GAVO DaCHS: DirectGrammars and Boosters

Author: Markus Demleitner
Email: gavo@ari.uni-heidelberg.de

Date: 2023-02-09

Contents

Introduction

The import architecture of DaCHS with grammars and rowmakers producing material suitable for SQL INSERT statements is designed to be flexible and as declarative as possible. Its one big drawback is that once you have to ingest more than a couple of million rows (or fewer rows with hundreds of columns) it tends to become slow, leading to ingestion times in excess of hours or even days.

To remedy this, DaCHS supports "boosters", programs that bypass both DaCHS' intestines and SQL INSERT statements, both of which are responsible for quite some overhead. Boosters, in constrast, use C code to fetch data and output binary COPY material to be dumped into the table. The net result are very significant speedups; a factor of 100 is easily attainable.

Of course, there are several downsides. One is that you have to write (and probably debug) C code, and schema changes will become fairly painful, requiring surgery in the C code (a notable exception are direct grammars reading from FITS binary tables; the latter contain sufficient metadata to allow fully automatic code generation in simple cases). Also, direct grammars can only operate on single tables; data descriptors containing more than one `make` cannot have direct grammars. As direct grammars talk to the database engine fairly directly, the table definition must have `onDisk="true"`.

Rowmakers given in `make` elements sitting behind direct grammars are ignored; any manipulations to the data coming in must be made within the C code. It is not an error for rowmakers to be present, though. This lets you test and debug with normal DaCHS grammars and then use a booster for the whole (potentially big) dataset by just commenting out the conventional grammar and commenting in the direct grammar. This is especially useful for table compares (e.g., using `gavo info`) to verify that the booster does the same thing as the conventional grammar/rowmaker combo.

A quick start on using boosters:

1) Replace your data element's grammar with the direct grammar spec, which would look somewhat like this:

```
<directGrammar id="fits" type="fits" cBooster="res/boosterfunc.c"/>
```

2) Generate the booster:

```
gavo mkboost your/rd#fits > res/boosterfunc.c
```

3) Edit res/boosterfunc.c (may be optional for fits boosters)

4) Import your data:

```
gavo imp your/rd
```

In the directGrammar element, the path in the `cBooster` attribute is interpreted relative to the RD's resdir. The type argument says rougly what kind of source you're parsing from. Values allowed here include:

- `col` (the default) – parse from stuff conventionally handled by a `columnGrammar`
- `bin` – parse from data that has fixed-length binary records (this is stuff that a `binaryGrammar` would grok)
- `split` – parse from files that have fields separated by some constant sequence of character (conventionally, these can be parsed by a `reGrammar`)
- `fits` – parse from FITS binary tables (that's what a `fitsTableGrammar` can read).
- `hdf5vaex` – parse from an HDF5 file in VAEX convention (i.e., the table comes as one dataset per column, found at `/table/columns/<colname>/data`.

The `mkboost` subcommand receives a reference of to the `directGrammar` element – that is, the RD id, a hash, and the XML id of the grammar – as an argument.

Booster Source Code

Booster Structure

Once you have generated the booster source, you are free to change it in whatever way you fancy. On schema updates, unfortunately, you will have to merge in changes manually, as we have not found a sensible and general way to preserve arbitrary source changes when (re-)generating a booster. If you have a creative idea how better to separate generated and hand-made code, we're certainly interested. However, as usual: version control helps.

The code generated starts somewhat like this:

```
#include <math.h>
#include <string.h>
#include "boosterskel.h"

#define QUERY_N_PARS 33

enum outputFields {
  fi_localid,             /* Identifier, text */
  fi_pmra,                /* PM (alpha), real */
  fi_pmde,                /* PM (delta), real */
```

QUERY_N_PARS here is the number of columns in the destination table. It is also the number of items in the outputFields enum, which in turn defines the structure of the record written to the database. All this must perfectly match what DaCHS runs to create the table, and hence it is generally unwise to manually modify it. Conversely, again: If you add or remove a column, you will have to replace this part of your booster with what DaCHS generates then.

Within outputFields are the column names in the destination table prepended with a simple fi_. If you only use these names to access fields, cutting and pasting on later schema changes should be fairly painless and safe.

While you should not need to change any of this, you in general have to change the getTuple function. What it looks like strongly depends on the sort of booster you're generating for; even the prototypes of this function depend on the sort of booster.

What is common is that getTuple needs to return a Field array. All boosters declare the return value like this:

```
static Field vals[QUERY_N_PARS];
```

– it needs to be static as a pointer to it is returned from the function; do not rely on anything in there to be stable across function calls, though, as the serialization to COPY material might mess around in that memory. The name vals is expected by, e.g., the F macro and must therefore not be changed.

Field is defined in boosterskel.h somewhat like this:

```
typedef struct Field_s {
  valType type;
  int length; /* ignored for anything but VAL_TEXT and VAL_ARRAY */
  union {
    char *c_ptr;
    unsigned char *arr_ptr;
```

```
    double c_double;
    float c_float;
    int64_t c_int64;
    int32_t c_int32;
    int16_t c_int16;
    int8_t c_int8;
    time_t time;
  } val;
} Field;
```

To see the definition your version of DaCHS has, run `dachs adm dump src/boosterskel.h`. The `type` in there is one of:

```
typedef enum valType_e {
  VAL_NULL,
  VAL_BOOL,
  VAL_CHAR,
  VAL_SHORT,
  VAL_INT,
  VAL_BIGINT,
  VAL_FLOAT,
  VAL_DOUBLE,
  VAL_TEXT,
  VAL_JDATE,  /* a julian year ("J2000.0"); this is stored as a simple double */
  VAL_DATE,   /* date expressed as a time_t */
  VAL_DATETIME, /* date and time expressed as a time_t */
  VAL_ARRAY,  /* pre-serialised by pack*Array, length is # of *bytes* */
} valType;
```

(again, `adm dumpDF` might give you a more recent version).

JDATE is a julian day number to be dumped as a date (rather than a datetime). For other ways to represent dates and datetimes, see below.

You can, and frequently will, fill the `vals` by hand. There are, however, a couple of functions that take care of some standard situations:

- `void linearTransform(Field *field, double offset, double factor)` – changes field in place to `offset+factor*oldValue`. Handles NULL correctly, silently does nothing for anything non-numeric
- `void parseFloatWithMagicNULL(char *src, Field *field, int start, int len, char *magicVal)` – parses a float from a decimal serialisation in `src[start:start+len]` into field, writing NULL when magicVal is found in the field.
- `void parseDouble(char *src, Field *field, int start, int len)` – parses a double from `src[start:start+len]` into field, writing NULL if it's whitespace only.
- `void parseInt(char *src, Field *field, int start, int len)` – parses a 32-bit int from `src[start:start+len]` into field.
- `void parseShort(char *src, Field *field, int start, int len)` – parses a 16-bit int from `src[start:start+len]` into field.
- `void parseBlankBoolean(char *src, Field *field, int srcInd)` – parses a boolean such that field becomes true when src[srcInd] is nonempty.

- `void parseBigint(char *src, Field *field, int start, int len)` – parses a 64-bit int from `src[start:start+len]` into field.
- `void parseString(char *src, Field *field, int start, int len, char *space)` – copies len bytes starting at start from src into space (you are responsible for allocating that; usually, a static buffer should do, since the postgres input is generated before the next input line is parsed) and stuffs the whole thing into field.
- `void parseChar(char *src, Field *field, int srcInd)` – copies the value at `*src` into `field`, making that a char.
- `MAKE_NULL(fi)` – makes the column fi points to NULL
- `MAKE_DOUBLE(fi, value)` – puts a C double `value` into the column fi points to
- `MAKE_BIGINT(fi, value)` – puts a C long long `value` into the column fi points to
- `MAKE_FLOAT(fi, value)` – puts a C float `value` into the column fi points to
- `MAKE_SHORT(fi, value)` – puts a C short `value` into the column fi points to
- `MAKE_CHAR(fi, value)` – puts a C char `value` into the column fi points to
- `MAKE_JDATE(fi, value)` – writes a float/double `value` as a julian date (a postgres timestamp).
- `MAKE_TEXT(fi, value)` – note that you must manage the memory of value yourself. In particular, it must not be automatic memory of getTuple, since that will not be valid when the tuple actually is built. Most commonly, you'll be using a static buffer here.
- `MAKE_CHAR_NULL(fi, value, nullvalue)` – makes fi a char with value unless value==nullvalue; in that case, fi becomes a NULL
- `MAKE_ARRAY(fi, bufptr, lenOfBuf)` – see Arrays.
- `double mjdToJYear(mjd)` – returns a julian year for mjd
- `AS2DEG(field)` – turns a field value in arcsecs to degrees
- `MAS2DEG(field)` – turns a field value in milli-arcsecs to degrees

Of course, you can also manually copy or delimit data and use fieldscanf as documented in split boosters

As said above, `dachs imp` will compile and link your code together with the files `boosterskel.c` and `boosterskel.h` that come with the DaCHS distribution. If you insist on manually building a booster, use:

```
gavo admin dumpDF src/boosterskel.c # or .h
```

to pull these files into your working directory. Watch the build commands DaCHS executes when you change a booster source and run `dachs imp` to figure out the compile and link commands appropriate for your source.

Filling vals Manually

The F(index) macro lets you access the field info directly. So, you could enter a fixed-length piece of memory into fi_magic like this:

```
static char bufForMagic[8];

memcpy(bufForMagic, inputLine+20, 8);
F(fi_magic)->type = VAL_TEXT;
F(fi_magic)->val.c_ptr =  bufForMagic;
F(fi_magic)->length = 8;
```

Having static buffers in getTuple is usually ok since the COPY input is generated before getTuple is called again.
It is quite common to have to handle null values. In the example above, this could look like this if a NULL for magic were signified by a F in inputLine[19]:

```
static char bufForMagic[8];

if (inputLine[19]=='F') {
  F(fi_magic)->type = VAL_NULL;
} else {
  memcpy(bufForMagic, inputLine+20, 8);
  ...
```

Skipping a record

If you need to skip a record, do:

```
longjmp(ignoreRecord)
```

in getTuple. That works independently of the booster type.

Dates and times

The boosters treat "normal" dates and datetimes as struct tm-s. If you need a larger range, use VAL_JDATE, which lets you store julian dates in floats. Julian dates are serialized to dates rather than datetimes.
To parse VAL_DATE or VAL_DATETIME, you will write something like:

```
fieldscanf(curCont, fi_date, VAL_DATE, "%Y-%m-%d");
```

if parsing from date strings. If your input is something weird, figure out a way to generate a struct tm as defined in time.h. Then write:

```
struct tm timeParts;
timeParts.tm_sec = 12;
...
timeParts.tm_year = 1920;
F(fi_dt)->val.time = timeParts;
F(fi_dt).type = VAL_DATETIME;
```

(or VAL_DATE, as the case may be).
Having said all this, long experience has taught us it's ususally best do have dates and such in the database as MJD or julian years. You can format those to ISO strings (or, really, anything else you want) on output by using display hints on outpuField or even column itself.
MJDs are just so much easier to handle within ADQL queries. Support for timestamps, on the other hand, is extremely lousy.

Arrays

Boosters can write arrays, but for now it's a bit rudimentary. If you have ideas for abstracting this a bit, do contact the authors.

Note: If you have boosters still calling `packFloatArray(buf, float_arr, nVals)`, replace that call with `packArray(buf, vals, nVals, 700, sizeof(float), 1)` (or make that final thing a 0 if you're on a big-endian platform).

The central (and mean) function has the signature:

```
size_t packArray(
        unsigned char *buf,  /* the target buffer */
        unsigned char *vals, /* the values to serialise */
        int nVals,           /* number of items in vals */
        int itemoid,         /* postgres UID of the target type */
        size_t itemsize,     /* size of each item in vals */
        int littleEndian     /* have this 1 when incoming data is little-endian */
        ) -> #bytes left in buf.
```

This writes a postgres binary array representation of `nVals` items of length itemsize into `buf` and returns the number of bytes put in there. `buf` needs to hold at least `36+nVals*(4+itemsize)` bytes.

This needs the postgres oid for the the type to be written. These are fairly constant, so here are oids for the more common types (you can figure these out by running `select oid, typname from pg_type`):

bigint	20
smallint	21
integer	23
real	700
double precision	701

Note that `packArray` only re-packs the bytes; hence, `vals` must already contain values appropriate for postgres' type.

The `littleEndian` parameter will be 1 in most common circumstances (dumping native data on x86); it makes `packArray` flip the byte order, which in postgres dumps is always big endian. If you are on a big-endian machine or have big-endian data in memory or have big-endian floats (which may happen on some little-endian architectures, I'm told), make it 0.

This function goes together with the `MAKE_ARRAY(fi, buf, nbytes)` macro that tells postgres that in the referenced field info there's `nbytes` of serialised array material. As with strings, you need to make `buf` live beyond the execution time of `getTuple`, and again we recommend doing that with static arrays. Hence, to write a 4-array collected from four consecutive array items starting at index 7 into a matarr database column, you could have in your getTuple:

```
#define ARRLEN 4
#define ARRBUFSZ (36+ARRLEN*8)

static unsigned char matbuf[ARRBUFSZ];
float tmparr[ARRLEN];  /* does not need to be static */
```

```
size_t tmplen;
...
for (i=0; i<ARRLEN; i++) {
  tmparr[i] = data[7+i][rowIndex];
}
tmplen = packArray(matbuf, tmparr, ARRLEN, 700, sizeof(float), 1);
MAKE_ARRAY(fi_matarr, matbuf, tmplen);
```

Booster Types

Line-based boosters

These are boosters that read from a text file, line by line. The maximumg line length is set by the direct grammar's recordSize attribute.
It is up to the parsing function to split and digest this text line.

Col boosters

For col boosters, the getTuple function looks somewhat like this:

```
Field *getTuple(char *inputLine)
{
  static Field vals[QUERY_N_PARS];

  parseWhatever(inputLine, F(fi_localid), start, len);
  parseFloat(inputLine, F(fi_pmra), start, len);
  parseFloat(inputLine, F(fi_pmde), start, len);
  parseFloat(inputLine, F(fi_raerr), start, len);
```

Here, it's your job to fill out start and len (at least; start is zero-based). `gavo mkboost` inserts parseXXX function calls according to the table metadata, which should be what you want in general. Add scaling or other processing as required.

Split boosters

When the input data comes as xSV (e.g., values separated by vertical bars, commas, or tabs), give a `splitChar` and set the `type` attribute to `split` in the `directGrammar`.
This then creates a source like:

```
 char *curCont = strtok(inputLine, "\t");
fieldscanf(curCont, fi_objid, VAL_INT_64);
curCont = strtok(NULL, "\t");
fieldscanf(curCont, fi_run, VAL_SHORT);
```

etc. Thus, the input line is parsed using strtok, and each value is parsed using the `fieldscanf` function. This function takes the string containing the literal in the first argument, the field index in the second, and finally the type specifier. If the data comes in the sequence of the table columns, the generated source *might* just work.
Warning: C's standard strtok function merges adjacent separators, i.e., `foo|bar||baz` would just yield three tokens, foo, bar, and baz. With astronomical data, this is typically not what you want. Therefore, the generated booster function will have a line like:

```
#define strtok strtok_u
```

Delete it in case that you need the POSIX strtok behaviour. This would in particular apply if you have whitespace separated data with a variable number of blanks (which, however, would suggest that you're really looking at material for a col booster).

Bin boosters

When you get binary data of fixed record length, set the `recordSize` attribute on the `DirectGrammar` element:

```
<directGrammar type="bin" recordSize="300"...
```

Note that a `recordSize` larger than `INPUT_LINE_MAX` will cause a buffer overflow.
You are mainly on your own in terms of segmentation, but for entering values, you can use the `MAKE_*` macros discussed above.
For these in particular, use the the portable type specifiers for integral types, viz., `int8_t`, `int16_t`, `int32_t`, and `int64_t` and these names with a `u` in front.
In particular with binary boosters, it is essential you always properly cast what you read, e.g.,:

```
MAKE_DOUBLE(fi_dej2000, -90+*(int32_t*)(line+4)/1e6); /* SPD */
```

when a declination is given as mas of south polar distance.

FITS boosters

These read from FITS binary tables, which come with enough metadata that it might just be possible to make DaCHS generate a booster working without editing any C code. To build one of those, DaCHS inspects the first file matched by the parent data's `sources` element (which also means these won't work outside of a `data` element). DaCHS where there is a match for a table column after lowercasing the name in the FITS table, DaCHS will automatically add a suitable mapping instruction. FITS table columns without a match in the database table are ignored, table columns without matches in the FITS are set to NULL.
FITS binary tables are organized by columns rather than by rows, bearing witness to their FORTRAN heritage. The way the boosters are currently generated, all these columns are completely read into memory, which means you cannot ingest FITS binary tables that do not fit into your machine's memory. Fixing this would be fairly straightforward (patches are welcome, but we'll also fix this if you ask for it).
In case your column names in the database are different from the FITS input (modulo case), you can use Element mapKeys to inform the direct grammar. For instance, `<mapKeys> raj2000:RA, dej2000:DEC </mapKeys>` will map column named RA in your source file to column named raj2000 in your database table and analogously for DEC. If you don't do this, only column names from your DB table will be read and imported.
If you need to postprocess the items, we recommend you do that again in the `getTuple` function (note how that gets passed the row index) for maintainability, rather than directly after reading the rows.
Attention: The system will not warn you if the type of a column in the table is not compatible with what you have in the database. If it is, the program will probably silently dump garbage into the db, though if you're lucky it'll crash. This is almost on purpose. It will let you do manual type conversions like, for example, making a 64 bit integer from a string as follows:

```
if (nulls[18][rowIndex]) {
  MAKE_NULL(fi_ppmxl);
} else {
```

```
    parseBigint(((char**)(data[18]))[rowIndex], F(fi_ppmxl), 0, 19);
}
return vals;
```

(we could admittedly warn you if this kind of thing becomes necessary, and we'll gratefully accept patches for that).

HDF5 boosters

HDF5 is a fairly complex binary format that allows any number of different serialisations of tables. The Element hdf5Grammar can parse some of these.

For boosters, currently only the vaex dialect is available (ask for others). To use it, select `type="hdf5vaex"` in the direct grammar.

As with FITS, there is some basic metadata in HDF5 files that lets DaCHS infer mappings when HDF5 and column names match or when mapKeys provides some. Hence, again, to generate the booster, you will have to have at least one data file so DaCHS can infer this mapping.

The `getTuple` for hdf5vaex boosters has two arguments: `rowIndex` (for informational purposes) and `data`, which is of type `InRec*`. This type is a record that DaCHS defines for you and that has the input file's columns as names. Hence, in the ideal case, you can directly build the `vals` entries from `data` members, somewhat like this:

```
MAKE_BIGINT(fi_source_id, ((uint64_t)(data->source_id)));
MAKE_FLOAT(fi_a0_best, ((float)(data->A0_best)));
```

– this is what DaCHS generates.

Note that HDF5 does not define portable NULL representations (or if it does, then DaCHS knows nothing about them). Hence, you will have to worry about NULLs yourself.

Debugging

The source code generated by `gavo mkboost` typically is really mean. The preference is to make it coredump rather than give fancy errors, under the assumption that error messages from the booster would in general help less than the post-mortem dumps; this of course also means that you should not use direct grammars to parse from potentially malicious sources unless you substantially harden the generated code.

To figure out what's wrong if things go wrong, say:

```
ulimit -c unlimited  # bash and friends
gavo imp q
gdb bin/booster core
where # that's for gdb
```

This should give you the line where things failed, and of course the full power of gdb to inspect how that happened.

In case things just go wrong without segfaulting or otherwise producing a core dump, figure out a condition you would like to inspect and add code like:

```
#include <stdlib.h>
...

if (condition) {
  abort();
}
```

The core dump will then reflect the situation at the time of the call to abort (well, a few frames up, anyway).

As a short example, consider a gdb session where the author I forgot to use the mapKeys in a FITS directGrammar for columns which are filled from the Binary table. This resulted in a segmentation fault, which made gdb say:

```
gdb:
  Program terminated with signal 11, Segmentation fault.
  #0  0x0000000000406cde in getTuple (data=0x7fff592a41a0, nulls=0x7fff592a4250, rowIndex=0)
```

To figure out where the program crashed, say:

```
(gdb) where
#0  0x0000000000406cde in getTuple (data=0x7fff592a41a0, nulls=0x7fff592a4250,
    rowIndex=0) at func.c:73
#1  0x0000000000407784 in createDumpfile (argc=2, argv=0x7fff592a53c8)
    at func.c:296
#2  0x0000000000406bdf in main (argc=2, argv=0x7fff592a53c8)
    at boosterskel.c:673
```

In the traceback, you can see the frame you're interested in and go there using up (or down, if you're too far up):

```
(gdb) up
#1  0x0000000000407784 in createDumpfile (argc=2, argv=0x7fff592a53c8)
    at func.c:296
296   in func.c
```

Incidentally, you could instruct gdb to use your boosterfunc.c file as the source file for func.c (that's the temporary name of that file when DaCHS built the binary in a sandbox). But it's probably as straightforward to just check the source code in your editor and figure out what variables you're interested in. In this case, this might be the number of the row where the crash happened (we are in the main row-reading loop of the booster):

```
(gdb) print i
$8 = 0
```

Voila, we crashed on the first row already. Let's go back into getTuple to figure out which column was bad:

```
(gdb) down
#0  0x0000000000406cde in getTuple (data=0x7fff592a41a0, nulls=0x7fff592a4250,
    rowIndex=0) at func.c:73
73    in func.c
```

Looking up line 73, there's (in this example) an access to `nulls[0][rowIndex]`. Could this dereference a null pointer? See for yourself:

```
(gdb) print nulls[0]
$9 = 0x0
```

Right – so that's where the trouble starts (in this case, the underlying reason was a DaCHS bug, as that array should never have been uninitialized).