HyPE: A Hybrid Query Processing Engine for Co-Processing in Database Systems

User Manual

Version 0.2

Otto-von-Guericke-University Magdeburg
School of Computer Science
Department of Technical and Business Information Systems
Database and Information Systems Group

Authors:

Sebastian Breß

Robin Haberkorn

Steven Ladewig

Contents

Chapter 1

Documentation

1.1 Introduction

HyPE is a library build for automatic selection of processing units for co-processing in database systems. The long-term goal of the project is to implement a fully fledged query processing engine, which is able to automatically generate and optimize a hybrid CPU/GPU physical query plan from a logical query plan. It is a research prototype developed by the `Otto-von-Guericke University Magdeburg` in collaboration with `Ilmenau University of Technology`.

See the publications listed below for more details.

1.2 Features

Currently, HyPE supports the following features:

- Entirely written in C++
- Decides on the (likely) optimal algorithm w.r.t. to a user specified optimization criterion for an operation
- Unrestricted use in parallel applications due to thread-safety
- Easily extensible by utilizing a plug-in architecture
- **New:** Runs under Linux and Windows
- **New:** Supports the following compilers: g++ (>=4.5), clang, and Visual C++
- Requires (almost) no knowledge about executed algorithms, just the relevant features of the datasets for an algorithms execution time
- Collects statistical information to help the user to fine tune HyPE's parameters for their use case
- Provides a parallel execution engine for operators

1.3 Detailed Documentation

1. **Installation** (p. 7)
2. **Tutorial: How to use HyPE** (p. 11)
3. **Available Components** (p. 15)
4. **Configure HyPE** (p. 19)
5. **Extend HyPE** (p. 21)
6. **FAQ** (p. 25)

1.4 Project Members

1.4.1 Project Members:

- `Sebastian Breß` (University of Magdeburg)
- Klaus Baumann (University of Magdeburg)
- Robin Haberkorn (University of Magdeburg)
- Steven Ladewig (University of Magdeburg)
- Tobias Lauer (Jedox AG)
- Gunter Saake (University of Magdeburg)
- Norbert Siegmund (University of Magdeburg)

1.4.2 Project Partners:

- Felix Beier (Ilmenau University of Technology)
- Ladjel Bellatreche (LIAS/ISEA-ENSMA, Futuroscope, France)
- Hannes Rauhe (Ilmenau University of Technology)
- Kai-Uwe Sattler (Ilmenau University of Technology)

1.4.3 Former Project Members:

- Ingolf Geist (University of Magdeburg)

1.5 Publications

Bibliography

[1] S. Breß. Ein selbstlernendes Entscheidungsmodell für die Verteilung von Datenbankoperationen auf CPU/GPU-Systemen. Master thesis, University of Magdeburg, Germany, March 2012. In German.

[2] S. Breß, F. Beier, H. Rauhe, E. Schallehn, K.-U. Sattler, and G. Saake. Automatic Selection of Processing Units for Coprocessing in Databases. In *ADBIS*, pages 57–70. Springer, 2012.

[3] S. Breß, I. Geist, E. Schallehn, M. Mory, and G. Saake. A Framework for Cost based Optimization of Hybrid CPU/GPU Query Plans in Database Systems. *Control and Cybernetics*, 41(4), 2012. to appear.

[4] S. Breß, S. Mohammad, and E. Schallehn. Self-Tuning Distribution of DB-Operations on Hybrid CPU/GPU Platforms. In *GvD*, pages 89–94. CEUR-WS, 2012.

[5] S. Breß, E. Schallehn, and I. Geist. Towards Optimization of Hybrid CPU/GPU Query Plans in Database Systems. In *GID*, pages 27–35. Springer, 2012.

Chapter 2

Installation

HyPE uses `CMake` as a cross-platform build system. Generally, building HyPE is similar on all platforms but we will nevertheless highlight some platform-specifics on this page.

There might also be precompiled binaries for your platform and toolchain.

Note that C++ libraries for one platform (e.g. Windows) built with different toolchains or merely different toolchain versions are generally not interchangeable.

2.1 Prerequisites

HyPE depends on the following third-party libraries and tools:

- `boost::filesystem`, `boost::system`, `boost::thread`, `boost::program_options`, `boost::chrono`
- Headers of the "C++ Technical Report on Standard Library Extensions", or boost::tr1
- `Loki Library`
- `ALGLIB` (included, usually no need install separately)
- `Doxygen` (only if you would like to build the documentation)

2.2 Installing on Linux

HyPE has been tested with the following compilers/build-environments on Linux:

- Ubuntu 11.04 (32-bit)
- Ubuntu 12.04 (64-bit)
- gcc/g++ (v4.6.3)

To install all prerequisites on Ubuntu Linux, install the following packages:

```
sudo apt-get install gcc g++ make cmake doxygen graphviz libboost-all-dev
     build-essential gnuplot-x11 ffmpeg libloki-dev libloki*
```

Now you can configure the HyPE package. We advise you to do a CMake out-of-source-tree build, for instance using the following commands from a shell:

```
cd hype-library
mkdir build
cd build
cmake ../
```

If you encounter any errors during configuration, or want to tweak HyPE's build system you may want to run ‘ccmake‘ to manipulate CMake's cache and re-generate the build-system. The appropriate options are documented.

To build HyPE, use:

```
make
```

To build and run the test suite, type:

```
make check
```

To build the documentation, type:

```
make hype-doc
```

To install HyPE, type (as root):

```
make install
```

2.3 Installing on Windows

HyPE was tested with Windows using the following toolchains:

- `Cygwin`
- `Minimal GNU for Windows (32-bit)`, GCC 4.7.2
- `Visual Studio C++ 2010 Express` (32-bit)

Naturally, other toolchains like MinGW-64 and newer versions of Visual Studio might work as well.

In any case, if you would like to build HyPE's documentation, install Doxygen via its Windows installer. If you let the installer add Doxygen to ‘PATH‘, the build system will locate it automatically.

You must also install CMake, presumably via its Windows installer. Let the installer add CMake to ‘PATH‘ as well.

2.3.1 Installing with Minimal GNU for Windows

To install HyPE using the MinGW toolchain, first build MinGW versions of the dependant libraries. They do not necessarily have to be installed into the MinGW/MSYS path hierarchy.

Boost **must** be built for the MinGW toolchain, Visual Studio builds will not work. If you cannot find an appropriate binary, build the binaries from a MinGW command shell (cmd.exe):

```
cd boost_1_53_0
boostrap mingw
bjam toolset=gcc
```

The Loki library **must** also be built for the MinGW toolchain. At first you must remove Loki's 'src/LevelMutex.cpp', since it is broken on MinGW and not required by HyPE. From a MinGW shell (MSYS Bash), type:

```
cd loki-0.1.7/
mingw32-make build-static build-shared OS=Windows
```

Now you are ready to build HyPE. From a MSYS Shell, type:

```
cd hype-library
mkdir build
cd build
cmake-gui ../
```

Choose "MSYS Makefiles" as the build system to generate and click _Configure_ to configure the HyPE package. There will be errors. In the CMake cache

- set 'BOOST_ROOT' to the build location of Boost for MinGW
- configure the 'Boost_USE_' options; presumably enable 'Boost_USE_STATIC_-LIBS'

Now _Configure_ again - Boost should be properly configured now but not the Loki library. So in the cache (advanced), set

- 'Loki_INCLUDE_DIRS' to the 'include/' subdirectory of Loki (e.g. 'C:/loki-0.1.-7/include')
- 'Loki_LIBRARIES' to the path of Loki's static library ('libloki.a') or DLL ('libloki.dll')

HyPE should now **Configure** properly and you can click **Generate**.

From a MSYS Shell you may now build HyPE just like you would on Linux:

```
cd hype-library/build
make
```

It may also be installed into the MinGW/MSYS paths.

Note: In order to run the test suite, copy all necessary DLLs to 'hype-library/build/examples/unitests/'.

2.3.2 Installing with Visual Studio C++

To install HyPE using the Visual Studio C++ toolchain, first build MSVC versions of the dependant libraries.

Boost **must** be built for the MSVC toolchain, MinGW builds will not work. If you cannot find an appropriate binary, build the sources from Windows command shell (cmd.exe):

```
cd boost_1_53_0
boostrap
bjam
```

The Loki library **must** also be built for the MSVC toolchain. From a Windows shell (cmd.exe) using Visual Studio C++ 2010, type:

```
cd loki-0.1.7/
set VS80COMNTOOLS=%VS100COMNTOOLS%
make.msvc.bat
```

Now you are ready to build HyPE. First generate the Visual Studio project files using C-Make. To do so, start CMake GUI and select the HyPE source directory. You may select a different build directory for out-of-source-tree builds. Click _Configure_ to configure the HyPE package. There will be errors. In the CMake cache

- set ‘BOOST_ROOT‘ to the build location of Boost for MSVC
- configure the ‘Boost_USE_‘ options; presumably enable ‘Boost_USE_STATIC_-LIBS‘

Now _Configure_ again - Boost should be properly configured now but not the Loki library. So in the cache (advanced), set

- ‘Loki_INCLUDE_DIRS‘ to the ‘include/‘ subdirectory of Loki (e.g. ‘C:/loki-0.1.-7/include‘)
- ‘Loki_LIBRARIES‘ to the path of Loki’s MSVC static library (‘loki.lib‘)

HyPE should now **Configure** properly and you can click **Generate**.

In Visual Studio, you can now open the generated solution file ‘HyPE.sln‘ (located in the build directory) and build the entire solution or a specific target.

Note: In order to run the test suite, copy all necessary DLLs to ‘hype-library/build/examples/unitests/‘ (or whatever your build directory is).

Note also, that when performing a _Debug_ build, all dependent libraries must be debug versions as well. Debug versions of Boost are automatically selected by the build system, but Loki library Debug versions must be manually built and selected in the C-Make cache.

Chapter 3

Tutorial: How to use HyPE

HyPE is organized as a library to allow easy integration in existing applications. You can choose between a dynamic and a static version of the library. Note that you have to link against the libraries HyPE uses, if you use the static version.

ATTENTION: HyPE uses the thread library of boost for it's advanced features. - There is a bug concerning applications compiled with the g++ compiler, because boost thread does not properly export all symbols until version 1.48. The Bug was fixed in `Boost 1.49`. The workaround is to statically link against boost thread. Further details can be found `here`.

To integrate HyPE in your project, you have to include the header file

```
#include <hype.hpp>
```

and link against hype:

```
g++ -g  -Wl,-rpath,${PATH_TO_HYPE_LIB}/lib -Wall -Werror -o <you application's
       name> <object files> -I${PATH_TO_HYPE_LIB}/include -Bstatic -lboost_thread -
      pthread -Bdynamic -L${PATH_TO_HYPE_LIB}/lib -lhype -lboost_system -lboost_filesystem
       -lboost_program_options-mt -lloki -lrt
```

The general concept of HyPE is to make decisions for your applications, which algorithm (processing device) should be used to perform an operation. Therefore, you first have to specify the Operations you wish to make decisions for and second you have to register your available algorithms for these operations. First we need a reference to the global Scheduler:

```
hype::Scheduler& scheduler=hype::Scheduler::instance();
```

HyPE uses two major abstractions: First, a DeviceSpecification, which defines information to a processing device, e.g., a CPU or GPU. Second, is an AlgorithmSpecification, which encapsulates algorithm specific information, e.g., the name, the name of the operation the algorithm belongs to as well as the learning and the load adaption strategy.

As example, we will create the configuration for the most common case: A system with one CPU and one dedicated GPU:

```
DeviceSpecification cpu_dev_spec(hype::PD0, //by convention, the first CPU has
       Device ID: PD0  (any system has at least one)
                                 hype::CPU, //a CPU is from type CPU
                                 hype::PD_Memory_0); //by convention, the host
       main memory has ID PD_Memory_0

DeviceSpecification gpu_dev_spec(hype::PD1, //different porcessing device
       (naturally)
                                 hype::GPU, //Device Type
                                 hype::PD_Memory_1); //separate device memory
```

Now, we have to define the algorithms. Note that an algorithm may utilize only one processeng device at a time (e.g., the GPU).

```
AlgorithmSpecification cpu_alg("CPU_Algorithm",
                               "SORT",
                               hype::StatisticalMethods::Least_Squares_1D,
                               hype::RecomputationHeuristics::Periodic,
                               hype::OptimizationCriterions::ResponseTime);

AlgorithmSpecification gpu_alg("GPU_Algorithm",
                               "SORT",
                               StatisticalMethods::Least_Squares_1D,
                               RecomputationHeuristics::Periodic,
                               OptimizationCriterions::ResponseTime);
```

Note that the GPU algorithm is only executable on the GPU and hence, should be assigned only to DeviceSpecifcations of ProcessingDeviceType GPU. **ATTENTION: the algorithm name in the AlgorithmSpecification has to be unique!** Let's assume that our CPU algorithm runs only on the CPU and the GPU algorithms runs only on the GPU. We define this by calling the method Scheduler::addAlgorithm:

```
scheduler.addAlgorithm(cpu_alg, cpu_dev_spec); //add CPU Algorithm to CPU
       Processing Device
scheduler.addAlgorithm(gpu_alg, gpu_dev_spec); //add GPU Algorithm to GPU
       Processing Device
```

We are now ready to use the scheduling functionality of HyPE. First, we have to identify the parameters of a data set that have a high impact on the algorithms execution time. In case of our sorting example, we identify the size of the input array as the most important feature value. Note that HyPE supports n feature values (n>=1).

To tell HyPE the feature value(s) of the data set that is to be processed, we have to store them in a hype::Tuple object. By convention, the first entry quantifies the size of the input data, and the second (if any) should contain the selectivity of a database operator.

```
hype::Tuple t;
t.push_back(Size_of_Input_Dataset);//for our sort operation, we only need the
       data size
```

Now, HyPE knows about your hardware and your algorithms. We can now let HyPE do scheduling decisions. HyPE needs two informations to perform scheduling decisions: First is a OperatorSpecification, which defines the operation that should be executed ("SORT"), and the feature vector of the input data (t). Furthermore, we have to specify the location of the input data as well as the desired location for the output data, so HyPE can take the cost for possible copy operations into account.

```
OperatorSpecification op_spec("SORT",
                              t,
                              hype::PD_Memory_0, //input data is in CPU RAM
                              hype::PD_Memory_0); //output data has to be
       stored in CPU RAM
```

The second information, which HyPE needs, is a specification of constraints on the processing devices. For some applications, operations cannot be executed on all processing devices for arbitrary data. For example, if a GPU has not enough memory to process a data set, the operation will fail (and will probably slow down other operations). Since HyPE cannot know this, because it does not know the semantic of the operations, the user can specify constraints, on which type of processing device the operation may be executed. In our case, we have no constraints and just default construct a Device-Constraint object.

```
DeviceConstraint dev_constr;
```

No we can ask HyPE were to execute our operation:

```
SchedulingDecision sched_dec = scheduler.getOptimalAlgorithm(op_spec,
      dev_constr);
```

Note that the application **always** has to execute the algorithm HyPE chooses, otherwise, all following calls to sched_dec.getOptimalAlgorithm() will have undefined behavior. - Since HyPE uses a feedback loop to refine the estimations of algorithm execution times, you have to measure the execution times of your algorithms and pass them back to Hy-PE. HyPE provides a high level interface for algorithm measurement:

```
AlgorithmMeasurement alg_measure(sched_dec); //has to be directly before
       algorithm execution
   //execute the choosen algortihm
alg_measure.afterAlgorithmExecution();  //has to be directly after algorithm
       termination
```

The AlgorithmMeasurement object starts a timer and afterAlgorithmExecution() stops the timer. Note that the constructor of the AlgorithmMeasurement object needs a - SchedulingDecision as parameter. When we put the usage of the SchedulingDecision together, we get the following code skeleton:

```
if(sched_dec.getNameofChoosenAlgorithm()=="CPU_Algorithm"){
   AlgorithmMeasurement alg_measure(sched_dec);
      //execute "CPU_Algorithm"
   alg_measure.afterAlgorithmExecution();
}else if(sched_dec.getNameofChoosenAlgorithm()=="GPU_Algorithm"){
   AlgorithmMeasurement alg_measure(sched_dec);
      //execute "GPU_Algorithm"
   alg_measure.afterAlgorithmExecution();
}
```

Some applications have their own time measurement routines and wish to use their own timer framework. To support such applications, HyPE offers a direct way to add a measured execution time in nanoseconds for a corresponding SchedulingDecision:

```
uint64_t begin=hype::core::getTimestamp();
CPU_algorithm(t[0]);
uint64_t end=hype::core::getTimestamp();
//scheduling decision and measured execution time in nanoseconds!!!
scheduler.addObservation(sched_dec,end-begin);
```

The complete source code of this example can be found in the documentation online-_learning::cpp and in the examples directory of HyPE (examples/use_as_online_-framework/online_learning.cpp).

Chapter 4

Available Components

4.1 Statistical Methods

A Statistical Method learns the relation between the feature values of the input dataset and an algorithms execution time. It is a central part of HyPE, implementing the learning based execution time estimation. Hence, it is crucial to select the appropriate statistical method depending on the algorithm and the application environment. Currently, HyP-E supports one dimensional Least Square Method and Multi Linear Fitting. Statistical methods are defined in the type hype::StatisticalMethods.

4.1.1 Least Square Method

HyPE uses the least square solver of the ALGLIB Project. It is usually the candidate to choose, if an algorithm only depends on one input features, such as sorting.

4.1.2 Multi Linear Fitting

HyPE uses the multi linear fitting functionality of the ALGLIB Project. You should choose Multi Linear Fitting, if an algorithm depends on multiple input features, such as selections (data size, selectivity). **Note that Multi Linear Fitting is currently limited to two features, but will support more in future versions of HyPE.**

4.2 Recomputation Heuristics

A Recomputation Heuristic implements the load adaption functionality of HyPE. If the load situation of a system dramatically changes, then it is very likely that the execution time of algorithm will change as well. To ensure sufficiently exact estimations, the learned approximation functions have to be updated. However, there is now 'perfect' point in time when to recompute the approximation functions. Therefore, the user can select a Recomputation Heuristic, which is appropriate for the application. Recomputation heuristics are defined in the type hype::RecomputationHeuristics.

4.2.1 Periodic Recomputation

The Periodic Recomputation Heuristic will recompute the approximation function of an algorithm after X executions of this algorithm. X is called Recomputation Period and can be configured as well (see **Configure HyPE** (p. 19) for details). You should use this Recomputation Heuristic if you want that HyPE refines its estimations at runtime to adapt at changing data, load, etc.

4.2.2 Oneshot Recomputation

The Oneshot Recomputation Heuristik will compute the approximation functions once after the initial training phase. You should choose this optimization criterion, if significant changes in the load in your system is seldom or have little impact on algorithms execution time.

4.2.3 Error based Recomputation (under development)

4.3 Optimization Criterions

An Optimization Criterion specifies what an "optimal" algorithm for your application is. Should it be the fastest? Or would you like to select algorithms in a way that the throughput of your system is optimized? Therefore, we implemented several strategies to make HyPE configurable and better usable for a wide range of applications. Optimization criteria are defined in the type hype::OptimizationCriterions.

4.3.1 Response Time

The idea of Response Time optimization is to reduce the execution time of one operation by selecting the (estimated) fastest algorithm.

4.3.2 Waiting Time Aware Response Time

Waiting Time Aware Response Time (WTAR) is an extension of the simple response time algorithm. WTAR takes into account the load on all processing devices and allocates for an operation O the processing device, were the sum of the waiting time, until the previous oeprators finished, and the estimated execution time of O is minimal. This is the recommended optimization algorithm for HyPE.

4.3.3 Round Robin

The round robin strategy allocates processing devices for operations in turns, distributing a workload of operations on all processing devices. This approach works well in case operation need roughly the same time on all processing devices (e.g., on homogeneous hardware). However, in case one processing device is significantly faster than the other processing devices, the round strategy will under utilize the faster processing device, and over utilize the slower processing devices.

4.3.4 Threshold-based Outsourcing

Threshold-based Outsourcing is an extension of Response Time. The idea is to force the use of a slower processing device to relieve the fastest processing device and distribute the workload on all available processing devices. However, the algorithm has to ensure that the operation's response time does not significantly increase. Therefore, an operation may be executed on a slower processing device, if and only if the expected slowdown is under a certain threshold W.

4.3.5 Probability-based Outsourcing

Probability-based Outsourcing computes for each scheduling decision the estimated execution times of the avaialble algorithms. Then, each algorithm gets assigned a probability, depending on the estimated execution time. Faster algorithms (on faster processing devices) get a higher probability to be executed then slower algorithms. Depending on the probability, an algorithm is chosen randomly for execution.

Chapter 5

Configure HyPE

HyPE can be configured in four ways. First, modify the Static_Configuration of HyPE, which sets default values for all variables. Second, update the configuration at runtime. The class Runtime_Configuration provides methods to change all modifiable variables. Note that not all variables are modifiable during runtime. Third, create a configuration file 'hype.conf', and add the variables with their corresponding values. Note, that the structure of the file for each line is variable_name=value and one line may at most contain one assignment. Fourth, specify parameter values in environment variables.

- modify the hype::core::Static_Configuration of HyPE (requires recompilation)
- update the configuration at runtime using hype::core::Runtime_Configuration
- create a configuration file 'stemod.conf', and add the variables with their corresponding values
  - **help** produce help message
  - **length_of_trainingphase** set the number algorithms executions to complete training
  - **history_length** set the number of measurement pairs that are kept in the history (important for precision of approximation functions)
  - **recomputation_period** set the number of algorithm executions to trigger recomputation
  - **algorithm_maximal_idle_time** set maximal number of operation executions, where an algorithm was not executed; forces retraining of algorithm
  - **retraining_length** set the number of algorithm executions needed to complete a retraining phase (load adaption feature)
  - **ready_queue_length** set the number of operators that are queued on a processing device, before scheduling decision stops scheduling new operators (The idea is to wait how the done scheduling decisions turn out and to adjust the scheduling accordingly)
- specify parameter values in environment variables:

- HYPE_LENGTH_OF_TRAININGPHASE set the number algorithms executions to complete training
- HYPE_HISTORY_LENGTH set the number of measurement pairs that are kept in the history (important for precision of approximation functions)
- HYPE_RECOMPUTATION_PERIOD set the number of algorithm executions needed to complete a retraining phase (load adaption feature)
- HYPE_ALGORITHM_MAXIMAL_IDLE_TIME set maximal number of operation executions, where an algorithm was not executed; forces retraining of algorithm
- HYPE_RETRAINING_LENGTH set the number of algorithm executions to trigger recomputation
- HYPE_READY_QUEUE_LENGTH set the number of operators that are queued on a processing device, before scheduling decision stops scheduling new operators (The idea is to wait how the done scheduling decisions turn out and to adjust the scheduling accordingly)

Chapter 6

Extend HyPE

6.1 Statistical Methods

HyPE learns the correlation between features of the input data set and the resulting execution time of an algorithm on a specific processing device. To allow the user to fine tune the statistical method, HyPE provides a plug-in architecture, where the user can choose either from the set of implemented statistical methods, or alternatively, implement and integrate the preferred statistical method in HyPE.

To create a new statistical method, the user has to inherit from the abstract base class hype::core::StatisticalMethod and implement its pure virtual methods. Since HyPE uses a plug-in architecture based on factories, a static member function **create** should be defined, which returns a pointer to a new instance of your new statistical method (e.g., Least_Squares_Method_1D).

```
#include <core/statistical_method.hpp>

namespace hype{
   namespace core{

      class Least_Squares_Method_1D : public StatisticalMethod {
      public:
         Least_Squares_Method_1D();

         virtual const EstimatedTime computeEstimation(const Tuple&
      input_values);

         virtual bool recomuteApproximationFunction(Algorithm& algorithm);

         virtual bool inTrainingPhase() const throw();

         virtual void retrain();

         static Least_Squares_Method_1D* create(){
            return new Least_Squares_Method_1D();
         }

         virtual ~Least_Squares_Method_1D();
      };
   }; //end namespace core
}; //end namespace hype
```

After the user added the class, he needs to extend the enumeration hype::Statistical-

Methods::StatisticalMethod in file global_definitions.hpp by a new member, which identifies the plug-in. Finally, the user has to register the plug-in in the class hype::core::-PluginLoader in file pluginloader.cpp.

6.2 Recomputation Heuristics

HyPE is capable of refining estimated execution times at run-time. Depending on the application, a run-time refinement is beneficial or causes only additional overhead. To support a wide range of applications, HyPE allows to fine tune the runtime refinement on a per algorithm basis (e.g, for the same operation, we can have runtime adaption on the CPU, but not on the GPU.)

HyPE provides a plug-in architecture, where the user can choose either from the set of implemented recomputation heuristics, or alternatively, implement and integrate the preferred recomputation heuristic in HyPE.

To create a new recomputation heuristic, the user has to inherit from the abstract base class hype::core::RecomputationHeuristic and implement its pure virtual methods. -Since HyPE uses a plug-in architecture based on factories, a static member function **create** should be defined, which returns a pointer to a new instance of your new recomputation heuristic (e.g., Oneshotcomputation).

```
#include <core/recomputation_heuristic.hpp>

namespace hype{
   namespace core{

      class Oneshotcomputation : public RecomputationHeuristic {
         public:
         Oneshotcomputation();
         //returns true, if approximation function has to be recomputed and
        false otherwise
         virtual bool internal_recompute(Algorithm& algortihm);

         static Oneshotcomputation* create(){
            return new Oneshotcomputation();
         }

      };
   }; //end namespace core
}; //end namespace hype
```

After the user added the class, he needs to extend the enumeration hype::-RecomputationHeuristics::RecomputationHeuristic in file global_definitions.hpp by a new member, which identifies the plug-in. Finally, the user has to register the plug-in in the class hype::core::PluginLoader in file pluginloader.cpp.

6.3 Optimization Criterions

To add a new optimization criterion, the user has to inherit from the abstract class hype-::core::OptimizationCriterion and implement its pure vitual methods. Since HyPE uses a plug-in architecture based on factories, a static member function **create** should be defined, which returns a pointer to a new instance of your new optimization criterion, which we will call "NewResponseTime".

```
#include <core/optimization_criterion.hpp>

namespace hype{
   namespace core{
      class NewResponseTime : public OptimizationCriterion{
         public:
         NewResponseTime(const std::string& name_of_operation);

         virtual const SchedulingDecision getOptimalAlgorithm_internal(const
      Tuple& input_values, Operation& op, DeviceTypeConstraint dev_constr);
         //factory function
         static NewResponseTime* create(){
            return new NewResponseTime("");
         }
      }
   };
};
```

After the user added the class, he needs to extend the enumeration hype::Optimization-Criterions::OptimizationCriterion in file global_definitions.hpp by a new member, which identifies the plug-in. Finally, the user has to register the plug-in in the the class hype-::core::PluginLoader in file pluginloader.cpp.

Chapter 7

FAQ

1. **What is HyPE?**

   HyPE is a Hybrid query Processing Engine build for automatic selection of processing units for co-processing in database systems. The long-term goal of the project is to implement a fully fledged query processing engine, which is able to automatically generate and optimize a hybrid CPU/GPU physical query plan from a logical query plan.

2. **When should I use HyPE?**

   You can use HyPE whenever you want to decide on a CPU and a GPU implementation of an operation in your application at run-time. In other words, any GPU accelerated application can make use of hype to effectively utilize existing processing ressources.

3. **Under which license is HyPE distributed?**

   HyPE is licenced under the `GNU LESSER GENERAL PUBLIC LICENSE - Version 3`.

4. **Which platforms are currently supported?**

   HyPE compiles and runs under Linux and Windows (Cygwin). Native Windows support is planned for future releases.

5. **I have a problem in using HyPE, how can I get help?**

   For information about the project, technical questions and bug reports: please contact the development team via `Sebastian Breß`.

Chapter 8

Class Index

8.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 9

File Index

9.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 10

Class Documentation

10.1 hype::core::HYPE_EXPORT Protocol Reference

This class represents an easy to use interface for time measurement of an algorithms execution time.

Public Member Functions

- **AlgorithmMeasurement** (const **SchedulingDecision** &scheduling_decision)

  constructs an AlgorithmMeasurement object
- void **afterAlgorithmExecution** ()

  stops the time of an algorithms execution and adds obtained data to algorithm statistics.
- **SchedulingDecision** (Algorithm &alg_ref, const EstimatedTime &estimated_-time_for_algorithm, const Tuple &feature_values)

  constructs a SchedulingDecision object by assigning neccessary informations to all fields of the object
- const std::string **getNameofChoosenAlgorithm** () const

  returns the name of the choosen algorithm
- const EstimatedTime **getEstimatedExecutionTimeforAlgorithm** () const

  returns the estimated execution time of the choosen algorithm
- const Tuple **getFeatureValues** () const

  returns the feature values that were the basis for this decision
- const **DeviceSpecification getDeviceSpecification** () const throw ()

  returns the ComputeDevice the chosen algorithm utilizes
- bool **operator==** (const **SchedulingDecision** &sched_dec) const
- **AlgorithmSpecification** (const std::string &alg_name, const std::string &op-_name, StatisticalMethods::StatisticalMethod stat_meth=StatisticalMethods::-Least_Squares_1D, RecomputationHeuristics::RecomputationHeuristic recomp-_heur=RecomputationHeuristics::Periodic, OptimizationCriterions::Optimization-Criterion opt_crit=OptimizationCriterions::ResponseTime)

constructs an AlgorithmSpecification object by assigning necessary informations to all fields of the object

- const std::string & **getAlgorithmName** () const throw ()

  returns the algorithm's name
- const std::string & **getOperationName** () const throw ()

  returns the name of the operation the algorithm belongs to
- const std::string **getStatisticalMethodName** () const throw ()

  returns the name of the statistical method that is used for the algorithm
- const std::string **getRecomputationHeuristicName** () const throw ()

  returns the name of the recomputation heuristic that is used for the algorithm
- const std::string **getOptimizationCriterionName** () const throw ()

  returns the name of the optimization criterion of the operation the algorithm belongs to
- **OperatorSpecification** (const std::string &operator_name, const Tuple &feature-_vector, ProcessingDeviceMemoryID location_of_input_data, ProcessingDevice-MemoryID location_for_output_data)

  constructs an OperatorSpecification object by assigning necessary informations to all fields of the object
- const std::string & **getOperatorName** () const throw ()

  returns the operations's name
- const Tuple & **getFeatureVector** () const throw ()

  returns the feature vector of this operator
- ProcessingDeviceMemoryID **getMemoryLocation** () const throw ()

  returns the memory id where the input data is stored
- **DeviceSpecification** (ProcessingDeviceID pd, ProcessingDeviceType pd_t, -ProcessingDeviceMemoryID pd_m)

  constructs an DeviceSpecification object by assigning necessary informations to all fields of the object
- ProcessingDeviceID **getProcessingDeviceID** () const throw ()

  returns the processing device's ProcessingDeviceID
- ProcessingDeviceType **getDeviceType** () const throw ()

  returns the processing device's device type
- ProcessingDeviceMemoryID **getMemoryID** () const throw ()

  returns the processing device's memory id
- **operator ProcessingDeviceID** ()

  implicit conversion to an object of type ProcessingDeviceID
- **operator ProcessingDeviceType** ()

  implicit conversion to an object of type ProcessingDeviceType
- **operator ProcessingDeviceMemoryID** ()

  implicit conversion to an object of type ProcessingDeviceMemoryID
- bool **operator==** (const **DeviceSpecification** &) const

  overload of operator== for this class
- **DeviceConstraint** (DeviceTypeConstraint dev_constr=ANY_DEVICE, -ProcessingDeviceMemoryID pd_mem_constr=PD_Memory_0)

  constructs an DeviceConstraint object by assigning necessary informations to all fields of the object

- DeviceTypeConstraint **getDeviceTypeConstraint** () const

  returns the DeviceTypeConstraint
- **operator DeviceTypeConstraint** ()

  implicit conversion to an object of type DeviceTypeConstraint
- **operator ProcessingDeviceMemoryID** ()

  implicit conversion to an object of type ProcessingDeviceMemoryID
- **operator DeviceTypeConstraint** () const

  implicit conversion to an object of type DeviceTypeConstraint
- **operator ProcessingDeviceMemoryID** () const

  implicit conversion to an object of type ProcessingDeviceMemoryID
- **EstimatedTime** ()
- **EstimatedTime** (double time_in_nanoseconds)
- double **getTimeinNanoseconds** () const
- **MeasuredTime** ()
- **MeasuredTime** (double time_in_nanoseconds)
- double **getTimeinNanoseconds** () const

10.1.1 Detailed Description

A DeviceConstraint restricts the type of processing device, which HyPE may choose to process an operator.

A DeviceSpecification defines a processing device that is available for performing computations.

A OperatorSpecification defines the operator that the user wants to execute.

An AlgorithmSpecification specifies all relevant information about an algorithm, such as the algorithm's name or the name of the operation the algorithms belongs to.

This class represents a scheduling decision for an operation w.r.t. a user specified set of features of the input data.

The user just have to create an object of AlgorithmMeasurement. Then, the algorithm that is to measure is executed. After that, the user has to call **afterAlgorithmExecution()** (p. 36). The framework takes care of the rest.

Internally, **afterAlgorithmExecution()** (p. 36) stops the time and adds the received data to the algorithms statistics. **It is crucial that the application executes the algortihm specified by the SchedulingDecision or the libraries behaviour is undefined.**

Author

Sebastian Breß

Version

0.1

Date

2012

A user has to execute the algortihm suggested by the SchedulingDecision, or the library will not work. The user can determine the algorithm to excute by calling **getNameof-ChoosenAlgorithm()** (p. 36). The user can then measure the execution time of the algorithm by using an AlgorithmMeasurement object.

Author

Sebastian Breß

Version

0.1

Date

2012

Author

Sebastian Breß

Version

0.2

Date

2013

Copyright

GNU LESSER GENERAL PUBLIC LICENSE - Version 3, `http://www.gnu.org/licenses/lgpl-3.0.txt`

Hence, it contains the name of the operation and the features of the input data set as well as the two memory ids, where the first identifies where the input data is stored, and the second specifies where the input data is stored. Note that HyPE needs this information to estimate the overhead of data transfers in case the data needs to be copied to use a certain processing device.

Author

Sebastian Breß

Version

0.2

Date

2013

Copyright

GNU LESSER GENERAL PUBLIC LICENSE - Version 3, `http://www.gnu.-org/licenses/lgpl-3.0.txt`

It consists of a ProcessingDeviceID, which has to be unique, a processing device type (e.g., CPU or GPU) and a memory id, which specifies the memory that the processing devices uses. By convention, the host's CPU has the processing device id of 0, is a processing device from type CPU and the CPU's main memory has memory id 0.

Author

Sebastian Breß

Version

0.2

Date

2013

Copyright

GNU LESSER GENERAL PUBLIC LICENSE - Version 3, `http://www.gnu.-org/licenses/lgpl-3.0.txt`

This is especially important if an algorithms does not support a certain data type on a certain processing device (e.g., no filter operations on an array of strings on the GPU). On default construction, no constraint is defined.

Author

Sebastian Breß

Version

0.2

Date

2013

Copyright

GNU LESSER GENERAL PUBLIC LICENSE - Version 3, `http://www.gnu.-org/licenses/lgpl-3.0.txt`

10.1.2 Member Function Documentation

10.1.2.1 hype::core::HYPE_EXPORT::AlgorithmMeasurement (const SchedulingDecision & *scheduling_decision*) [explicit]

The constructor will fetch the current time. Therefore, it starts a timer to measure the execution time of the choosen algorithm. Therefore, the user should construct the - AlgorithmMeasurement object directly before the algorithms call. Directly after the algorthm finished execution, **afterAlgorithmExecution()** (p. 36) has to be called to ensure a precise time measurement.

Parameters

cheduling_-decision	a reference to a SchedulingDecision

10.1.2.2 void hype::core::HYPE_EXPORT::afterAlgorithmExecution ( )

To ensure a precise time measurement, **afterAlgorithmExecution()** (p. 36) has to be called directly after the algorthm finished execution.

10.1.2.3 hype::core::HYPE_EXPORT::SchedulingDecision (Algorithm & *alg_ref,* const EstimatedTime & *estimated_time_for_algorithm,* const Tuple & *feature_values*)

Parameters

alg_ref	Reference to Algorithm that was choosen
estimated_-time_for_-algorithm	estimated execution time for algorithm
feature_-values	features of the input data set that were previously specified by the user

10.1.2.4 const std::string hype::core::HYPE_EXPORT::getNameofChoosen-Algorithm ( ) const

Returns

name of choosen algorithm

10.1.2.5 const EstimatedTime hype::core::HYPE_EXPORT::getEstimatedExecution-TimeforAlgorithm ( ) const

Returns

estimated execution time of the choosen algorithm

10.1.2.6 const Tuple hype::core::HYPE_EXPORT::getFeatureValues ( ) const

Returns

feature values of input data set

10.1.2.7 const DeviceSpecification hype::core::HYPE_EXPORT::getDeviceSpecification ( ) const throw ()

Returns

ComputeDevice the chosen algorithm utilizes

10.1.2.8 hype::core::HYPE_EXPORT::AlgorithmSpecification (const std::string & *alg_name*, const std::string & *op_name*, StatisticalMethods::StatisticalMethod *stat_meth* = `StatisticalMethods::Least_Squares_1D`, RecomputationHeuristics::RecomputationHeuristic *recomp_heur* = `RecomputationHeuristics::Periodic`, OptimizationCriterions::OptimizationCriterion *opt_crit* = `OptimizationCriterions::ResponseTime`)

Parameters

alg_name	name of the algorithm
op_name	name of the operation the algorithms belongs to
stat_meth	the statistical method used for learning the algorithms behavior (optional)
recomp_heur	the recomputation heuristic used for adapting the algorithms approximation function (optional)
opt_crit	the optimization criterion of the operation the algorithm belongs to (optional)

10.1.2.9 hype::core::HYPE_EXPORT::OperatorSpecification (const std::string & *operator_name*, const Tuple & *feature_vector*, ProcessingDeviceMemoryID *location_of_input_data*, ProcessingDeviceMemoryID *location_for_output_data*)

Parameters

operator_name	the operations's name
feature_vector	the feature vector of this operator
location_of_input_data	the memory id where the input data is stored
location_for_output_data	the memory id where the output data is stored

10.1.2.10 hype::core::HYPE_EXPORT::DeviceSpecification (ProcessingDeviceID *pd,* ProcessingDeviceType *pd_t,* ProcessingDeviceMemoryID *pd_m*)

Parameters

pd	the unique id of the processing device
pd_t	type of the processing device (e.g., CPU or GPU)
pd_m	unique id of the memory the processing device uses

10.1.2.11 hype::core::HYPE_EXPORT::DeviceConstraint (DeviceTypeConstraint *dev_constr =* `ANY_DEVICE`*,* ProcessingDeviceMemoryID *pd_mem_constr =* `PD_Memory_0`)

Parameters

dev_constr	a device type constraint (e.g., CPU_ONLY or ANY_DEVICE for now restriction)
pd_mem_-constr	memory id, where the data should be stored when processed (experi-mental)

10.1.2.12 hype::core::HYPE_EXPORT::operator DeviceTypeConstraint ( )

non-const version

10.1.2.13 hype::core::HYPE_EXPORT::operator ProcessingDeviceMemoryID ( )

non-const version

10.1.2.14 hype::core::HYPE_EXPORT::operator DeviceTypeConstraint ( ) const

const version

10.1.2.15 hype::core::HYPE_EXPORT::operator ProcessingDeviceMemoryID ( ) const

const version

10.2 hype::HYPE_EXPORT Protocol Reference

The Scheduler is the central component for interaction of the application and the library.

Public Member Functions

- bool **addAlgorithm** (const AlgorithmSpecification &alg_spec, const Device-Specification &dev_spec)

  adds an Algorithm to the AlgorithmPool of an operation defined by alg_spec on the processing device defined by dev_spec.
- bool **setOptimizationCriterion** (const std::string &name_of_operation, const std-::string &name_of_optimization_criterion)

  assigns the OptimizationCriterion name_of_optimization_criterion to Operation name-_of_operation
- bool **setStatisticalMethod** (const std::string &name_of_algorithm, const std-::string &name_of_statistical_method)

  assigns the StatisticalMethod name_of_statistical_method to an existing Algorithm
- bool **setRecomputationHeuristic** (const std::string &name_of_algorithm, const std::string &name_of_recomputation_strategy)

  assigns the StatisticalMethod name_of_statistical_method to an existing Algorithm
- const SchedulingDecision **getOptimalAlgorithm** (const OperatorSpecification &op_spec, const DeviceConstraint &dev_constr)

  Returns a Scheduling Decision, which contains the name of the estimated optimal Algorithm w.r.t. the user specified optimization criterion.
- bool **addObservation** (const SchedulingDecision &sched_dec, const double &measured_execution_time)

  adds an observed execution time to the algorithm previously choosen by getOptimal-AlgorithmName.
- core::EstimatedTime **getEstimatedExecutionTime** (const OperatorSpecification &op_spec, const std::string &alg_name)

Static Public Member Functions

- static Scheduler & **instance** ()

  This method implements the singelton concept for the Scheduler class to avoid multiple instances.

10.2.1 Detailed Description

The Scheduler provides two main functionalities. First, it provides the service to decide on the optimal algorithm for an operation w.r.t. a user specified optimization criterion. Second, the Scheduler implements an interface to add new Observations to the executed Algorithm. Hence, it is the central component for interaction of the application and the library. Since it is not meaningful to have multiple instances of the Scheduler class, it is not possible to create multiple Scheduler instances. This property is implemented by using the singelton concept. Additionally, the Scheduler enables the user to setup the Operations with their respective Algorithms as well as to configure for each algorithm a statistical method and a recomputation heuristic and for each operation an optimization criterion. Note that the statistical method and the recomputation statistic can be exchanged at run-time, because the Algortihm uses the pointer to implementation technique (or pimpl-idiom). This class is the interface for using stemod. It forwards calls to the Scheduler in stemod::core and implements thread safety.

Author

Sebastian Breß

Version

0.1

Date

2012

Copyright

GNU LESSER GENERAL PUBLIC LICENSE - Version 3, `http://www.gnu.org/licenses/lgpl-3.0.txt`

10.2.2 Member Function Documentation

10.2.2.1 static Scheduler& **hype::HYPE_EXPORT::instance** ( ) `[static]`

Returns

Reference to Scheduler instance.

10.2.2.2 bool **hype::HYPE_EXPORT::addAlgorithm** (const AlgorithmSpecification & *alg_spec*, const DeviceSpecification & *dev_spec*)

If the specified operation does not exist, it is created. Multiple calls to addAlgorithm with an AlgorithmSpecification having the same Operation name will add the respective algorithms to the algorithm pool of the specified Operation

Parameters

alg_spec	defines properties of the algorithm, e.g., name, the operation it belongs to, etc.
dev_spec	defines properties of the processing device the algorithm runs on, e.g., device type (CPU or GPU) and the device id

Returns

returns true on success and false otherwise

10.2.2.3 bool hype::HYPE_EXPORT::setOptimizationCriterion (const std::string & *name_of_operation*, const std::string & *name_of_optimization_criterion*)

Parameters

name_of_-operation	name of the Operation
name_of_-optimization-_criterion	Name of OptimizationCriterion

Returns

returns true on success and false otherwise

10.2.2.4 bool hype::HYPE_EXPORT::setStatisticalMethod (const std::string & *name_of_algorithm*, const std::string & *name_of_statistical_method*)

Parameters

name_of_-algorithm	Name of Algorithm
name_of_-statistical_-method	assigns the StatisticalMethod name_of_statistical_method to an existing Algorithm

Returns

returns true on success and false otherwise

10.2.2.5 bool hype::HYPE_EXPORT::setRecomputationHeuristic (const std::string & *name_of_algorithm*, const std::string & *name_of_recomputation_strategy*)

Parameters

name_of_-algorithm	Name of Algorithm
name_of_-recomputation-_strategy	assigns the RecomputationHeuristic name_of_recomputation_strategy to an existing Algorithm

Returns

returns true on success and false otherwise

10.2.2.6 const SchedulingDecision hype::HYPE_EXPORT::getOptimalAlgorithm (const OperatorSpecification & *op_spec,* const DeviceConstraint & *dev_constr*)

Parameters

op_spec	OperatorSpecification, contains all available information about the operator to execute
dev_constr	DeviceConstraint, restricting the available algorithms to a subset of the algorithm pool (e.g., allow only CPU algorithms)

Returns

SchedulingDecision, which contains the suggested algortihm for the specified information

10.2.2.7 bool hype::HYPE_EXPORT::addObservation (const SchedulingDecision & *sched_dec,* const double & *measured_execution_time*)

Parameters

sched_dec	the scheduling decision, this observation belongs to
measured_-execution_-time	measured execution time, in nanoseconds!!!

Returns

true on success and false in case an error occured

Chapter 11

File Documentation

11.1 documentation.hpp File Reference

This file contains additional documentation, like the generated web pages in the doxygen documentation.

11.1.1 Detailed Description

Author

Sebastian Breß

Version

0.1

Date

2012

Copyright

Index