Mercurial's Query Languages

Martin Geisler
⟨`mg@aragost.com`⟩

Gearconf, Düsseldorf
June 10th, 2011

Outline

Introduction

Revision Sets
- Predicates
- Functions
- Operators

File Sets
- Working Copy Status and Path
- File Content

Conclusion

Outline

Introduction

Revision Sets
- Predicates
- Functions
- Operators

File Sets
- Working Copy Status and Path
- File Content

Conclusion

Confusing Histories

Big projects can give rise to a branchy history:

- several concurrent branches
- many developers pushing changes

Confusing Histories

Big projects can give rise to a branchy history:

- several concurrent branches
- many developers pushing changes

Mercurial help you to cut away the unnecessary fluff:

- Revision sets selects revisions (Mercurial 1.6):

```
$ hg log -r "branch('stable') and user('Martin')"
```

  Can be used in all places where Mercurial expects revisions

Confusing Histories

Big projects can give rise to a branchy history:

- several concurrent branches
- many developers pushing changes

Mercurial help you to cut away the unnecessary fluff:

- Revision sets selects revisions (Mercurial 1.6):

```
$ hg log -r "branch('stable') and user('Martin')"
```

  Can be used in all places where Mercurial expects revisions

- File sets selects files in revisions (Mercurial 1.9 or 2.0):

```
$ hg revert "set:added() and size('>20MB')"
```

  Can be used in all places where Mercurial expects file names

Flexibility

The query languages lets you solve hard problems:

- Imagine you have a dirty working copy:

```
$ hg status
M index.html
A logo.png
```

But how can you see the diff of `index.html` only?

Flexibility

The query languages lets you solve hard problems:

- Imagine you have a dirty working copy:

```
$ hg status
M index.html
A logo.png
```

But how can you see the diff of `index.html` only?

- Easy! You use your nifty Unix shell:

```
$ hg diff $(hg status --no-status --modified)
```

Flexibility

The query languages lets you solve hard problems:

- Imagine you have a dirty working copy:

```
$ hg status
M index.html
A logo.png
```

  But how can you see the diff of `index.html` only?

- Easy! You use your nifty Unix shell:

```
$ hg diff $(hg status --no-status --modified)
```

- With file sets you can do

```
$ hg diff "set:modified()"
```

  and it will work on all platforms

Implementation

When a revision set is evaluated it is:

tokenized: split input into operators, symbols, strings

Implementation

When a revision set is evaluated it is:

tokenized:	split input into operators, symbols, strings
parsed:	build parse tree based on operator precedence

Implementation

When a revision set is evaluated it is:

tokenized:	split input into operators, symbols, strings
parsed:	build parse tree based on operator precedence
optimized:	reorders parse tree to evaluate cheap parts first:

```
contains("README") and 1.0::1.5
```

starts with a manifest-based query — reorder to:

```
1.0::1.5 and contains("README")
```

Implementation

When a revision set is evaluated it is:

tokenized:	split input into operators, symbols, strings
parsed:	build parse tree based on operator precedence
optimized:	reorders parse tree to evaluate cheap parts first:

```
contains("README") and 1.0::1.5
```

starts with a manifest-based query — reorder to:

```
1.0::1.5 and contains("README")
```

executed: go through tree and evaluate predicates

Quoting

How to handle special characters:

- You will need to quote your queries on the command line:

```
$ hg log -r parents()
zsh: parse error near ‘()’
```

Quoting

How to handle special characters:

- You will need to quote your queries on the command line:

```
$ hg log -r parents()
zsh: parse error near ‘()’
```

- Strings in queries can be in single- or double-quotes:

```
$ hg log -r "user('Martin')"
```

Quoting

How to handle special characters:

- You will need to quote your queries on the command line:

```
$ hg log -r parents()
zsh: parse error near ‘()’
```

- Strings in queries can be in single- or double-quotes:

```
$ hg log -r "user('Martin')"
```

- Escape characters are supported

```
$ hg log -r "keyword('first line\nsecond line')"
```

Quoting

How to handle special characters:

- You will need to quote your queries on the command line:

```
$ hg log -r parents()
zsh: parse error near ‘()’
```

- Strings in queries can be in single- or double-quotes:

```
$ hg log -r "user('Martin')"
```

- Escape characters are supported

```
$ hg log -r "keyword('first line\nsecond line')"
```

- Use a raw string to disable the escape characters:

```
$ hg log -r "grep(r'Bug\s*\d+')"
```

Outline

Introduction

Revision Sets
- Predicates
- Functions
- Operators

File Sets
- Working Copy Status and Path
- File Content

Conclusion

Outline

Introduction

Revision Sets
- Predicates
- Functions
- Operators

File Sets
- Working Copy Status and Path
- File Content

Conclusion

Predicates

Predicates select changesets for inclusion in the resulting set:

- `closed()`, `head()`, `merge()`: simple changeset properties
- `author(string)`, `date(interval)`: search by user name or by commit date

```
$ hg log -r "author('Martin') and merge()"
```

- `grep(regex)`, `keyword(string)`: search in commit message, user name, changed file names for a regular expression or a substring

Matching by Files in Changesets

Matching by how a file changed:

- `adds(pattern)`: a file matching pattern was added
- `modifies(pattern)`: a file matching pattern was modified
- `removes(pattern)`: a file matching pattern was removed

Matching by Files in Changesets

Matching by how a file changed:

- `adds(pattern)`: a file matching pattern was added
- `modifies(pattern)`: a file matching pattern was modified
- `removes(pattern)`: a file matching pattern was removed
- `file(pattern)`: combination of all the above

Matching by Files in Changesets

Matching by how a file changed:

- `adds(pattern)`: a file matching pattern was added
- `modifies(pattern)`: a file matching pattern was modified
- `removes(pattern)`: a file matching pattern was removed
- `file(pattern)`: combination of all the above
- `contains(pattern)`: a file matching pattern was present

Outline

Introduction

Revision Sets
- Predicates
- Functions
- Operators

File Sets
- Working Copy Status and Path
- File Content

Conclusion

Following the Changeset Graph

A common task is to follow the graph from a particular changeset:

- `::set` or `ancestors(set)`: ancestors of changesets in set
- `set::` or `descendants(set)`: descendants of changesets in set
- `X::Y`: a combination of the above, finding changesets between X and Y

Following the Changeset Graph

A common task is to follow the graph from a particular changeset:

- `::set` or `ancestors(set)`: ancestors of changesets in set
- `set::` or `descendants(set)`: descendants of changesets in set
- `X::Y`: a combination of the above, finding changesets between X and Y

Changes that need to be merged into the default branch:

```
$ hg log -r "ancestors(stable) - ancestors(default)"
$ hg log -r "::stable - ::default"
```

Family Relations

- `ancestor(single, single)`: greatest common ancestor of the two changesets. Used to find out what needs to be merged in a merge between X and Y:

```
$ hg log -r "ancestor(X, Y)::Y"
```

- `children(set)`, `parents([set])`: set of all children/parents of set
- `heads(set)`, `roots(set)`: changesets from set with no children/parents in set

Parents and Grand Parents

Going from a changeset to the parent changeset is easy:

- `p1([set])`, `p2([set])`: the first/second parent of changesets in set or of the working copy if no set is given
- `x^`, `x^2`: the first/second parent of `x`
- `x~n`: the *n*'th first ancestor of `x`, `x~0` is `x`, `x~3` is `x^^^`

To see both sides of a merge changeset M use

```
$ hg diff -r "p1(M):M" && hg diff -r "p2(M):M"
```

or the shorter

```
$ hg diff -c M && hg diff -r "M^2:M"
```

The Next Push

The `hg outgoing` command tells what will be pushed, and so does this function:

- `outgoing([path])`: changesets not in the destination repository

The Next Push

The `hg outgoing` command tells what will be pushed, and so does this function:

- `outgoing([path])`: changesets not in the destination repository

It is now easy to see what you will push as a single diff:

```
$ hg diff -r "outgoing()"
```

The Next Push

The `hg outgoing` command tells what will be pushed, and so does this function:

- `outgoing([path])`: changesets not in the destination repository

It is now easy to see what you will push as a single diff:

```
$ hg diff -r "outgoing()"
```

It is also easy to reset a repository:

```
$ hg strip "outgoing()"
```

People familiar with Git will know this as

```
$ git reset --hard origin/master
```

Final Touches on Your Query

Trimming, cutting, manipulating the set:

- `max(set)`, `min(set)`: the changeset with minimum/maximum revision number in the set

Final Touches on Your Query

Trimming, cutting, manipulating the set:

- `max(set)`, `min(set)`: the changeset with minimum/maximum revision number in the set
- `reverse(set)`: the set is ordered; this reverses it

Final Touches on Your Query

Trimming, cutting, manipulating the set:

- `max(set)`, `min(set)`: the changeset with minimum/maximum revision number in the set
- `reverse(set)`: the set is ordered; this reverses it
- `limit(set, n)`, `last(set, n)`: the first/last *n* changesets

Final Touches on Your Query

Trimming, cutting, manipulating the set:

- `max(set)`, `min(set)`: the changeset with minimum/maximum revision number in the set
- `reverse(set)`: the set is ordered; this reverses it
- `limit(set, n)`, `last(set, n)`: the first/last *n* changesets
- `sort(set[, [-]key...])`: sorting the set by revision number, branch name, changeset message, user name, or date

Solving Ambiguities

When you do `hg log -r "foo"`, Mercurial checks

1. is `foo` a bookmark?
2. is `foo` a tag?
3. is `foo` a branch name?

First match wins.

Solving Ambiguities

When you do `hg log -r "foo"`, Mercurial checks

1. is `foo` a bookmark?
2. is `foo` a tag?
3. is `foo` a branch name?

First match wins.
You can override this using predicates:

- `bookmark([name])`, `tag([name])`: the changeset with the given bookmark or tag, or all bookmarked/tagged changesets
- `branch(name)`: changesets on the given branch
- `branch(set)`: changesets on the branches of the given set, normally used with a single changeset:

```
$ hg log -r "branch(tip)"
```

Outline

Introduction

Revision Sets
- Predicates
- Functions
- Operators

File Sets
- Working Copy Status and Path
- File Content

Conclusion

Operators

You can combine two revision sets using:

- `x and y` or `x & y`: changesets in both `x` and `y`
- `x or y` or `x | y` or `x + y`: changesets in either `x` or `y`
- `x - y`: changesets in `x` but not in `y`

Examples

- Heads on the current branch:

```
$ hg log -r "head() and branch(.)"
```

  Closed heads:

```
$ hg log -r "head() and closed()"
```

  Reopened branches:

```
$ hg log -r "closed() and not head()"
```

- Open heads on the current branch:

```
$ hg log -r "head() and branch(.) and not closed()"
```

- Bugfixes that are not in a tagged release:

```
$ hg log -r "keyword(bug) and not ::tagged()"
```

Outline

Introduction

Revision Sets
- Predicates
- Functions
- Operators

File Sets
- Working Copy Status and Path
- File Content

Conclusion

Selecting Files

File sets let you:

- select files from working copy
- select files from old revisions

Hopefully part of Mercurial 1.9 (July) or 2.0 (November)

Outline

Introduction

Revision Sets
- Predicates
- Functions
- Operators

File Sets
- Working Copy Status and Path
- File Content

Conclusion

Working Copy Status

The proposed predicates are:

- `modified()`, `added()`, `removed()`, `deleted()`, `unknown()`, `ignored()`, `clean()`: status flags

Working Copy Status

The proposed predicates are:

- `modified()`, `added()`, `removed()`, `deleted()`, `unknown()`, `ignored()`, `clean()`: status flags
- `copied()`: copied files, quite hard to extract today

Working Copy Status

The proposed predicates are:

- `modified()`, `added()`, `removed()`, `deleted()`, `unknown()`, `ignored()`, `clean()`: status flags
- `copied()`: copied files, quite hard to extract today
- `ignorable()`: tracked files that *would* be ignored

Working Copy Status

The proposed predicates are:

- `modified()`, `added()`, `removed()`, `deleted()`, `unknown()`, `ignored()`, `clean()`: status flags
- `copied()`: copied files, quite hard to extract today
- `ignorable()`: tracked files that *would* be ignored
- `tracked()`: all tracked files

Working Copy Status

The proposed predicates are:

- `modified()`, `added()`, `removed()`, `deleted()`, `unknown()`, `ignored()`, `clean()`: status flags
- `copied()`: copied files, quite hard to extract today
- `ignorable()`: tracked files that *would* be ignored
- `tracked()`: all tracked files
- `conflicted()`: like `hg resolve -list` after a merge

Searching by Path

We can replace the `find` Unix command:

- `glob(P)` instead of `find -path P`
- `regex(P)` instead of `find -regex P`

Remember that this also works on old revisions:

```
$ hg status -r 1.0::2.0 "set:glob(src/*.h)"
A src/foo.h
M src/bar.h
```

This shows that `foo.h` is a new header file in version 2.0.

File Type Predicates

Other `find`-like predicates will be:

- `executable()`, `symlink()`: file type
- `perm()`, `owner()`: file permissions
- `date()`, `size()`: other file meta data

Outline

Introduction

Revision Sets
- Predicates
- Functions
- Operators

File Sets
- Working Copy Status and Path
- File Content

Conclusion

Looking Into Files

Matching files by content:

- `grep()`: like the Unix `grep` we all love

Looking Into Files

Matching files by content:

- `grep()`: like the Unix `grep` we all love
- `contains()`: simple sub-string matching

Looking Into Files

Matching files by content:

- grep(): like the Unix grep we all love
- contains(): simple sub-string matching
- binary(), text(): does file contain a NUL byte?

```
$ hg add "set:unknown() and not binary()"
```

Looking Into Files

Matching files by content:

- grep(): like the Unix grep we all love
- contains(): simple sub-string matching
- binary(), text(): does file contain a NUL byte?

```
$ hg add "set:unknown() and not binary()"
```

- decodes(): check if file can be decoded with the given character set, such as UTF-8, UTF-16, . . .
  Lets you find mistakes:

```
$ hg status --all "set:glob('**.py') and not decodes('UTF-8')"
C src/foo.py
```

Looking Into Files

Matching files by content:

- `grep()`: like the Unix `grep` we all love
- `contains()`: simple sub-string matching
- `binary()`, `text()`: does file contain a NUL byte?

```
$ hg add "set:unknown() and not binary()"
```

- `decodes()`: check if file can be decoded with the given character set, such as UTF-8, UTF-16, . . .
  Lets you find mistakes:

```
$ hg status --all "set:glob('**.py') and not decodes('UTF-8')"
C src/foo.py
```

- `eol()`: line-ending type, Unix (LF) or DOS (CRLF)

Adding New Predicates

The feature will be extensible, some possible future extensions:

- `magic()`: recognize files based on file content, like the `file` program in Unix
- `locked()`: files locked for exclusive access by my `lock` extension

Outline

Introduction

Revision Sets
- Predicates
- Functions
- Operators

File Sets
- Working Copy Status and Path
- File Content

Conclusion

Conclusion

In short:

- revision sets lets you zoom in on the right part of the history
- file sets will let you pick out the relevant files
- both mechanisms are completely general

Conclusion

In short:

- revision sets lets you zoom in on the right part of the history
- file sets will let you pick out the relevant files
- both mechanisms are completely general

Please get in touch if you have more questions:

- Email: `mg@aragost.com`
- IRC: `mg` in `#mercurial` on `irc.freenode.net`

Conclusion

In short:

- revision sets lets you zoom in on the right part of the history
- file sets will let you pick out the relevant files
- both mechanisms are completely general

Please get in touch if you have more questions:

- Email: `mg@aragost.com`
- IRC: `mg` in `#mercurial` on `irc.freenode.net`

Thank you!