

Extending STORM by Adding Extractors and Indexes

Adding a New Extractor

A new extractor is created by subclassing `Extractor`, which is defined in `Extractor.h`. You must implement the pure virtual methods:

```
virtual int firstTuple(Tuple & tup, int & error) = 0;  
virtual int nextTuple(Tuple & tup, int & error) = 0;
```

The objective of each of these methods is to “fill in” the tuple object passed in by reference. These methods return 0 if a tuple can be returned and nonzero if the end of the set of tuples has been reached. A nonzero return indicates that no tuple was filled in, but it doesn't mean an error occurred. If an actual error occurred, you should set the error parameter to nonzero. The error parameter coming in will be set to zero, so you don't need to write to it at all, unless an actual error occurred (such as some file or resource does not exist). If a tuple can be returned, the implementation of these two functions must create the tuple by filling in the ``tup'` reference parameter in one of several ways:

1. Filling in the raw `char *` buffer returned via ``tup.getBuffer()'`. In this case, you should expect that the raw `char*` buffer will be laid out in memory as the dataset has been defined in the dataset schema. In other words, if you have your dataset defined as

```
[ABDATA]  
A = INT2  
B = UINT4
```

then you should copy the value for A into bytes 0 and 1 (in 0-based indexing) of the raw buffer and the value for B into bytes 2-5. You will note that each tuple in the ABDATA dataset is 6 bytes long.

2. Filling in individual Attributes via ``tup.getAttr()'`
3. Filling in individual Attributes via ``tup.getAttrValue()'`

Using either 2 or 3 is a more programmatic way of doing things. In this way, you can set particular attribute values by name. For example:

```
Types::uint4 someval = 19;  
tup.getAttr("B")->setValue(&someval);
```

You probably also want to implement the method:

```
virtual int readChunk(ChunkInfo * ci);
```

which is your opportunity to read from the `ChunkInfo` object the attributes that describe the chunk you are supposed to fetch tuples out of.

Normally, you must also implement the following virtual methods to help break your dataset into chunks (which helps to pipeline the entire STORM process) during the process of index creation and during normal querying when an index is not present:

```

virtual int firstChunkForIndexing(ChunkInfo & chunkInfo,
                                int & error);
virtual int nextChunkForIndexing(ChunkInfo & chunkInfo,
                                int & error);

```

A “chunk” represents a portion of a dataset, larger than a single tuple but smaller than the entire dataset. By using “chunks,” STORM can be built as a pipeline, where chunks of already extracted data can be passed to filtering and the clients while additional chunks are still being extracted from the data source. A chunk in STORM is implemented as a set of key/value pairs which describe the chunk. Each extractor can decide on which key/value pairs are used, however in many cases the default will be sufficient. The chunks need to be described by the extractor, as the extractor knows the file format it is associated with.

Normally your extractor will describe its chunks to its index using the default IEP (Index-Extractor Protocol). This means that the above two methods will need to set the following key/value pairs in the chunkInfo passed to them:

- FILESETID (INT4): Each extractor exists for one filesetID (corresponding to a Data-X line in the DatasetList). This needs to be set, so that the Index (which may reside on a separate host) knows which fileset this chunk comes from.
- FILESETIDHPIDX (INT4): If a data entry is using partitions (i.e. there is more than 1 file listed in the data entry), the option is to either use horizontal (knife cutting through the virtual table horizontally) or vertical partitions. If horizontal partitioning is used, then a given filesetID covers tuples from more than one file, each of which contains whole tuples; in this case, the variable FILESETIDHPIDX is used to describe which file in the fileset is referred to. The original purpose reason to introduce the FILESETIDHPIDX key/value pair is for datasets with a large number of files; since an extractor filter is launched inside a thread for each data entry, a large number of data entries will result in a large number of threads and simultaneous I/O, causing I/O bottlenecks. Using horizontal partitioning gets around this problem, as the # of data entries can be reduced, with each data entry consisting of multiple files, cutting down on the # of simultaneous threads and I/O requests. The BinaryExtractor built into STORM supports this key/value pair, although some custom extractors may not, as it complicates the extractor development process. If FILESETIDHPIDX is not used in an extractor, it can just be ignored altogether (or always set to 0 or somesuch).
- OFFSET (UINT8): This describes the offset into the current fileset where the chunk begins.
- SIZE (UINT8): This describes the size of the chunk, in bytes.

As an example, let's continue the ABDATA example above. Let's say your extractor decided that the next chunk was going to consist of 20 tuples, beginning at offset 24. The implementation of nextChunkForIndexing might do the following:

```

Types::uint8 offset, size;
Types::int4 zero = 0;
offset = (Types::uint8)24;
size = 20 * _dss->getCompactedSchema()->getFixedSize();

chunkInfo.setAttrValue("FILESETID", &_filesetID);
//_filesetID is a member variable of Extractor that

```

```
chunkInfo.setAttrValue("FILESETIDHPIDX", &zero);
chunkInfo.setAttrValue("OFFSET", &offset);
chunkInfo.setAttrValue("SIZE", &size);
```

The "fixed size" of the schema describes how many bytes each tuple occupies in the file. This field is meaningful because STORM allows "views" of a record in a file, where you only identify certain fields in a file record as being part of the tuple, while the other fields are ignored. With the ABDATA example, the fixed size would be 6, since no views are used. However, if we specified

```
[ABDATA]
A = INT2
B = UINT4
<FixedSize> = 8
```

then the fixed size would be 8, meaning that we have 2 additional bytes each record (at the end of the record in this case) in the file that we are skipping over, since we're not interested in including them in the ABDATA table.

```
virtual Tuple * createTuple();
```

Normally you do not need to override this method, unless you are creating a Tuple that is different (e.g. has different number of attributes) from the dataset's idea of the Tuple. When implementing createTuple() assume that STORM takes control of the memory created and you lose ownership (and thus the responsibility of freeing the memory.)

As an example of adding a new extractor, refer to the sample tarball at

<http://web.bmi.ohio-state.edu/resources/software/storm/STORMVariableLengthRecordExtractor.tar.gz>

Adding a New Index

A new index is created by subclassing Index which is defined in Index.h You must implement the following methods:

```
virtual int firstChunk(ChunkInfo & chunkInfo, int & error)=0;
virtual int nextChunk(ChunkInfo & chunkInfo, int & error)=0;
// by default creates a chunkinfo conforming to IEP protocol
virtual ChunkInfo* createChunkInfo();
```

As an example of adding a new index, refer to RtreeIndex.cpp in the STORM source distribution.

Implementation Using Dynamic Libraries

Extractors and Indexes are loadable from dynamic libraries (typically .so files). Libraries that provide extractors should expose functions named "createExtractor" and "destroyExtractor". Libraries which provide indexes should expose functions named "createIndex" and "destroyIndex". It is possible to provide both indexes and extractors in the same dynamic library, since the creation and destroy functions have different names. These functions have the following signatures:

```
extern "C" Extractor *createExtractor(char *name,
    DatasetSchema *dsd, Dataset *ds, TupleSpec *outtuplespec);
extern "C" Index *createIndex(char *name, Query *q);
extern "C" void destroyIndex(Index * i);
extern "C" void destroyExtractor(Extractor * e);
```

As an example of using a dynamic library to create indexes and extractors, refer to `builtinIE.cpp` in the STORM source distribution.

Compiling

You will need to build a shared library for use by STORM. On linux using g++, you would do this by:

```
g++ -fPIC -shared -o <libraryname> <source-list>
```

You will need to add to your default include search path so that the DataCutter headers and STORM headers are found by g++. These flags will be sufficient after replacing the place holders with the full paths:

```
-I/DATACUTTER-INSTALL-ROOT/include/datacutter/
-I/STORM-INSTALL-ROOT/include/storm
```

You may need to link against `libdc.so`, which is in `/DATACUTTER-INSTALL-ROOT/lib/`, and `libSTORM.so` which is in `/STORM-INSTALL-ROOT/lib/`.

Since STORM is C++ code, you will need to use the same compiler and version of compiler to extend STORM as was used to build STORM in the first place, otherwise there will be binary incompatibilities.

You may also need to set your `LD_LIBRARY_PATH` (on linux) or `LIBPATH` (on AIX) to the directory containing the resulting `.so` file, so that at runtime, STORM can find your extractor library.

Modifying STORM to Use Your Extractors and Indexes

To add your new index, start `stormd` via

```
$ stormd
```

and then in another shell run

```
$ storm add indextype INDEXNAME yourIndexLibLoader.so STORMDefaultIEP
```

Similarly, to add an extractor, run

```
$ storm add extractor EXTRACTORNAME yourExtractorLibLoader.so STORMDefaultIEP
```

If you are developing a new index/extractor pair that speaks a protocol other than the default IEP, you will need to change 'STORMDefaultIEP' in the above two commands, and also issue the following command as a prerequisite before adding the extractor and/or index:

\$ storm add IEP

and interactively add all your new IEPs.