HEPnOS Documentation

Argonne National Laboratory

Apr 06, 2023

CONTENTS

HEPnOS is an transient, in-memory, distributed storage system for high energy physics (HEP) workflows running on supercomputers. It is based on software components fron the Mochi project and was designed in the context of the SciDAC-4 “HEP on HPC” collaboration between Argonne National Laboratory and FermiLab.

This website gathers documentation and tutorials on how to install it and use it.

CHAPTER ONE

CONTENTS

1.1 Installing

The recommended way to install the HEPnOS and its dependencies is to use Spack. Spack is a package management tool designed to support multiple versions and configurations of software on a wide variety of platforms and environments.

1.1.1 Installing Spack and the Mochi repository

First, you will need to install Spack as explained here. Once Spack is installed and available in your path, clone the following git reporitory and add it as a Spack namespace.

```
git clone https://github.com/mochi-hpc/mochi-spack-packages.git
spack repo add mochi-spack-packages
```

You can then check that Spack can find HEPnOS by typping:

```
spack info hepnos
```

You should see something like the following.

```
CMakePackage:   hepnos

Description:
    Object store for High Energy Physics, build around Mochi components

Homepage: https://github.com/hepnos/HEPnOS
... (more lines follow) ...
```

1.1.2 Installing HEPnOS

Installing HEPnOS is then as simple as typping the following.

```
spack install hepnos
```

1.1.3 Loading and using HEPnOS

Once installed, you can load HEPnOS using the following command.

```
spack load hepnos
```

This will load HEPnOS and its dependencies (Mercury, Thallium, Argobots, etc.). You are now ready to use HEPnOS!

1.1.4 Using the HEPnOS client library with cmake

Within a cmake project, HEPnOS can be found using:

```
find_package(hepnos REQUIRED)
```

You can now link targets as follows.

```
add_executable(my_hepnos_client source.c)
target_link_libraries(my_hepnos_client hepnos)
```

1.1.5 Using the HEPnOS client libraries with pkg-config

Support for pkg-config is still preliminary and not completely supported. Once loaded into your working environment, `pkg-config --libs --cflags hepnos` will give you the CFLAGS and LDFLAGS you need to link your code against HEPnOS. However this command will not give you CFLAGS or LDFLAGS for the Boost serialization library, which does not itself support pkg-config. You will have to add these manually.

1.2 Concepts and data organization

HEPnOS handles data in a hierarchy of DataSets, Runs, SubRuns, and Events. Each of these constructs can be used to store data objects, or Products.

1.2.1 DataSets

DataSets are named containers. They can contain other DataSets, as well as Runs. DataSet can be seen as the equivalent of file system directories. While HEPnOS enables iterating over the DataSets stored in a parent DataSet, it has not been deesigned to efficiently handle a large number of them. Operations on a DataSet include creating a child DataSet, creating a Run, iterating over the child DataSets, iterating over Runs, searching for child DataSets by name and child Runs by run number.

1.2.2 Runs

Runs are numbered containers. They are identified by an integer between 0 and *InvalidRunNumber*, and can contain only SubRuns. Operations on a Run include creating and accessing individual SubRuns, iterating over SubRuns, and searching for specific SubRuns.

1.2.3 SubRuns

SubRuns are numbered containers. They are identified by an integer between 0 and *InvalidSubRunNumber*-1, and can contain only Events. Operations on a SubRun include creating and accessing individual Events, iterating over events, and searching for specific Events.

1.2.4 Events

Events are numbered containers. They are identified by an integer between 0 and *InvalidEventNumber*-1. They may only be used to store and load Products.

1.2.5 Products

Products are *key/value* pairs where the *key* is formed of a string label and the C++ type of the *value* object, while *value* is the data from the stored C++ object. While Products can be stored in DataSets, Runs, SubRuns, and Events, they are typically only stored in Events.

As the only type of named container, DataSets are a convenient way of naming data coming out of an experiment or a step in a workflow. Runs, SubRuns, and Events are stored in a way that optimizes search and iterability in a distributed manner. A DataSet can be expected to store a large number of runs themselves containing a large number of subruns and ultimately events. Products are stored in a way that does not make them iterable. It is not possible, from a container, to list the contained Products. The label and C++ type of a Product have to be known in order to retrieve the corresponding Product data from a container.

1.3 Deployment

1.3.1 Creating a configuration file

HEPnOS relies on the Bedrock Mochi microservice for bootstrapping and configuration.

The first step before deploying HEPnOS is to create a configuration file. This configuration file should be in JSON format and at least contain the following.

```
{
   "ssg" : [
     {
         "name" : "hepnos",
         "bootstrap" : "mpi",
         "group_file" : "hepnos.ssg",
         "swim" : { "disabled" : true }
     }
   ],
   "libraries" : {
     "yokan" : "libyokan-bedrock-module.so"
   },
   "providers" : [
     {
         "name" : "hepnos",
         "type" : "yokan",
         "config" : {
             "databases" : [
```

(continues on next page)

(continued from previous page)

```
                {
                    "name" : "hepnos-datasets",
                    "type" : "map",
                    "config": {}
                },
                {
                    "name" : "hepnos-runs",
                    "type" : "map",
                    "config": {}
                },
                {
                    "name" : "hepnos-subruns",
                    "type" : "map",
                    "config": {}
                },
                {
                    "name" : "hepnos-events",
                    "type" : "map",
                    "config": {}
                },
                {
                    "name" : "hepnos-products",
                    "type" : "map",
                    "config": {}
                }
            ]
        }
    }
  ]
}
```

This example configuration file only provides the bare minimum to get started. The *"ssg"* section sets up the group management component. The only important field here is the name of the group file, which we will use later.

The *"providers"* section should contain at least one Yokan provider with a number of databases. These databases must have a name that starts with *"hepnos-datasets"*, *"hepnos-runs"*, *"hepnos-subruns"*, *"hepnos-events"*, or *"hepnos-products"*. At least one database for each type of data should be provided, but you are free to use more than one database for some types of data, as long as their name starts with the above prefixes. For example, you can have two databases to store events, named *"hepnos-events-1"* and *"hepnos-events-2"*.

1.3.2 Configuring with the HEPnOS Wizard

An easy way of creating a HEPnOS configuration for Bedrock is to use the HEPnOS Wizard, which can be installed as follows.

```
$ spack install py-hepnos-wizard
```

Once installed and loaded, you can use it as follows.

```
$ hepnos-gen-config --address na+sm --output=myconfig.json
```

The only required parameter is the *address*, which should be a valid protocol accepted by the underlying Mercury library (e.g. *na+sm*, *ofi+tcp*, and so on).

Passing –*help* to hepnos-gen-config will provide information on all the available arguments and their meaning.

1.3.3 Deploying HEPnOS on a single node

To deploy HEPnOS on a single node, simply ssh into the node and type the following.

```
bedrock na+sm -c config.json
```

Change *na+sm* into the protocol that you want to use for communication. This tells Bedrock to start and initialize itself with the provided configuration. The command will block. To run it as a daemon, put it in the background, use nohup, or another other mechanism available on your platform.

1.3.4 Deploying HEPnOS on multiple nodes

The bedrock program can just as simply be deployed on multiple nodes, using your favorite MPI laucher (mpirun, aprun, etc.), for instance:

```
mpirun -np 4 -f hostfile bedrock na+sm -c config.json
```

1.3.5 Getting connection information

Once deployed, run the following command to obtain connection information readable by the client.

```
hepnos-list-databases na+sm -s ssg-file > connection.json
```

Where *ssg-file* is the name of the SSG file as specified in your HEPnOS configuration file.

This command will query the service and print a JSON representation of the information required for a client to connect to HEPnOS (addresses, database ids, etc.). Hence we pipe its output to a *connection.json* file that the clients will use later.

Important: On some platforms, you will need to launch this command as an MPI application running on a single process/node (typically if your login node does not connect to the compute nodes via the same type of network as across compute nodes).

1.4 Client connection and service shutdown

The following code sample showcases how to initialize a *DataStore* object in a client program. This object is the main entry point to the HEPnOS storage system. Its `connect()` function takes the name of a JSON file as a parameter, which should be the file created by the *hepnos-list-databases* command.

```
#include <iostream>
#include <string>
#include <hepnos.hpp>

using namespace hepnos;

int main(int argc, char** argv) {
```

(continues on next page)

(continued from previous page)

```
    if(argc != 3) {
        std::cerr << "Usage: " << argv[0] << "<protocol> <configfile>" << std::endl;
        exit(-1);
    }

    DataStore datastore = DataStore::connect(argv[1], argv[2]);

    // ...

    // only if you want to shutdown the HEPnOS service
    datastore.shutdown();
}
```

The `DataStore::connect()` function may also take an additional parameter to supply a JSON configuration file for the underlying Margo layer (see the Margo documentation for more information on the format of this configuration file).

An useful example of Margo JSON file could be one that sets up a dedicated progress thread:

```
{
    "use_progress_thread": true
}
```

Setting this value to `true` can be useful if the application relies on asynchronous operations (`AsyncEngine`).

The `DataStore::shutdown()` method can be used to tell the HEPnOS service to shutdown.

Important: The `DataStore::shutdown()` method should be called *by only one* client and will terminate *all* the HEPnOS server processes. If HEPnOS is setup to use in-memory databases, you will loose all the data stored in HEPnOS. If multiple clients call this method, they will either block or fail, depending on the network protocol used by HEPnOS.

1.5 Accessing DataSets

The example code bellow shows how to create DataSets inside other DataSets, how to iterate over all the child datasets of a parent DataSet, how to access a DataSet using an "absolute path" from a parent DataSet, and how to search for DataSets.

main.cpp (show/hide)

```
#include <iostream>
#include <string>
#include <hepnos.hpp>

using namespace hepnos;

int main(int argc, char** argv) {

    if(argc != 3) {
        std::cerr << "Usage: " << argv[0] << " <protocol> <configfile>" << std::endl;
```

(continues on next page)

(continued from previous page)

```
        exit(-1);
    }

    DataStore datastore = DataStore::connect(argv[1], argv[2]);
    // Get the root of the DataStore
    DataSet root = datastore.root();
    // Create a DataSet
    DataSet example3 = root.createDataSet("example3");
    // Create 5 DataSets in example3
    for(unsigned i = 0; i < 5; i++) {
        std::string datasetName = "sub";
        datasetName += std::to_string(i+1);
        example3.createDataSet(datasetName);
    }
    // Iterate over the child datasets
    // This is equivalent to using begin() and end()
    std::cout << "Datasets in example3: " << std::endl;
    for(auto& dataset : example3) {
        std::cout << dataset.name() << std::endl;
    }

    // access a DataSet by its full name
    DataSet sub2 = root["example3/sub2"];

    // find the sub3 DataSet
    DataSet::iterator it = example3.find("sub3");
    std::cout << it->fullname() << std::endl;

    // lower_bound("sub3") will point to the sub3 dataset
    DataSet::iterator lb = example3.lower_bound("sub3");
    // upper_bound("sub3") will point to the sub4 dataset
    DataSet::iterator ub = example3.upper_bound("sub3");
}
```

The DataSet class presents an interface very similar to that of an `std::map<std::string,DataSet>`, providing users with `begin` and `end` functions to get forward iterators, as well as `find`, `lower_bound`, and `upper_bound` to search for DataSets. DataSets are sorted in alphabetical order when iterating.

1.6 Accessing Runs

The example code bellow shows how to create Runs inside DataSets, how to iterate over all the runs in a DataSet, how to access a Run from a parent DataSet, and how to search for Runs.

main.cpp (show/hide)

```
#include <iostream>
#include <string>
#include <hepnos.hpp>

using namespace hepnos;

```

(continues on next page)

(continued from previous page)

```
int main(int argc, char** argv) {

    if(argc != 3) {
        std::cerr << "Usage: " << argv[0] << " <protocol> <configfile>" << std::endl;
        exit(-1);
    }

    DataStore datastore = DataStore::connect(argv[1], argv[2]);
    // Get the root of the DataStore
    DataSet root = datastore.root();
    // Create a DataSet
    DataSet example4 = root.createDataSet("example4");
    // Create 5 Runs 42 ... 46
    for(unsigned i = 0; i < 5; i++) {
        example4.createRun(i+42);
    }
    // Iterate over the Runs
    std::cout << "Runs:" << std::endl;
    for(auto& run : example4.runs()) {
        std::cout << run.number() << std::endl;
    }

    // access a Run by its number
    Run run43 = example4[43];

    // find the Run 43
    RunSet::iterator it = example4.runs().find(43);
    std::cout << it->number() << std::endl;

    // lower_bound(43) will point to the Run 43
    RunSet::iterator lb = example4.runs().lower_bound(43);
    // upper_bound(43) will point to the Run 44
    RunSet::iterator ub = example4.runs().upper_bound(43);
}
```

The Runs in a DataSets can be accessed using the `DataSet::runs()` method, which produces a `RunSet` object. A `RunSet` is a view of the DataSet for the purpose of accessing Runs.

The RunSet class presents an interface very similar to that of an `std::map<RunNumber,Run>`, providing users with `begin` and `end` functions to get forward iterators, as well as `find`, `lower_bound`, and `upper_bound` to search for specific Runs. Runs are sorted in increasing order of run number.

1.7 Accessing SubRuns

The example code bellow shows how to create SubRuns inside Runs, how to iterate over all the SubRuns in a Run, how to access a SubRun from a Run, and how to search for SubRuns.

main.cpp (show/hide)

```
#include <iostream>
#include <string>
```

(continues on next page)

(continued from previous page)

```
#include <hepnos.hpp>

using namespace hepnos;

int main(int argc, char** argv) {

    if(argc != 3) {
        std::cerr << "Usage: " << argv[0] << " <protocol> <configfile>" << std::endl;
        exit(-1);
    }

    DataStore datastore = DataStore::connect(argv[1], argv[2]);
    // Get the root of the DataStore
    DataSet root = datastore.root();
    // Create a DataSet
    DataSet example5 = root.createDataSet("example5");
    // Create a Run 0
    Run run = example5.createRun(0);
    // Create 5 SubRuns 42 ... 46
    for(unsigned i = 0; i < 5; i++) {
        run.createSubRun(i+42);
    }
    // Iterate over the SubRuns
    std::cout << "SubRuns:" << std::endl;
    for(auto& subrun : run) {
        std::cout << subrun.number() << std::endl;
    }

    // access a SubRun by its number
    SubRun subrun43 = run[43];

    // find the SubRun 43
    Run::iterator it = run.find(43);
    std::cout << it->number() << std::endl;

    // lower_bound(43) will point to the SubRun 43
    Run::iterator lb = run.lower_bound(43);
    // upper_bound(43) will point to the SubRun 44
    Run::iterator ub = run.upper_bound(43);
}
```

The Run class presents an interface very similar to that of an `std::map<SubRunNumber,SubRun>`, providing users with `begin` and `end` functions to get forward iterators, as well as `find`, `lower_bound`, and `upper_bound` to search for specific SubRuns. SubRuns are sorted in increasing order of subrun number.

1.8 Accessing Events

1.8.1 Accessing from a SubRun

The example code bellow shows how to create SubRuns inside Runs, how to iterate over all the SubRuns in a Run, how to access a SubRun from a Run, and how to search for SubRuns.

main.cpp (show/hide)

```cpp
#include <iostream>
#include <string>
#include <hepnos.hpp>

using namespace hepnos;

int main(int argc, char** argv) {

    if(argc != 3) {
        std::cerr << "Usage: " << argv[0] << "<protocol> <configfile>" << std::endl;
        exit(-1);
    }

    DataStore datastore = DataStore::connect(argv[1], argv[2]);
    // Get the root of the DataStore
    DataSet root = datastore.root();
    // Create a DataSet
    DataSet example6 = root.createDataSet("example6");
    // Create a Run 0
    Run run = example6.createRun(0);
    // Create a SubRun 13
    SubRun subrun = run.createSubRun(13);
    // Create 5 Events 42 ... 46
    for(unsigned i = 0; i < 5; i++) {
        subrun.createEvent(i+42);
    }
    // Iterate over the Events
    std::cout << "Events:" << std::endl;
    for(auto& event : subrun) {
        std::cout << event.number() << std::endl;
    }

    // access a Event by its number
    Event event43 = subrun[43];

    // find the Event 43
    SubRun::iterator it = subrun.find(43);
    std::cout << it->number() << std::endl;

    // lower_bound(43) will point to the Event 43
    SubRun::iterator lb = subrun.lower_bound(43);
    // upper_bound(43) will point to the Event 44
    SubRun::iterator ub = subrun.upper_bound(43);
}
```

The SubRun class presents an interface very similar to that of an `std::map<EventNumber,Event>`, providing users with `begin` and `end` functions to get forward iterators, as well as `find`, `lower_bound`, and `upper_bound` to search for specific Events. Events are sorted in increasing order of event number.

1.8.2 Accessing from a DataSet

Events are stored in SubRuns, hence they can be accessed from their parent SubRun, as shown above. They can also be accessed directly from their parent DataSet, providing a more convenient way of iterating through them without having to iterate through intermediate Run and SubRun levels.

The following example code shows how to use the `DataSet::events()` method to get an `EventSet` object.

main.cpp (show/hide)

```cpp
#include <iostream>
#include <string>
#include <hepnos.hpp>

using namespace hepnos;

int main(int argc, char** argv) {

    if(argc != 3) {
        std::cerr << "Usage: " << argv[0] << " <protocol> <configfile>" << std::endl;
        exit(-1);
    }

    DataStore datastore = DataStore::connect(argv[1], argv[2]);
    // Get the root of the DataStore
    DataSet root = datastore.root();
    // Create a DataSet
    DataSet example7 = root.createDataSet("example7");
    // Create 5 Runs with 5 SubRuns with 5 Events
    std::cout << "Creating Runs, SubRuns, and Events" << std::endl;
    for(unsigned i=0; i < 5; i++) {
        auto run = example7.createRun(i);
        for(unsigned j=0; j < 5; j++) {
            auto subrun = run.createSubRun(j);
            for(unsigned k=0; k < 5; k++) {
                auto event = subrun.createEvent(k);
            }
        }
    }
    // Iterate over the events directly from the example7 DataSet
    std::cout << "Iterating over all Events" << std::endl;
    for(auto& event : example7.events()) {
        SubRun subrun = event.subrun();
        Run run = subrun.run();
        std::cout << "Run " << run.number()
                  << ", SubRun " << subrun.number()
                  << ", Event " << event.number()
                  << std::endl;
    }
```

(continues on next page)

(continued from previous page)

```
    // Iterate target by target
    std::cout << "Iterating over all Events target by target" << std::endl;
    unsigned numTargets = datastore.numTargets(ItemType::EVENT);
    for(unsigned target = 0; target < numTargets; target++) {
        std::cout << "Target " << target << std::endl;
        for(auto& event : example7.events(target)) {
            SubRun subrun = event.subrun();
            Run run = subrun.run();
            std::cout << "Run " << run.number()
                      << ", SubRun " << subrun.number()
                      << ", Event " << event.number()
                      << std::endl;
        }
    }
}
```

The EventSet object is a view of all the Events inside a give DataSet. It provides `begin` and `end` methods to iterate over the events.

The `DataSet::events()` method can accept an integer argument representing a given *target number*. The available number of targets can be obtained using `DataStore::numTargets()`, passing `ItemType::EVENT` to indicate that we are interested in the number of targets that are used for storing events. Passing such a target number to `DataSet::events()` will restrict the view of the resulting EventSet to the Events stored in that target. This feature allows parallel programs to have distinct processes interact with distinct targets.

Note that the Events in an EventSet are not sorted lexicographically by (run number, subrun number, event number). Rather, the EventSet provides a number of guarantees on its ordering of Events:

- In an EventSet restricted to a single target, the Events are sorted lexicographically by (run number, subrun number, event number).
- All the Events of a given SubRun are gathered in the same target, hence an EventSet restricted to a single target will contain *all* the Events of *a subset* of SubRuns of *a subset of Runs*.
- When iterating through an EventSet (whether it is restricted to a specific target or not), we are guaranteed to see all the Events of a SubRun before another SubRun starts.

In the above sample program, iterating over the global EventSet yields the same result as iterating over restricted EventSet by increasing target number.

This EventSet feature can be useful if one wants to have *N* clients iterate over all the events in a given dataset. Each client can retrieve events from a single or a subset of targets that way. However, we encourage the reader to consider using the *ParallelEventProcess* class in this situation, as it also provides load-balancing across clients.

1.9 Creating and accessing Products

DataSets, Runs, SubRuns, and Events can store *Products*. A Product is an instance of any C++ object. Since the mechanism for storing and loading products is the same when using DataSets, Runs, SubRuns, and Events, the following code sample illustrates only how to store products in events.

main.cpp (show/hide)

```
#include <iostream>
#include <string>
```

(continues on next page)

(continued from previous page)

```
#include <hepnos.hpp>
#include <boost/serialization/string.hpp>
#include <boost/serialization/vector.hpp>

using namespace hepnos;

struct Particle {

    std::string name;
    double x, y, z;

    Particle() {}

    Particle(const std::string& name, double x, double y, double z)
    : name(name), x(x), y(y), z(z) {}

    template<typename Archive>
    void serialize(Archive& ar, const unsigned int version) {
        ar & name;
        ar & x;
        ar & y;
        ar & z;
    }
};

int main(int argc, char** argv) {

    if(argc != 3) {
        std::cerr << "Usage: " << argv[0] << " <protocol> <configfile>" << std::endl;
        exit(-1);
    }

    DataStore datastore = DataStore::connect(argv[1], argv[2]);
    // Get the root of the DataStore
    DataSet root = datastore.root();
    // Create a DataSet, a Run, a SubRun, and an Event
    DataSet example8 = root.createDataSet("example8");
    Run run = example8.createRun(1);
    SubRun subrun = run.createSubRun(4);
    Event event = subrun.createEvent(32);
    // Store a product into the event
    {
        Particle p("electron", 3.4, 4.5, 5.6);
        ProductID pid = event.store("mylabel", p);
    }
    // Reload a product from the event
    {
        Particle p;
        bool b = event.load("mylabel", p);
        if(b) std::cout << "Particle loaded succesfully" << std::endl;
        else  std::cout << "Particle wasn't loaded" << std::endl;
    }
```

(continues on next page)

(continued from previous page)

```
    // Store a section of a vector into the event
    {
        std::vector<Particle> v;
        for(unsigned i=0; i < 5; i++) {
            v.emplace_back("electron", i*4, i*2, i+1);
        }
        // store only the sub-vector [1,3[ (2 elements)
        event.store("myvec", v, 1, 3);
    }
    // Load the vector
    {
        std::vector<Particle> v;
        event.load("myvec", v);
        std::cout << "Reloaded " << v.size() << " particles" << std::endl;
    }
}
```

In this example, we want to store instances of the Particle class. For this, we need to provide a serialization function for Boost to use when serializing the object into storage.

We then use the `Event::store()` method to store the desired object into the event. This method takes a *label* as a first argument. The pair *(label, product type)* uniquely addresses a product inside an event. It is not possible to overwrite an existing product. Hence multiple products of the same type may be stored in the same event using different labels. The same label may be used to store products of different types in the same event.

The second part of the example shows how to use the vector storage interface. In this example, the `Event::store` function is used to store a sub-vector of the vector *v*, from index 1 (included) to index 3 (excluded). The type of product stored by this way is `std::vector<Particle>`. Hence it can be reloaded into a `std::vector<Particle>` later on.

1.10 Optimizing accesses

Creating and accessing millions of Runs, SubRuns, or Events can hace a large performance impact. Hence, multiple optimizations are available to speed them up.

1.10.1 Batching writes

The creation of Runs, SubRuns, and Events, as well as the storage of data products can be batched. The following code sample illustrates how to use the `WriteBatch` object for this purpose.

main.cpp (show/hide)

```
#include <iostream>
#include <string>
#include <hepnos.hpp>
#include <boost/serialization/string.hpp>
#include <boost/serialization/vector.hpp>

using namespace hepnos;

struct Particle {

```

(continues on next page)

(continued from previous page)

```
    std::string name;
    double x, y, z;

    Particle() {}

    Particle(const std::string& name, double x, double y, double z)
    : name(name), x(x), y(y), z(z) {}

    template<typename Archive>
    void serialize(Archive& ar, const unsigned int version) {
        ar & name;
        ar & x;
        ar & y;
        ar & z;
    }
};

int main(int argc, char** argv) {

    if(argc != 3) {
        std::cerr << "Usage: " << argv[0] << " <protocol> <configfile>" << std::endl;
        exit(-1);
    }

    DataStore datastore = DataStore::connect(argv[1], argv[2]);
    // Get the root of the DataStore
    DataSet root = datastore.root();
    // Create a DataSet
    DataSet example11 = root.createDataSet("example11");
    // Create a Run, a SubRun, and an Event, but delay
    // the actual creation using a WriteBatch
    {
        WriteBatch batch(datastore);
        Run run = example11.createRun(batch, 1);
        SubRun subrun = run.createSubRun(batch, 4);
        Event event = subrun.createEvent(batch, 32);
        // Store a product into the event
        Particle p("electron", 3.4, 4.5, 5.6);
        ProductID pid = event.store(batch, "mylabel", p);
        // The batch is flushed at the end of the scope
    }
    // Reload a product from the event
    {
        auto run = example11[1];
        auto subrun = run[4];
        auto event = subrun[32];
        Particle p;
        bool b = event.load("mylabel", p);
        if(b) std::cout << "Particle loaded succesfully" << std::endl;
        else  std::cout << "Particle wasn't loaded" << std::endl;
    }
}
```

The WriteBatch object is initialized with a datastore. A second argument, `unsigned int max_batch_size` (which defaults to 128), can be provided to indicate that at most this number of operations may be batched together. When this number of operations have been added to the batch, the batch will automatically flush its content. The WriteBatch can be flushed manually using `WriteBatch::flush()`, and any remaining operations will be flushed automatically when the WriteBatch goes out of scope.

The WriteBatch object can be passed to `DataSet::createRun`, `Run::createSubRun`, `SubRun::createEvent`, as well as all the `store` methods.

Note: The `max_batch_size` doesn't represent the total number of items that have to be written to trigger a flush. The WriteBatch internally keeps as many batches of key/value pairs as the number of underlying databases, each batch with its own limit of `max_batch_size`. Hence if `max_batch_size` is 128 and the client has written 254 items, 127 of which will go into one database and 127 other will go into another database, the WriteBatch won't automatically flush any of these batches until they reach 128.

1.10.2 Prefetching reads

Prefetching is a common technique to speed up read accesses. Used alone, the Prefetcher class will read batches of items when iterating through a container. The following code sample examplifies its use.

main.cpp (show/hide)

```
#include <iostream>
#include <string>
#include <hepnos.hpp>
#include <boost/serialization/string.hpp>
#include <boost/serialization/vector.hpp>

using namespace hepnos;

struct Particle {

    std::string name;
    double x, y, z;

    Particle() {}

    Particle(const std::string& name, double x, double y, double z)
    : name(name), x(x), y(y), z(z) {}

    template<typename Archive>
    void serialize(Archive& ar, const unsigned int version) {
        ar & name;
        ar & x;
        ar & y;
        ar & z;
    }
};

int main(int argc, char** argv) {

    if(argc != 3) {
```

(continues on next page)

(continued from previous page)

```
        std::cerr << "Usage: " << argv[0] << " <protocol> <configfile>" << std::endl;
        exit(-1);
    }

    DataStore datastore = DataStore::connect(argv[1], argv[2]);

    // Get the root of the DataStore
    DataSet root = datastore.root();
    // Create a DataSet
    DataSet example12 = root.createDataSet("example12");
    // Create a Run, a SubRun, and many Events
    Run run = example12.createRun(1);
    SubRun subrun = run.createSubRun(4);
    for(unsigned i = 0; i < 20; i++) {
        Event event = subrun.createEvent(32+i);
        // Store a product into the event
        Particle p("electron", 3.4+i, 4.5+i, 5.6+i);
        ProductID pid = event.store("mylabel", p);
    }
    // Reload using a Prefetcher
    Prefetcher prefetcher(datastore);
    // Enable loading Particle objects associated with the label "mylabel"
    prefetcher.fetchProduct<Particle>("mylabel");
    // Loop over the events in the SubRun using the Prefetcher
    for(auto& event : prefetcher(subrun)) {
        Particle p;
        bool b = event.load(prefetcher, "mylabel", p);
        if(b) std::cout << "Particle loaded succesfully" << std::endl;
        else  std::cout << "Particle wasn't loaded" << std::endl;
    }
}
```

The Prefetcher object is initialized with a DataStore instance, and may also be passed a `unsigned int cache_size` and `unsigned int batch_size`. The cache size is the maximum number of items that can be prefetched and stored in the prefetcher's cache. The batch size is the number of items that are requested from the underlying DataStore in a single operation.

A Prefetcher instance can be passed to most functions from the RunSet, Run, and SubRun classes that return an iterator. This iterator will then use the Prefetcher when iterating through the container. The syntax illustrated above, passing the subrun to the `Prefetcher::operator()()` method, shows a simple way of enabling prefetching in a modern C++ style `for` loop.

By default, a Prefetcher will not prefetch products. To enable prefetching products as well, the `Prefetcher::fetchProduct<T>(label)` can be used. This method *does NOT load any products*, it tells the Prefetcher to prefetch products of type T with the specified label as the iteration goes on. The `load` function that is used to load the product then needs to take the prefetcher instance as first argument so that it looks in the prefetcher's cache first rather than in the datastore.

Important: If prefetching is enabled for a given product/label, it is expected that the client program consumes the prefetched product by calling `load`. If it does not, the prefetcher's memory will fill up with prefetched products that are never consumed.

1.10.3 Using asynchronous operations

Most of the operations on Runs, SubRuns, and Events, as well as Prefetcher and WriteBatch, can be turned asynchronous simply by using an `AsyncEngine` instance. The following code examplifies how.

main.cpp (show/hide)

```
#include <iostream>
#include <string>
#include <hepnos.hpp>
#include <boost/serialization/string.hpp>
#include <boost/serialization/vector.hpp>

using namespace hepnos;

struct Particle {

    std::string name;
    double x, y, z;

    Particle() {}

    Particle(const std::string& name, double x, double y, double z)
    : name(name), x(x), y(y), z(z) {}

    template<typename Archive>
    void serialize(Archive& ar, const unsigned int version) {
        ar & name;
        ar & x;
        ar & y;
        ar & z;
    }
};

int main(int argc, char** argv) {

    if(argc != 3) {
        std::cerr << "Usage: " << argv[0] << " <protocol> <configfile>" << std::endl;
        exit(-1);
    }

    DataStore datastore = DataStore::connect(argv[1], argv[2]);

    // Get the root of the DataStore
    DataSet root = datastore.root();
    // Create a DataSet
    DataSet example13 = root.createDataSet("example13");
    {
        AsyncEngine async(datastore,1);
        // Create a Run, a SubRun, and many Events
        Run run = example13.createRun(async, 1);
        SubRun subrun = run.createSubRun(async, 4);
        for(unsigned i = 0; i < 20; i++) {
            Event event = subrun.createEvent(async, 32+i);
```

(continues on next page)

(continued from previous page)

```
            // Store a product into the event
            Particle p("electron", 3.4+i, 4.5+i, 5.6+i);
            ProductID pid = event.store(async, "mylabel", p);
        }
    }
    // Reload using a Prefetcher and AsyncEngine
    {
        Run run = example13[1];
        SubRun subrun = run[4];

        AsyncEngine async(datastore, 1);
        Prefetcher prefetcher(async);
        // Enable loading Particle objects associated with the label "mylabel"
        prefetcher.fetchProduct<Particle>("mylabel");
        // Loop over the events in the SubRun using the Prefetcher
        for(auto& event : prefetcher(subrun)) {
            Particle p;
            bool b = event.load(prefetcher, "mylabel", p);
            if(b) std::cout << "Particle loaded succesfully" << std::endl;
            else  std::cout << "Particle wasn't loaded" << std::endl;
        }
    }
}
```

The AsyncEngine object is initialized with a DataStore instance and a number of threads to spawn. Note that using 0 threads is perfectly fine since the AsyncEngine turns all communication operations into non-blocking operations, the lack of background threads will not prevent the AsyncEngine from being able to make some amount of progress in the background.

The AsyncEngine object can be passed to `DataSet::createRun`, `Run::createSubRun`, `SubRun::createEvent`, as well as all the `store` methods. When used, these operations will be queued in the AsyncEngine and eventually execute in the background.

The AsyncEngine instance can also be passed to the constructor of WriteBatch and Prefetcher. When used with a WriteBatch, the AsyncEngine will continually take operations from the WriteBatch, batch them, and execute them. Hence the batches issued by the AsyncEngine may be smaller than the maximum batch size of the WritBatch object.

When used with a Prefetcher, the Prefetcher will prefetch asynchronously using the AsyncEngine's threads.

1.11 Under the hood

This section explains how HEPnOS organizes its data internally. It is not necessary to read this to use HEPnOS, and this internal orhanization is subject to change. This section is targetted at users who want to have more technical understanding of the inner workings of HEPnOS.

HEPnOS uses Yokan, Mochi's key/value storage service, to store its data. A HEPnOS service hence consists of a collection of remotely-accessible key/value storage databases, each of which will store only one kind of object (DataSet, Run, SubRun, Event, or Product).

1.11.1 DataSet databases

DataSet-storing databases store the full name of the dataset (e.g. *aaa/bbb/ccc*) as key, prefixed with a single byte that represents the level of the dataset (e.g. *aaa* has a level of 0, *aaa/bbb* has a level of 1, *aaa/bbb/ccc* a level of 2, and so on). These keys are sorted lexicographically. The initial byte ensures that all the root datasets are listed first, then the datasets with 1 nesting level, and so on, which simplifies HEPnOS' iterations over datasets (to list the child datasets of *aaa/bbb* for instance, we list all the keys that start with *[2]aaa/bbb*, where *[2]* is a single byte containig the value 2). This also means that the level of nesting cannot exceed 255.

Each dataset key is associated with a unique UUID value. This UUID is generated when the dataset is created, and is used when forming keys for Runs, SubRuns, Events, and Products.

Assuming the HEPnOS service has multiple DataSet databases, this key/value pair will be stored in one of them, selected by consistent hashing of the key.

1.11.2 Run databases

Databases storing Runs use keys formed by concateting the parent DataSet's UUID and the 64-bit Run's number in big-endian format. Using DataSet UUIDs rather than the DataSet's name allows for fixed-size keys (24 bytes). The keys are sorted lexicographically, which enables easy iterations. For instance, to list all the Runs in a given DataSet, we simply query the list of keys that start with that DataSet's UUID. The fact that the big-endian format is used ensures that the database can simply sort the keys by looking at its bytes, with having to reinterpret them as UUID+integer.

These databases do not associate keys with any values (though they could in the future, e.g. to associate metadata with runs).

Assuming the HEPnOS service has multiple Run databases, the database that will store a given Run is determined via consistent hashing of the UUID part of the key (not the Run number part). This leads to all the Runs belonging to the same DataSet ending up in the same database, which simplifies iterations and search.

1.11.3 SubRun databases

Databases storing SubRuns use keys formed by concatenating the parent DataSet's UUID, the parent Run number, and the SubRun number, both in big-endian format, leading to fixed-sized keys (32 bytes), sorted lexicographically.

Like for Runs, these databases do not associate keys with any values.

Assuming the HEPnOS service has multiple SubRun databases, the database that will store a given SubRun is determined via consistent hashing of the UUID + Run number part of the key (but not the SubRun number part). This leads to all the SubRuns belonging to the same Run ending up in the same database, which simplifies iterations and search.

1.11.4 Event databases

Event databases follow the same principle as Run and SubRun databases, using 40-byte keys formed of the UUID, Run number, SubRun number, and Event number, sorted, with no associated value.

Assuming the HEPnOS service has multiple Event databases, the database that will store a given Event is determined via consistent hashing of the UUID + Run number + SubRun number part of the key (but not the Event number part). This leads to all the Events belonging to the same SubRun ending up in the same database.

1.11.5 Product databases

Products are stored in databases with a key of the form `[item][label]#[type]`, where `[item]` is a 40-byte representation of its container (UUID + Run number + SubRun number + Event number; some of these numbers can be set to InvalidRunNumber, InvalidSubRunNumber, and InvalidEventNumber respectively if the product is contained in a DataSet, Run, or SubRun). `[label]` is the user-provided string label associated with the Product. `[type]` is the name of the C++ type of the Product.

This database is also sorted lexycographically. Hence all the products belonging to the same container are next to each other in the database, and within a given container, products with the same label are also next to each other.

Assuming the HEPnOS service has multiple Product databases, the database that will store a given Product is determined via consistent hashing of the UUID + Run number + Subrun part of the key (same as for Events). Hence all the products associated with the same Event, regardless of their type or label, will be stored in the same database, and all the products associated with Events in the same SubRun will be stored in the same database. This strategy is used to improve batching opportunities.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search