A Command Line Metaphor for Big Data

John Plevyak
jplevyak@vzite.com

June 9, 2010

Abstract

Simplicity and composability are the hallmarks of the familiar unix command line with its concepts of files of text lines or bytes, pipes and processes. A similarly powerful set of concepts for Big Data includes tables of bytes, text lines, Protobuf, Avro and other typed records, dataflow paths and distributed processes. Moreover because of the size and nature of the data as well as the computations involved a metaphor for Big Data benefits from the concept of table views which can be computed lazily, on demand and from non-local data sources. We propose a unix-style interface to data and computations and a set of composible tools. Here we discuss the data model, file system model, execution model and views interspursed with examples drawn from a set of tools for basic operations.

1 Data Model

The unix data model is based on files composed of bytes but which can be treated like a sequence of text lines. While this model works for Big Data as well, a more sophisticated and increasingly popular model defines files as sequences of typed records. These records can be fixed or variable sized and defined using any of a number of popular serializers including protobufs, Avro, json, xml, etc. This model has a number of advantages including 1) composition of higher level typed operators 2) random access by record number 3) placement of complete records on a data node 4) updates of variable sized records and 5) reduced the sematic gap between hadoop data and tradiational database tables. Except for updates, this model can be used with HDFS relatively easily. For emulation on linux and other operating systems, extended attributes are used to store the type information separately from the data so that the disk data need not be changed to accomodate the type information. A key aspect of the data model is that a given data source or sink is homogeneous. An Avro container can contain data under a number of Avro schemas, but in this model it is tagged with a type to which all the data is converted so that it seems to contain homogeneous data from the point of view of a filter program.

Here in Figure 1 we import a local file which starts as bytes as a text file. We then convert it by coercing the text lines to integers and then print the first

```
(vz) cp file:listofnumbers.text l.text text

(vz) cat -t int l.text > foo.t

(vz) type foo.t
int

(vz) cat --limit 3 -t text foo.t
123
345
21
```

Figure 1: Conversion from raw text

3 rows as text.

```
(vz) type foo.t
struct { int a; int b; int c; };

(vz) select a foo.t > bar.t

(vz) type bar
int
(vz) cat --limit 3 text < bar.t
123
345
21
```

Figure 2: Select from C/C++ struct

Figure 2 shows how a generic select program can be used to extract a named column from a table. Note how this program operates as a like a link in a unix pipe chain.

Figure 3 demonstrates coercion via Avro serialization. Serialization frameworks that permit reading of data under an updated schema can be used to convert data in this manner.

2 File Model

A standard file system attempts to produce a consistent single system view even for files which are stored remotely or shared concurrently. Big Data (Hadoop) style distributed computations are somewhat different in that each node has local storage and state. A file system model which takes this into account includes 1) *node files* which are unique to each node 2) *global files* which are

```
(vz) cat --limit 1 -t baz.avro biz.t
{ name = "Joe", value = 12 }

(vz) type biz.t
record A {
  string name;
  long value;
}

(vz) cat baz.avro
record A {
  string name;
  long v2 = 0;
}

(vz) cat --limit 1 -t baz.avro < biz.t
{ name = "Joe", v2 = 0 }
```

Figure 3: Genavaro type conversion

replicated on every node and 3) *tables* which are disributed and which may have records local to a node. These three types of files reflect three different use cases. Node files are useful for logs, machine specific configuration, intermediate results etc. Global files are useful for disributing global configuration, program and libraries files etc. Tables are viewed as present only on nodes which contain at least one records. Files and tables can be interconverted such that, for example, the collected local results of a sort can be promoted to a table. Different types of files can be combined to perform useful collective operations as programs execute on nodes which have access to all input files.

```
(vz) grep Error output.log
```

Figure 4: Grep for errors in log file

The `grep` global program in Figure 4 is executed independently on all nodes which have have an output.log file, the results are combined and set to stdout.

```
(vz) sum /node/memory

(vz) cat /node/memory /node/1
```

Figure 5: Report available memory

The `sum` global program in Figure 5 is executed on all nodes to sum the contents of the node file `/node/memory` which contains the amount of installed memory. Since `/node/1` only exists on the node with id 1, the second command reports only the amount of memory on node 1.

3 Execution Model

In addition to the issues of distributed execution handled by Hadoop, the execution model must handle typed files. The type of the input files can vary and we can view the node code and the output type as a function of the input file types. Execution therefor consists of a staging phase where the "program" is run with the input types as arguments to produce the node code and the output type. For more dynamic languages the node code might be identical for all input types while for more static languages staging might entail compilation against generated serialization/deserialization code and headers. Even for dynamic languages it might make sense in the case of long running jobs to verify type correctness before consuming resources. Programs may also take the output type as an argument and be expected to coerce their output results to match. A trivial example is the "cat" program which will combine all inputs and optionally attempt to convert them to a target type.

```
(vz) program [-x]* arg* inputfile* [outputfile] [< inputfile] [> outputfile]
```

Figure 6: Command conventions

A program takes a set of options and arguments followed input files, an output file and an output file type and redirections (Figure 6). As with unix, per program convetions dictate which elements of the command line are present and how they are interpreted (e.g. in “cat a b” both “a” and “b” are input files while in “cp a b” “a” is an input file and “b” an output file).

The `where` program in Figure 7 examines the type of the input file and produces an identical output type and a program which passes through only records whose `name` field matches `Joe`.

When commands are chained (Figure 8), the programs are staged in order. Commands can be grouped to create anonymous input files.

4 Views

Views are tables are produced on demand. They can be computed like /dev/random, lazily computed from other files/tables, or loaded from external sources. Be-

```
(vz) where 'name == "Joe"' biz.t
```

Figure 7: Where

```
(vz) where 'salary < 20000' biz.t | where 'boss == "Joe"' | count

(vz) cat (where 'salary < 20000' biz.t | where 'boss == "Joe"')
            (where 'salary > 50000' baz.t) | count
```

Figure 8: Chaining and grouping of commands

```
## selected columns

(vz) view bar.t where 'b > 50' foo.t | select a,c

## views of views

(vz) view baz.t where 'a < 2' bar.t

## remote view

(vz) view namerank.t mysql://me:mypasswd@dbserver.mysite.com:9000/mydb/mytable |
       select name,rank

## visualizing views

(vz) view foo.t
foo.t = where 'a < 2' bar.t
  bar.t = select a,c -
    - = where 'b > 50' foo.t
```

Figure 9: Creating Views

cause tables can be so large it make even less sense to copy them or maintain derivative copies than for regular files. However, it is still convenient to map them into a homogenous namespace within a file system so that they can be treated like first class files. This is also a good way to bridge heterogeneous systems without duplication and to encapsulate access credentials and methods securely and map them into the local security model. Any command can be converted into a view. As with file types, views can be emulated on linux and other operating systems using extended attributes.

5 Implementation Issue

Staging is similar to bulding a query plan except that the underlying execution model is imperative rather than declarative. Nevertheless staging can use other execution environment information such as data layout and resource availability and constraints. Another optimization is to push input requirements backward along the data flow to reduce the amount of information carried between nodes. For example, in figure 7, the ”count” operation only needs to know the number of records and the ”where boss..” operation needs to only to have records with the single boss field. Furthermore, all the operations except the final count sum can be done local to the data.