# BDS Data Egress

11 февраля 2016 г.     18:35

## Introduction

BDS has essentially three data egress mechanisms: Download, Provenance and Query. These are described in detail in the following sections. The primary idea is to provide data that scientists care about in a form they can easily use and this involves a certain amount of transition software since the data are stored in the system primarily according to systemic (not scientific) considerations.

## Download

This part is not too complicated. The primary topics to discuss here are:

- A recapitulation of the MATLAB comments: From tables to CSV files.
- UI download

## Provenance

This part is under development (2/16). It concerns the use of Provenance files which enable data download and other things.

## Query

Queries are formulated in the UI and produce things like Dataset lists, Table lists and Tables. The problem we state here is one of science utility: How do we transform a Query result that may have inconvenient attributes into something more useful for scientists?

For this discussion we focus on Row Query results, which are tables. Examples of inconveniences: Such a table might have row degeneracy; or the scientists may want to calculate a simple index as a new column not available in a current level. Here we describe how to implement these sorts of 'in passing' improvements to Query results.

Let's begin with the first example where a Row Query Result (RQR) has row degeneracy: Multiple rows concern the same entity like a chemical formula. We find this in FTICR-MS Level 1.3 queries across multiple Datasets where the same formula will appear in more than one row. Such degenerate rows can be consolidated, for example with the following Python script:

MS14Conso
lidate

Its input is a single table produced by BDS row query to MS 1.3, its outputs are two tables which can be described by following pseudo database requests:

```
SELECT * EXCEPT $DatasetID, $TableID, nrMatches, Index, I FROM query_result
    GROUP BY Formula
    CONSOLIDATE
        mean_mass, Fe, Na, Cl, P, S, O, N, H, C, Meas_m/z -> EXPECT BEING CONSTANT
        * -> EXPECT SINGLE NOT MISSING VALUE
    ADD COLUMNS
        nrMatches = map * (\val -> if isNaN val then 0 else 1) | fold (+)
        I = fold * (+) / nrMatches

SELECT $DatasetID, $TableID, samples FROM query_result
    GROUP BY $DatasetID, $TableID
    CONSOLIDATE
        samples -> filter samples (any (!isNaN)) | count
```

***A question arises: how to incorporate that kind of output modulation into BDS?***

Three rejected ideas:

- Introduce a new Type for each query target.
- Make the query structure and query engine more complex.
- Allow for user-supplied post processors.

Each had merits but we settled on a fourth option, 'built-in postprocessors':

So, bullet 4 it is.

- The admin provides system with algorithms in some standard form (as with TYPE processing), users can specify for each of their queries and which existing post processors (if any) they want to use
    - PRO

        - We can implement processing of **ABSOLUTELY ANY** complexity this way
        - It's relatively easy to implement
        - Easy to use
    - CON
        - Set of available post-processors is fixed at any given moment of time, introduction of a new one requires action from BDS admin

***In which form should admin provide post-processors?***

Do we want to be able to create complete provenances for query results like the ones we have for datasets? If so, then the only possibility is to provide post-processors as Angara workflows (like processors for dataset) and to make query engine utilize Angara for processing.

Rob prefers an approach that is less formal: Provide a tarball or a GitHub URL of the code / documentation attached to a particular post-processor used. I prefer this despite its being a bit schizophrenic: On the one hand we have the provenance machinery of Angara (Modeling Environment) so why not make use of it? On the other hand this is a sort of Uncontrolled Zone outside of A/ME where we might anticipate the work is less formal and more experimental. It still requires admin attention but not at the level of BDS system workflow; so I am proposing a concentric-ring model where the central disc is the formalized A/ME processing of Levels and the outer ring is optional post-processors with less formality and 'use at your own risk'.

Post-processors can be written in the following languages, in order of implementation priority:
- Python 2
- MATLAB
- R
- Python 3
- Scala
- Cuda
- ALGOL 58
- LISP
- 6502 Assembly Language
- HP 67 RPN
- TeX
- F#
- Pascale
- C
- BASIC
- Prolog
- FORTRAN 77
- COBOL