How pandas infers data types when parsing CSV files

Last updated on January 11, 2018, in python

I was always wondering how pandas infers data types and why sometimes it takes a lot of memory when reading large CSV files. Well, it is time to understand how it works.

This article describes a default C-based CSV parsing engine in pandas.

First off, there is a `low_memory` parameter in the `read_csv` function that is set to `True` by default. Instead of processing whole file in a single pass, it splits CSV into chunks, which size is limited by the number of lines. A special heuristic determines the number of lines — `2**20 / number_of_columns` rounded down to the nearest power of two.

Parsing process

The parsing process starts with a tokenizer, which splits each line into fields (tokens). The tokenizing engine does not make any assumptions about the data and stores each column as an array of strings.

The next step involves data conversion. If there is no type specified, pandas will try to infer the type automatically.

The inference module is written in C and Cython, here is a pseudo code of the main logic behind type inference:

```
def try_int64(column):
    result = np.empty(column, dtype=np.uint64)
    na_count = 0

    for i, item in enumerate(column):

        try:
            value = str_to_int64(item)
        except OverflowError:
            raise OverflowError
        except:
            # parsing error, stop the process
            return None, na_count

        if is_nan(value):
            # There is no way to store NaNs in the integer column
            na_count += 1
            return None, na_count

        result[i] = value
    return result, na_count


def convert_column_data(column, dtype):
    result = None

    if is_integer_dtype(dtype):
        try:
            result, na_count = try_int64(column)
        except OverflowError:
            # try unsigned int
            result, na_count = try_uint64(column)

        if result is None and na_count > 0:
            # Integer column has NA values
            return None
    elif is_float_dtype(dtype):
        result, na_count = try_float64(data)
    elif is_bool_dtype(dtype):
        result = try_bool(data)
    elif is_object_dtype(dtype):
        result = to_unicode_string(data)
    else:
        result = to_unicode_string(data)

    return result


def infer_dtype(column):
    dtype_order = ['int64', 'float64', 'bool', 'object']
    for dtype in dtype_order:
        result = convert_column_data(column, dtype)

        if result is not None:
            # Successful inference, exit from the loop
            return result

    return None
```

I've omitted some part of the logic, but the code is still pretty self-explanatory.

As it turns out, there is no magic involved in the type inference. At first, pandas trying to convert all values to an integer. If an error occurs, then pandas jumps to the next data type. The last data type is called an `object`, which is simply an array of strings.

See parser.pyx for more details.

Example of corner case

Let's say we have a large CSV file with `low_memory` set to `False`. Where one of the columns has an integer type, but its last value is set to a random string.

Not only it takes more memory while converting the data, but the pandas also converts all the data three times (to an int, float, and string). As a result, you will get a column with an `object` data type.

```
>>> # Clean version
... df = pd.read_csv('col.csv', low_memory=False, verbose=1)
Tokenization took: 1167.96 ms
Type conversion took: 969.31 ms
Parser memory cleanup took: 32.00 ms
>>> df['column'].dtype, df.shape
(dtype('int64'), (10000000, 1))

>>> # Malformed version of the CSV file
... df = pd.read_csv('col_malformed.csv', low_memory=False, verbose=1)
Tokenization took: 1239.30 ms
Type conversion took: 7675.89 ms
Parser memory cleanup took: 38.01 ms
>>> df['column'].dtype, df.shape
(dtype('O'), (10000000, 1))
```

I'm not blaming pandas for this; it's just that the CSV is a bad format for storing data.

Type specification

Pandas allows you to explicitly define types of the columns using `dtype` parameter. However, the converting engine always uses "fat" data types, such as `int64` and `float64.` So even if you specify that your column has an `int8` type, at first, your data will be parsed using an `int64` datatype and then downcasted to an `int8`.

```
if is_integer_dtype(dtype):
    try:
        result, na_count = _try_int64(self.parser, i, start,
                                      end, na_filter, na_hashset)
        if user_dtype and na_count is not None:
            if na_count > 0:
                raise ValueError("Integer column has NA values in "
                                 "column {column}".format(column=i))
    except OverflowError:
        result = _try_uint64(self.parser, i, start, end,
                             na_filter, na_hashset)
        na_count = 0

    if result is not None and dtype != 'int64':
        # numpy's astype cast, creates a copy of an array
        result = result.astype(dtype)

    return result, na_count
```

Popular posts in Python category

python, advanced python, pandas

Share RSS

Comments

Jair 2 years ago #

Very helpful, thanks!

reply

Dmytro 1 year, 10 months ago #

Awesome, thanks for explanation!

reply

NAME

MESSAGE

Plain text or markdown

Send