ConcatVsAppend

November 25, 2023

1 Concat vs Append

It is well known that "concatenating" objects incrementally in languages with automatic memory allocation and garbage collection (like Python, or Java) is really a bad idea.

Consider the simple piece of code

```
total = empty
for part in parts:
  total = total + part
```

where `+` has to be intended as some concatenation operation, and `parts` has N components of size S each.

At every iteration the loop will create an instance of `total` of increasing size: at first S, then $2 \cdot S$, up to the last instance of size $N \cdot S$. Of course the $N - 1$ intermediate instances of `total` will be at some point garbage collected, but it is inevitable to create increasing size concatenation in the process.

So the total space allocated will be $S + 2 \cdot S + \cdots + N \cdot S = S \cdot (1 + 2 + \cdots N) = S \cdot N \cdot (N - 1)/2$, that is a space proportional to N^2. The larger the number of parts, the worst this approach becomes.

It is very easy to see this issue in practice. Let's start for simplicity with list concatenation; populate `parts` with 10k list of 100 integers.

[1]:
```
parts = [list(range(100)) for _ in range(10_000)]
```

If we start from an empty list and perform iterative concatenations, it will take a lot of time

[12]:
```
%%time

slow_all_parts = []
for part in parts:
  slow_all_parts = slow_all_parts + part
```

```
CPU times: user 6.68 s, sys: 8.74 ms, total: 6.69 s
Wall time: 6.73 s
```

By extending the list (an operation that does not create N instances, one per iterate, but that copies `part` at the end of `fast_all_parts`), the time is three orders of magnitude smaller.

[13]:
```
%%time

fast_all_parts = []
for part in parts:
  fast_all_parts.extend(part)
```

```
CPU times: user 2.34 ms, sys: 4.64 ms, total: 6.98 ms
Wall time: 7.07 ms
```

The results, obviously, coincide.

[4]:
```
slow_all_parts == fast_all_parts
```

[4]:
```
True
```

Let turn to Pandas. Let's transform our lists in `DataFrames`

[14]:
```
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)

import pandas as pd

data_frames = [pd.DataFrame(part) for part in parts]
```

Present code in ms2query uses `append` that creates a new intermediate dataframe at every iteration. The execution time is again very high.

[16]:
```
%%time

slow_all_dataframes = pd.DataFrame()
for df in data_frames:
  slow_all_dataframes = slow_all_dataframes.append(df)
```

```
CPU times: user 7.21 s, sys: 475 ms, total: 7.68 s
Wall time: 7.68 s
```

The code emits warning suggesting to use `concat` in place of `append`; this will not change much, since also this method will create intermediate results of growing size.

[17]:
```
%%time

slow_all_dataframes = pd.DataFrame()
for df in data_frames:
  slow_all_dataframes = pd.concat([slow_all_dataframes, df])
```

```
CPU times: user 6.25 s, sys: 371 ms, total: 6.62 s
Wall time: 6.63 s
```

Again, the real advantage comes from using the correct approach, that avoids completely the creation of intermediate concatenations.

[18]:
```
%%time

fast_all_dataframes = pd.concat(data_frames)
```

```
CPU times: user 93.2 ms, sys: 3.47 ms, total: 96.7 ms
Wall time: 95.6 ms
```

The advantage now is of two orders of magnitude, but remains still worth considering! And of course the difference will be much more relevant if the number of objects that we concatenate will grow.

Again, the two results of course coincide.

[23]:
```
slow_all_dataframes.equals(fast_all_dataframes)
```

[23]:
```
True
```