Real-world HDF5: applications in finance

Ivan Smirnov
Financial data

— External (captured):
  — Market data
  — Reference data

— Internal (generated):
  — Computation results
  — Pipeline caches
  — Preprocessed data
Market data

— Native market data (event-level)
— Market data from external providers
— Reference/static data
— Captured and stored daily
— Most downstream tasks use market data
Native market data

— Raw packets captured at the exchange
— Each exchange has its own protocol
— Message-based, nested, not tabular
— ... but can be tabularized with a bit of work
— Can be used to "replay" the market
Data normalization

— Direct approach to replaying market data:
  — Decode the raw packet stream for a particular day
  — A proper parser has to be used for that exchange/day
  — Feed decoded packets to exchange-specific book builder
— Alternatively:
  — Decode packet stream, convert to format-agnostic HDF5
  — Use book builder to generate exchange-agnostic HDF5
  — Can build applications on top that don't have to worry about all the low-level details
Market data and HDF5

- Raw data / format-agnostic / exchange-agnostic
- Hierarchy of exchanges, products, dates, etc
- Attributes to store metadata
- Write speed doesn't matter (to jobs)
- Read speed *does* matter (to users)
- Very compressable with shuffling (e.g. ~10x)
- 1-10GB/day/stream compressed HDF5
Internal data and HDF5

— Storing and sharing source data and results
— HDF5-based cache for computation pipelines
— Storing structured application logs
— Single data format to rule them all
Why HDF5?

— Cross-platform, cross-language
— Self-contained and schema-less
— Great Python support (h5py)
— Awesome compression (blosc)
— Low entry barrier
— Reasonably fast reads
Too many fields?..

If you try to create a dataset with a few 1000s of fields:

Unable to create dataset (object header message is too large)

Can anything be done about the 64K limit?..
Structured types in C++: simpler API?..

- Serializing/reading arrays of C/C++ structs requires manually creating CompType at runtime
- Downstream users need to do that as well
- Gets much worse with nested structs/classes
- Special types (enums) need to be mapped manually
- Lots of boilerplate, not very scalable
Simplified C++ hack interface with type mapping:

// shape.h

enum class Colour : uint8_t {
    Red = 1,
    Blue = 2
};

H5_DEFINE_ENUM_TYPE(Colour, Red, Blue)

struct Shape {
    uint32_t n_edges;
    int8_t label;
    double weight;
    Colour colour;
};

H5_DEFINE_COMPOUND_TYPE(Shape, n_edges, label, weight, colour)
Can be now used as:

```cpp
#include <shape.h>

... // write:
    std::vector<Shape> shapes;
    // ...
    auto file = H5::H5File("shapes.h5", H5F_ACC_TRUNC);
    h5::write_array(file, "shapes", shapes);

    // read:
    std::vector<Shape> shapes = h5::read_array(file, "shapes");
```

How it works:

- Macros generate specializations for `h5::type_descriptor<T>`
- Built-in specializations for all primitive types
- Downstream code can make use of upstream specializations
Multi-threaded reading/writing?..

— Anything simpler than MP/MPI for parallel access?..
— Multi-process columnar access is not fun; may result in lots of copying
— A stripped-down *multi-threaded read-only* version of the library?..
— Writing (logging) to different files in multiple threads?..
Faster metadata lookup for partitioned data?..

— For highly partitioned data (10Ks of datasets), metadata lookup/access time becomes noticeable
— Repacking metadata in larger blocks doesn't help
— What's the idiomatic way of dealing with this?..
Blosc support in h5py?..

— Blosc = the best compression filters (👍)
— h5py = the most popular HDF5 interface (👍)
— No easy way to link both (🤔)
— h5py/#611 (2015) - most commented/upvoted issue
— Can c-blosc be shipped with h5py?.. HDF5?..
— venv/conda-friendly HDF5 plug-in discovery system?