Granite Datasource Tutorial

R. Daniel Bergeron
Philip J. Rhodes
August 31, 2005

1 Introduction

The Granite Scientific Database System (Granite SDB) provides a comprehensive set of classes for accessing multidimensional scientific data organized in a wide variety of formats. An important component of Granite is its support for multidimensional array-based data sets through the *DataSource* class. This document describes the basic functionality supported by the Granite *DataSource* and related classes through a series of programming examples. The goal is to provide a simple introduction to basic functionality; not all options and parameters are described.

Conceptually, the datasource model represents rectilinear topologies using an array. The array axes form an *index space*, and each element of the array contains a single *datum*, which has one or more fields or attributes. Physical datasources may be directly associated with a file or network stream. A composite datasource combines one or more component datasources. For example, the *AttributeJoinDataSource* can join one or more attributes taken from each of several component datasources to produce a single, unified representation of the component datasets. Similarly, the *BlockJoinDataSource* can form a single view of several component datasources by joining their index spaces. These two datasources form the core of support for *multisource* data, in which data is combined from several different sources. The datasource model also supports *adaptive resolution* for cell oriented rectilinear data.

Basic concepts: data not stored in Datasource, Datasource class is abstract, Datum class abstract, avoid user knowledge of internal storage framework while still providing opportunity for efficient access to the data.

Define key classes: DataCollection, Datasource, DataBlock, Datum, Iterators

2 Physical Datasources

The *PhysicalDatasource* class provides the key interface to physical files; it is the first point of access to the data. Granite does not predefine any particular data format for data; the intent is to access the original data as generated by the scientist's application. Consequently, it is necessary to provide a metadata file that describes how that data is organized. The metadata file must describe both *file* and *data* attributes and is written in XML notation; we call these *fdl* files (fdl is an abbreviation for *file definition language*).

2.1 FDL file format

Granite FDL files are defined using XML. The Granite system uses an XML data type definition file (a *dtd* file) that describes the legal input. There are two general categories of metadata: file attributes and data attributes. File attributes define the file name, its array dimensionality and bounds, whether the file is in binary or ascii format, its byte order, the axis ordering, and how the fields in the data set are stored in the file (by point or by attribute). Data attributes describe the data type, and name of each field in the data set.

The following example is a complete specification that describes a 100 by 200 binary file with a single float attribute, called *value*.

The *recordSize* attribute only needs to be specified if there is "padding" data in each record that is not described in the field information. The *fileOffset* field is used if there is header information that precedes the data; it defaults to "0". The *fieldLayout* can be either "byPoint" or "byAttribute"; the default is "byPoint". The *axisOrdering* field describes the order in which the axes are stored where the value gives the order of the axes from slowest changing to fastest changing; the default ordering is "0 1 2 …" which is *row ordering* in 2D (i.e., all the elements of a row are stored contiguously, followed by the next row). The *lower* attributed of the *Bounds* field can be omitted; "0" is the default. Using the defaults, the above example could be specified with the following simpler file.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE FileDescriptor PUBLIC "-//SDB//DTD//EN"  "fdl.dtd">
<FileDescriptor
                fileName= "twoDim.bin"
                fileType= "binary"
        >
  <Bounds upper= "99"/>
  <Bounds upper= "199"/>

<Field fieldName= "value"   fieldType= "float"/>
</FileDescriptor>
```

2.2 Opening a data file

The code segment below creates a *PhysicalDataSource* from an fdl file named *twoDim.xfdl*; the second String argument is an arbitrary name associated with the datasource during execution. Creating the datasource does not actually open the data file or read any data. The data file gets opened when the datasource is *activated*, but no file reads are issued until the program makes an explicit request for data via a *datum* or *subblock* method call.

```
DataSource theData =
               DataSource.createPhysicalDatasource( “twoDim.xfdl”, “ds2d” );
theData.activate();              // activate is analogous to opening a file
```

3 Data access

The data in a datasource can be read a single *datum* at a time or by requesting a rectilinear *subblock* from the datasource. Of course, we need a way to identify which datum or subblock we desire. Each point in a datasource is identified by its array position, which is encapsulated in the *IndexSpaceId* class. The *ISBounds* class uses 2 *IndexSpaceID*'s to define a lower and upper bound of a rectilinear subblock.

3.1 IndexSpaceID class

Each point in a datasource is associated with the position of that point in the multidimensional array in which it is stored. An *n*-dimensional datasource requires *n* indexes to access each data point. Granite supports a variety of forms of index space identifiers; the simplest and most straightforward version is the *IntegerIndexSpaceID* class which simply maintains an *n*-entry array of integers representing the indexes. The code segment below creates a 2d *IndexSpaceID* which identifies position [20,30] in the datasource.

```
IndexSpaceID position = new IntegerIndexSpaceID( 20, 30 );
```

Other specific constructors exist for 1d and 3d index spaces and a generic constructor accepts an integer array argument and can be used for index spaces of any dimensionality.

3.2 ISBounds class

A rectilinear subblock of a datasource can be defined by an instance of the *ISBounds* class using 2 *IndexSpaceID*'s, as shown in the example below which identifies a 10 x 10 block with a lower left corner at [20,30].

```
ISBounds blockDef = new ISBounds( new IntegerIndexSpaceID( 20, 30 ),
                                  new IntegerIndexSpaceID( 29, 39 ));
```

3.3 Datum class and datum method

The *Datum* class is an abstract class with children classes that are designed to handle particular combinations of attribute types in as efficient a manner as possible. For example, if all the attributes are of the same type, it is most efficient to use a homogeneous child of *Datum* based on that type: *DoubleDatum, FloatDatum, IntDatum, ShortDatum,* or *ByteDatum*. The *MixedDatum* can be used to include mixed type attributes in a single datum. Although this can be convenient, there is a potential performance cost. The *MixedDatum* object encapsulates all the attributes in a single Java *ByteBuffer* that requires a conversion from a byte sequence to the desired type (*double, float, int, short,* or *byte*) every time the data value is accessed. In the (unlikely) event that the original data is stored in this form, the overhead may be acceptable. [Support for *MixedDatum* is not yet complete.]

In general, it is desirable to avoid references to the child classes of *Datum.* There are methods of both *DataSource* and *DataBlock* that create a *Datum* object of an appropriate type based on the underlying data format. Such methods also allow the programmer to extract all the attributes of a specific type in a homogeneous *Datum* for that type.

The simplest (but relatively inefficient) mechanism for accessing data in a *DataCollection* (a *DataSource* or a *DataBlock*) is to issue a request for the data of one position at a time using the *datum* method call.

```
DataCollection theData;
// set theData to reference a DataSource or DataBlock
Datum datum = theData.datum( position ); // return datum at the position
```

This *Datum* object contains all of the data fields at the specified location in the index space of the *DataCollection*. The actual data values can be accessed individually as *float, double, int*, *short,*

```
float  fval0 = datum.getFloat( 0 );  // get first attribute as a float
short  sval1 = datum.getShort( 1 );  // get second attribute as a short
```

or *byte* values; they are converted from the internal type as needed.

If it is necessary to access a large number of data points one datum at a time, the *datum* method used above would require construction of a new *Datum* object for each *datum* method invocation. Since Java object construction is expensive, it is desirable in this case to allocate a single *Datum* object and re-use it in successive *datum* method calls as shown below.

```
Datum datum = theData.createDatum(); // create Datum of right type/size
theData.datum( datum, position );    // fill it with values from position
```

3.4 DataBlock class and the subblock method

In general, it is far more efficient to retrieve data from a datasource by extracting multiple data points with each query. The *subblock* method extracts an arbitrary rectilinear set of data points from the datasource in a single operation. The data is returned in an instance of the *DataBlock* class. There several important subclasses of *DataBlock*, including especially *BasicBlock* and *CompositeBlock*. A *BasicBlock* provides the most efficient implementation of *DataBlock* if all of the data contained in the block has the same type. A *CompositeBlock* is composed of *BasicBlocks;* it supports mixed types by organizing the data variates by type in separate *BasicBlocks*. An important aspect of all of the *DataBlock* implementations is that the application program can process the data in a *DataBlock* without being aware of the actual internal storage characteristics. [Most of the following examples are based on the use of *BasicBlock* objects.]

As with the *datum* method, there is a *subblock* method that creates and returns the requested information as a *DataBlock* and a version that stores data into a pre-allocated *DataBlock*.

```
ISBounds  blockBnds = new ISBounds( new IntegerIndexSpaceID( 20, 30 ),
                                    new IntegerIndexSpaceID( 29, 39 ));
Block     bb1       = theData.subblock( blockBnds );  // return a subblock
ISBounds  blkBnds   = new ISBounds( new IntegerIndexSpaceID( 10, 10 ));
Block     bb2       = theData.createBlock( blkBnds );
theData.subblock( bb2, blockBnds );   // fill a subblock passed in
```

Once a *DataBlock* has been filled with data, the application program can extract the data one datum at a time with the *datum* method or a rectilinear subblock at a time with the *subblock* method. [*DataSource* and *DataBlock* are both children of the *DataCollection* class which defines the *datum* and *subblock* methods.]

3.5 Direct access to *DataSource* and *DataBlock* data

Although the *Datum* object is an important conceptual component of Granite and can be a convenient mechanism for data access, it can be more efficient to extract data directly from the *DataCollection* without going through the *Datum* class. It is als often necessary or convenient to extract all or a portion of the data from a *DataSource* or *DataBlock* as a single one-dimensional array. Granite provides a family of convenience functions that facilitate such access, including data conversion if desired.

For multipoint access to multidimensional data sets, the order of the data storage in the returned array is determined by the internal storage associated with the *DataSource* or *DataBlock.* [We may want to have a method that will return the axis order associated with the storage or perhaps a convenience that will transform any one-dimensional array according to an arbitrary axis ordering.] If the desired access type and organization match the internal storage format, these functions will return a reference to the actual data, rather than a copy; this can be a significant performance issue for access to very large data sets.

The methods described below can be applied to either *DataSource* or *DataBlock* objects. Because the result of these operations assemble all the data into a single array, it is possible to get into memory-related performance issues if they are applied to very large objects.

3.5.1 Single attribute access

The application program can extract a single attribute value in any data type at a single point in the IndexSpace or from all points in one operation as shown in the following examples.

```
DataCollection theData = … ;
float fValue = theData.getFloat( isid, attrIndex ); // get a single value
int   iValue = theData.getInt( isid, attrIndex );
short all[]  = theData.getShorts( attrIndex );  // return all vals of 1 attr
```

3.5.2 Multiple attribute access

The application program can extract all the attribute values in any data type for either a single point or for all the values in a *DataCollection*. The single point versions return all the attributes at a specified location in a single array. Data conversion is performed if the internal data types do not match the requested type.

```
int    iValues[] = theData.getInts( isid );
double dValues[] = theData.getDoubles( isid );
```

The multiple point access methods can return the data in either *point-order* or *attribute-order*. Point-order storage stores all of the attributes from a single point contiguously; whereas attribute-order storage stores all the values for a single attribute for the entire data collection contiguously. For example, given a data collection with attributes N, C, and O, point-order storage appears in the array as "NCONCONCO …. NCO" and attribute-order appears as "NN…NCC…COO…O". Multiple point access examples are shown below.

```
float fValues[] = theData.getFloats();          // point order
int   iValues[] = theData.getIntsByAttribute(); // attribute order
```

3.5.3 Values access

There is also a Granite class that encapsulates arbitrary type data as a one-dimensional array that is accessible to the application. It may be convenient for the application to use this class for an intermediate representation. [This cois especially true if we decide to make Values a more powerful tool; for example, we could provide multidimensional access to the data values.] See

```
Values vals = theData.getValues();     // get internal Granite representation
```

the Granite JavaDoc for details.

3.6 Attribute projections

It is often desirable for an application to extract only a subset of the attributes that are stored in a data set. This is called an *attribute projection*. Attribute projections can be applied to *Datum, DataBlock*, and *DataSource* access and can be specified by using *RecordSpec* object.

The *RecordSpec* class allows the application to select a subset of the attributes by providing a simple integer array containing the indexes of the desired attributes. The attribute indexes do not need to be listed in order, which allows the application to reorder as well as select attributes.

```
RecordSpec rs = new RecordSpec( new int[]{2,3,0} ); // select attrs 2,3,0
Datum datum = theData.createDatum( rs ); // create Datum for the attr subset
theData.datum( datum, position, rs );    // attributes 2,3,0 at position
```

4 Iterators

In general scientific data processing relies heavily on some form of iteration over some or all of the data under consideration. Very often the iteration pattern is known in advance of the processing and can be specified in a concise form. This can be extremely important when accessing huge data sets stored on disk since disk access is very slow. The caching and pre-fetching facilities of Granite are most effective when the application code uses Granite iterators to define the iteration pattern. Granite datasource functionality supports both datum iteration and block iteration. Datum iteration is specified using an *ISIterator* object and block iteration is defined using an *ISBoundsIterator*.

4.1 *ISIterator* and datum iteration

The *ISIterator* class extends *IndexSpaceID* so it also identifies a position in the index space. It is instantiated with an *ISBounds* object that defines the set of *IndexSpaceID* values that the iterator should visit. The values are visited in basic traversal through the axes with the right most index varying most frequently and the left most index varying least frequently. There is a constructor that accepts an *AxisOrdering* argument that can be used to modify the default ordering. The code fragment below constructs an *ISIterator* and then uses it to access all the values in a datasource.

```
ISBounds   dsBnds = ds2d.getBounds();
ISIterator iter   = new ISIterator( dsBounds );
for ( iter.init(); iter.valid(); iter.next() )
{
   float nextValue = ds2d.datum( iter ).getFloat( 0 );
   System.out.println( ((IndexSpaceID)iter).toString() + "=" + nextValue );
}
```

The iterator is initialized on construction, so the invocation of *iter.init()* isn't strictly necessary, but the general format of this *for* loop specification is a good model and is very obvious code. There is also a *hasMoreElements()* method that can be convenient for some iterations, but you have to be careful. For example, in this case we cannot replace *iter.valid()* with *iter.hasMoreElements()* since *hasMoreElements()* is *false* when the iterator is at the last element of the bounds.

4.2 *ISBoundsIterator* and block iteration

The *ISBoundsIterator* supports block iteration over a datasource or a data block. The blocks in the iteration can overlap with each other or can have gaps between them. The example below is a simple iteration with non-overlapping blocks that "cover" the datasource (i.e., there are no gaps).

```
ISBounds   iterBlk = new ISBounds( new IntegerIndexSpaceID( 2, 2 )); // 2x2 block
ISBoundsIterator iter   = new ISBoundsIterator( dsBounds, iterBlk );
BasicBlock  bBlock = new BasicBlock( iterBlk );  // make block of proper size
for ( iter.init(); iter.valid(); iter.next() )
{
   ds2d.subblock( bBlock, iter );
   // do something with the data in bBlock
}
```

5 Cell Datasources

6 Caching and prefetching

7 Multisource datasources

8 Configuration

9 RDB system notes

Note: points 1, 3, 4, 8, and 9 are all incorporated into point 10.

1. We need to move more "low-level" data access specifications "up" to the DataBlock level. As we now have it defined, we are forcing the programmer to know a lot about the internal representation of the data in order to extract data values as arrays of the primitive types (getFloatAttributes, for example). I have to check what kind of block it is, and then issue the appropriate call. Furthermore, if the data isn't stored as a float array, but I need it that way, I have to write special code for each situation. The system should be providing that. We should probably implement some subset of all the BasicBlock features that we require all DataBlock implementations to provide. Certainly getFloatAttribute(isid, int attrib) and getFloatAttributes(int attrib). This would leave the getFloatAttributes() for the "reference" version. See point 10.
2. What is the difference between BasicBlock.subblock and BasicBlock.subBasicBlock? The javadoc implies that the difference is copy vs. reference: subBasicBlock *copies*, whereas subblock is a reference. That's much too subtle.
3. Both previous comments raise the specter of copy vs. reference methods – how much overhaul do we need to clean that up?? See points 10 and 11.
4. "getIndexed" method names really should go – or at least we should deprecate them and add alternative versions that work because of the parameter specifications. See point 10.
5. Should we implement getDouble and getInteger methods for Datum? very easy to do.
6. Should we implement DoubleDatum and IntegerDatum classes? should be easy to do.
7. Should we have convenience constructors for ISBounds for 1,2, and 3d similar to those we have for IntegerIndexSpaceID?
8. Should we provide a convenience DataBlock method for returning each attribute as an nD float array (in addition to our current 1-d array)? See point 10.
9. Semantics is a problem for some of the method names: getFloatAttribute really should be getFloat*Value*. Standard terminology talks about Attribute/Value pairs – "attribute" refers to the field (as in carbon or oxygen) that contains a "value" such as 4.5. Our methods are returning values given an attribute (specified as an index). Perhaps we can shorten it even more to getFloat or getShort as we do with Datum; or should we adopt the Java conventions and go to floatValue and intValue? See point 10.
10. It would be nice to standardize the interfaces to the low level explicit data access and type variation for Datum, DataBlock, DataSource, Lattice, etc. I first thought that we might be able to define a single interface specification, but that doesn't work. However, we might be able to have an interface for Indexable or for Indexable plus Lattice. (Makes me think that it would be nice to have a class name for Indexable that implies associated data). I would envision methods like one of the 2 columns below (d => Datum, b => block, s => datasource, l => lattice). In all cases, these methods are intended to be usable for all data formats, so that conversions are done if the internal storage format does not match the requested type. There would have to be other methods specific to BasicBlock

and CompositeBlock that would return references to the raw arrays. I don't know how valuable it would be to follow the Java convention since we would have no methods that exactly match Java's floatValue() method; on the other hand, it might help to reinforce the semantics that the returned values are not necessarily the same as the way the data is stored.

For	*similar to current Granite convention*	*Java convention*
d	float getFloat(int field);	float floatValue(int field);
d	int getInteger(int field);	int intValue(int field);
b,s,l	float getFloat(ISId, field);	float floatValue(ISId, field);
b,s,l	int getInteger(ISId, field);	int intValue(ISId, field);
b,s,l	float getFloat(int index1d, int field);	float floatValue(index1d, field);
b,s,l	int getInteger(int index1d, int field);	int intValue(index1d, field);
b,s,l	float[] getFloats(int field);	float[] floatValues(int field);
b,s,l	float[] getFloats( ); // all fields, point order	float[] floatValues();
b,s,l	float[] getFloats(boolean attrOrder);	float[] floatValues(boolean aOrder)
I'm not sure about the options below, but they might be useful		
b,s,l	float[][] getFloats2d(int field);	float[][] floatValues2d(int field);
b,s,l	float[][][] getFloats3d(int field);	float [][][] floatValues3d(field);
b,s,l	float[][][] getFloats2d();	float[][][] getFloats3d();
b,s,l	n-d could be handled with nd array class?	

11. I'm thinking a bit about the copy vs. reference issue, too. For this set of methods (and maybe others), what if we say that "get" methods return references if possible, but may end up being copies. We can then provide "copy" analogs (if we think that is desirable). Alternatively, we could have versions of the "get" methods that pass in the arrays to be returned; these would always be copy operations. The only problem with this is that it would be nice to have convenient utility methods to allocate the correct size array. This seems to work already for the 1d array cases:

```
float[] array = new float[ block.size() ]; // size() works for single attribute
block.getFloat( field, array );

float[] allFloats = new float[ block.size()*block.numAttributes()  ];
block.getFloats( allFloats );
```

For the higher dimension arrays, it's more complicated. Maybe we don't bother with those. Those versions always require copy operations anyway, so there isn’t much to be gained by pre-allocating the arrays. Also, these operations are also already being applied to (large?) collections of data values, so it’s not clear that there will be so many iterations that we have to get too concerned about Java object creation overhead – at least not now.