A High-Performance Computing Forecast: Partly Cloudy

Ke Zhai
University of Maryland

Cloud Computing

- Vast amounts of data by Google, Amazon, etc.
- Utility Computing: Pay-per-Usage
- Advantage?
  - Reduce overall cost
  - Increase service availability & Elasticity
  - Virtualization & abstraction
- Concerns?
  - Performance vs. Service quality guarantee
  - Resource sharing vs. Security & privacy
  - Freedom & unique vs. Dedicated & universal

Shall “we” let the cloud fly?

HPC requirements (from most general to most demanding)

- Network Communications
  - Low bandwidth = maximize data locality
  - Commercially available network infrastructures were “one and two orders of magnitude inferior to big computer center facilities”
- HPC Administrative Workload Support
  - Does cloud computing yields a cost reduction in administration, i.e., software maintenance / software licenses?
- Data Analysis and Visualization
  - Expertise in installing, tuning and maintaining vs. expertise in using the software/system.
- Dataset Management
  - Cloud provides better confidence in data integrity due to its distributed nature.
  - However, what about security and privacy? Quite often, they are deal breaker!

Shall “we” let the cloud fly? (cont.)

HPC requirements (from most general to most demanding)

- Capacity Computing for Job-Stream Throughput Workflows
  - Are jobs easily parallizable, or strictly sequentialized?
- Co-orporative Computing with Weak Scaling
  - Increase a single application’s performance with increased system scale, but permits the problem size to grow proportionally with system scale.
- Capacity Computing with Strong Scaling
  - Reduce a fixed-size problem’s execution time proportional to the increase in system scale.

Hence, clouds provide a substantial flexibility for HPC institutions, as it could support different requirements to different extension.

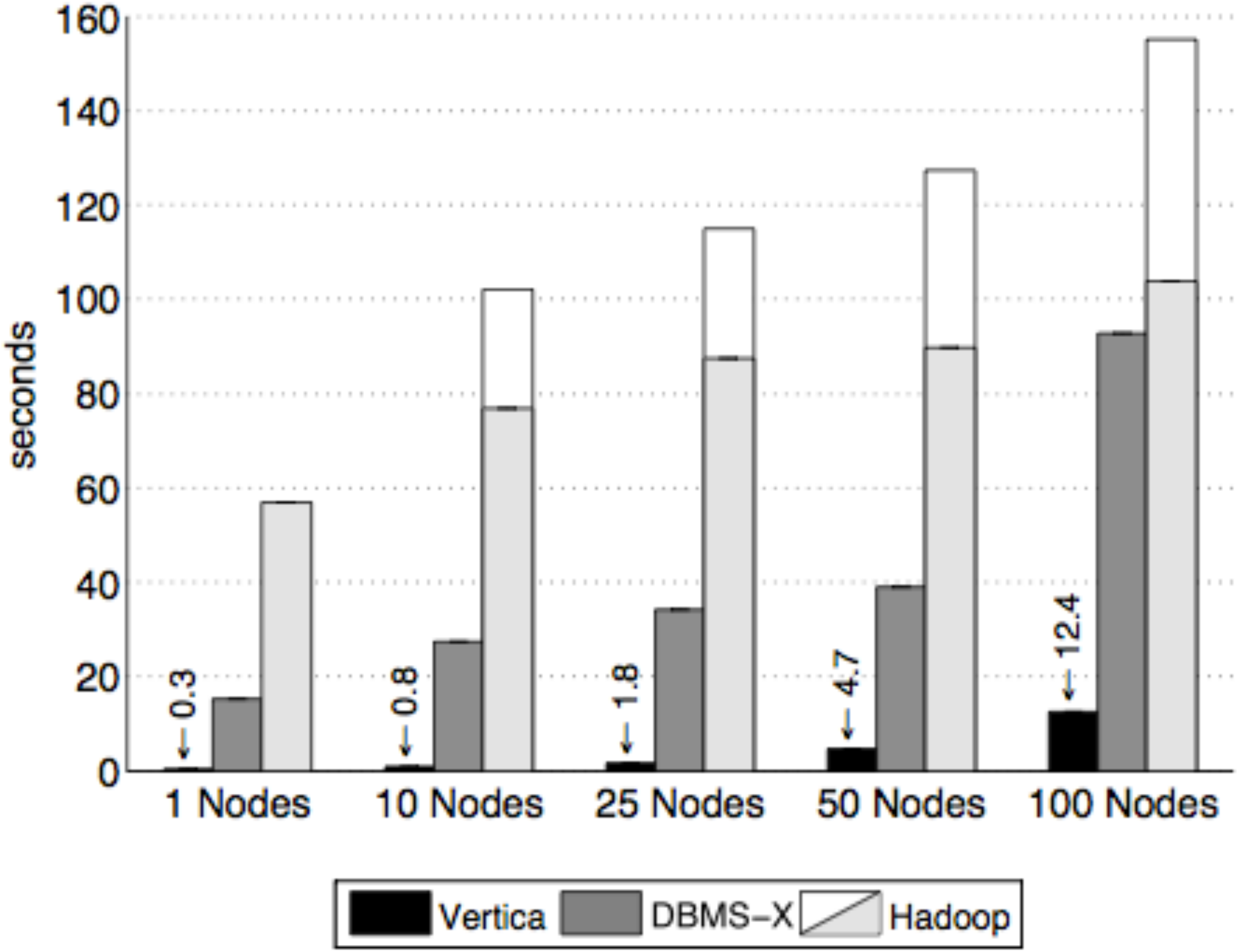
Manimal - Automatic Optimization for MapReduce Programs

Ke Zhai
University of Maryland

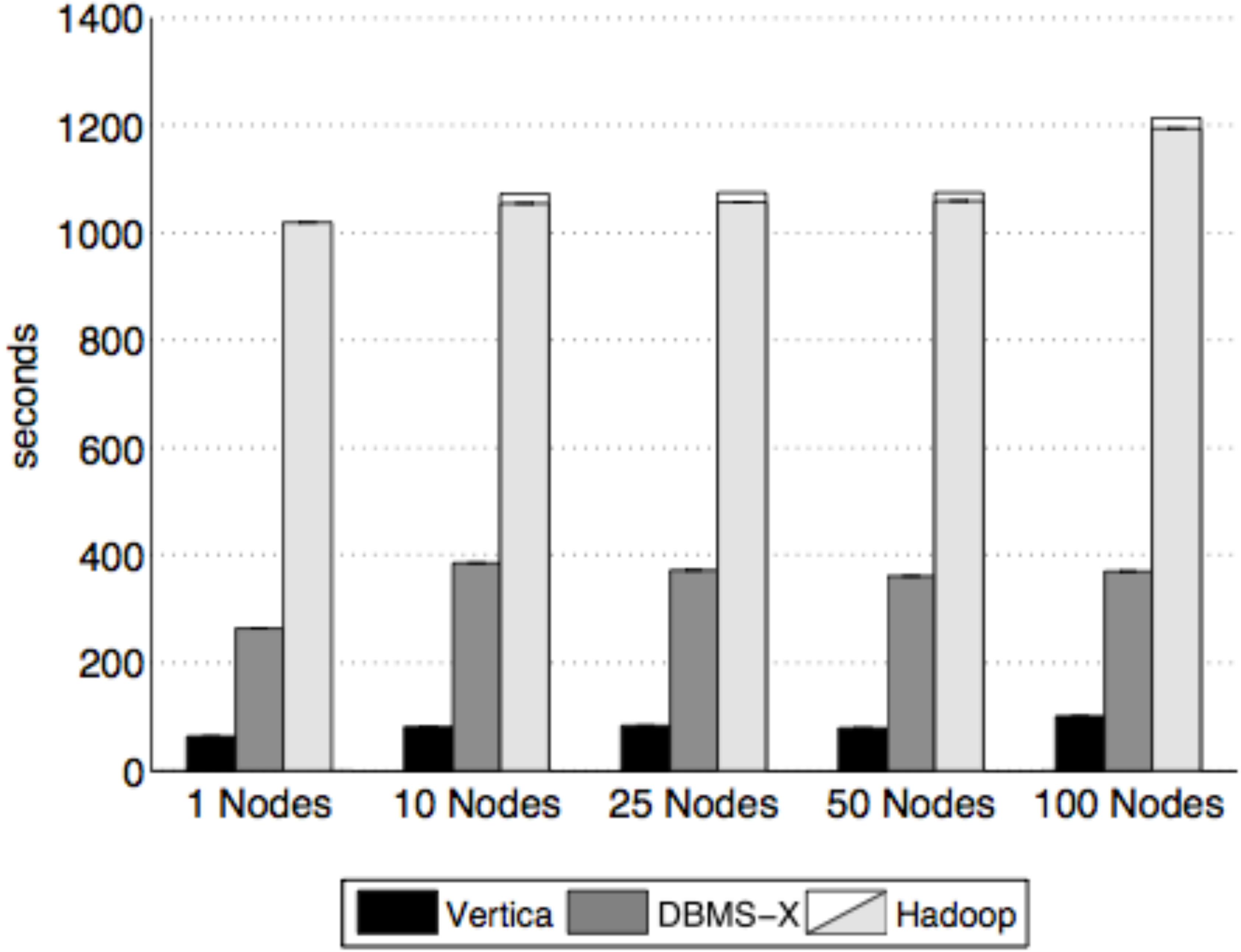
Original slides from
Michael J. Cafarella & Christopher Ré

Data-Centric Programming

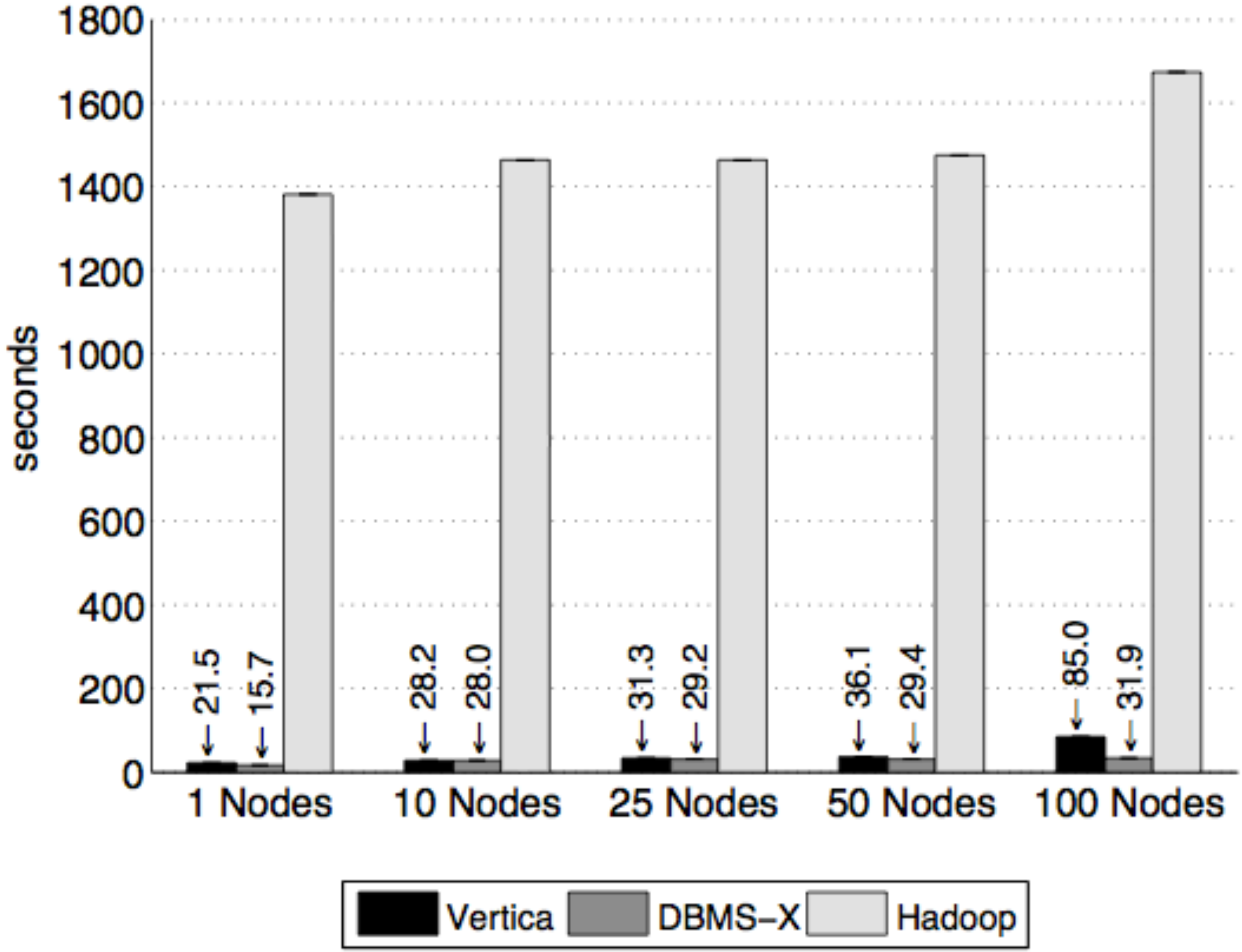
- MapReduce has become very popular, for lots of good reasons
  - Easy to write distributed programs
  - Built-in reliability on large clusters
  - Bytestreams, not relations
  - “Schema-later”, or “schema-never”
  - Your choice of programming languages
  - Hadoop relatively easy to administer
- But currently, MapReduce programming comes at a cost



160
140
120
100
80
60
40
20
0
seconds
0.3
0.8
1.8
4.7
12.4
1 Nodes
10 Nodes
25 Nodes
50 Nodes
100 Nodes
Vertica
DBMS−X
Hadoop



seconds
1400
1200
1000
800
600
400
200
0
1 Nodes
10 Nodes
25 Nodes
50 Nodes
100 Nodes
Vertica
DBMS−X
Hadoop



1800
1600
1400
1200
1000
800
600
400
200
0
seconds
21.5
15.7
28.2
28.0
31.3
29.2
36.1
29.4
85.0
31.9
1 Nodes
10 Nodes
25 Nodes
50 Nodes
100 Nodes
Vertica
DBMS−X
Hadoop

MapReduce Performance

- (Pavlo, et al, SIGMOD '09) show that MapReduce runs 2-50x slower than RDBMS on same hardware!
- Developers currently choose:
  - Efficiency of RDBMS? or
  - Dev & parallel advantages of MapReduce?
- **Manimal** is a *hybrid system,* combining MapReduce programming model and well-known execution techniques
- Techniques today only found in RDBMS, but should be in MapReduce, too

Related Work

- Lots of recent MapReduce activity
  - Task scheduling *(Isard et al, SOSP, 2009; Zaharia et al, OSDI, 2008)*
  - Managing different cluster systems *(Hindman et al, HotCloud 2009)*
  - Joins *(Afrati & Ullman, EDBT 2010; Yang et al, SIGMOD 2007)*
  - Largely novel, MapReduce-specific optimization techniques
- Manimal detects & applies existing optimizations, does not offer new ones

Manimal

- MapReduce programmers *can* express anything, but in practice express RDBMS-style ops
  - Detect optimization opportunities in code
  - Determine when indexes support optimizations
  - Exploit opportunities with well-known techniques from decades of research
- Just like a marriage between MapReduce and RDBMS
  - execution/programming framework (physical body) from MapReduce, and the optimization (intelligent system) from RDBMS

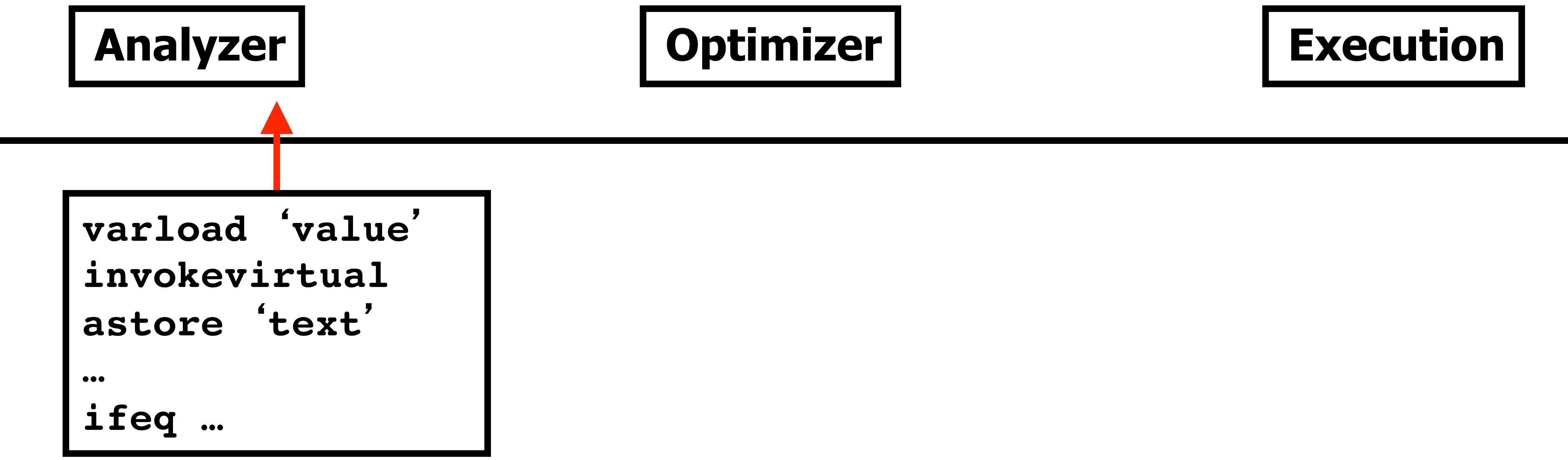
Execution Framework

```
public void map(Text k, Text v,
               OutputCollector<Text, LongWritable> out){
 Matcher m = pattern.matcher(v.toString());
 while (m.find()) {
   out.collect(new Text(m.group(1)),
               new LongWritable(1));
 }
}
```

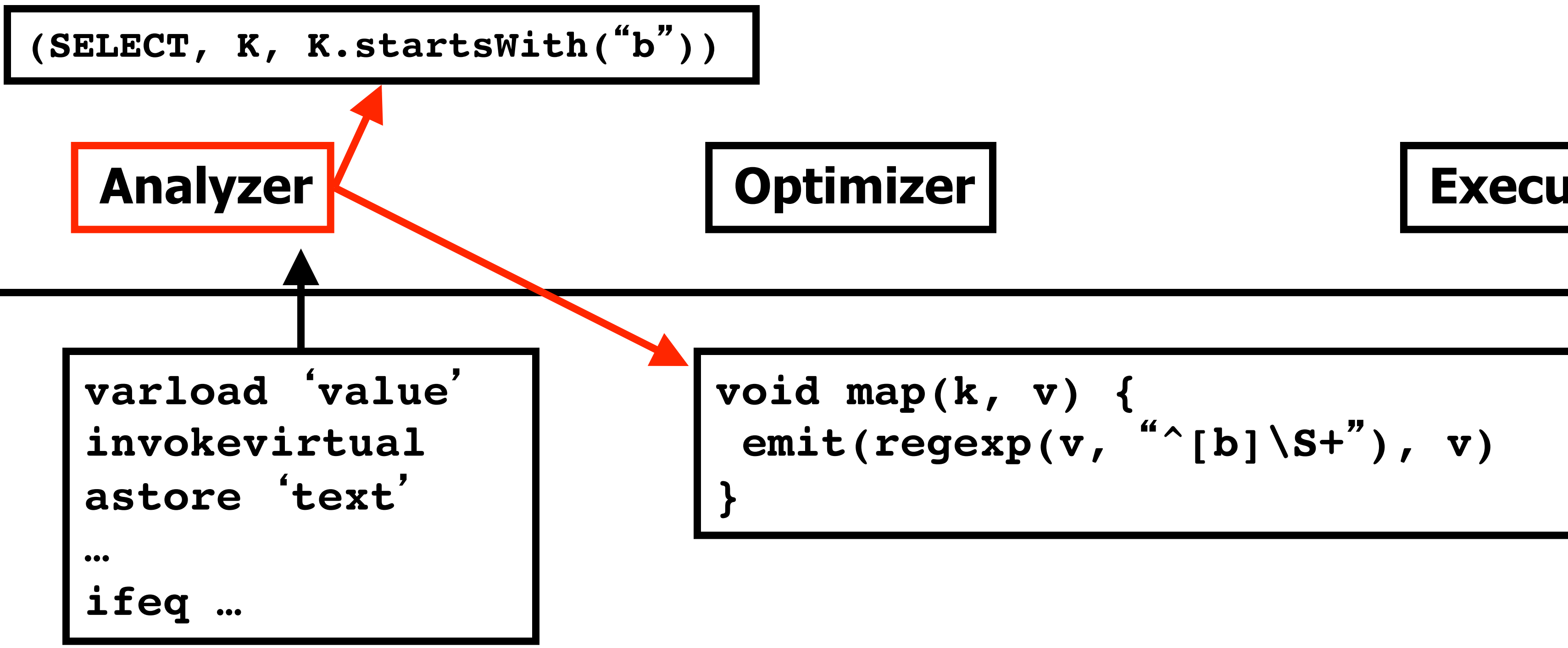
Compiler

```
varload 'value'
invokevirtual
astore 'text'
…
ifeq …
```

Execution Framework

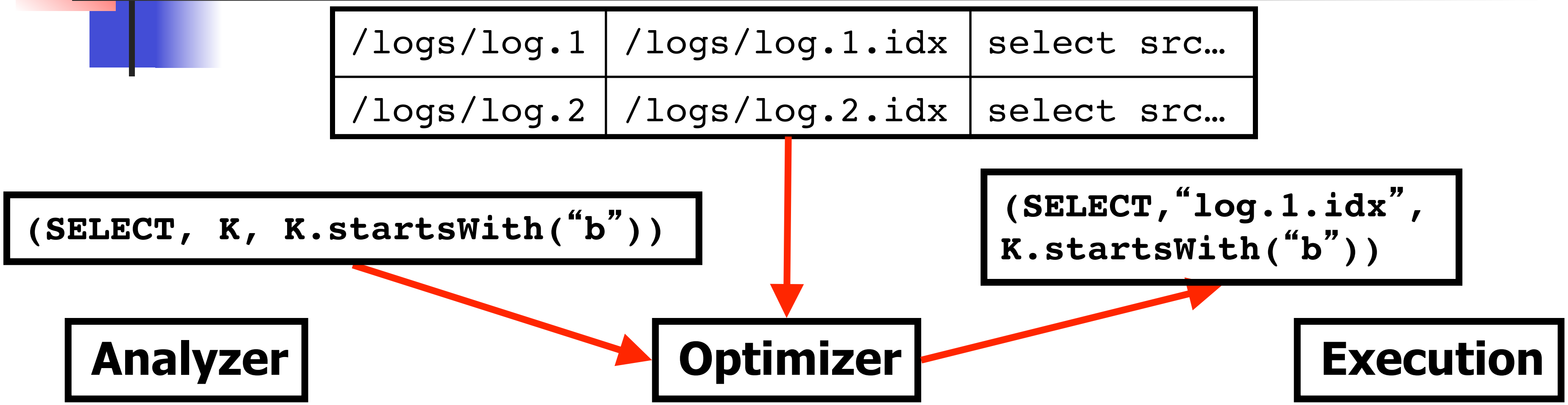


Execution Framework



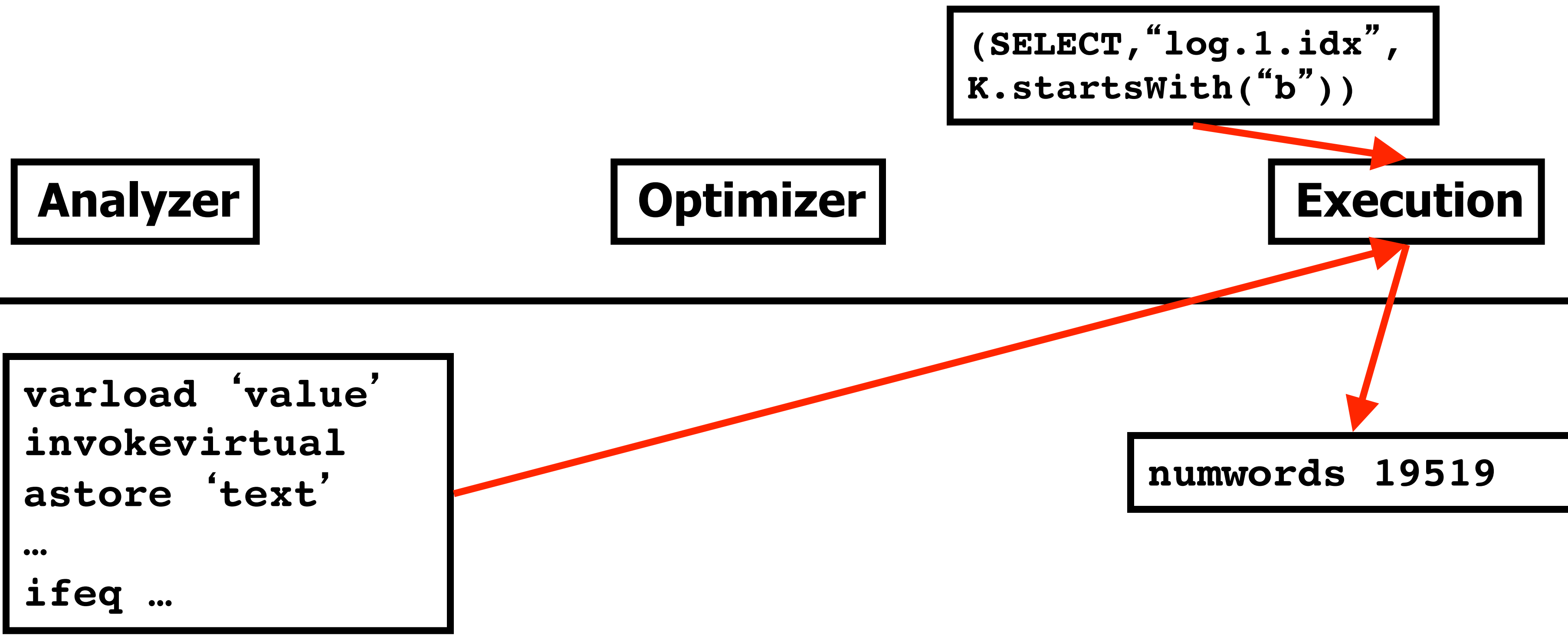
Analyzer in: user program
Analyzer out: optimization descriptor
index-generation program

Execution Framework



Optimizer in: optimization descriptor
catalog
Optimizer out: execution descriptor

Execution Framework



Execution in: execution descriptor
user program
Execution out: program output

Optimizations

- Selection
  - Use index on the keys, generated by analyzer
- Projection
  - As early as possible, even before the map() function.
- Data Compression
  - Delta-compression ~ store the differences
  - Direct-operation ~ skip decompress operation

Review: Analyzer

- Detecting optimizations is "best-effort"
  - False negatives not great, but OK
  - False positives catastrophic

```
(SELECT, K, K.startsWith("b"))
```

Analyzer

Optimizer

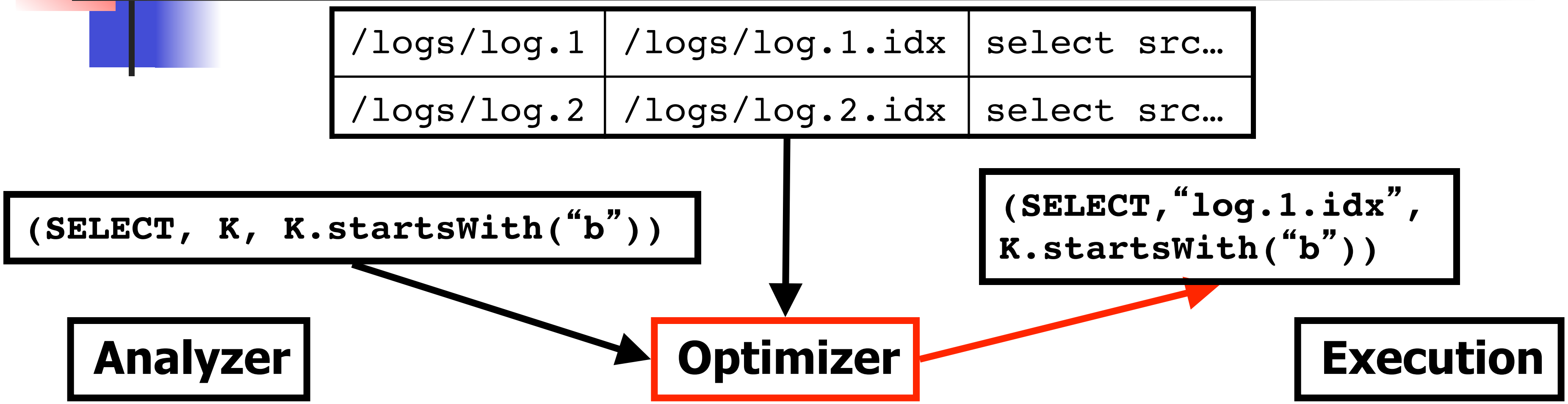
Execution

```
varload 'value'
invokevirtual
astore 'text'
…
ifeq …
```

```
void map(k, v) {
 emit(regexp(v, "^[b]\S+"), v)
}
```

Analyzer reads user program.
Best-effort system to discover "valid optimizations"
Emits **optimization descriptor & index-gen program**

Review: Optimizer

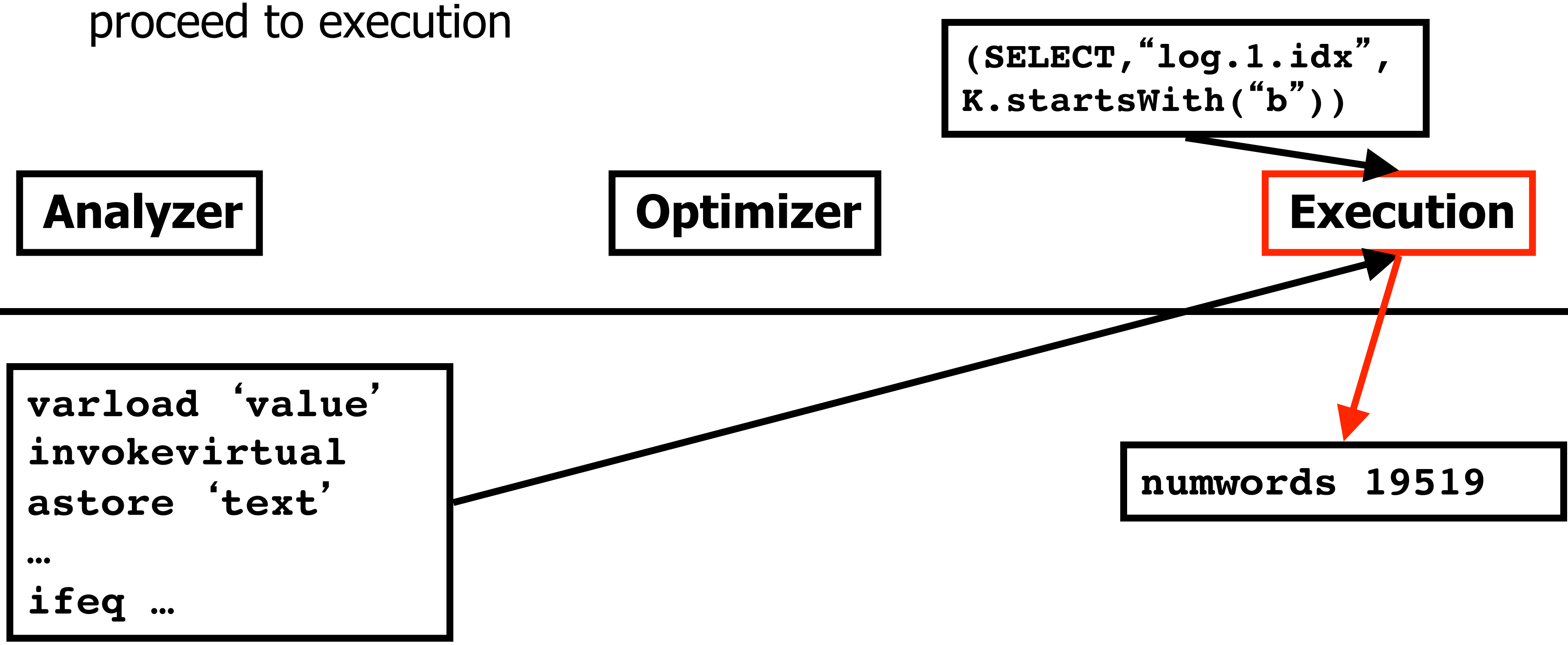


- How to rank different optimizations?
  - Long run: cost based approach?
  - **Short run: rule based approach?**

Optimizer accepts optimizer descriptor, catalog of indexes
Decides which available optimization technique to apply
Emits **execution descriptor**

Review: Execution

- Gathers all the optimizations and proceed to execution



Execution system takes user program, exec-desc, indexes
Employs appropriate optimization technique to exec code
Emits **final MapReduce output**

Control Flow Graph (CFG)

```
void map(String k, WebPage v) {
  if (v.rank > 1)
    emit(k, 1);
}
```

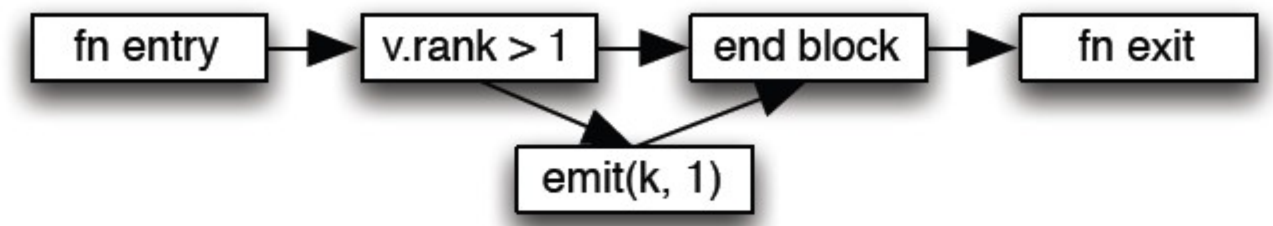


Figure 4: The control flow graph for the function in Section 2. Nodes represent basic blocks. Edges represent possible control flow paths.

- No control flow → one entrance and one exit point → merge-able into a single basic block
- Encapsulate all possible paths the program might take during execution

Data Flow Analysis

```
void map(String k, WebPage v) {
  if (v.rank > 1)
    emit(k, 1);
}
```

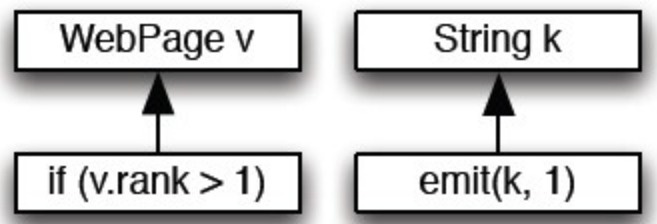


Figure 5: Some use-def chains for the map() function in Section 2. Nodes represent instructions or variable definitions; each edge's source requires data from its target. We show Java statements for clarity, but the actual graph is computed on bytecodes.

- Statement d is reach a use of that variable's definition (assignment) at statement u, if u is reachable from d in CFG & no intervening.
- Encapsulate all possible paths the program might take during execution

Analyzer: Selection

Optimization plan is safe ➔ produce the same result

- Statement s
- Paths(s): CFG paths that reach s
- Cond(s): selection condition
- getUseDef(s): get all dependencies
- isFunc(useDefChain):
  - Use depends only on map() parameters or constants
  - All the function called must not be dnf (did no find???)

```
function findSelect(mapperStmts):
allFunc ← true, condPaths ← {}, dnf ← false
for s ∈ mapperStmts do
  if isEmit(s) then
    for all path ∈ paths(s) do
      condPaths ← condPaths ∪ conds(path)
      dnf ← dnf OR conj(conds(path))
for all condPath ∈ condPaths do
  for all cond ∈ condPath do
    if not isFunc(getUseDef(cond)) then
      allFunc ← false
if allFunc return dnf else return {}
```

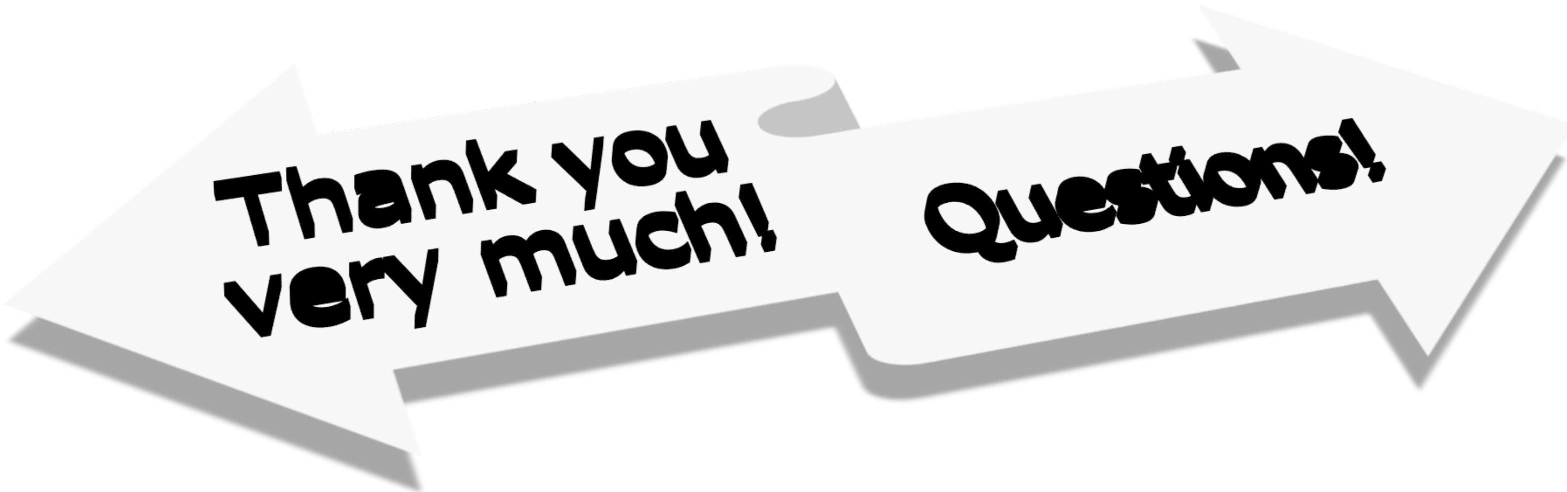
Figure 3: The analyzer detection algorithm for selection. $conj()$ returns a conjunction of the logical conditional expressions in its input.

Analyzer: Projection & Compression

```
function findProject(mapperStmts, paramFields):
allPaths ← {}, usedFields ← {}
for s ∈ mapperStmts do
    if isEmit(s) then
        for all path ∈ paths(s) do
            allPaths ← allPaths ∪ conds(path) ∪ s
for all path ∈ allPaths do
    for all stmt ∈ path do
        usedFields ← usedFields ∪
        fieldsIn(getUseDef(stmt))
return paramFields − usedFields
```

Figure 6: The analyzer detection algorithm for projection. The *paramFields* **enumerates all the fields from the serialized** `key`, `value` **pair. The function** *fieldsIn(useDefChain)* **returns the parameter fields that appear in the passed-in series of statements.**

Projection & Compression: trivial (not as critical as selection, data is less dependent, most like to be safe)

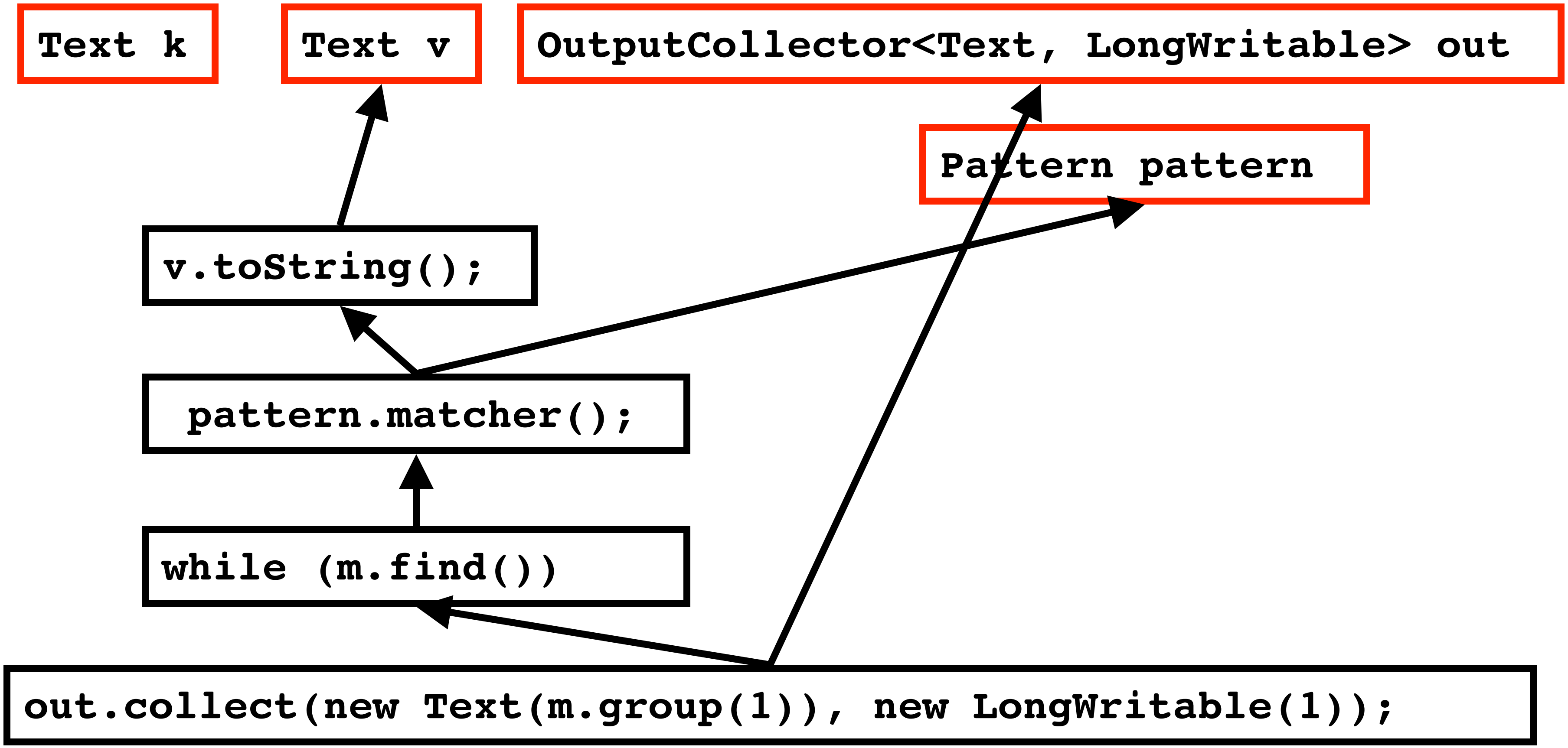

Thank you very much!
Questions!

Analysis Algorithm

```
public void map(Text k, Text v,
               OutputCollector<Text, LongWritable> out){
 Matcher m = pattern.matcher(v.toString());
 while (m.find()) {
   out.collect(new Text(m.group(1)),
               new LongWritable(1));
 }
}
```

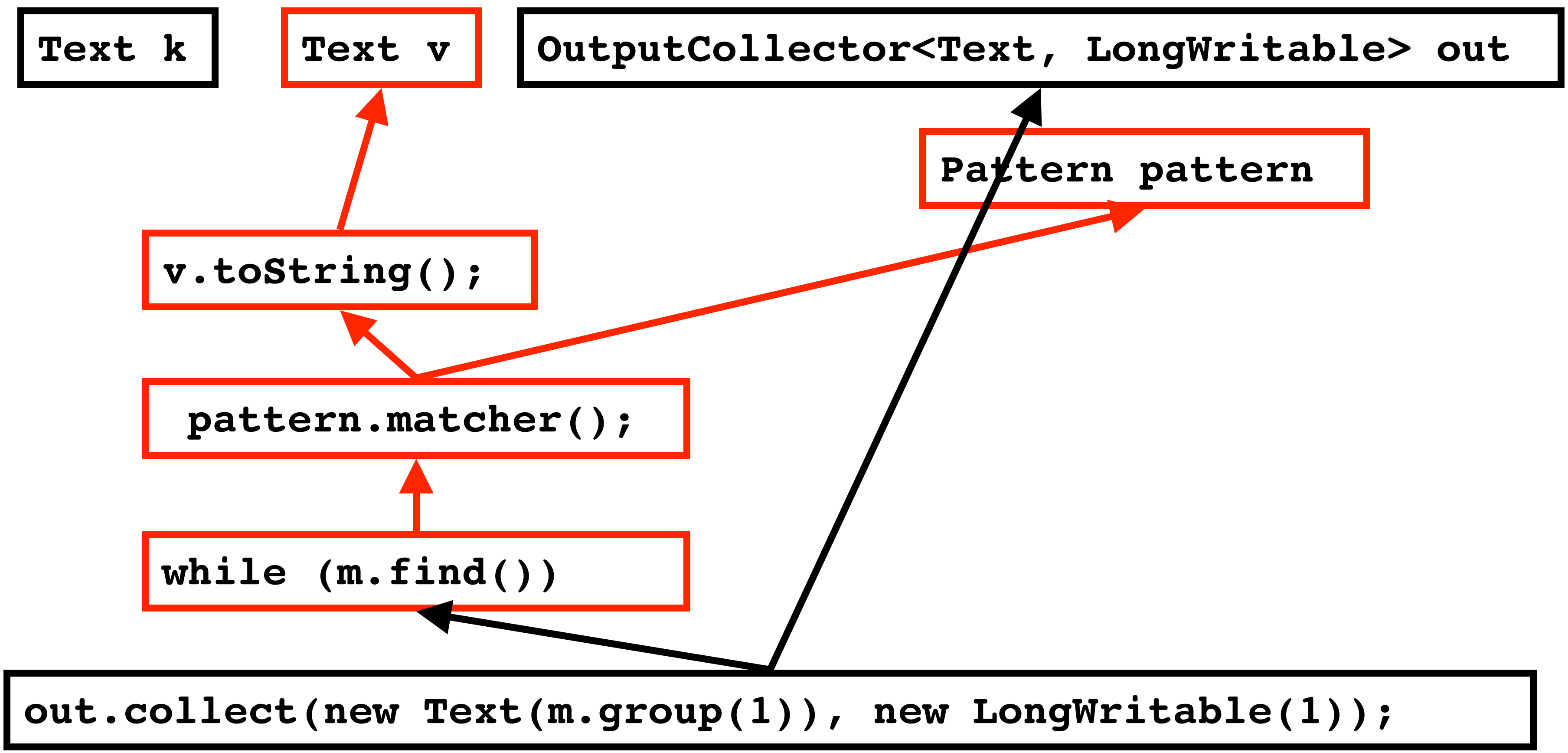
- Step 1: Construct DAG representation of user code

Analysis Algorithm



- Step 2: Check variable dependencies in while… test, check if chain is strictly functional, deps on inputs

Analysis Algorithm



- Step 3: Check if selection criteria can be evaluated w/ efficient available technique, eg, B+Tree

Experiments: Analysis

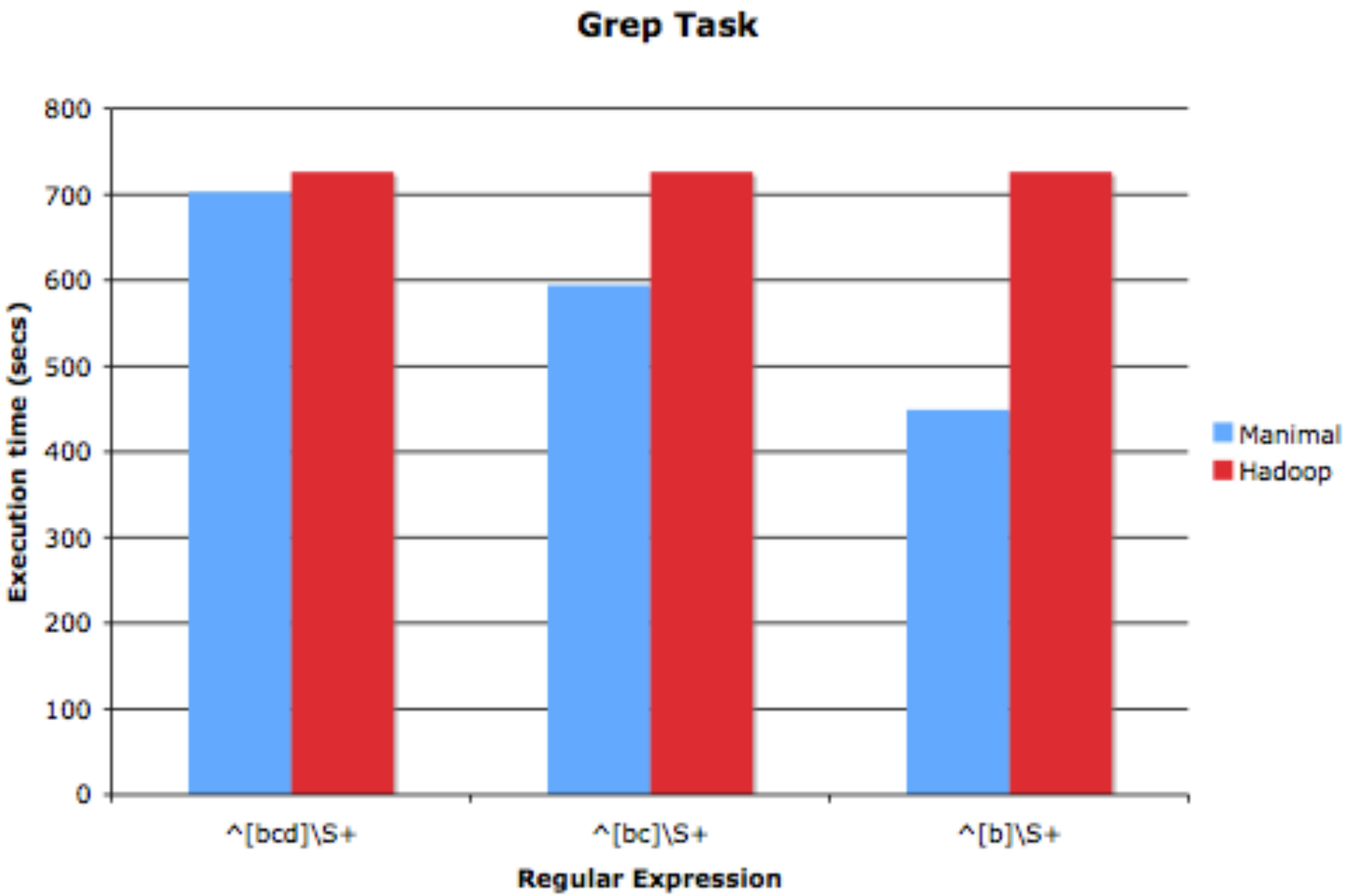
- Test MapReduce programs:
  - Four from Pavlo, SIGMOD '09
  - Twelve from Mahout project
- Of 16 input programs, 4 had selection-style test
- Manimal detected 3 of 4 optimization opportunities, and no unsafe opts.
  - Missed optimization requires knowledge of java.util.Hashtable semantics

Experiments: Performance

- Optimize "regexp line counter" map() from beforehand:

```
public void map(Text k, Text v,
               OutputCollector<Text, LongWritable> out){
 Matcher m = pattern.matcher(v.toString());
 while (m.find()) {
   out.collect(new Text(m.group(1)),
               new LongWritable(1));
 }
}
```

- 10 cluster nodes, 100GB of random-gen text
- Three regexps, all apply to start of line
- Selectivity varies 16.1% - 4.6%



- Runs in 63% of the time as conventional MR

Future Optimizations

- Using few fields of deserialized object? (E.g., a Web page)
  - Analyzer determines accessed fields
  - Employ column-oriented storage to avoid reading unnecessary bytes
- Using relatively large keys? (E.g., URLs)
  - Analyzer determines if keys are used exclusively in equality tests
  - Operate directly on compressed data
- Later: joins, HAVING clauses in reduce()
- Even later: optimizing non-perf criteria

Conclusions

- Manimal provides framework for applying well-known optimization techniques to MapReduce
  - Safe and effective analysis technique
  - Demonstrated perf. gain for selection
  - Points way to more sophisticated MapReduce execution