the length of the <*payload*> in bytes, and <*payload*> is a *JSON* formatted term.

Note that messages that have no payload (see below) actually contain an empty JSON string <*payload*> = *""* and <*payload-len*> = *2*.

3.10.2 Messages from the Worker to Disco

WORKER

Announce the startup of the worker.

The payload is a dictionary containing the following information:

"version" The version of the message protocol the worker is using, as a string. The current version is *"1.0"*.

"pid" The integer *pid* of the worker.

The worker should send this so it can be properly killed, (e.g. if there's a problem with the *job*). This is currently required due to limitations in the Erlang support for external spawned processes.

The worker should send a *WORKER* message before it sends any others. Disco should respond with an *OK* if it intends to use the same version.

TASK

Request the task information from Disco.

The worker should send a *TASK* message with no payload. Disco should respond with a *TASK* message, and a payload containing the following task information as a dictionary:

"host" The host the *task* is running on.

"master" The host the *master* is running on.

"jobname" The name of the *job* this task is a member of.

"taskid" The internal Disco id of the *task*.

"mode" The mode or phase of the *job*. This is currently either *"map"* or *"reduce"*, although more modes may be added in future releases.

"disco_port" The value of the `DISCO_PORT` setting, which is the port the Disco master is running on, and the port used to retrieve data from Disco and *DDFS*. This is used to convert URLs with the *disco* and *ddfs* schemes into *http* URLs.

"put_port" The value of the `DDFS_PUT_PORT` setting. This can be used by the worker to upload results to *DDFS*.

"disco_data" The value of the `DISCO_DATA` setting.

"ddfs_data" The value of the `DDFS_DATA` setting. This can be used to read *DDFS* data directly from the local filesystem after it has been ascertained that the *DDFS* data is indeed local to the current host.

"jobfile" The path to the *The Job Pack* file for the current job. This can be used to access any *Additional Job Data* that was uploaded as part of the *The Job Pack*.

INPUT

Request input for the task from Disco.

To get the complete list of current inputs for the task, the worker can send an *INPUT* message with no payload. Disco should respond with an *INPUT* message, and a payload containing a two-element tuple (list in *JSON*).

The first element is a flag, which will either be *'more'* or *'done'*. *'done'* indicates that the input list is complete, while *'more'* indicates that more inputs could be added to the list in the future, and the worker should continue to poll for new inputs.

The second element is a list of inputs, where each input is a specified as a three-element tuple:

```
input_id, status, replicas
```

where *input_id* is an integer identifying the input, and *status* and *replicas* follow the format:

```
status ::= 'ok' | 'busy' | 'failed'
replicas ::= [replica]
replica ::= rep_id, replica_location
```

It is possible for an input to be available at multiple locations; each such location is called a *replica*. A *rep_id* is an integer identifying the replica.

The *replica_location* is specified as a URL. The protocol scheme used for the *replica_location* could be one of *http*, *disco*, *dir* or *raw*. A URL with the *disco* scheme is to be accessed using HTTP at the *disco_port* specified in the *TASK* response from Disco. The *raw* scheme denotes that the URL itself (minus the scheme) is the data for the task. The data needs to be properly URL encoded, for instance using Base64 encoding. The *dir* is like the *disco* scheme, except that the file pointed to contains lines of the form

<*label*> 'SP' <*url*> '\n'

The *'label'* comes from the *'label'* specified in an *OUTPUT* message by a task, while the *'url'* points to a file containing output data generated with that label. This is currently how labeled output data is communicated by upstream tasks to downstream ones, e.g. from map tasks to reduce tasks, or from tasks in the final phase of a previous job to the tasks in the first phase of a subsequent job (see *Data Flow in Disco Jobs*).

One important optimization is to use the local filesystem instead of HTTP for accessing inputs when they are local. This can be determined by comparing the URL hostname with the *host* specified in the *TASK* response, and then converting the URL path into a filesystem path using the *disco_data* or *ddfs_data* path prefixes for URL paths beginning with *disco/* and *ddfs/* respectively.

The common input status will be *'ok'* - this indicates that as far as Disco is aware, the input should be accessible from at least one of the specified replica locations. The *'failed'* status indicates that Disco thinks that the specified locations are inaccessible; however, the worker can still choose to ignore this status and attempt retrieval from the specified locations. A *'busy'* status indicates that Disco is in the process of generating more replicas for this input, and the worker should poll for additional replicas if needed.

It is recommended that the worker attempts the retrieval of an input from the replica locations in the order specified in the response. That is, it should attempt retrieval from the first replica, and if that fails, then try the second replica location, and so on.

When a worker polls for any changes in task's input, it is preferable not to repeatedly retrieve information for inputs already successfully processed. In this case, the worker can send an *INPUT* message with an *'exclude'* payload that specifies the *input_ids* to exclude in the response. In this case, the *INPUT* message from the worker should have the following payload:

```
['exclude', [input_id]]
```

On the other hand, when a worker is interested in changes in replicas for a particular set of inputs, it can send an *INPUT* message with an *include* payload that requests information only for the specified *input_ids*. The *INPUT* message from the worker in this case should have the following payload:

```
['include', [input_id]]
```

INPUT_ERR

Inform Disco that about failures in retrieving inputs.

The worker should inform Disco if it cannot retrieve an input due to failures accessing the replicas specified by Disco in the *INPUT* response. The payload of this message specifies the input and the failed replica locations using their identifiers, as follows:

```
[input_id, [rep_id]]
```

If there are alternative replicas that the worker can try, Disco should respond with a *RETRY* message, with a payload specifying new replicas:

```
[[rep_id, replica_location]]
```

If there are no alternatives, and it is not possible for Disco to generate new alternatives, Disco should reply with a *FAIL* message (which has no payload).

If Disco is in the process of generating new replicas, it should reply with a *WAIT* message and specify an integer duration in seconds in the payload. The worker should then poll for any new replicas after the specified duration.

MSG

Send a message (i.e. to be displayed in the ui).

The worker can send a *MSG* message, with a payload containing a string. Disco should respond with an *OK*.

OUTPUT

The worker should report its output(s) to Disco.

For each output generated by the worker, it should send an *OUTPUT* message specifying the type and location of the output, and optionally, its label:

```
[output_location, output_type, label]
```

The *output_type* can be either *'disco'*, *'part'* or *'tag'*. *'disco'* and *'part'* outputs are used for local outputs, while *'tag'* specifies a location within *DDFS*.

Local outputs have locations that are paths relative to *jobhome*.

Labels are currently only interpreted for *'part'* outputs, and are integers that are used to denote the partition for the output.

DONE

Inform Disco that the worker is finished.

The worker should only send this message (which has no payload) after syncing all output files, since Disco normally terminates the worker when this message is received. The worker should not exit immediately after sending this message, since there is no guarantee if the message will be received by Disco if the worker exits. Instead, the worker should wait for the response from Disco (as it should for all messages).

ERROR

Report a failed input or transient error to Disco.

The worker can send a *ERROR* message with a payload containing the error message as a string. This message will terminate the worker, but not the job. The current task will be retried by Disco. See also the information above for the *DONE* message.

FATAL

Report a fatal error to the master.

The worker can send an *FATAL* message, with a payload containig the error message as a string. This message will terminate the entire job. See also the information above for the *DONE* message.

PING

No-op - always returns *OK*.

Worker can use *PING* as a heartbeat message, to make sure that the master is still alive and responsive.

3.10.3 Messages from Disco to the Worker

OK

A generic response from Disco. This message has the payload *"ok"*.

FAIL

A possible response from Disco for an *INPUT_ERR* message, as described above.

RETRY

A possible response from Disco for an *INPUT_ERR* message, as described above.

WAIT

A possible response from Disco for an *INPUT_ERR* message, as described above.

3.10.4 Sessions of the Protocol

On startup, the worker should first send the *WORKER* message, and then request the *TASK* information. The *taskid* and *mode* in the *TASK* response can be used, along with the current system time, to create a working directory within which to store any scratch data that will not interact with other, possibly concurrent, workers computing other tasks of the same job. These messages can be said to constitute the initial handshake of the protocol.

The crucial messages the worker will then send are the *INPUT* and *OUTPUT* messages, and often the *INPUT_ERR* messages. The processing of the responses to *INPUT* and *INPUT_ERR* will be determined by the application. The worker will usually end a successful session with one or more *OUTPUT* messages followed by the *DONE* message. Note that it is possible for a successful session to have several *INPUT_ERR* messages, due to transient network conditions in the cluster as well as machines going down and recovering.

An unsuccessful session is normally ended with an *ERROR* or *FATAL* message from the worker. An *ERROR* message terminates the worker, but does not terminate the job; the task will possibly be retried on another host in the cluster. A *FATAL* message, however, terminates both the worker, and the entire job.

3.10.5 Considerations when implementing a new Worker

You will need some simple and usually readily available tools when writing a new worker that implements the Disco protocol. Parsing and generating messages in the protocol requires a *JSON* parser and generator. Fetching data from the replica locations specified in the Disco *INPUT* responses will need an implementation of a simple HTTP client. This HTTP client and *JSON* tool can also be used to persistently store computed results in *DDFS* using the REST API.

The protocol does not specify the data contents of the Disco job inputs or outputs. This leaves the implementor freedom to choose an appropriate format for marshalling and parsing data output by the worker tasks. This choice will have an impact on how efficiently the computation is performed, and how much disk space the marshalled output uses.

CHAPTER 4

Reference

4.1 disco – API Reference

4.1.1 disco.core – Disco Core Library

The disco.core module provides a high-level interface for communication with the Disco master. It provides functions for submitting new jobs, querying the status of the system, and getting results of jobs.

class disco.core.**Disco**(*master=None*, *settings=None*, *proxy=None*)

The Disco object provides an interface to the Disco master. It can be used to query the status of the system, or to submit a new job.

Parameters **master** (*url*) – address of the Disco master, e.g. disco://localhost.

See Also:

Disco.new_job() and disco.job for more on creating jobs.

blacklist(*node*)

Blacklists *node* so that tasks are no longer run on it. New in version 0.2.4.

clean(*jobname*)

Deletes job metadata. Deprecated since version 0.4: Use Disco.purge() to delete job results, deleting job metadata only is strongly discouraged.

Note: After the job has been cleaned, there is no way to obtain the result urls from the master. However, no data is actually deleted by Disco.clean(), in contrast to Disco.purge().

---

events(*jobname*, *offset=0*)

Returns an iterator that iterates over job events, ordered by time. It is safe to call this function while the job is running, thus it provides an efficient way to monitor job events continuously. The iterator yields tuples offset, event.

Parameters **offset** (*int*) – skip events that occurred before this *offset*

New in version 0.2.3.

See Also:

DISCO_EVENTS for information on how to enable the console output of job events.

jobinfo (*jobname*)
Returns a dict containing information about the job.

joblist ()
Returns a list of jobs and their statuses.

jobpack (*jobname*)
Return the `disco.job.JobPack` submitted for the job.

kill (*jobname*)
Kills the job.

new_job (*name*, ***jobargs*)
Submits a new job request to the master using `disco.job.Job`.

nodeinfo ()
Returns a dictionary describing status of the nodes that are managed by this Disco master.

oob_get (*jobname*, *key*)
Returns an out-of-band value assigned to *key* for the job.

OOB data can be stored and retrieved for job tasks using `disco.task.Task.get()` and `disco.task.Task.put()`.

oob_list (*jobname*)
Returns all out-of-band keys for the job.

See Also:

Out-of-band results

profile_stats (*jobname*, *mode=''*, *stream=<open file '<stdout>', mode 'w' at 0x7f322ffa7150>*)
Returns results of job profiling. *The Job Dict* must have had the `profile` flag enabled.

Parameters

- **mode** (*'map' or 'reduce' or ''*) – restricts results to the map or reduce phase, or not.
- **stream** (*file-like object*) – alternate output stream. See the pstats.Stats constructor.

The function returns a pstats.Stats object. For instance, you can print out results as follows:

```
job.profile_stats().sort_stats('cumulative').print_stats()
```

New in version 0.2.1.

purge (*jobname*)
Deletes all metadata and data related to the job.

results (*jobspec*, *timeout=2000*)
Returns a list of results for a single job or for many concurrently running jobs, depending on the type of *jobspec*.

Parameters

- **jobspec** (`disco.job.Job`, string, or list) – If a job or job name is provided, return a tuple which looks like:

  ```
  status, results
  ```

  If a list is provided, return two lists: inactive jobs and active jobs. Both the lists contain elements of the following type:

```
jobname, (status, results)
```

where status is one of: `'unknown job'`, `'dead'`, `'active'`, or `'ready'`.

- **timeout** (*int*) – wait at most this many milliseconds, for at least one on the jobs to finish.

Using a list of jobs is a more efficient way to wait for multiple jobs to finish. Consider the following example that prints out results as soon as the jobs (initially `active`) finish:

```
while active:
  inactive, active = disco.results(jobs)
  for jobname, (status, results) in inactive:
    if status == 'ready':
      for k, v in result_iterator(results):
        print(k, v)
      disco.purge(jobname)
```

Note how the list of active jobs, `active`, returned by `Disco.results()`, can be used as the input to this function as well.

wait (*jobname*, *poll_interval=2*, *timeout=None*, *clean=False*, *show=None*)
Block until the job has finished. Returns a list of the result urls.

Parameters

- **poll_interval** (*int*) – the number of seconds between job status requests.
- **timeout** (*int or None*) – if specified, the number of seconds before returning or raising a `disco.JobError`.
- **clean** (*bool*) – if *True*, call `Disco.clean()` when the job has finished. Deprecated since version 0.4.
- **show** (*bool or string*) – enables console output of job events. The default is provided by `DISCO_EVENTS`. New in version 0.2.3.

whitelist (*node*)
Whitelists *node* so that the master may submit tasks to it. New in version 0.2.4.

disco.core.**classic_iterator** (*urls*, *reader=function disco_input_stream*, *input_stream=(function task_input_stream*, *)*, *notifier=function notifier*, *params=None*, *ddfs=None*)
An iterator over records as seen by the classic map interface.

Parameters

- **reader** (`disco.classic.worker.func.input_stream()`) – shortcut for the last input stream applied.
- **input_stream** (sequence of `disco.classic.worker.func.input_stream()`) – used to read from a custom file format.
- **notifier** (`disco.classic.worker.func.notifier()`) – called when the task opens a url.

disco.core.**result_iterator** (**args*, ***kwargs*)
Backwards compatible alias for `classic_iterator()`

4.1.2 `disco.ddfs` – Client interface for Disco Distributed Filesystem

See also: *Disco Distributed Filesystem*.

Note: Parameters below which are indicated as tags can be specified as a *tag://* URL, or the name of the tag.

class disco.ddfs.**DDFS**(*master=None*, *proxy=None*, *settings=None*)
Opens and encapsulates a connection to a DDFS master.

Parameters **master** – address of the master, for instance `disco://localhost`.

attrs(*tag*, *token=None*)
Get a list of the attributes of the tag `tag` and their values.

blobs(*tag*, *ignore_missing=True*, *token=None*)
Walks the tag graph starting at *tag*.

Yields only the terminal nodes of the graph (*blobs*).

Parameters **ignore_missing** (*bool*) – Whether or not missing tags will raise a `disco.error.CommError`.

chunk(*tag*, *urls*, *replicas=None*, *retries=10*, *delayed=False*, *update=False*, *token=None*, *chunk_size=67108864*, ***kwargs*)
Chunks the contents of *urls*, pushes the chunks to ddfs and tags them with *tag*.

delattr(*tag*, *attr*, *token=None*)
Delete the attribute `attr` of the tag `tag`.

delete(*tag*, *token=None*)
Delete `tag`.

exists(*tag*)
Returns whether or not `tag` exists.

get(*tag*, *token=None*)
Return the tag object stored at `tag`.

getattr(*tag*, *attr*, *token=None*)
Return the value of the attribute `attr` of the tag `tag`.

list(*prefix=''*)
Return a list of all tags starting with `prefix`.

pull(*tag*, *blobfilter=<function <lambda> at 0x45635f0>*, *token=None*)
Iterate over the blobs in a `tag` after optionally applying a `blobfilter` over the blob names.

push(*tag*, *files*, *replicas=None*, *retries=10*, *delayed=False*, *update=False*, *token=None*)
Pushes a bunch of files to ddfs and tags them with *tag*.

Parameters **files** (a list of `paths`, `(path, name)`-tuples, or `(fileobject, name)`-tuples.) – the files to push as blobs to DDFS. If names are provided, they will be used as prefixes by DDFS for the blobnames. Names may only contain chars in `r'[^A-Za-z0-9_\-@:]'`.

put(*tag*, *urls*, *token=None*)
Put the list of `urls` to the tag `tag`.

Warning: Generally speaking, concurrent applications should use `DDFS.tag()` instead.

setattr(*tag*, *attr*, *val*, *token=None*)
Set the value of the attribute `attr` of the tag `tag`.

tag (*tag*, *urls*, *token=None*, ***kwargs*)
Append the list of `urls` to the `tag`.

urls (*tag*, *token=None*)
Return the urls in the `tag`.

walk (*tag*, *ignore_missing=True*, *tagpath=()*, *token=None*)
Walks the tag graph starting at *tag*.

Yields a 3-tuple *(tagpath, tags, blobs)*.

Parameters **ignore_missing** (*bool*) – Whether or not missing tags will raise a `disco.error.CommError`.

4.1.3 `disco.error` – Errors with special meaning in Disco

exception `disco.error.`**CommError** (*msg*, *url*, *code=None*)
An error caused by the inability to access a resource over the network.

exception `disco.error.`**DataError** (*msg*, *url*, *code=None*)
An error caused by an inability to access a data resource.

These errors are treated specially by Disco master in that they are assumed to be recoverable. If Disco thinks an error is recoverable, it will retry the task on another node. If the same task fails on several different nodes, the master may terminate the job.

exception `disco.error.`**DiscoError**
The base class for all Disco errors

exception `disco.error.`**JobError** (*job*, *message*)
An error that occurs when an invalid job request is made. Instances of this error have the following attributes:

job
The `disco.job.Job` for which the request failed.

message
The error message.

4.1.4 `disco.job` – Disco Jobs

This module contains the core objects for creating and interacting with Disco jobs. Often, `Job` is the only thing you need in order to start running distributed computations with Disco.

Jobs in Disco are used to encapsulate and schedule computation pipelines. A job specifies a *worker*, the worker environment, a list of inputs, and some additional information about how to run the job. For a full explanation of how the job is specified to the Disco *master*, see *The Job Pack*.

A typical pattern in Disco scripts is to run a job synchronously, that is, to block the script until the job has finished. This can be accomplished using the `Job.wait()` method:

```
from disco.job import Job
results = Job(name).run(**jobargs).wait()
```

class `disco.job.`**Job** (*name=None*, *master=None*, *worker=None*, *settings=None*)
Creates a Disco Job with the given name, master, worker, and settings. Use `Job.run()` to start the job.

Parameters

- **name** (*string*) – the job name. When you create a handle for an existing job, the name is used as given. When you create a new job, the name given is used as the `jobdict.prefix` to construct a unique name, which is then stored in the instance.
- **master** (url of master or `disco.core.Disco`) – the Disco master to use for submitting or querying the job.
- **worker** (`disco.worker.Worker`) – the worker instance used to create and run the job. If none is specified, the job creates a worker using its `Job.Worker` attribute.

`Worker`
Defaults to `disco.worker.classic.worker.Worker`. If no *worker* parameter is specified, `Worker` is called with no arguments to construct the `worker`.

Note: Note that due to the mechanism used for submitting jobs to the Disco cluster, the submitted job class cannot belong to the *__main__* module, but needs to be qualified with a module name. See `examples/faq/chain.py` for a simple solution for most cases.

`proxy_functions` = ('clean', 'events', 'kill', 'jobinfo', 'jobpack', 'oob_get', 'oob_list', 'profile_stats', 'purge', 'results',
These methods from `disco.core.Disco`, which take a jobname as the first argument, are also accessible through the `Job` object:

- `disco.core.Disco.clean()`
- `disco.core.Disco.events()`
- `disco.core.Disco.kill()`
- `disco.core.Disco.jobinfo()`
- `disco.core.Disco.jobpack()`
- `disco.core.Disco.oob_get()`
- `disco.core.Disco.oob_list()`
- `disco.core.Disco.profile_stats()`
- `disco.core.Disco.purge()`
- `disco.core.Disco.results()`
- `disco.core.Disco.wait()`

For instance, you can use *job.wait()* instead of *disco.wait(job.name)*. The job methods in `disco.core.Disco` come in handy if you want to manipulate a job that is identified by a jobname instead of a `Job` object.

`run`(***jobargs*)
Creates the `JobPack` for the worker using `disco.worker.Worker.jobdict()`, `disco.worker.Worker.jobenvs()`, `disco.worker.Worker.jobhome()`, `disco.task.jobdata()`, and attempts to submit it. This method executes on the client submitting a job to be run. More information on how job inputs are specified is available in `disco.worker.Worker.jobdict()`. The default worker implementation is called `classic`, and is implemented by `disco.worker.classic.worker`.

Parameters jobargs (*dict*) – runtime parameters for the job. Passed to the `disco.worker.Worker` methods listed above, along with the job itself. The interpretation of the jobargs is performed by the worker interface in `disco.worker.Worker` and the class implementing that interface (which defaults to `disco.worker.classic.worker`).

Raises `disco.error.JobError` if the submission fails.

Returns the `Job`, with a unique name assigned by the master.

class `disco.job.`**JobPack**(*jobdict*, *jobenvs*, *jobhome*, *jobdata*)

This class implements *The Job Pack* in Python. The attributes correspond to the fields in the *job pack* file. Use `dumps()` to serialize the `JobPack` for sending to the master.

jobdict

The dictionary of job parameters for the *master*.

See also *The Job Dict*.

jobenvs

The dictionary of environment variables to set before the *worker* is run.

See also *Job Environment Variables*.

jobhome

The zipped archive to use when initializing the *job home*. This field should contain the contents of the serialized archive.

See also *The Job Home*.

jobdata

Binary data that the builtin `disco.worker.Worker` uses for serializing itself.

See also *Additional Job Data*.

dumps()

Return the serialized `JobPack`.

Essentially encodes the `jobdict` and `jobenvs` dictionaries, and prepends a valid header.

classmethod load(*jobfile*)

Load a `JobPack` from a file.

4.1.5 `disco.schemes` – Default input streams for URL schemes

By default, Disco looks at an input URL and extracts its scheme in order to figure out which input stream to use.

When Disco determines the URL scheme, it tries to import the name *input_stream* from *disco.schemes.scheme_[SCHEME]*, where *[SCHEME]* is replaced by the scheme identified. For instance, an input URL of *http://discoproject.org* would import and use `disco.schemes.scheme_http.input_stream()`.

`disco.schemes.scheme_disco.`**input_stream**(*fd*, *size*, *url*, *params*)

Opens the path on host locally if it is local, otherwise over http.

`disco.schemes.scheme_file.`**input_stream**(*fd*, *size*, *url*, *params*)

Opens the url locally on the node.

`disco.schemes.scheme_http.`**input_stream**(*fd*, *sze*, *url*, *params*)

Opens the specified url using an http client.

`disco.schemes.scheme_raw.`**input_stream**(*fd*, *sze*, *url*, *params*)

Opens a StringIO whose data is everything after the url scheme.

For example, *raw://hello_world* would return *hello_world* when read by the task.

4.1.6 disco.settings – Disco Settings

Settings can be specified in a Python file and/or using environment variables. Settings specified in environment variables override those stored in a file. The default settings are intended to make it easy to get Disco running on a single node. **make install** will create a more reasonable settings file for a cluster environment, and put it in `/etc/disco/settings.py`

Disco looks in the following places for a settings file:

- The settings file specified using the command line utility `--settings` option.
- `~/.disco`
- `/etc/disco/settings.py`

Possible settings for Disco are as follows:

DISCO_DATA
Directory to use for writing data. Default obtained using `os.path.join(DISCO_ROOT, data)`.

DISCO_DEBUG
Sets the debugging level for Disco. Default is `1`.

DISCO_ERLANG
Command used to launch Erlang on all nodes in the cluster. Default usually `erl`, but depends on the OS.

DISCO_EVENTS
If set, events are logged to *stdout*. If set to `json`, events will be written as JSON strings. If set to `nocolor`, ANSI color escape sequences will not be used, even if the terminal supports it. Default is unset (the empty string).

DISCO_FLAGS
Default is the empty string.

DISCO_HOME
The directory which Disco runs out of. If you run Disco out of the source directory, you shouldn't need to change this. If you use `make install` to install Disco, it will be set properly for you in `/etc/disco/settings.py`.

DISCO_HTTPD
Command used to launch *lighttpd*. Default is `lighttpd`.

DISCO_MASTER_HOME
Directory containing the Disco `master` directory. Default is obtained using `os.path.join(DISCO_HOME, 'master')`.

DISCO_MASTER_HOST
The hostname of the master. Default obtained using `socket.gethostname()`.

DISCO_MASTER_ROOT
Directory to use for writing master data. Default obtained using `os.path.join(DISCO_DATA, '_%s' % DISCO_NAME)`.

DISCO_MASTER_CONFIG
Directory to use for writing cluster configuration. Default obtained using `os.path.join(DISCO_ROOT, '%s.config' % DISCO_NAME)`.

DISCO_NAME
A unique name for the Disco cluster. Default obtained using `'disco_%s' % DISCO_PORT`.

DISCO_LOG_DIR
Directory where log-files are created. The same path is used for all nodes in the cluster. Default is obtained using `os.path.join(DISCO_ROOT, 'log')`.

DISCO_PID_DIR
Directory where pid-files are created. The same path is used for all nodes in the cluster. Default is obtained using `os.path.join(DISCO_ROOT, 'run')`.

DISCO_PORT
The port the workers use for *HTTP* communication. Default is `8989`.

DISCO_ROOT
Root directory for Disco-written data and metadata. Default is obtained using `os.path.join(DISCO_HOME, 'root')`.

DISCO_ROTATE_LOG
Whether to rotate the master log on startup. Default is `False`.

DISCO_USER
The user Disco should run as. Default obtained using `os.getenv(LOGNAME)`.

DISCO_JOB_OWNER
User name shown on the job status page for the user who submitted the job. Default is the login name @ host.

DISCO_WWW_ROOT
Directory that is the document root for the master *HTTP* server. Default obtained using `os.path.join(DISCO_MASTER_HOME, www)`.

DISCO_GC_AFTER
How long to wait before garbage collecting job-generated intermediate and result data. Only results explictly saved to DDFS won't be garbage collected. Default is `100 * 365 * 24 * 60 * 60` (100 years). (Note that this setting does not affect data in DDFS.)

DISCO_WORKER_MAX_MEM
How much memory can be used by worker in total. Worker calls resource.setrlimit(RLIMIT_AS, limit) to set the limit when it starts. Can be either a percentage of total available memory or an exact number of bytes. Note that `setrlimit` behaves differently on Linux and Mac OS X, see *man setrlimit* for more information. Default is `80%` i.e. 80% of the total available memory.

Settings to control the proxying behavior:

DISCO_PROXY_ENABLED
If set, enable proxying through the master. This is a master-side setting (set in `master:/etc/disco/settings.py`). Default is `''`.

DISCO_PROXY
The address of the proxy to use on the client side. This is in the format `http://<proxy-host>:<proxy-port>`, where `<proxy-port>` normally matches the value of `DISCO_PROXY_PORT` set on the master.

Default is `''`.

DISCO_PROXY_PORT
The port the master proxy should run on. This is master-side setting (set in `master:/etc/disco/settings.py`). Default is `8999`.

Settings to control the scheduler behavior:

DISCO_SCHEDULER
The type of scheduler that disco should use. The only options are *fair* and *fifo*. Default is `fair`.

DISCO_SCHEDULER_ALPHA
Parameter controlling how much the `fair` scheduler punishes long-running jobs vs. short ones. Default is .001 and should usually not need to be changed.

Settings used by the testing environment:

DISCO_TEST_DISCODB
Whether or not to run `discodb` tests. Default is `''`.

DISCO_TEST_HOST
The hostname that the test data server should bind on. Default is `DISCO_MASTER_HOST`.

DISCO_TEST_PORT
The port that the test data server should bind to. Default is `9444`.

Settings used by DDFS:

DDFS_ROOT
Deprecated since version 0.4. Use `DDFS_DATA` instead. Only provided as a default for backwards compatability. Default is obtained using `os.path.join(DISCO_ROOT, 'ddfs')`.

DDFS_DATA
The root data directory for DDFS. Default is obtained using `DDFS_ROOT`.

DDFS_PUT_PORT
The port to use for writing to DDFS nodes. Must be open to the Disco client unless proxying is used. Default is `8990`.

DDFS_PUT_MAX
The maximum default number of retries for a *PUT* operation. Default is `3`.

DDFS_GET_MAX
The maximum default number of retries for a *GET* operation. Default is `3`.

DDFS_READ_TOKEN
The default read authorization token to use. Default is `None`.

DDFS_WRITE_TOKEN
The default write authorization token to use. Default is `None`.

DDFS_GC_INITIAL_WAIT
The amount of time to wait after startup before running GC (in minutes). Default is `''`, which triggers an internal default of 5 minutes.

DDFS_PARANOID_DELETE
Instead of deleting unneeded files, DDFS garbage collector prefixes obsolete files with `!trash.`, so they can be safely verified/deleted by an external process. For instance, the following command can be used to finally delete the files (assuming that `DDFS_DATA = "/srv/disco/ddfs"`):

```
find /srv/disco/ddfs/ -perm 600 -iname '!trash*' -exec rm {} \;
```

Default is `''`.

The following settings are used by DDFS to determine the number of replicas for data/metadata to keep (it is not recommended to use the provided defaults in a multinode cluster):

DDFS_TAG_MIN_REPLICAS
The minimum number of replicas for a tag operation to succeed. Default is `1`.

DDFS_TAG_REPLICAS
The number of replicas of tags that DDFS should aspire to keep. Default is `1`.

DDFS_BLOB_REPLICAS
The number of replicas of blobs that DDFS should aspire to keep. Default is `1`.

4.1.7 disco.task – Disco Tasks

This module defines objects for interfacing with *tasks* assigned by the master.

class disco.task.**Task**(*host=''*, *jobfile=''*, *jobname=''*, *master=None*, *disco_port=None*, *put_port=None*, *ddfs_data=''*, *disco_data=''*, *mode=None*, *taskid=-1*)

Encapsulates the information specific to a particular *task* coming from the master.

host
The name of the host this task is running on.

jobname
The name of the *job* this task is part of.

master
The name of the master host for this task.

mode
The phase which this task is part of. Currently either *map* or *reduce*.

taskid
The id of this task, assigned by the master.

uid
A unique id for this particular task instance.

get(*key*)
Gets an out-of-band result for the task with the key *key*.

Given the semantics of OOB results, this means that only the reduce phase can access results produced in the preceding map phase.

path(*name*)

Returns The *name* joined to the `taskpath`.

put(*key*, *value*)
Stores an out-of-band result *value* (bytes) with the key *key*.

Key must be unique in this job. Maximum key length is 256 characters. Only characters in the set `[a-zA-Z_\-:0-9@]` are allowed in the key.

disco.task.**jobdata**(**objs*)

Returns *Additional Job Data* needed for instantiating the `disco.job.Job` on the node.

4.1.8 disco.util – Helper functions

This module provides utility functions that are mostly used by Disco internally. Deprecated since version 0.4: `disco.util.data_err()`, `disco.util.err()`, and `disco.util.msg()` will be removed completely in the next release, in favor of using normal Python **raise** and **print** statements.

disco.util.**data_err**(*message*, *url*)
Deprecated since version 0.4: raise `disco.error.DataError` instead. Raises a `disco.error.DataError`. A data error should only be raised if it is likely that the error is transient. Typically this function is used by map readers to signal a temporary failure in accessing an input file.

disco.util.**err**(*message*)
Deprecated since version 0.4: raise `disco.error.DiscoError` instead. Raises a `disco.error.DiscoError`. This terminates the job.

disco.util.**jobname**(*url*)
Extracts the job name from *url*.

This function is particularly useful for using the methods in `disco.core.Disco` given only the results of a job. A typical case is that you no longer need the results. You can tell Disco to delete the unneeded data as follows:

```
from disco.core import Disco
from disco.util import jobname

Disco().purge(jobname(results[0]))
```

disco.util.**kvgroup**(*kviter*)
Group the values of consecutive keys which compare equal.

Takes an iterator over `k, v` pairs, and returns an iterator over `k, vs`. Does not sort the input first.

disco.util.**msg**(*message*)
Deprecated since version 0.4: use **print** instead. Sends the string *message* to the master for logging. The message is shown on the web interface. To prevent a rogue job from overwhelming the master, the maximum *message* size is set to 255 characters and job is allowed to send at most 10 messages per second.

disco.util.**parse_dir**(*dir*, *partition=None*)
Translates a directory URL (`dir://...`) to a list of normal URLs.

This function might be useful for other programs that need to parse results returned by `disco.core.Disco.wait()`, for instance.

Parameters **dir** – a directory url, such as `dir://nx02/test_simple@12243344`

4.1.9 `disco.worker` – Python Worker Interface

In Disco, *workers* do the brunt of the data processing work. When a `disco.job.Job` is created, it gets passed a `Worker` instance, which is responsible for defining the fields used by the `disco.job.JobPack`. In most cases, you don't need to define your own Worker subclass in order to run a job. The Worker classes defined in `disco` will take care of the details of creating the fields necessary for the `disco.job.JobPack`, and when executed on the nodes, will handle the implementation of the *The Disco Worker Protocol*.

There is perhaps a subtle, but important, distinction between a *worker* and a `Worker`. The former refers to any binary that gets executed on the nodes, specified by `jobdict.worker`. The latter is a Python class, which handles details of submitting the job on the client side, as well as controlling the execution of user-defined code on the nodes. A `Worker` can be subclassed trivially to create a new *worker*, without having to worry about fulfilling many of the requirements for a well-behaving worker. In short, a `Worker` provides Python library support for a Disco *worker*. Those wishing to write a worker in a language besides Python may make use of the Worker class for submitting jobs to the master, but generally need to handle the *The Disco Worker Protocol* in the language used for the worker executable.

The `Classic Worker` is a subclass of `Worker`, which implements the classic Disco *mapreduce* interface.

The following steps illustrate the sequence of events for running a *job* using a standard `Worker`:

1. **(client) instantiate a `disco.job.Job`**
   (a) if a worker is supplied, use that worker
   (b) otherwise, create a worker using `disco.job.Job.Worker` (the default is `disco.worker.classic.worker.Worker`)
2. **(client) call `disco.job.Job.run()`**
   (a) create a `disco.job.JobPack` using: `Worker.jobdict()`, `Worker.jobenvs()`, `Worker.jobhome()`, `disco.task.jobdata()`

(b) submit the `disco.job.JobPack` to the master

3. (node) master unpacks the *job home*
4. (node) master executes the `jobdict.worker` with current working directory set to the *job home* and environment variables set from *Job Environment Variables*
5. (node) worker requests the `disco.task.Task` from the master
6. (node) worker runs the *task* and reports the output to the master

class `disco.worker.`**`Input`**(*input*, *task=None*, ***kwds*)
An iterable over one or more `Worker` inputs, which can gracefully handle corrupted replicas or otherwise failed inputs.

Parameters **open** (*function*) – a function with the following signature:

```
def open(url):
    ...
    return file
```

used to open input files.

class `disco.worker.`**`MergedInput`**(*input*, *task=None*, ***kwds*)
Produces an iterator over the minimal head elements of the inputs.

class `disco.worker.`**`Output`**(*path_type_partition*, *open=None*)
A container for outputs from `workers`.

Parameters **open** (*function*) – a function with the following signature:

```
def open(url):
    ...
    return file
```

used to open new output files.

`path`
The path to the underlying output file.

`type`
The type of output.

`partition`
The partition label for the output (or None).

`file`
The underlying output file handle.

class `disco.worker.`**`ParallelInput`**(*input*, *task=None*, ***kwds*)
Produces an iterator over the unordered records in a set of inputs.

Usually require the full set of inputs (i.e. will block with streaming).

class `disco.worker.`**`SerialInput`**(*input*, *task=None*, ***kwds*)
Produces an iterator over the records in a list of sequential inputs.

class `disco.worker.`**`Worker`**(***kwargs*)
A `Worker` is a `dict` subclass, with special methods defined for serializing itself, and possibly reinstantiating itself on the nodes where *tasks* are run.

The `Worker` base class defines the following parameters:

Parameters

- **map** (*function or None*) – called when the `Worker` is `run()` with a `disco.task.Task` in mode *map*. Also used by `jobdict()` to set `jobdict.map?`.
- **reduce** (*function or None*) – called when the `Worker` is `run()` with a `disco.task.Task` in mode *reduce*. Also used by `jobdict()` to set `jobdict.reduce?`.
- **save** (*bool*) – whether or not to save the output to *Disco Distributed Filesystem*.
- **profile** (*bool*) – determines whether `run()` will be profiled.

bin
The path to the *worker* binary, relative to the *job home*. Used to set `jobdict.worker` in `jobdict()`.

defaults()

Returns dict of default values for the `Worker`.

getitem(*key*, *job*, *jobargs*, *default=None*)
Resolves `key` in the following order: #. `jobargs` (parameters passed in during `disco.job.Job.run()`) #. `job` (attributes of the `disco.job.Job`) #. `self` (items in the `Worker` dict itself) #. `default`

input(*task*, *merged=False*, ***kwds*)

Parameters

- **task** (`disco.task.Task`) – the task for which to retrieve input.
- **merged** (*bool*) – if specified, returns a `MergedInput`.
- **kwds** (*dict*) – additional keyword arguments for the `Input`.

Returns an `Input` to iterate over the inputs from the master.

jobdict(*job*, ***jobargs*)
Creates *The Job Dict* for the `Worker`.

Makes use of the following parameters, in addition to those defined by the `Worker` itself:

Parameters

- **input** (*list of urls or list of list of urls*) – used to set `jobdict.input`. Disco natively handles the following url schemes:
  - `http://...` - any HTTP address
  - **`file://...` or no scheme - a local file.** The file must exist on all nodes where the tasks are run. Due to these restrictions, this form has only limited use.
  - `tag://...` - a tag stored in *Disco Distributed Filesystem*
  - `raw://...` - pseudo-address: use the address itself as data.
  - `dir://...` - used by Disco internally.
  - `disco://...` - used by Disco internally.

  See Also:

  `disco.schemes`.
- **name** (*string*) – directly sets `jobdict.prefix`.
- **owner** (*string*) – directly sets `jobdict.owner`. If not specified, uses `DISCO_JOB_OWNER`.
- **scheduler** (*dict*) – directly sets `jobdict.scheduler`.

Uses `getitem()` to resolve the values of parameters.

Returns the *job dict*.

jobenvs (*job*, ***jobargs*)

Returns *Job Environment Variables* dict.

jobhome (*job*, ***jobargs*)

Returns the *job home* (serialized).

Calls `jobzip()` to create the `disco.fileutils.DiscoZipFile`.

jobzip (*job*, ***jobargs*)
A hook provided by the `Worker` for creating the *job home* zip.

Returns a `disco.fileutils.DiscoZipFile`.

classmethod main ()
The main method used to bootstrap the `Worker` when it is being executed.

It is enough for the module to define:

```
if __name__ == '__main__':
    Worker.main()
```

Note: It is critical that subclasses check if they are executing in the `__main__` module, before running `main()`, as the worker module is also generally imported on the client side.

---

output (*task*, *partition=None*, ***kwds*)

Parameters

- **task** (`disco.task.Task`) – the task for which to create output.
- **partition** (*string or None*) – the label of the output partition to get.
- **kwds** (*dict*) – additional keyword arguments for the `Output`.

Returns the previously opened `Output` for *partition*, or if necessary, a newly opened one.

run (*task*, *job*, ***jobargs*)
Called to do the actual work of processing the `disco.task.Task`. This method runs in the Disco cluster, on a server that is executing one of the tasks in a job submitted by a client.

4.1.10 `disco.worker.classic` – Classic Disco Worker Interface

`disco.worker.classic.worker` – Classic Disco Runtime Environment

When a Job is constructed using the classic `Worker` defined in this module, Disco runs the `disco.worker.classic.worker` module for every job task. This module reconstructs the `Worker` on the node where it is run, in order to execute the *job functions* which were used to create it.

Classic Workers resolve all parameters using `getitem()`.

Thus, users can subclass `Job` as a convenient way to specify fixed parameters. For example, here's a simple distributed grep from the Disco `examples/` directory:

```
"""
An example using Job classes to implement grep.

Could be run as follows (assuming this module can be found on sys.path):

disco run grep.Grep -P params pattern [tag_url_or_path] ...
"""
from disco.job import Job
from disco.worker.classic.func import nop_map

class Grep(Job):
    map = staticmethod(nop_map)
    params = r''

    @staticmethod
    def map_reader(fd, size, url, params):
        import re
        pattern = re.compile(params)
        for line in fd:
            if pattern.match(line):
                yield url, line
```

class disco.worker.classic.worker.**Params**(***kwargs*)

Deprecated since version 0.4: `Params` objects aren't generally needed, since entire modules are sent with the `Worker`, state can be stored as in normal Python. Classic parameter container for map / reduce tasks.

This object provides a way to contain custom parameters, or state, in your tasks.

You can specify any number of `key, value` pairs to the `Params`. The pairs will be available to task functions through the *params* argument. Each task receives its own copy of the initial params object. *key* must be a valid Python identifier. *value* can be any Python object.

class disco.worker.classic.worker.**Worker**(***kwargs*)

A `disco.worker.Worker`, which additionally supports the following parameters, to maintain the **Classic Disco Interface**:

Note: The classic worker tries to guess which modules are needed automatically, for all of the *job functions* specified below, if the *required_modules* parameter is not specified. It sends any local dependencies (i.e. modules not included in the Python standard library) to nodes by default.

If guessing fails, or you have other requirements, see `disco.worker.classic.modutil` for options.

Parameters

- **map** (`disco.worker.classic.func.map()`) – a function that defines the map task.
- **map_init** (`disco.worker.classic.func.init()`) – initialization function for the map task. This function is called once before the task starts. Deprecated since version 0.4: *map_init* has not been needed ever since `InputStreams` were introduced. Use *map_input_stream* and/or *map_reader* instead.
- **map_input_stream** (sequence of `disco.worker.classic.func.input_stream()`) – The given functions are chained together and the final resulting `disco.worker.classic.func.InputStream` object is used to iterate over input entries. New in version 0.2.4.

- **map_output_stream** (sequence of `disco.worker.classic.func.output_stream()`) – The given functions are chained together and the `disco.worker.classic.func.OutputStream.add()` method of the last returned `disco.worker.classic.func.OutputStream` object is used to serialize key, value pairs output by the map. New in version 0.2.4.
- **map_reader** (`None` or `disco.worker.classic.func.input_stream()`) – Convenience function to define the last `disco.worker.classic.func.input_stream()` function in the *map_input_stream* chain.

  If you want to use outputs of an earlier job as inputs, use `disco.worker.classic.func.chain_reader()` as the *map_reader*. Changed in version 0.3.1: The default is `None`.
- **combiner** (`disco.worker.classic.func.combiner()`) – called after the partitioning function, for each partition.
- **reduce** (`disco.worker.classic.func.reduce()`) – If no reduce function is specified, the job will quit after the map phase has finished. New in version 0.3.1: Reduce now supports an alternative signature, `disco.worker.classic.func.reduce2()`, which uses an iterator instead of `out.add()` to output results.Changed in version 0.2: It is possible to define only *reduce* without *map*. See also *Do I always have to provide a function for map and reduce?*.
- **reduce_init** (`disco.worker.classic.func.init()`) –

  initialization function for the reduce task. This function is called once before the task starts.

  Deprecated since version 0.4: *reduce_init* has not been needed ever since `InputStreams` were introduced. Use *reduce_input_stream* and/or *reduce_reader* instead.
- **reduce_input_stream** (sequence of `disco.worker.classic.func.output_stream()`) – The given functions are chained together and the last returned `disco.worker.classic.func.InputStream` object is given to *reduce* as its first argument. New in version 0.2.4.
- **reduce_output_stream** (sequence of `disco.worker.classic.func.output_stream()`) – The given functions are chained together and the last returned `disco.worker.classic.func.OutputStream` object is given to *reduce* as its second argument. New in version 0.2.4.
- **reduce_reader** (`disco.worker.classic.func.input_stream()`) – Convenience function to define the last `disco.worker.classic.func.input_stream()` if *map* is specified. If *map* is not specified, you can read arbitrary inputs with this function, similar to *map_reader*.

  Default is `disco.worker.classic.func.chain_reader()`. New in version 0.2.
- **required_files** (*list of paths or dict*) – additional files that are required by the worker. Either a list of paths to files to include, or a dictionary which contains items of the form `(filename, filecontents)`. Changed in version 0.4: The worker includes *required_files* in `jobzip()`, so they are available relative to the working directory of the worker.
- **required_modules** (see *How to specify required modules*) – required modules to send with the worker.

- **merge_partitions** (*bool*) – whether or not to merge partitioned inputs during reduce.

  Default is `False`.

- **partition** (`disco.worker.classic.func.partition()`) – decides how the map output is distributed to reduce.

  Default is `disco.worker.classic.func.default_partition()`.

- **partitions** (*int or None*) – number of partitions, if any.

  Default is `1`.

- **sort** (*boolean*) – flag specifying whether the intermediate results, that is, input to the reduce function, should be sorted. Sorting is most useful in ensuring that the equal keys are consequent in the input for the reduce function.

  Other than ensuring that equal keys are grouped together, sorting ensures that keys are returned in the ascending order. No other assumptions should be made on the comparison function.

  The external program `sort` is used to sort the input on disk. In-memory sort can easily be performed by the tasks themselves.

  Default is `False`.

- **sort_buffer_size** (*string*) – how much memory can be used by external sort.

  Passed as the '-S' option to Unix *sort* (see *man sort*). Default is `10%` i.e. 10% of the total available memory.

- **params** (*object*) – object that is passed to worker tasks to store state The object is serialized using the *pickle* module, so it should be pickleable.

  A convenience class `Params` is provided that provides an easy way to encapsulate a set of parameters. `Params` allows including functions in the parameters.

- **ext_params** – if either map or reduce function is an external program, typically specified using `disco.util.external()`, this object is used to deliver parameters to the program.

  See `disco.worker.classic.external`.

- **status_interval** (*int*) – print "K items mapped / reduced" for every Nth item. Setting the value to 0 disables messages.

  Increase this value, or set it to zero, if you get "Message rate limit exceeded" error due to system messages. This might happen if your tasks are really fast. Decrease the value if you want more messages or you don't have that many data items.

  Default is `100000`.

`disco.worker.classic.worker.`**`get`**`(*args, **kwargs)`
See `disco.task.Task.get()`.

`disco.worker.classic.worker.`**`put`**`(*args, **kwargs)`
See `disco.task.Task.put()`.

`disco.worker.classic.worker.`**`this_host`**`()`
Returns hostname of the node that executes the current task.

`disco.worker.classic.worker.`**`this_inputs`**`()`
Returns the inputs for the *worker*.

disco.worker.classic.worker.**this_master**()
Returns hostname and port of the disco master.

disco.worker.classic.worker.**this_name**()
Returns the jobname for the current task.

disco.worker.classic.worker.**this_partition**()
For a map task, returns an integer between *[0..nr_maps]* that identifies the task. This value is mainly useful if you need to generate unique IDs in each map task. There are no guarantees about how ids are assigned for map tasks.

For a reduce task, returns an integer between *[0..partitions]* that identifies this partition. You can use a custom partitioning function to assign key-value pairs to a particular partition.

disco.worker.classic.func — Functions for constructing Classic Disco jobs

A Classic Disco job is specified by one or more *job functions*. This module defines the interfaces for the job functions, some default values, as well as otherwise useful functions.

class disco.worker.classic.func.**InputStream**
A file-like object returned by the `map_input_stream` or `reduce_input_stream` chain of `input_stream()` functions. Used either to read bytes from the input source or to iterate through input entries.

read(*num_bytes=None*)
Reads at most *num_bytes* from the input source, or until EOF if *num_bytes* is not specified.

class disco.worker.classic.func.**OutputStream**
A file-like object returned by the `map_output_stream` or `reduce_output_stream` chain of `output_stream()` functions. Used to encode key, value pairs add write them to the underlying file object.

add(*key*, *value*)
Adds a key, value pair to the output stream.

close()
Close the output stream.

path
The path on the local filesystem (used only for saving output to DDFS).

write(*data*)
Deprecated since version 0.3. Writes *data* to the underlying file object.

disco.worker.classic.func.**chain_reader**(*stream*, *size*, *url*, *ignore_corrupt=False*)
Input stream for Disco's internal compression format.

disco.worker.classic.func.**combiner**(*key*, *value*, *buffer*, *done*, *params*)
Returns an iterator of `(key, value)` pairs or `None`.

Parameters

- **key** – key object emitted by the `map()`
- **value** – value object emitted by the `map()`
- **buffer** – an accumulator object (a dictionary), that combiner can use to save its state. The function must control the *buffer* size, to prevent it from consuming too much memory, by calling `buffer.clear()` after each block of results. Note that each partition (as determined by the key and `partition()`) gets its own buffer object.
- **done** – flag indicating if this is the last call with a given *buffer*

- **params** – the object specified by the *params* parameter

This function receives all output from the `map()` before it is saved to intermediate results. Only the output produced by this function is saved to the results.

After `map()` has consumed all input entries, combiner is called for the last time with the *done* flag set to `True`. This is the last opportunity for the combiner to return something.

`disco.worker.classic.func.`**`default_partition`**(*key*, *nr_partitions*, *params*)
Returns `hash(key) % nr_partitions`.

`disco.worker.classic.func.`**`disco_input_stream`**(*stream*, *size*, *url*, *ignore_corrupt=False*)
Input stream for Disco's internal compression format.

`disco.worker.classic.func.`**`disco_output_stream`**(*stream*, *partition*, *url*, *params*)
Output stream for Disco's internal compression format.

`disco.worker.classic.func.`**`gzip_line_reader`**(*fd*, *size*, *url*, *params*)
Yields as many lines from the gzipped fd as possible, prints exception if fails.

`disco.worker.classic.func.`**`gzip_reader`**(*fd*, *size*, *url*, *params*)
Wraps the input in a `gzip.GzipFile` object.

`disco.worker.classic.func.`**`init`**(*input_iter*, *params*)
Perform some task initialization.

Parameters **input_iter** – an iterator returned by a `reader()`

Typically this function is used to initialize some modules in the worker environment (e.g. `ctypes.cdll.LoadLibrary()`), to initialize some values in *params*, or to skip unneeded entries in the beginning of the input stream.

`disco.worker.classic.func.`**`input_stream`**(*stream*, *size*, *url*, *params*)

Parameters

- **stream** – `InputStream` object
- **size** – size of the input (may be `None`)
- **url** – url of the input

Returns a triplet (`InputStream`, size, url) that is passed to the next *input_stream* function in the chain. The last `disco.func.InputStream` object returned by the chain is used to iterate through input entries.

Using an `input_stream()` allows you to customize how input urls are opened.

Input streams are used for specifying the *map_input_stream*, *map_reader*, *reduce_input_stream*, and *reduce_reader* parameters for the `disco.worker.classic.worker.Worker`.

`disco.worker.classic.func.`**`make_range_partition`**(*min_val*, *max_val*)
Returns a new partitioning function that partitions keys in the range *[min_val:max_val]* into equal sized partitions.

The number of partitions is defined by the *partitions* parameter

`disco.worker.classic.func.`**`map`**(*entry*, *params*)
Returns an iterable of (key, value) pairs given an *entry*.

Parameters

- **entry** – entry coming from the input stream
- **params** – used to maintain state between calls to the map function.

For instance:

```
def fun_map(e, params):
    return [(w, 1) for w in e.split()]
```

This example takes a line of text as input in *e*, tokenizes it, and returns a list of words as the output.

The map task can also be an external program. For more information, see *disco.worker.classic.external - Classic Disco External Interface*.

disco.worker.classic.func.**map_input_stream**(*stream*, *size*, *url*, *params*)
An `input_stream()` which looks at the scheme of `url` and tries to import a function named `input_stream` from the module `disco.schemes.scheme_SCHEME`, where SCHEME is the parsed scheme. If no scheme is found in the url, `file` is used. The resulting input stream is then used.

disco.worker.classic.func.**map_output_stream**(*stream*, *partition*, *url*, *params*)
An `output_stream()` which returns a handle to a task output. The handle ensures that if a task fails, partially written data is ignored.

disco.worker.classic.func.**nop_map**(*entry*, *params*)
No-op map.

This function can be used to yield the results from the input stream.

disco.worker.classic.func.**nop_reduce**(*iter*, *out*, *params*)
No-op reduce.

This function can be used to combine results per partition from many map functions to a single result file per partition.

disco.worker.classic.func.**notifier**(*urls*)

Parameters **urls** (*url or list of urls*) – a list of urls gives replica locators.

disco.worker.classic.func.**old_netstr_reader**(*fd*, *size*, *fname*, *head=''*)
Reader for Disco's default/internal key-value format.

Reads output of a map / reduce job as the input for a new job. Specify this function as your `map_reader()` to use the output of a previous job as input to another job.

disco.worker.classic.func.**output_stream**(*stream*, *partition*, *url*, *params*)

Parameters

- **stream** – `OutputStream` object
- **partition** – partition id
- **url** – url of the input

Returns a pair (`OutputStream`, url) that is passed to the next *output_stream* function in the chain. The `OutputStream.add()` method of the last `OutputStream` object returned by the chain is used to output entries from map or reduce.

Using an `output_stream()` allows you to customize where and how output is stored. The default should almost always be used.

disco.worker.classic.func.**partition**(*key*, *nr_partitions*, *params*)
Returns an integer in `range(0, nr_partitions)`.

Parameters

- **key** – is a key object emitted by a task function
- **nr_partitions** – the number of partitions
- **params** – the object specified by the *params* parameter

disco.worker.classic.func.**re_reader**(*item_re_str*, *fd*, *size*, *fname*, *output_tail=False*, *read_buffer_size=8192*)

A map reader that uses an arbitrary regular expression to parse the input stream.

Parameters **item_re_str** – regular expression for matching input items

The reader works as follows:

1. X bytes is read from *fd* and appended to an internal buffer *buf*.
2. `m = regexp.match(buf)` is executed.
3. If *buf* produces a match, `m.groups()` is yielded, which contains an input entry for the map function. Step 2. is executed for the remaining part of *buf*. If no match is made, go to step 1.
4. If *fd* is exhausted before *size* bytes have been read, and *size* tests `True`, a `disco.error.DataError` is raised.
5. When *fd* is exhausted but *buf* contains unmatched bytes, two modes are available: If `output_tail=True`, the remaining *buf* is yielded as is. Otherwise, a message is sent that warns about trailing bytes. The remaining *buf* is discarded.

Note that `re_reader()` fails if the input streams contains unmatched bytes between matched entries. Make sure that your *item_re_str* is constructed so that it covers all bytes in the input stream.

`re_reader()` provides an easy way to construct parsers for textual input streams. For instance, the following reader produces full HTML documents as input entries:

```
def html_reader(fd, size, fname):
    for x in re_reader("<HTML>(.*?)</HTML>", fd, size, fname):
        yield x[0]
```

disco.worker.classic.func.**reduce**(*input_stream*, *output_stream*, *params*)

Takes three parameters, and adds reduced output to an output object.

Parameters

- **input_stream** – `InputStream` object that is used to iterate through input entries.
- **output_stream** – `OutputStream` object that is used to output results.
- **params** – the object specified by the *params* parameter

For instance:

```
def fun_reduce(iter, out, params):
    d = {}
    for k, v in iter:
        d[k] = d.get(k, 0) + 1
    for k, c in d.iteritems():
        out.add(k, c)
```

This example counts how many times each key appears.

The reduce task can also be an external program. For more information, see *disco.worker.classic.external - Classic Disco External Interface*.

disco.worker.classic.func.**reduce2**(*input_stream*, *params*)

Alternative reduce signature which takes 2 parameters.

Reduce functions with this signature should return an iterator of `key, value` pairs, which will be implicitly added to the `OutputStream`.

For instance:

```
def fun_reduce(iter, params):
    from disco.util import kvgroup
    for k, vs in kvgroup(sorted(iter)):
        yield k, sum(1 for v in vs)
```

This example counts the number of values for each key.

disco.worker.classic.func.**reduce_input_stream**(*stream*, *size*, *url*, *params*)
An `input_stream()` which looks at the scheme of `url` and tries to import a function named `input_stream` from the module `disco.schemes.scheme_SCHEME`, where SCHEME is the parsed scheme. If no scheme is found in the url, `file` is used. The resulting input stream is then used.

disco.worker.classic.func.**reduce_output_stream**(*stream*, *partition*, *url*, *params*)
An `output_stream()` which returns a handle to a task output. The handle ensures that if a task fails, partially written data is ignored.

disco.worker.classic.func.**sum_combiner**(*key*, *value*, *buf*, *done*, *params*)
Sums the values for each key.

This is a convenience function for performing a basic sum in the combiner.

disco.worker.classic.func.**sum_reduce**(*iter*, *params*)
Sums the values for each key.

This is a convenience function for performing a basic sum in the reduce.

disco.worker.classic.func.**task_input_stream**(*stream*, *size*, *url*, *params*)
An `input_stream()` which looks at the scheme of `url` and tries to import a function named `input_stream` from the module `disco.schemes.scheme_SCHEME`, where SCHEME is the parsed scheme. If no scheme is found in the url, `file` is used. The resulting input stream is then used.

disco.worker.classic.func.**task_output_stream**(*stream*, *partition*, *url*, *params*)
An `output_stream()` which returns a handle to a task output. The handle ensures that if a task fails, partially written data is ignored.

`disco.worker.classic.modutil` – Parse and find module dependencies

New in version 0.2.3. This module provides helper functions to be used with the `required_modules` parameter in `Worker`. These functions are needed when your job functions depend on external Python modules and the default value for `required_modules` does not suffice.

By default, Disco tries to find out which modules are required by job functions automatically. If the found modules are not included in the Python standard library or other package that is installed system-wide, it sends them to nodes so they can be used by the Disco worker process.

Sometimes Disco may fail to detect all required modules. In this case, you can override the default value either by providing a list of requirements manually, or by generating the list semi-automatically using the functions in this module.

How to specify required modules

The `required_modules` parameter accepts a list of module definitions. A module definition may be either a module name, e.g. `"PIL.Image"`, or a tuple that specifies both the module name and its path, e.g. `("mymodule", "lib/mymodule.py")`. In the former case, the `disco.worker.classic.worker.Worker` only imports the module, assuming that it has been previously installed to the node. In the latter case, Disco sends the module file to nodes before importing it, and no pre-installation is required.

For instance, the following is a valid list for `required_modules`:

```
required_modules = ["math", "random", ("mymodule", "lib/mymodule.py")]
```

This expression imports the standard modules `math` and `random` and sends a custom module `lib/mymodule.py` to nodes before importing it.

Note that Disco sends only the files that can be found in your *PYTHONPATH*. It is assumed that files outside `PYTHONPATH` belong either to the Python standard library or to another package that is installed system-wide. Make sure that all modules that require automatic distribution can be found in your *PYTHONPATH*.

Automatic distribution works only for individual modules and not for packages nor modules that require a specific directory hierarchy. You need to install packages and modules with special requirements manually to your nodes.

Typical use cases

The following list describes some typical use cases for `required_modules`. The list helps you decide when to use the `find_modules()` and `locate_modules()` functions.

- (**default**) If you want to find and send all required modules used by your job functions recursively (i.e. also modules that depend on other modules are included), you don't need to specify `required_modules` at all. This equals to:

  ```
  required_modules = modutil.find_modules(job_functions)
  ```

  where *job_functions* is a list of all job functions: *map*, *map_init*, *combiner* etc.

- If you want to find and import all required modules, but not send them, or you want to disable recursive analysis, use `find_modules()` explicitly with the `send_modules` and `recursive` parameters.
- If you want to send a known set of modules (possible recursively) but you don't know their paths, use `locate_modules()`.
- If you want to send a known set of modules. provide a list of *(module name, module path)* tuples.
- If you just want to import specific modules, or sub-modules in a pre-installed package (e.g. `PIL.Image`), provide a list of module names.

Any combinations of the above are allowed. For instance:

```
required_modules = find_modules([fun_map]) + [("mymodule", "/tmp/mymodule.py"), "time", "random"]
```

is a valid expression.

Functions

exception `disco.worker.classic.modutil.`**`ModUtilImportError`**(*error*, *function*)
Error raised when a module can't be found by `disco.worker.classic.modutil`.

`disco.worker.classic.modutil.`**`find_modules`**(*functions*, *send_modules=True*, *recurse=True*, *exclude=()*)
Tries to guess and locate modules that are used by *functions*. Returns a list of required modules as specified in *How to specify required modules*.

Parameters

- **functions** – The functions to search for required modules.

- **send_modules** – If `True`, a `(module name, module path)` tuple is returned for each required local module. If `False`, only the module name is returned and detected modules are not sent to nodes; this implies `recurse=False`.
- **recurse** – If `True`, this function includes all modules that are required by *functions* or any other included modules. In other words, it tries to ensure that all module files required by the job are included. If `False`, only modules that are directly used by *functions* are included.

`disco.worker.classic.modutil.`**`locate_modules`**(*modules*, *recurse=True*, *include_sys=False*)

Finds module files corresponding to the module names specified in the list *modules*.

Parameters

- **modules** – The modules to search for other required modules.
- **recurse** – If `True`, recursively search for local modules that are used in *modules*.

A module is local if it can be found in your `PYTHONPATH`. For modules that can be found under system-wide default paths (e.g. `/usr/lib/python`), just the module name is returned without the corresponding path, so system-wide modules are not distributed to nodes unnecessarily.

This function is used by `find_modules()` to locate modules used by the specified functions.

`disco.worker.classic.modutil.`**`parse_function`**(*function*)

Tries to guess which modules are used by *function*. Returns a list of module names.

This function is used by `find_modules()` to parse modules used by a function. You can use it to check that all modules used by your functions are detected correctly.

The current heuristic requires that modules are accessed using the dot notation directly, e.g. `random.uniform(1, 10)`. For instance, required modules are not detected correctly in the following snippet:

```
a = random
a.uniform(1, 10)
```

Also, modules used in generator expressions, like here:

```
return ((k, base64.encodestring(v)) for k, v in d.iteritems())
```

are not detected correctly.

`disco.worker.classic.external` - Classic Disco External Interface

Note: Since Disco 0.4, you can write workers in any language without any dependencies to Python, using *The Disco Worker Protocol*. Use Disco External Interface if primarily you want to use Python with only parts of your job written in another language.

An external interface for specifying map and reduce functions as external programs, instead of Python functions. This feature is useful if you have already an existing program or a library which could be useful for a Disco job, or your map / reduce *task* is severely CPU or memory-bound and implementing it, say, in C, would remedy the problem.

Note that this external interface is not suitable for speeding up jobs that are mostly IO bound, or slowed down due to overhead caused by Disco. Actually, since the external interface uses the standard input and output for communicating with the process, the overhead caused by Disco is likely to increase when using the external interface. However, if the task is CPU or memory-bound, the additional communication overhead is probably minimal compared to gained benefits.

Easy approach using the *ctypes* module

In many cases there is an easier alternative to the external interface: You can write the CPU-intensive functions in C and compile them to a shared library which can be included in the *required_files* list of `disco.core.Disco.new_job()`. Here is an example:

```
def fast_map(e, params):
        return [("", params.mylib.fast_function(e))]

def map_init(iter, params):
        ctypes.cdll.LoadLibrary("mylib.so")
        params.mylib = ctypes.CDLL("mylib.so")

Disco("disco://discomaster").new_job(
        name = "mylib_job",
        input = ["http://someinput"],
        map = fast_map,
        map_init = map_init,
        required_files = ["mylib.so"],
        required_modules = ["ctypes"])
```

If this approach works for you, there is no need to read this document further. For more information, see documentation of the ctypes module.

External interface

The external program reads key-value pairs from the standard input and outputs key-value pairs to the standard output. In addition, the program may read parameters from the standard input when the task starts, and it may output log messages to the standard error stream. This interface should be easy to implement in any programming language, although C is used in examples below.

The key-value pairs are both read and written in the following format:

```
<key-size><key><value-size><value>
```

Here *key-size* and *value-size* are 32-bit integers, encoded in little-endian, which specify the sizes of the key and the value in bytes. *key* and *value* correspond to the key and the value strings.

For instance, the following C function reads a key-value pair from the standard input:

```
void read_kv(char **key, char **val)
{
        unsigned int len;
        *key = *val = NULL;
        /* read key */
        if (!fread(&len, 4, 1, stdin))
                return 0;
        if (len){
                *key = malloc(len);
                fread(*key, len, 1, stdin);
        }
        /* read value */
        fread(&len, 4, 1, stdin);
        if (len){
                *val = malloc(len);
```

```
                fread(*val, len, 1, stdin);
        }
        return 1;
}
```

Outputting a key-value pair works correspondingly using *fwrite()*. Using the function defined above, one can iterate through all input pairs as follows:

```
char *key, *val;
while (read_kv(&key, &val)){
        /* do something with key and value */
        free(key);
        free(val);
}
```

The external program must read key-value pairs from the standard input as long as there is data available. The program must not exit before all the input is consumed.

Note that extra care must be taken with buffering of the standard output, so that the output pairs are actually sent to the receiving program, and not kept in an internal buffer. Call *fflush(stdout)* if unsure.

External program is started with one command line argument: "map" or "reduce". This makes it possible to use a single binary to handle both map and reduce by using the command line argument to choose the function it should execute.

Map and reduce tasks follow slightly different interfaces, as specified below.

External map An external map task must read a key-value pair from *stdin* as specified above, and before reading the next pair, output a result list which may be empty. The output list is defined as follows:

```
<num-pairs>[<pair_0>...<pair_{num_pairs}>]
```

where *num-pairs* is a 32-bit integer, which may be zero. It is followed by exactly *num-pairs* consequent key-value pairs as defined above.

Inputs for the external map are read using the *map_reader*. The map reader may produce each input entry as a single string that is used as the in a key-value pair where the key is an empty string. Alternatively, the reader may return a pair of strings as a tuple, in which case both the key and the value are specified.

The map finishes when the result list for the final key-value pair is received.

External reduce In contrast to the external map, the external reduce is not required to match each input with a result list. Instead, the external reduce may output a result list, as specified above, any time it wants, also after all the inputs have been exhausted. As an extreme case, it may not produce any output at all.

The reduce finishes when the program exits.

Logging When outputting messages to the standard error, the following format must be used

```
void msg(const char *msg){
     fprintf(stderr, "**<MSG> %s\n", msg);
}

void die(const char *msg){
     fprintf(stderr, "**<ERR> %s\n", msg);
```

```
    exit(1);
}
```

Each line must have the first seven bytes as defined above, and the line must end with a newline character. The *msg()* function above is subject to the same limits as the standard `disco_worker.msg()` message function.

Parameters Any parameters for the external program must be specified in the *ext_params* parameter for `disco.core.Job()`. If *ext_params* is specified as a string, Disco will provide it as is for the external program in the standard input, before any key-value pairs. It is on the responsibility of the external program to read all bytes that belong to the parameter set before starting to receive key-value pairs.

As a special case, the standard C interface for Disco, as specified below, accepts a dictionary of string-string pairs as *ext_params*. The dictionary is then encoded by `disco.core.Job()` using the `disco.worker.classic.netstring` module. The *netstring* format is extremely simple, consisting of consequent key-value pairs. An example how to parse parameters in this case can be found in the `read_parameters()` function in *ext/disco.c*.

Usage

An external task consists of a single executable main program and an arbitrary number of supporting files. All the files are written to a single flat directory on the target node, so the program must be prepared to access any supporting files on its current working directory, including any libraries it needs.

Any special settings, or environment variables, that the program needs to be set can be usually arranged by a separate shell script that prepares the environment before running the actual executable. In that case your main program will be the shell script, and the actual executable one of the supporting files.

An external program absolutely must not read any files besides the ones included in its supporting files. It must not write to any files on its host, to ensure integrity of the runtime environment.

An external map or reduce task is specified by giving a dictionary, instead of a function, as the *fun_map* or *reduce* parameter in `disco.core.Job()`. The dictionary contains at least a single key-value pair where key is the string *"op"* and the value the actual executable code. Here's an example:

```
disco.job("disco://localhost:5000",
          ["disco://localhost/myjob/file1"],
          fun_map = {"op": open("bin/external_map").read(),
                     "config.txt": open("bin/config.txt").read()})
```

The dictionary may contain other keys as well, which correspond to the file names (not paths) of the supporting files, such as *"config.txt"* above. The corresponding values must contain the contents of the supporting files as strings.

A convenience function `disco.util.external()` is provided for constructing the dictionary that specifies an external task. Here's the same example as above but using `disco.util.external()`:

```
disco.job("disco://localhost:5000",
          ["disco://localhost/myjob/file1"],
          fun_map = disco.external(["bin/external_map", "bin/config.txt"]))
```

Note that the first file in the list must be the actual executable. The rest of the paths may point at the supporting files in an arbitrary order.

Disco C library

Disco comes with a tiny C file, *ext/disco.c* and a header, *ext/disco.h* which wrap the external interface behind a few simple functions. The library takes care of allocating memory for incoming key-value pairs, without doing malloc-free for each pair separately. It also takes care of reading a parameter dictionary to a Judy array which is like a dictionary object for C.

Here's a simple external map program that echoes back each key-value pair, illustriating usage of the library.

```
#include <disco.h>

int main(int argc, char **argv)
{
        const Pvoid_t params = read_parameters();
        Word_t *ptr;
        JSLG(ptr, params, "some parameter");
        if (!ptr)
                die("parameter missing");

        p_entry *key = NULL;
        p_entry *val = NULL;

        int i = 0;
        while (read_kv(&key, &val)){
                if (!(i++ % 10000))
                        msg("Got key <%s> val <%s>", key->data, val->data);
                write_num_prefix(1);
                write_kv(key, val);
        }
        msg("%d key-value pairs read ok", i);
        return 0;
}
```

The following functions are available in the library

Pvoid_t **read_parameters**()
: This function must be called before any call to the function `read_kv()`. It returns the parameter dictionary as a Judy array of type *JudySL*. See JudySL man page for more information.

void **die**(const char **msg*)
: Kills the job with the message *msg*.

int **read_kv**(p_entry ***key*, p_entry ***val*)
: Reads a key-value pair from the standard input. `read_kv()` can re-use *key* and *value* across many calls, so there is no need to *free()* them explicitely. If you need to save a key-value pair on some iteration, use `copy_entry()` to make a copy of the desired entry. Naturally you are responsible for freeing any copy that isn't needed anymore, unless you re-use it as a `copy_entry()` destination. To summarize, you need to call *free()* for entries that won't be re-used in a `copy_entry()` or `read_kv()` call.

  Returns key and value strings in `p_entry` structs.

  p_entry
  : Container type for a string.

  p_entry.len
  : Length of the string

  p_entry.sze
  : Size of the allocated buffer. Always holds *len* <= *sze*.

p_entry.data
Actual string of the size *len*, ending with an additional zero byte.

void **write_num_prefix**(int *num*)
Writes the *num_pairs* prefix for the result list as defined above. This call must be followed by *num* `write_kv()` calls.

void **write_kv**(const p_entry **key*, const p_entry **val*)
Writes a key-value pair to the standard output. Must be preceded with a `write_num_prefix()` call.

In addition, the library contains the following utility functions:

void ***dxmalloc**(unsigned int *size*)
Tries to allocate *size* bytes. Exits with `die()` if allocation fails.

void **copy_entry**(p_entry ***dst*, const p_entry **src*)
Copies *src* to *dst*. Grows *dst* if needed, or allocates a new `p_entry` if *dst* = *NULL*.

`disco.worker.classic.external.`**package**(*files*)
Packages an external program, together with other files it depends on, to be used either as a map or reduce function.

Parameters **files** – a list of paths to files so that the first file points at the actual executable.

This example shows how to use an external program, *cmap* that needs a configuration file *cmap.conf*, as the map function:

```
disco.new_job(input=["disco://localhost/myjob/file1"],
              fun_map=disco.util.external(["/home/john/bin/cmap",
                                           "/home/john/cmap.conf"]))
```

All files listed in *files* are copied to the same directory so any file hierarchy is lost between the files.

4.2 `disco` – Disco command line utility

disco is a fully-Python startup/configuration script which supports several exciting features. The new startup script makes it even easier to get up and running with a Disco cluster.

Note: This is the manpage for the **disco** command. Please see *Setting up Disco* for more information on installing Disco.

Hint: The documentation assumes that the executable `$DISCO_HOME/bin/disco` is on your system path. If it is not on your path, you can add it:

```
ln -s $DISCO_HOME/bin/disco /usr/local/bin
```

If `/usr/local/bin` is not in your `$PATH`, use an appropriate replacement. Doing so allows you to simply call **disco**, instead of specifying the complete path.

Run **disco help** for information on using the command line utility.

See Also:

The `ddfs` command.

See `disco.settings` for information about Disco settings.

4.2.1 Job History

For commands which take a jobname, or which support `-j`, the special arguments `@` and `@?<string>` are replaced by the most recent job name and the most recent job with name matching `<string>`, respectively.

For example:

```
disco results @
```

Would get the results for the most recent job, and:

```
disco results @?WordCount
```

Would get the results for the last job with name containing `WordCount`.

4.3 `ddfs` – DDFS command line utility

ddfs is a tool for manipulating data stored in *Disco Distributed Filesystem*. Some of the **ddfs** utilities also work with data stored in Disco's temporary filesystem.

Note: This is the manpage for the **ddfs** command. Please see *Disco Distributed Filesystem* for more general information on DDFS.

Hint: The documentation assumes that the executable `$DISCO_HOME/bin/ddfs` is on your system path. If it is not on your path, you can add it:

```
ln -s $DISCO_HOME/bin/ddfs /usr/local/bin
```

If `/usr/local/bin` is not in your `$PATH`, use an appropriate replacement. Doing so allows you to simply call **ddfs**, instead of specifying the complete path.

Run **ddfs help** for information on using the command line utility.

See Also:

The `disco` command.

See `disco.settings` for information about Disco settings.

Python Module Index

d