Today's Thursday Tutorial

Be(com)ing a MonetDB Developer

Craftsmen's Tools, Tips & Tricks for Genii

http://monetdb.cwi.nl/

Stefan.Manegold@cwi.nl

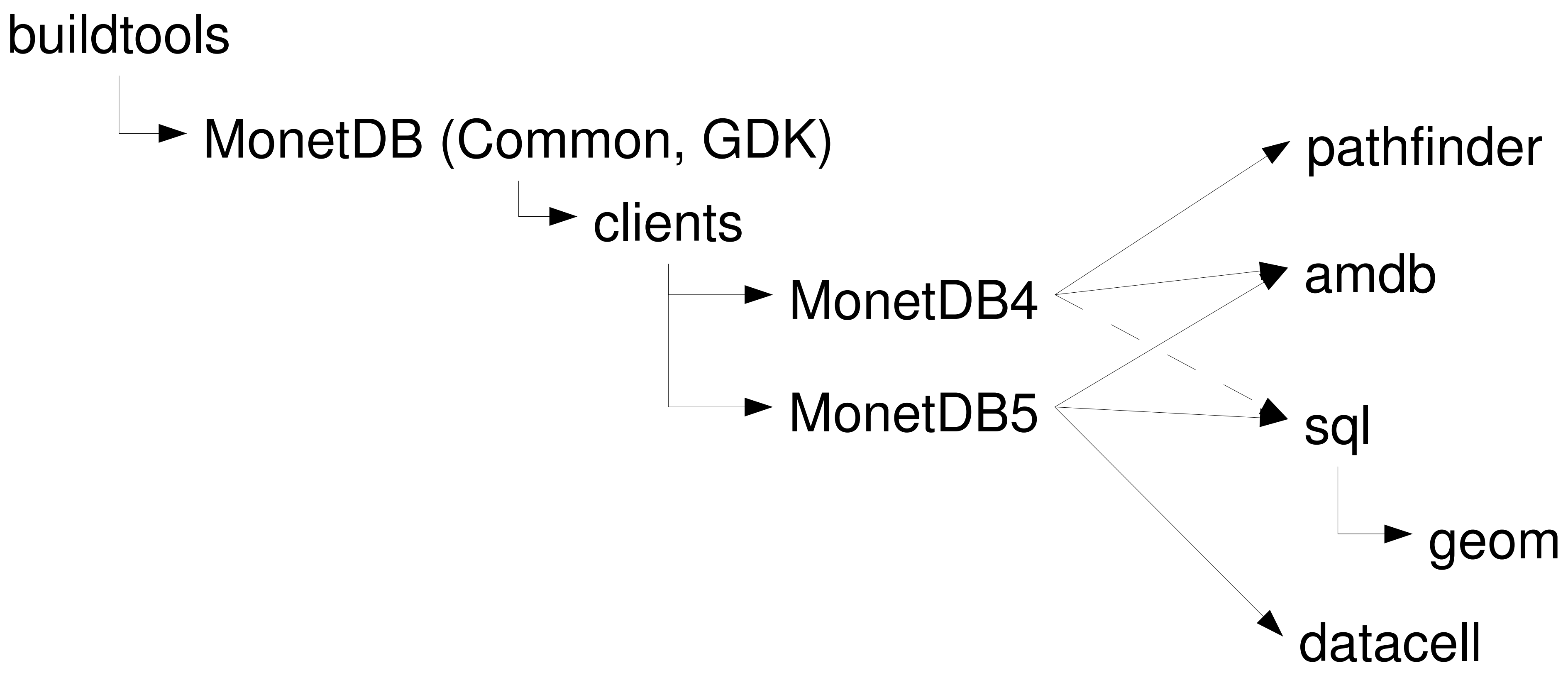
Topics

- Mailing lists
- Packages, dependencies, versions
- CVS
- Compilation
- Manual testing (Mtest.py)
- Automatic testing (TestWeb)

Mailing lists (@lists.sf.net)

- For *all* developers
  - monetdb-developers, monetdb-bugs, monetdb-*-checkins
- For *all* users (and developers)
  - monetdb-users
- For *all* users *and* developers
  - Monetdb-announce
- Details:
  - http://monetdb.cwi.nl/Development/MailChannels/
  - http://sourceforge.net/mail/?group_id=56967

Packages & Dependencies



Versions & Branches

	Release "Stable"		Development "Current"
	Version	CVS-branch	Version
Buildtools	1.20	MonetDB_1-20	1.21
MonetDB	1.20	MonetDB_1-20	1.21
Clients	1.20	Clients_1-20	1.21
MonetDB4	4.20	MonetDB_4-20	4.21
MonetDB5	5.2	MonetDB_5-2	5.3
Sql	2.20	SQL_2-20	2.21
Pathfinder	0.20	XQuery_0-20	0.21

Versions & Branches

- New features must be implemented, tested & checked-in on the development trunk (HEAD), only.
- Bug-fixes must be implemented, tested & checked-in on the release branches, only.
- Bug-fixes on the latest release branches are automa[tng]ically propagated (currently by Sjoerd or Stefan) to the HEAD on irregular basis (or on request).
- Bug-fixes on the latest release branches will eventually be release in bug-fix releases
- Bug-fixes on old (abandoned) release branches are neither propagated nor released (other than being in CVS).

What's new in *.20?

- Bug fixes
- `MapiClient` → `mclient` plus enhanced functionality, cf.
  - `mclient --help`
  - `mclient --language=<lang> --help`
- XQuery: automatic use of value indices (dynamic optimization)
- ...

`man cvs`: checkout & update

- `export CVS_RSH=ssh`
- Initial checkout

  ```
  cvs -d username@monetdb.cvs.sf.net:/cvsroot/monetdb
    co -P [-r branchname] modulename
  ```

- Updating your checkout (in `.../modulename/`)
  - To a branch: `cvs [-q] up -dP -r branchname`
  - To the trunk (`HEAD`): `cvs [-q] up -dP -A`
  - To whatever you have: `cvs [-q] up -dP`
  - Local changes are *not* lost during `cvs up` !
- Checking the status of your checkout: cvs [-q] status [-v]

`man cvs`: defaults & help

- For convenience, create `~/.cvsrc` with (e.g.)
  - `cvs -q`
  - `update -dP`
  - `diff -u`
  - `status -v`
- Handy:
  - `cvs --help`
  - `cvs --help-commands`
  - `cvs --help-synonyms`
  - `cvs -H commandname`
  - `man cvs`

`man cvs`: diff & commit

- Be(come) aware of what you changed *before* checking it in
  - `cvs diff`
- Gather (closely) related changes into a single `cvs commit` call, or at least use the identical check-in message
- Use separate `cvs commit` calls for unrelated changes
- Think of check-ins as transactions and recall ACID
- Check in early but not pre-mature
- If you checked-out/updated from a branch, your check-ins will automatically go to that branch
- Read your own check-in emails to double-check / verify your changes

\`man cvs\`: log messages

- Be(come) aware that some people indeed read check-in mails and want to understand / learn from them
- Be(come) aware that commit messages are not only read at time of check-in and in order / context of each other, but also individually as any later point in time as log messages (`cvs log`) when analyzing the (development of) the code
- Ideal: (self-contained) message describes reason, content and consequences of your changes
- Ok: message describes only reason and consequences; content is in the diff

`man cvs`: log messages

- Counter examples:
  - `<Empty log message>`
  - `“sorry”`
  - `“fixed bugs”`
  - `“did this at home and/or in the train”`
  - `“forgot this earlier”`
  - Messages that (obviously) don't match the changes

`man cvs`: accident recovery

- Undo accidental check-in of `MyPath/MyFile` (resulting in revision 1.23)

```
cvs up -j1.23 -j1.22 MyPath/MyFile
```

```
cvs diff -r1.22 MyPath/MyFile
```
-> no diffs!

```
cvs ci -m'undo of accidental check-in' MyPath/MyFile
```

- no/wrong log message with check-in of `MyPath/MyFile` (-> revision 1.23)

```
cvs admin -m1.23:'<correct log message>' MyPath/MyFile
```

`man cvs`: accident recovery

- Check-in to `HEAD` instead of branch (resulting in revision 1.23)
  - propagate to branch, but *leave* in `HEAD`

```
cvs up -r<branch> MyPath/MyFile

cvs up -j1.22 -j1.23 MyPath/MyFile

cvs diff MyPath/MyFile

cvs ci -m'
  back-ported bug-fix from HEAD:
  <original log message>
  identical check-in, expecting no conflicts during
  propagation
  ' MyPath/MyFile

cvs up -A MyPath/MyFile
```

- Do never copy/overwrite files checked-out from CVS!

`man cvs`: accident recovery

- Check-in to branch instead of `HEAD` (resulting in revision 1.23.4.5)
  - propagate to `HEAD` and *remove* from branch

```
cvs up -A MyPath/MyFile

cvs up -j1.23.4.4 -j1.23.4.5 MyPath/MyFile

cvs diff MyPath/MyFile

cvs ci -m'
  moving accidental check-in from <branch> to HEAD:
  <original log message>
  undo in <branch> follows immediately
  ' MyPath/MyFile

cvs up -r<branch> MyPath/MyFile

cvs up -j1.23.4.5 -j1.23.4.4 MyPath/MyFile

cvs diff -r1.23.4.4 MyPath/MyFile -> no diffs!

cvs ci -m'moved accidental change to HEAD' MyPath/MyFile
```

`man cvs`: Do & Don't

- Do never copy/overwrite files checked-out from CVS!
- Do never modify the files in `CVS/` (`Root`, `Repository`, `Entries`, `Tag`) by hand!
- Concentrate and be(come) aware of what you're doing!
- Be(come) aware that you're not playing all by yourself but in a team!
- Do never try to cover/hide your mistakes, but ask for help in case you get stuck or are in doubt!

Compilation: Environment

```
source=MySourcePath
build=MyBuildPath
prefix=MyInstallationPath


export PATH=”$prefix/bin:$PATH”
export PYTHONPATH="$prefix/`python -c 'import
  distutils.sysconfig; print
  distutils.sysconfig.get_python_lib(0,0,"")'`"
```

Compilation: Configure options

		Default		Recommended	
		Stable	Current	Dev/DBG	Exp/OPT
`--enable-strict`	=	No	Yes	Yes	Yes
`--enable-debug`	=	No	No	Yes	No
`--enable-optimize`	=	Yes	No	No	Yes
`--enable-assert`	=	No	Yes	Yes	No
`--enable-oid32`	=	No	No	?	?

See `configure --help` for more

```
conf='<desired combination of the above>'
```

- Default: `CFLAGS='-g -O2'`
- `--enable-debug`: `CFLAGS='-g'`
- `--enable-optimize`: `CFLAGS='-O6 ...'`

Compilation: all from scratch

```
for i in buildtools MonetDB clients MonetDB{4,5} ... ;
do
   ( cd $source/$i && \
     ./bootstrap && \
     mkdir -p $build/$i && cd $build/$i && \
     $source/$i/configure --prefix=$prefix $conf && \
     make && make install \
   ) || break;
done
```

Compilation: re-compile

- `configure` not required (`make [Makefile]` calls it if necessary)
- Code changes (incl. `*.in` files)
  - `cd $build/$i && make Makefile && make && make install`
- buildtools changes and/or `*.ag` file changes
  - `cd $source/$i && ./bootstrap`
  - `cd $build/$i && make Makefile && make && make install`

Compilation: clean-up

- `cd $build/$i && make uninstall`
  - `cd $build/$i && make install`
- `rm -r $prefix`
  - `cd $build/$i && make install`
- `cd $build/$i && make [dist]clean`
  - `cd $build/$i && make && make install`
- `rm -r $build/$i`
  - `mkdir -p $build/$i && cd $build/$i && $source/$i/configure --prefix=$prefix $conf && make && make install`
- `cd $source/$i && ./de-bootstrap`
  - `cd $source/$i && ./bootstrap && cd $build/$i && make && make install`

Compilation: multi-version setup

- Sources:
  - `.../Stable/source/*/`
  - `.../Current/source/*/`
- Builds:
  - `.../Stable/build.DBG/*/`
  - `.../Stable/build.OPT/*/`
  - `.../Current/build.DBG/*/`
  - `.../Current/build.OPT/*/`
- Prefixes:
  - `.../Stable/prefix.DBG/`
  - `.../Stable/prefix.OPT/`
  - `.../Current/prefix.DBG/`
  - `.../Stable/prefix.OPT/`

Background & Goals

- **Correctness**
- **Stability**
- **Portability**
- **Compatibility between MonetDB and its add-ons**

Contents

- "Testing" includes ("by hand" or "automatically"):
  - Single-platform compilation
  - Single-platform functionality testing ("Mtest.py")
  - Adding & maintaining tests and their "stable" output
  - Multi-platform compilation
  - Multi-platform functionality testing
  - Gathering & "aggregation" of multi-platform results
  - Checking/monitoring multi-platform results
  - Maintaining/extending/improving Mtest.py/Mapprove.py
  - Maintaining the "TestTools" for multi-platform testing

Single-platform compilation

- Bootstrap
- Configure
- Make
-
-
-
-
- Make install

Cetero Censeo code must compile successfully on at least one platform before being checked-in!

Single-platform compilation

• Bootstrap	• Bootstrap
• Configure	• Configure
• Make	• Make
•	• Make check
•	• Make html
•	• Make dist
•	• Make rpm
• Make install	• Make install

Single-platform compilation

- Bootstrap
- Configure
- Make
-
-
-
-
- Make install

- Bootstrap
- Configure
- Make
- Make check
- Make html
- Make dist
- Make rpm
- Make install

- Bootstrap
- Configure
- Make
- Make check
- Make doc
- Make dist
- Make rpm
- Make install
- Make install_doc

Cetero Censeo code must compile successfully on at least one platform before being checked-in!

Single-platform Functionality Testing

- `Mtest.py`: “handy” tool to run tests
  - Execute test
  - Collect output (stdout & stderr) in files
  - Filter output through `Mfilter.py` to mark-up known/expected variations (paths, time, date, ...)
  - Compare created output to stored “stable” (correct) output (`Mdiff`)
  - Mark-up difference (`Mdiff`)
  - Build web-page for all tests run
  - Provide user with single url as entry point

Cetero Censeo code must be tested successfully on at least one platform before being checked-in!

Single-platform Functionality Testing

- What are “tests”?
  - Arbitrary executable file `TST` (with 'x'-bit set): `TST >TST.test.out 2>TST.test.err`
  - MIL script: `TST.milS` / `TST.milC`
    `Mserver <TST.milS (Mserver&); MapiClient <TST.milC; Mshutdown`
  - SQL query: `TST.sql`
  - Xquery query: `TST.xq`
  - ...
  - See MonetDB/src/testing/README for details

Cetero Censeo code must be tested successfully on at least one platform before being checked-in!

Single-platform Functionality Testing

- What Mtest.py cannot do for you:
  - Create initial stable output for new tests
  - Verify correctness of output
  - Tell the origin/cause of differences
  - Tell, whether differences were expected/intended or rather indicate bugs
  - ...

Adding & maintaining tests and their “stable” output

- See demo
- Mapprove.py:
  - Install (“approve”) output of last Mtest.py run as new stable output
  - Removes error messages ('^!...') unless explicitely asked to keep them (“-f”)
  -
- Do always read Mapprove.py's output carefully!
- System-specific stable output

Nightly Multi-platform Testing

- /ufs/monet/repository/TestTools
  - Collection of bash scripts
- 1 “server”/”master”, several “clients”/”slaves”
- Server:
  - “cvs update” the code
  - Scp code (as tar-files) to clients
  - Start testing on clients via ssh
  - Collect results from clients
  - Build “TestWeb”
  - Send nasty mails

Pre-Check-In Tasks (Everybody!)

- make sure your code compiles (at least on your platform)
- run Mtest
- see Mtest output and generated webpage to check the impact of your code/test changes
- fix your code and/or fix/update the stable output where necessary
- re-run Mtest
- use “cvs diff” to verify what you are about to checkin
-
- [illegible]

Post-Check-In Tasks (Everybody!)

- check testing result mails to
  - see whether your changes did compile and test well on other platforms
  - your changes to MonetDB might have impact on any (known/tested) add-on
- check TestWeb for details
- fix code that does not compile/work properly
- add system-specific stable output where necessary

(Daily!??) Tasks for Project-Maintainers

- Check/monitor multi-platform results (mails & TestWeb)
- Notify developers of changes / bugs / problems due to their latest checkins
-