THE DECLARATIVE IMPERATIVE

EXPERIENCES AND CONJECTURES IN DISTRIBUTED LOGIC

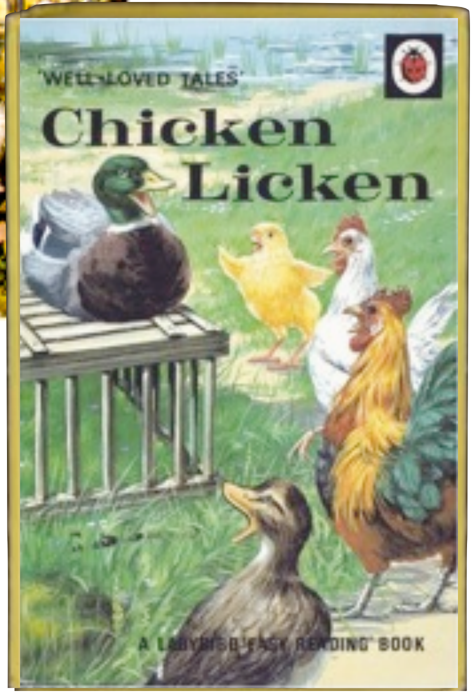
JOSEPH M HELLERSTEIN BERKELEY

TODAY

- two unfinished stories
  - urgency & resurgency
- a dedalus primer
- experience
- implications and conjecture

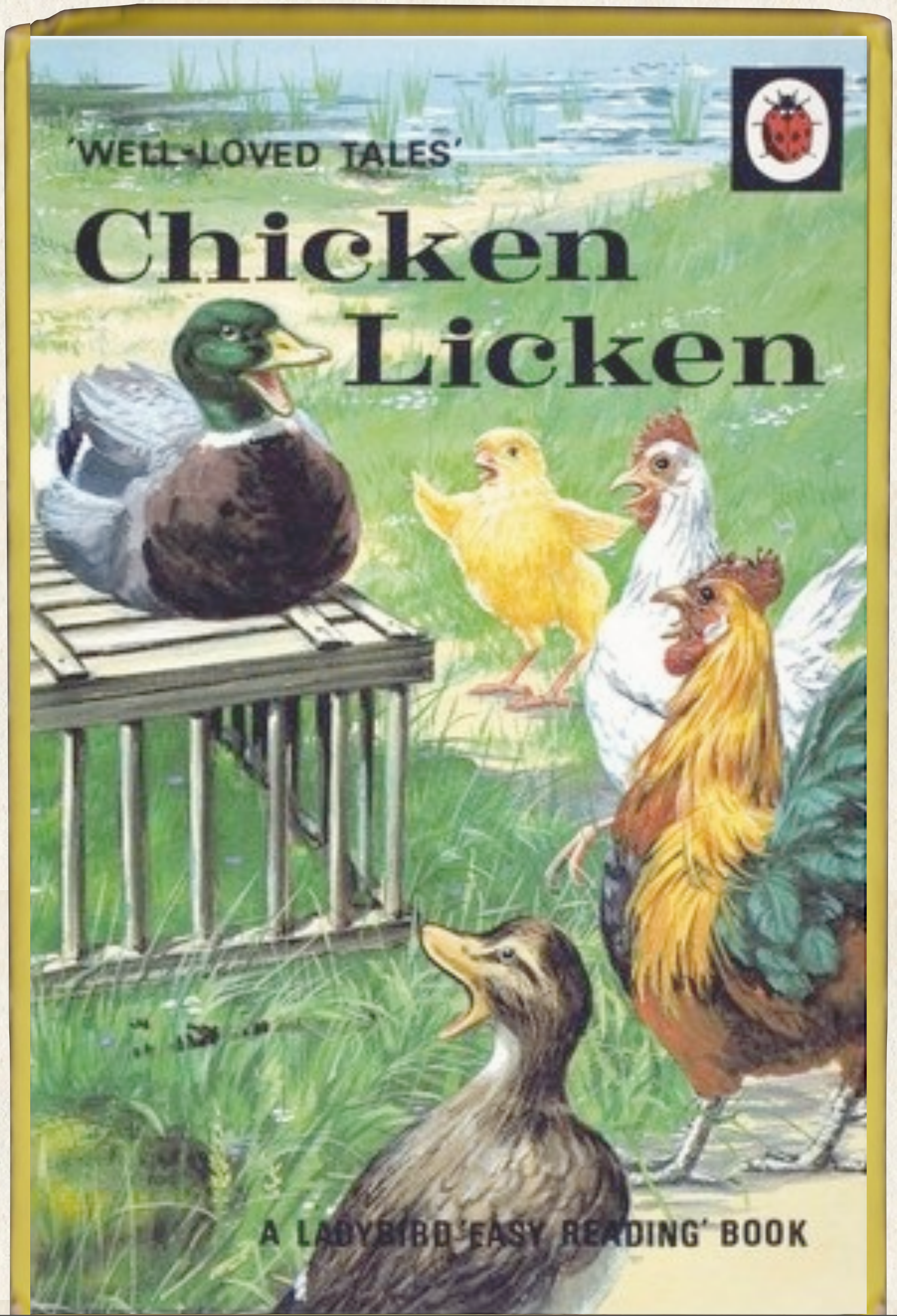
TODAY

- two unfinished stories
  - urgency & resurgency
- a dedalus primer
- experience
- implications and conjecture



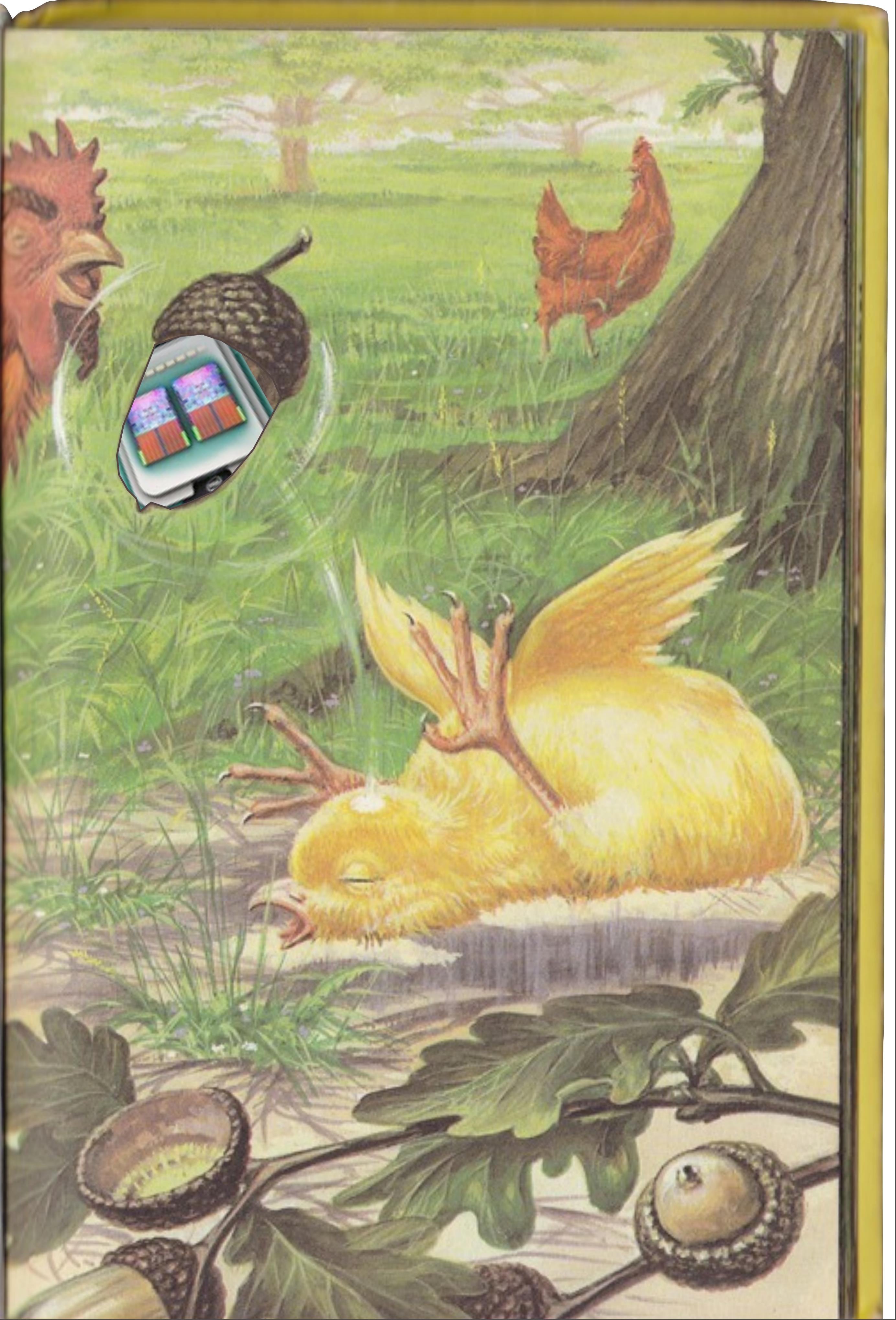
STORY #1: URGENCY

A.K.A.
The Programming Crisis



DOOM AND GLOOM

Once upon a time there was a little chicken called Chicken Licken. One day, processor clock speeds stopped following Moore's Law. Instead, hardware vendors started making multicore chips — one of which dropped on Chicken Licken's head.

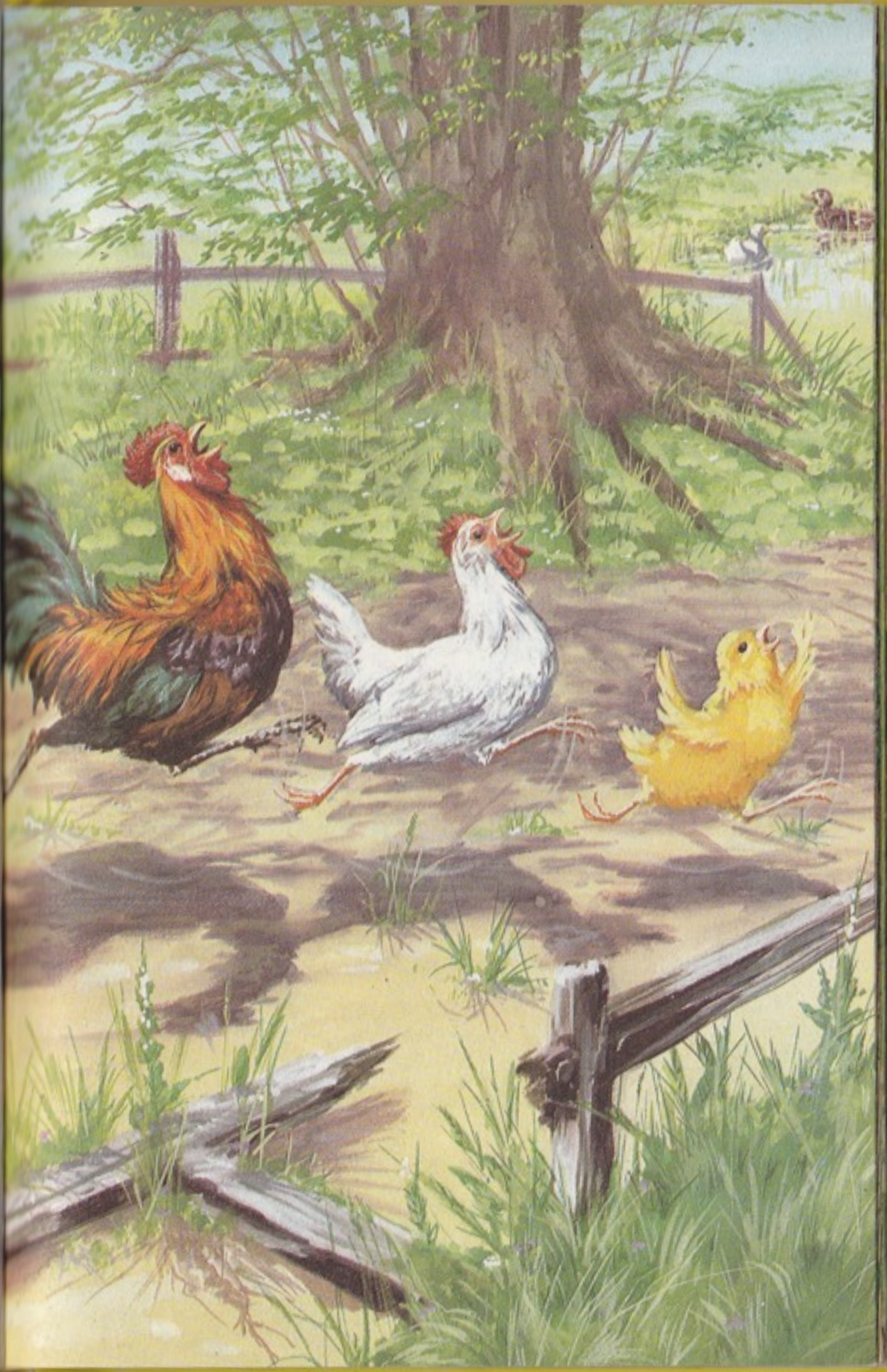


"The sky is falling! The sky is falling! Computers won't get any faster unless programmers learn to write parallel code!" squawked Chicken Licken.

Henny Penny clucked in agreement: "Worse, there is Cloud Computing on the horizon, and it requires programmers to write parallel AND distributed code!"

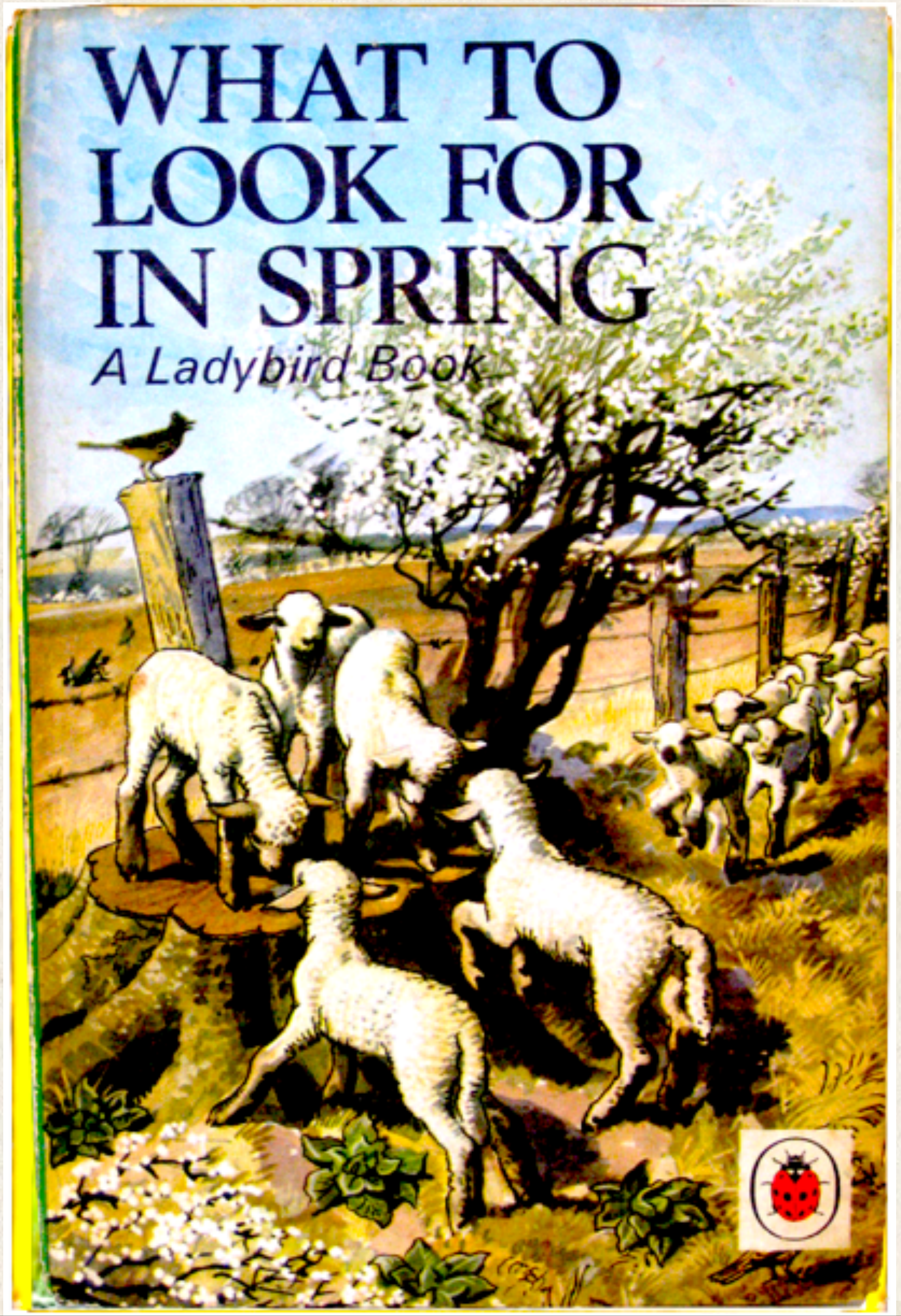
"I would be panicked if I were in industry!" said John Hennessy, then President of Stanford University.

Many of his friends agreed, and together they set off to tell the funding agencies.



STORY #2: RESURGENCY

A.K.A.
Springtime for Datalog



SPRINGTIME FOR DATALOG

In a faraway land, database theoreticians had reason for cheer. Datalog variants, like crocuses in the snow, were cropping up in fields well outside the walled garden of PODS where they were first sown.

http://www.flickr.com/photos/14293046@N00/3451413312/

Many examples of Datalog were blossoming:

- security protocols
- compiler analysis
- natural language processing
- probabilistic inference
- modular robotics
- multiplayer games

And, in a patch of applied ground in Berkeley, a small group was playing with Datalog for networking and distributed systems.

Spring, John Collier

The Berkeley folk named their project BOOM, short for the Berkeley Orders Of Magnitude project. The name commemorated Jim Gray's twelfth grand challenge, to make it Orders Of Magnitude easier to write software.

They also chose a name for the language in the BOOM project:

Bloom.

THE END OF THE STORY?

Doom and Gloom?

BOOM and Bloom!

THE END OF THE STORY?

Doom and Gloom?

- be not chicken licken!

BOOM and Bloom!

THE END OF THE STORY?

Doom and Gloom?

- be not chicken licken!
- give in to spring fever

BOOM and Bloom!

http://www.flickr.com/photos/60145846@N00/258950784/

THE DECLARATIVE IMPERATIVE

- a dark period for programming, yes.
- but we have seen the light ... long ago!
  - 1980's:
    - parallel SQL
    - computationally complete extensions to query langauges
- a way forward: extend languages that parallelize easily
  - be not "embarrassed" by your parallelism
- spread the news: spring is dawning!
  - crisis is opportunity
  - go forth from the walled garden
  - be fruitful and multiply

http://www.flickr.com/photos/megpi/861969/

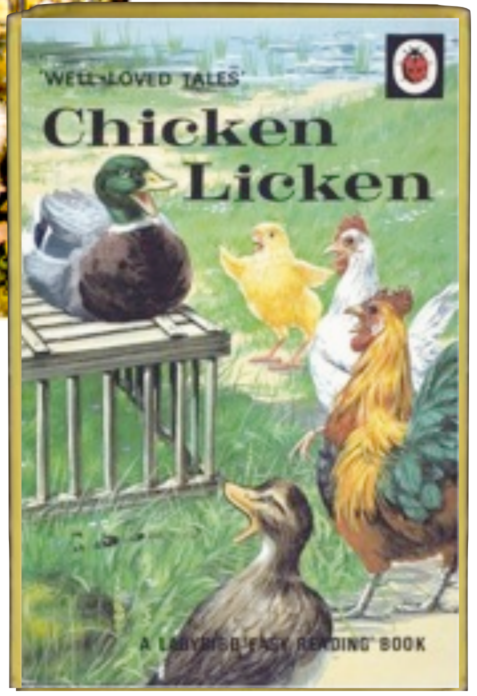
ALONG THE WAY: TASTY PODS STUFF

"Concepts are delicious snacks with which we try to alleviate our amazement"
— A. J. Heschel

- parallel complexity models for the cloud
- expressivity of logics w.r.t such models
- uncovering parallelism via LP properties
- semantics of distributed consistency
- time, time travel and fate

TODAY

- two unfinished stories
- a dedalus primer
- experience
- implications and conjecture



TODAY

- two unfinished stories
- a dedalus primer
- experience
- implications and conjecture

A BRIEF INTRODUCTION TO DEDALUS

http://ulyssesseen.com/landing/2009/04/stephen-dedalus/

Stephen Dedalus

A BRIEF INTRODUCTION TO DEDALUS

http://ulyssesseen.com/landing/2009/04/stephen-dedalus/

Stephen Dedalus

Daedalus (and Icarus)

DEDALUS IS DATALOG

+ stratified negation/aggregation

+ a `successor` relation

+ a common final attribute in every predicate

+ unification on that last attribute

BASIC DEDALUS

BASIC DEDALUS

- deductive rules

```
p(X, T) :- q(X, T).
```

(i.e. "plain old datalog", timestamps required)

BASIC DEDALUS

- deductive rules

  ```
  p(X, T) :- q(X, T).
  ```

  (i.e. "plain old datalog", timestamps required)

- inductive rules

  ```
  p(X, U) :- q(X, T), successor(T, U).
  ```

  (i.e. induction in time)

BASIC DEDALUS

- deductive rules

  `p(X, T) :- q(X, T).`

  (i.e. "plain old datalog", timestamps required)

- inductive rules

  `p(X, U) :- q(X, T), successor(T, U).`

  (i.e. induction in time)

- asynchronous rules

  `p(X, Z) :- q(X, T), choice({X, T}, {Z}).`

  (i.e. Z chosen non-deterministically per binding in the body [GZ98])

SUGARED DEDALUS

- deductive rules
  ```
  p(X, T) :- q(X, T).
  ```
- inductive rules
  ```
  p(X, U) :- q(X, T), successor(T, U).
  ```
- asynchronous rules
  ```
  p(X, Z) :- q(X, T), choice({X, T}, {Z}).
  ```

SUGARED DEDALUS

- deductive rules

```
p(X) :- q(X).
```

- inductive rules

```
p(X)@next :- q(X).
```

- asynchronous rules

```
p(X)@async :- q(X).
```

SUGARED DEDALUS

- deductive rules

```
p(X) :- q(X).
```

(omit ubiquitous timestamp attributes)

- inductive rules

```
p(X)@next :- q(X).
```

(sugar for induction in time)

- asynchronous rules

```
p(X)@async :- q(X).
```

(sugar for non-determinism in time)

A LITTLE PROGRAM

A LITTLE PROGRAM

```
state('flip')@1.
```

A LITTLE PROGRAM

```
state('flip')@1.


toggle('flop') :- state('flip').
```

A LITTLE PROGRAM

```
state('flip')@1.


toggle('flop') :- state('flip').

toggle('flip') :- state('flop').
```

A LITTLE PROGRAM

```
state(‘flip’)@1.


toggle(‘flop’) :- state(‘flip’).

toggle(‘flip’) :- state(‘flop’).

state(X)@next :- toggle(X).
```

A LITTLE PROGRAM

```
state('flip')@1.


toggle('flop') :- state('flip').

toggle('flip') :- state('flop').

state(X)@next :- toggle(X).

announcement(X)@async :- toggle(X).
```

PERSISTENCE:
BE PERSISTENT

PERSISTENCE: BE PERSISTENT

- "Accumulate-only" storage:

```
pods(X)@next :- pods(X).
pods('Ullman')@1982.
```

PERSISTENCE: BE PERSISTENT

- "Accumulate-only" storage:

```
pods(X)@next :- pods(X).
pods('Ullman')@1982.
```

- Updatable storage:

```
pods(X)@next :- pods(X), !del_pods(X).
pods('Libkin')@1996.
del_pods('Libkin')@2009.
```

PERSISTENCE: BE PERSISTENT

- "Accumulate-only" storage:

```
pods(X)@next :- pods(X).
pods('Ullman')@1982.
```

- Updatable storage:

```
pods(X)@next :- pods(X), !del_pods(X).
pods('Libkin')@1996.
del_pods('Libkin')@2009.
```

note: deletion via breaking induction
Libkin did publish in PODS '09

ATOMICITY & VISIBILITY

ATOMICITY & VISIBILITY

- Example: priority queue

ATOMICITY & VISIBILITY

- Example: priority queue

```
pq(V, P)@next :- pq(V, P), !del_pq(V, P).
```

ATOMICITY & VISIBILITY

- Example: priority queue

```
pq(V, P)@next :- pq(V, P), !del_pq(V, P).
qmin(min<P>) :- pq(V, P).
```

ATOMICITY & VISIBILITY

- Example: priority queue

```
pq(V, P)@next :- pq(V, P), !del_pq(V, P).
qmin(min<P>) :- pq(V, P).
```

← qmin "sees" only the current timestamp

ATOMICITY & VISIBILITY

- Example: priority queue

```
pq(V, P)@next :- pq(V, P), !del_pq(V, P).
qmin(min<P>) :- pq(V, P).        ← qmin "sees" only the current timestamp
del_pq(V,P) :- pq(V,P), qmin(P).
out(V,P)@next :- pq(V,P), qmin(P).
```

ATOMICITY & VISIBILITY

- Example: priority queue

```
pq(V, P)@next :- pq(V, P), !del_pq(V, P).
qmin(min<P>) :- pq(V, P).              ← qmin "sees" only the current timestamp
del_pq(V,P) :- pq(V,P), qmin(P).       } removes min from pq, adds to out.
out(V,P)@next :- pq(V,P), qmin(P).     } atomically visible at "next" time
```

ATOMICITY & VISIBILITY

- Example: priority queue

```
pq(V, P)@next :- pq(V, P), !del_pq(V, P).
qmin(min<P>) :- pq(V, P).              ← qmin "sees" only the current timestamp
del_pq(V,P) :- pq(V,P), qmin(P).        \  removes min from pq, adds to out.
out(V,P)@next :- pq(V,P), qmin(P).      /  atomically visible at "next" time
```

- Two Dedalus features working together:
  - timestamp unification controls visibility
  - temporal induction "synchronizes" timestamp assignment

TODAY

- two unfinished stories
- a dedalus primer
- experience
- implications and conjecture

TODAY

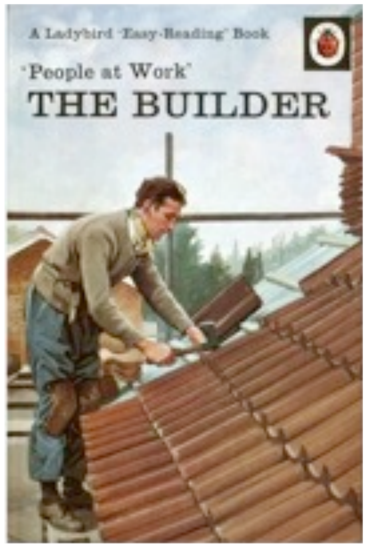
- two unfinished stories
- a dedalus primer
- experience
- implications and conjecture

TODAY

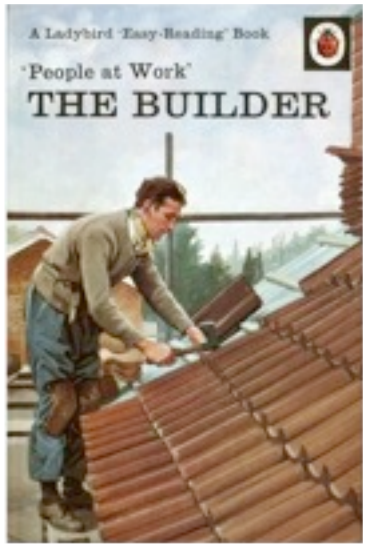
- two unfinished stories
- a dedalus primer
- experience
- implications and conjectur

WHAT TO LOOK FOR IN SPRING
A Ladybird Book
Chicken Licken
A LADYBIRD LEARNING TO READ BOOK
GOING TO SCHOOL
A Ladybird 'Easy-Reading' Book
'People at Work'
THE BUILDER

BUT FIRST, A GAME



EXPERIENCE

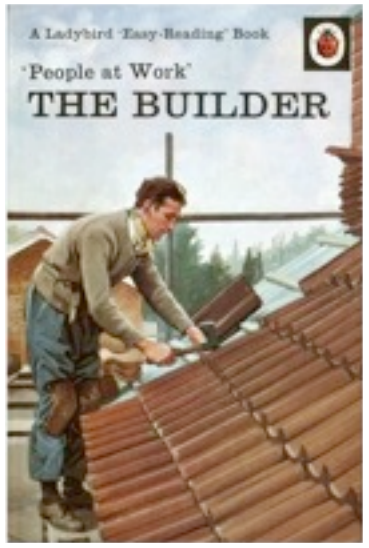


EXPERIENCE

No practical applications of recursive query theory ... have been found to date.

...

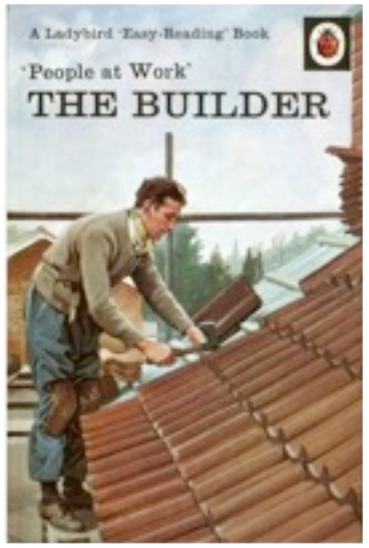
I find it sad that the theory community is so disconnected from reality that they don't even know why their ideas are irrelevant.



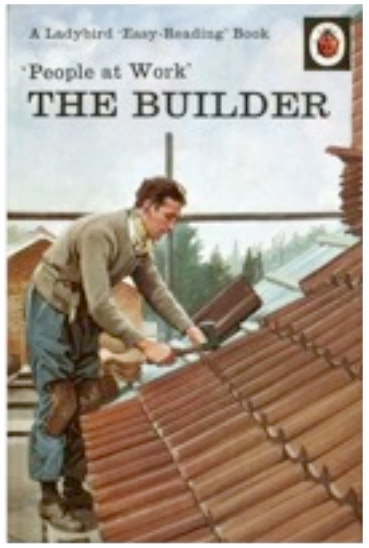
EXPERIENCE

No practical applications of recursive query theory ... have been found to date.
...
I find it sad that the theory community is so disconnected from reality that they don't even know why their ideas are irrelevant.

Hellerstein and Stonebraker,
Readings in Database Systems
3rd edition (1998)

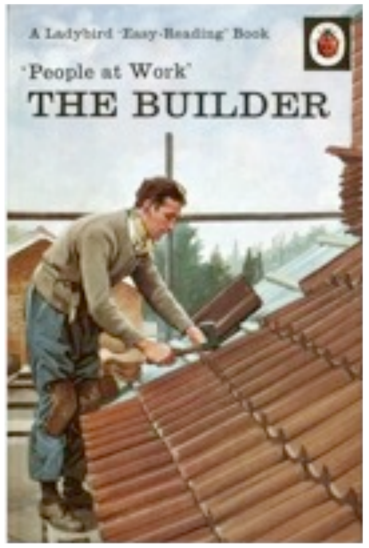


MORE EXPERIENCE



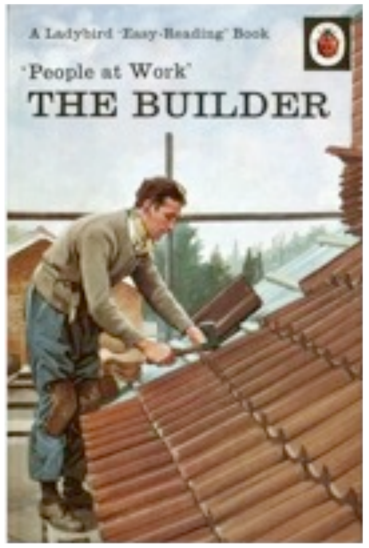
MORE EXPERIENCE

- In the last 7 years we have built



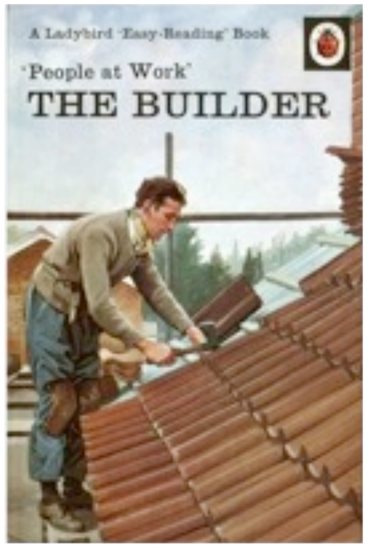
MORE EXPERIENCE

- In the last 7 years we have built
  - distributed crawlers [Coo04,Loo04]



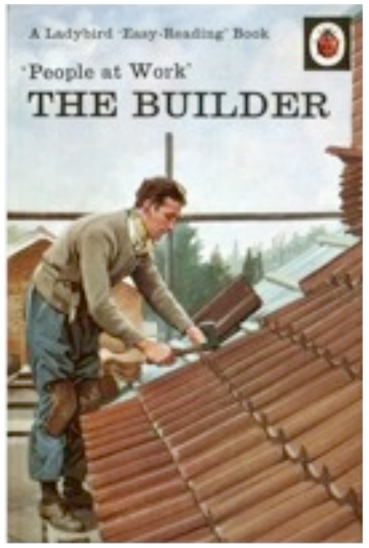
MORE EXPERIENCE

- In the last 7 years we have built
  - distributed crawlers [Coo04,Loo04]
  - network routing protocols [Loo05a,Loo06b]



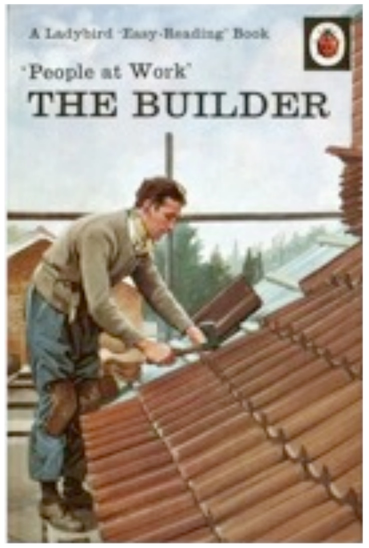
MORE EXPERIENCE

- In the last 7 years we have built
  - distributed crawlers [Coo04,Loo04]
  - network routing protocols [Loo05a,Loo06b]
  - overlay networks (e.g. Chord) [Loo06a]



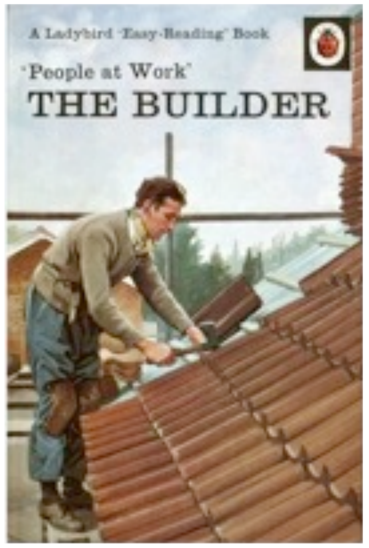
MORE EXPERIENCE

- In the last 7 years we have built
  - distributed crawlers [Coo04,Loo04]
  - network routing protocols [Loo05a,Loo06b]
  - overlay networks (e.g. Chord) [Loo06a]
  - a full-service embedded sensornet stack [Chu07]



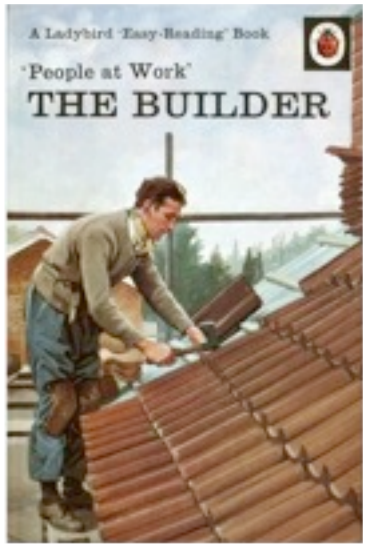
MORE EXPERIENCE

- In the last 7 years we have built
  - distributed crawlers [Coo04,Loo04]
  - network routing protocols [Loo05a,Loo06b]
  - overlay networks (e.g. Chord) [Loo06a]
  - a full-service embedded sensornet stack [Chu07]
  - network caching/proxying [Chu09]



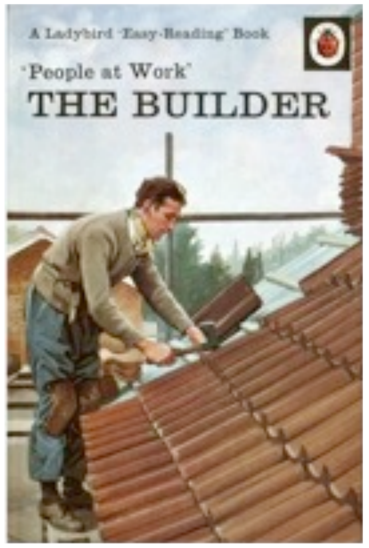
MORE EXPERIENCE

- In the last 7 years we have built
  - distributed crawlers [Coo04,Loo04]
  - network routing protocols [Loo05a,Loo06b]
  - overlay networks (e.g. Chord) [Loo06a]
  - a full-service embedded sensornet stack [Chu07]
  - network caching/proxying [Chu09]
  - relational query optimizers (System R, Cascades, Magic Sets) [Con08]



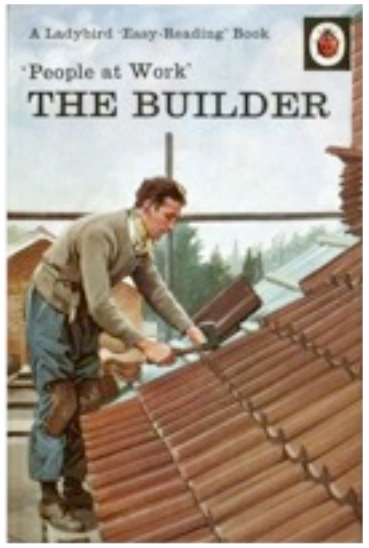
MORE EXPERIENCE

- In the last 7 years we have built
  - distributed crawlers [Coo04,Loo04]
  - network routing protocols [Loo05a,Loo06b]
  - overlay networks (e.g. Chord) [Loo06a]
  - a full-service embedded sensornet stack [Chu07]
  - network caching/proxying [Chu09]
  - relational query optimizers (System R, Cascades, Magic Sets) [Con08]
  - distributed Bayesian inference (e.g. junction trees) [Atul09]



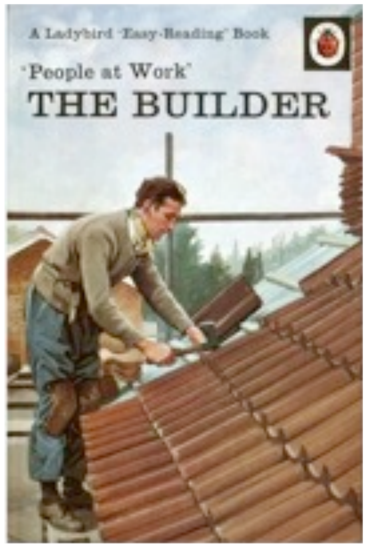
MORE EXPERIENCE

- In the last 7 years we have built
  - distributed crawlers [Coo04,Loo04]
  - network routing protocols [Loo05a,Loo06b]
  - overlay networks (e.g. Chord) [Loo06a]
  - a full-service embedded sensornet stack [Chu07]
  - network caching/proxying [Chu09]
  - relational query optimizers (System R, Cascades, Magic Sets) [Con08]
  - distributed Bayesian inference (e.g. junction trees) [Atul09]
  - distributed consensus and commit (Paxos, 2PC) [Alv09]



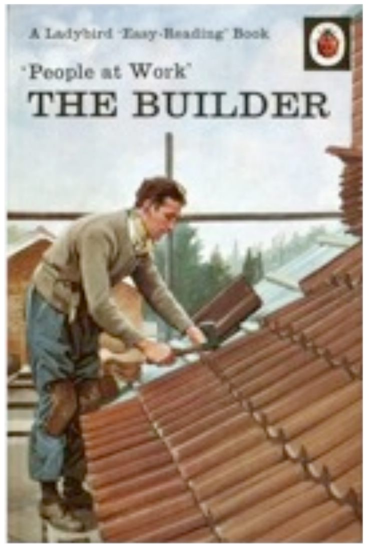
MORE EXPERIENCE

- In the last 7 years we have built
  - distributed crawlers [Coo04,Loo04]
  - network routing protocols [Loo05a,Loo06b]
  - overlay networks (e.g. Chord) [Loo06a]
  - a full-service embedded sensornet stack [Chu07]
  - network caching/proxying [Chu09]
  - relational query optimizers (System R, Cascades, Magic Sets) [Con08]
  - distributed Bayesian inference (e.g. junction trees) [Atul09]
  - distributed consensus and commit (Paxos, 2PC) [Alv09]
  - a distributed file system (HDFS) [Alv10]



MORE EXPERIENCE

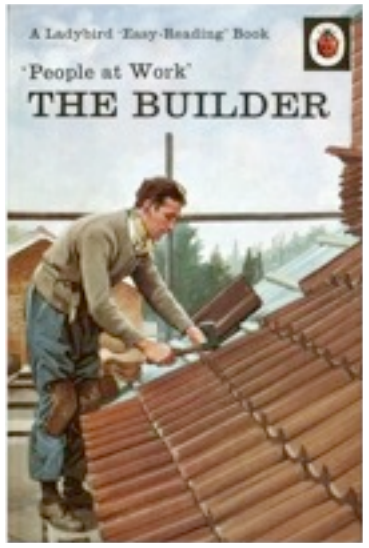
- In the last 7 years we have built
  - distributed crawlers [Coo04,Loo04]
  - network routing protocols [Loo05a,Loo06b]
  - overlay networks (e.g. Chord) [Loo06a]
  - a full-service embedded sensornet stack [Chu07]
  - network caching/proxying [Chu09]
  - relational query optimizers (System R, Cascades, Magic Sets) [Con08]
  - distributed Bayesian inference (e.g. junction trees) [Atul09]
  - distributed consensus and commit (Paxos, 2PC) [Alv09]
  - a distributed file system (HDFS) [Alv10]
  - map-reduce job scheduler [Alv10]



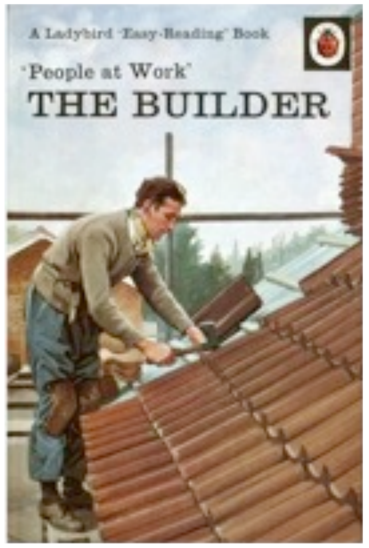
MORE EXPERIENCE

- In the last 7 years we have built
  - distributed crawlers [Coo04,Loo04]
  - network routing protocols [Loo05a,Loo06b]
  - overlay networks (e.g. Chord) [Loo06a]
  - a full-service embedded sensornet stack [Chu07]
  - network caching/proxying [Chu09]
  - relational query optimizers (System R, Cascades, Magic Sets) [Con08]
  - distributed Bayesian inference (e.g. junction trees) [Atul09]
  - distributed consensus and commit (Paxos, 2PC) [Alv09]
  - a distributed file system (HDFS) [Alv10]
  - map-reduce job scheduler [Alv10]

+ OOM smaller code
+ data independence (optimization)
− 90% declarative Datalog variants:
  Overlog, NDLog, SNLog, ...

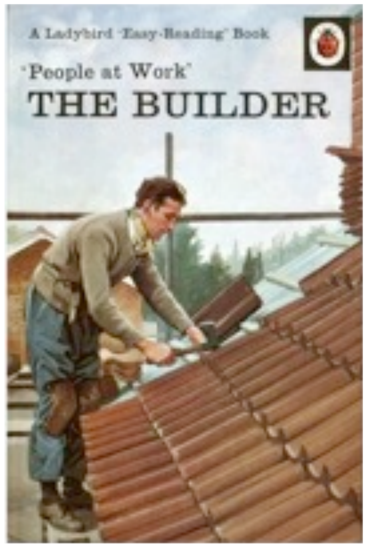


DESIGN PATTERNS



DESIGN PATTERNS

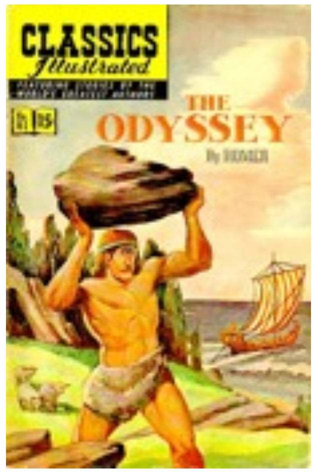
- despite flaws in our languages, patterns emerged
- three main categories today

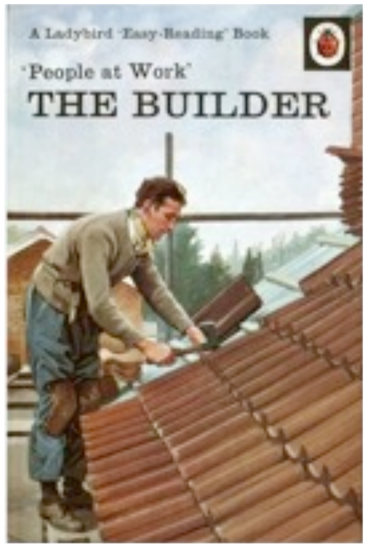


DESIGN PATTERNS

- despite flaws in our languages, patterns emerged
- three main categories today

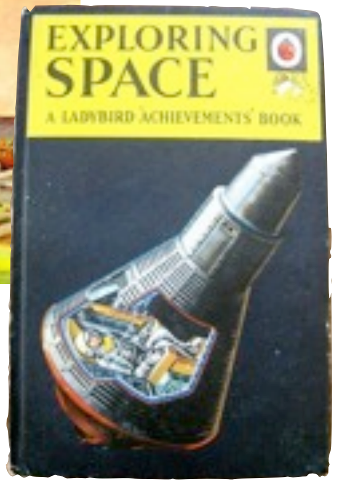
1. recursion ("rewriting the classics")

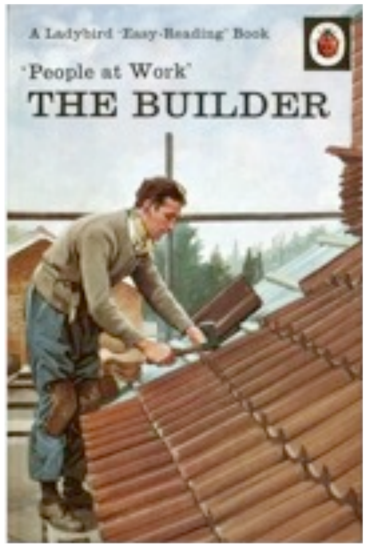




DESIGN PATTERNS

- despite flaws in our languages, patterns emerged
- three main categories today
  1. recursion ("rewriting the classics")
  2. communication across space-time





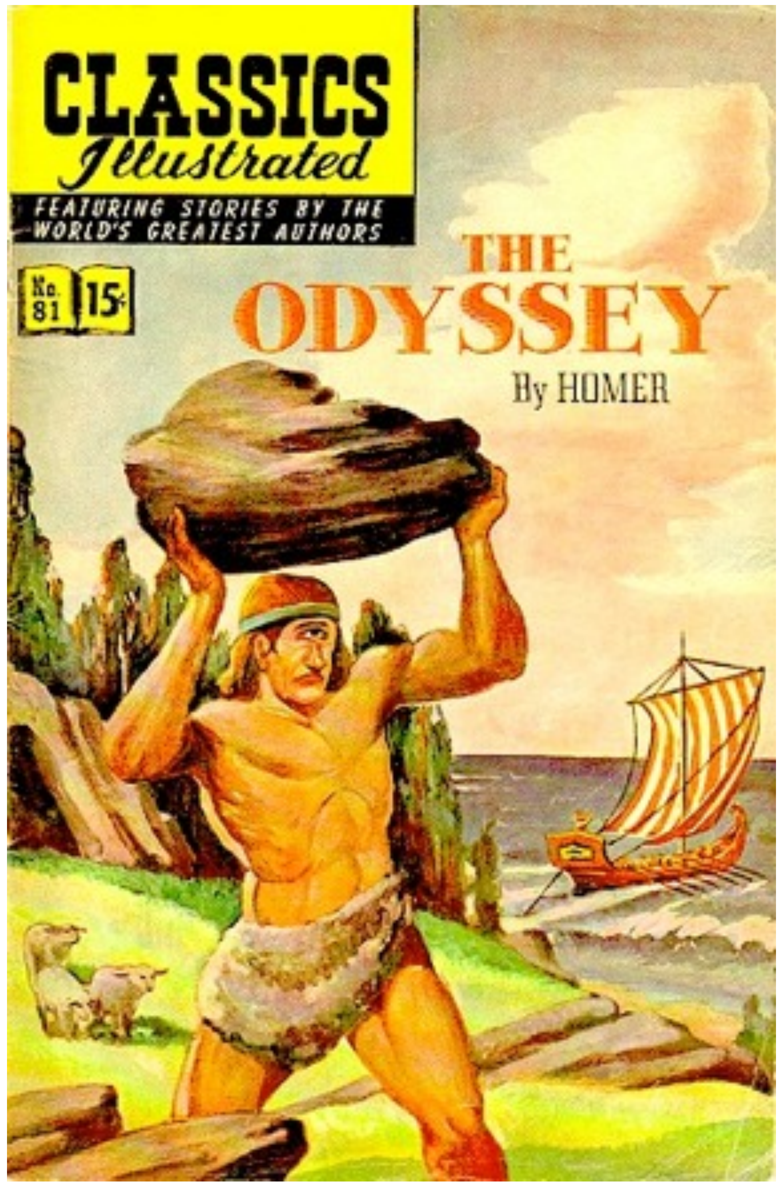
DESIGN PATTERNS

- despite flaws in our languages, patterns emerged
- three main categories today

1. recursion (“rewriting the classics”)
2. communication across space-time
3. engine architecture: threads/events

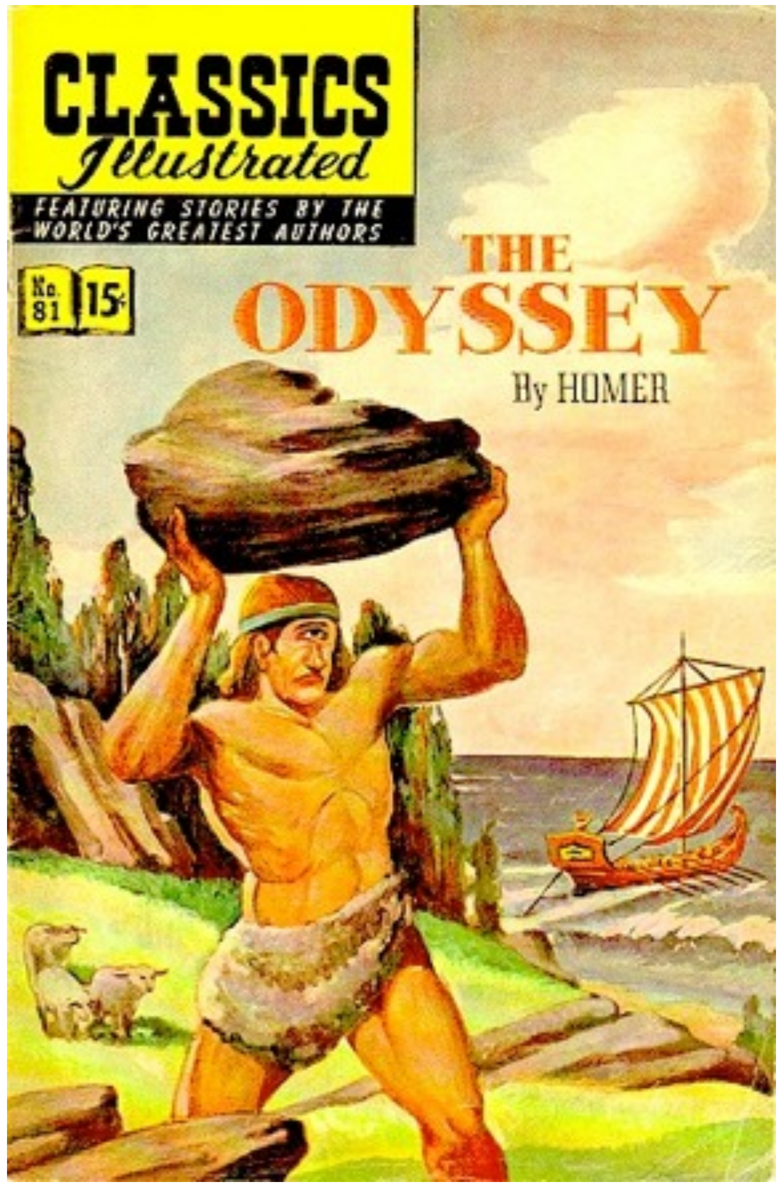
CLASSICS Illustrated
THE ODYSSEY
EXPLORING SPACE
A LADYBIRD ACHIEVEMENTS BOOK
A Ladybird Junior Science Book
LEVERS, PULLEYS and ENGINES

1. RECURSION (REWRITING THE CLASSICS)



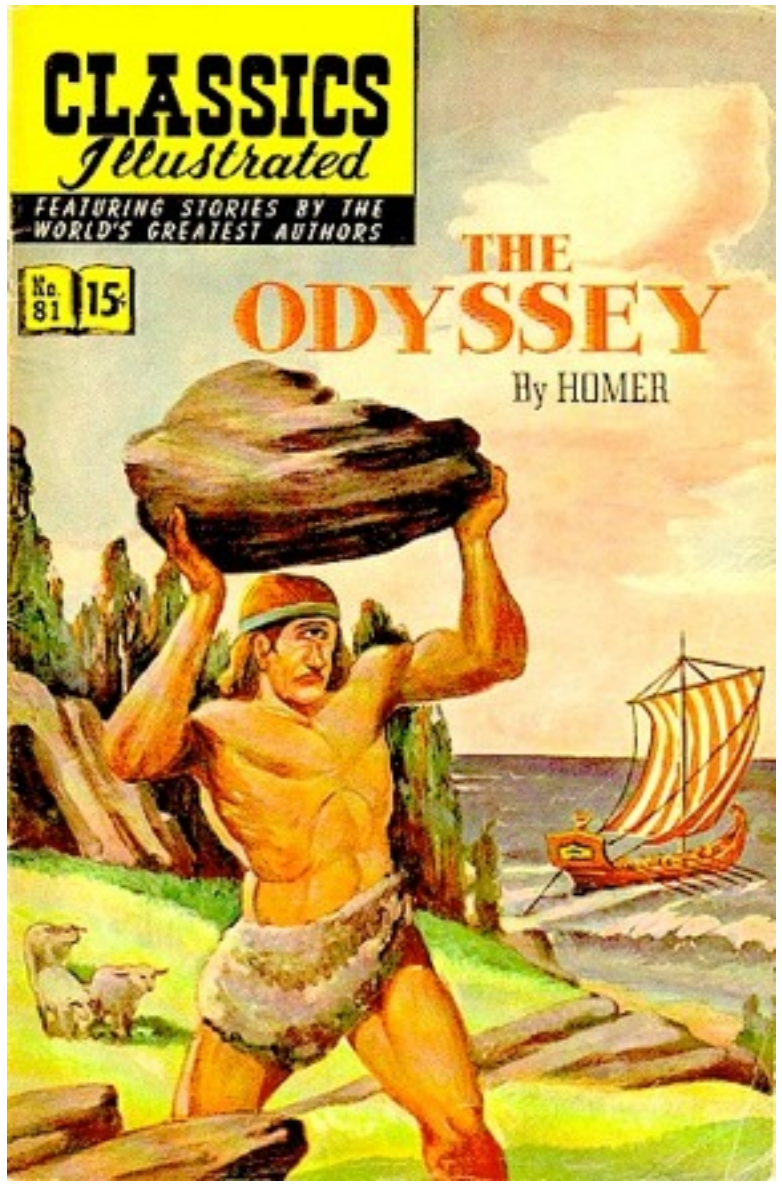
1. RECURSION (REWRITING THE CLASSICS)

- finding closure without the Ancs*



1. RECURSION (REWRITING THE CLASSICS)

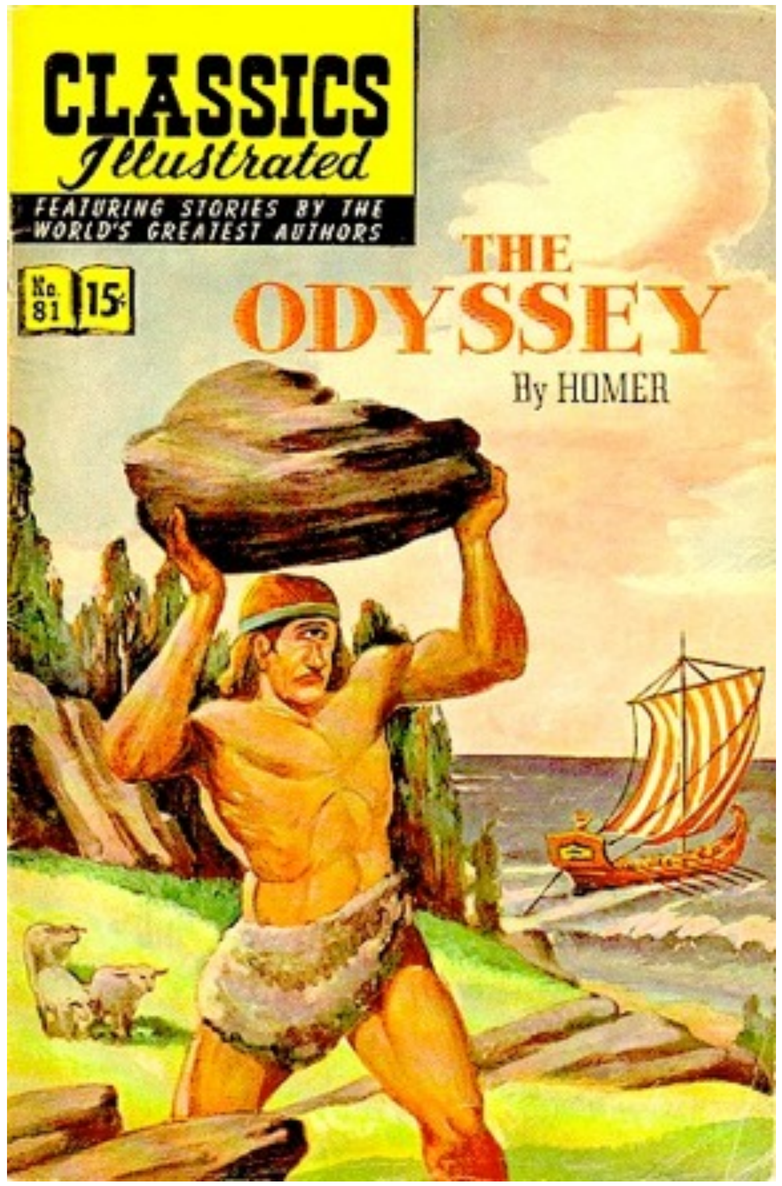
- finding closure without the Ancs*



* SIGMOD people can EMP-athize!

1. RECURSION (REWRITING THE CLASSICS)

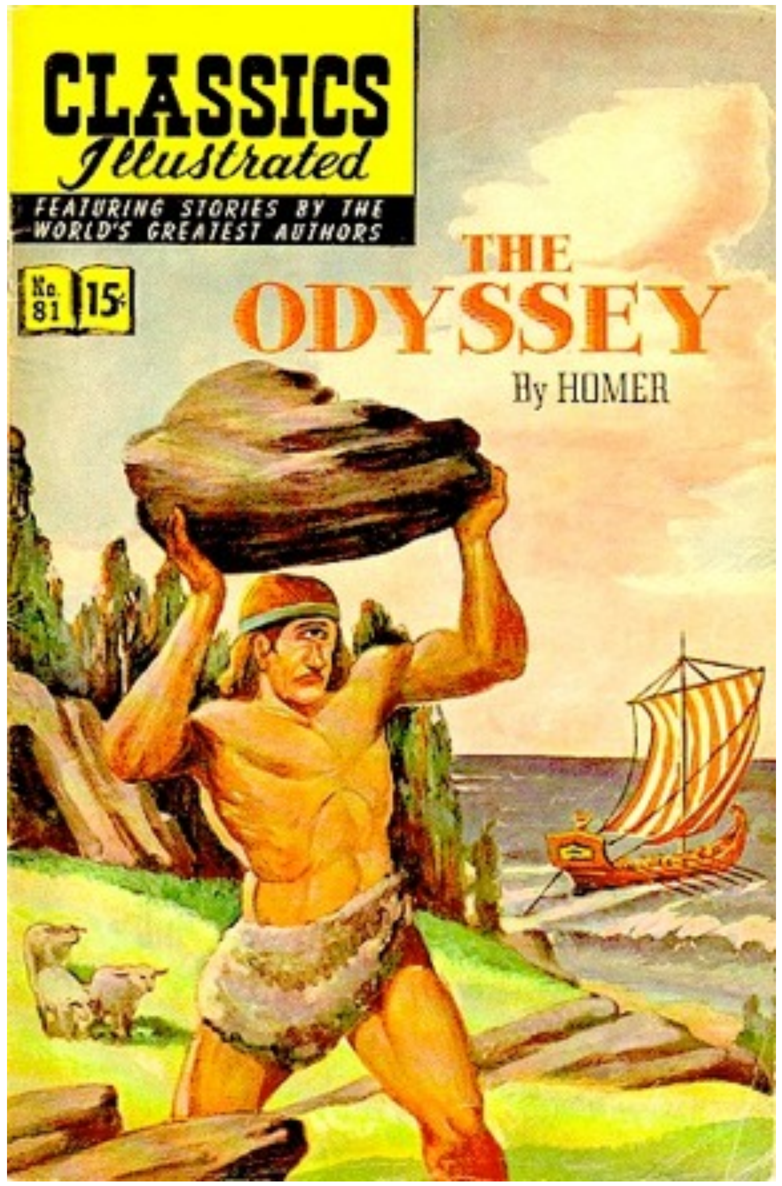
- finding closure without the Ancs*
  - the web is a graph.



* SIGMOD people can EMP-athize!

1. RECURSION (REWRITING THE CLASSICS)

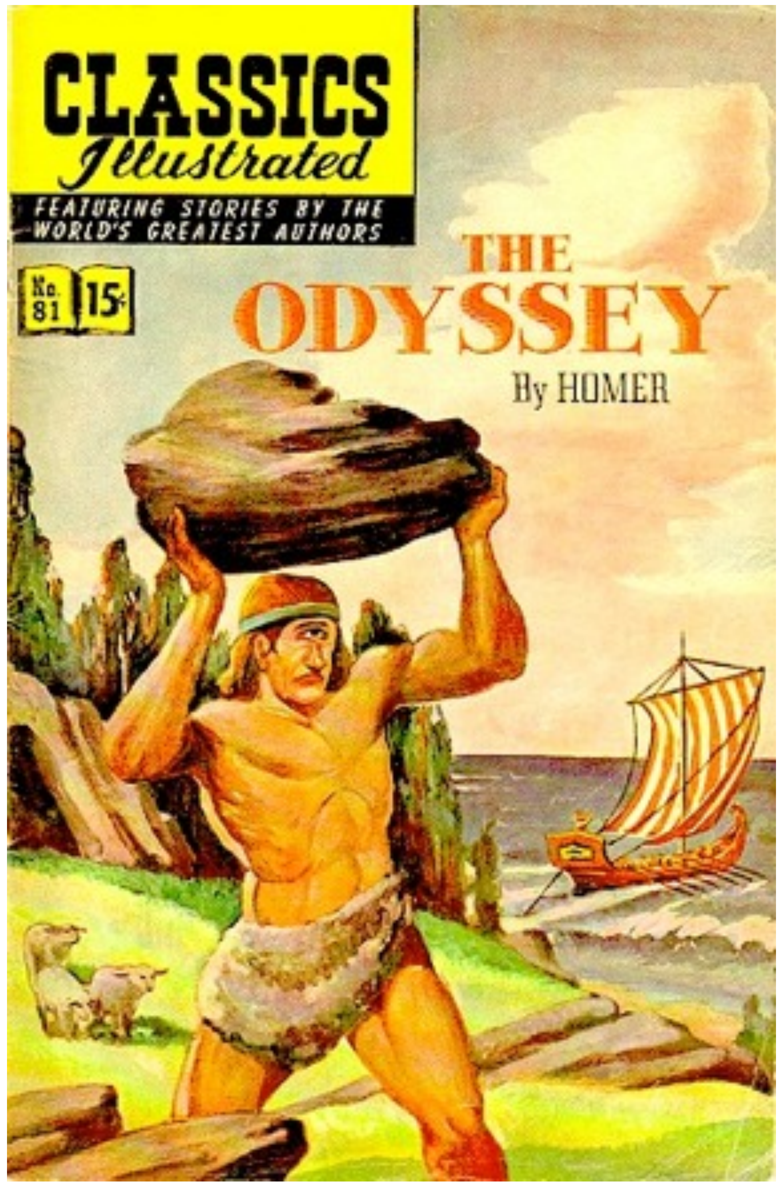
- finding closure without the Ancs*
  - the web is a graph.
    - e.g. crawlers = simple monotonic reachability



* SIGMOD people can EMP-athize!

1. RECURSION (REWRITING THE CLASSICS)

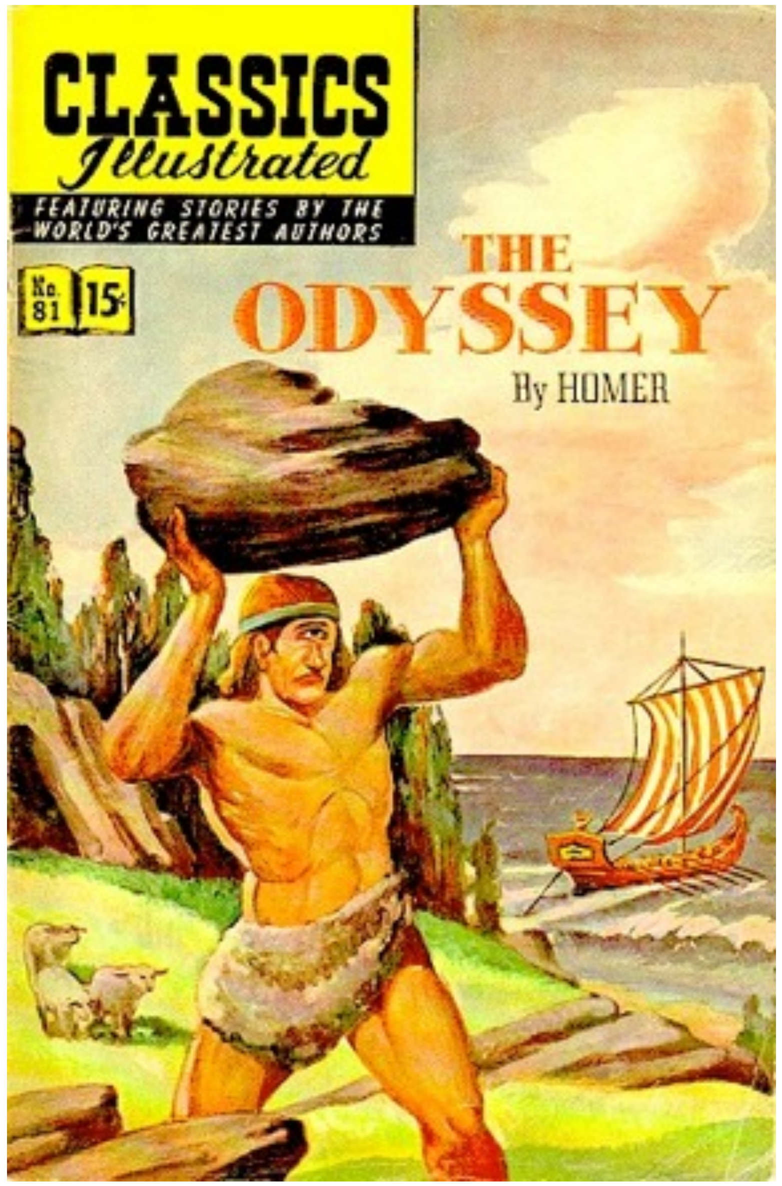
- finding closure without the Ancs*
  - the web is a graph.
    - e.g. crawlers = simple monotonic reachability
  - the internet is a graph.



* SIGMOD people can EMP-athize!

1. RECURSION (REWRITING THE CLASSICS)

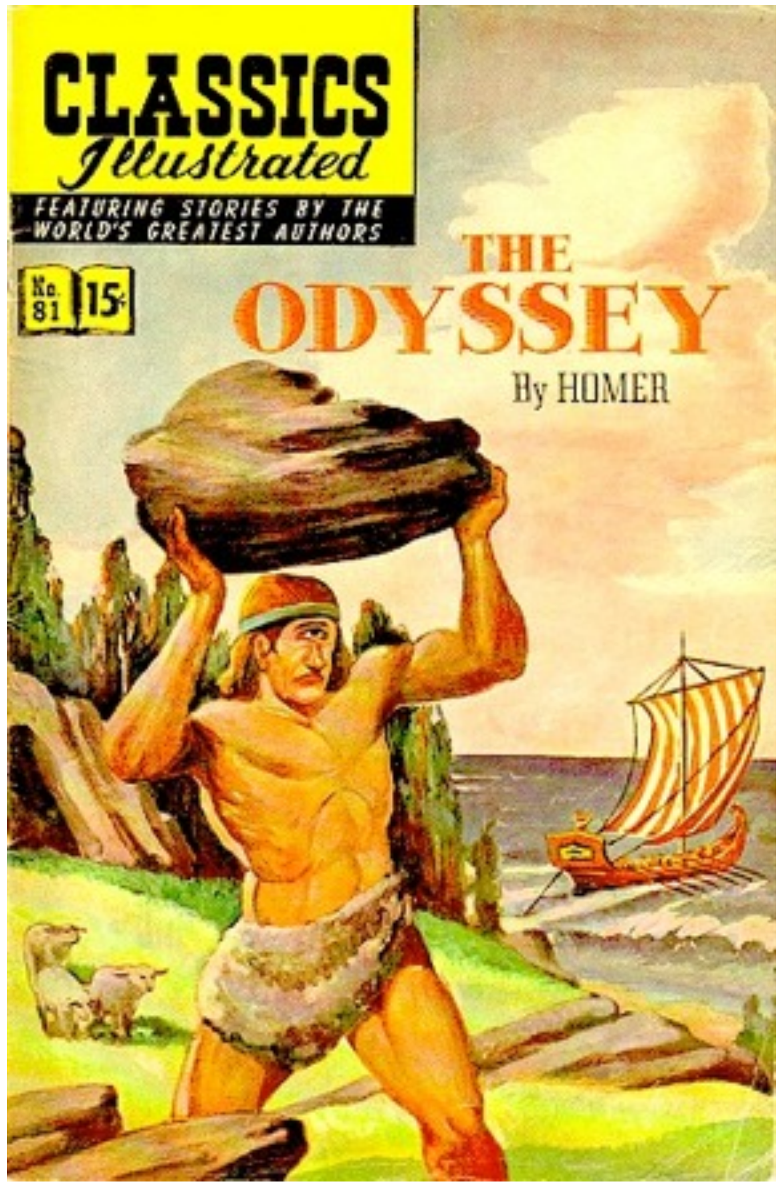
- finding closure without the Ancs*
  - the web is a graph.
    - e.g. crawlers = simple monotonic reachability
  - the internet is a graph.
    - e.g. routing protocols, overlay nets



* SIGMOD people can EMP-athize!

1. RECURSION (REWRITING THE CLASSICS)

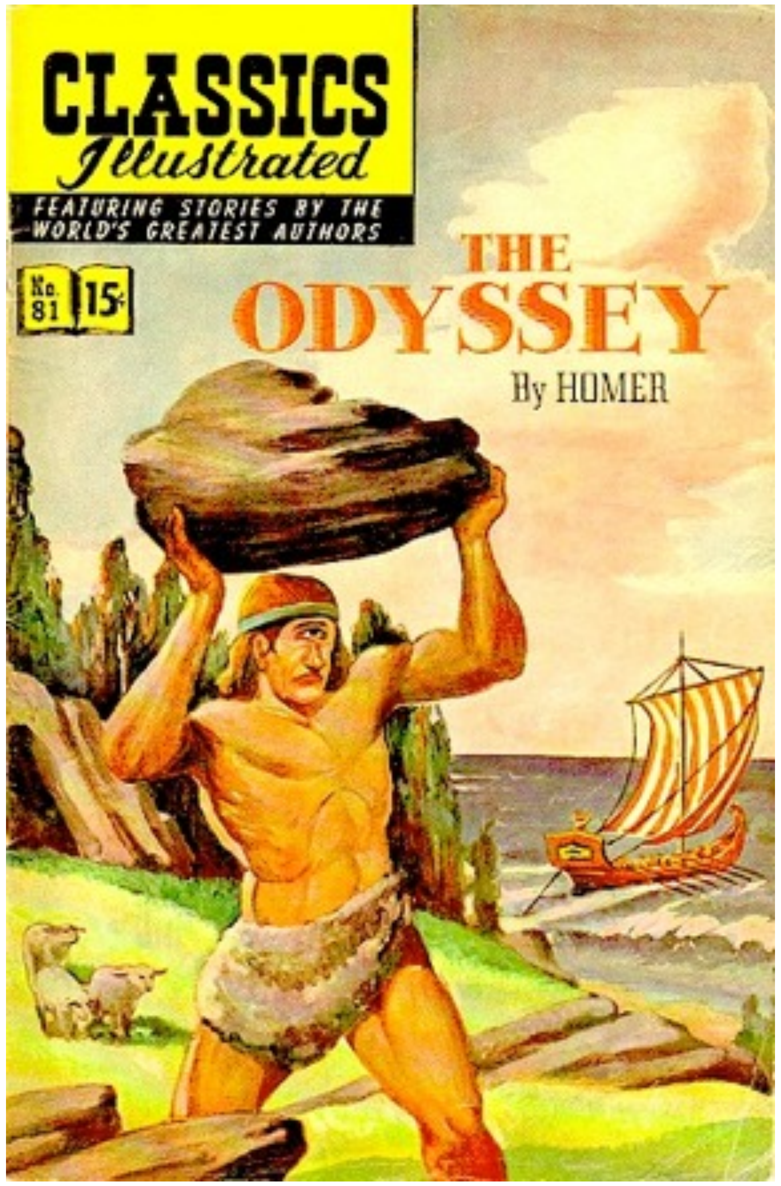
- finding closure without the Ancs*
  - the web is a graph.
    - e.g. crawlers = simple monotonic reachability
  - the internet is a graph.
    - e.g. routing protocols, overlay nets
  - recursive queries matter!
    - [Coo04,Loo04,Loo05,Loo06a,Loo06b]



* SIGMOD people can EMP-athize!

1. RECURSION (REWRITING THE CLASSICS)

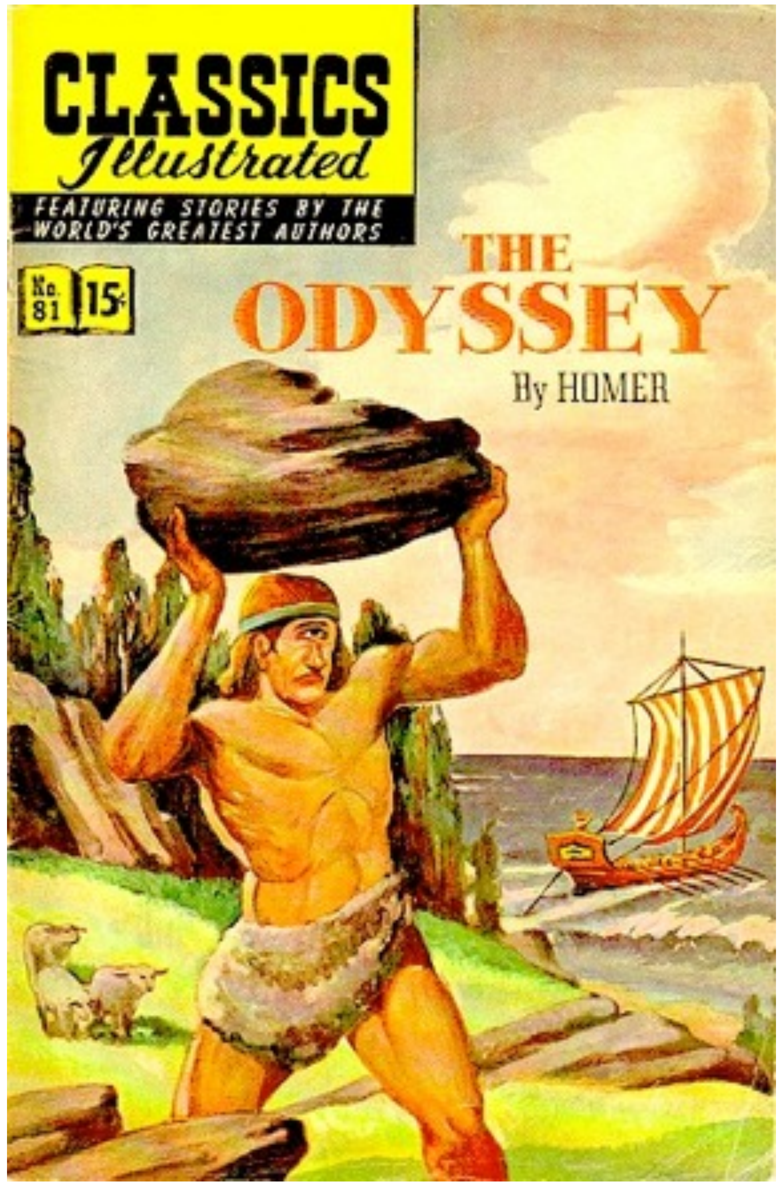
- finding closure without the Ancs*
  - the web is a graph.
    - e.g. crawlers = simple monotonic reachability
  - the internet is a graph.
    - e.g. routing protocols, overlay nets
  - recursive queries matter!
    - [Coo04,Loo04,Loo05,Loo06a,Loo06b]
- challenges:



* SIGMOD people can EMP-athize!

1. RECURSION (REWRITING THE CLASSICS)

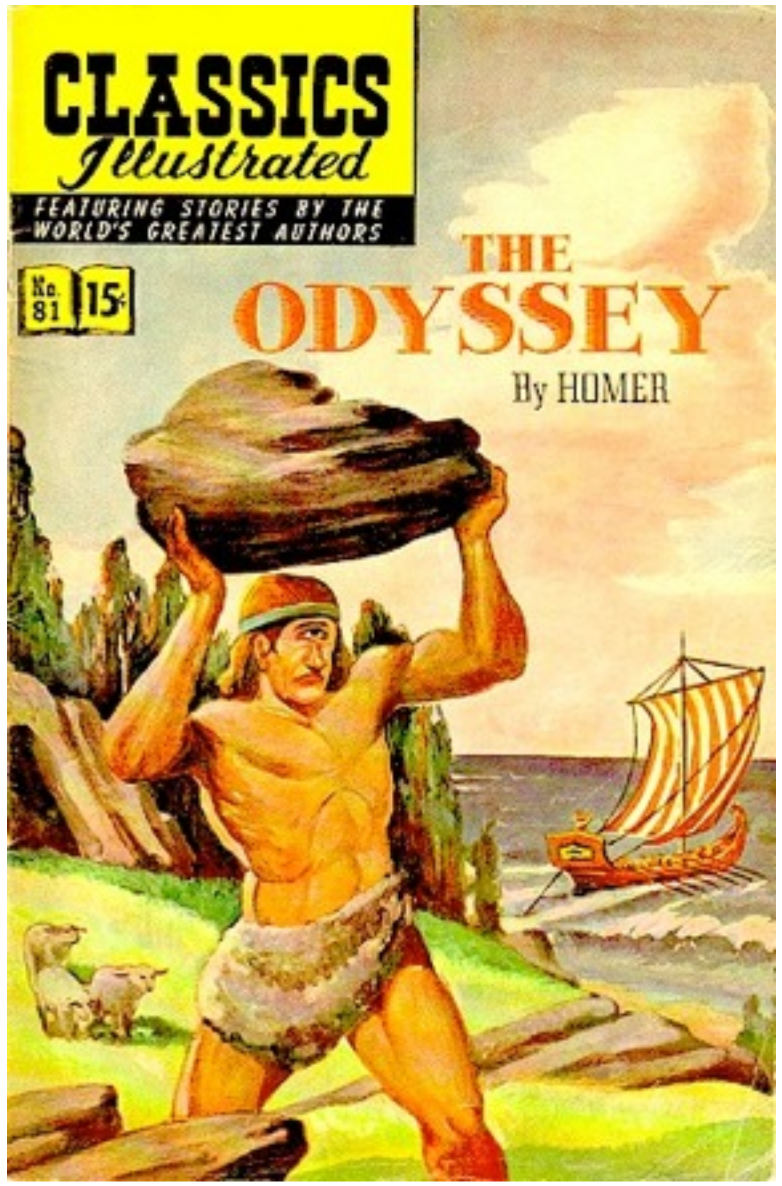
- finding closure without the Ancs*
  - the web is a graph.
    - e.g. crawlers = simple monotonic reachability
  - the internet is a graph.
    - e.g. routing protocols, overlay nets
  - recursive queries matter!
    - [Coo04,Loo04,Loo05,Loo06a,Loo06b]
- challenges:
  - distributed join semantics



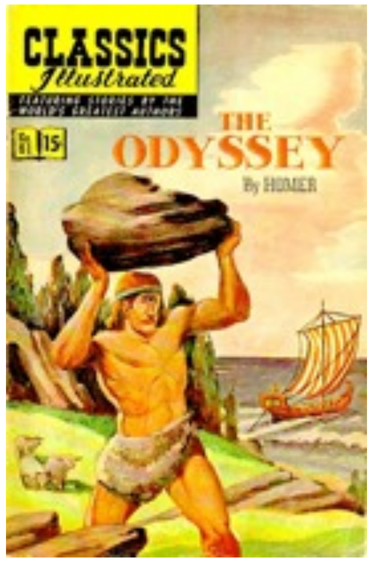
* SIGMOD people can EMP-athize!

1. RECURSION (REWRITING THE CLASSICS)

- finding closure without the Ancs*
  - the web is a graph.
    - e.g. crawlers = simple monotonic reachability
  - the internet is a graph.
    - e.g. routing protocols, overlay nets
  - recursive queries matter!
    - [Coo04,Loo04,Loo05,Loo06a,Loo06b]
- challenges:
  - distributed join semantics
  - asynchronous fixpoint computation



* SIGMOD people can EMP-athize!

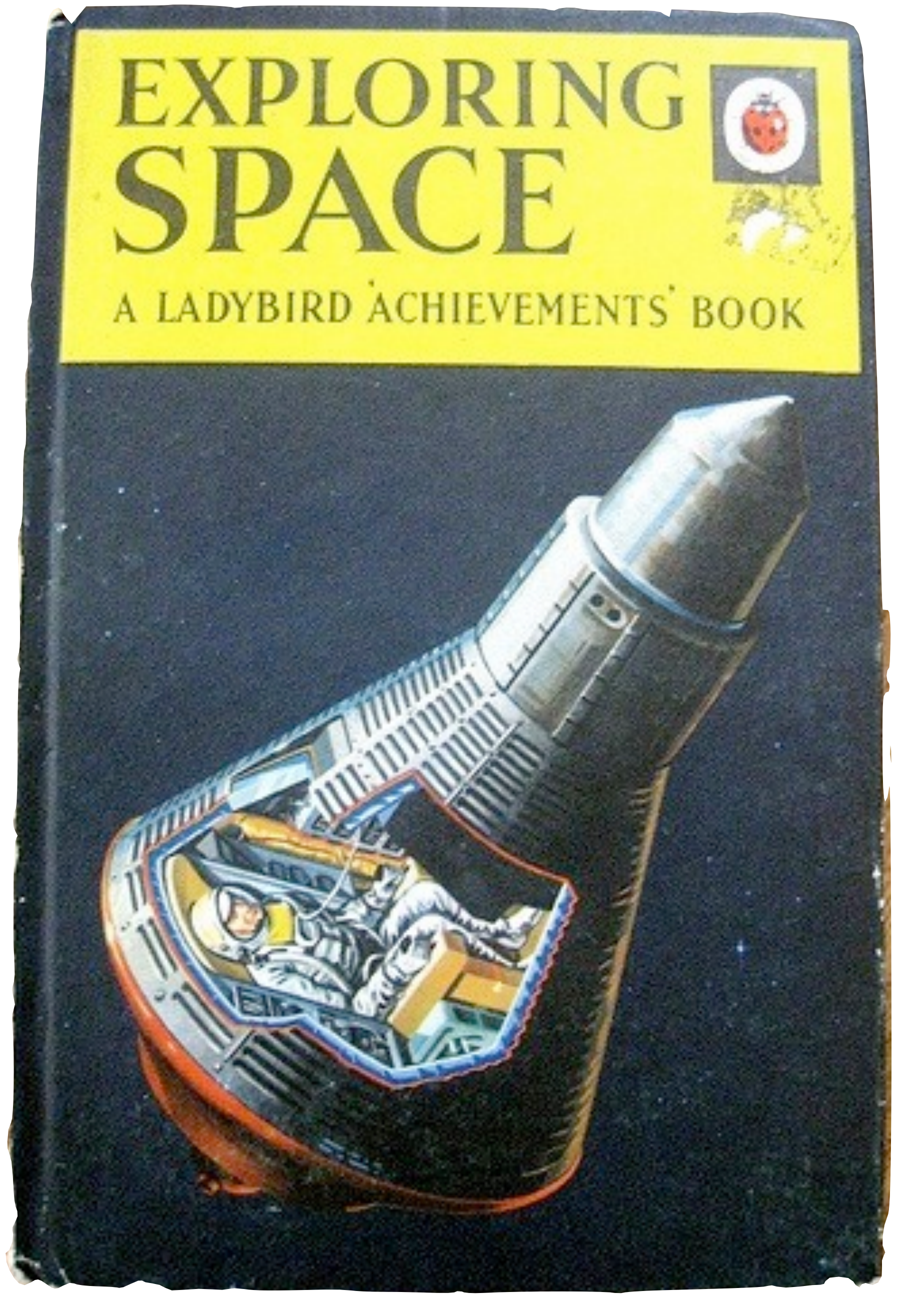


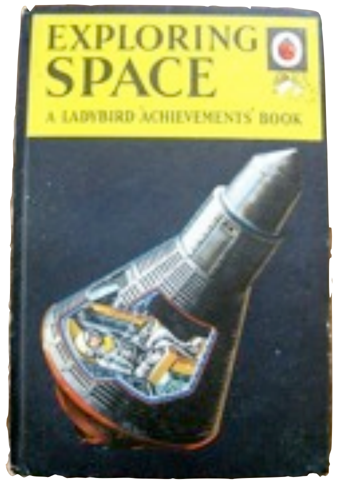
RECURSION + CHOICE = DYNAMIC PROGRAMMING

- many examples
  - shortest paths [Loo05,Loo06b]
  - query optimization
    - Evita Raced: an overlog optimizer written in overlog [Con08]
      - bottom-up and top-down DP written in datalog
  - Viterbi inference [Wan10]
- main challenge
  - distributed stratification

2. SPACE & COMMUNICATION

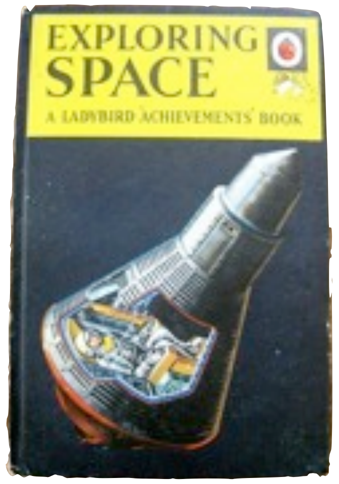
- location specifiers
  - partition a relation across machines
- communication “falls out”
  - declare each tuple’s “resting place”





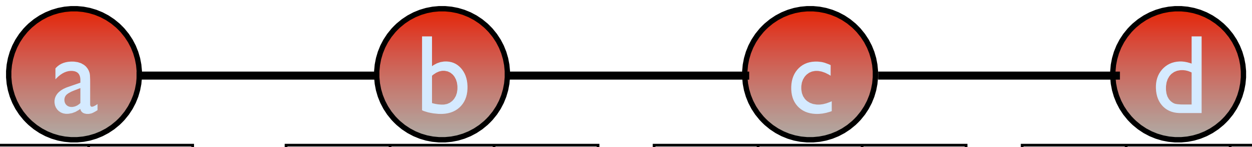
LOCSPECS INDUCE COMMUNICATION

- link(@X,Y,C)
- path(@X,Y,Y,C) :- link(@X,Y,C)
- path(@X,Z,Y,C+D) :- link(@X,Y,C), path(@Y,Z,N,D)



LOCSPECS INDUCE COMMUNICATION

- link(@X,Y,C)
- path(@X,Y,Y,C) :- link(@X,Y,C)
- path(@X,Z,Y,C+D) :- link(@X,Y,C), path(@Y,Z,N,D)



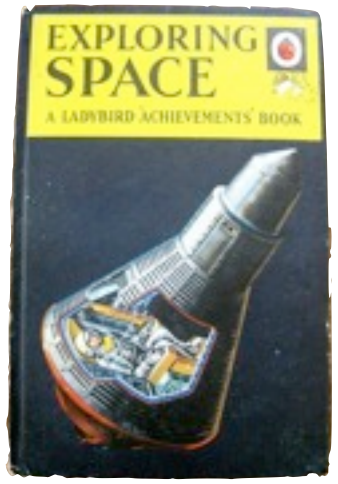
link:

a	b	1

b	a	1
b	c	1

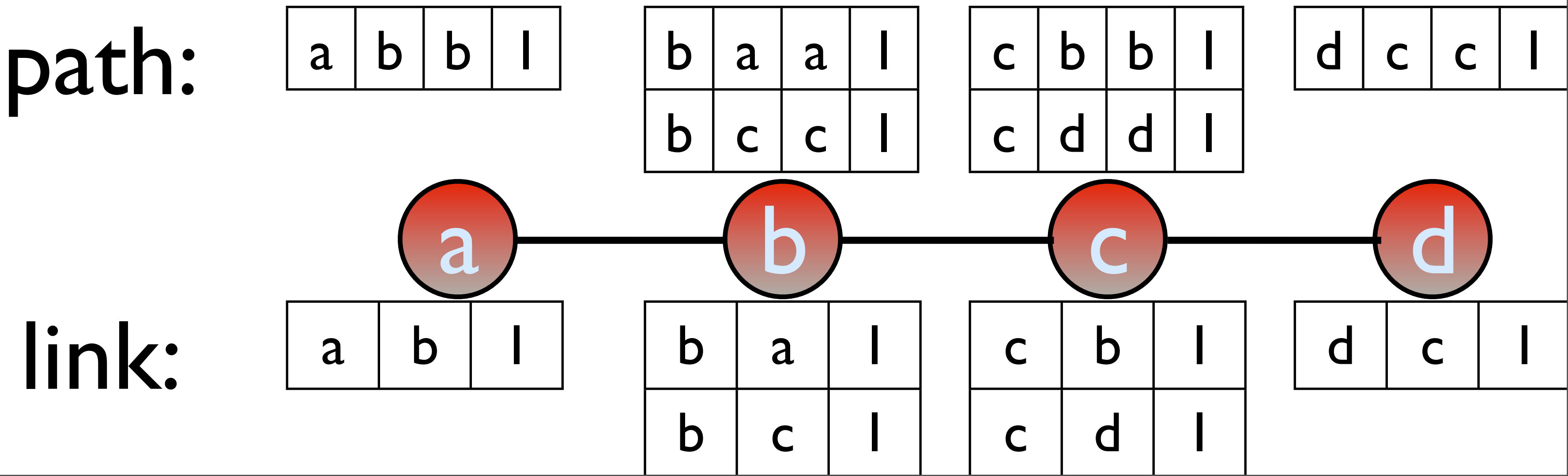
c	b	1
c	d	1

d	c	1



LOCSPECS INDUCE COMMUNICATION

- link(@X,Y,C)
- path(@X,Y,Y,C) :- link(@X,Y,C)
- path(@X,Z,Y,C+D) :- link(@X,Y,C), path(@Y,Z,N,D)



path at a:

a	b	b	1

path at b:

b	a	a	1
b	c	c	1

path at c:

c	b	b	1
c	d	d	1

path at d:

d	c	c	1

link at a:

a	b	1

link at b:

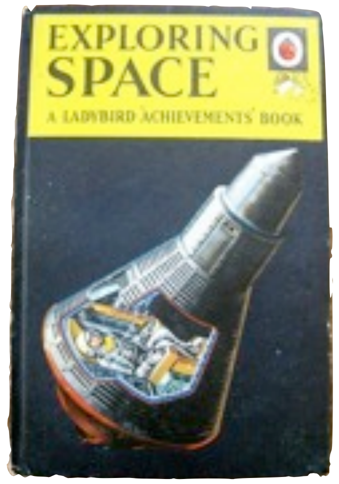
b	a	1
b	c	1

link at c:

c	b	1
c	d	1

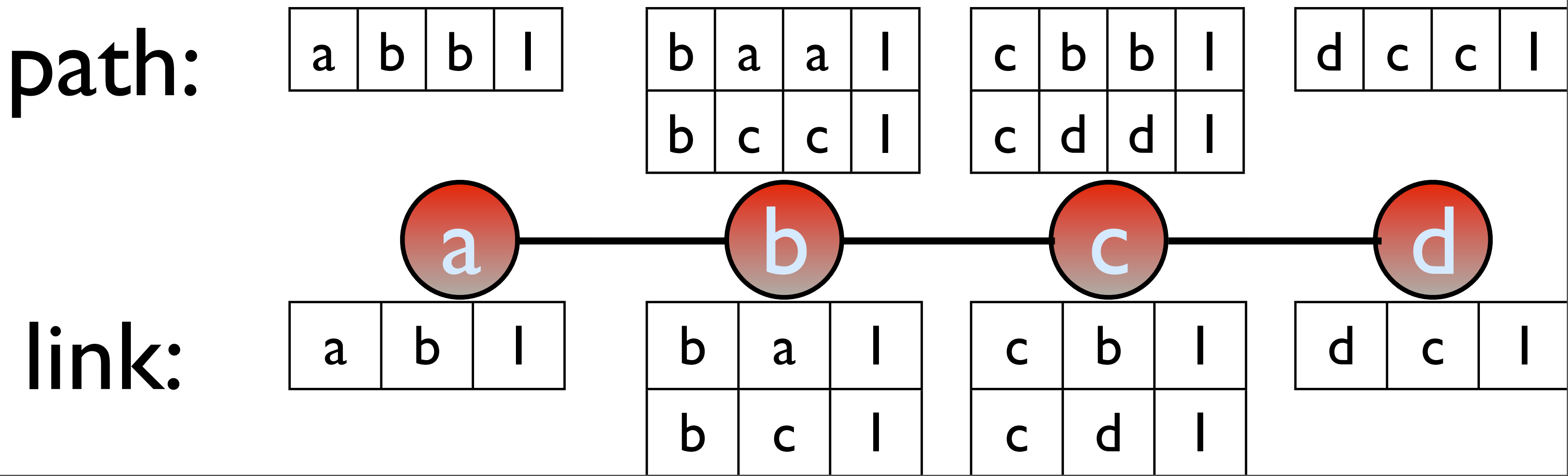
link at d:

d	c	1



LOCSPECS INDUCE COMMUNICATION

- link(@X,Y,C)
- path(@X,Y,Y,C) :- link(@X,Y,C)
- path(@X,Z,Y,C+D) :- link(@X,Y,C), path(@Y,Z,N,D)



path:

a	b	b	1

b	a	a	1
b	c	c	1

c	b	b	1
c	d	d	1

d	c	c	1

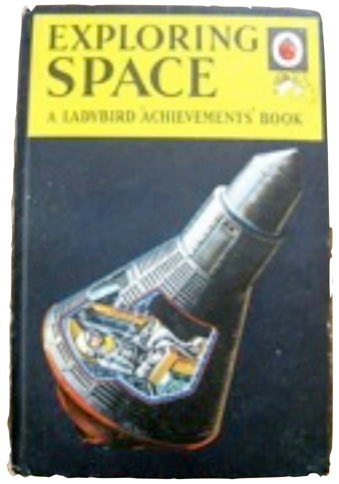
link:

a	b	1

b	a	1
b	c	1

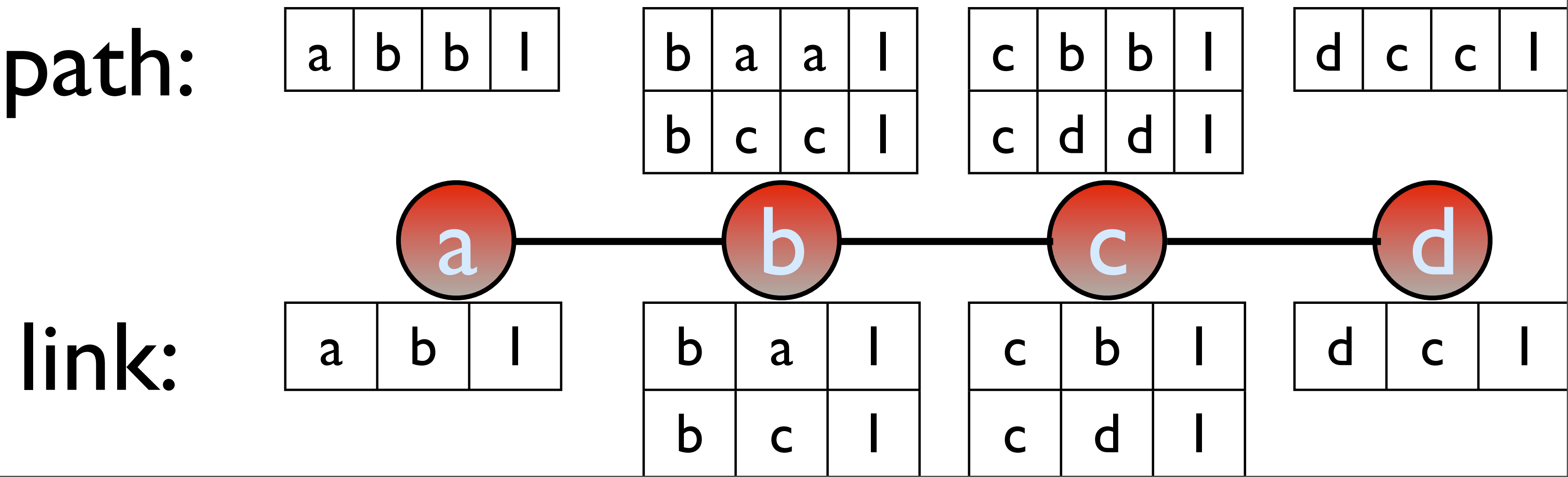
c	b	1
c	d	1

d	c	1



LOCSPECS INDUCE COMMUNICATION

- link(@X,Y,C)
- path(@X,Y,Y,C) :- link(@X,Y,C)
- path(@X,Z,Y,C+D) :- link(@X,Y,C), path(@Y,Z,N,D)



path:

a	b	b	l

b	a	a	l
b	c	c	l

c	b	b	l
c	d	d	l

d	c	c	l

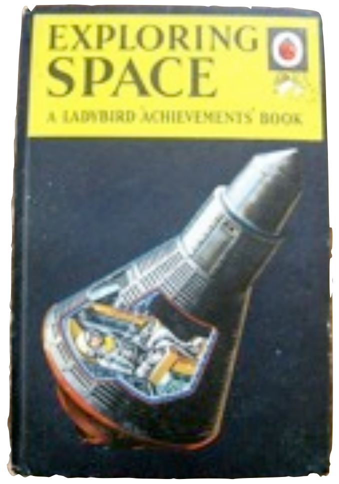
link:

a	b	l

b	a	l
b	c	l

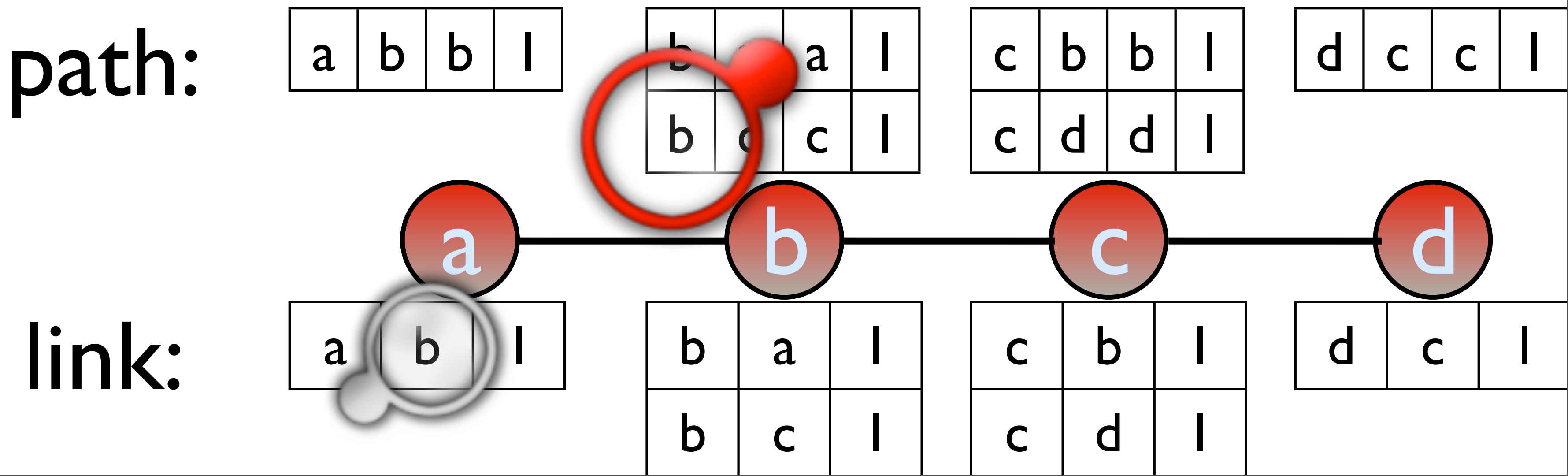
c	b	l
c	d	l

d	c	l



LOCSPECS INDUCE COMMUNICATION

- link(@X,Y,C)
- path(@X,Y,Y,C) :- link(@X,Y,C)
- path(@X,Z,Y,C+D) :- link(@X,Y,C), path(@Y,Z,N,D)



path:

a	b	b	1

b	[illegible]	a	1
b	c	c	1

c	b	b	1
c	d	d	1

d	c	c	1

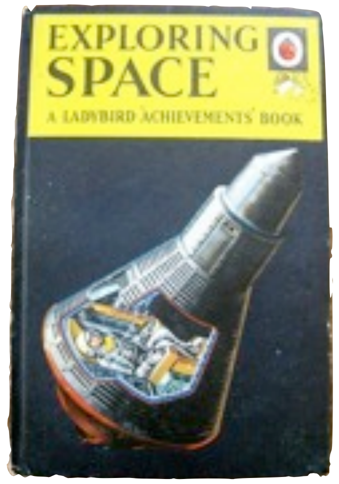
link:

a	b	1

b	a	1
b	c	1

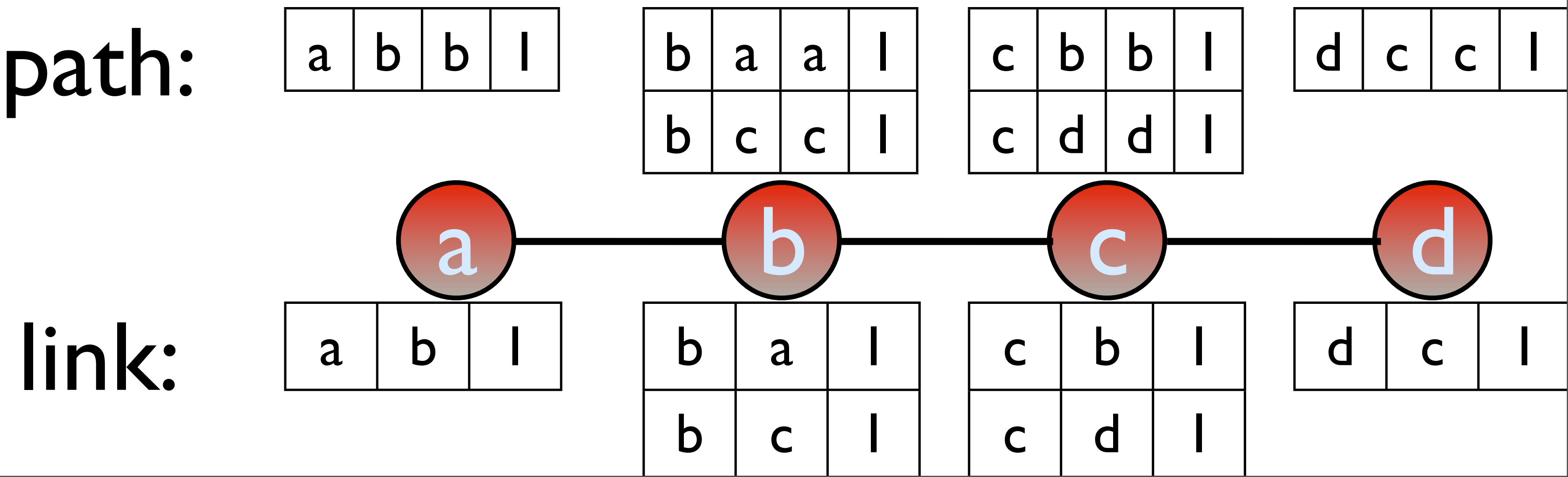
c	b	1
c	d	1

d	c	1



LOCSPECS INDUCE COMMUNICATION

- link(@X,Y,C)
- path(@X,Y,Y,C) :- link(@X,Y,C)
- path(@X,Z,Y,C+D) :- link(@X,Y,C), path(@Y,Z,N,D)



path:

a	b	b	l

b	a	a	l
b	c	c	l

c	b	b	l
c	d	d	l

d	c	c	l

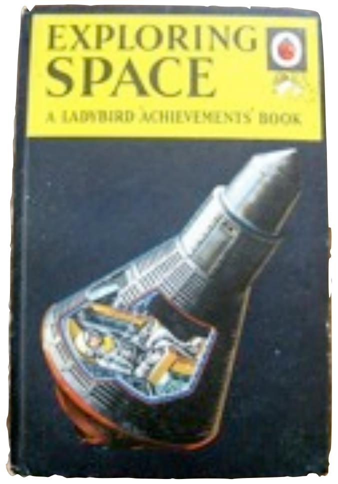
link:

a	b	l

b	a	l
b	c	l

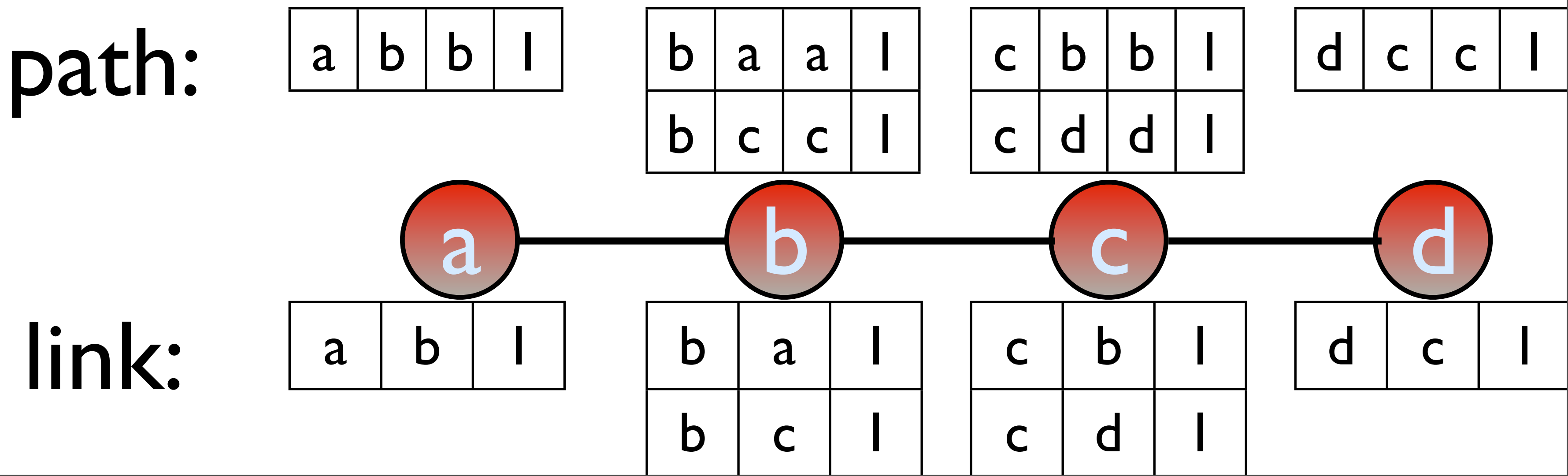
c	b	l
c	d	l

d	c	l



LOCSPECS INDUCE COMMUNICATION

- link(@X,Y,C)
- path(@X,Y,Y,C) :- link(@X,Y,C)
- path(@X,Z,Y,C+D) :- link(@X,Y,C), path(@Y,Z,N,D)



path:

a	b	b	1

b	a	a	1
b	c	c	1

c	b	b	1
c	d	d	1

d	c	c	1

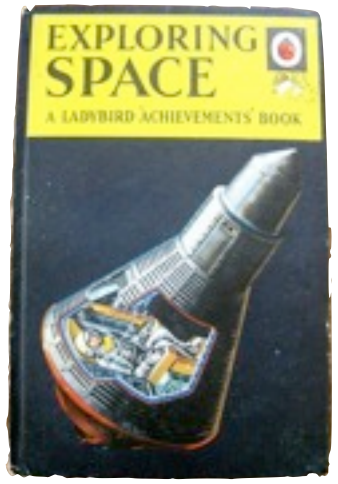
link:

a	b	1

b	a	1
b	c	1

c	b	1
c	d	1

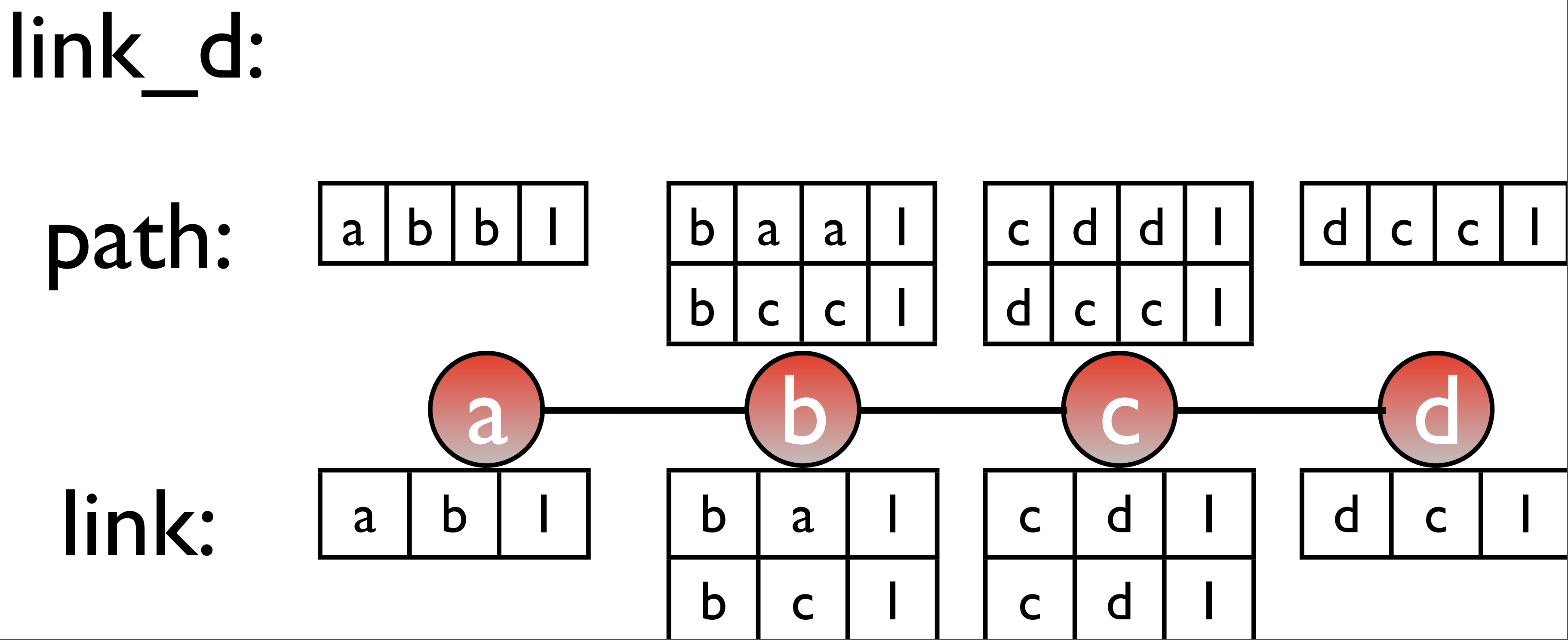
d	c	1

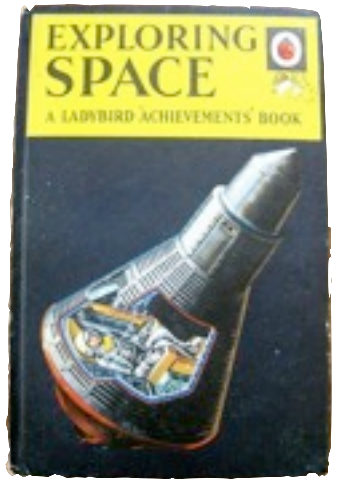


LOCSPECS INDUCE COMMUNICATION

Localization Rewrite

- link(@X,Y)
- path(@X,Y,Y,C) :- link(@X,Y,C)
  - link_d(X,@Y,C) :- link(@X,Y,C)
  - path(@X,Z,Y,C+D) :- link_d(X,@Y,C), path(@Y,Z,N,D)

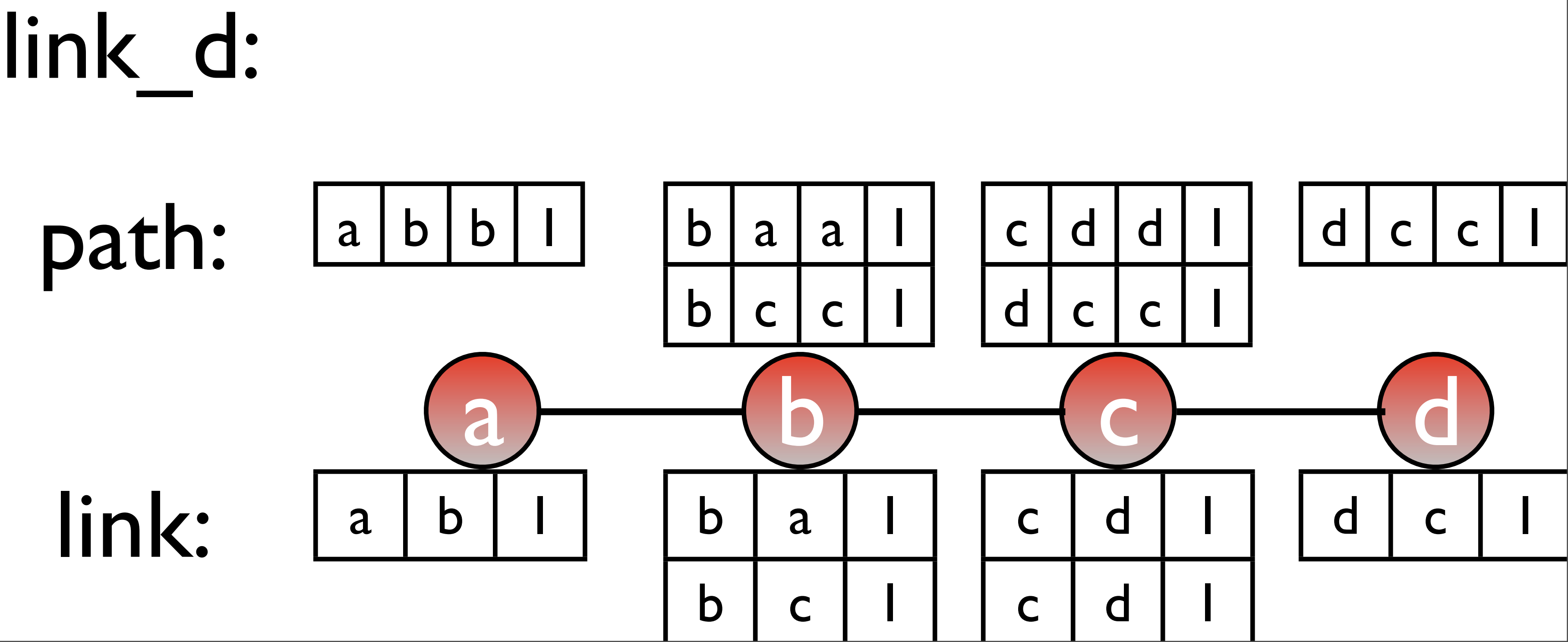


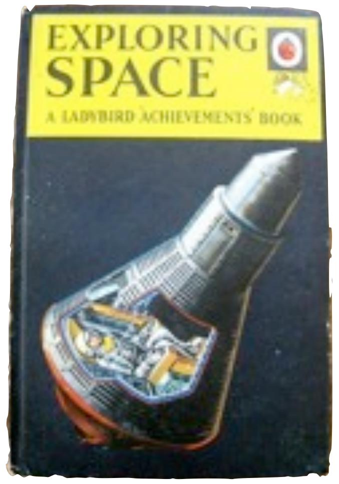


LOCSPECS INDUCE COMMUNICATION

- link(@X,Y)
- path(@X,Y,Y,C) :- link(@X,Y,C)
  - link_d(X,@Y,C) :- link(@X,Y,C)
  - path(@X,Z,Y,C+D) :- link_d(X,@Y,C), path(@Y,Z,N,D)

Localization Rewrite

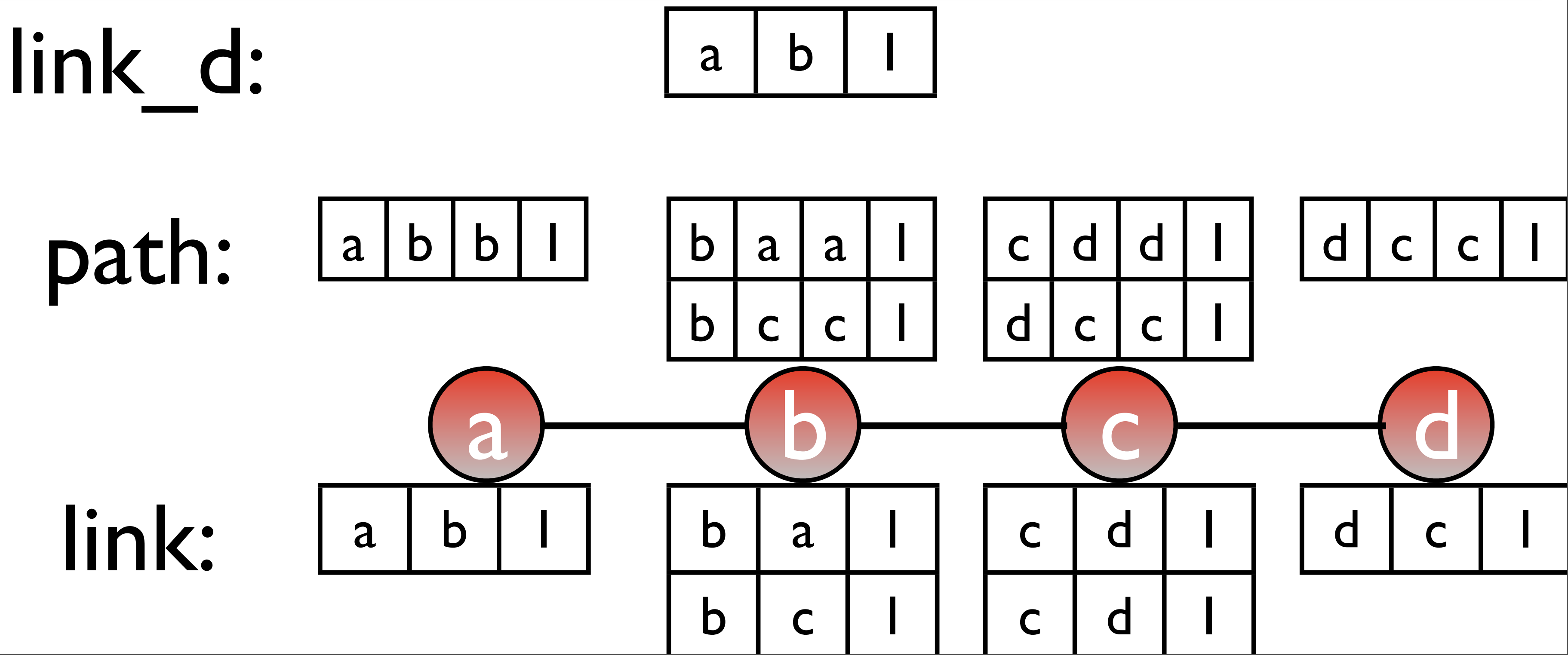


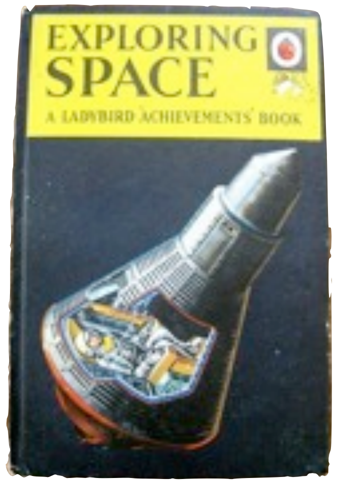


LOCSPECS INDUCE COMMUNICATION

Localization Rewrite

- link(@X,Y)
- path(@X,Y,Y,C) :- link(@X,Y,C)
  - link_d(X,@Y,C) :- link(@X,Y,C)
  - path(@X,Z,Y,C+D) :- link_d(X,@Y,C), path(@Y,Z,N,D)



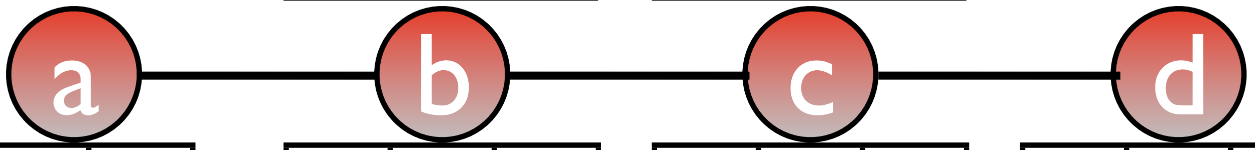


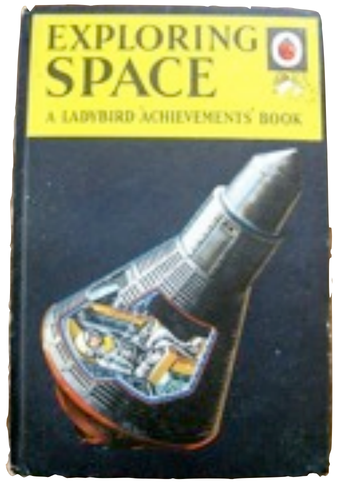
LOCSPECS INDUCE COMMUNICATION

Localization Rewrite

- link(@X,Y)
- path(@X,Y,Y,C) :- link(@X,Y,C)
  - link_d(X,@Y,C) :- link(@X,Y,C)
  - path(@X,Z,Y,C+D) :- link_d(X,@Y,C), path(@Y,Z,N,D)

	a	b	c	d
link_d:	b a 1	a b 1	b c 1	c d 1
		c b 1	d c 1	
path:	a b b 1	b a a 1	c d d 1	d c c 1
		b c c 1	d c c 1	
link:	a b 1	b a 1	c d 1	d c 1
		b c 1	c d 1	

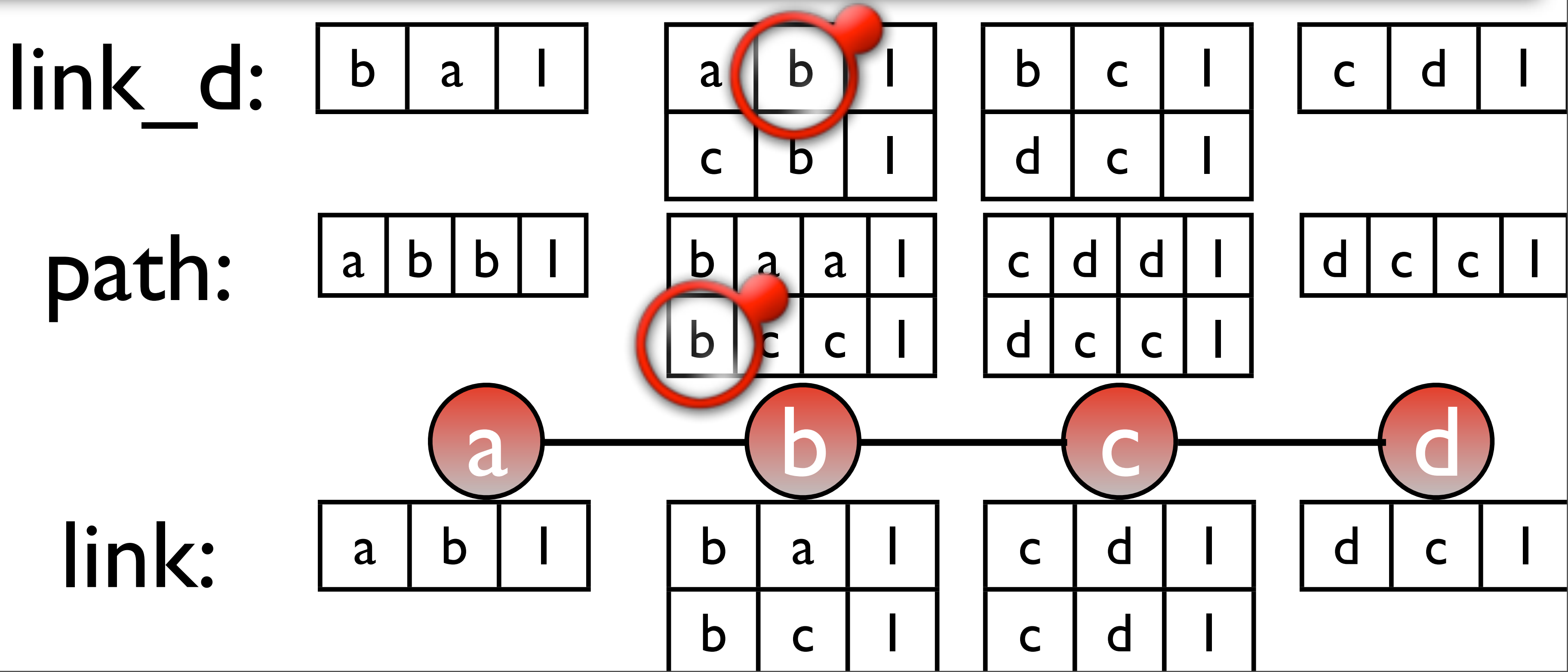


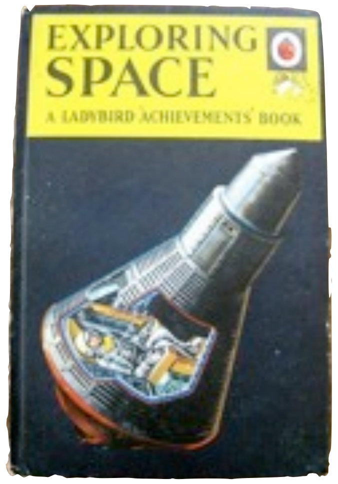


LOCSPECS INDUCE COMMUNICATION

Localization Rewrite

- link(@X,Y)
- path(@X,Y,Y,C) :- link(@X,Y,C)
  - link_d(X,@Y,C) :- link(@X,Y,C)
  - path(@X,Z,Y,C+D) :- link_d(X,@Y,C), path(@Y,Z,N,D)

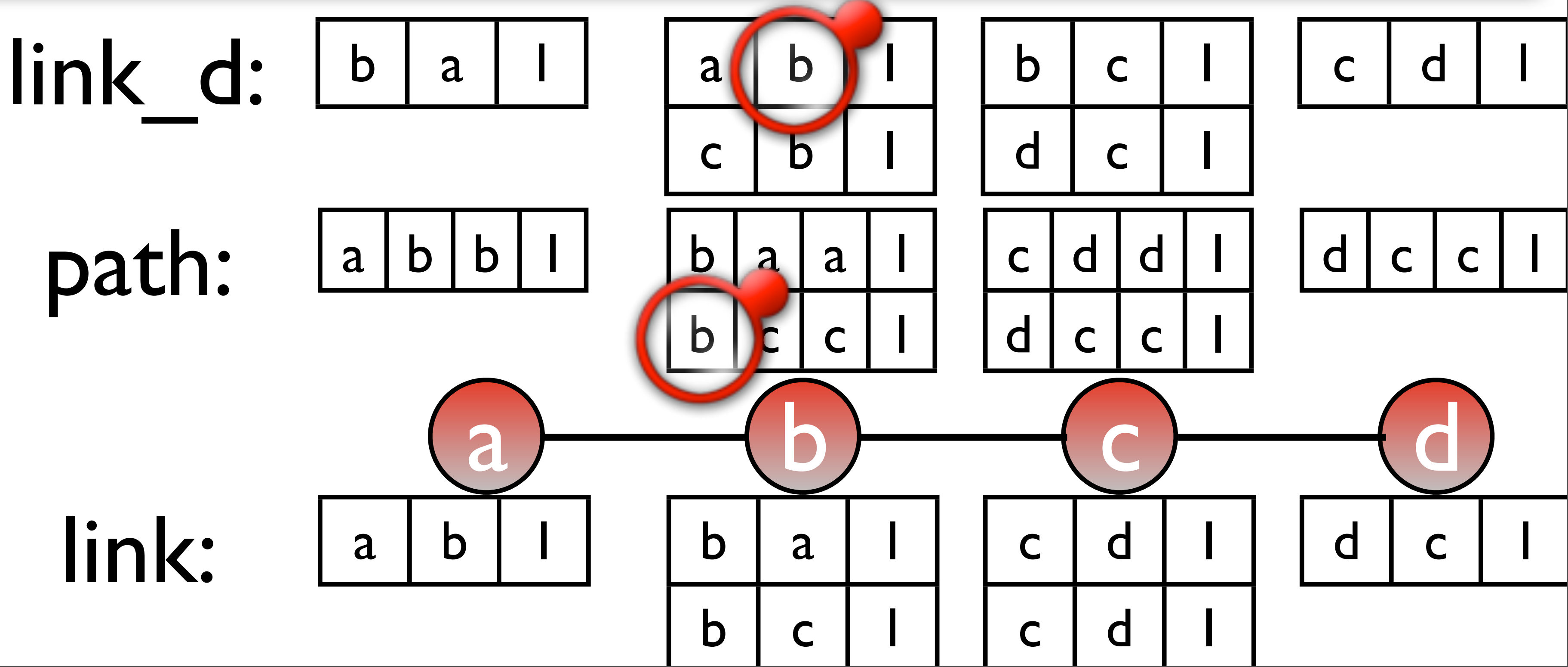




LOCSPECS INDUCE COMMUNICATION

Localization Rewrite

- link(@X,Y)
- path(@X,Y,Y,C) :- link(@X,Y,C)
  - link_d(X,@Y,C) :- link(@X,Y,C)
  - path(@X,Z,Y,C+D) :- link_d(X,@Y,C), path(@Y,Z,N,D)



link_d:

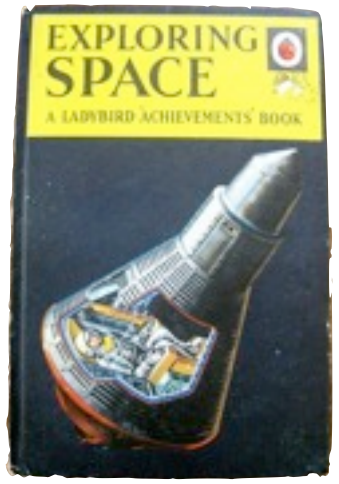
a	b	c	d
b a 1	a b 1	b c 1	c d 1
	c b 1	d c 1	

path:

a	b	c	d
a b b 1	b a a 1	c d d 1	d c c 1
	b c c 1	d c c 1	

link:

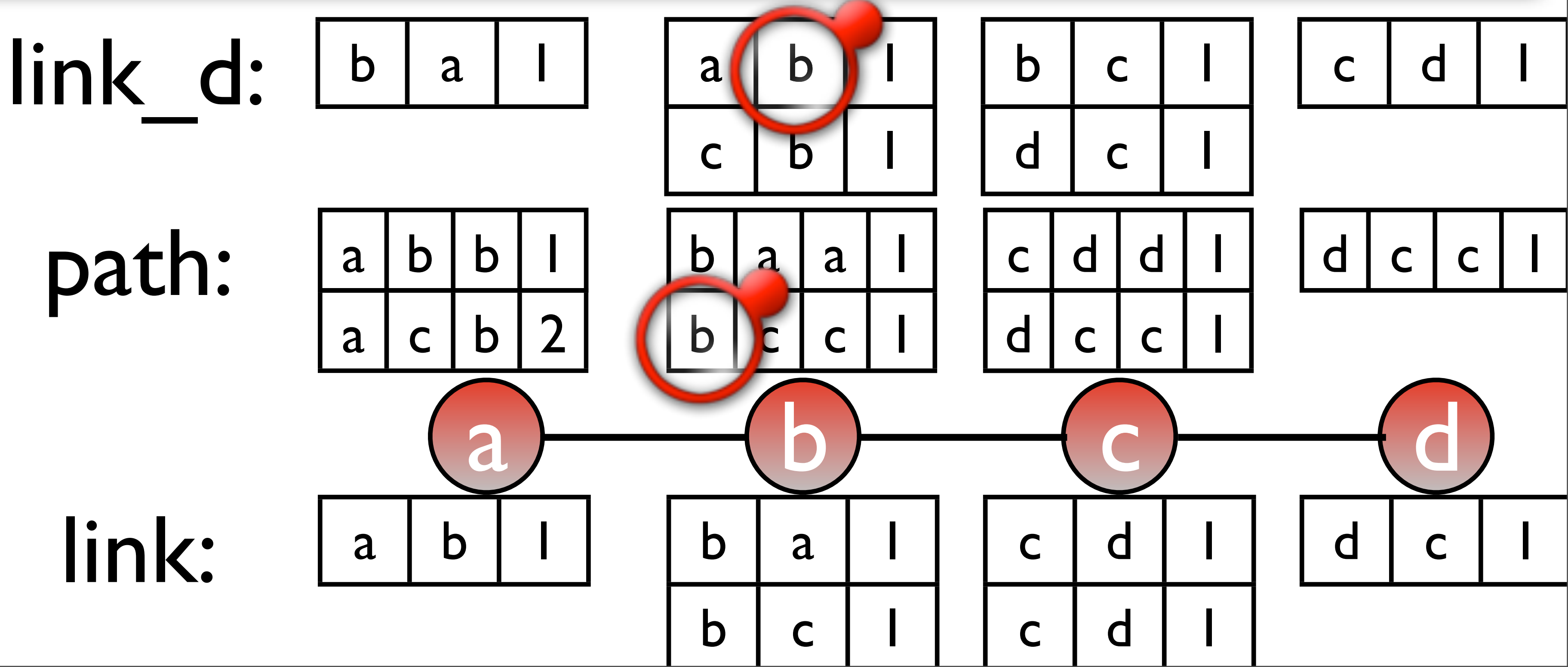
a	b	c	d
a b 1	b a 1	c d 1	d c 1
	b c 1	c d 1	



LOCSPECS INDUCE COMMUNICATION

Localization Rewrite

- link(@X,Y)
- path(@X,Y,Y,C) :- link(@X,Y,C)
  - link_d(X,@Y,C) :- link(@X,Y,C)
  - path(@X,Z,Y,C+D) :- link_d(X,@Y,C), path(@Y,Z,N,D)



link_d:

b	a	1

a	b	1
c	b	1

b	c	1
d	c	1

c	d	1

path:

a	b	b	1
a	c	b	2

b	a	a	1
b	c	c	1

c	d	d	1
d	c	c	1

d	c	c	1

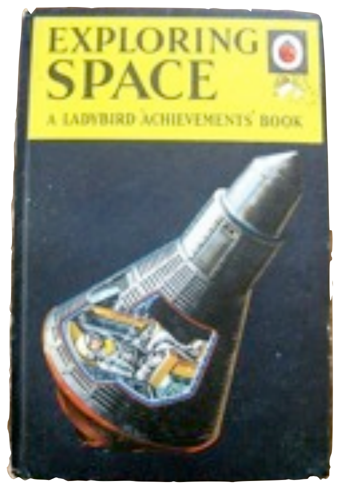
link:

a	b	1

b	a	1
b	c	1

c	d	1
c	d	1

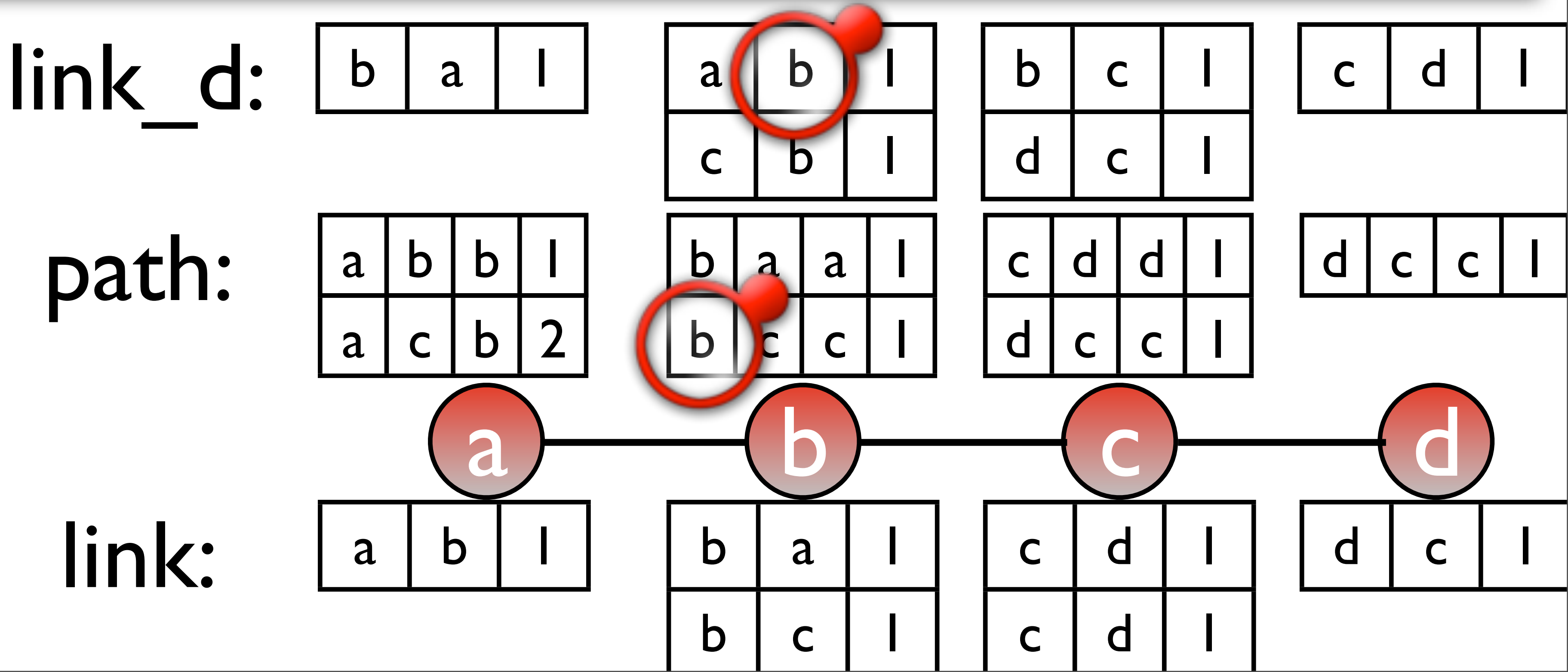
d	c	1



LOCSPECS INDUCE COMMUNICATION

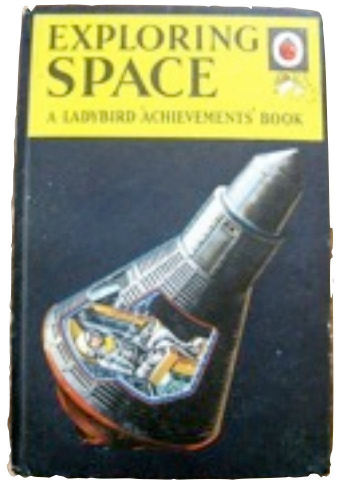
Localization Rewrite

- link(@X,Y)
- path(@X,Y,Y,C) :- link(@X,Y,C)
  - link_d(X,@Y,C) :- link(@X,Y,C)
  - path(@X,Z,Y,C+D) :- link_d(X,@Y,C), path(@Y,Z,N,D)



link_d:	b a 1	a b 1	b c 1	c d 1
		c b 1	d c 1	
path:	a b b 1	b a a 1	c d d 1	d c c 1
	a c b 2	b c c 1	d c c 1	
	a	b	c	d
link:	a b 1	b a 1	c d 1	d c 1
		b c 1	c d 1	

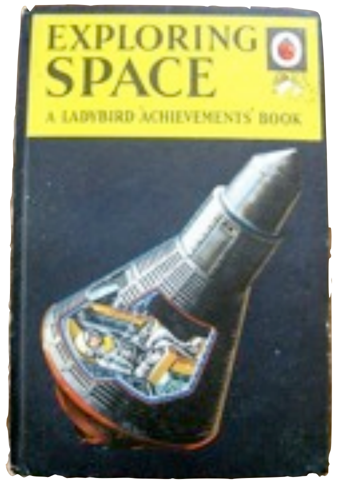
THIS IS DISTANCE VECTOR



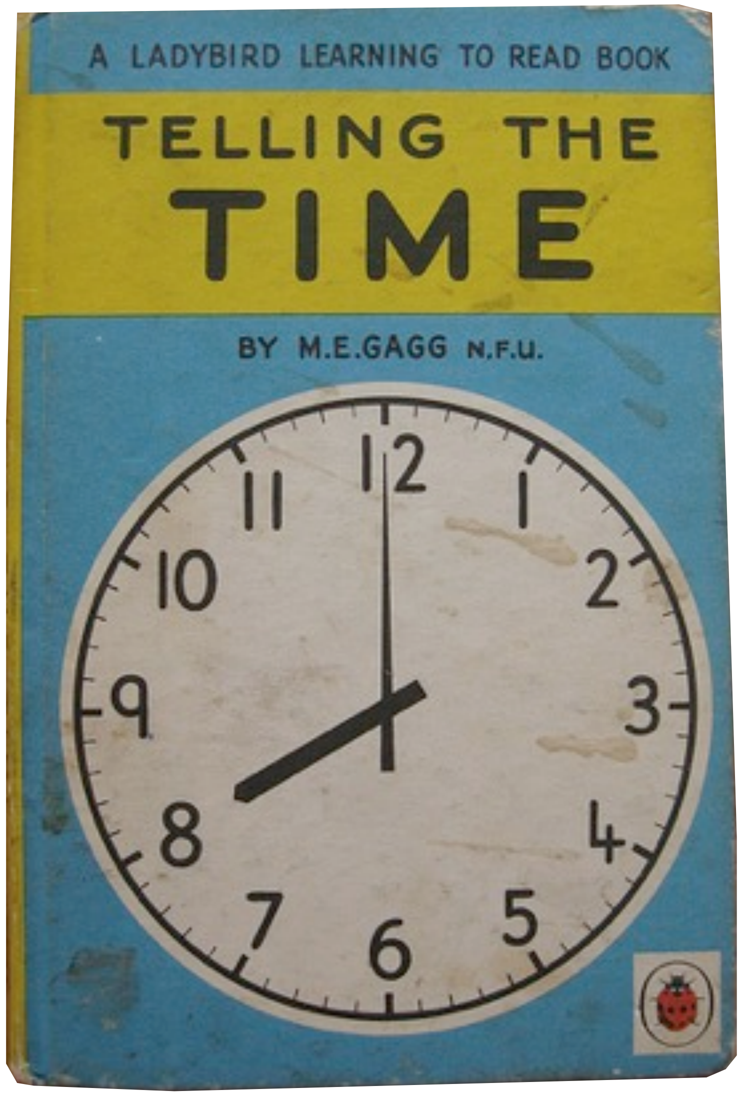
THE MYTH OF THE GLOBAL DATABASE

- the problem with space?
- distributed join consistency

  -
    ```
    path(@X,Z,Y,C+D) :-
            link(@X,Y,C), path(@Y,Z,N,D)
    ```
  - needs coordination, e.g. 2PC?
  - “localized” *async* rules more “honest”
- perils of a false abstraction



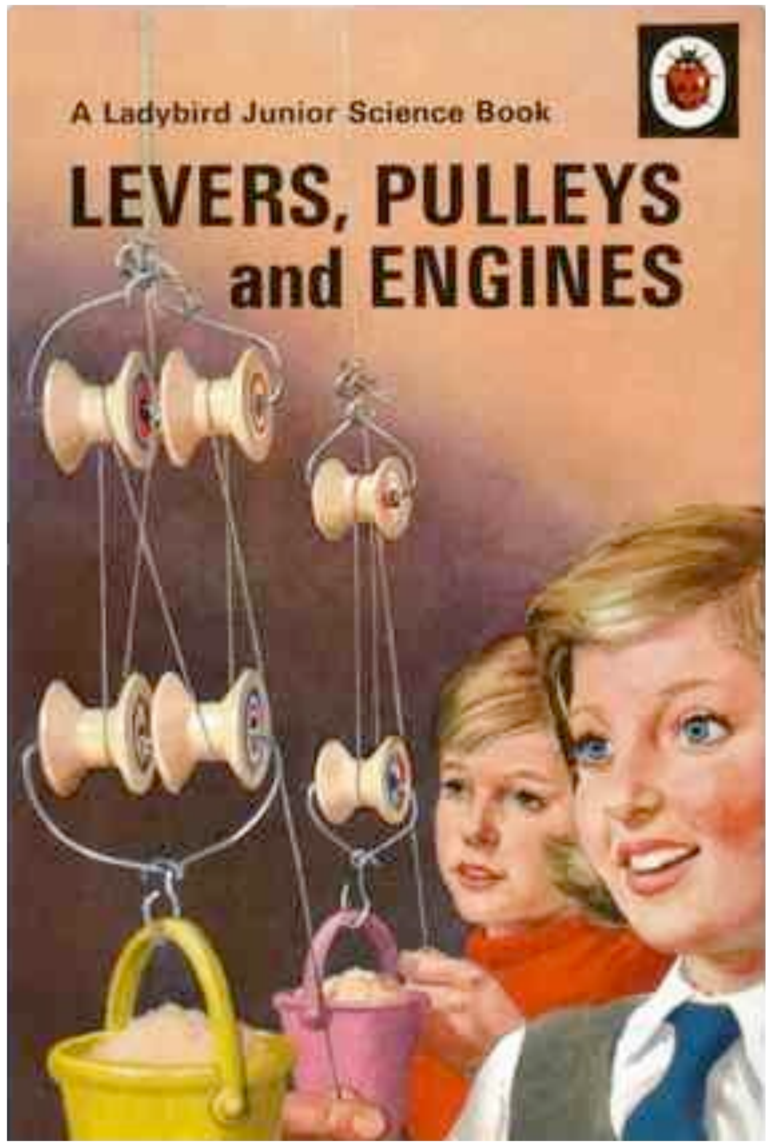
THE MYTH OF THE GLOBAL DATABASE



- the problem with space?
- distributed join consistency
  - `path(@X,Z,Y,C+D) :-`
    `link(@X,Y,C), path(@Y,Z,N,D)`
  - needs coordination, e.g. 2PC?
  - “localized” *async* rules more “honest”
- perils of a false abstraction

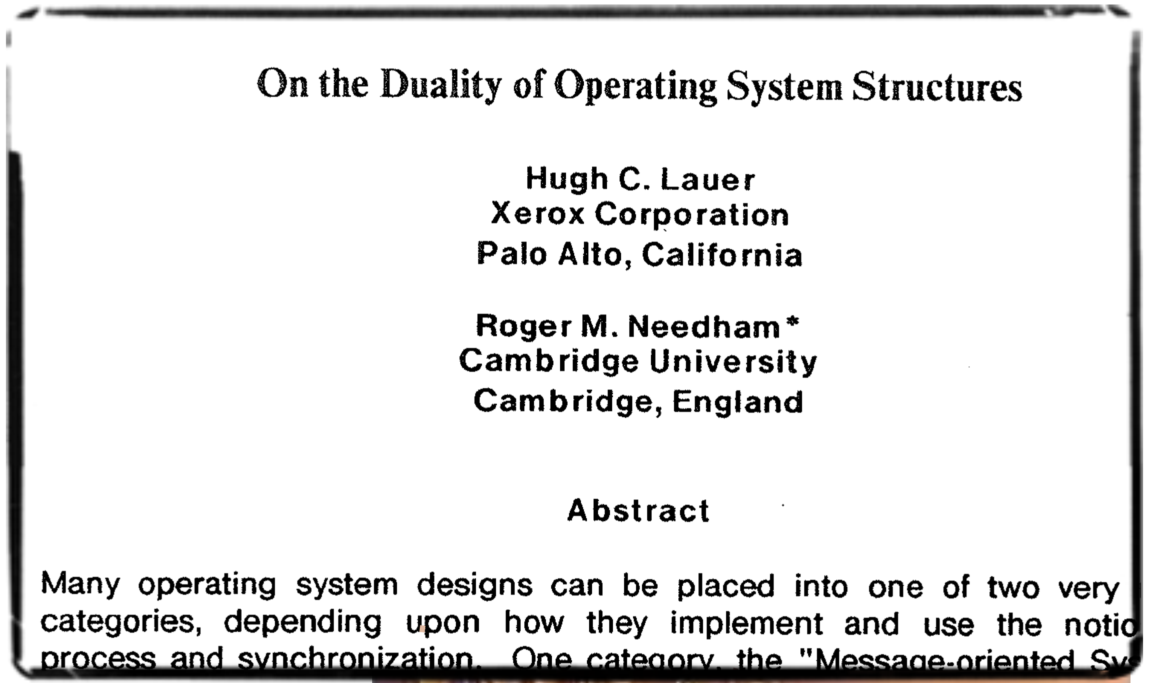
3. ENGINE ARCHITECTURE

- engine architecture
  - threads? events?
  - join!
    - session state w/events
- modeling ephemera
  - events, timeouts, soft-state
- in the paper



3. ENGINE ARCHITECTURE

- engine architecture
  - threads? events?
  - join!
    - session state w/events
- modeling ephemera
  - events, timeouts, soft-state
- in the paper



On the Duality of Operating System Structures

Hugh C. Lauer
Xerox Corporation
Palo Alto, California

Roger M. Needham*
Cambridge University
Cambridge, England

Abstract

Many operating system designs can be placed into one of two very
categories, depending upon how they implement and use the notio
process and synchronization. One category, the "Message-oriented Sy

TODAY

- two unfinished stories
- a dedalus primer
- experience
- implications and conjectur

WHAT TO LOOK FOR IN SPRING
A Ladybird Book
Chicken Licken
A LADYBIRD LEARNING TO READ BOOK
GOING TO SCHOOL
A Ladybird 'Easy-Reading' Book
'People at Work'
THE BUILDER

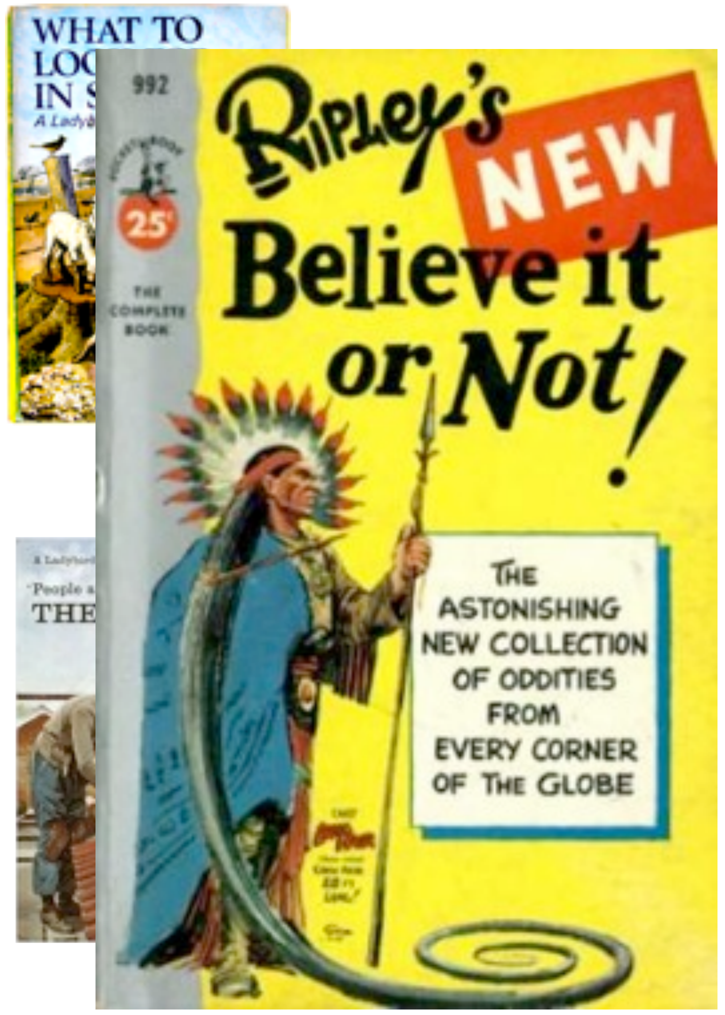
TODAY

WHAT TO LOOK FOR IN SPRING
A Ladybird Book
Chicken Licken
A LADYBIRD LEARNING TO READ BOOK
GOING TO SCHOOL
A Ladybird 'Easy Reading' Book
'People at Work'
THE BUILDER

- two unfinished stories
- a dedalus primer
- experience
- implications and conjecture

TODAY

- two unfinished stories
- a dedalus primer
- experience
- implications and conjecture



IMPLICATIONS AND CONJECTURES

- the CALM conjecture
- the CRON conjecture
- Coordination Complexity
- the Fateful Time conjecture

IMPLICATIONS AND CONJECTURES

- the CALM conjecture
- the CRON conjecture
- Coordination Complexity
- the Fateful Time conjecture

IMPLICATIONS AND CONJECTURES

- the CALM conjecture
- the CRON conjecture
- Coordination Complexity
- the Fateful Time conjecture

IMPLICATIONS AND CONJECTURES

- the CALM conjecture
- the CRON conjecture
- Coordination Complexity
- the Fateful Time conjecture

IMPLICATIONS AND CONJECTURES

- the CALM conjecture
- the CRON conjecture
- Coordination Complexity
- the Fateful Time conjecture

BUT FIRST, THE ENDGAME!

COUNTING WAITS. WAITING COUNTS.

- distributed aggregation?
  - esp. with recursion?!
  - requires coordination (consider "count-to-zero")
  - *counting requires waiting*
- coordination protocols?
  - all entail "voting"
    - 2PC, Paxos, BFT
  - *waiting requires counting*

IMPLICATIONS AND CONJECTURES

- the CALM conjecture
- the CRON conjecture
- Coordination Complexity
- the Fateful Time conjecture

THE FUSS ABOUT EVENTUAL CONSISTENCY

- cloud folks, etc. don't like transactions
    - they involve waiting (counting)
- *eventually consistent* storage
    - no waiting
    - loose Consistency, but Availability during network Partitions
        - things work out when partitions "eventually" reconnect
        - (see Brewer's CAP Theorem)
- spawned the noSQL movement

MONOTONIC? EVENTUALLY CONSISTENT!

- my definition of *eventual consistency*
  - given: distributed system, finite trace of messages
  - *eventual consistency* if the final state of the system is independent of message ordering
    - and ensuring so does not require coordination!
- more than the usual
  - typical focus is on replicas and versions of state
  - we are interested in consistency of a *whole program*
    - replication is a special case: `p_rep(X, @r)@async :- p(X, @a).`

EXAMPLE: SHOPPING CART

- shopping: a growing to-do list
  - e.g., *"add n units of item X to cart"*
  - e.g., *"delete m units of item Y from cart"*
  - easily supported by eventually-consistent infrastructure
- check-out: aggregation
  - compute totals
  - validate stock-on-hand, confirm with user (and move on to billing logic)
  - typically supported by richer infrastructure. *not e.c.*
- a well-known pattern
  - "general ledger", "escrow transactions", etc.

THE CALM CONJECTURE

CONJECTURE 1. Consistency And Logical Monotonicity (CALM). *A program has an eventually consistent, coordination-free evaluation strategy iff it is expressible in (monotonic) Datalog*.

- monotonic ⇒ EC
  - via pipelined semi-naive evaluation (PSN)
    - positive derivations can "accumulate"
- !monotonic ⇒ !EC
  - distributed negation/aggregation
    - the end of the game!

THE CALM CONJECTURE

CONJECTURE 1. Consistency And Logical Monotonicity (CALM).
A program has an eventually consistent, coordination-free evaluation strategy iff it is expressible in (monotonic) Datalog.

- monotonic ⇒ EC
  - via pipelined semi-naive evaluation (PSN)
    - positive derivations can "accumulate"
- !monotonic ⇒ !EC
  - distributed negation/aggregation
    - the end of the game!

CALM IMPLICATIONS

- NoSQL = Datalog!
  - ditto lock-free data structures
- whole-program tests over e.c. storage
- automatic relaxation of consistent programs
- synthesis of coordination/compensation

IMPLICATIONS AND CONJECTURES

- the CALM conjecture
- the CRON conjecture
- Coordination Complexity
- the Fateful Time conjecture

CAUSALITY (WHAT ABOUT PODC?)

- Lamport and his Clock Condition
  - given a partial order → (happens-before)
  - and a per-node clock *C*
  - for any events *a*, *b*
    if $a \rightarrow b$ then $C(a) < C(b)$
- Respect Time & the (partial) Order!

TIME IS FOR (NON-MONOTONIC) SUCKERS!

TIME IS FOR (NON-MONOTONIC) SUCKERS!

Time flies like an arrow.

TIME IS FOR (NON-MONOTONIC) SUCKERS!

Time flies like an arrow.

Fruit flies like a banana.
— Groucho Marx

TIME TRAVEL

- we *can* send things back in time!
  - nobody said we couldn't!
  - theoretician@async(X) :- pods(X).
- but ... temporal paradoxes?
  - e.g. the grandfather paradox

THE GRANDFATHER PARADOX

THE GRANDFATHER PARADOX

```
parent(X, Z) :- has_baby(X,Y,Z).
```

THE GRANDFATHER PARADOX

```
parent(X, Z) :- has_baby(X,Y,Z).
parent(Y, Z) :- has_baby(X,Y,Z).
```

THE GRANDFATHER PARADOX

```
parent(X, Z) :- has_baby(X,Y,Z).
parent(Y, Z) :- has_baby(X,Y,Z).
parent@next(X,Y) :- parent(X,Y),
                    !del_p(X,Y).
```

THE GRANDFATHER PARADOX

```
parent(X, Z) :- has_baby(X,Y,Z).
parent(Y, Z) :- has_baby(X,Y,Z).
parent@next(X,Y) :- parent(X,Y),
                    !del_p(X,Y).
anc(X, Y) :- parent(X, Y).
```

THE GRANDFATHER PARADOX

```
parent(X, Z) :- has_baby(X,Y,Z).
parent(Y, Z) :- has_baby(X,Y,Z).
parent@next(X,Y) :- parent(X,Y),
                    !del_p(X,Y).
anc(X, Y) :- parent(X, Y).
anc(X, Y) :- parent(X,Z),
             anc(Z,Y).
```

THE GRANDFATHER PARADOX

```
parent(X, Z) :- has_baby(X,Y,Z).
parent(Y, Z) :- has_baby(X,Y,Z).
parent@next(X,Y) :- parent(X,Y),
                    !del_p(X,Y).
anc(X, Y) :- parent(X, Y).
anc(X, Y) :- parent(X,Z),
             anc(Z,Y).
```

```
kill@async(X,Y) :- mistreat(Y,X).
```

THE GRANDFATHER PARADOX

```
parent(X, Z) :- has_baby(X,Y,Z).
parent(Y, Z) :- has_baby(X,Y,Z).
parent@next(X,Y) :- parent(X,Y),
                    !del_p(X,Y).
anc(X, Y) :- parent(X, Y).
anc(X, Y) :- parent(X,Z),
             anc(Z,Y).

kill@async(X,Y) :- mistreat(Y,X).
del_p(Y, Z) :- kill(X, Y).
```

THE GRANDFATHER PARADOX

```
parent(X, Z) :- has_baby(X,Y,Z).
parent(Y, Z) :- has_baby(X,Y,Z).
parent@next(X,Y) :- parent(X,Y),
                    !del_p(X,Y).
anc(X, Y) :- parent(X, Y).
anc(X, Y) :- parent(X,Z),
             anc(Z,Y).
```

```
kill@async(X,Y) :- mistreat(Y,X).
del_p(Y, Z) :- kill(X, Y).
```

Murder is Non-Monotonic.

THE CRON CONJECTURE

CONJECTURE 2. Causality Required Only for Non-Monotonicity. (CRON). *Program semantics require causal message ordering if and only if the messages participate in non-monotonic derivations*.

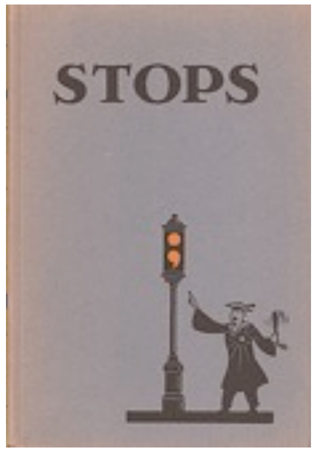
- intuition: local stratification

  assume a cycle through non-monotonic predicates across timesteps.

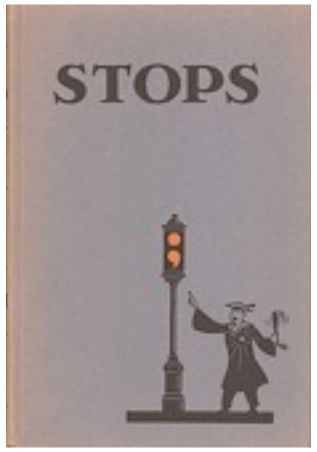
  looping derivations prevented if timestamps are monotonic

IMPLICATIONS AND CONJECTURES

- the CALM conjecture
- the CRON conjecture
- Coordination Complexity
- the Fateful Time conjecture



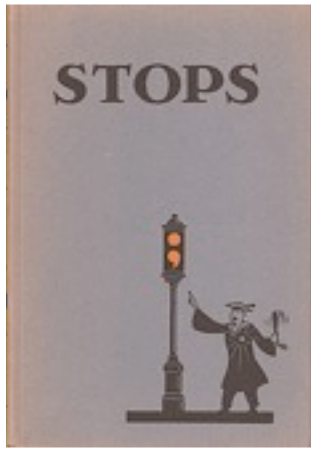
UNSTRATIFIABLE? SPEND SOME TIME.



UNSTRATIFIABLE? SPEND SOME TIME.

- this is a problem:

```
p(X) :- !p(X), q(X).
```



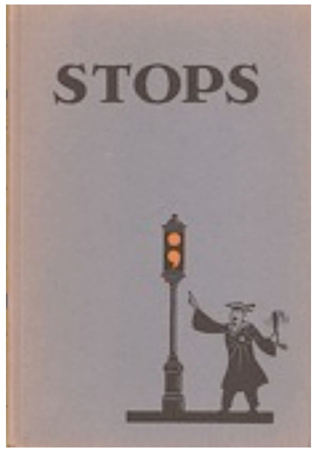
UNSTRATIFIABLE? SPEND SOME TIME.

- this is a problem:

```
p(X) :- !p(X), q(X).
```

- this is a solution:

```
q(X)@next :- q(X).
p(X)@next :- !p(X), q(X).
```



UNSTRATIFIABLE? SPEND SOME TIME.

- this is a problem:

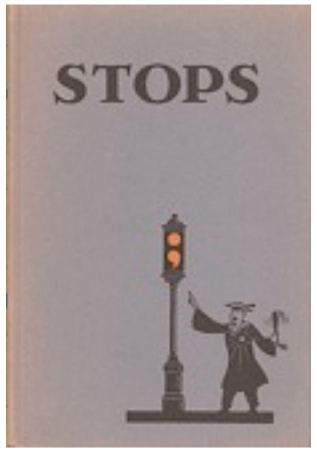
```
p(X) :- !p(X), q(X).
```

- this is a solution:

```
q(X)@next :- q(X).
p(X)@next :- !p(X), q(X).
```

- this is just dumb:

```
anc(X, Y)@next :- parent(X, Y).
anc(X, Y)@next :- parent(X,Z),
                          anc(Z,Y).
```



UNSTRATIFIABLE? SPEND SOME TIME.

- this is a problem:

```
p(X) :- !p(X), q(X).
```

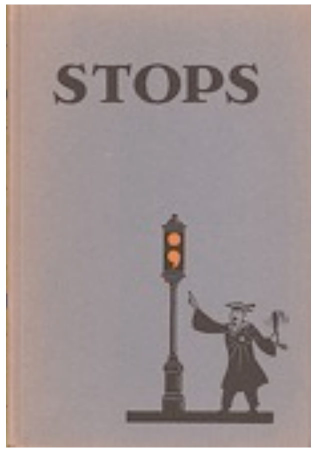
- this is just dumb:

```
anc(X, Y)@next :- parent(X, Y).
anc(X, Y)@next :- parent(X,Z),
                  anc(Z,Y).
```

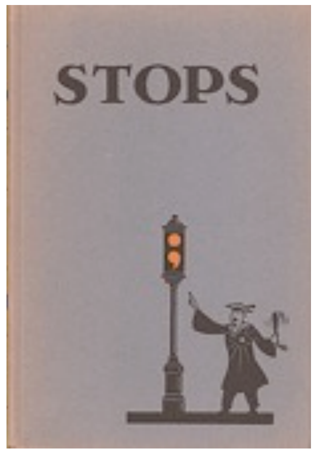
- this is a solution:

```
q(X)@next :- q(X).
p(X)@next :- !p(X), q(X).
```

- how does Dedalus time relate to complexity?

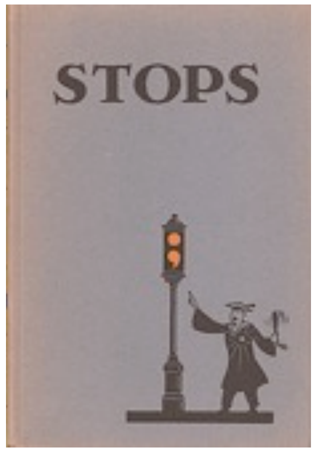


PRACTICAL (?? !!) SIDENOTE



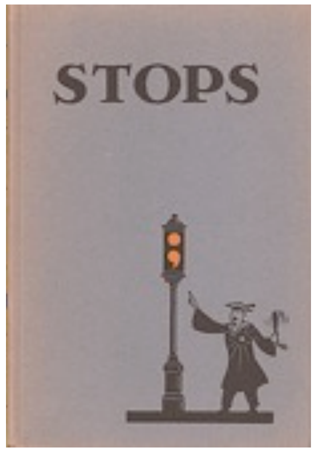
PRACTICAL (?? !!) SIDENOTE

- Challenge: win a benchmark with free computers.



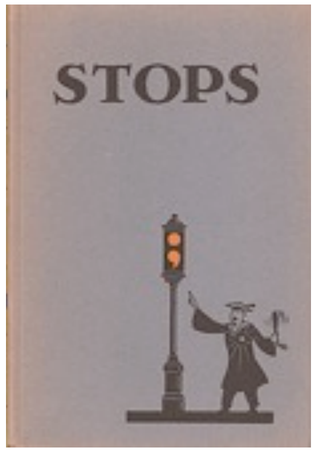
PRACTICAL (?? !!) SIDENOTE

- Challenge: win a benchmark with free computers.
- Yahoo Petasort:



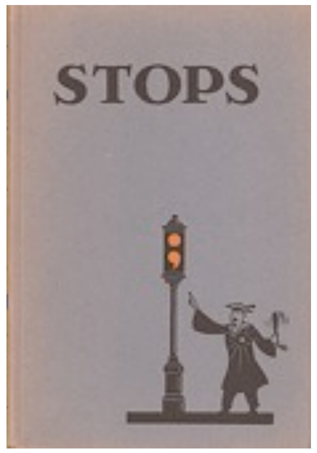
PRACTICAL (?? !!) SIDENOTE

- Challenge: win a benchmark with free computers.
- Yahoo Petasort:
  - 3,800 8-core, 4-disk machines



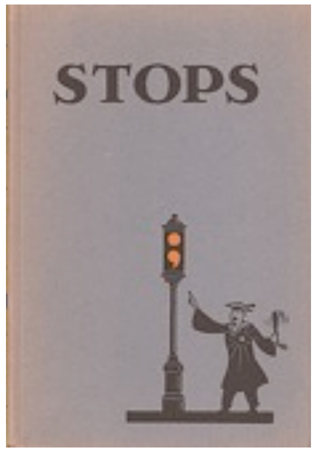
PRACTICAL (?? !!) SIDENOTE

- Challenge: win a benchmark with free computers.
- Yahoo Petasort:
  - 3,800 8-core, 4-disk machines
  - i.e. each core sorted 32 MB (1/512 of RAM!)



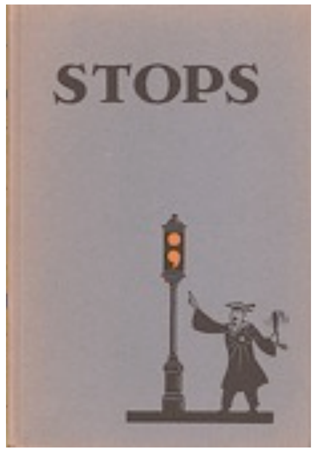
PRACTICAL (?? !!) SIDENOTE

- Challenge: win a benchmark with free computers.
- Yahoo Petasort:
  - 3,800 8-core, 4-disk machines
  - i.e. each core sorted 32 MB (1/512 of RAM!)
  - 3799/3800 of a Petabyte streamed across the network



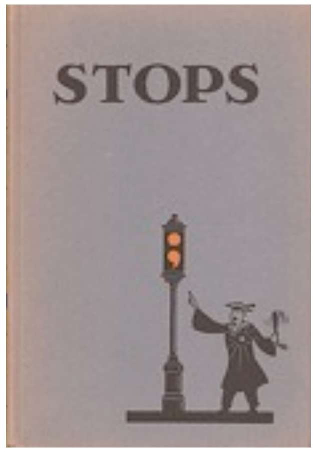
PRACTICAL (?? !!) SIDENOTE

- Challenge: win a benchmark with free computers.
- Yahoo Petasort:
  - 3,800 8-core, 4-disk machines
  - i.e. each core sorted 32 MB (1/512 of RAM!)
  - 3799/3800 of a Petabyte streamed across the network
  - 16.25 hours



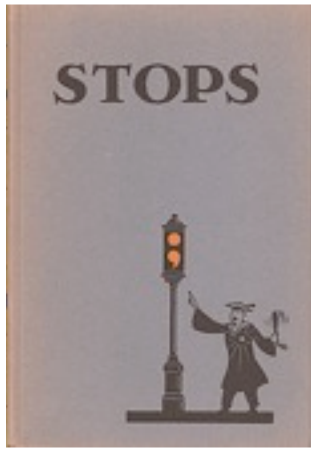
PRACTICAL (?? !!) SIDENOTE

- Challenge: win a benchmark with free computers.
- Yahoo Petasort:
  - 3,800 8-core, 4-disk machines
  - i.e. each core sorted 32 MB (1/512 of RAM!)
  - 3799/3800 of a Petabyte streamed across the network
  - 16.25 hours
- rental cost in the cloud



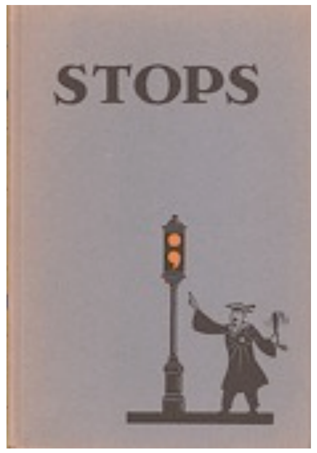
PRACTICAL (?? !!) SIDENOTE

- Challenge: win a benchmark with free computers.
- Yahoo Petasort:
  - 3,800 8-core, 4-disk machines
  - i.e. each core sorted 32 MB (1/512 of RAM!)
  - 3799/3800 of a Petabyte streamed across the network
  - 16.25 hours
- rental cost in the cloud
  - Amazon EC2 “High-CPU extra large” @ $0.84/hour



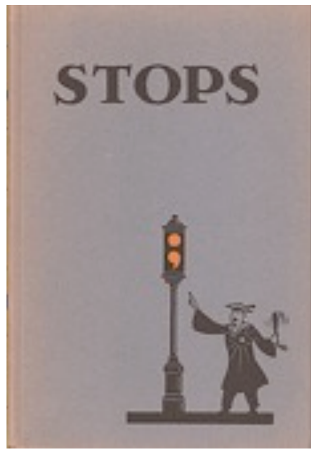
PRACTICAL (?? !!) SIDENOTE

- Challenge: win a benchmark with free computers.
- Yahoo Petasort:
  - 3,800 8-core, 4-disk machines
  - i.e. each core sorted 32 MB (1/512 of RAM!)
  - 3799/3800 of a Petabyte streamed across the network
  - 16.25 hours
- rental cost in the cloud
  - Amazon EC2 “High-CPU extra large” @ $0.84/hour
    - 3800 * 0.84 * 16.25 = $51,870



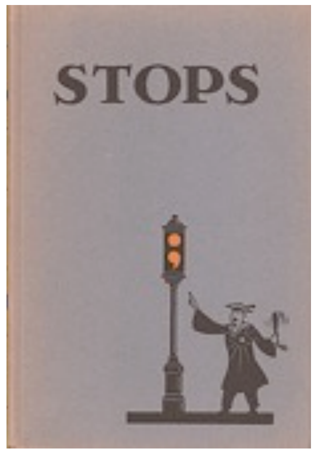
PRACTICAL (?? !!) SIDENOTE

- Challenge: win a benchmark with free computers.
- Yahoo Petasort:
  - 3,800 8-core, 4-disk machines
  - i.e. each core sorted 32 MB (1/512 of RAM!)
  - 3799/3800 of a Petabyte streamed across the network
  - 16.25 hours
- rental cost in the cloud
  - Amazon EC2 “High-CPU extra large” @ $0.84/hour
    - 3800 * 0.84 * 16.25 = $51,870
  - not a perfect clone, but rather impressive



PRACTICAL (?? !!) SIDENOTE

- Challenge: win a benchmark with free computers.
- Yahoo Petasort:
  - 3,800 8-core, 4-disk machines
  - i.e. each core sorted 32 MB (1/512 of RAM!)
  - 3799/3800 of a Petabyte streamed across the network
  - 16.25 hours
- rental cost in the cloud
  - Amazon EC2 “High-CPU extra large” @ $0.84/hour
    - 3800 * 0.84 * 16.25 = $51,870
  - not a perfect clone, but rather impressive
- pretty close to free

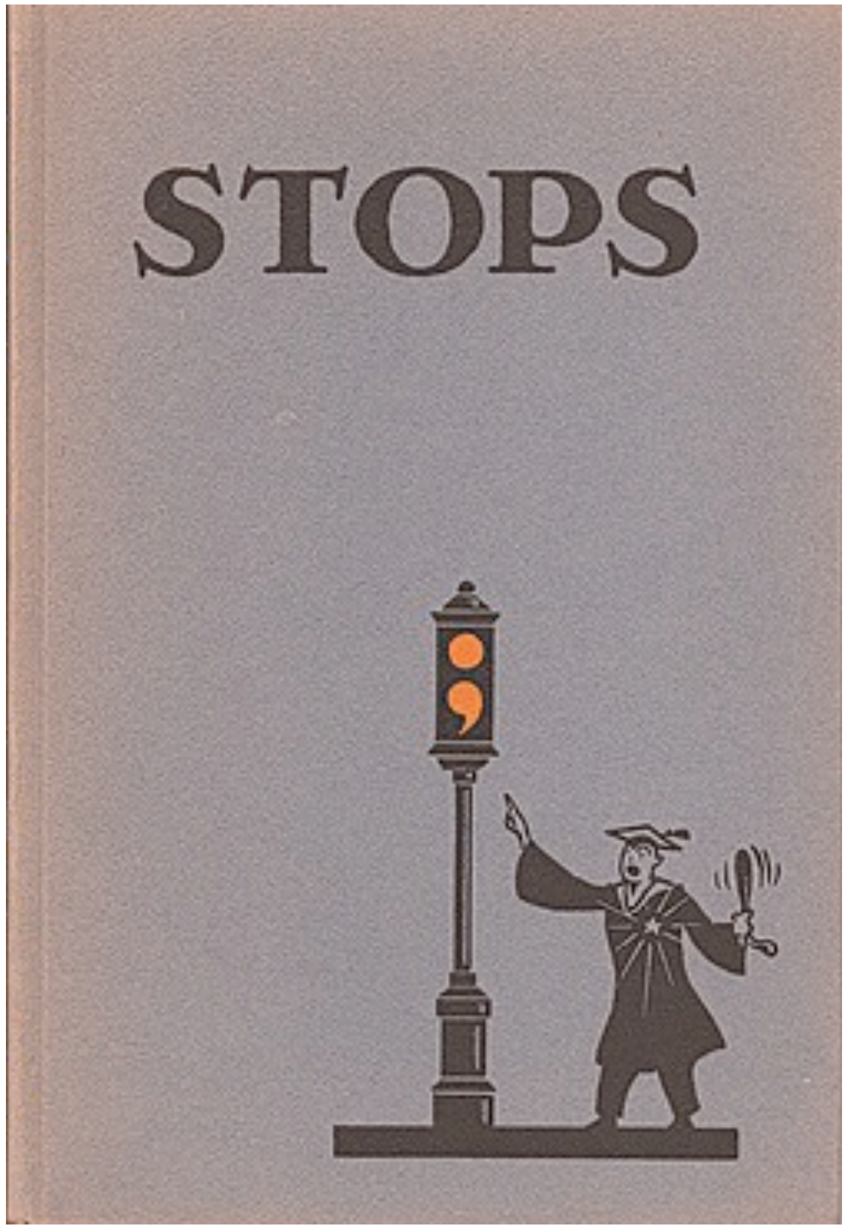


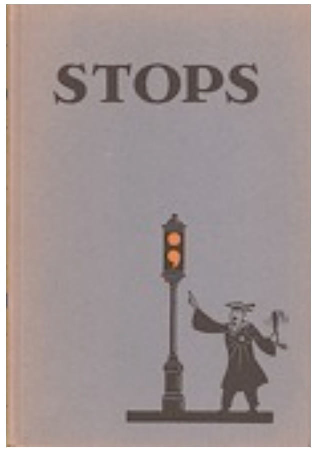
PRACTICAL (?? !!) SIDENOTE

- Challenge: win a benchmark with free computers.
- Yahoo Petasort:
  - 3,800 8-core, 4-disk machines
  - i.e. each core sorted 32 MB (1/512 of RAM!)
  - 3799/3800 of a Petabyte streamed across the network
  - 16.25 hours
- rental cost in the cloud
  - Amazon EC2 “High-CPU extra large” @ $0.84/hour
    - 3800 * 0.84 * 16.25 = $51,870
  - not a perfect clone, but rather impressive
- pretty close to free
  - so where’s the complexity?

COORDINATION COMPLEXITY

- coordination the main cost
  - failure/delay probabilities
  - compounded by queuing effects
- coordination complexity:
  - # of sequential coordination steps required for evaluation
- CALM: coordination manifest in logic!
  - coordination at stratum boundaries





DEDALUS TIME AND COORD COMPLEXITY

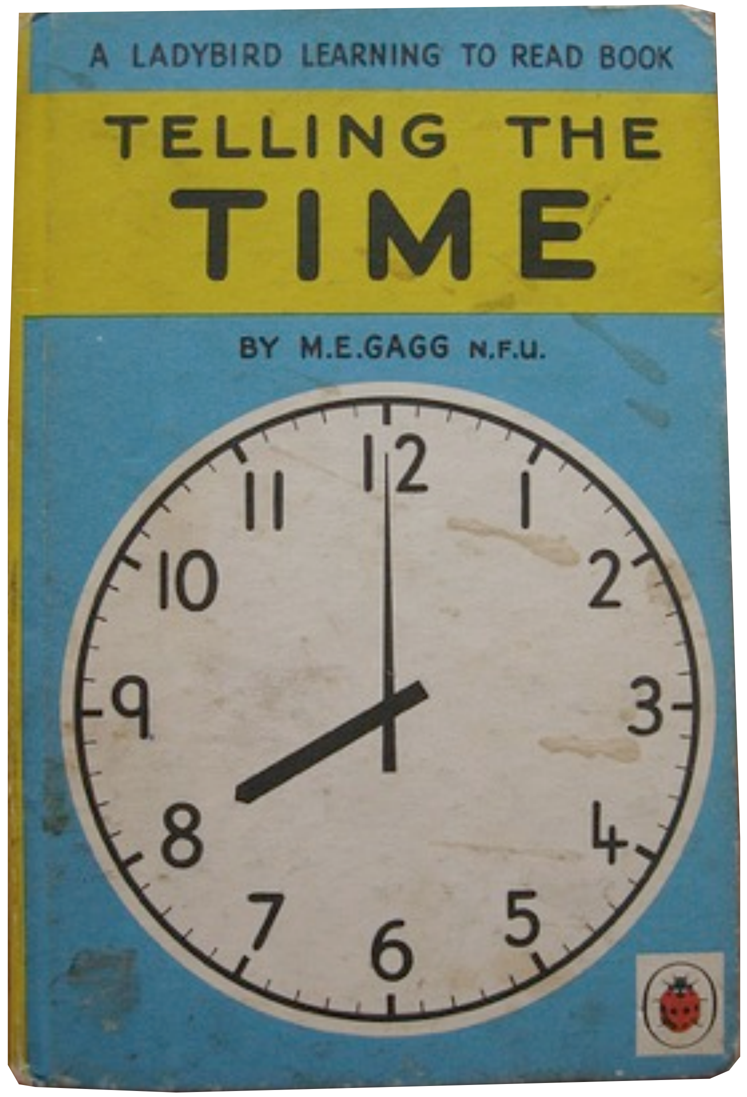
CONJECTURE 3. Dedalus Time $\Leftrightarrow$ Coordination Complexity. *The minimum number of Dedalus timesteps required to evaluate a program on a given input data set is equivalent to the program's Coordination Complexity.*

IMPLICATIONS AND CONJECTURES

- the CALM conjecture
- the CRON conjecture
- Coordination Complexity
- the Fateful Time conjecture

BUT WHAT IS TIME FOR?

- we've seen when we don't need it
  - monotonic deduction
- we've seen when we do need it
  - "spending time" examples
- if we need it but try to save it?
  - no unique minimal model!
    - multiple simultaneous worlds
    - paradoxes: inconsistent assertions in time

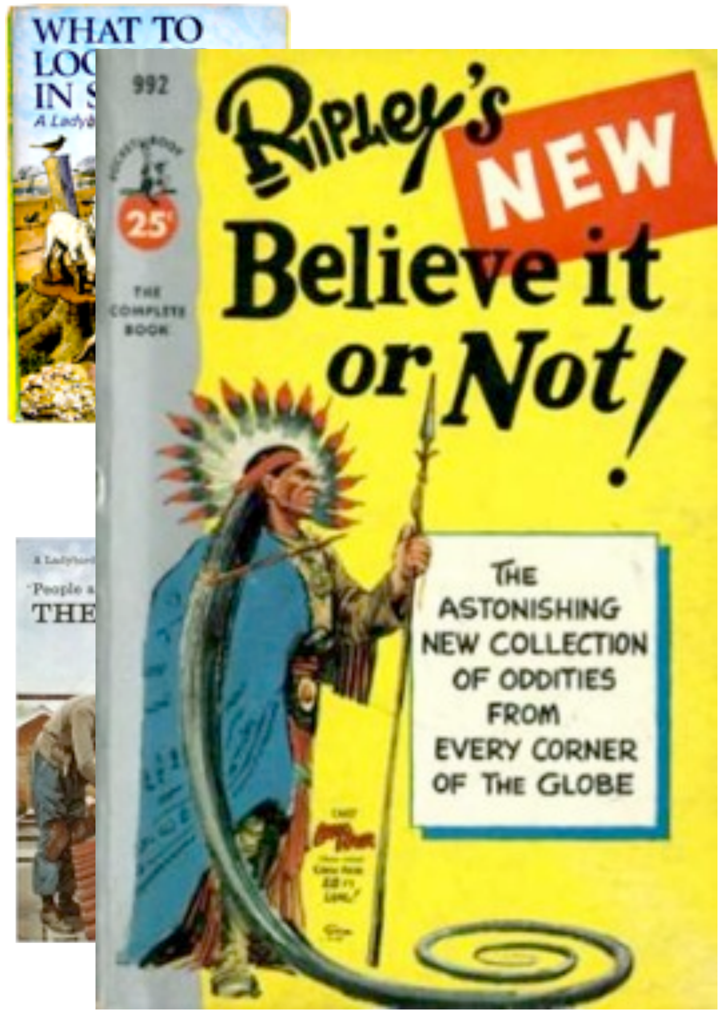


FATEFUL TIME

CONJECTURE 4. Fateful Time. *Any Dedalus program P can be rewritten into an equivalent* temporally-minimized *program P' such that each inductive or asynchronous rule of P' is* necessary*: converting that rule to a deductive rule would result in a program with no unique minimal model*.

- the purpose of time is to *seal fate:*
  - time = simultaneity + succession
    - dedalus: timestamp unification + inductive rules
  - multiple worlds ⇒ monotonic sequence of unique worlds

TODAY


WHAT TO LOO IN S
992
25¢
THE COMPLETE BOOK
Ripley's NEW
Believe it or Not!
THE ASTONISHING NEW COLLECTION OF ODDITIES FROM EVERY CORNER OF THE GLOBE

- two unfinished stories
- a dedalus primer
- experience
- implications and conjecture

TODAY

- two unfinished stories
- a dedalus primer
- experience
- implications and conjecture

WHAT TO LOOK FOR IN SPRING
A Ladybird Book
Chicken Licken
A LADYBIRD LEARNING TO READ BOOK
GOING TO SCHOOL
A Ladybird 'Easy Reading' Book
'People at Work'
THE BUILDER
Ripley's NEW Believe it or Not!
The astonishing new collection of oddities from every corner of the globe

WHAT NEXT? PITFALLS, PROMISE & POTENTIAL

- audacity of scope
  - pitfall: database languages *per se*
  - promise: data finally the central issue in computing
  - potential: attack the general case, change the way software is built
- formalism
  - pitfall: disconnection of theory/practice
  - promise: theory embodied in useful programming tools
  - potential: validate and extend a 30-year agenda
- networking
  - pitfall: the walled garden
  - promise: db topics connect pl, os, distributed systems, etc.
  - potential: db as an intellectual crossroads

CARPE DIEM

- affirm, refute, or ignore the conjectures
  - (thank you for indulging me)
- but do not miss this opportunity!
  - we can address a real crisis in computing
  - we have the ear of the broad community
  - time to sift through known results and apply them
  - undoubtedly there is more to do .. jump in!

JOINT WORK

- 7 years
- 3 systems (P2, Overlog, DSN)
- 6 PhD, 2 MS students
- friends in academia, industry

- special thanks to the BOOM team:

Peter ALVARO
Ras BODÍK
Tyson CONDIE
Neil CONWAY
Khaled ELMELEEGY
Haryadi GUNAWI
Thibaud HOTTELIER
William MARCZAK
Rusty SEARS

BERKELEY Database Research

YAHOO! RESEARCH

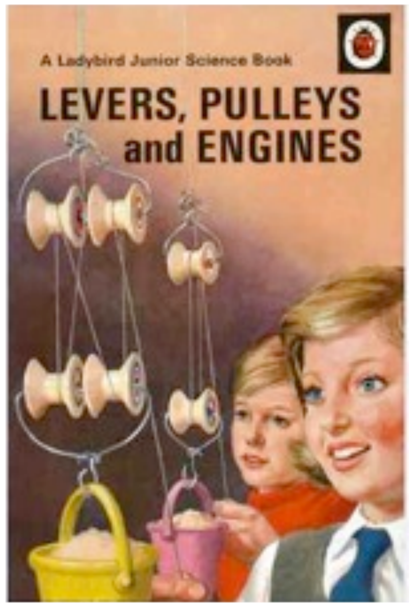
Intel Labs Berkeley

intel

web search: “springtime for datalog”

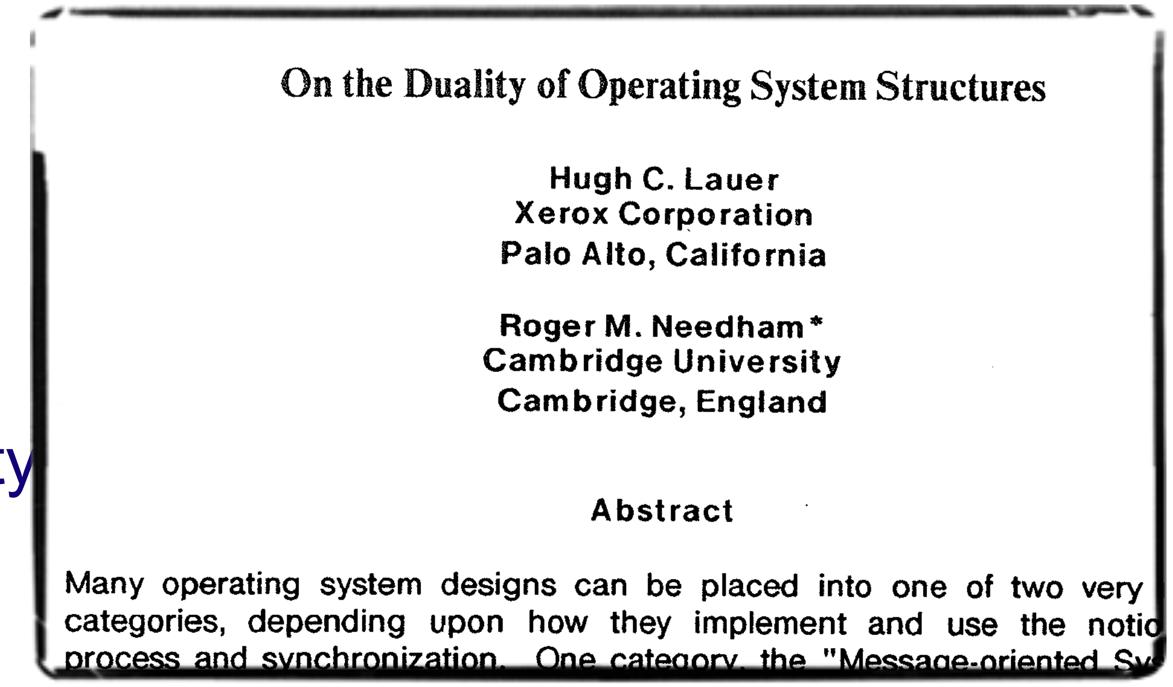
http://boom.cs.berkeley.edu

BACKUP



DESIGN PATTERN #3 EVENTS AND DISPATCH

- challenge: manage thousands of sessions on a server
  - A. “process” or “thread” per session.
    - stack variables and PC keep context
  - B: one single-threaded event-loop
    - state-machine per session on heap
    - problem: long tasks like I/O require care
  - arguments about scaling, programmability
- session mgmt is just data mgmt!
  - scale a join to thousands of tuples? big deal!!
  - programmability? hmm...



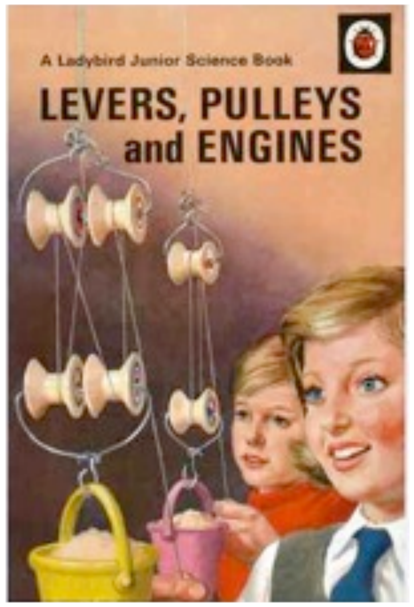
On the Duality of Operating System Structures

Hugh C. Lauer
Xerox Corporation
Palo Alto, California

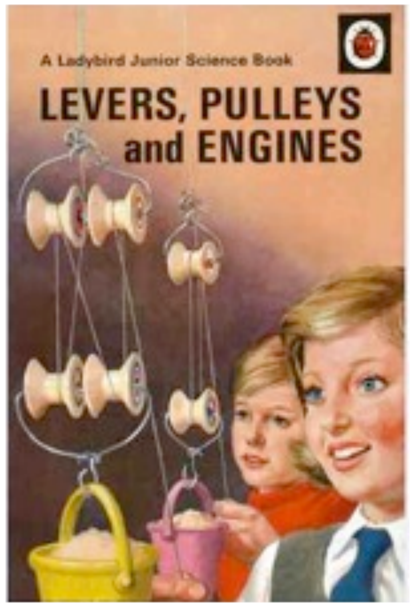
Roger M. Needham*
Cambridge University
Cambridge, England

Abstract

Many operating system designs can be placed into one of two very categories, depending upon how they implement and use the notio process and synchronization. One category, the "Message-oriented Sy

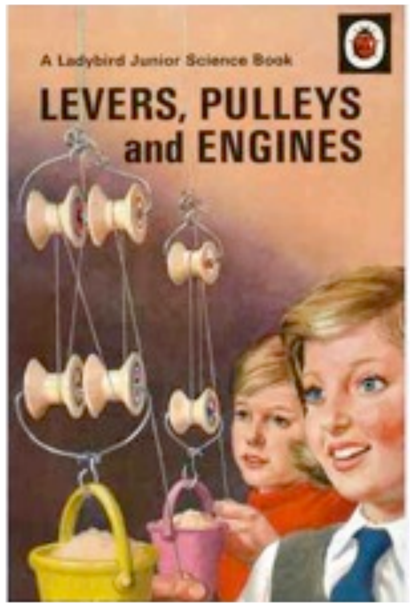


A THIRD WAY



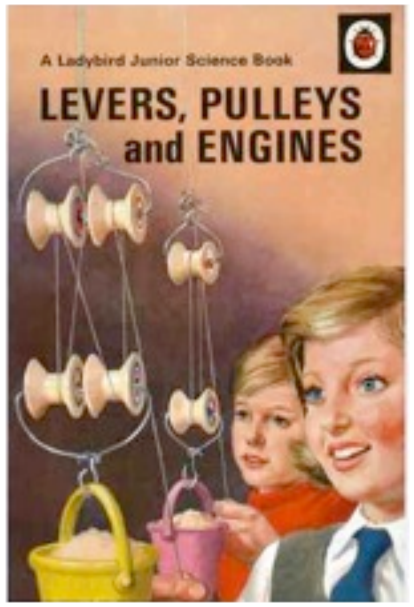
A THIRD WAY

```
// keep requests pending until a response is generated
pending(Id, Clnt, P) :- request(Clnt, Id, P).
pending(Id, Clnt, P)@next :- pending(Id, Clnt, P),
                            !response(Id, Clnt, _).
```



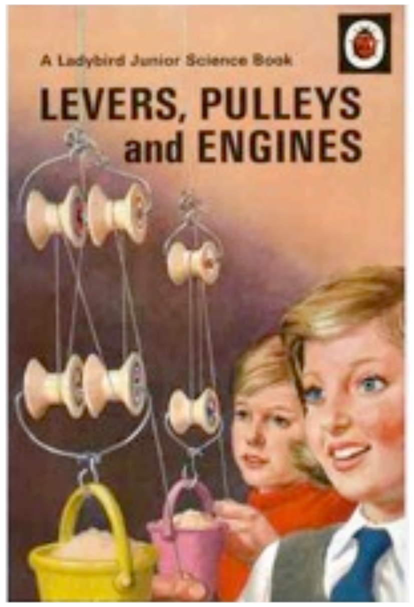
A THIRD WAY

```
// keep requests pending until a response is generated
pending(Id, Clnt, P) :- request(Clnt, Id, P).
pending(Id, Clnt, P)@next :- pending(Id, Clnt, P),
                              !response(Id, Clnt, _).

// call an asynchronous service, via input "interface" service_in()
service_out(P, Out)@async :- request(Id, Clnt, P),
                              service_in(P, Out).
```



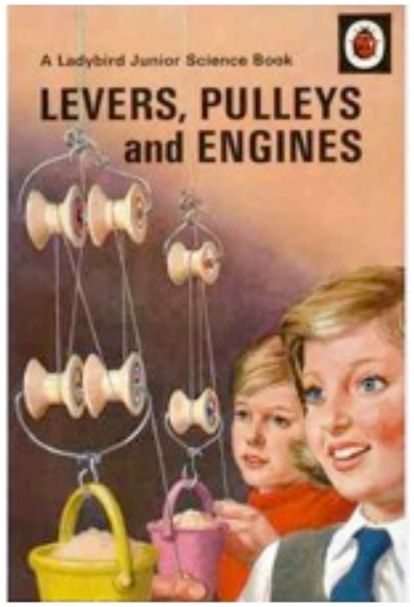
A THIRD WAY

```
// keep requests pending until a response is generated
pending(Id, Clnt, P) :- request(Clnt, Id, P).
pending(Id, Clnt, P)@next :- pending(Id, Clnt, P),
                             !response(Id, Clnt, _).

// call an asynchronous service, via input "interface" service_in()
service_out(P, Out)@async :- request(Id, Clnt, P),
                             service_in(P, Out).

// join service answers back to pending to form response
response(Clnt, Id, O) :- pending(Id, Clnt, P), service_out(P, O).
```



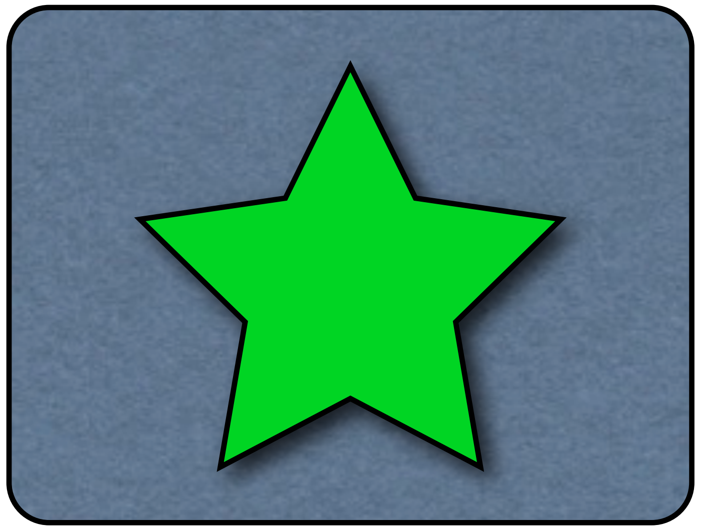
EPHEMERA

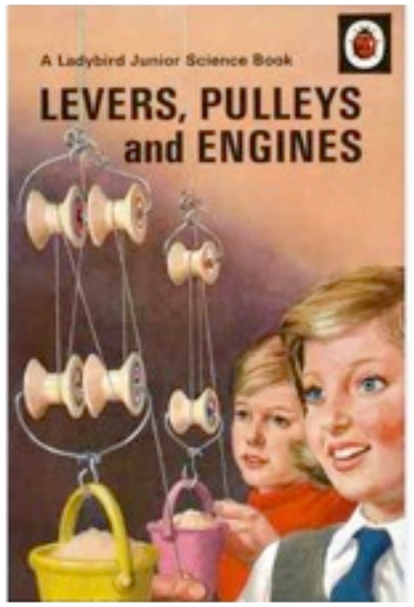
- 3 common distributed persistence models
  - stable storage (persistent)
  - event streams (ephemeral)
  - soft state (bounded persistence)



EPHEMERA

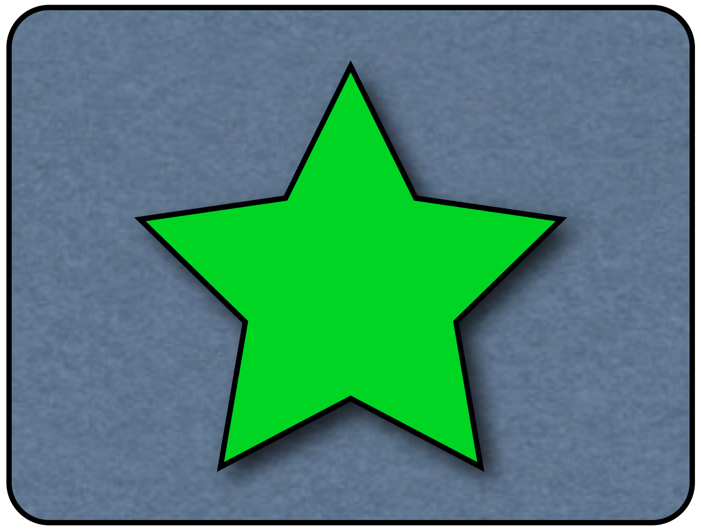
- 3 common distributed persistence models
  - stable storage (persistent)
  - event streams (ephemeral)
  - soft state (bounded persistence)

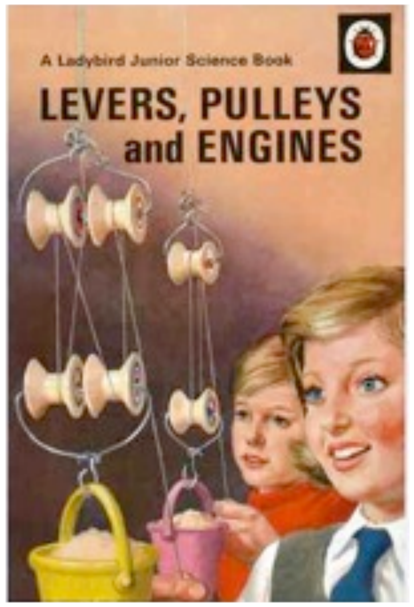




EPHEMERA

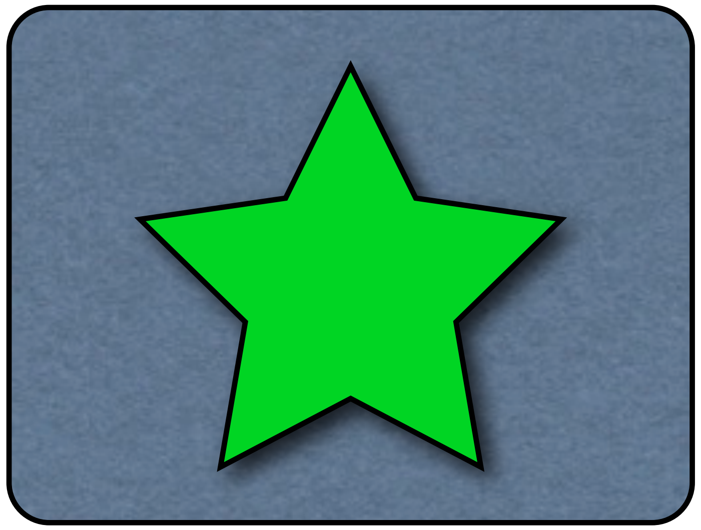
- 3 common distributed persistence models
  - stable storage (persistent)
  - event streams (ephemeral)
  - soft state (bounded persistence)

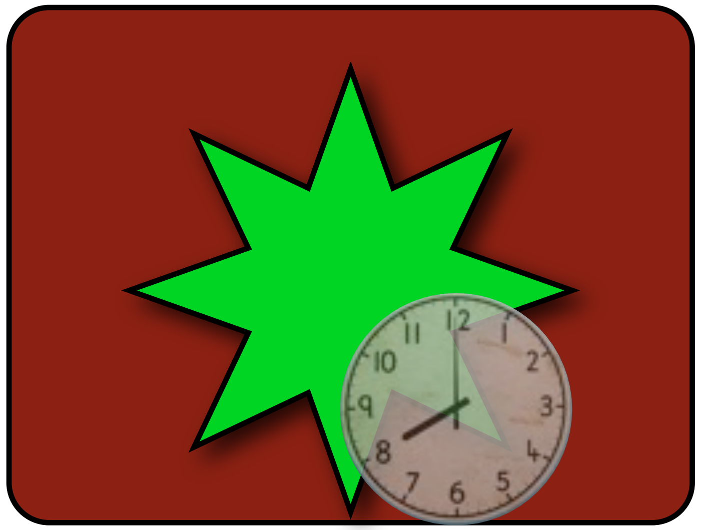


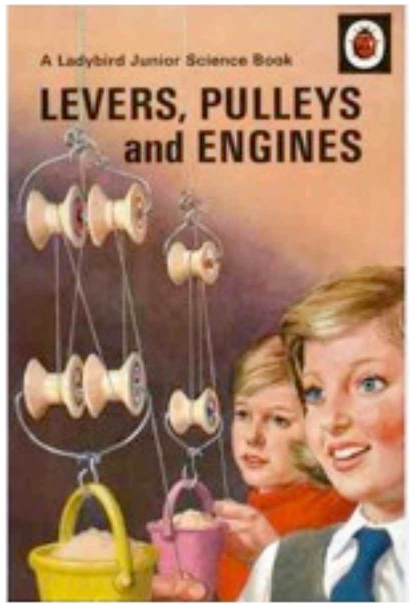


EPHEMERA

- 3 common distributed persistence models
  - stable storage (persistent)
  - event streams (ephemeral)
  - soft state (bounded persistence)

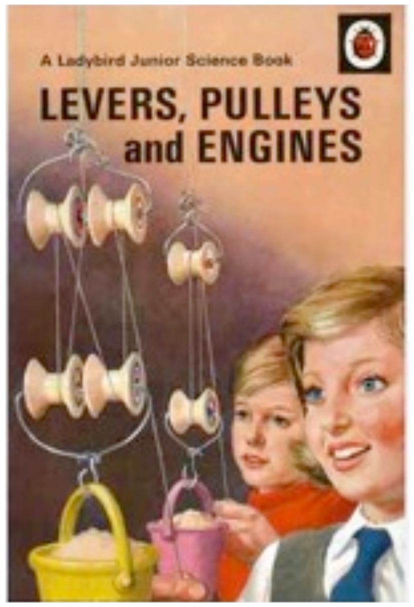






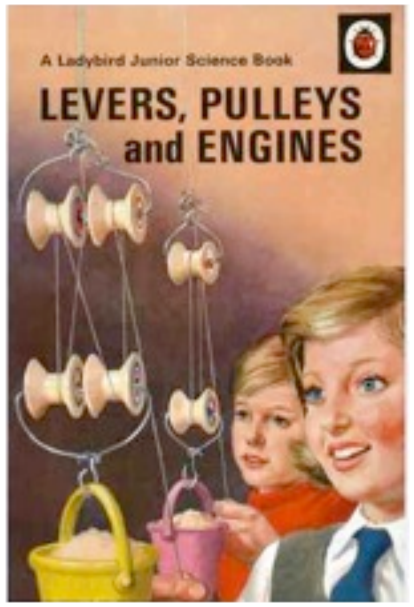
EPHEMERA

- 3 common distributed persistence models
  - stable storage (persistent)
  - event streams (ephemeral)
  - soft state (bounded persistence)



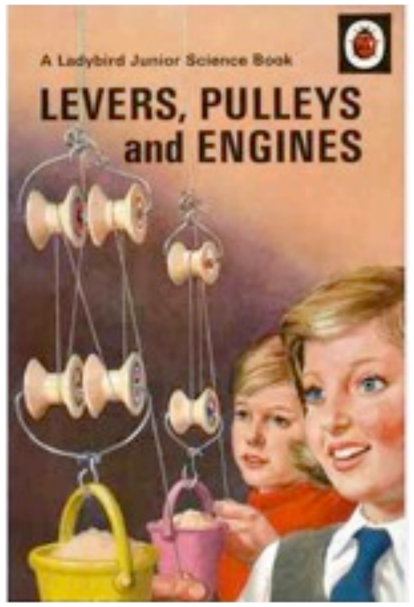
OVERLOG: PERIODICS AND PERSISTENCE

- Overlog provided metadata modifiers for persistence
  materialize(pods, infinity).
  materialize(cache, 60).
  - absence of a materialize clause implies an emphemeral event stream
- Overlog's built-in event stream:
  periodic(@Node, Id, Interval).
  - a declarative construct, to be evaluated in real-time



CACHING EXAMPLE IN OVERLOG

```
materialize(pods, infinity).
materialize(msglog, infinity).
materialize(link, infinity).
materialize(cache, 60).

cache(@N, X) :- pods(@M, X), link(@M, N),
                periodic(@M, _, 40).      ← cool!

msglog(@N, X) :- cache(@N, X).     ← but what does that mean??
```

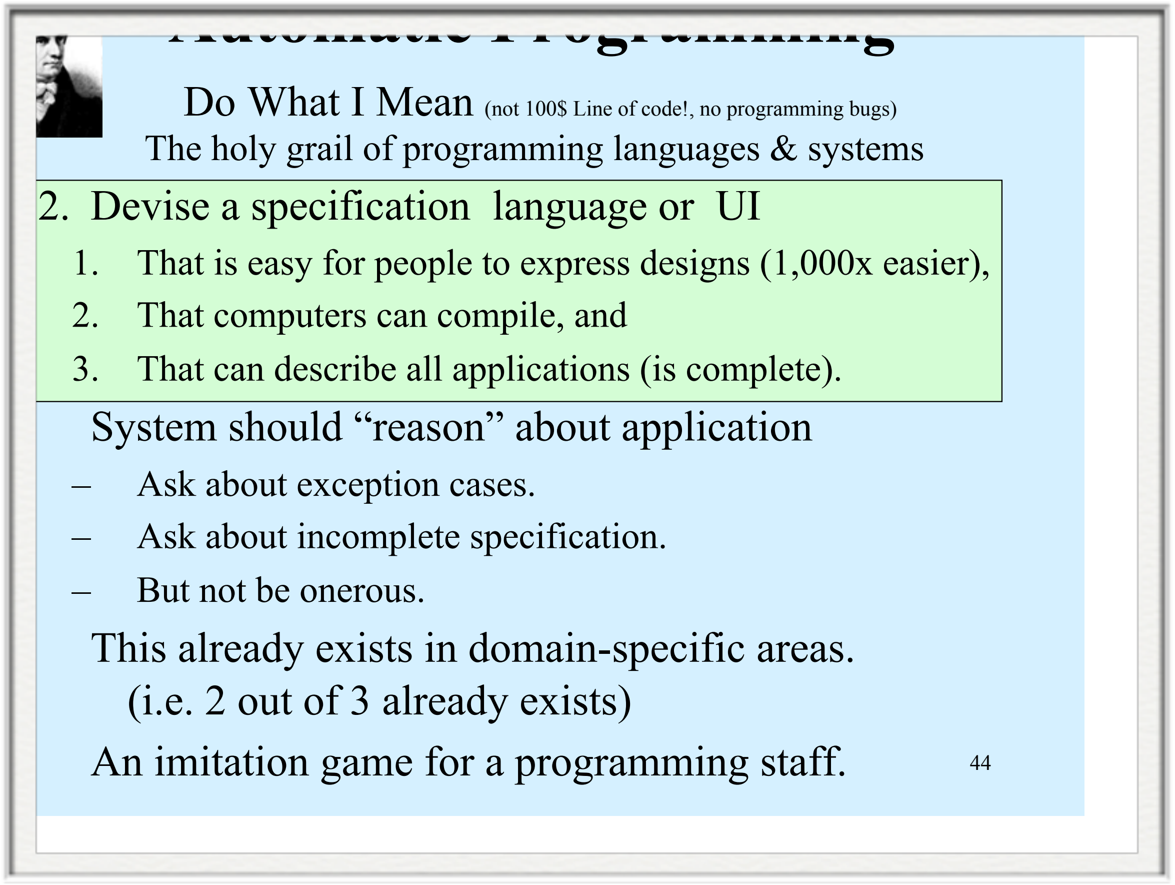


CACHING IN DEDALUS

```
pods(@M, X)@next :- pods(@M,X), !del_pods(@M,X).
msglog(@M,X)@next) , msglog(@M,X), !del_msglog(@M,X).
link(@M, X)@next :- link(@M,X), !del_link(@M,X).
cache(@M,X,Birth)@next :- cache(@M,X,Birth), now() - Birth > 60.

cache(@N, X) :- pods(@M, X), link(@M, N),
                periodic(@M, _, 40).              ← still cool!

msglog(@N, X) :- cache(@N, X).    ← in tandem with inductive rule above,
                                     msglog grounded in this base-case!
```

GRAY'S TWELFTH CHALLENGE

- "automatic" programming
  - Do What I Mean
  - 3 OOM "easier"
- with Memex, Turing Test, etc.
- predates multicore/cloud
  - the sky had already fallen?



Do What I Mean (not 100$ Line of code!, no programming bugs)
The holy grail of programming languages & systems

2. Devise a specification language or UI
   1. That is easy for people to express designs (1,000x easier),
   2. That computers can compile, and
   3. That can describe all applications (is complete).

System should "reason" about application
- Ask about exception cases.
- Ask about incomplete specification.
- But not be onerous.

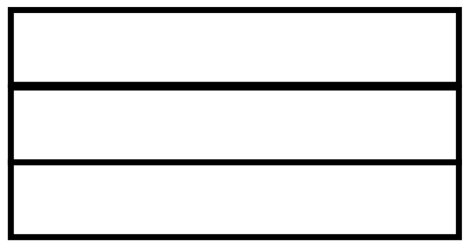
This already exists in domain-specific areas.
(i.e. 2 out of 3 already exists)
An imitation game for a programming staff.

44

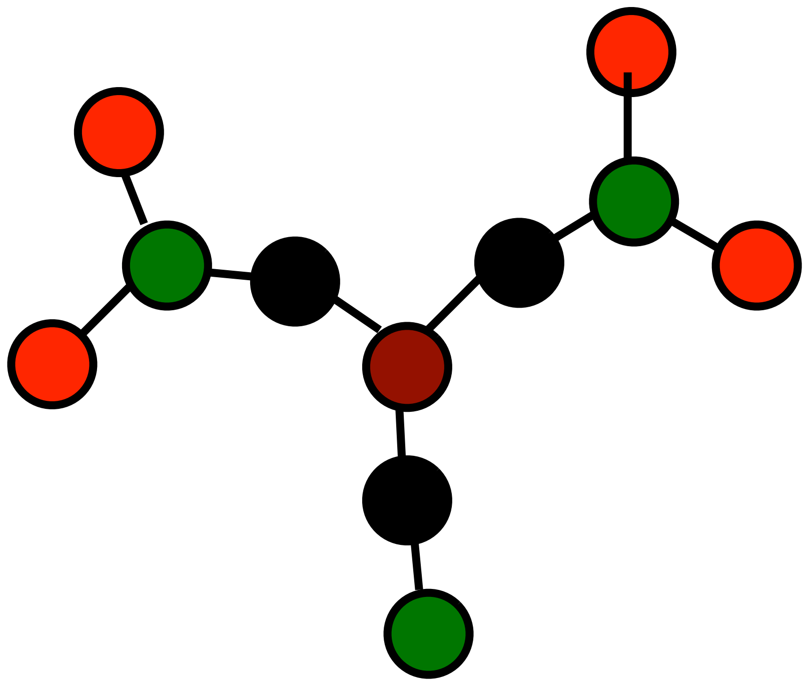
MONOTONIC? EMBARRASSING!

- Monotonic evaluation is order-independent
  - derivation trees "accumulate"
- Loo's Pipelined Semi-Naive evaluation
  - streaming (monotonic) Datalog
  - same # derivations as Semi-Naive
  - Intuition: network paths again

SEMI-NAIVE EVALUATION



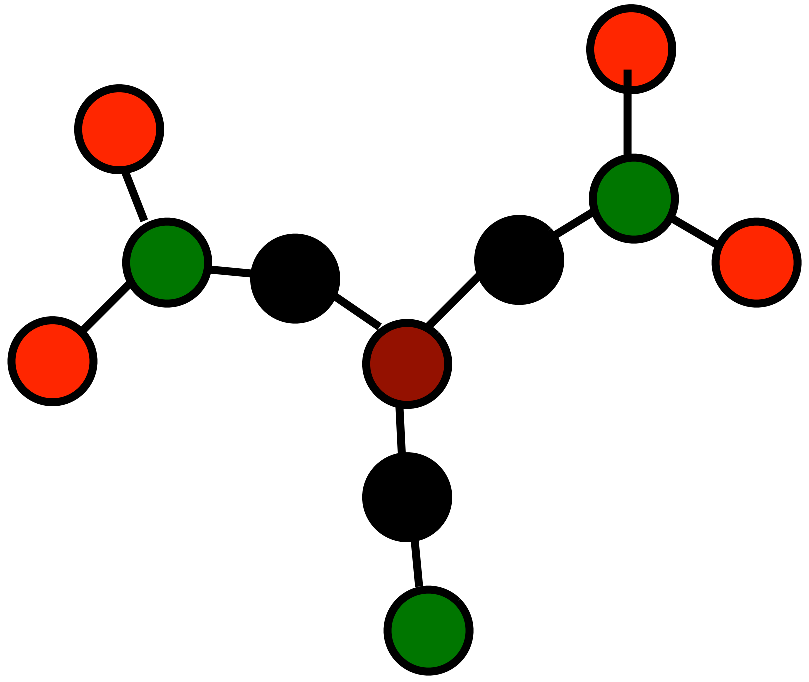
Link Table



Network

Slide courtesy Boon Thau Loo

SEMI-NAIVE EVALUATION



Link Table

Path Table

Network

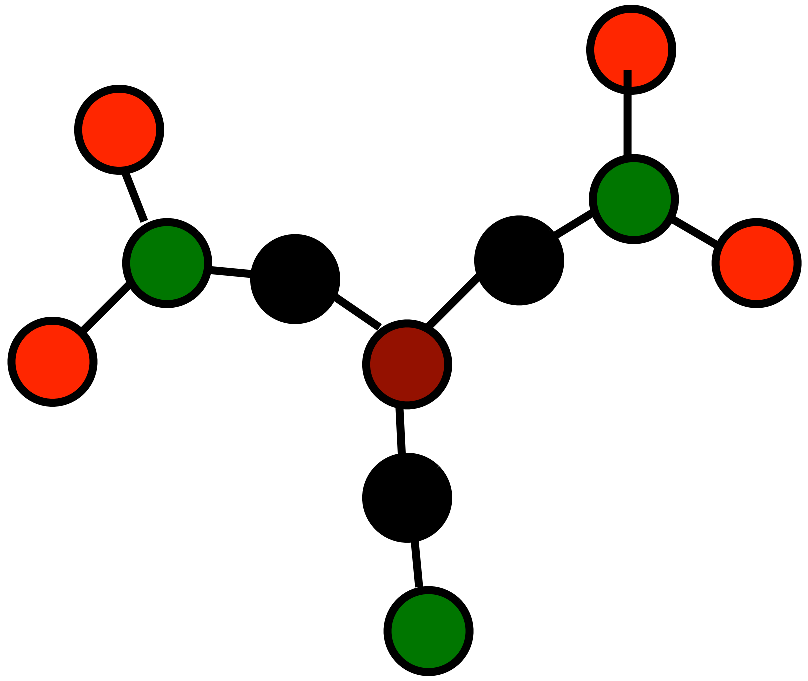
Slide courtesy Boon Thau Loo

SEMI-NAIVE EVALUATION

Link Table

	1-hop

Path Table



Network

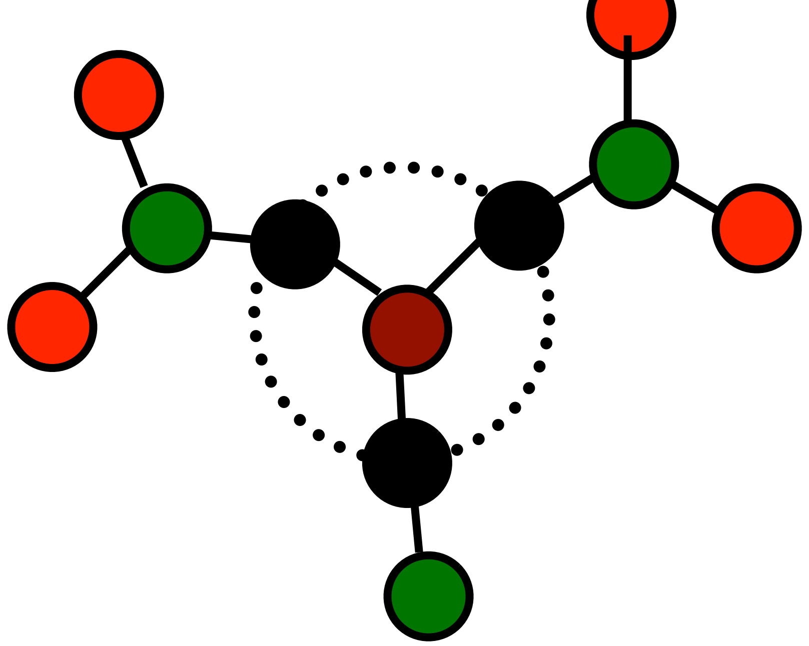
Slide courtesy Boon Thau Loo

SEMI-NAIVE EVALUATION

Link Table

	1-hop

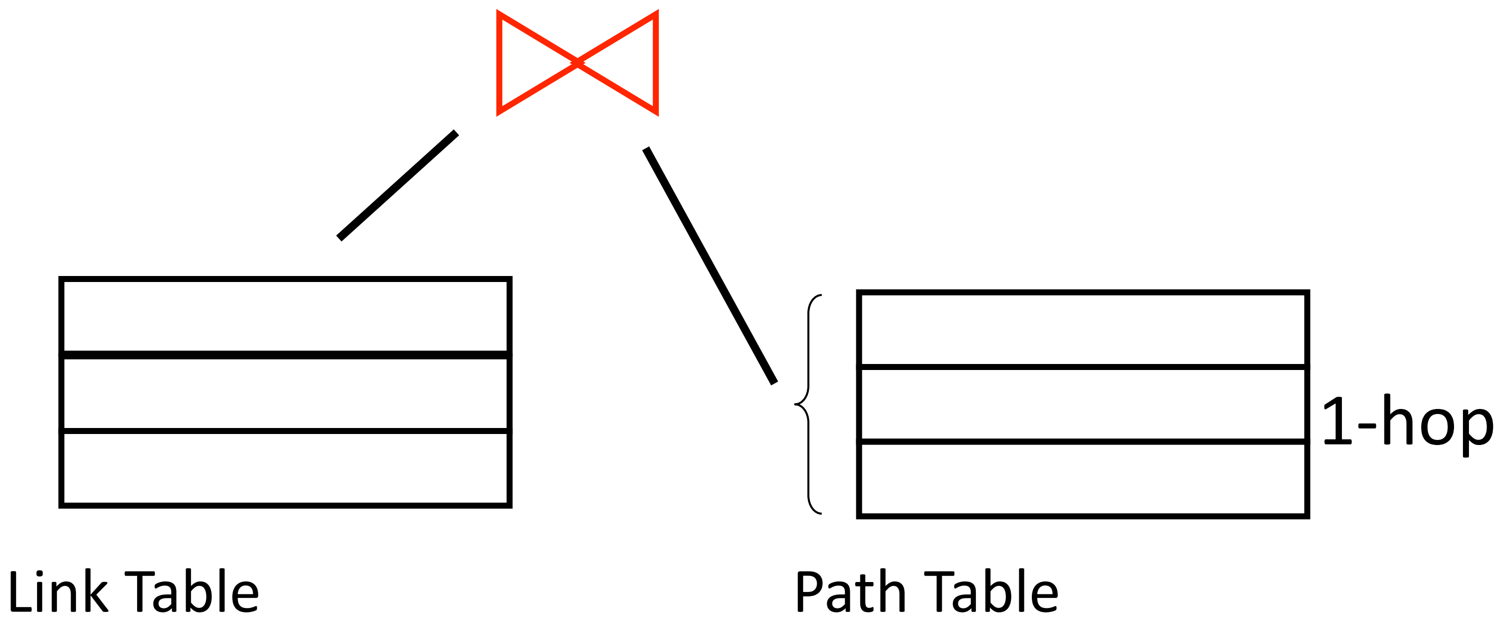
Path Table

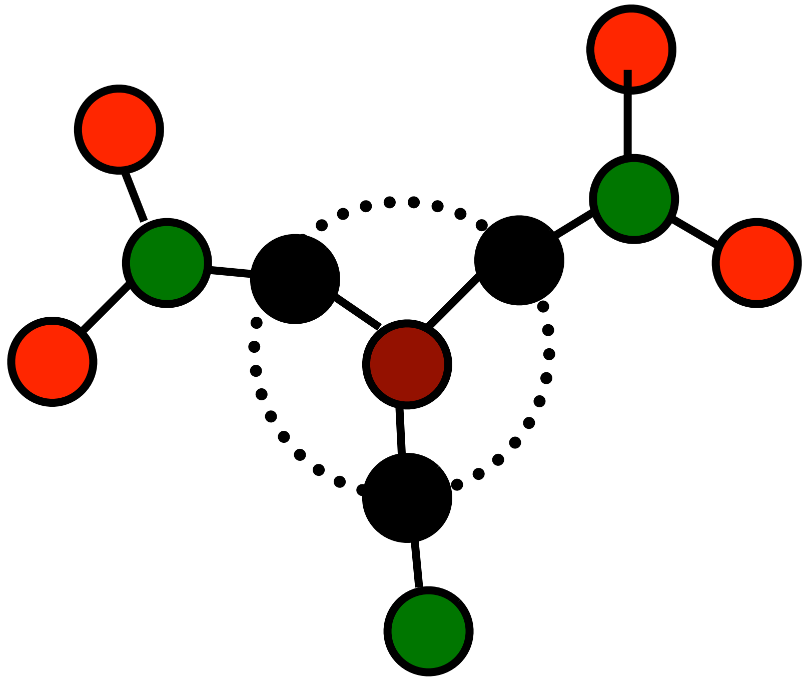


Network

Slide courtesy Boon Thau Loo

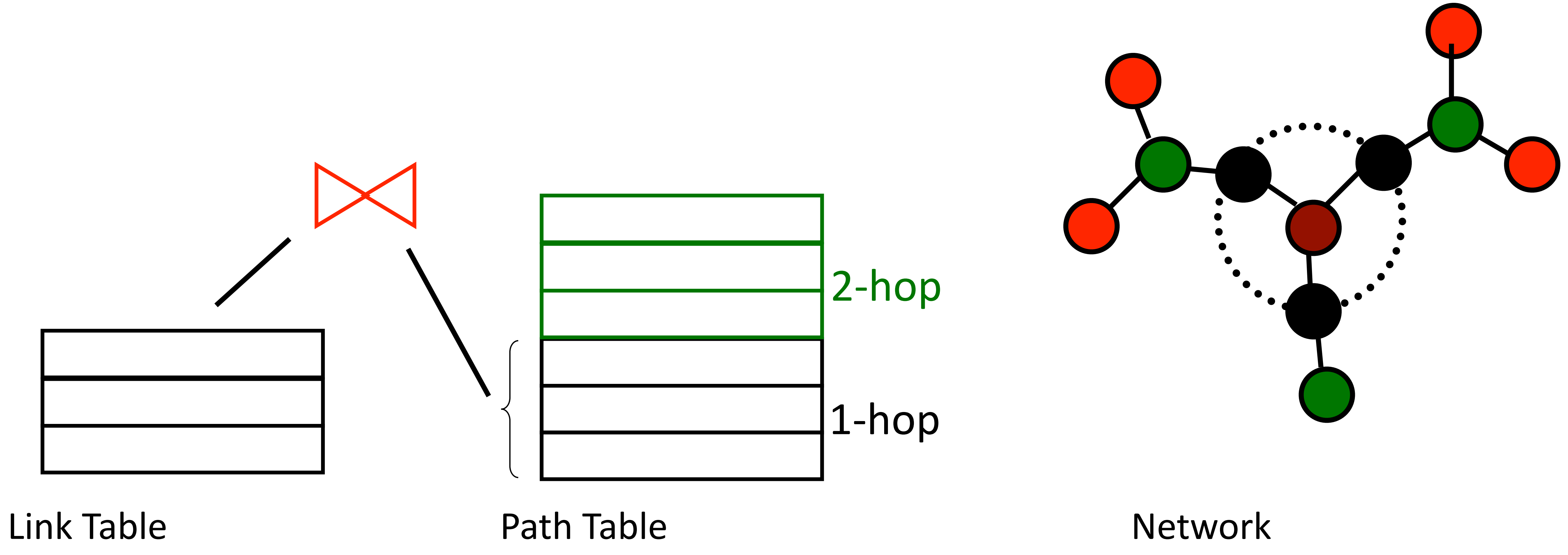
SEMI-NAIVE EVALUATION





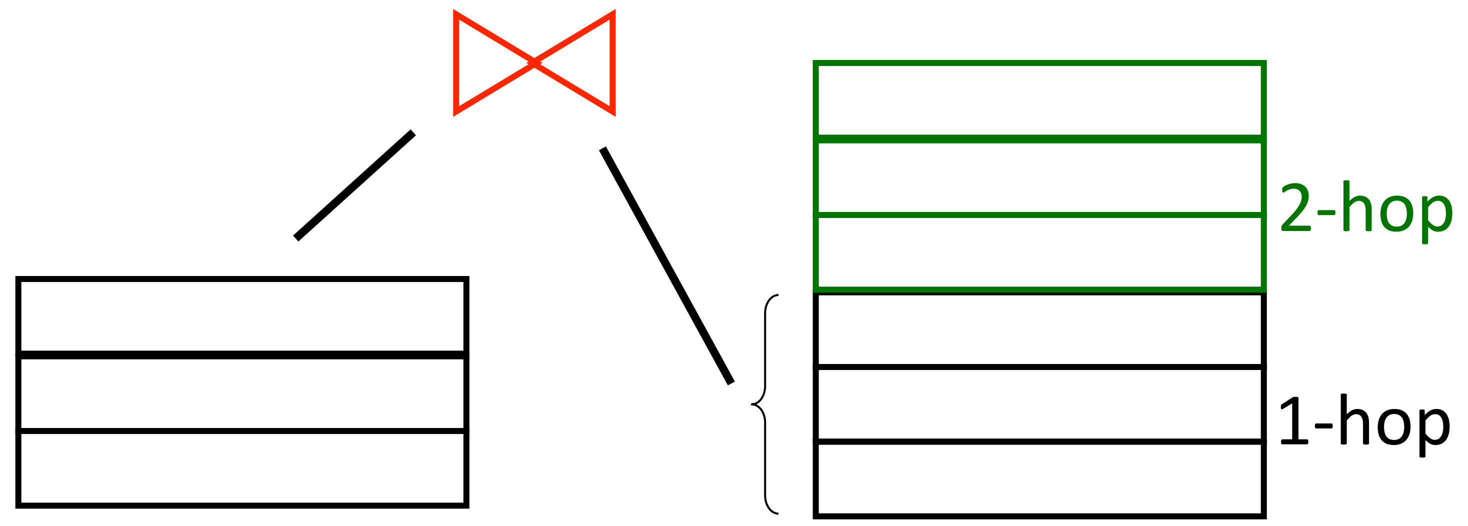
Slide courtesy Boon Thau Loo

SEMI-NAIVE EVALUATION



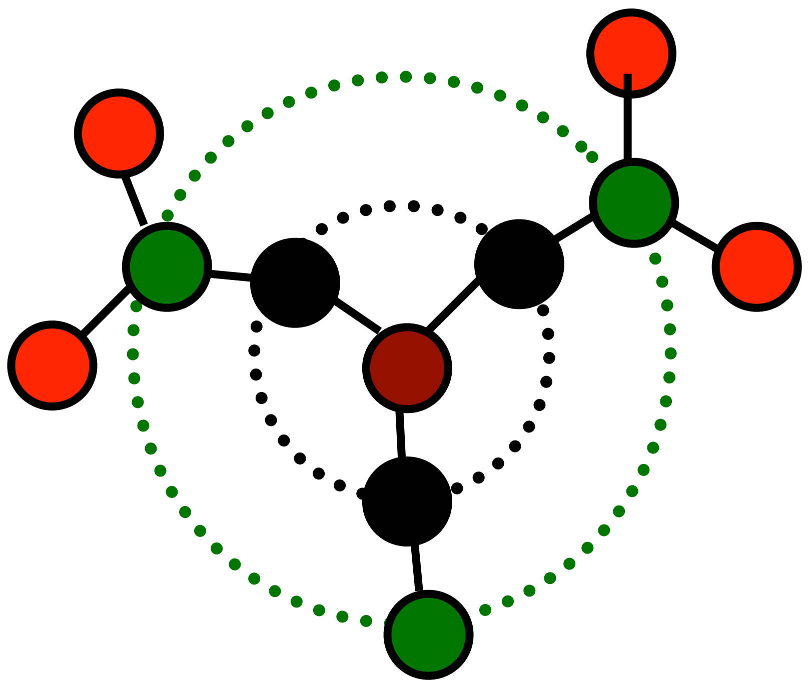
Slide courtesy Boon Thau Loo

SEMI-NAIVE EVALUATION



Link Table

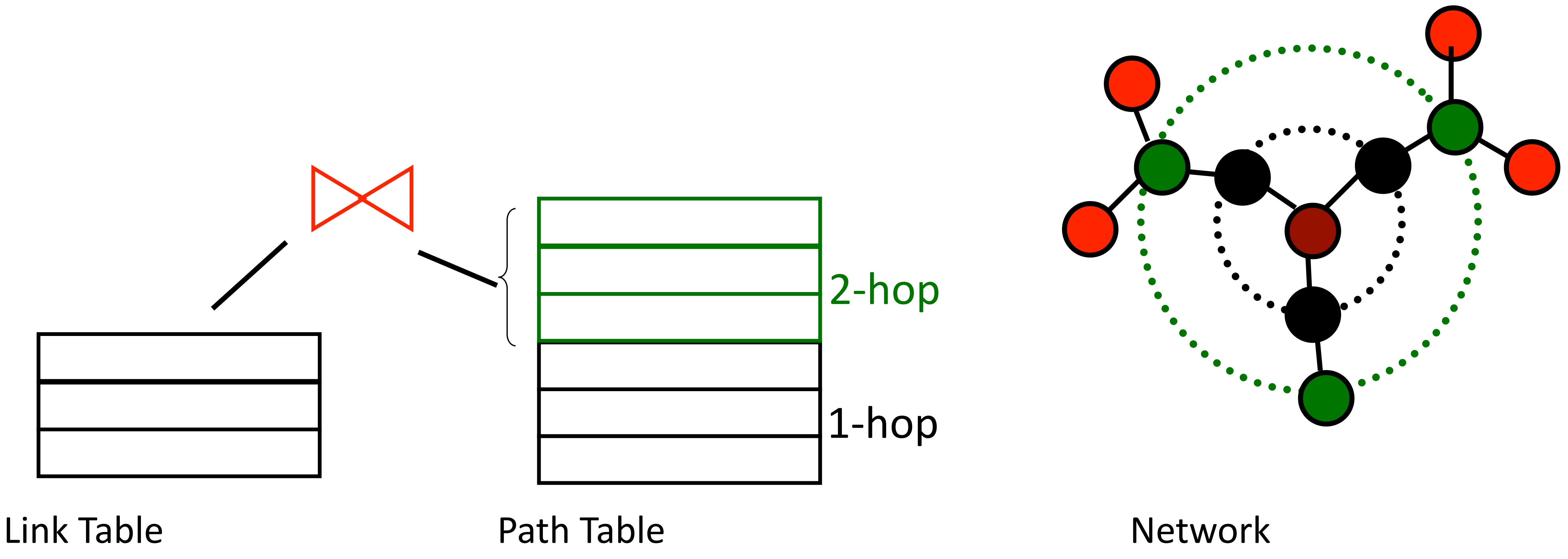
Path Table



Network

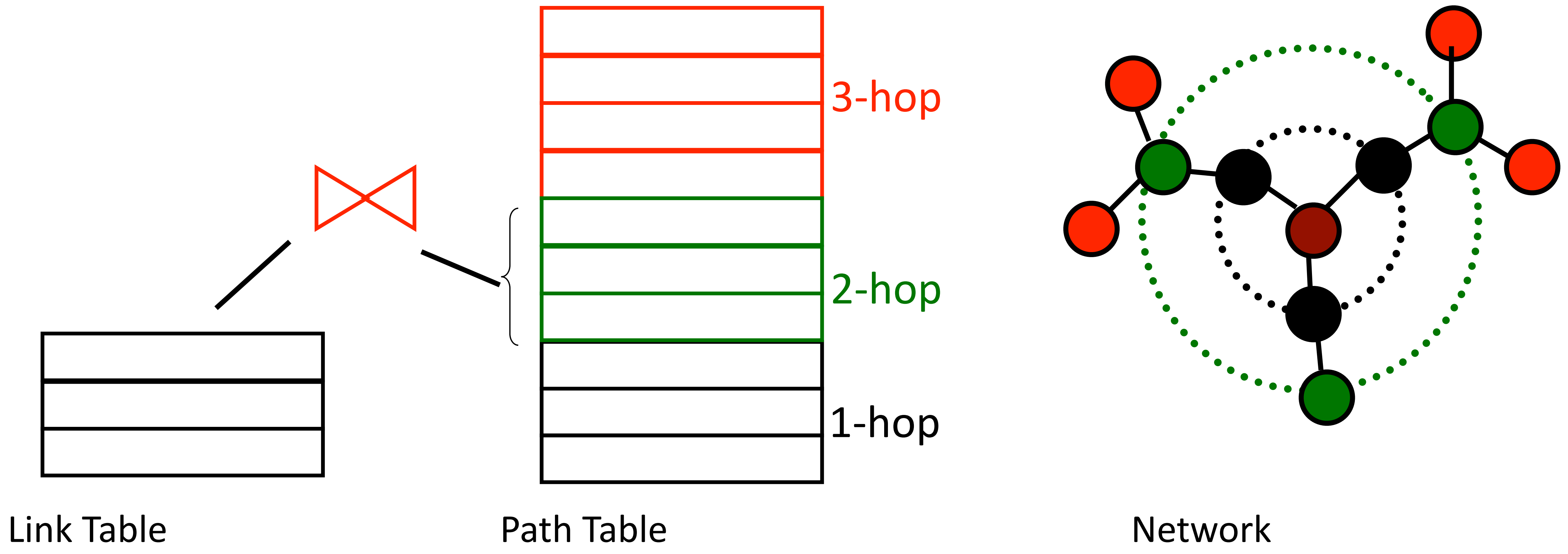
Slide courtesy Boon Thau Loo

SEMI-NAIVE EVALUATION



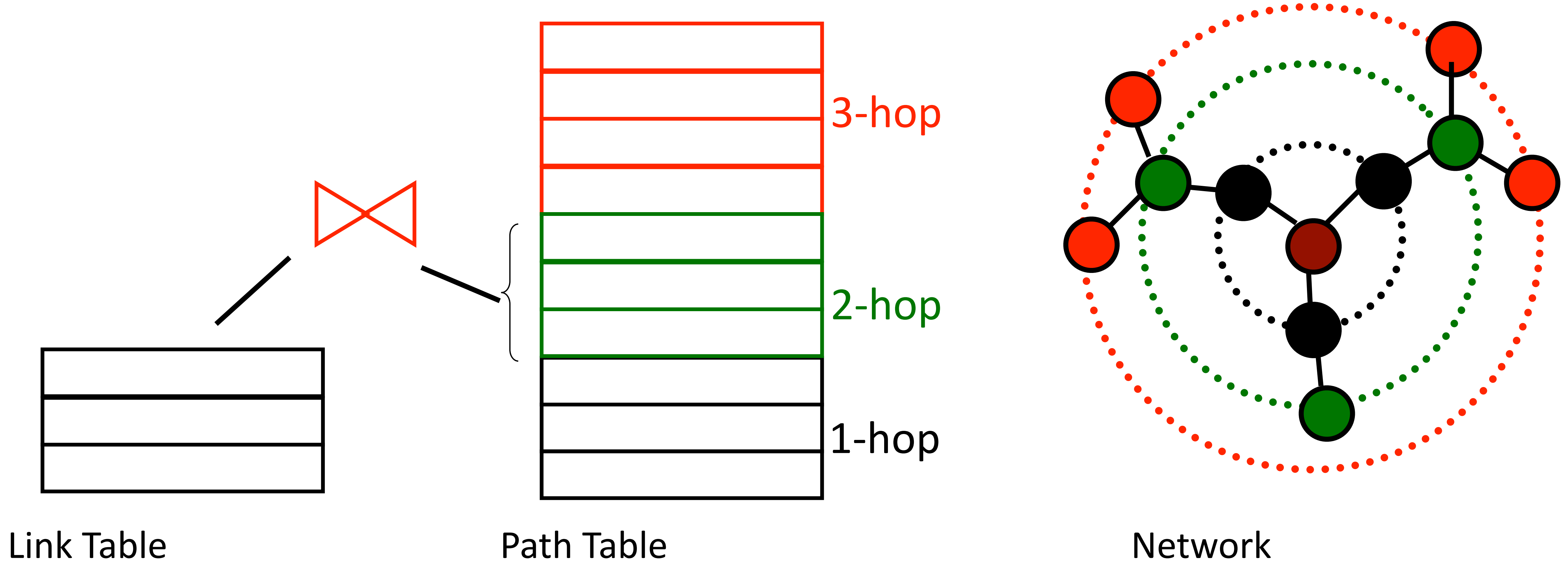
Slide courtesy Boon Thau Loo

SEMI-NAIVE EVALUATION



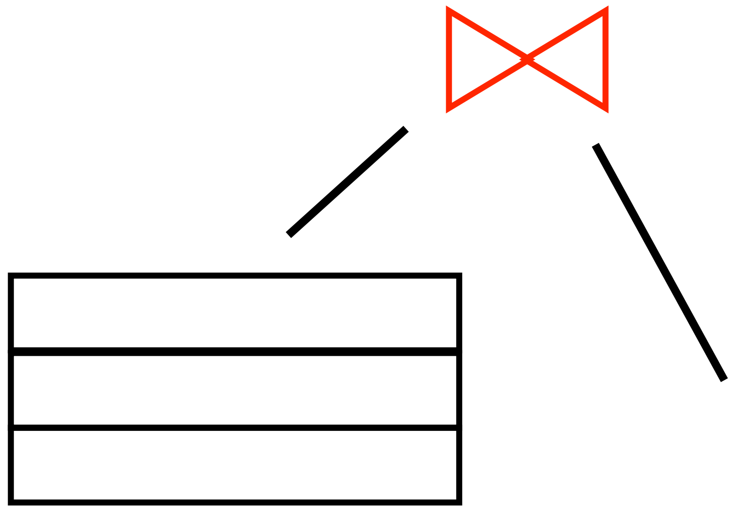
Slide courtesy Boon Thau Loo

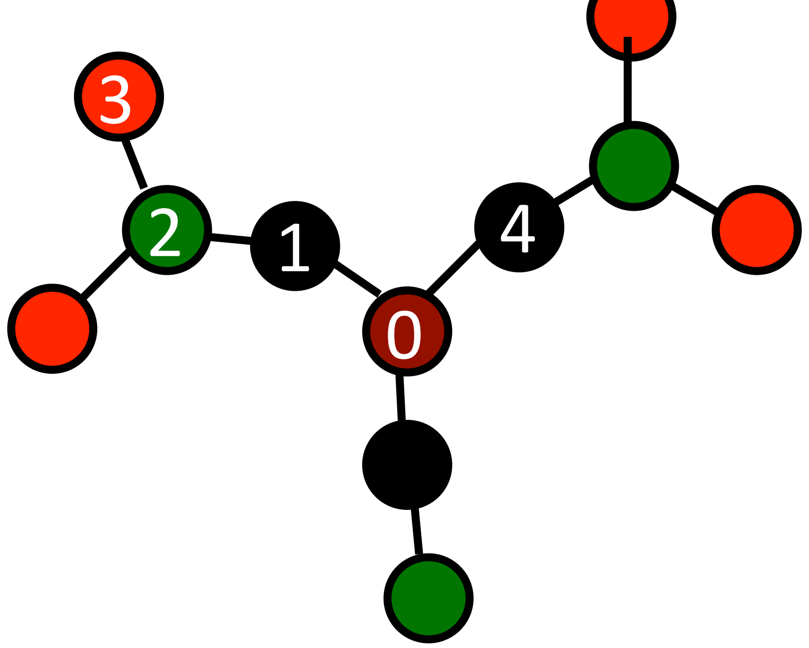
SEMI-NAIVE EVALUATION



Slide courtesy Boon Thau Loo

PIPELINED SEMI-NAIVE EVALUATION



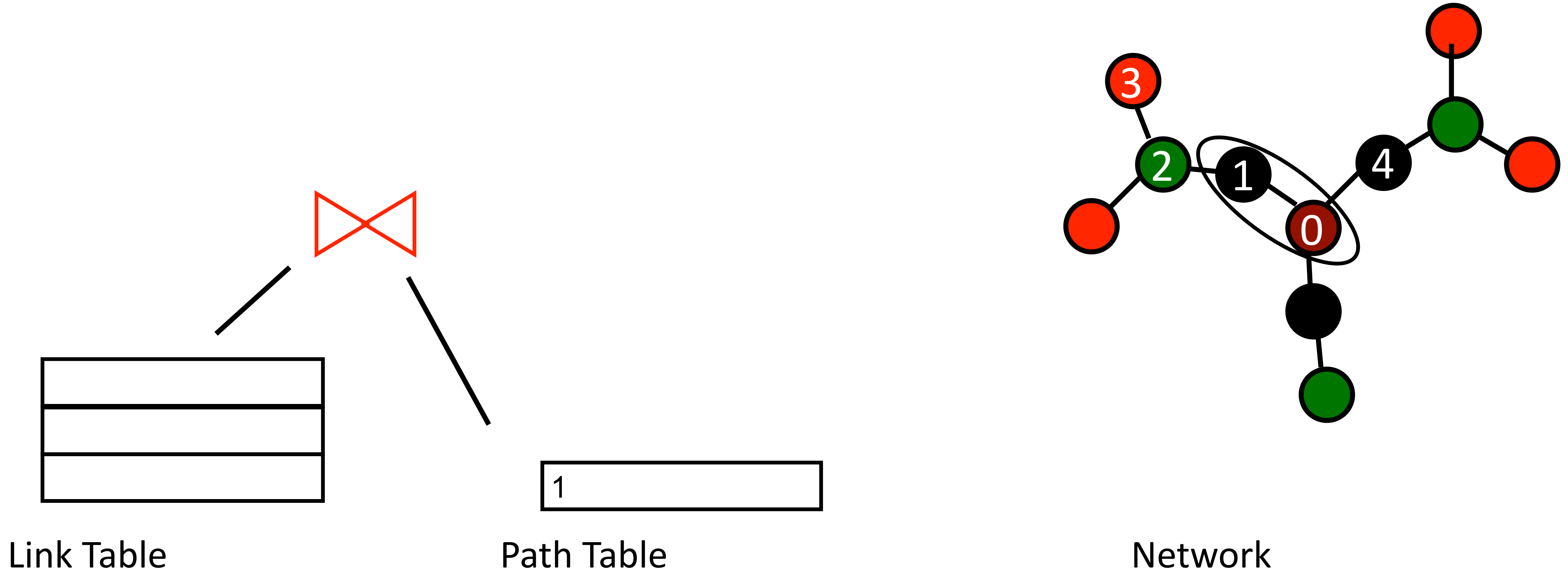


Link Table

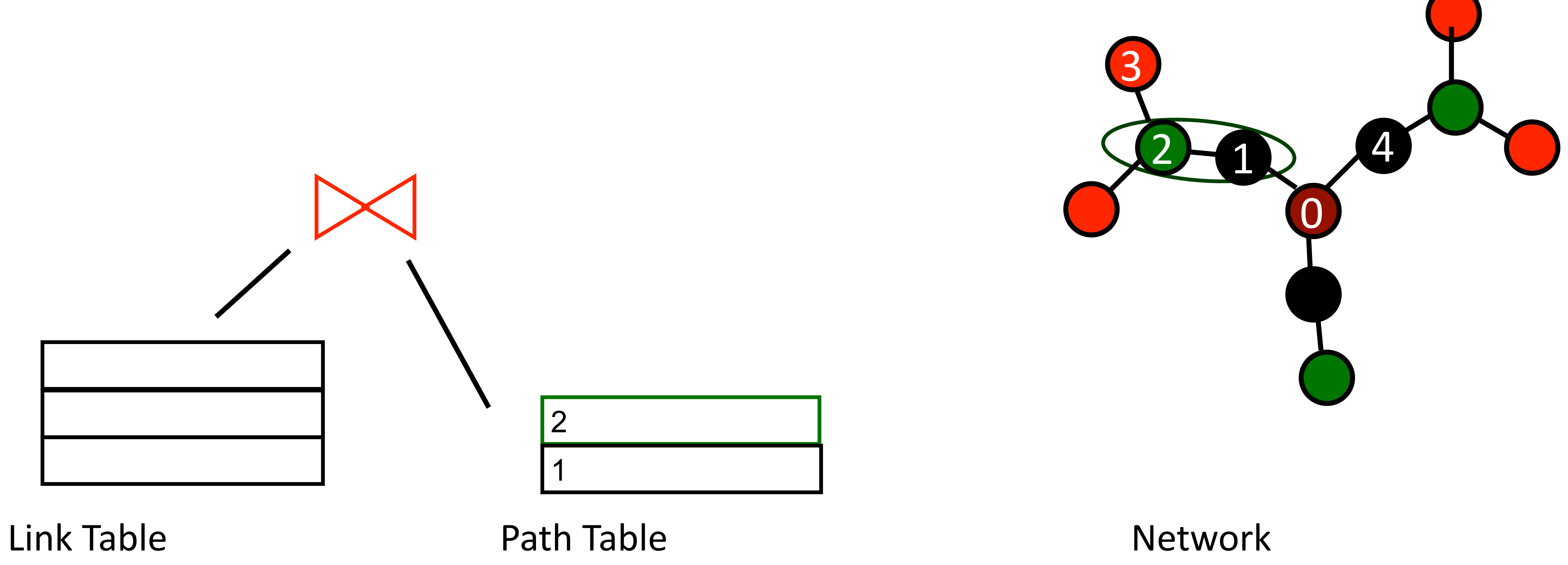
Path Table

Network

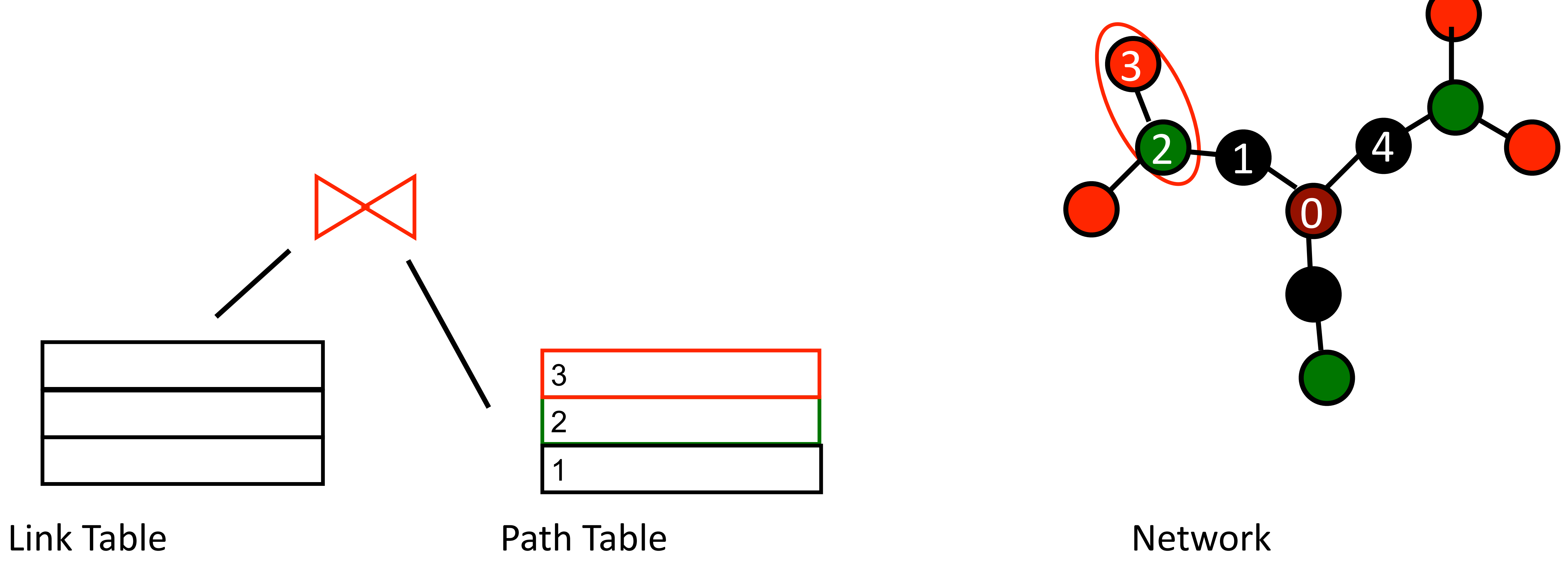
PIPELINED SEMI-NAIVE EVALUATION



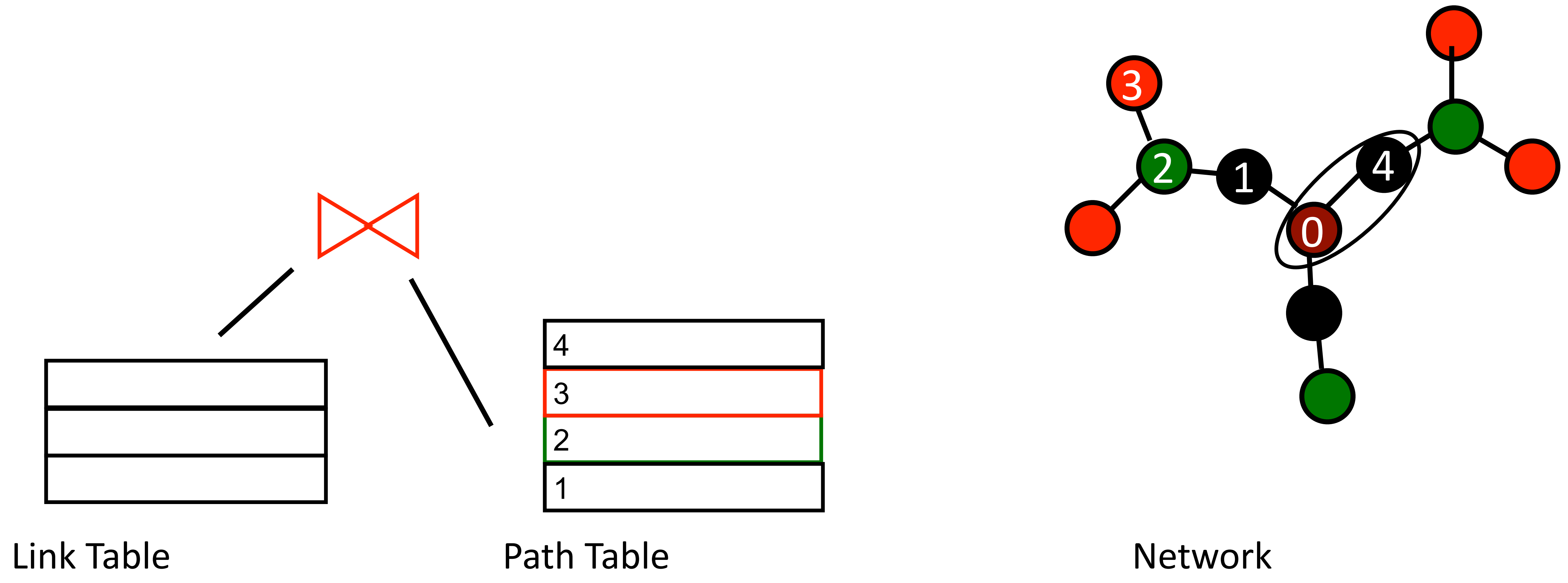
PIPELINED SEMI-NAIVE EVALUATION



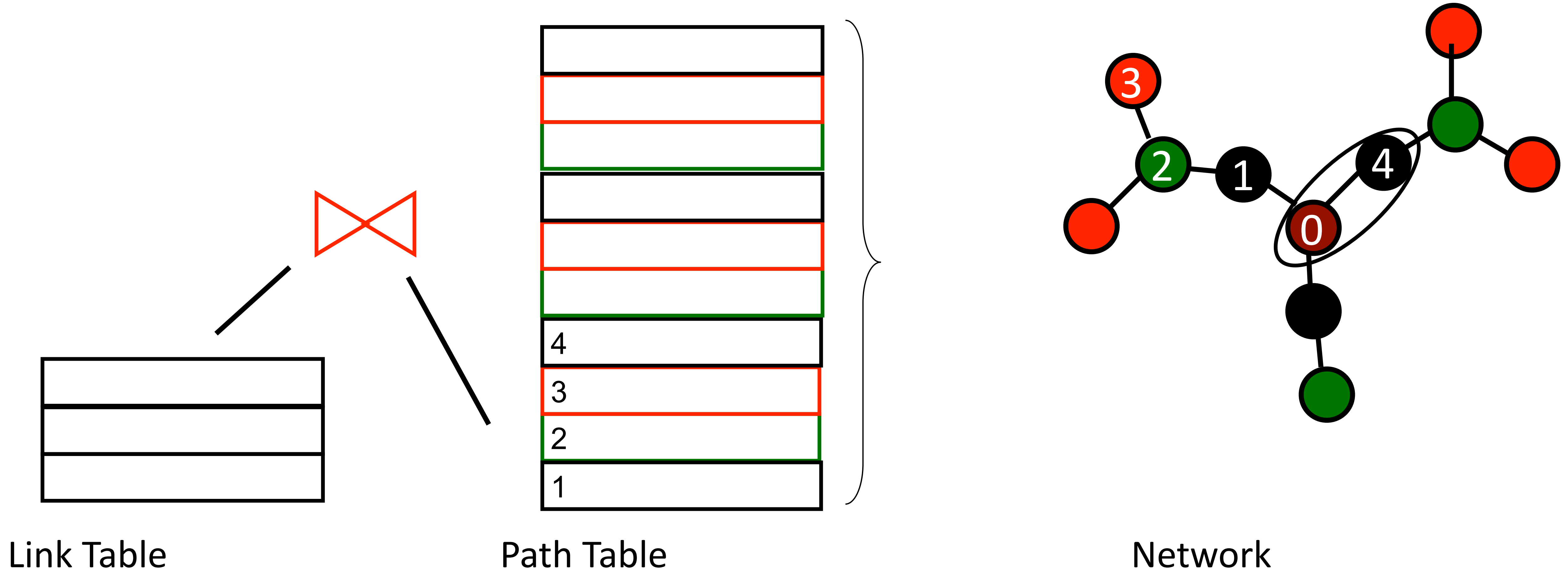
PIPELINED SEMI-NAIVE EVALUATION



PIPELINED SEMI-NAIVE EVALUATION



PIPELINED SEMI-NAIVE EVALUATION



BORGES SAID IT BETTER

- "The denial of time involves two negations: the negation of the succession of the terms of a series, negation of the synchronism of the terms in two different series."

  —Jorge Luis Borges, "A New Refutation of Time"