CoGaDB User Guide

Version 0.4.2

CoGaDB Development Team

cogadb@googlegroups.com

Prequisites

• Operating System: Ubuntu Version 12.04.3 LTS or higher (64 bit)

• Compiler: g++ version 4.6 or higher

Tools

• NVIDIA CUDA Toolkit 6.5 or higher

• cmake version 2.6 or higher

• make

• Doxygen (Documentation Generation)

• Pdf Latex (Documentation Generation)

• Bison (supported from version 2.5)

• Flex

• Mercurial (version control system)

• clang and LLVM (optional, if you want to use the query compiler)

Libraries

• Boost Libraries version 1.48 or higher

• Readline Library

• Intel TBB Library

• BAM Library (optional, required for Genomics Extension)

• XSLT Library (Documentation Generation)

• Google Sparse Hash

Hardware

• Intel CPU (optional, required for using Intel PCM Library)

• NVIDIA GPU (optional, required for GPU acceleration)

Source Code

After you got access to the repositories, you can checkout the source code using mercurial.

```
hg clone <your URL to gpudbms on Bitbucket>
hg clone https://YOURUSERNAME@bitbucket.org/bress/gpudbms
```

Note the structure of the repository. CoGaDB consists of three repositories:

1. The HyPE Library: The learning-based physical Optimizer

2. CoGaDB: The actual DBMS

3. GPUDBMS: A repository that combines the build systems of CoGaDB and HyPE, and also contains external libraries (e.g., Thrust or Intel PCM Libraries).

The reason of this separation is that the physical optimizer is organized as a library, and users of HyPE typically have their own database engine, and hence, do not want to checkout CoGaDB together with HyPE. After checkout, you need to switch to the development branches in CoGaDB and HyPE. Here are the commands:

```
cd cogadb
hg pull && hg update stable
cd ..
cd hype-library
hg pull && hg update stable
cd ..
```

Installation Steps

First you need to install all libraries specified in the perquisites section. In case you run an up-to-date Ubuntu, you can use the following commandline:

```
sudo apt-get install gcc g++ make cmake flex bison libtbb-dev libreadline6 libreadline6-dev doxygen
doxygen-gui graphviz texlive-bibtex-extra texlive-fonts-extra texlive-fonts-extra-doc ghostscript
texlive-fonts-recommended xsltproc libxslt1-dev nvidia-cuda-toolkit sharutils libsparsehash-dev
libbam-dev samtools clang
```

If you want to setup a system without documentation support, you can use the following commandline:

```
sudo apt-get install gcc g++ make cmake flex bison libtbb-dev libreadline6 libreadline6-dev nvidia-
cuda-toolkit libsparsehash-dev libbam-dev samtools clang
```

If you can also sacrifice GPU support, you do not have to install the CUDA library (ONLY works for BRANCHES development, cpp_query_compilation, genomics_default FOR NOW):

```
sudo apt-get install gcc g++ make cmake flex bison libtbb-dev libreadline6 libreadline6-dev
```

```
libsparsehash-dev libbam-dev samtools clang
```

Now only boost is missing. To install boost, you can use the following command line:

```
sudo apt-get install libboost-filesystem-dev libboost-system-dev libboost-thread-dev libboost-program-options-dev libboost-serialization-dev libboost-chrono-dev libboost-date-time-dev libboost-random-dev libboost-iostreams-dev
```

Now we are able to compile CoGaDB. Enter the main directory gpudbms in a terminal. We now have to generate a build system using cmake. It is considered good practice to separate the source tree from the separated build system. Therefore, we will generate the build system into a sub directory. Development is often easier if we have compiled the source code with debugging symbols and without optimization. On the other hand, we need to compile source with optimizations and without debugging symbols to achieve maximal performance. Therefore, we will generate two build systems. One builds CoGaDB in debugging mode, whereas the other builds CoGaDB in release mode. You can enter the following commands to achieve this:

```
mkdir debug_build
cd debug_build
cmake -DCMAKE_BUILD_TYPE=Debug ..
make
cd ..
mkdir release_build
cd release_build
cmake -DCMAKE_BUILD_TYPE=Release ..
make
cd ..
```

Now, debug_build contains the build system in debug mode, whereas the build system in release_build compiles CoGaDB in release mode. Just enter a directory, hit make, and launch using ./cogadb/bin/cogadbd. Since CoGaDB consists of many source files, you can speedup the build process by instructing make to compile multiple source files in parallel. You can do this by using the -j option of make, where you specify the number of parallel build processes, which is typically your number of CPU cores. That's it, you can now start developing CoGaDB!

Configuration of the CMake Build System Generator

You can influence the behavior of cmake by adjusting the Variables in the CMakeCache.txt. You can also set a variable using the commandline interface. Let us assume we want to change the build type of our build system from Debug to Release. The build type is stored in the variable CMAKE_BUILD_TYPE. Now, we can change the build type using the following command:

```
cmake -DCMAKE_BUILD_TYPE=Release ..
```

There are many variables that allow customization. We list here the most important ones:

```
ENABLE_GPU_ACCELERATION
ENABLE_SIMD_ACCELERATION
ENABLE_BRANCHING_SCAN
CMAKE_VERBOSE_MAKEFILE
```

```
CMAKE_CXX_FLAGS_RELEASE
CMAKE_CXX_FLAGS_DEBUG
```

Basics on Repository Management and Mercurial

The source code is managed by the decentralized version control system mercurial and is organized in branches. We have one branch for the current stable version of CoGaDB and HyPE. Both branches are named default. Then, there is one separate development branch in HyPE and CoGaDB. In CoGaDB, it is named parallel_gpu_kernels. In HyPE, it is named hybrid_query_optimization. By convention, it is forbidden to directly implement new features in stable and development branches. To start implementing new features, we need to checkout the newest version of the development branch, and create your own branch using the following commands:

```
hg update development
hg branch <branch name>
```

As a general rule, you should regularly merge changes from the development branch (Note: Typically, you will receive an Email, which explains new features and requests you to merge these changes in your branch.) You can do so using the following commandline: hg merge development Then, you need to resolve possible conflicts. Afterwards, you need to commit your merge step, and push it into the repository. You can do that using the following commands:

```
hg commit -m “merged changes from development branch into branch <YOUR BRANCH NAME>”
hg push
```

When you start working with mercurial, multiple people may change something in the same or different branch.
Thus, you should regularly pull those changes to stay up to date:

```
hg pull
```

However, we now only pulled the changes from the server, but we have yet to apply them to our source code:

```
hg update
```

In case the update fails, it means someone else made changes in your branch. Therefore, you have to merge those changes:

```
hg merge
```

In case they are conflicts, resolve them using your favorite merging tool. After a merging operation, you always need to commit your changes:

```
hg commit
```

If you want to upload your changes to the repository, you need to perform a push operation.

```
hg push
```

Implementing New Features

When we implement new features, we often create new source files, which we need to add to the build system and the source tree. Let us assume you add a file foo.cpp to CoGaDB's util directory:

```
cogadb/src/util/foo.cpp
```

Then, we need to tell mercurial it should keep track of the changes to these file by adding it to the source tree:

```
hg add cogadb/src/util/foo.cpp
```

Now, we have to tell cmake, that there is a new source file to compile. For this, we need to edit CoGaDB's CMakelists.txt file and add the source file to the list of source files for CoGaDB. This list starts with cuda_add_library and has the name cogadb. Now type make, and your source file is compiled and linked into CoGaDB. Note that you only need to add a file to the build system, if it is a translation unit, such as C (.c), C++ (.cpp), or CUDA (.cu) source files. You do NOT need to add header files (.h or .hpp) to the build system!

Bug Reports and Feature Requests

To keep track of found bugs and feature requests, we use the build-in issue tracking of Bitbucket: https://bitbucket.org/bress/cogadb/issues

When reporting a bug, it is important to provide the following information to ensure a timely reponse:

1.A short description of the error, including the expected behavior, and the observed behavior.

2.Compile CoGaDB in debug mode and provide a stack trace. You can provide a stacktrace using your favorite IDE or the commandline-based gdb: gdb ./cogadbd <Here comes the error> → type bt, to get a backtrace

3.Provide the commands and queries you gave CoGaDB. Typically, a user loads a database using the build-in scripting language of CoGaDB. The default script is startup.coga, which is executed on startup.

As for feature requests, they should be short and self-contained. Note that feature requests are implemented on a best-effort basis.

Setting up a Database and Issue Queries

At first, we have to create a directory, where CoGaDB can store its database:

```
set path_to_database=/home/DATA/coga_databases/ssb_sf1
```

Then, we have to create a database and import data. This can be done in two ways: using the sql interface (create table, insert into), or using a utility command. CoGaDB supports utility commands for importing databases from two common OLAP benchmarks: the TPC-H and the Star Schema Benchmark. Note that you have to generate the* .tbl files using the dbgen tool. Assuming we have generated a database for the star schema benchmark of scale factor one and stored the resulting *.tbl files in /home/DATA/benchmarks/star_schema_benchmark/SF1/, we can import the data with the following command:

```
create_ssb_database /home/DATA/benchmarks/star_schema_benchmark/SF1/
```

For the TPC-H benchmark, the command is create_tpch_database.

Now CoGaDB imports the data and stores them in the database. Depending on the scale factor, this can take a while. After the import finishes, we can start working with the database. Since CoGaDB is an in-memory database, we first have to load the database in the main memory:

```
loaddatabase
```

Then, we can start issuing queries. We can either use SQL or build-in aliases for stored queries. We provide stored queries for all queries of the star schema benchmark. The template command is ssbXY, which executes SSB-Query X.Y (X has to be a number between 1 and 4; Y has to be a number between 1 and 3 except when X is 3, in this case 4 is valid for Y as well). Now, we can launch queries:

```
CoGaDB>select sum(lo_extendedprice*lo_discount) as revenue from lineorder, dates where
lo_orderdate = d_datekey and d_weeknuminyear = 6 and d_year = 1994 and lo_discount between 5
and 7 and lo_quantity between 26 and 35;
+-------------+
| REVENUE     |
+=============+
| 2.49945e+10 |
+-------------+
1 rows
Execution Time: 155.28039 ms
```

You can get a complete list of supported script commands by typing help. You can issue any number of commands in the startup script startup.coga, to automatically load a database and set certain parameters.

Support

In case you have any questions or suggestions, please do not hesitate to contact the development team via the mailing list <cogadb@googlegroups.com> or Sebastian Breß <sebastian.bress@dfki.de>.