Incremental Computation

Alex Degtiar, Ram Raghunathan, Thomas Marshall

Motivation

- Big data processing takes a lot of time/resources (e.g. PageRank of entire internet)
- Data can change just a little (one new page gets added), but updating result requires recomputation across entire data set.

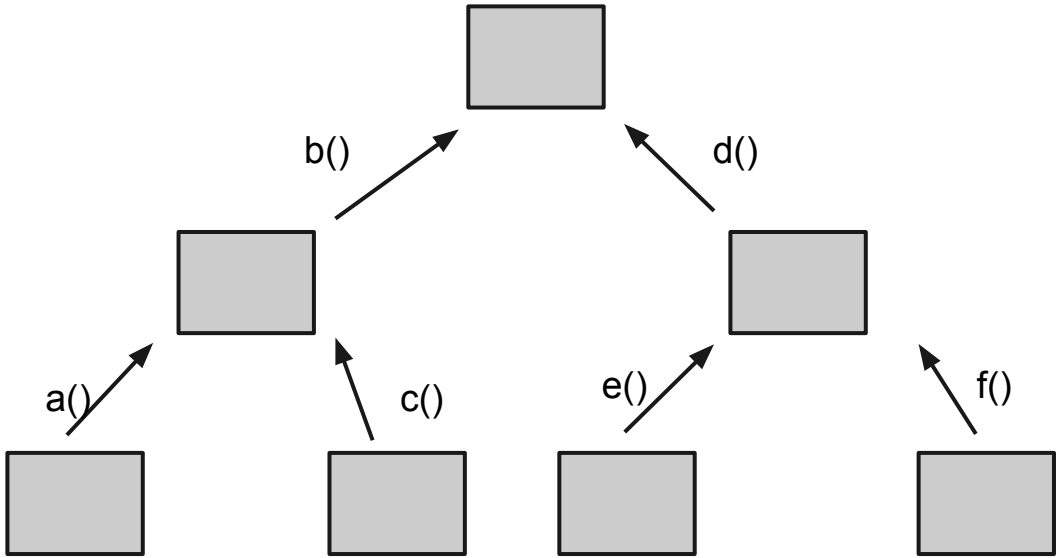
Prior Work

- Specialized systems - distributed data flow, materialized views
- In memory systems - Adaptive Functional Programming (Acar, 2006)

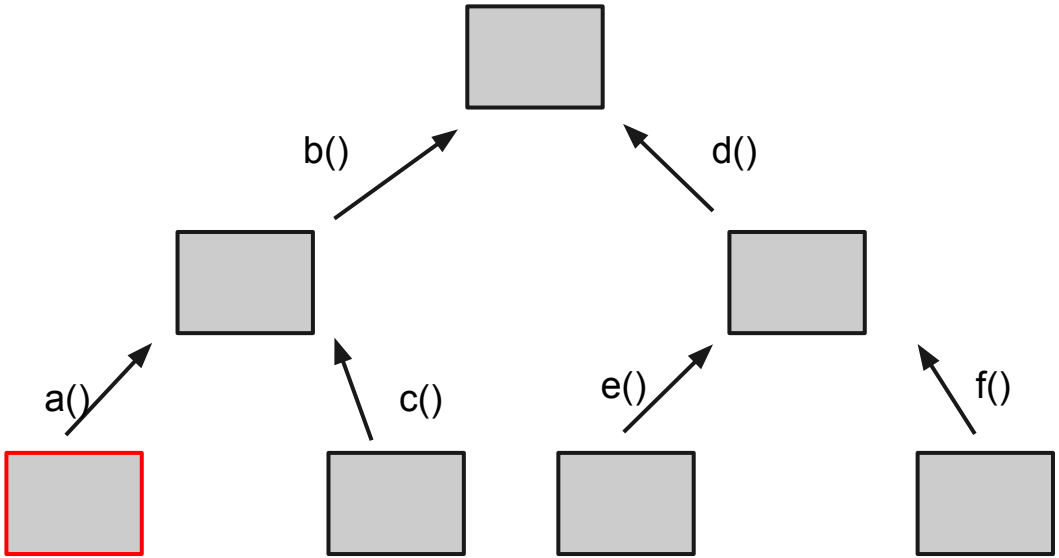
Theory

- Keep track of relationships between computation and data in a Dynamic Dependency Graph (DDG).
- When data changes, it can be propagated through the graph to minimize recomputation needed.

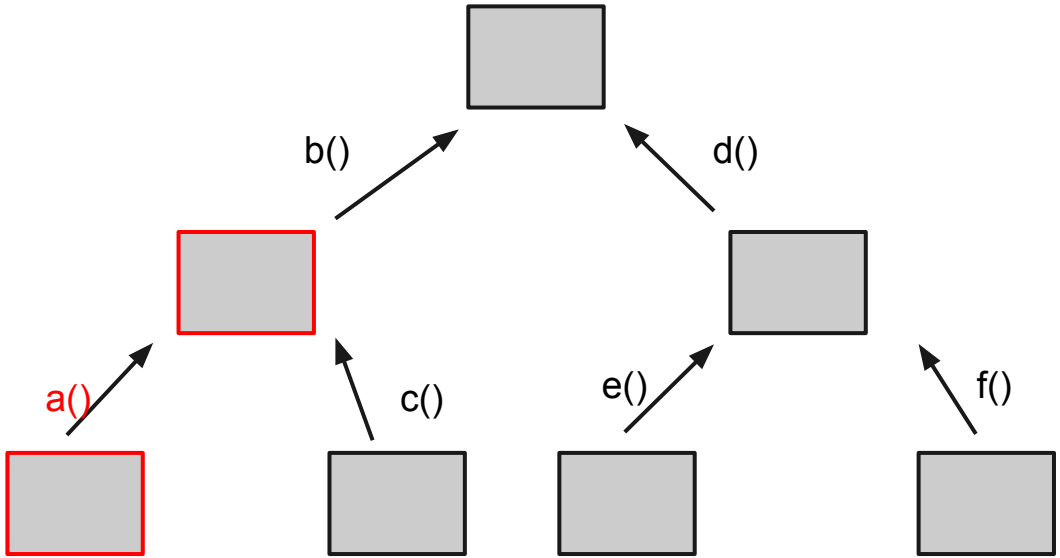
DDG



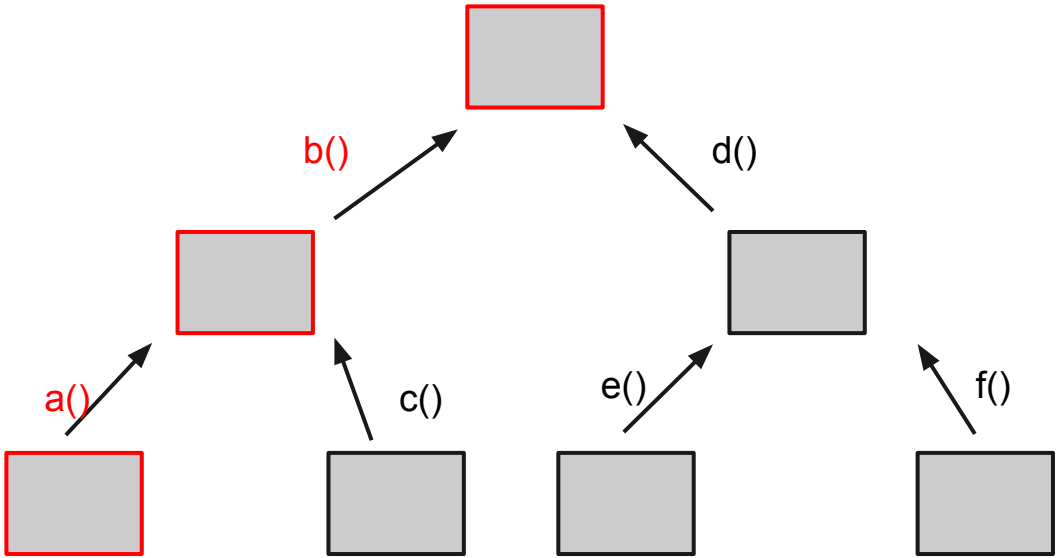
DDG



DDG



DDG



Implementation

- Back-end datastore - Postgres
  - Contains input data along with intermediate results and DDG.
- Python library
  - Contains primitives for interacting with data in a DDG-aware manner.

IncLib

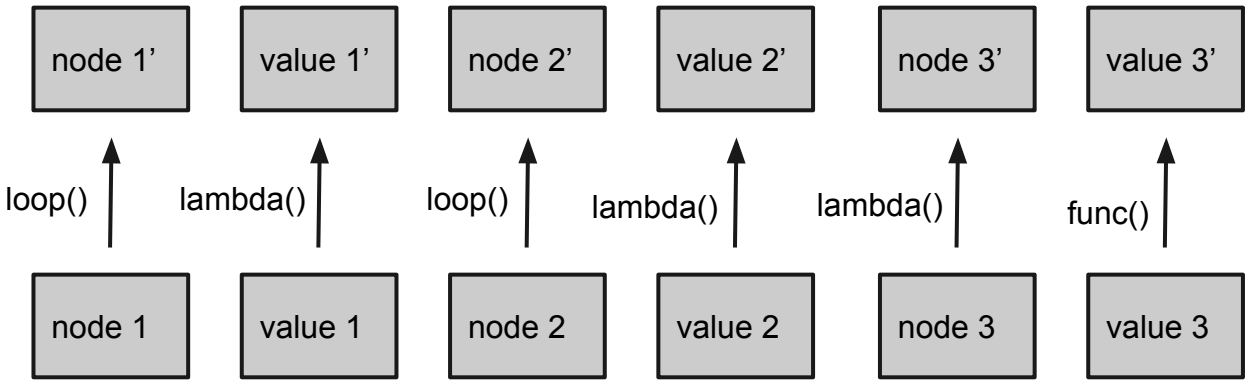
- How to track memory accesses?
  - modifiables
  - continuation passing style

```
def mod(initializer) -> mod

def write(mod, value)

def read(mod, func)

def select(pred)-> mod_list
```

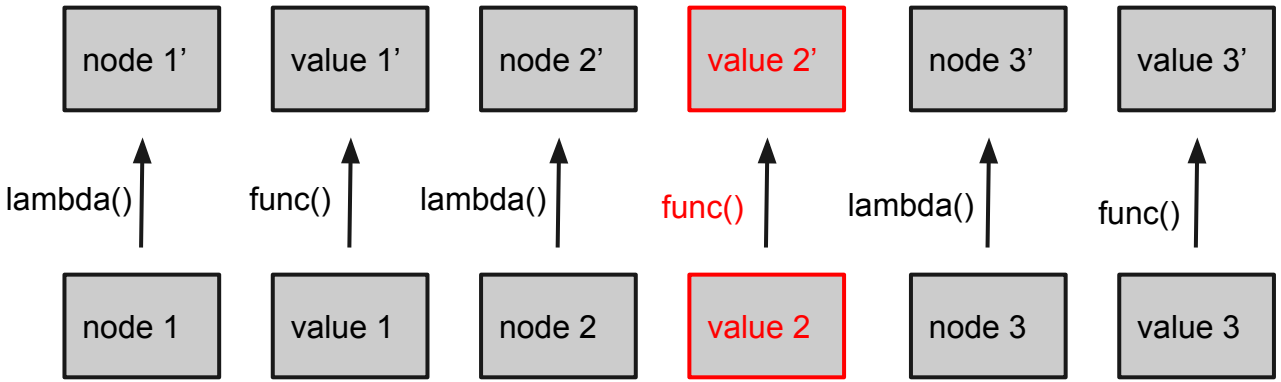
Continuation Passing Style

```
def map(mod_list):
  def m(lst, d):
    read(lst, lambda l:
      (val, nxt) = l
      if nxt is not None:
        write(d, (mod(lambda d: read(val, lambda v: write(d, func(v)))),
                  mod(lambda d: write(d, m(nxt, d))))
      else:
        write(d, (mod(lambda d: read(val, lambda v: write(d, func(v)))),
                  None)))
  return mod(lambda d: m(mod_list, d))
```

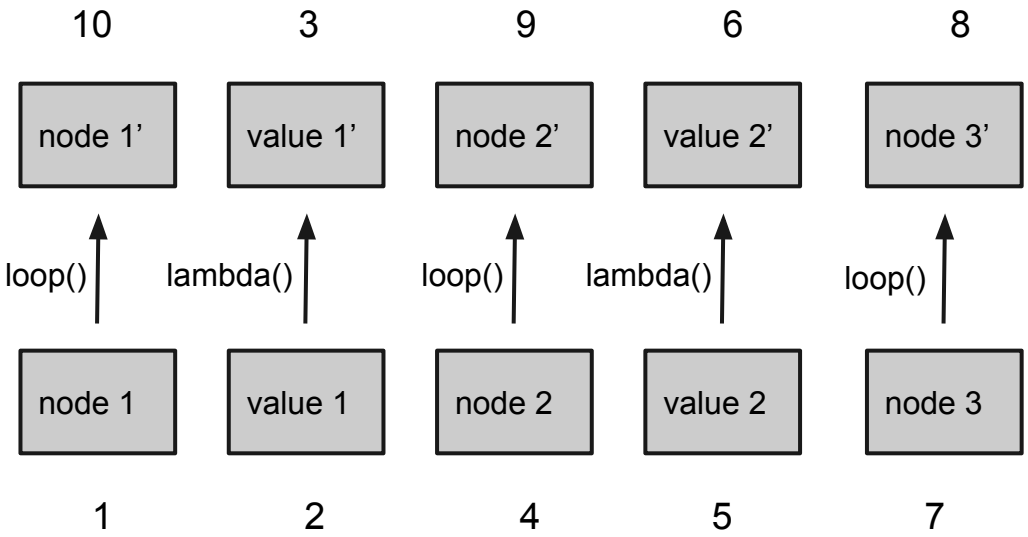
DDG of Map


node 1’
value 1’
node 2’
value 2’
node 3’
value 3’
loop()
lambda()
loop()
lambda()
lambda()
func()
node 1
value 1
node 2
value 2
node 3
value 3

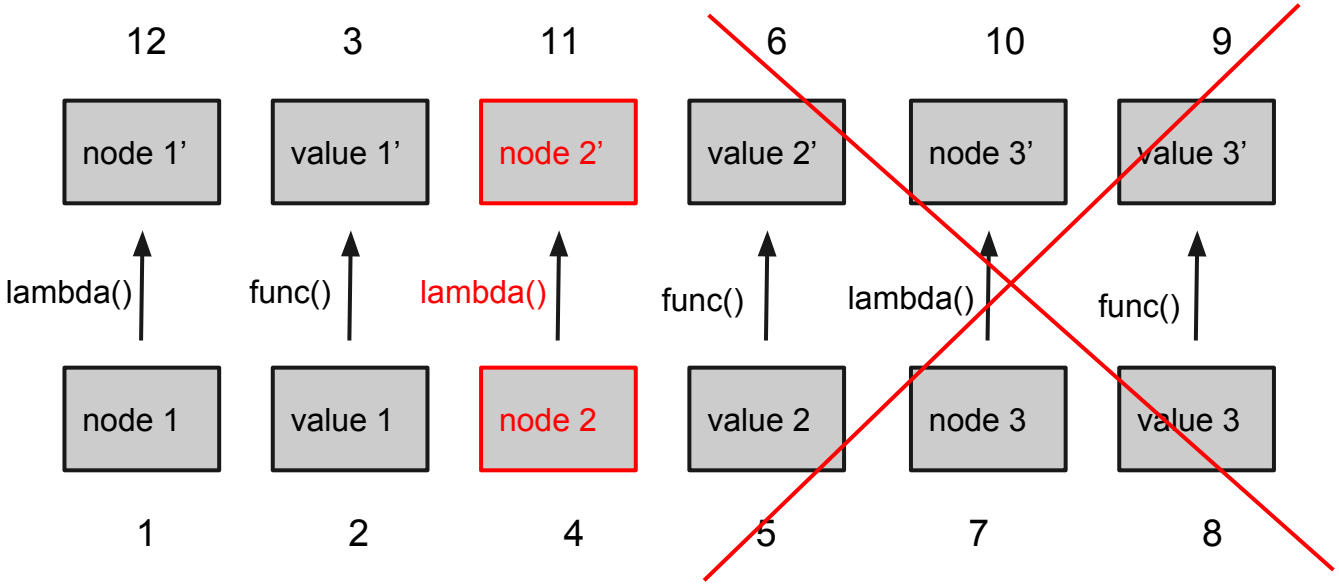
DDG of Map


node 1’
value 1’
node 2’
value 2’
node 3’
value 3’
lambda()
func()
lambda()
func()
lambda()
func()
node 1
value 1
node 2
value 2
node 3
value 3

DDG of Map


10
3
9
6
8
node 1’
value 1’
node 2’
value 2’
node 3’
loop()
lambda()
loop()
lambda()
loop()
node 1
value 1
node 2
value 2
node 3
1
2
4
5
7

DDG of Map

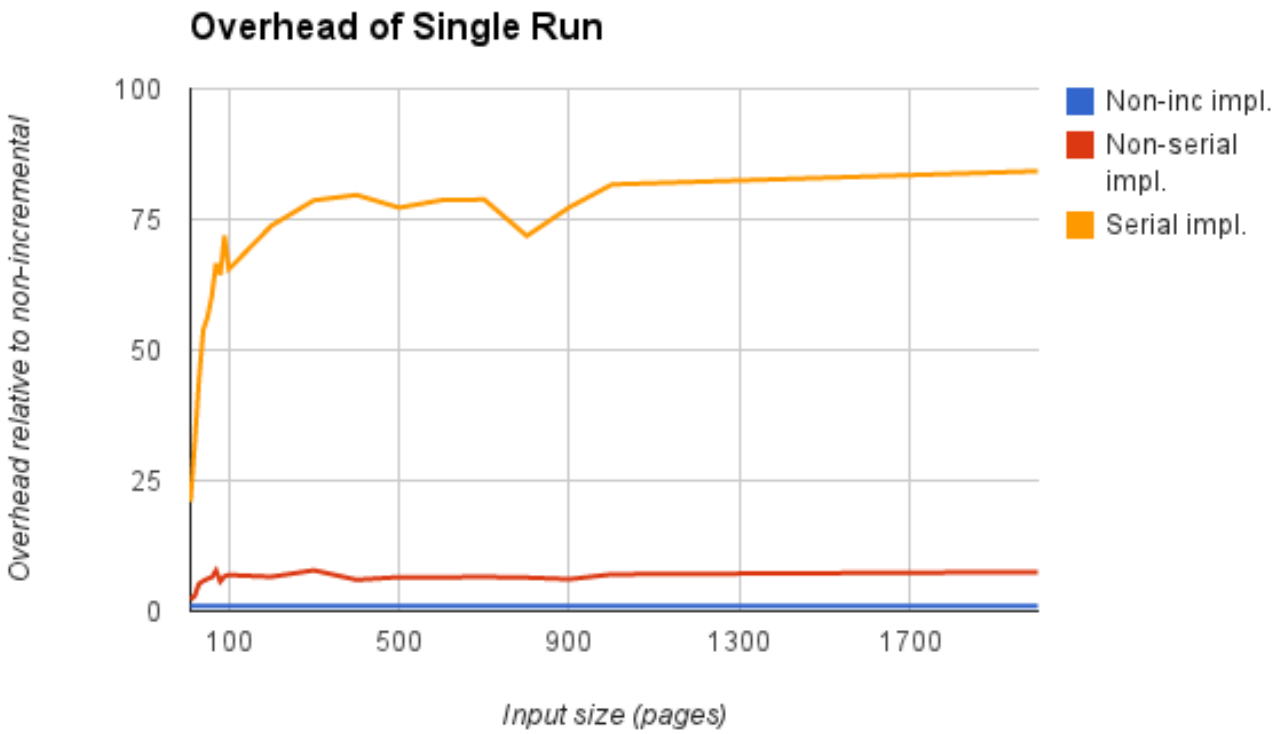

12
3
11
6
10
9
node 1’
value 1’
node 2’
value 2’
node 3’
value 3’
lambda()
func()
lambda()
func()
lambda()
func()
node 1
value 1
node 2
value 2
node 3
value 3
1
2
4
5
7
8

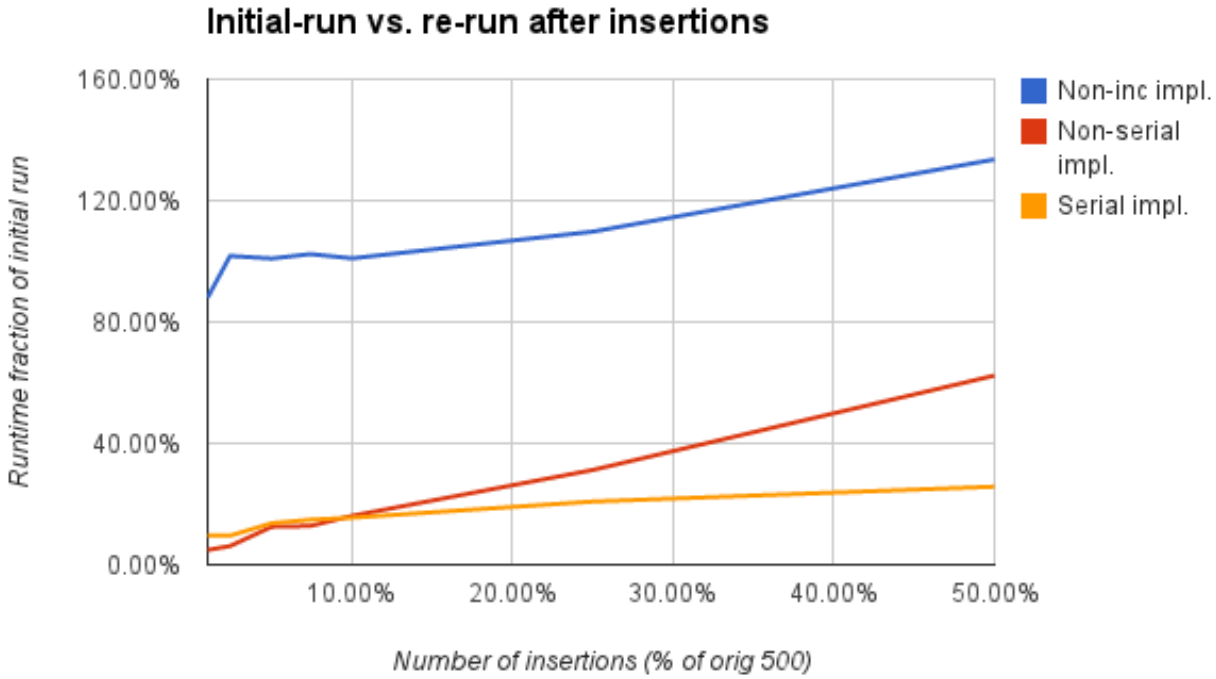
Experiments

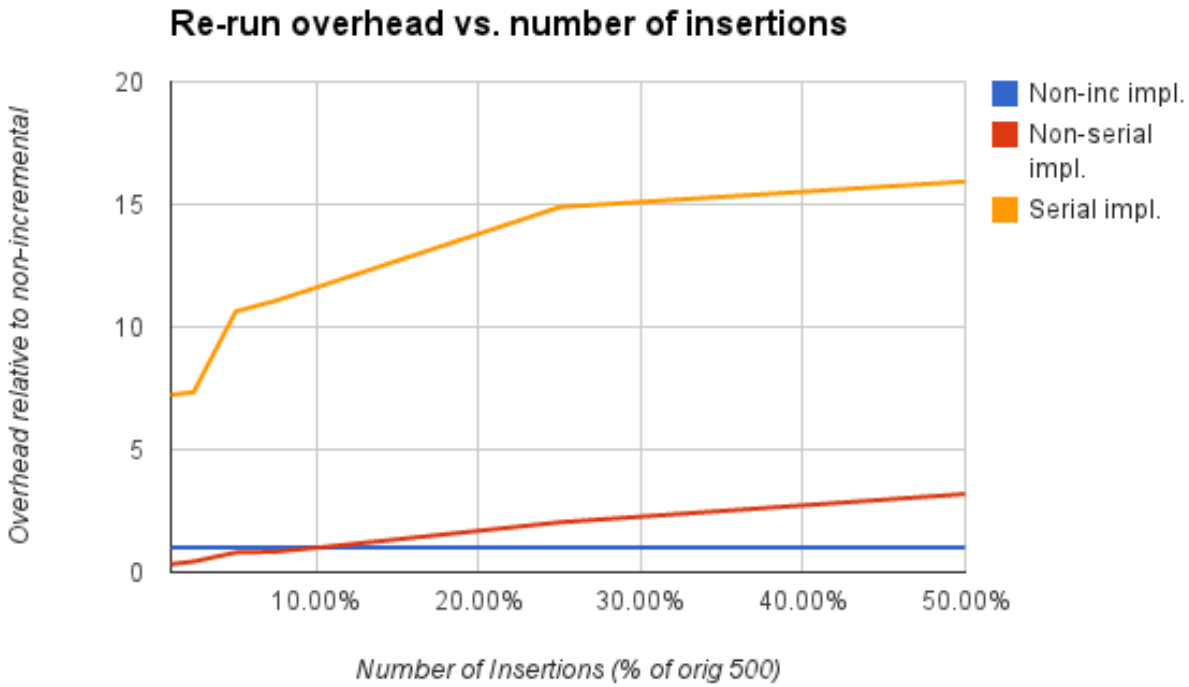
- Word Frequency calculation across “pages”
- Used the Wikipedia corpus as input
- Compared against typical non-incremental implementation
- Varied input size and the number of inserts/updates

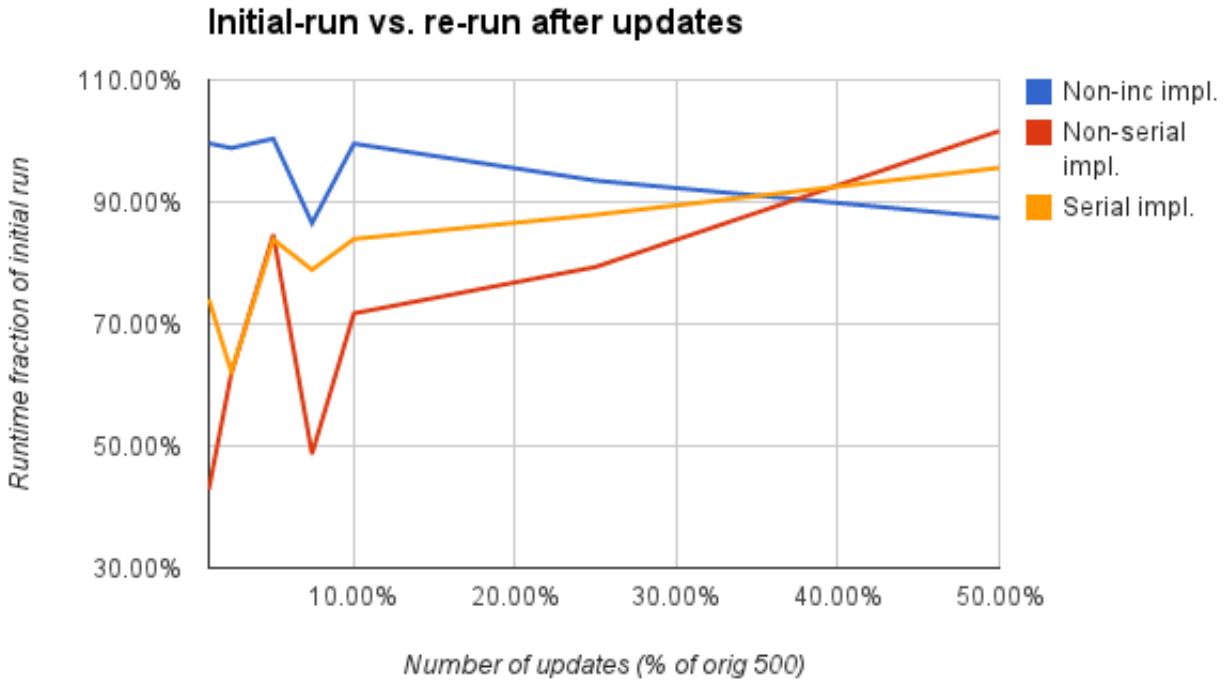
Results

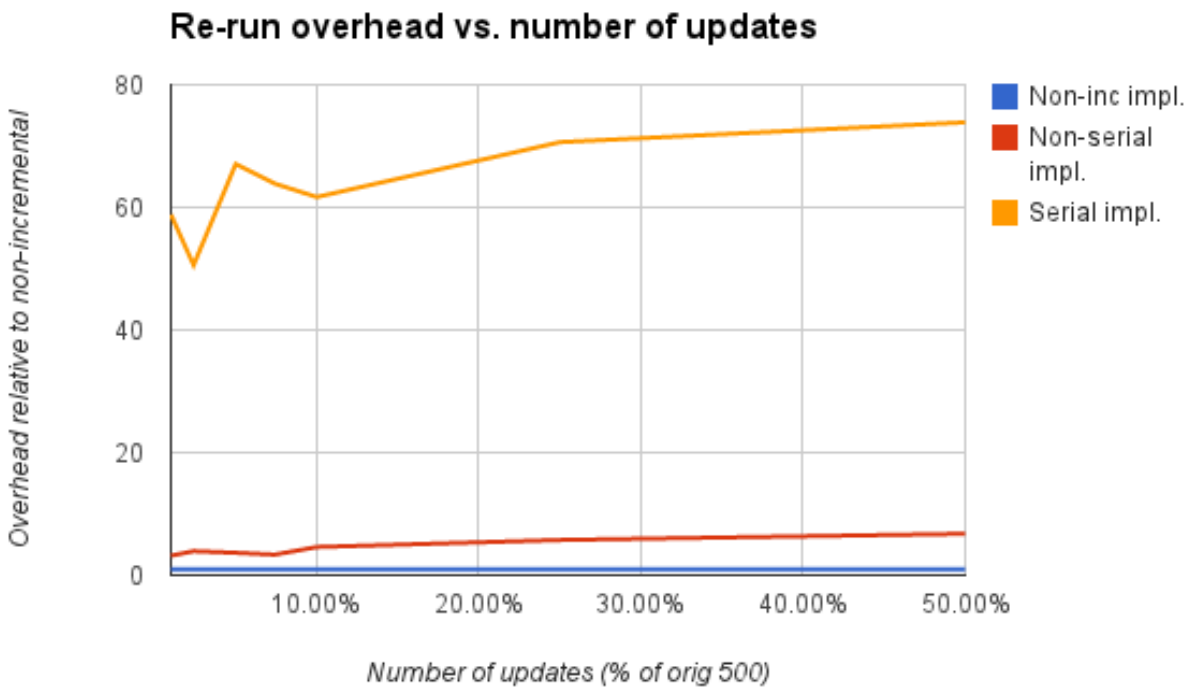
- Overhead of maintaining dependencies
  - Lots of CPU processing (esp. serialization)
  - High Postgres overhead, partially addressed by tuning. More substantially addressed by avoiding serialization.
  - Some space overhead in postgres
- Improvements with re-running after modifying data


Overhead of Single Run
Overhead relative to non-incremental
100
75
50
25
0
100
500
900
1300
1700
Input size (pages)
Non-inc impl.
Non-serial impl.
Serial impl.


Initial-run vs. re-run after insertions
160.00%
120.00%
80.00%
40.00%
0.00%
Runtime fraction of initial run
10.00%
20.00%
30.00%
40.00%
50.00%
Number of insertions (% of orig 500)
Non-inc impl.
Non-serial impl.
Serial impl.


Re-run overhead vs. number of insertions
Overhead relative to non-incremental
20
15
10
5
0
10.00%
20.00%
30.00%
40.00%
50.00%
Number of Insertions (% of orig 500)
Non-inc impl.
Non-serial impl.
Serial impl.


Initial-run vs. re-run after updates
110.00%
90.00%
70.00%
50.00%
30.00%
Runtime fraction of initial run
10.00%
20.00%
30.00%
40.00%
50.00%
Number of updates (% of orig 500)
Non-inc impl.
Non-serial impl.
Serial impl.


Re-run overhead vs. number of updates
Overhead relative to non-incremental
80
60
40
20
0
10.00%
20.00%
30.00%
40.00%
50.00%
Number of updates (% of orig 500)
Non-inc impl.
Non-serial impl.
Serial impl.

Conclusions

- Avoid serialization overhead
- Don’t need many Postgres features
- Utility of incremental computation depends on workload