Towards practical incremental recomputation for scientists

Philip J. Guo and Dawson Engler

Workshop on the Theory and Practice of Provenance

Feb 22, 2010

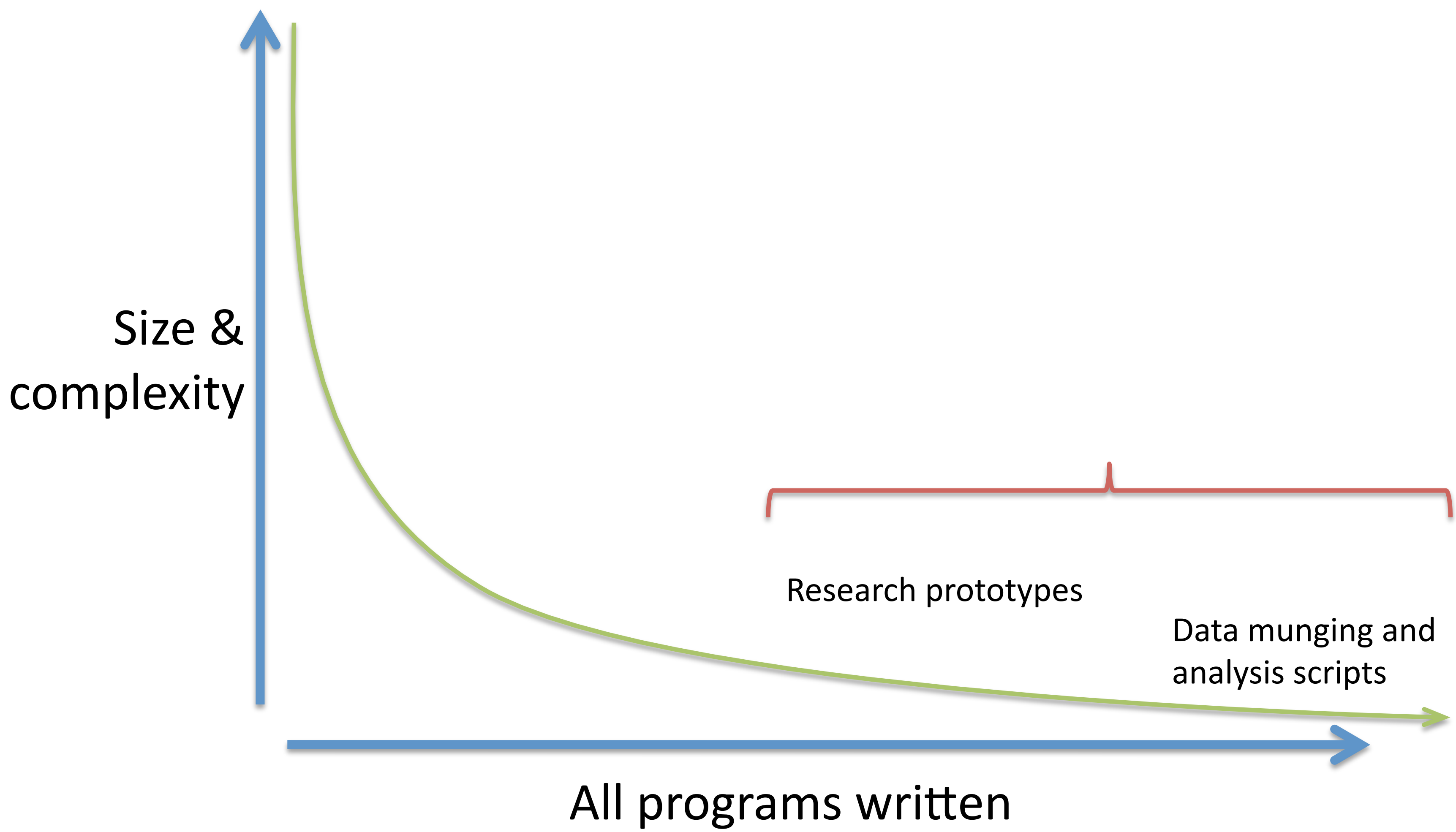
Talk outline

1. **Motivation**: ad-hoc data analysis scripts

2. **Technique**: fully automatic memoization

3. **Benefits**: faster iteration with simple code

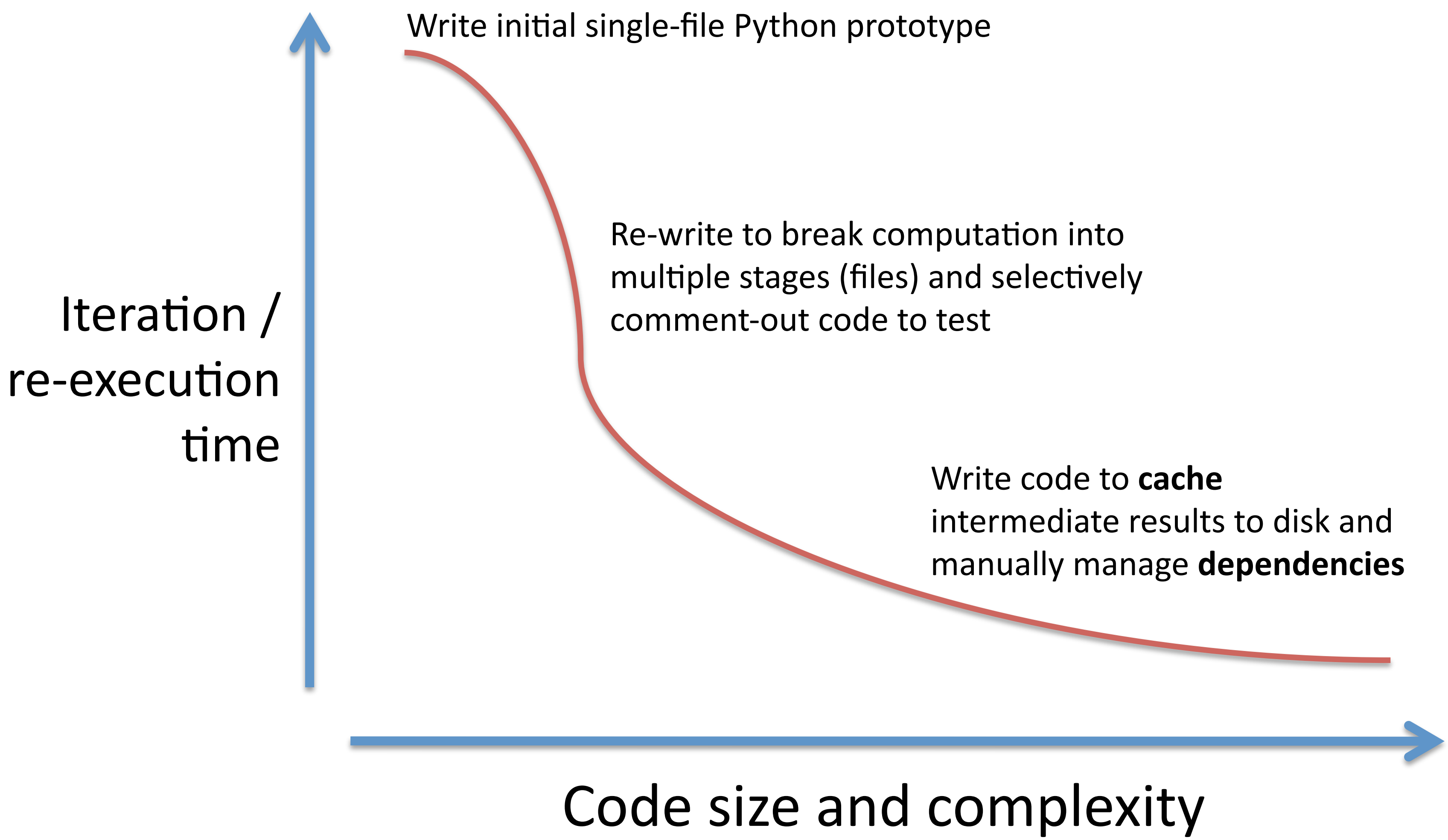
Types of programs



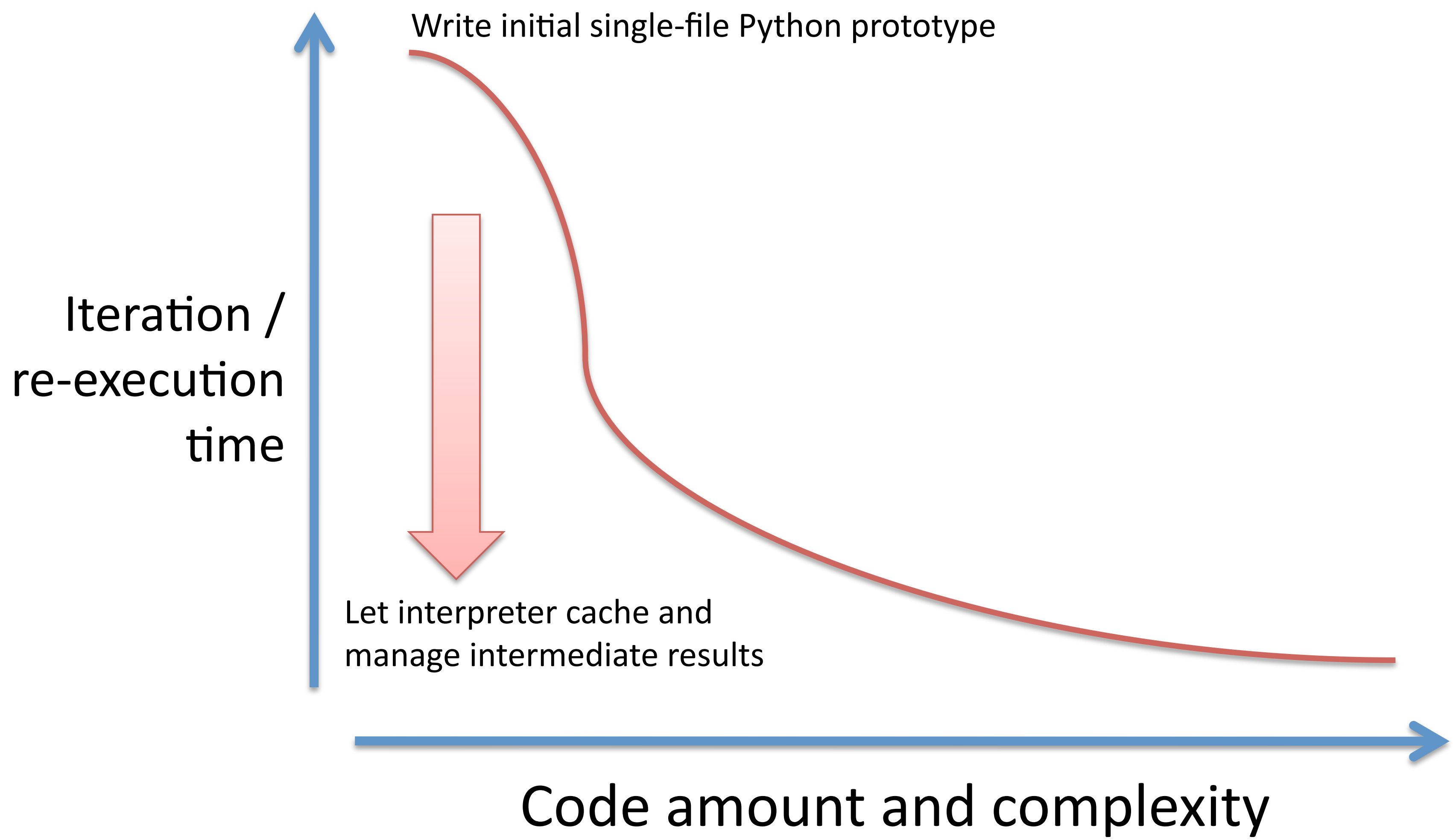
Problem

Scientific data processing and analysis scripts often execute for several minutes to hours, which slows down the scientist's iteration and debugging cycle.

Manually coping



Automated solution



Ideal workflow

1. Write simple first version of script
2. Execute and wait for **1 hour** to get results
3. Interpret results and notice a bug
4. Edit script slightly to fix that bug
5. Re-execute and wait for **a few seconds**
6. Enhance script with new functions
7. Re-execute and wait for a **few minutes**

Technique

Fully automatic and persistent memoization for a general-purpose imperative language

Traditional memoization

```
def Fib(n):
    if n <= 2:
        return 1
    else:
        return Fib(n-1) +  Fib(n-2)
```

Traditional memoization

```
MemoTable = {}

def Fib(n):
    if n <= 2:
        return 1
    else:
        if n in MemoTable:
            return MemoTable[n]
        else:
            MemoTable[n] = Fib(n-1) + Fib(n-2)
            return MemoTable[n]
```

Input (n)	Result
1	1
2	1
3	2
4	3
5	5
6	8
7	13
…	…

Auto-memoizing real programs

1. Code changes
2. External dependencies
3. Side-effects

Auto-memoizing real programs:

Detecting code changes

```
def stageC(datLst):
    res = ...  # run for 10 minutes munging datLst
    return res
```

Input (datLst)	Result
[1,2,3,4]	10
[5,6,7,8]	20
[9,10,11,12]	30

Auto-memoizing real programs:
Detecting code changes

```
def stageC(datLst):
    res = ...  # run for 10 minutes munging datLst
    return res
```

Input (datLst)	Code deps.	Result
[1,2,3,4]	stageC -> C_1	10
[5,6,7,8]	stageC -> C_1	20
[9,10,11,12]	stageC -> C_1	30

Auto-memoizing real programs:
Detecting code changes

```
def stageC(datLst):
    res = ...  # run for 10 minutes munging datLst
    return (res * -1)
```

Input (datLst)	Code deps.	Result
[1,2,3,4]	stageC -> C_1	10
[5,6,7,8]	stageC -> C_1	20
[9,10,11,12]	stageC -> C_1	30

Auto-memoizing real programs:

Detecting code changes

```
def stageC(datLst):
    res = ...  # run for 10 minutes munging datLst
    return (res * -1)
```

Input (datLst)	Code deps.	Result
[1,2,3,4]	stageC -> C_1	10
[5,6,7,8]	stageC -> C_1	20
[9,10,11,12]	stageC -> C_1	30
[1,2,3,4]	stageC -> C_2	-10

Auto-memoizing real programs:

Detecting file reads

```
def stageB(queryStr):
    db = sql_open_db("test.db")
    q = db.query(queryStr)
    res = ... # run for 10 minutes processing q
    return res
```

Input (queryStr)	Code deps.	Result
SELECT * FROM tbl1	stageB -> B_1	1
SELECT * FROM tbl2	stageB -> B_1	2

Auto-memoizing real programs:
Detecting file reads

```
def stageB(queryStr):
    db = sql_open_db("test.db")
    q = db.query(queryStr)
    res = ... # run for 10 minutes processing q
    return res
```

Input (queryStr)	Code deps.	File deps.	Result
SELECT * FROM tbl1	stageB -> B_1	test.db -> DB_1	1
SELECT * FROM tbl2	stageB -> B_1	test.db -> DB_1	2

Auto-memoizing real programs:
Detecting global variable reads

```
MULTIPLIER = 5
def stageB(queryStr):
    db = sql_open_db("test.db")
    q = db.query(queryStr)
    res = ... # run for 10 minutes processing q
    return (res * MULTIPLIER)
```

Input (queryStr)	Code deps.	File deps.	Result
SELECT * FROM tbl1	stageB -> B_2	test.db -> DB_1	5

Auto-memoizing real programs:

Detecting global variable reads

```
MULTIPLIER = 5
def stageB(queryStr):
    db = sql_open_db("test.db")
    q = db.query(queryStr)
    res = ... # run for 10 minutes processing q
    return (res * MULTIPLIER)
```

Input (queryStr)	Code deps.	File deps.	Global deps.	Result
SELECT * FROM tbl1	stageB -> B_2	test.db -> DB_1	MULTIPLIER -> 5	5

Auto-memoizing real programs:
Detecting global variable reads

```
MULTIPLIER = 10
def stageB(queryStr):
    db = sql_open_db("test.db")
    q = db.query(queryStr)
    res = ... # run for 10 minutes processing q
    return (res * MULTIPLIER)
```

Input (queryStr)	Code deps.	File deps.	Global deps.	Result
SELECT * FROM tbl1	stageB -> B_2	test.db -> DB_1	MULTIPLIER -> 5	5
SELECT * FROM tbl1	stageB -> B_2	test.db -> DB_1	MULTIPLIER -> 10	10

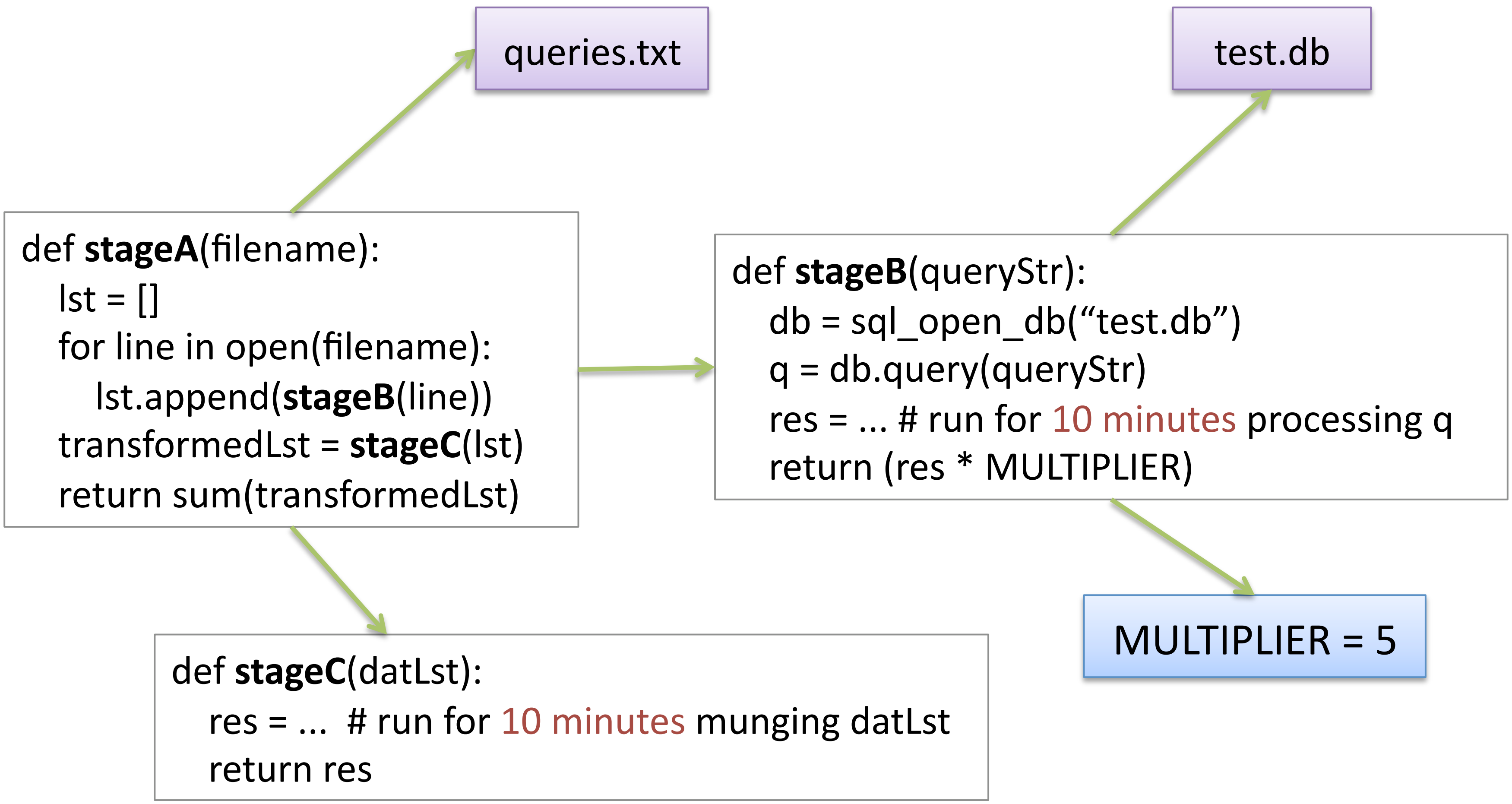
Auto-memoizing real programs:
Detecting transitive dependencies

```
def stageA(filename):
    lst = []
    for line in open(filename):
        lst.append(stageB(line))
    transformedLst = stageC(lst)
    return sum(transformedLst)
```

Input (filename)	Code deps.	File deps.	Global deps.	Result
queries.txt	stageA -> A_1	queries.txt -> Q_1		50

Auto-memoizing real programs:

Detecting transitive dependencies



Auto-memoizing real programs:

Detecting transitive dependencies

```
def stageA(filename):
    lst = []
    for line in open(filename):
        lst.append(stageB(line))
    transformedLst = stageC(lst)
    return sum(transformedLst)
```

Input (filename)	Code deps.	File deps.	Global deps.	Result
queries.txt	stageA -> A_1 stageB -> B_1 stageC -> C_1	queries.txt -> Q_1 test.db -> DB_1	MULTIPLIER -> 5	50

Auto-memoizing real programs:
Detecting impurity

"Before memoizing a given routine, the programmer needs to verify that there is no internal dependency on side effects. This is not always simple; despite attempts to encourage a functional programming style, programmers will occasionally discover that some routine their function depended upon had some deeply buried dependence on a global variable or the slot value of a CLOS [Common Lisp Object System] Instance." [Hall and Mayfield, 1993]

Auto-memoizing real programs:

Detecting impurity

- All functions start out pure
- Mark all functions on stack as impure when:
  - Mutating a non-local value
  - Writing to a file
  - Calling a non-deterministic function
- Data analysis functions mostly pure

Incremental recomputation

queries.txt

```
SELECT ...
SELECT ...
SELECT ...
SELECT ...
SELECT ...
```

```
def stageA(filename):
   lst = []
   for line in open(filename):
      lst.append(stageB(line))
   transformedLst = stageC(lst)
   return sum(transformedLst)
```

```
def stageB(queryStr):
   db = sql_open_db(“test.db”)
   q = db.query(queryStr)
   res = ... # run for 10 minutes processing q
   return (res * MULTIPLIER)
```

test.db

MULTIPLIER = 5

```
def stageC(datLst):
   res = ...  # run for 10 minutes munging datLst
   return res
```

Benefits

1. Less code and bugs
2. Faster iteration cycle
3. Real-time collaboration

Talk review

1. **Motivation**: ad-hoc data analysis scripts

2. **Technique**: fully automatic memoization

3. **Benefits**: faster iteration with simple code

Ongoing and future work

- Provenance browsing
- Database-aware caching
- Network-aware caching
- Lightweight programmer annotations
- Finer-grained tracking within functions

Implementation

- **Python** as target language
- **Plug-and-play** with no code changes
- **Low** run-time overhead
- Compatible with 3rd-party **libraries**