About Arrays in Postgres v4.2

The purpose of this note is to give a short overview of the way arrays are implemented, and hopefully to yield enough insight to writers of backend user-define adt code to be able to use direct access to Postgres Multidimensional Arrays.

Arrays in Postgres are implemented in a few compatible ways. If the total size of the array is small enough to be fit on single page, the an array attribute can be stored like any other attribute. However, many arrays useful for scientific computation exceed this limit, so array attributes may be stored in Postgres large objects.

Another enhancement is the facility for rearranging the way the data is stored to maximize access performance, identifying subsections or dimension frequently accessed together. This is called "Chunking", and is the subject of Sunita Sarawagi's dissertation research, and is discussed in much greater detail elsewhere.

All implementations share a common header, which is defined in src/backend/utils/adt/array.h. There are some utility macros defined in this file, for convenient access. Code implementing most array access is found in arrayfuncs.c and arrayutils.c, and the chunking code is in chunk.c (also in the utils/adt directory).

The header is a Postgres variable length object whose initial segment is described by the following C structure (remember, variable arrays don't work in C):

```
typedef struct {
        int     size;
        int     ndim;           /* # of dimensions */
        int     flags;          /* implementation flags */
#ifdef for_show_only
/*      int     dims[ndim];     /* size of each dimension */
/*      int     lbounds[ndim];  /* starting value for each dim */
#endif
} ArrayType;
```

Like any other VARLENA structure in postgres, the 4−byte size field is the total number of bytes to store whatever portion of the array is kept directly as a postgres attribute in the classes themselves. When the array references a large object, the size field does not include the number of bytes contained within the large object. The flags field encodes:

- Whether the array data is stored externally in a large object
- The type of large object, if so.
- Whether the array is chunked.

There is no indication in the header of the underlying type that the array is built on. This must be obtained through the attribute structures.

The common header is followed by a vector of 4-byte integers giving the size of each dimension, followed by a vector of 4-byte integers giving the lower bounds for each dimension. Unlike C, the default lower bound is 1.

In the case where the data is stored in a large object, the vectors of information are followed by a string name giving the Postgres path name of the large object. Otherwise the data directly follows the object.

One dimensional arrays of variable size objects are stored sequentially. Thus, accessing individual elements of an array of text may be time consuming! For fixed size data, the conventions are the same as C−language arrays, with the last index cycling the fastest.

An Example

If you wanted to compute the total number of elements in an array you would write the following C code:

```c
#include "tmp/c.h"
#include "utils/log.h"
#include "utils/palloc.h"
#include "utils/adt/array.h"

int
ArEx(arg)
	ArrayType *arg;
{
	int ndims, *dimp, total;

	ndims = ARR_NDIM(arg);
	dimp = ARR_DIMS(arg);

	for (total = *dimp++; --ndims > 0; )
		total *= dimp++;

	return (total);
}
```