Concurrent Revisions

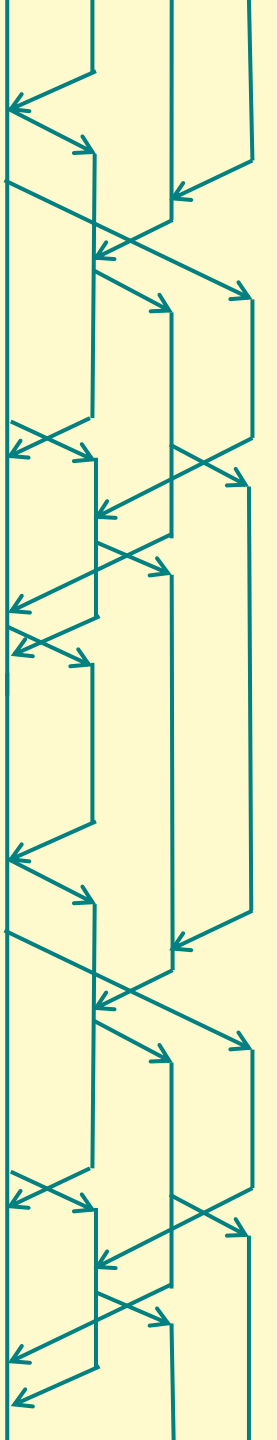
Lecture 2:
Incremental Computation
Eventual Consistency

Sebastian Burckhardt

Microsoft Research

Joint work with Manuel Faehndrich, Daan Leijen, Tom Ball
Caitlin Sadowski, Jaeheon Yi, Mooly Sagiv, Benjamin Wood

(see papers in OOPSLA 2011, ESOP 2012, ECOOP 2012)



Outline

- Parallel & Incremental Computation
  - Two for the price of one
- Eventual Consistency
  - Concurrent Revisions as a Consistency Model
- Cloud Types
  - How to work with eventually consistent state

PART I: TWO FOR THE PRICE OF ONE

Motivation: Compute-Mutate Loops

```
while (true) {
  compute();
  mutate();
}
```

- Deterministic
- No I/O
- Potentially Parallel

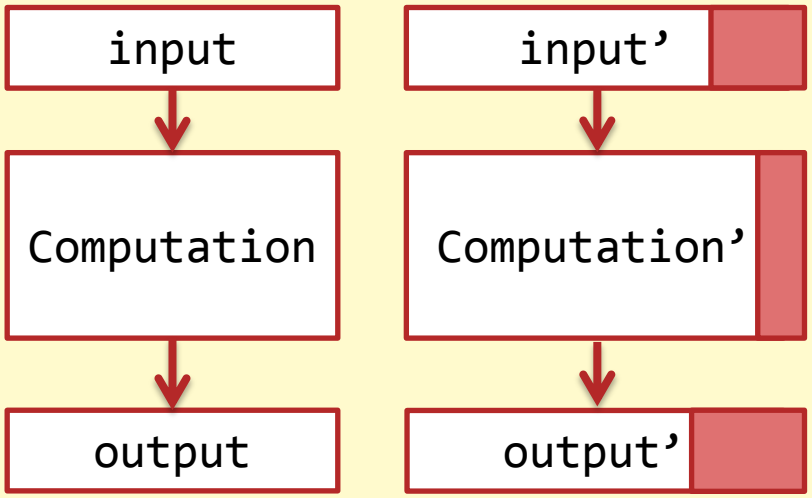
- Nondeterministic
- I/O

- Common pattern in applications (e.g. browser, games, compilers, spreadsheets, editors, forms, simulations)
- Goal: **perform better (faster, less power)**

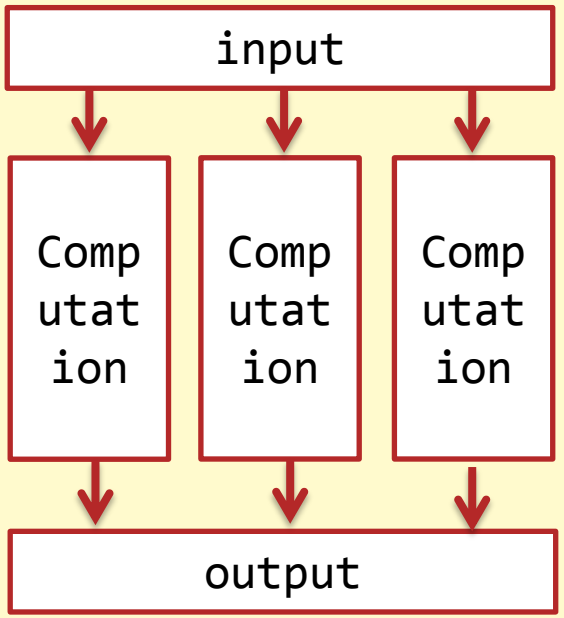
Compute-Mutate Loop Examples

	Compute - Deterministic - May be parallel - No I/O	**Mutate** - Nondeterministic - I/O
Browser	CSS Layout	DOM changes
Ray-Tracer	Render Picture	Change objects
Morph	Compute Blend	Change blended pictures
Compiler	Compile project	Edit source files
Spellcheck	Check words	Change document

Incremental Computation



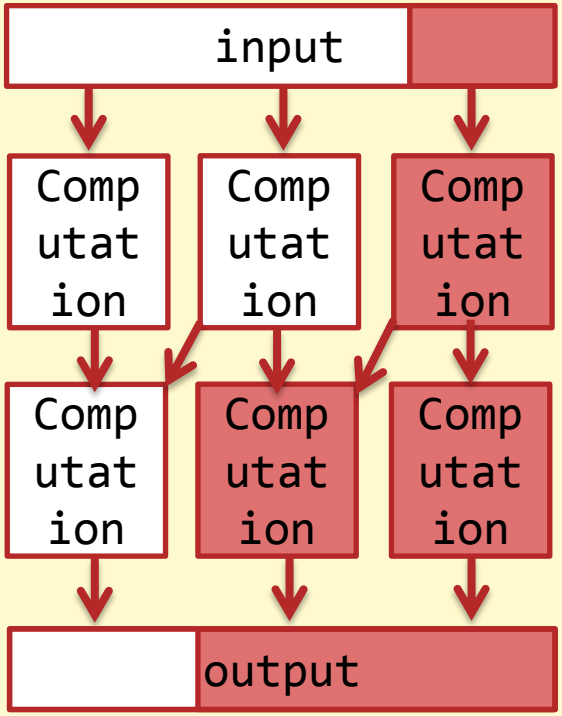
Parallel Computation



- Which one would you choose?
- Do we have to choose?

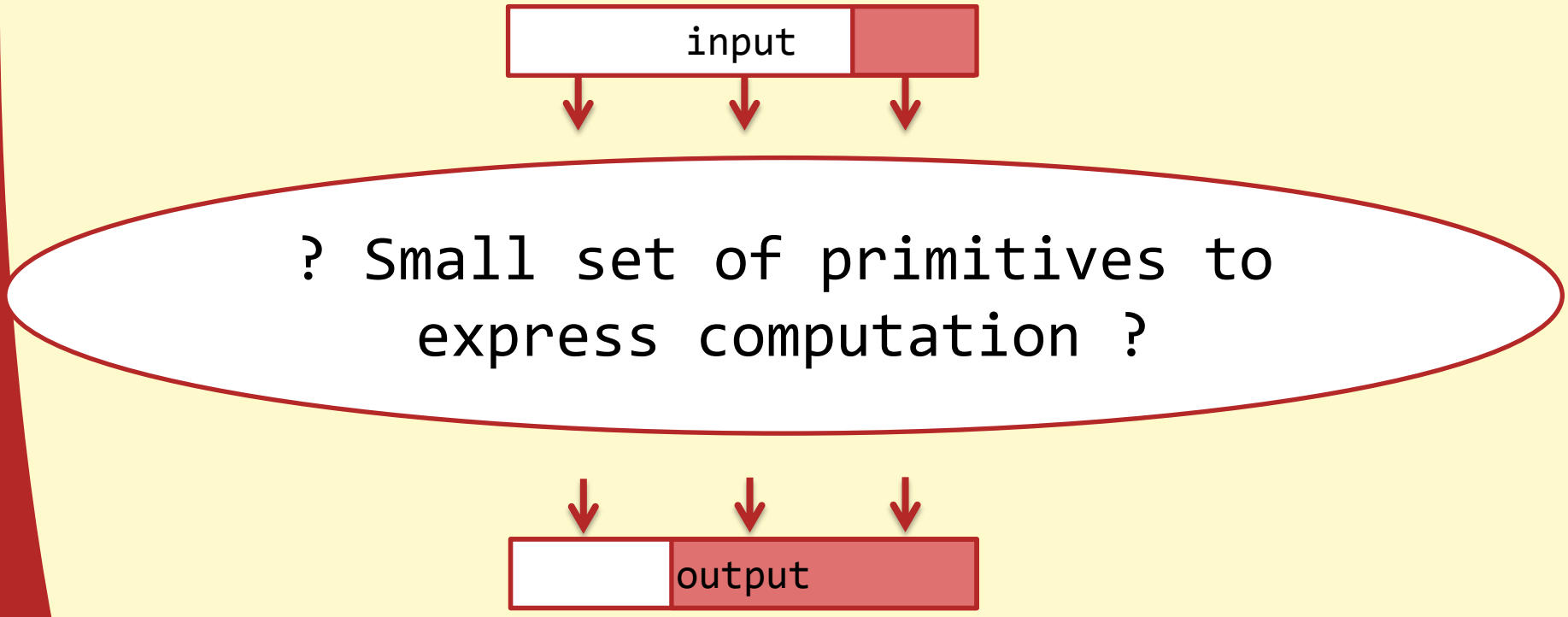
Incremental + Parallel Computation

Self-Adjusting Computation:
Dynamic Dependence Graph



Work Stealing:
Dynamic Task Graph

Two for the Price of One



- Wanted: Programming Model for *Parallel & Incremental* Computation

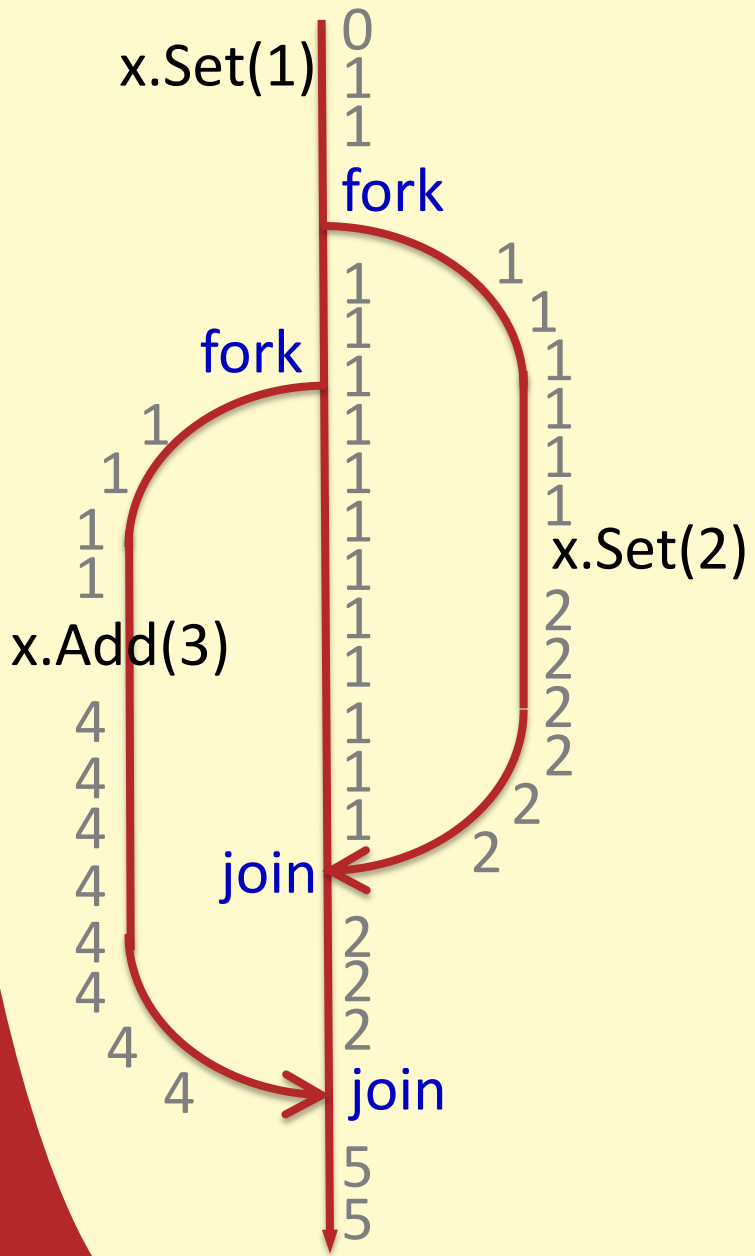
Our Primitives: fork, join, record, repeat

- Start with Deterministic Parallel Programming
  - Concurrent Revisions Model
  - fork and join Revisions (= Isolated Tasks)
  - Declare shared data and operations on it

- Add Primitives for record and repeat
  - c = record { f(); } for some computation f()
  - repeat c is equivalent to calling f() again, but faster
  - the compute-mutate loop does
    record – mutate – repeat – mutate – repeat …

Concurrent Revisions Model

[OOPSLA '10]
[ESOP '11]
[WoDet '11]



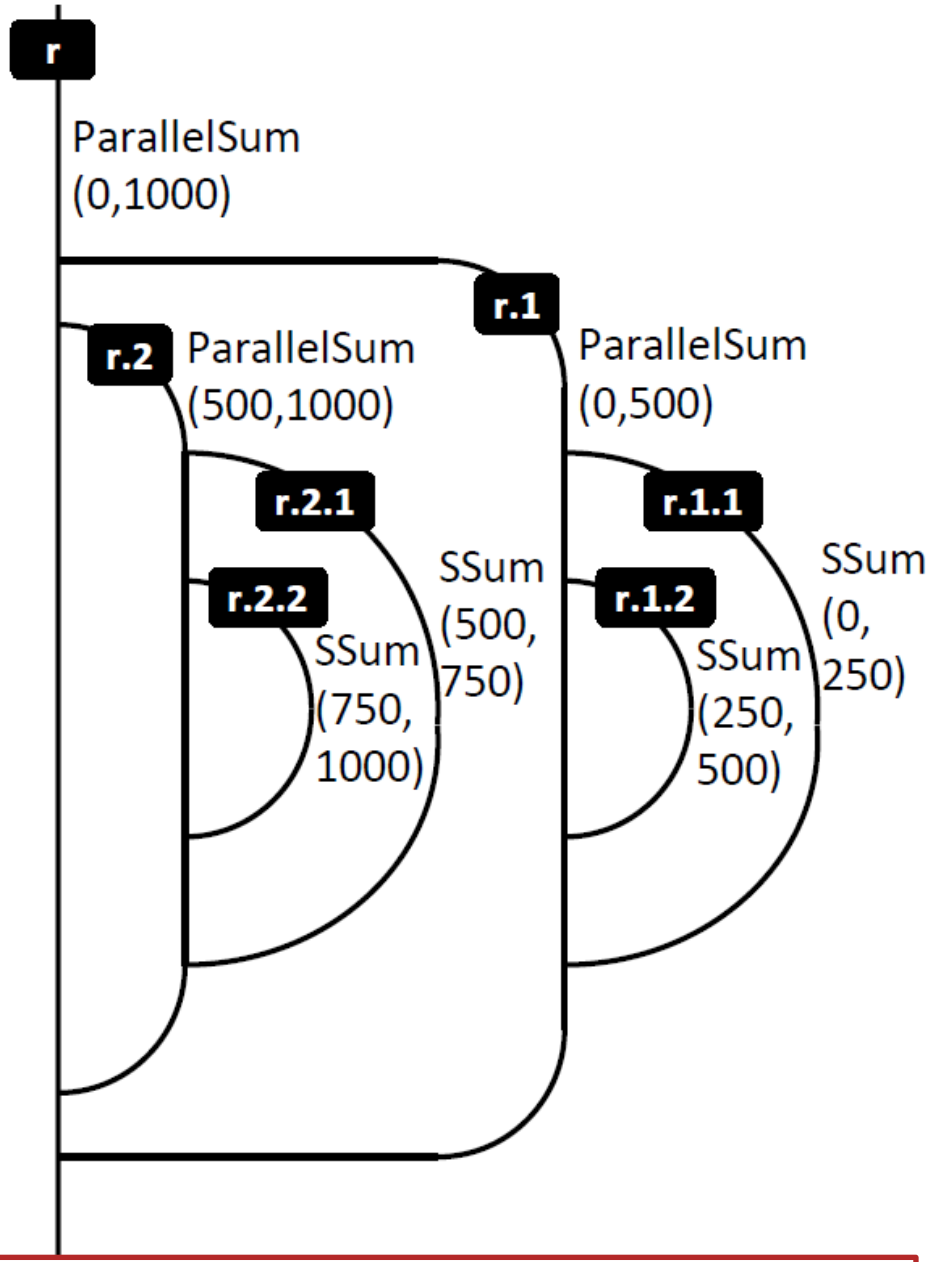
- Deterministic Parallelism by fork and join (creates concurrent tasks called *revisions*)
- Revisions are isolated
  - fork copies all state
  - join replays updates
- Use optimized types (copy on write, merge functions)

Example: Parallel Sum

```
int ParallelSum(Versioned<int>[] a, int from, int to) {
   if (to-from <= threshold)
      return SequentialSum(a, from, to);
   else {
      Versioned<int> sum := 0;
      Revision r1 := fork {
         sum.Add(ParallelSum(a, from, (from + to)/2));
      }
      Revision r2 := fork {
         sum.Add(ParallelSum(a, (from + to)/2, to));
      }
      join r2;
      join r1;
      return sum;
   }
}
```

Example

Step 1: Record




```
record { total := ParallelSum(a, 0, 1000); }
```

Example (Cont'd)

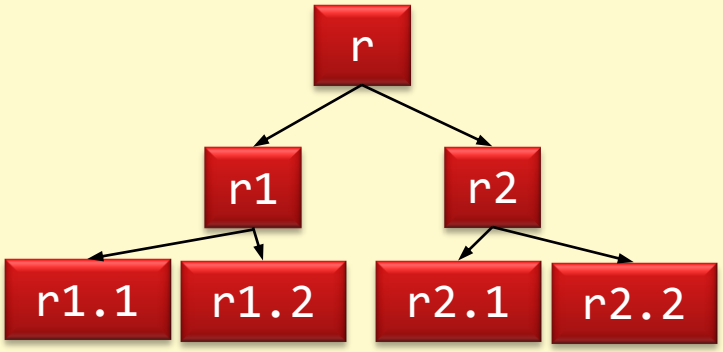
Step 2: Mutate

```
a[333] := a[333] + 1 ;
```

Step 3: Repeat

```
repeat ;
```

How does it work?



- On Record
  - Create ordered tree of summaries (summary=revision)
  - Revisions-Library already stores effects of revisions
    - Can keep them around to "reexecute" = join again
  - Can track dependencies
    - Record dependencies
    - Invalidate summaries
    - At time of fork, know if valid

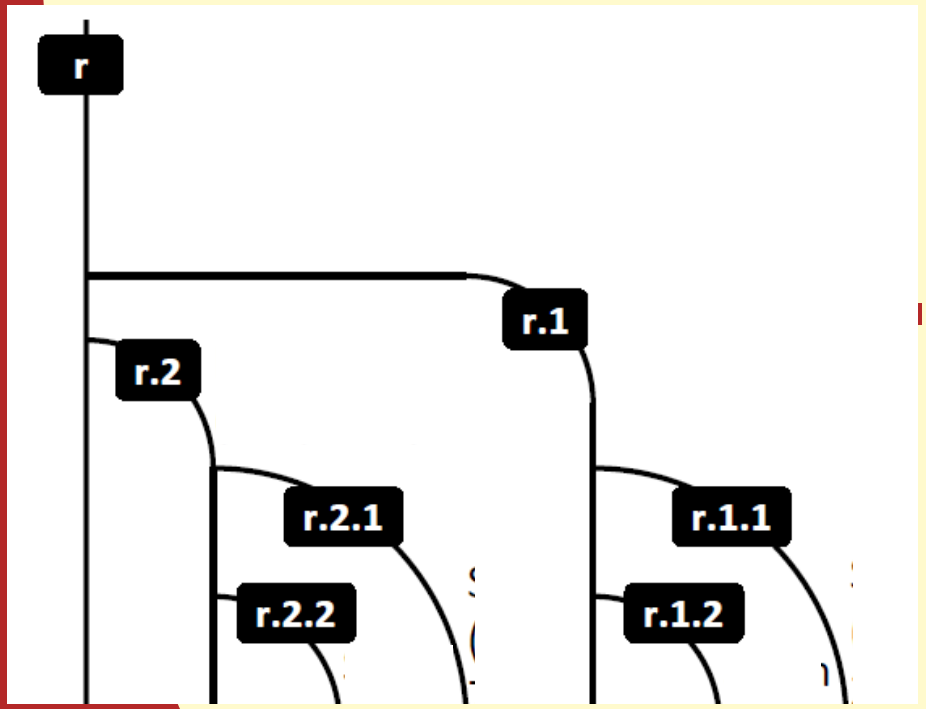
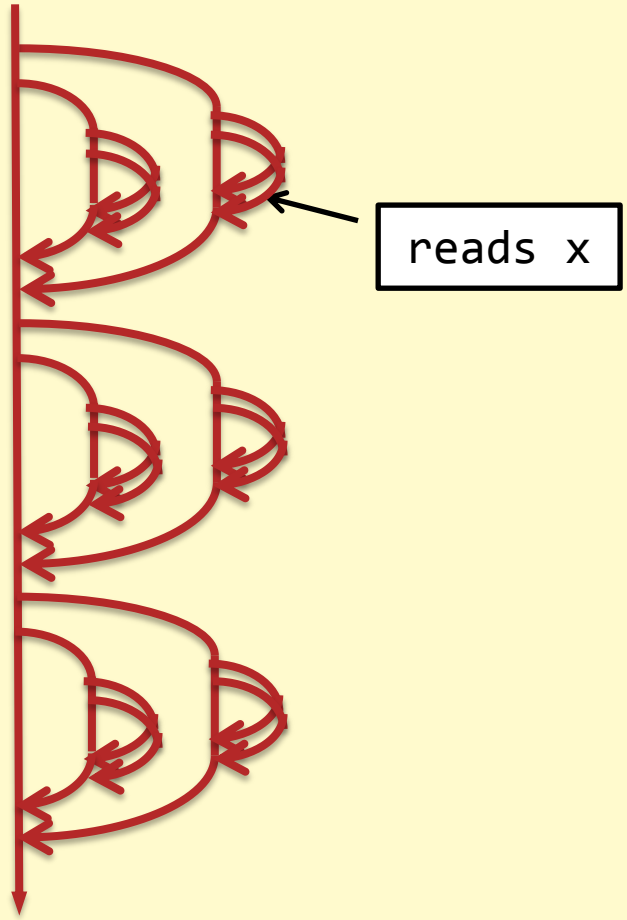


Illustration Example

- Consider computation shaped like this (e.g. our CSS layout alg. with 3 passes)

```
c = record {
  pass1();
  pass2();
  pass3();
}
```



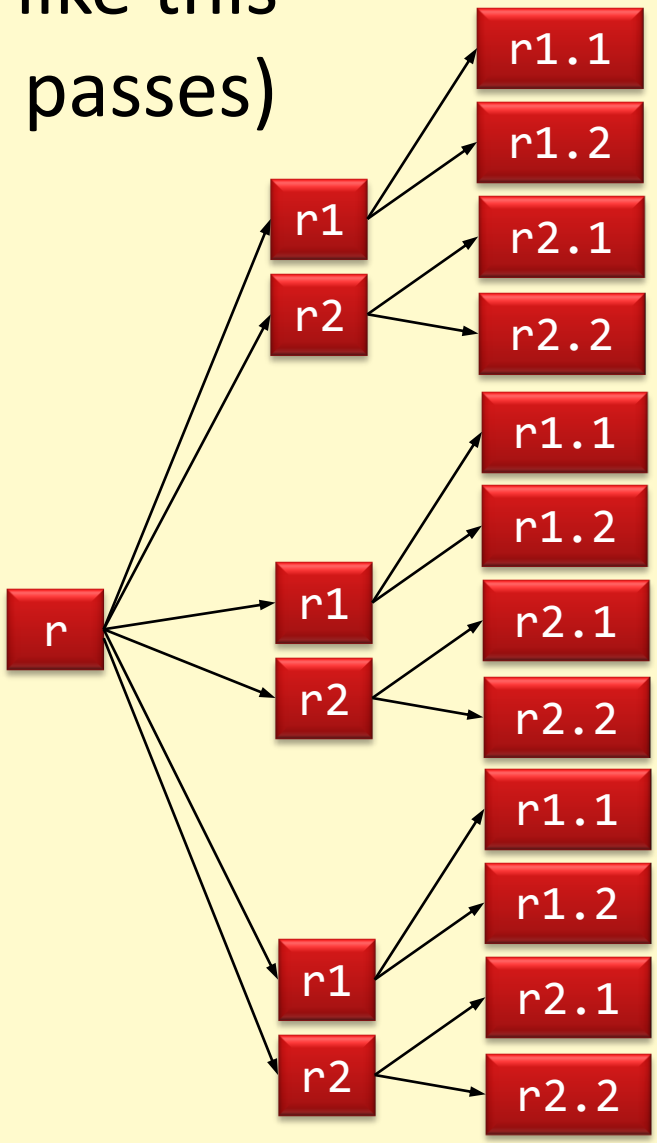
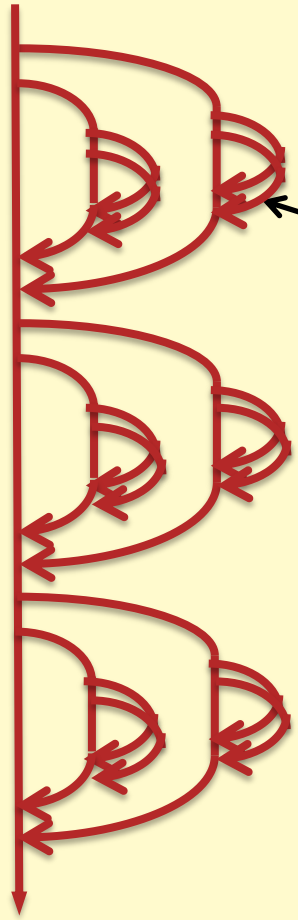


Illustration Example

- Consider computation shaped like this (e.g. our CSS layout alg. with 3 passes)

```
c = record {
  pass1();
  pass2();
  pass3();
}

x = 100;
```



Dep. on x

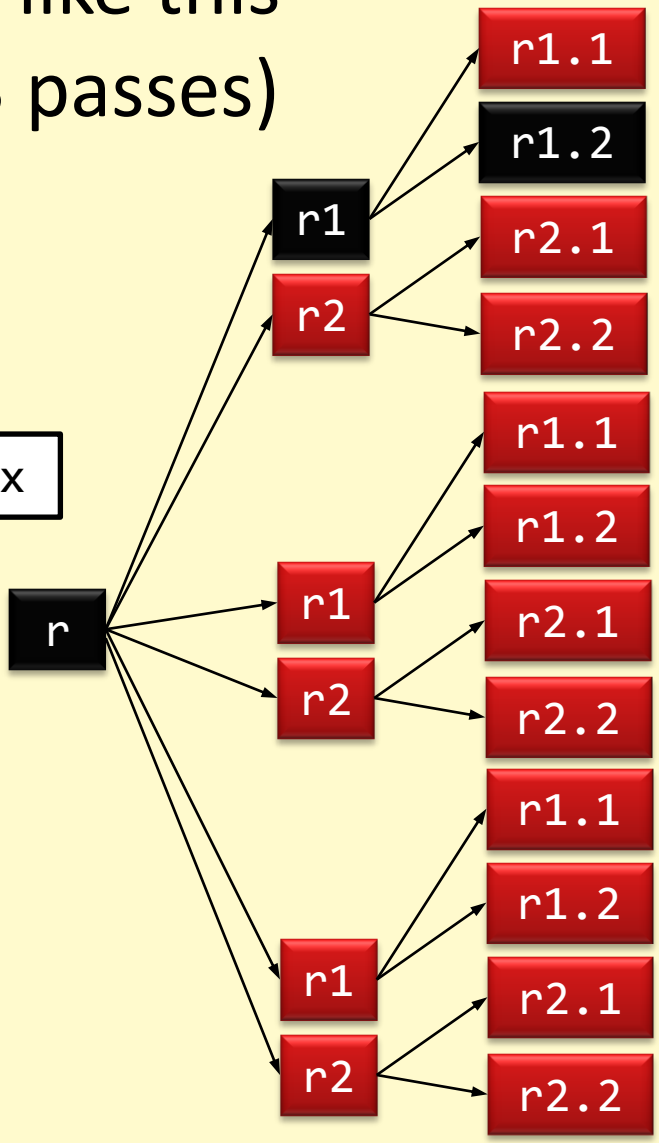


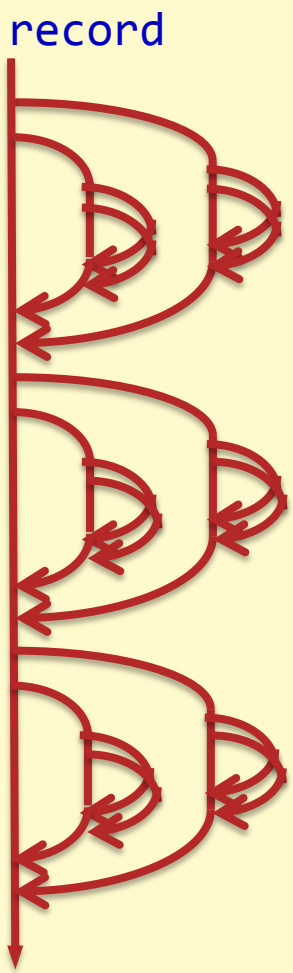
Illustration Example

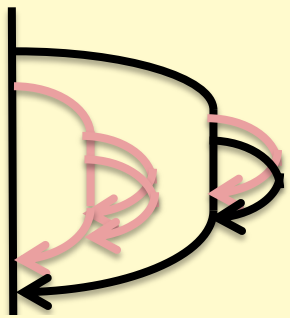
- Consider computation shaped like this (e.g. our CSS layout alg. with 3 passes)

```
c = record {
  pass1();
  pass2();
  pass3();
}

x = 100;

repeat c;
```





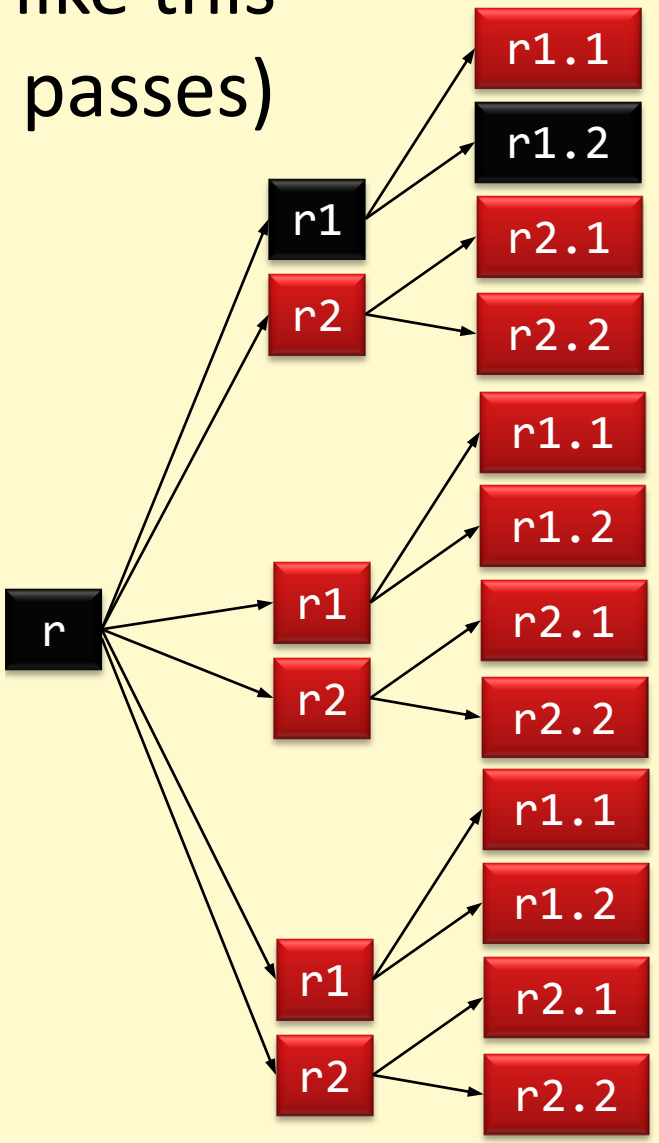


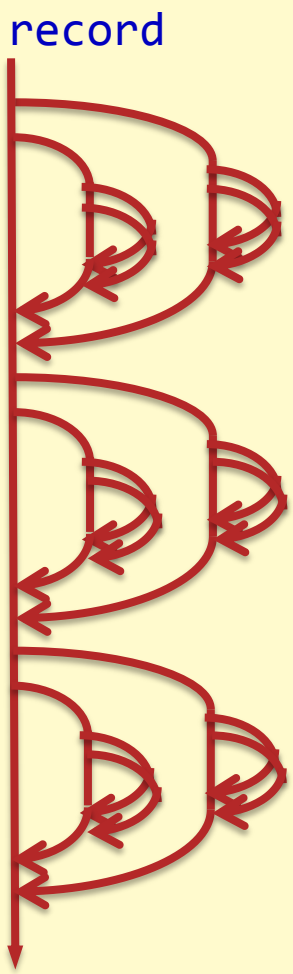
Illustration Example

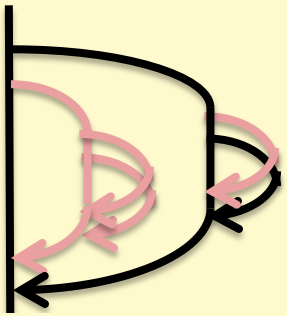
- Consider computation shaped like this (e.g. our CSS layout alg. with 3 passes)

```
c = record {
  pass1();
  pass2();
  pass3();
}

x = 100;

repeat c;
```





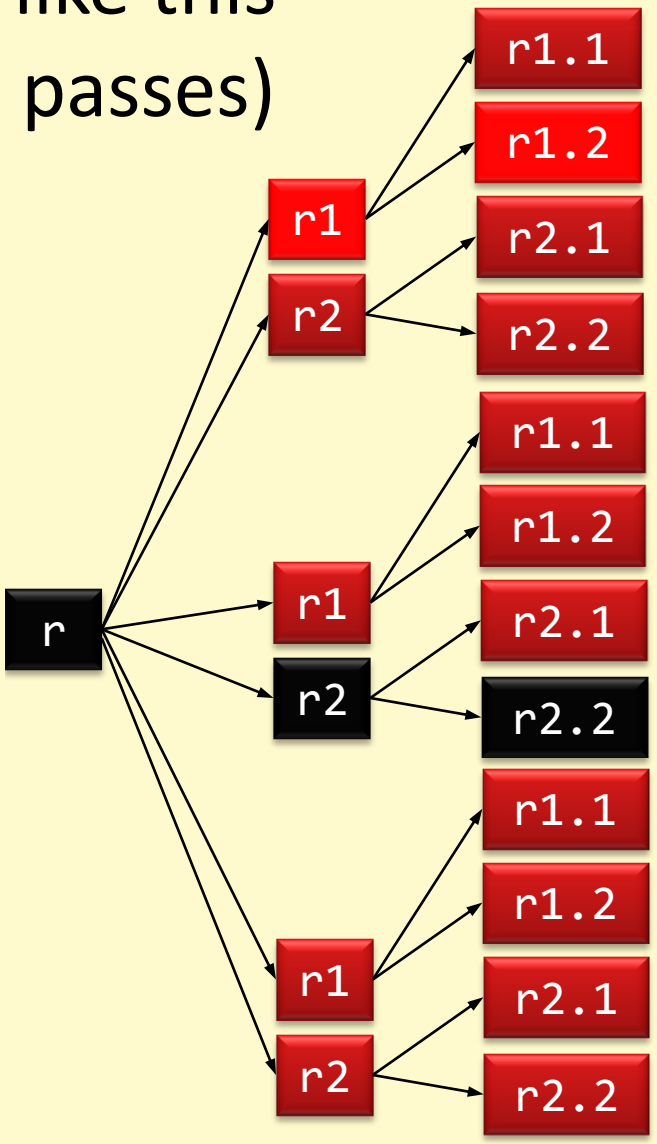


Illustration Example

- Consider computation shaped like this (e.g. our CSS layout alg. with 3 passes)

```
c = record {
  pass1();
  pass2();
  pass3();
}

x = 100;

repeat c;
```

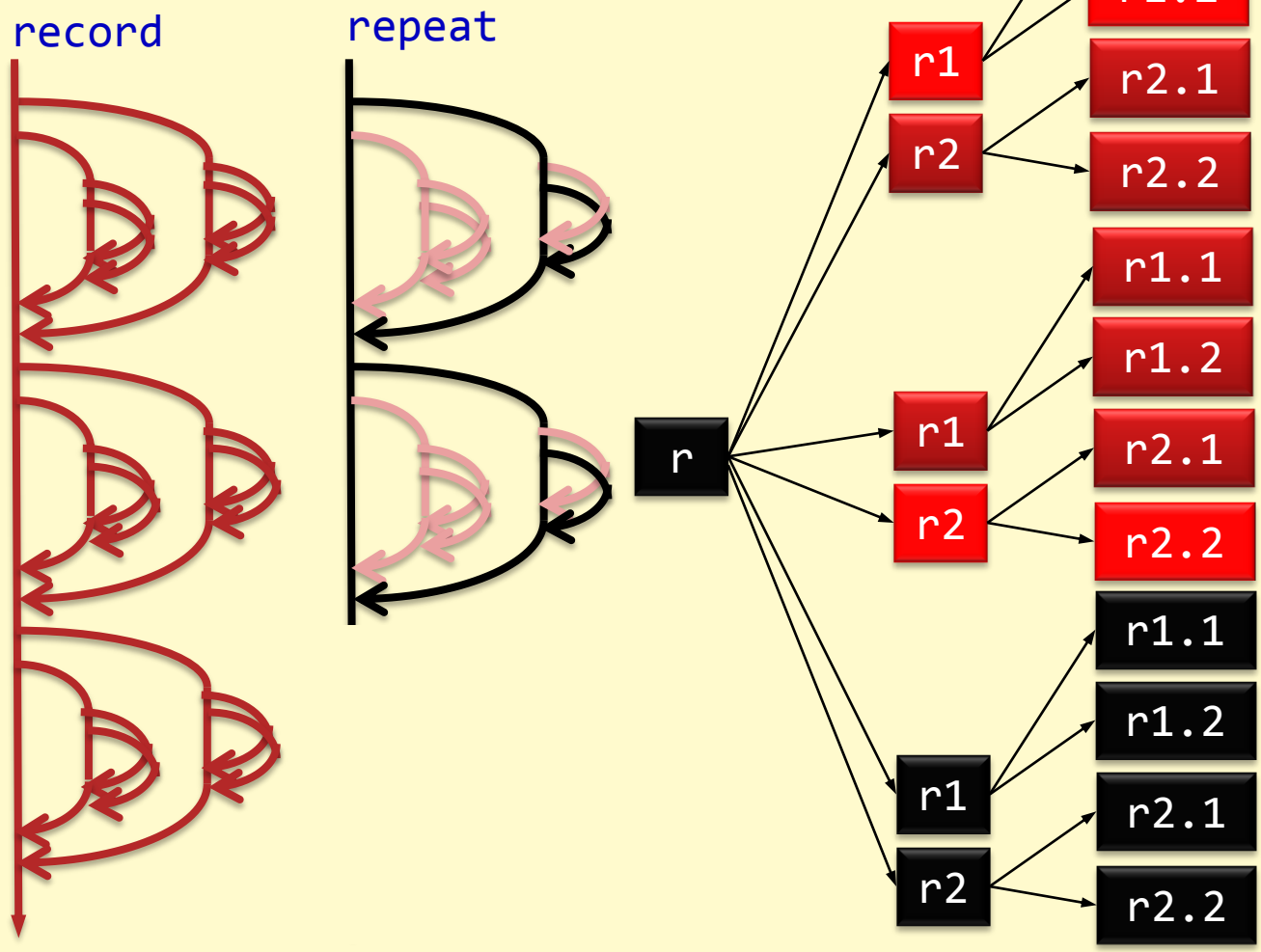


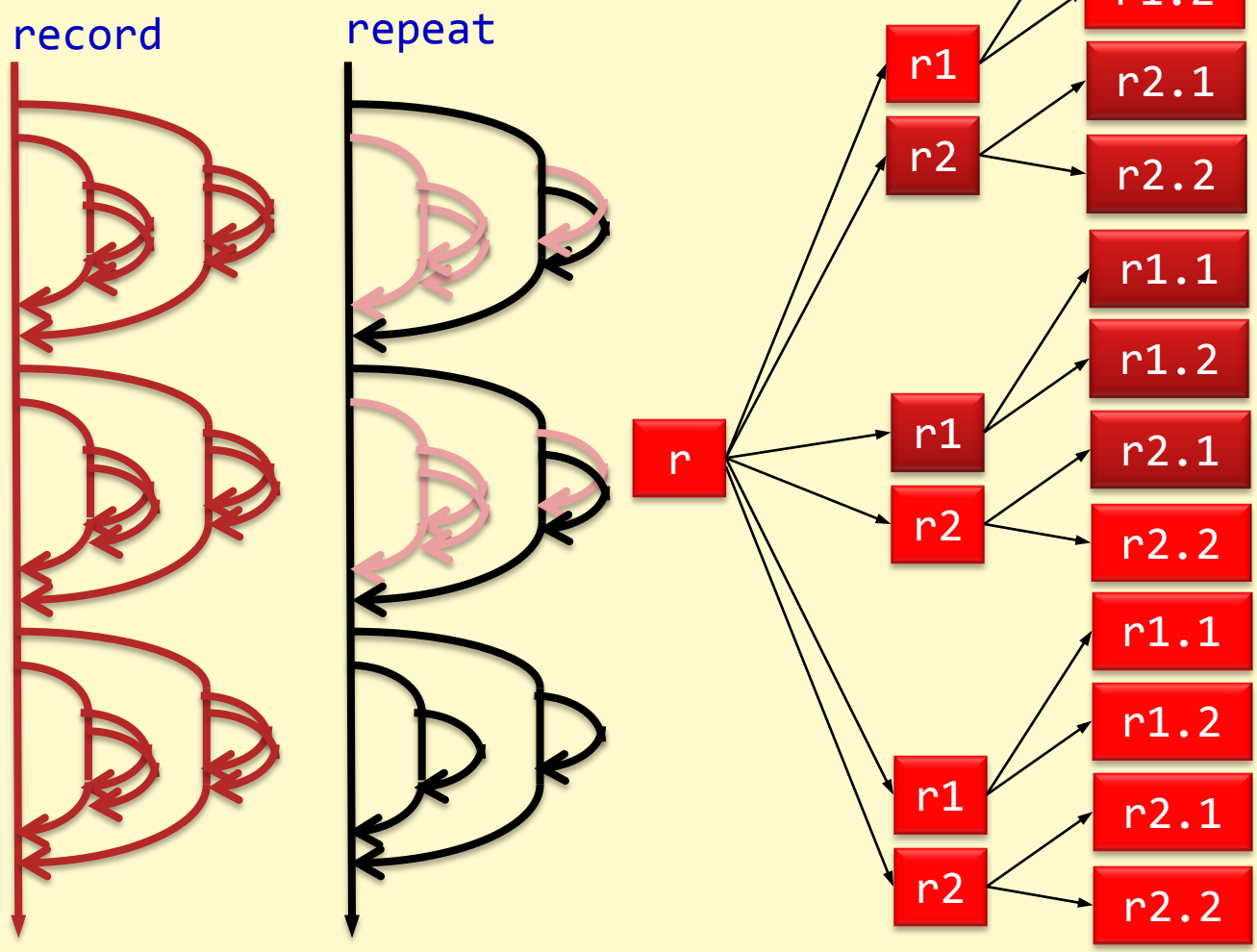
Illustration Example

- Consider computation shaped like this (e.g. our CSS layout alg. with 3 passes)

```
c = record {
  pass1();
  pass2();
  pass3();
}

x = 100;

repeat c;
```



Invalidating Summaries is Tricky

- May depend on external input
  - Invalidate between compute and repeat
- May depend on write by some other summary
  - Invalidate if write changes
  - Invalidate if write disappears
- Dependencies can change
  - Invalidate if new write appears between old write and read
- And all of this is concurrent

Who has to think about what

Our runtime

- Detects nature of dependencies
  - Dynamic tracking of reads and writes
- Records and replays effects of revisions
- Schedules revisions in parallel on multiprocessor (based on TPL, a work-stealing scheduler)

The programmer

- Explicitly structures the computation (fork, join)
- Declares data that participates in
  - Concurrent accesses
  - Dependencies
- Thinks about performance
  - Revision granularity
  - Applies Marker Optimization

Optimizations by Programmer

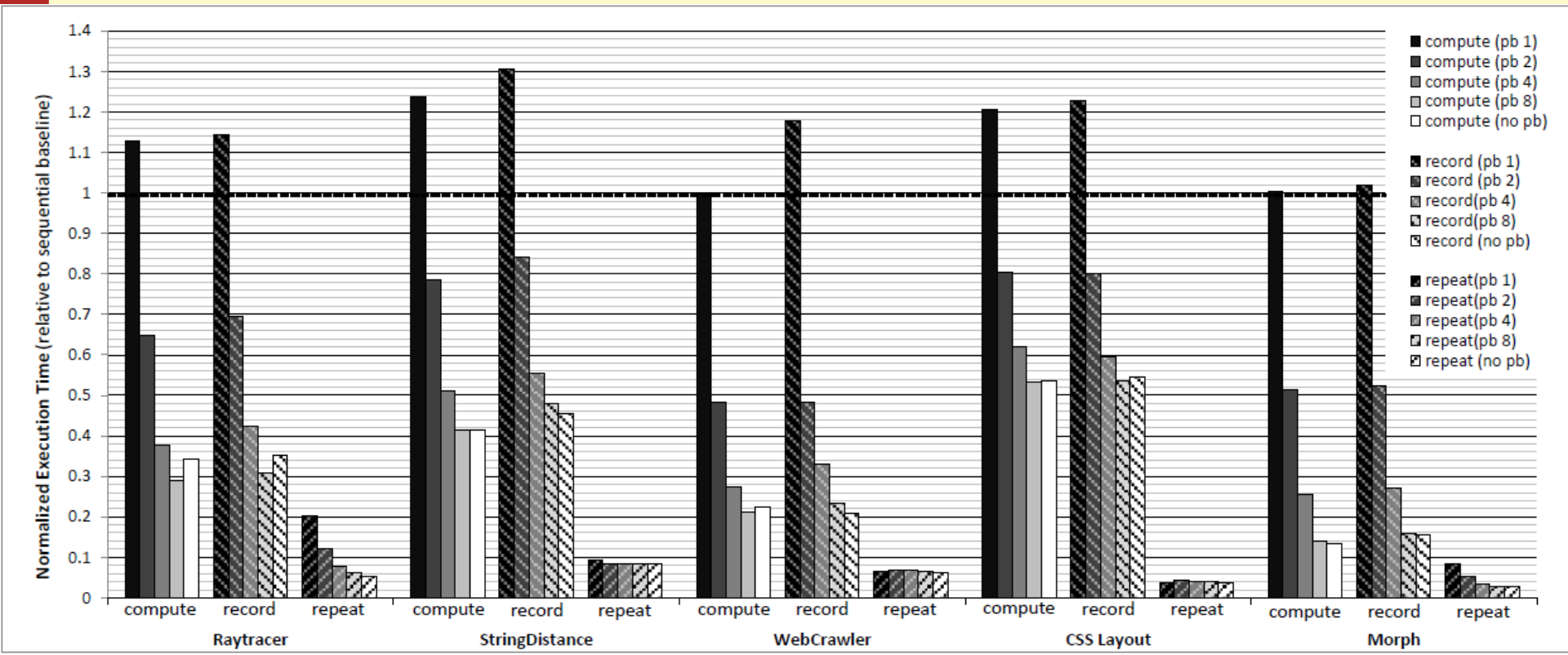
- Granularity Control
  - Problem: too much overhead if revisions contain not enough work
  - Solution: Use typical techniques (e.g. recursion threshold) to keep revisions large enough
- Markers
  - Problem: too much overhead if tracking individual memory locations (e.g. bytes of a picture)
  - Solution: Use *marker* object to represent a group of locations (e.g. a tile in picture)

Results on 5 Benchmarks

Benchmark	lines	baseline	parallel record		parallel repeat	
Raytracer	1409	2.140 s	751 ms	**2.8x**	116 ms	**18x**
StringDistance	618	268 ms	122 ms	**2.2x**	22.5 ms	**12x**
WebCrawler	525	16.3 s	3.42 s	**4.8x**	1.03 s	**16x**
CSS Layout	2674	185 ms	101 ms	**1.8x**	6.88 ms	**27x**
Morph	2238	143 s	22.2 s	**6.4x**	3.86 s	**37x**

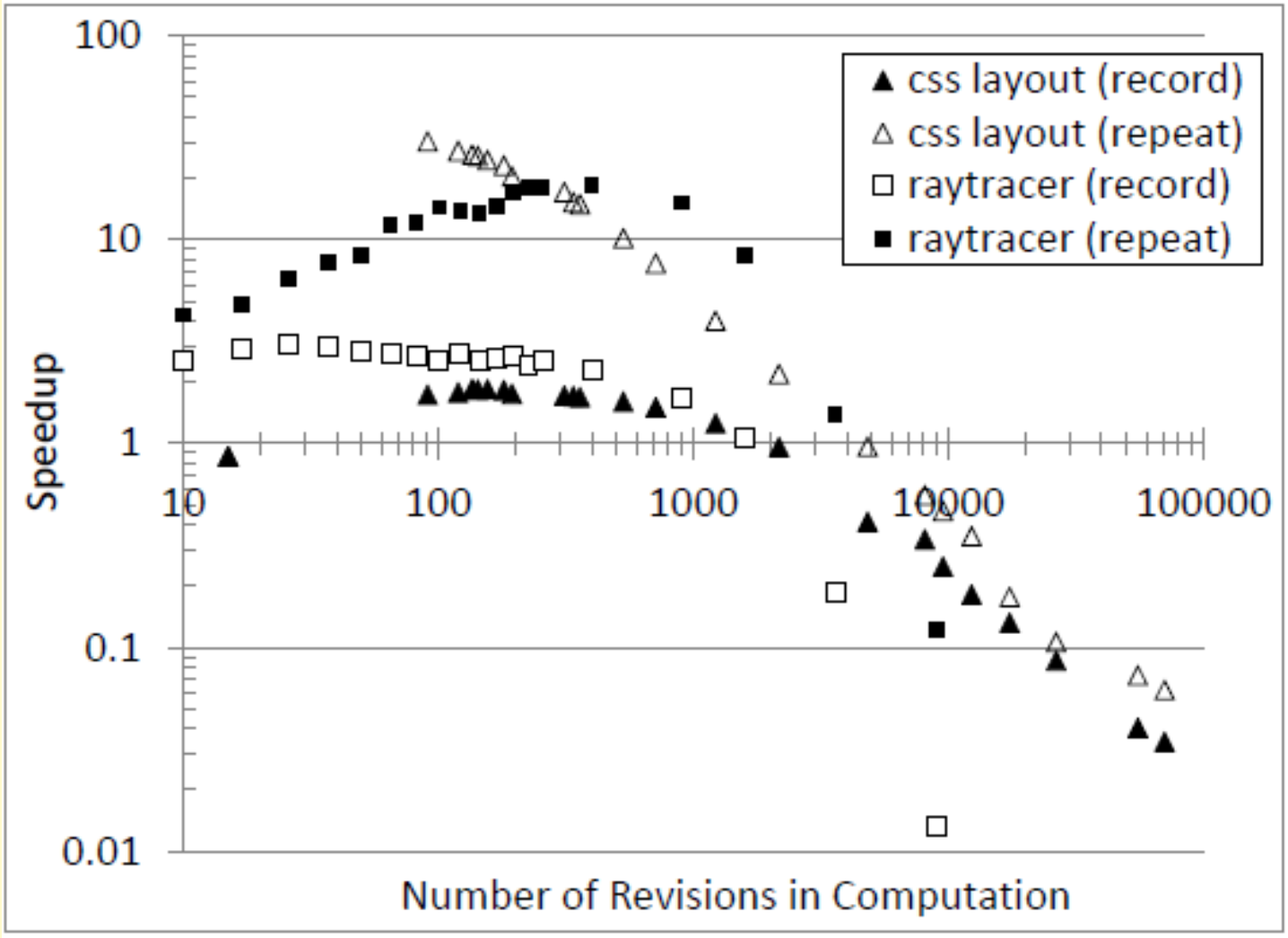
- On 8 cores,
  - recording is still faster than baseline (1.8x – 6.4x)
  - Repeat after small change is significantly faster than baseline (12x – 37x)
  - Repeat after large change is same as record

What part does parallelism play?



- Without parallelism, **record** is up to 31% slower than baseline
- Without parallelism, **repeat** after small change is still 4.9x – 24x faster than baseline

Controlling Task Granularity is Important



PART II : EVENTUAL CONSISTENCY

Overview

- Motivation
  Why eventual consistency
- How we can understand and build it
  Operational consistency model
- Formal Foundation
  Axiomatic consistency model

Post-PC World: Apps & Cloud

The CAP theorem [Brewer'00,Lynch&Gilbert'02]

- A distributed system cannot have:
  - Consistency
    - all nodes see the same data at the same time
  - Availability
    - Every request receives a response about whether it was successful or failed
  - Partition Tolerance
    - the system continues to operate despite arbitrary message loss

Where to compromise?

- Strong Consistency, Brittle Availability
  - Maintain illusion of single master copy
  - Cannot commit updates without server roundtrip
  - Changes are globally visible at time of commit
- Strong Availability, Eventual Consistency
  - Keep replicas on each client
  - Can commit updates locally without server connection
  - Changes are immediately visible locally, and eventually propagated to all other replicas

How Much Consistency do we Need?

- Strong Consistency
  - Updates are conditional on very latest state
  - Examples: bank accounts, seat reservations, …
  - See: classic OLTP (online transaction processing)

- Eventual Consistency
  - Updates are not conditional on very latest state
  - Examples: Ratings, Shopping Cart, Comments, Settings, Chat, Grocery List, Playlist, Calendar, Mailbox, Contacts …
  - But: how to program this?

Eventual Consistency

- What does it really mean?
- Need a consistency model!
- At intersection of various communities
  - Databases (relational storage, queries, …)
  - Multiprocessors (memory models, consistency…)
  - Distributed Systems (fault tolerance, availability)
  - Web programming (client apps, web services)

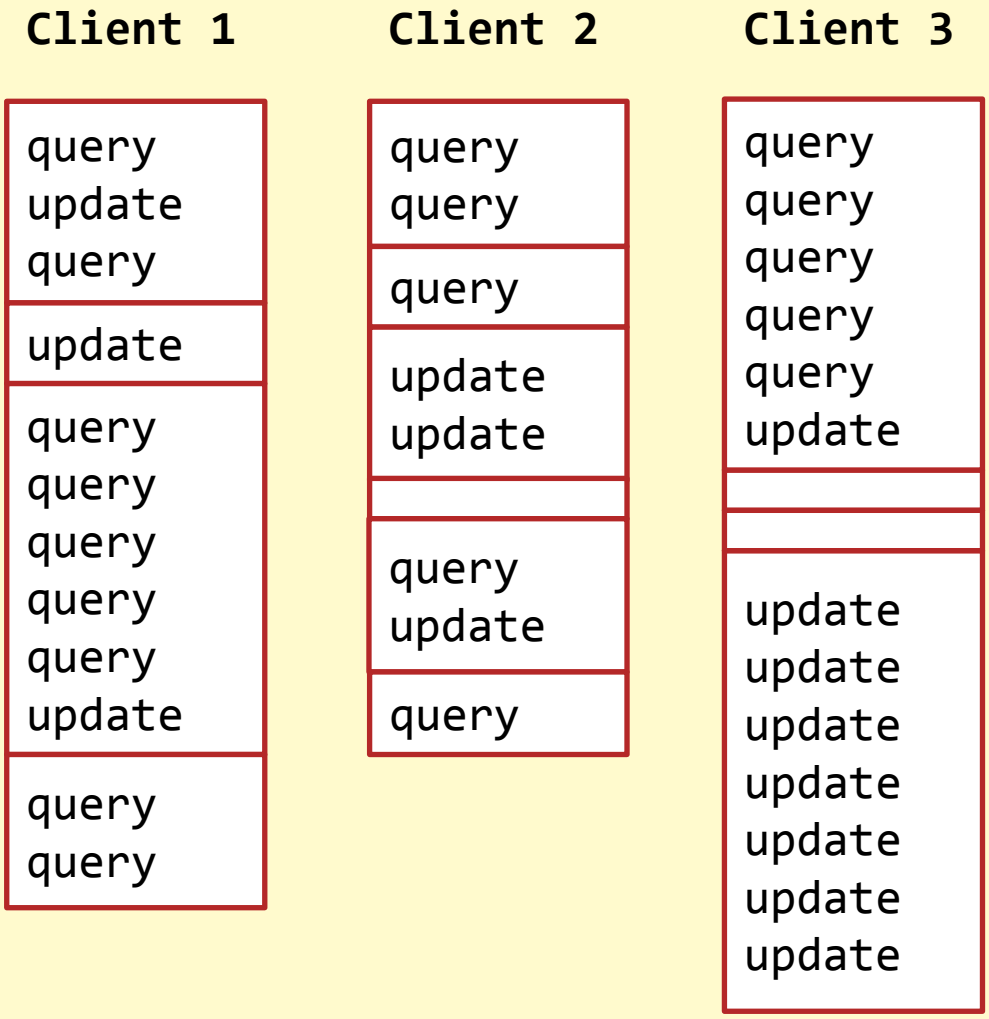
Consistency Models

- Fall mostly into 2 categories
  - **Operational**
    define valid executions as runs of [illegible]achine
  - **Axiomatic**
    define executions valid[illegible]
    subject to certain con[illegible]
- We do both in pape[illegible]
  - First, we define an abst[illegible]
    - Generalization of sequential [illegible]stency
  - Then, we define an operational model
    - More specific and intuitive than the abstract model
    - We prove that it implements the abstract model

We will focus almost exclusively on the **operational** model in this talk.

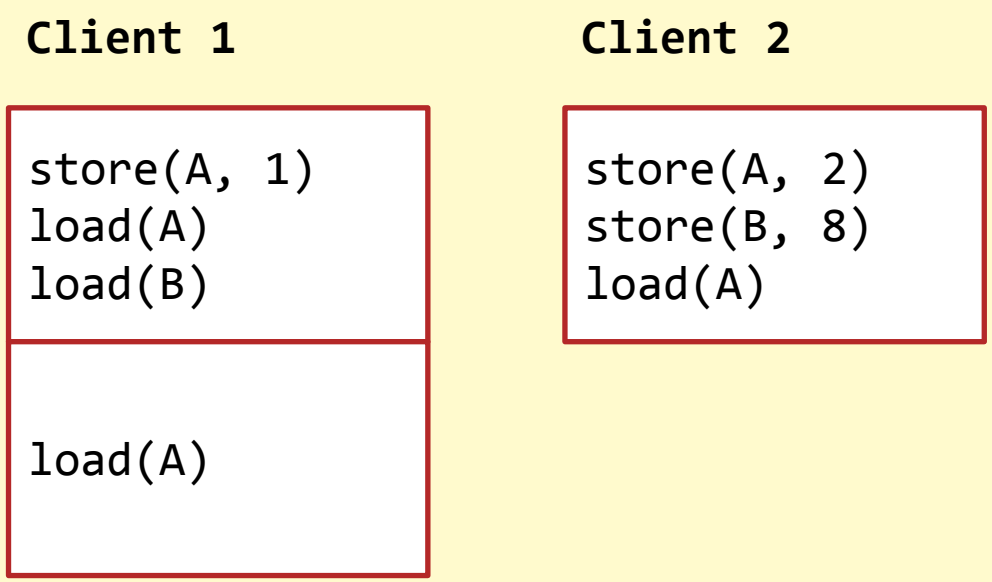
Abstract System Model

- Clients emit streams of transactions
- Each transaction contains a sequence of operations
- Operations are queries/updates of some data



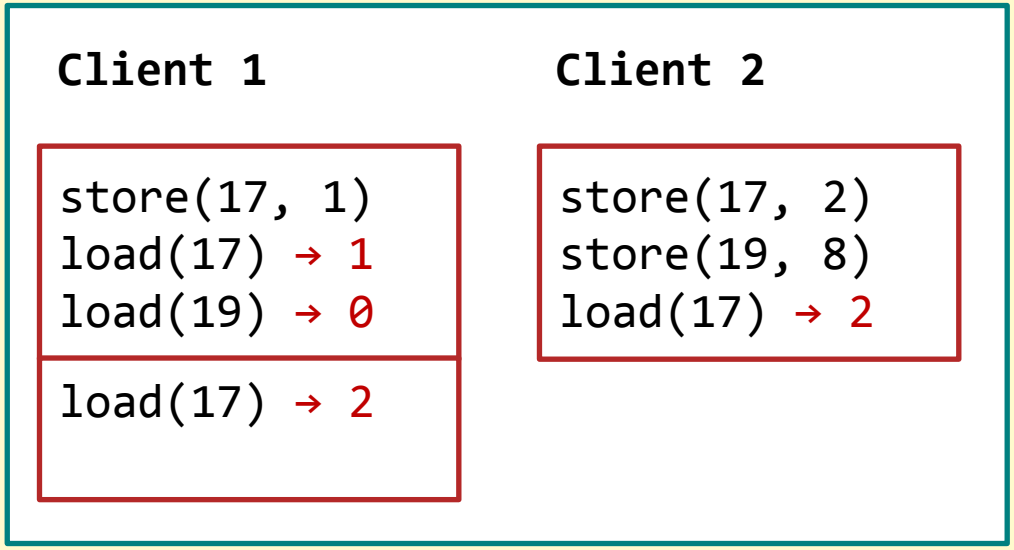
A familiar example of data: memory

- A random access memory interface (64 bits)
  - Queries = {load(a) | a ∈ Addresses}
  - Updates = {store(a,v) | a ∈ Addresses, v ∈ Values}



Terminology

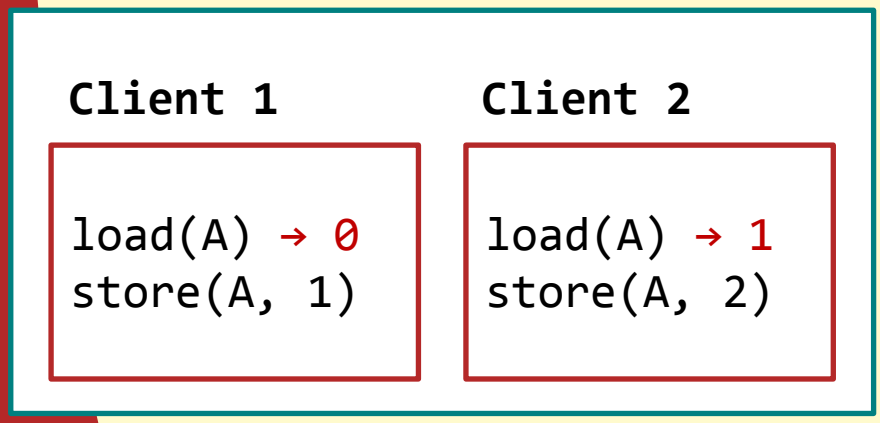
- History = collection of transaction sequences (one per client) including query results
  Example:

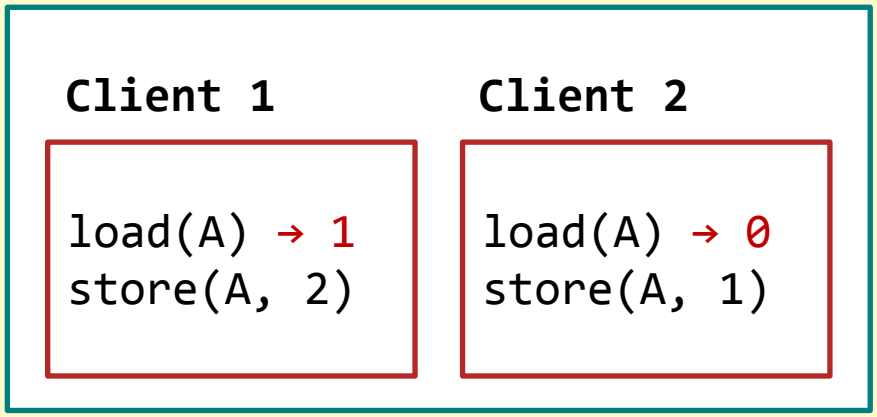


- Consistency model = set of valid histories

Example

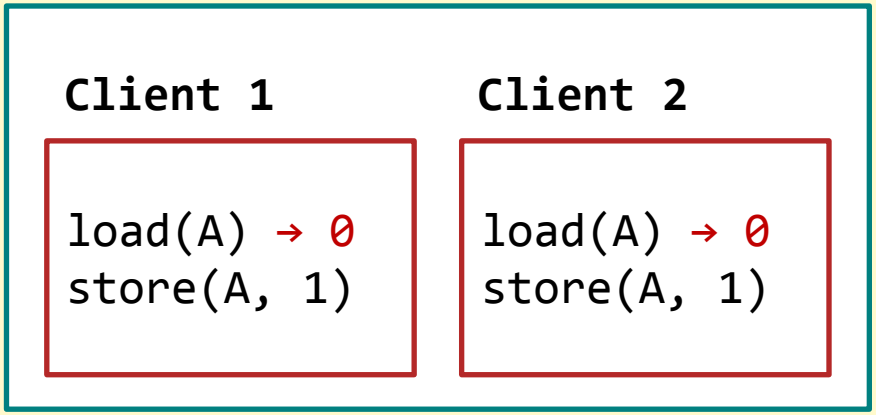
- Consider two clients performing a transaction to increment location A (initially 0)
- Under strong consistency, we would expect one of the following two histories





Example

- Now suppose clients are disconnected, yet still want to commit transactions.
  - Transactions cannot be serializable (CAP theorem).
  - Code does not work as intended.

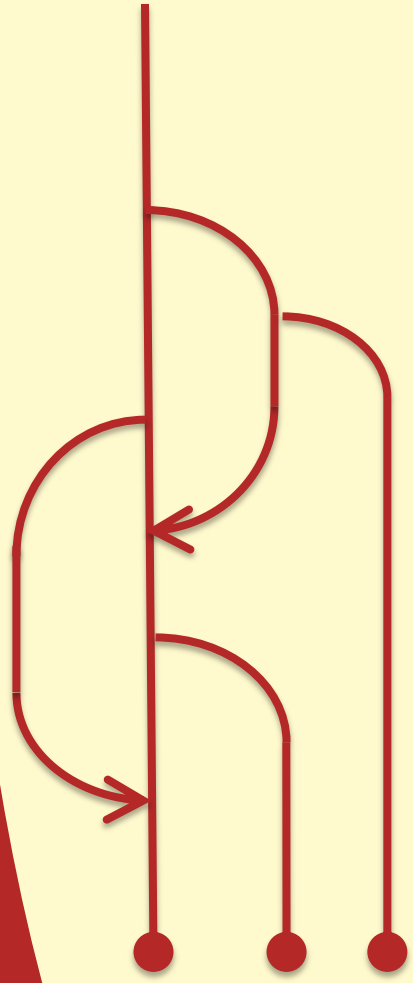


How can we understand such histories?

How can we write correct programs?

Concurrent Revisions

[OOPSLA'10] [WoDet'11] [ESOP'11] [OOSPLA'11] [ESOP'12] [ECOOP'12]

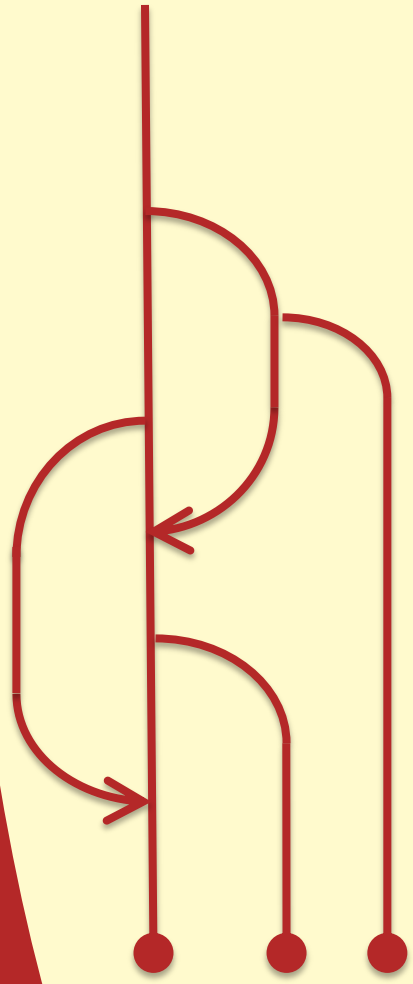


1. Model state as a *revision diagram*
   - *Fork:* creates revision (snapshot)
   - Queries/Updates target specific revision
   - *Join:* apply updates to joining revision
2. Raise data abstraction level
   - Record operations, not just states
   - At join, replay all updates
   - Use specially designed data types

Main Ingredient #1:

REVISION DIAGRAMS

Revision Diagrams



- Directed Graphs
  - One root
  - One or more terminals
- Operational construction rules
  - Fork
    - One terminal -> two terminals
  - Update/Query
    - Append to terminal
  - Join
    - Two terminals -> one terminal
    - Subject to join condition

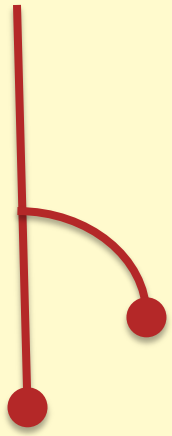
Revision Diagrams

- Directed Graphs
  - One root
  - One or more terminals
- Operational construction rules
  - Fork
    - One terminal -> two terminals
  - Update/Query
    - Append to terminal
  - Join
    - Two terminals -> one terminal
    - Subject to join condition

Revision Diagrams

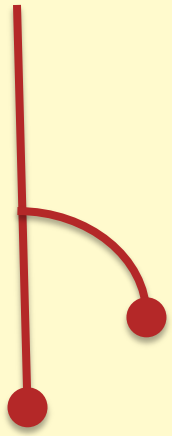
- Directed Graphs
  - One root
  - One or more terminals
- Operational construction rules
  - Fork
    - One terminal -> two terminals
  - Update/Query
    - Append to terminal
  - Join
    - Two terminals -> one terminal
    - Subject to join condition

Revision Diagrams



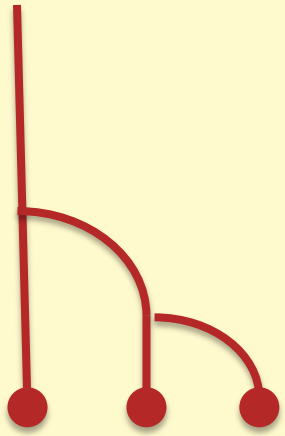
- Directed Graphs
  - One root
  - One or more terminals
- Operational construction rules
  - Fork
    - One terminal -> two terminals
  - Update/Query
    - Append to terminal
  - Join
    - Two terminals -> one terminal
    - Subject to join condition

Revision Diagrams



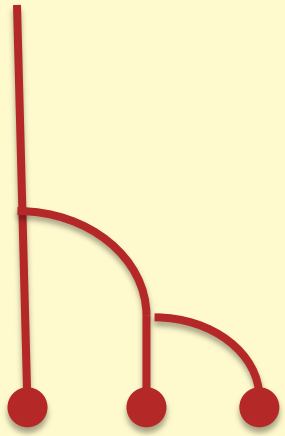
- Directed Graphs
  - One root
  - One or more terminals
- Operational construction rules
  - Fork
    - One terminal -> two terminals
  - Update/Query
    - Append to terminal
  - Join
    - Two terminals -> one terminal
    - Subject to join condition

Revision Diagrams



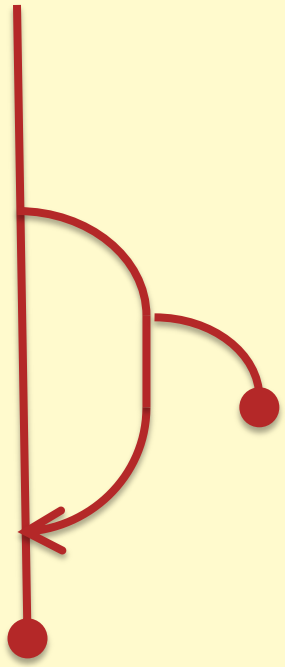
- Directed Graphs
  - One root
  - One or more terminals
- Operational construction rules
  - Fork
    - One terminal -> two terminals
  - Update/Query
    - Append to terminal
  - Join
    - Two terminals -> one terminal
    - Subject to join condition

Revision Diagrams



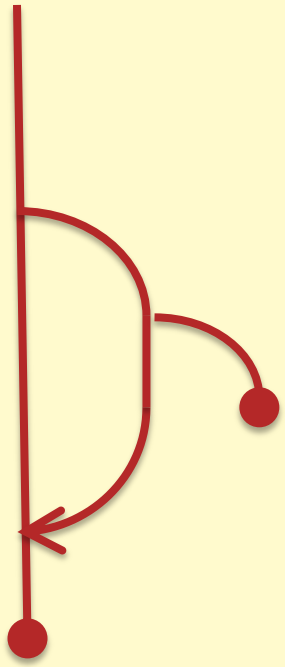
- Directed Graphs
  - One root
  - One or more terminals
- Operational construction rules
  - Fork
    - One terminal -> two terminals
  - Update/Query
    - Append to terminal
  - Join
    - Two terminals -> one terminal
    - Subject to join condition

Revision Diagrams



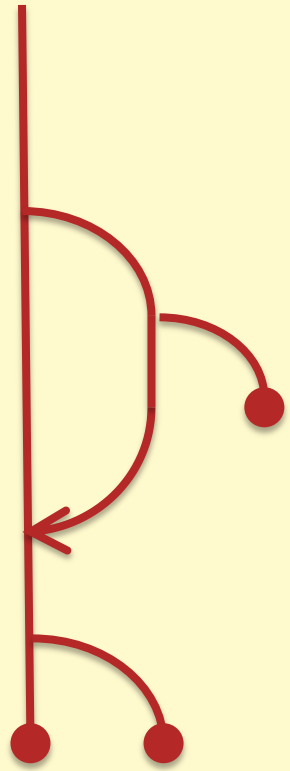
- Directed Graphs
  - One root
  - One or more terminals
- Operational construction rules
  - Fork
    - One terminal -> two terminals
  - Update/Query
    - Append to terminal
  - Join
    - Two terminals -> one terminal
    - Subject to join condition

Revision Diagrams



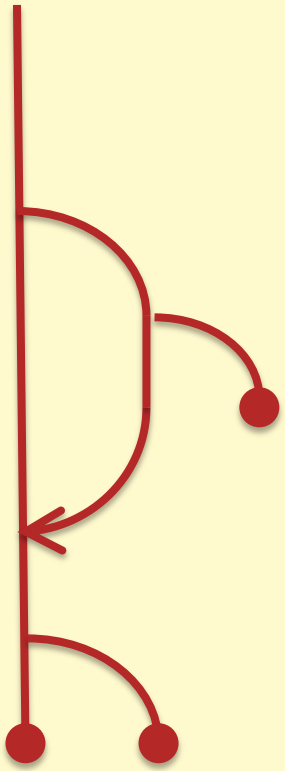
- Directed Graphs
  - One root
  - One or more terminals
- Operational construction rules
  - Fork
    - One terminal -> two terminals
  - Update/Query
    - Append to terminal
  - Join
    - Two terminals -> one terminal
    - Subject to join condition

Revision Diagrams



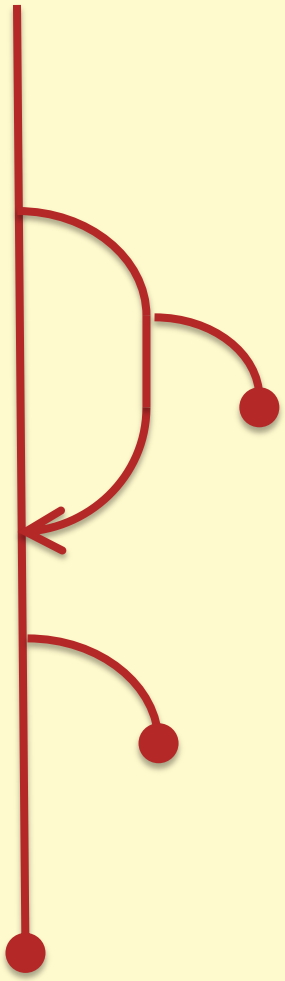
- Directed Graphs
  - One root
  - One or more terminals
- Operational construction rules
  - Fork
    - One terminal -> two terminals
  - Update/Query
    - Append to terminal
  - Join
    - Two terminals -> one terminal
    - Subject to join condition

Revision Diagrams



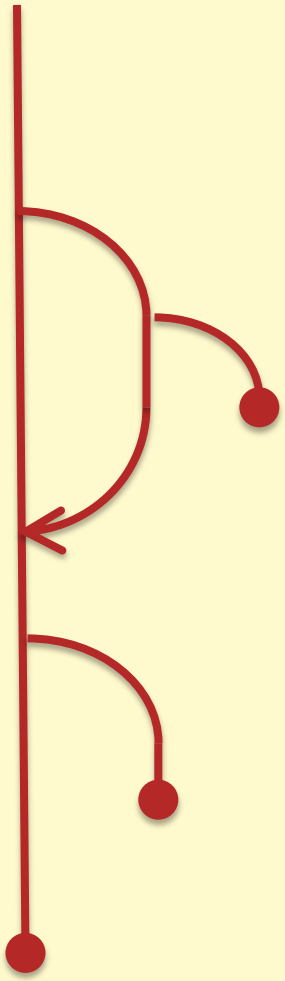
- Directed Graphs
  - One root
  - One or more terminals
- Operational construction rules
  - Fork
    - One terminal -> two terminals
  - Update/Query
    - Append to terminal
  - Join
    - Two terminals -> one terminal
    - Subject to join condition

Revision Diagrams



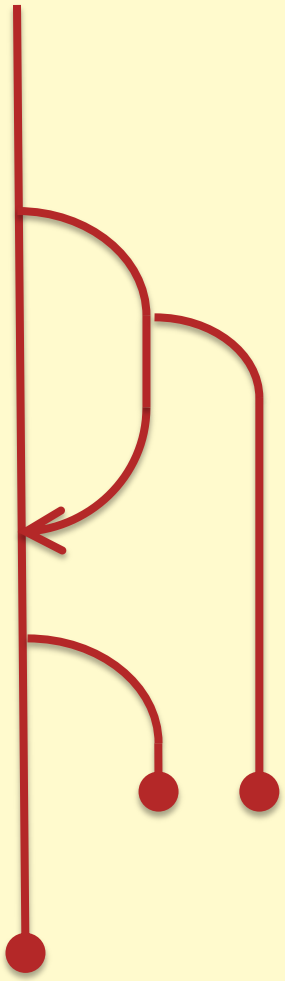
- Directed Graphs
  - One root
  - One or more terminals
- Operational construction rules
  - Fork
    - One terminal -> two terminals
  - Update/Query
    - Append to terminal
  - Join
    - Two terminals -> one terminal
    - Subject to join condition

Revision Diagrams



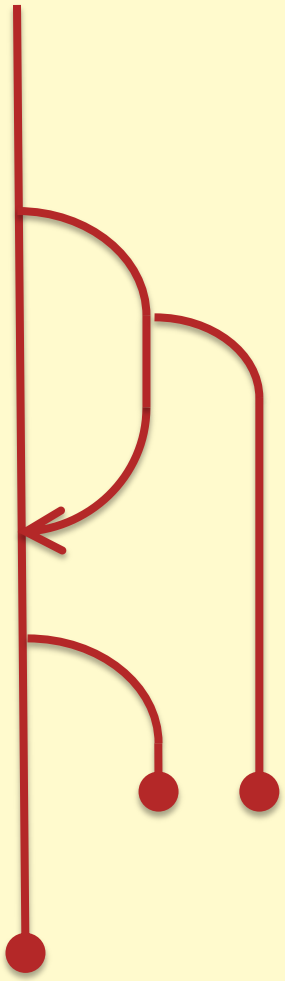
- Directed Graphs
  - One root
  - One or more terminals
- Operational construction rules
  - Fork
    - One terminal -> two terminals
  - Update/Query
    - Append to terminal
  - Join
    - Two terminals -> one terminal
    - Subject to join condition

Revision Diagrams



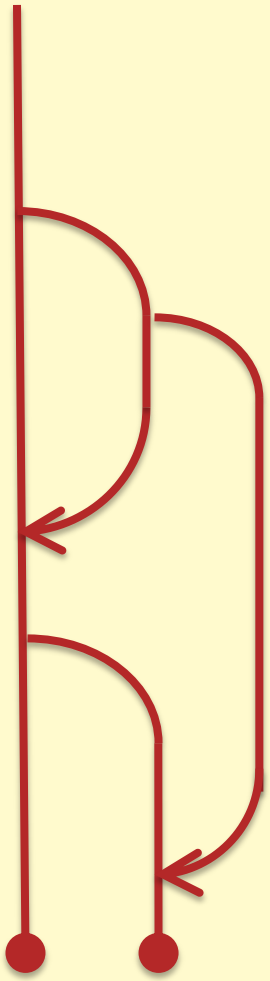
- Directed Graphs
  - One root
  - One or more terminals
- Operational construction rules
  - Fork
    - One terminal -> two terminals
  - Update/Query
    - Append to terminal
  - Join
    - Two terminals -> one terminal
    - Subject to join condition

Revision Diagrams



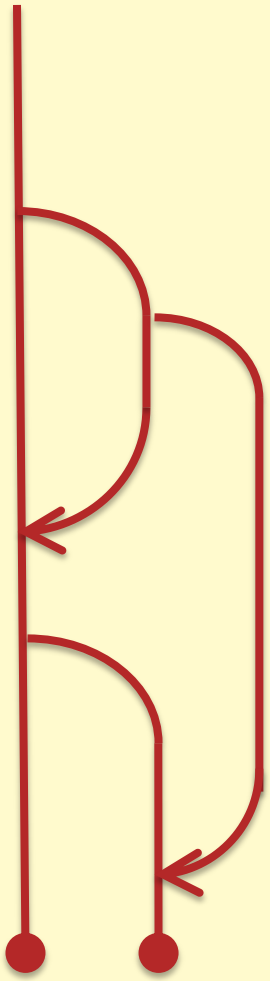
- Directed Graphs
  - One root
  - One or more terminals
- Operational construction rules
  - Fork
    - One terminal -> two terminals
  - Update/Query
    - Append to terminal
  - Join
    - Two terminals -> one terminal
    - Subject to join condition

Revision Diagrams



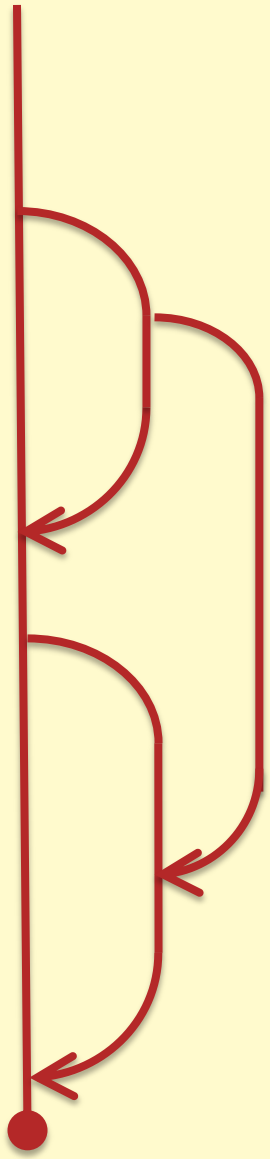
- Directed Graphs
  - One root
  - One or more terminals
- Operational construction rules
  - Fork
    - One terminal -> two terminals
  - Update/Query
    - Append to terminal
  - Join
    - Two terminals -> one terminal
    - Subject to join condition

Revision Diagrams



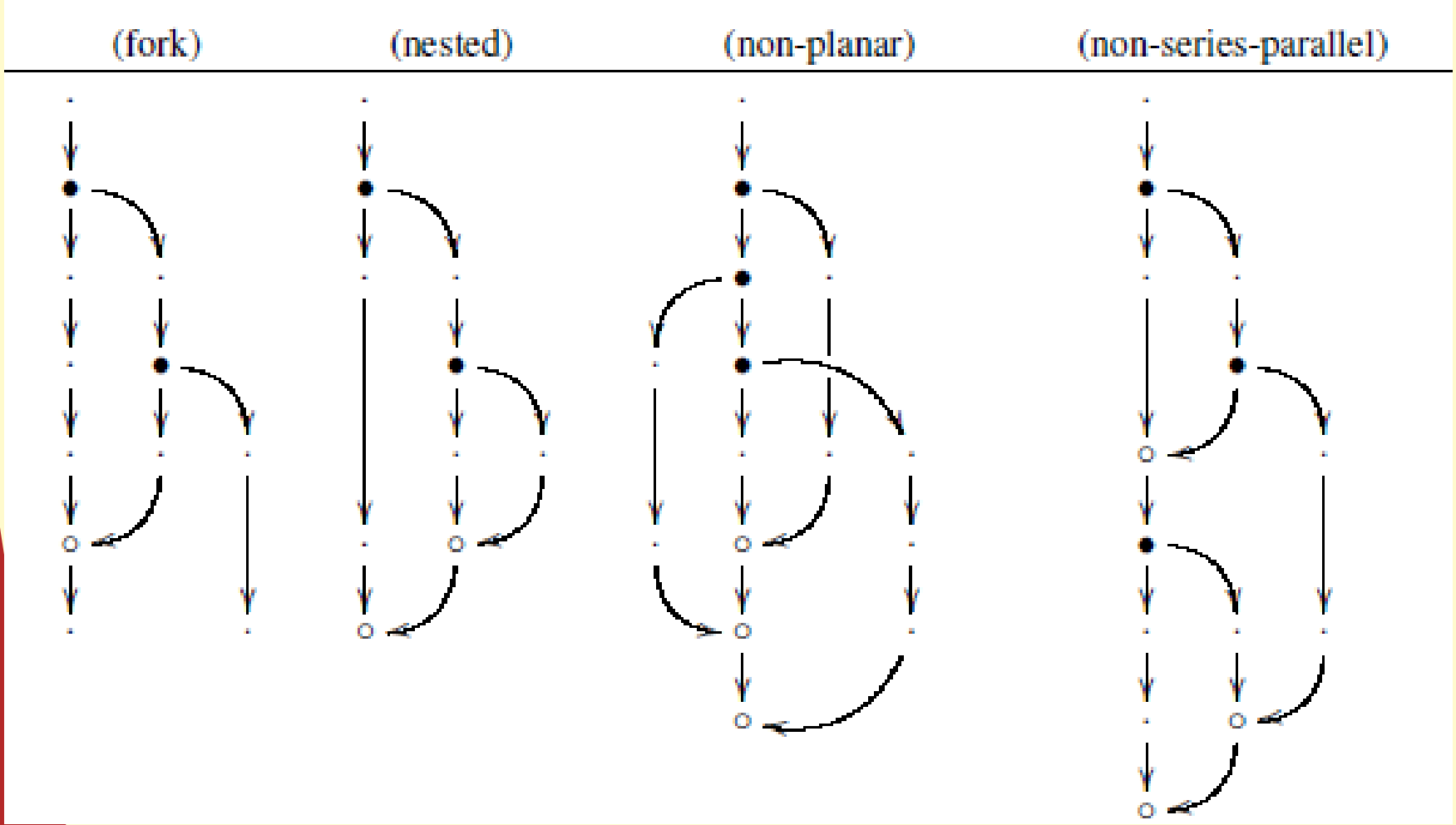
- Directed Graphs
  - One root
  - One or more terminals
- Operational construction rules
  - Fork
    - One terminal -> two terminals
  - Update/Query
    - Append to terminal
  - Join
    - Two terminals -> one terminal
    - Subject to join condition

Revision Diagrams



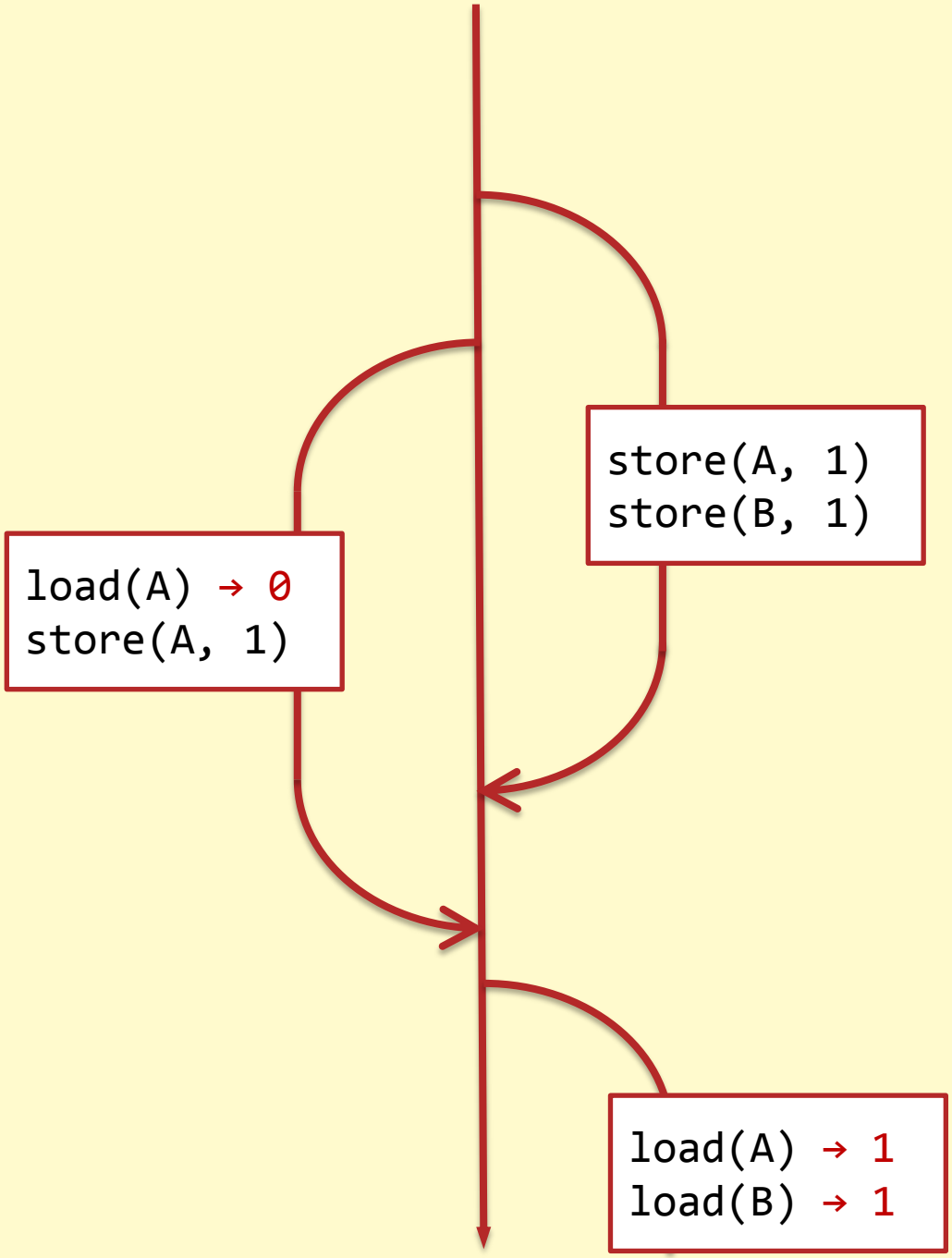
- Directed Graphs
  - One root
  - One or more terminals
- Operational construction rules
  - Fork
    - One terminal -> two terminals
  - Update/Query
    - Append to terminal
  - Join
    - Two terminals -> one terminal
    - Subject to join condition

Example Diagrams



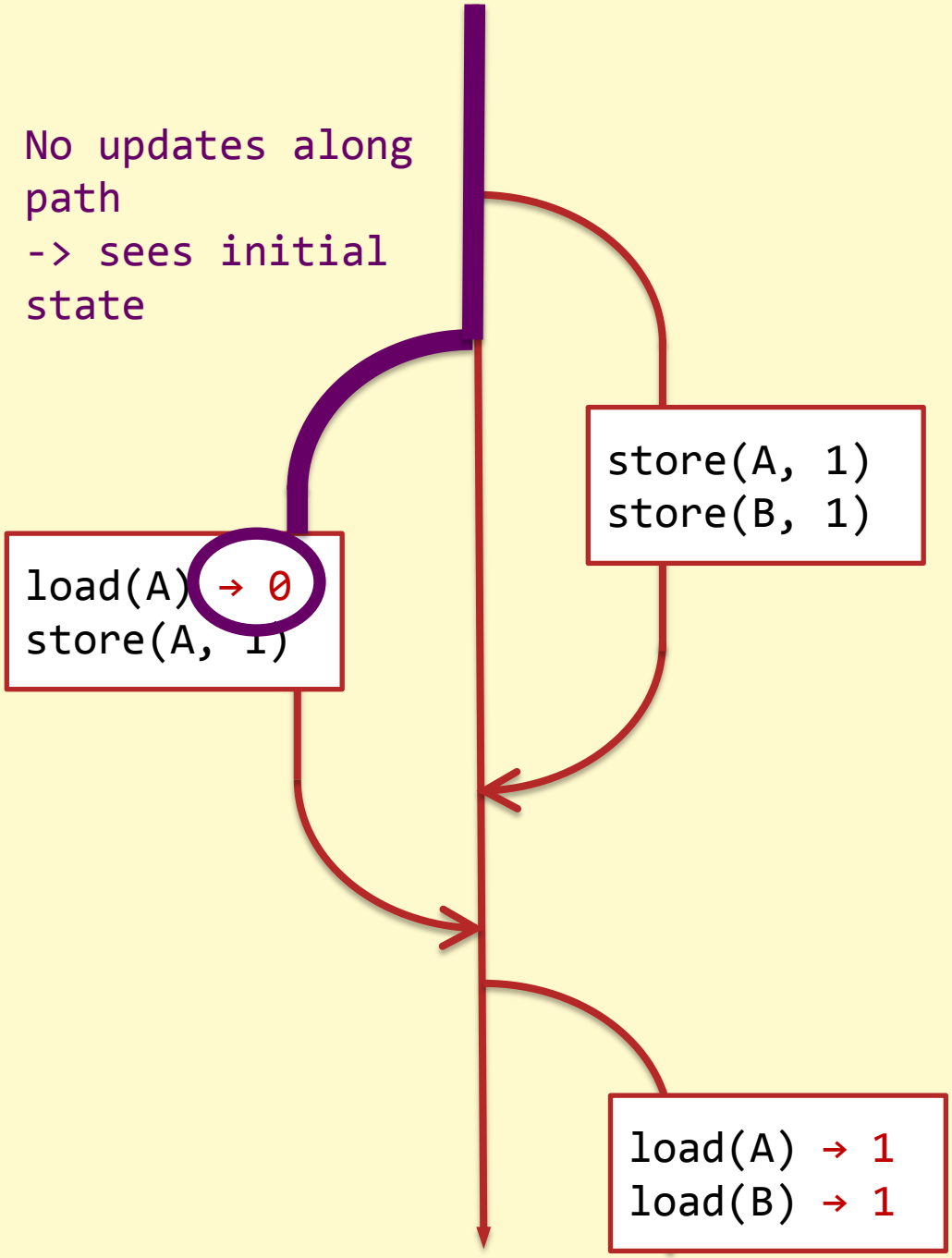
Semantics

- State determined by sequence of updates along path from root
- Operations are replayed at tip of arrow.



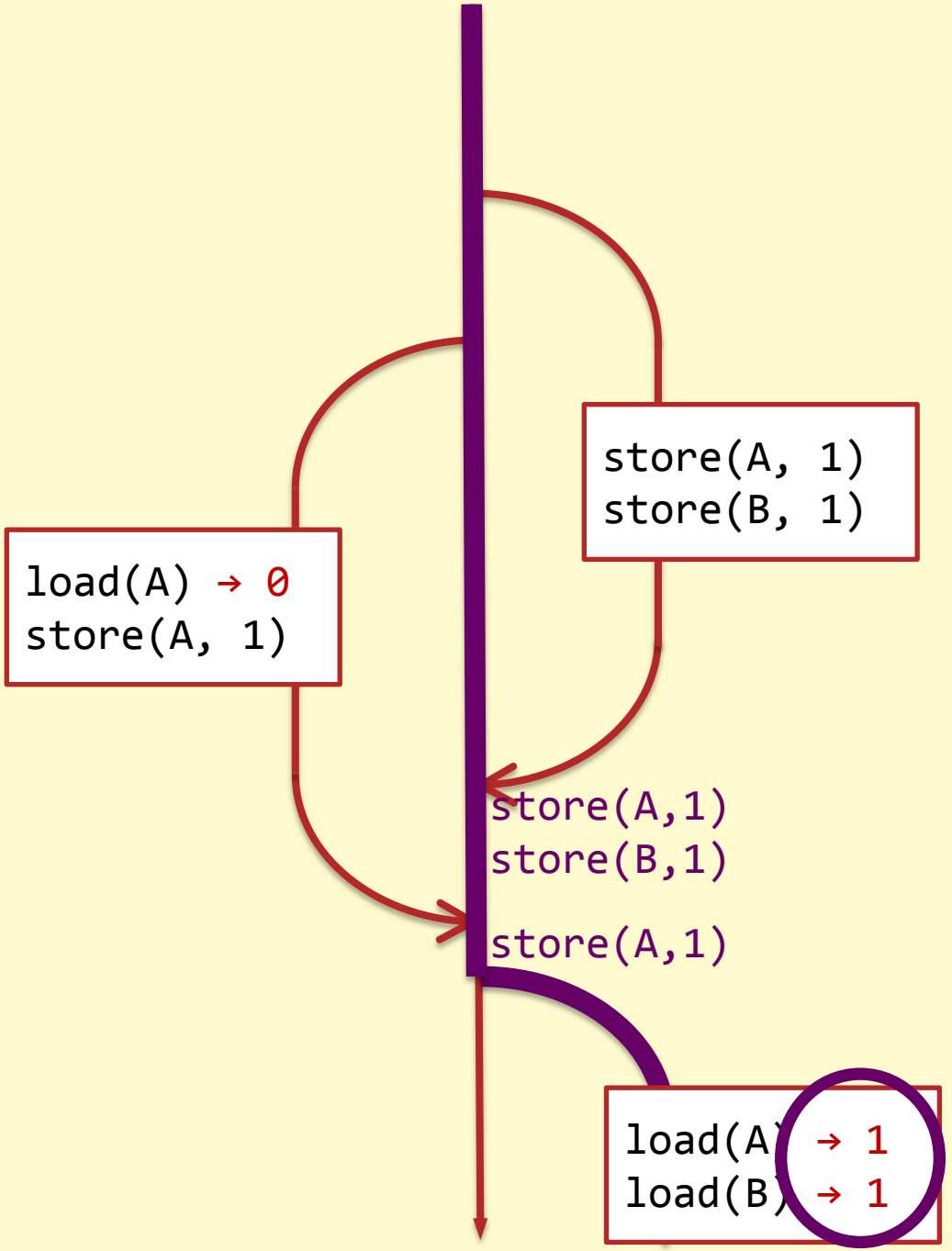
Semantics

- State determined by sequence of updates along path from root
- Updates are replayed at tip of arrow.



Semantics

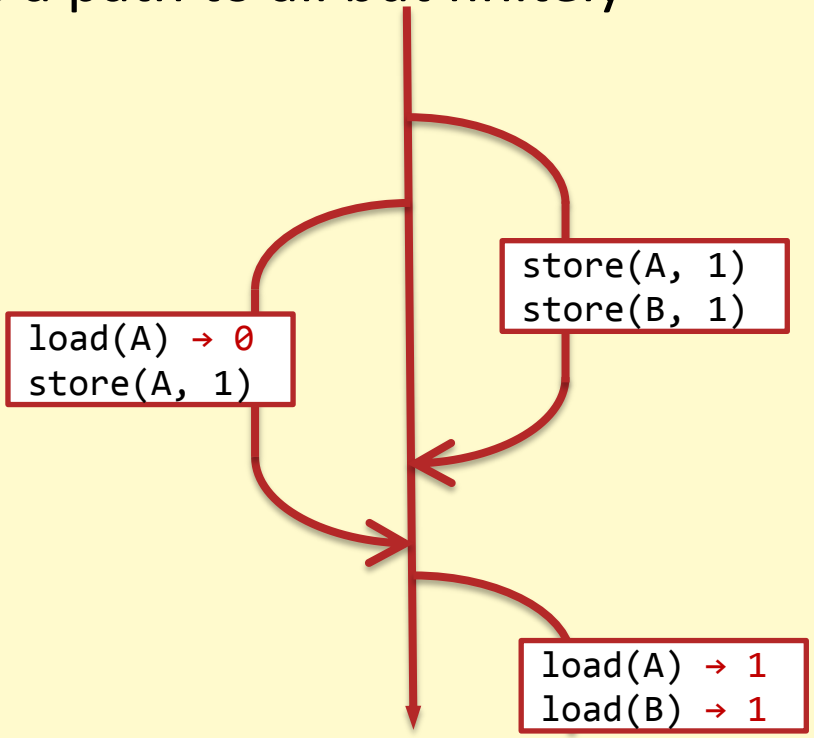
- State determined by sequence of updates along path from root
- Updates are replayed at tip of arrow.



Revision Consistency

- A history is *revision consistent* if we can place the transactions on a revision diagram s.t.
  - Query results are consistent with revision state
  - There is a path between successive transactions
  - Committed transactions have a path to all but finitely many vertices

Client 1	Client 2
load(A) → 0 store(A,1)	store(A, 1) store(B, 1)
	load(A) → 1 load(B) → 1



Revision Consistency Guarantees…

- Transactions don't fail
- Atomicity
  - Other clients see either all or none of the updates of a transactions
- Isolation
  - Updates by incomplete transactions are not visible to other clients
- See own Updates
  - Client sees effect of all updates it has performed up to this point
- Eventual Consistency
  - Completed Transactions eventually settle at a certain position within a common history prefix.
- Causality

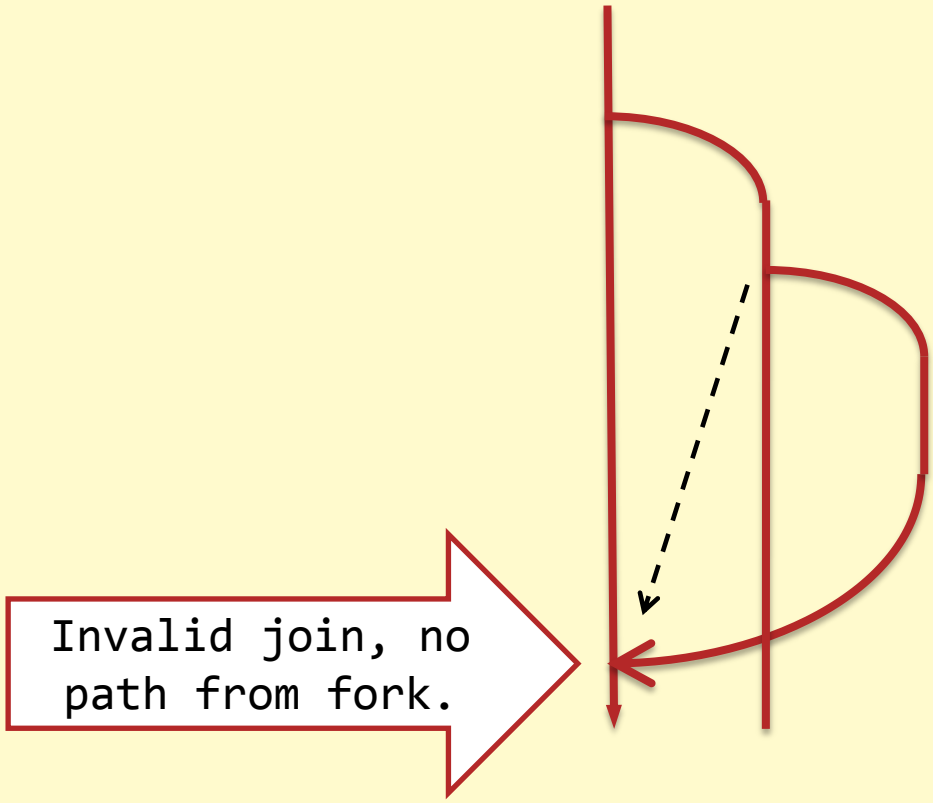
`This one required work to prove.`

  - If B sees updates of A, and C sees updates of B, then C sees updates of A

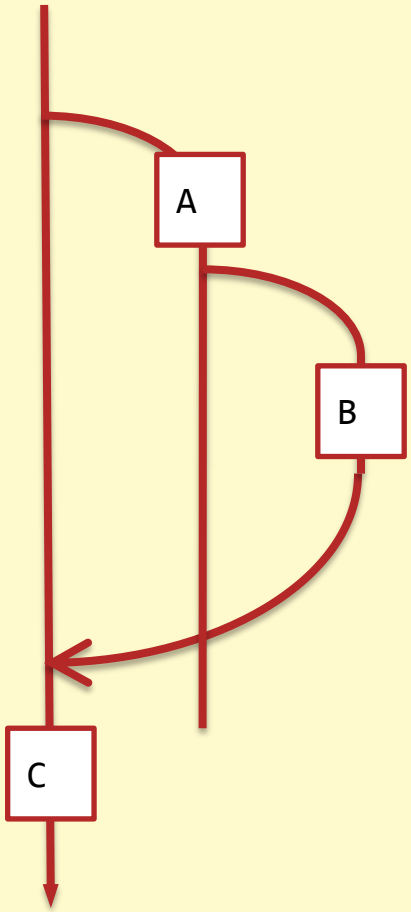
The join condition

Revision diagrams are subject to the join condition:

A revision can only be joined into vertices that are reachable from the fork.



Without join condition, causality is violated.



- B sees updates of A
- C sees updates of B
- But C does not see updates of A

- *Visibility is not transitive*.

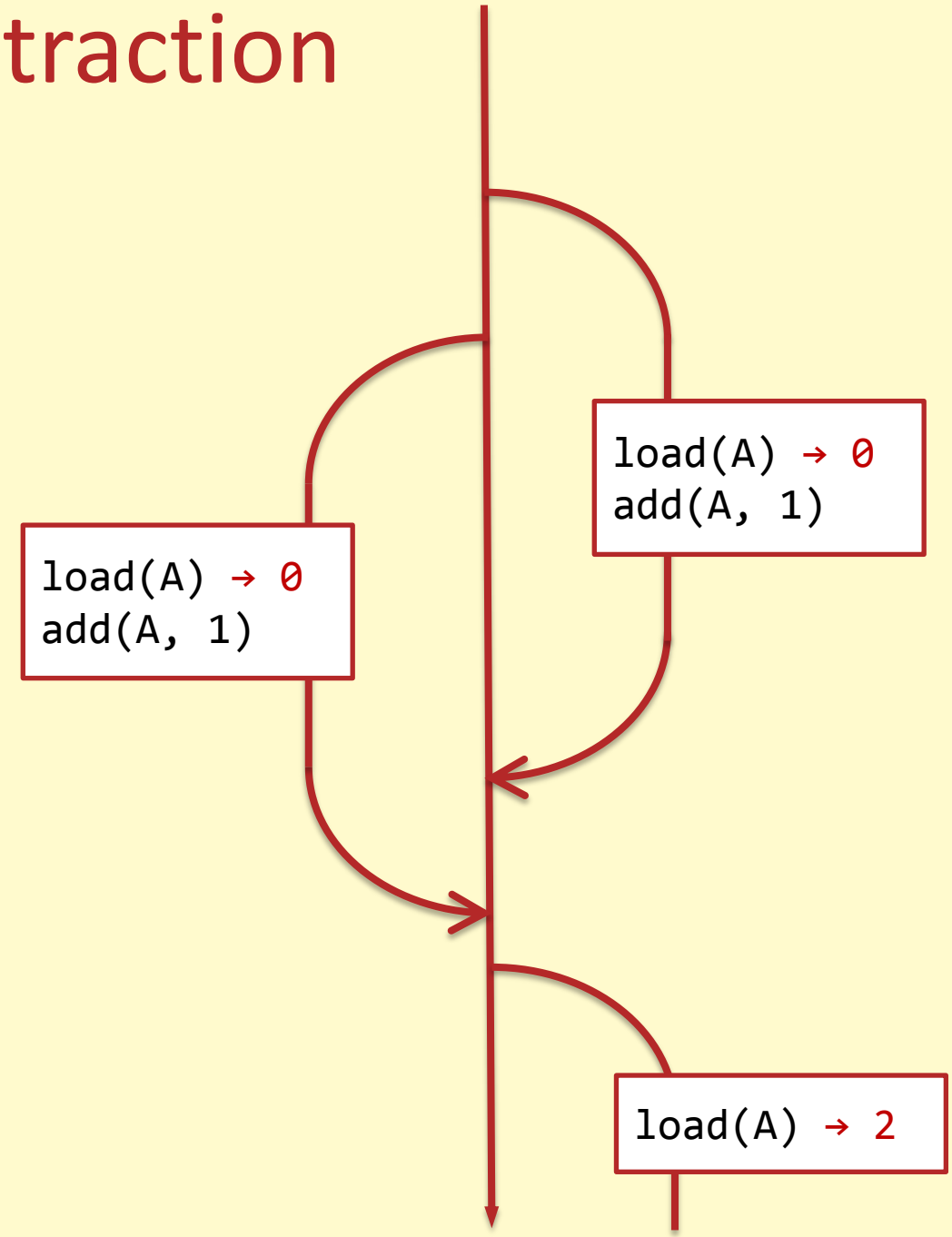
- We prove in paper: enforcing join condition is sufficient to guarantee transitive visibility.

Main Ingredient #2:

RAISE DATA ABSTRACTION

Raise Data Abstraction

- Richer data types enable updates that behave better
- Integer example:
  - Queries = {load(a)}
  - Updates = {store(a,v)} ∪ **{add(a,v)}**



Query-Update Automata (QUA)

- A query-update interface tuple (Q, V, U)
  - Q – Query operations
  - V – Values returned by query
  - U – Update operations
- A query update automaton $(S, s_0, \#)$ over (Q, V, U)
  - A set of states S
  - An initial state $s_0 \in S$
  - For each q in Q, an interpretation $q^{\#}: S \rightarrow V$
  - For each u in U, an interpretation $u^{\#}: S \rightarrow S$

Why QUAs ?

- QUAs keep the conversation general
  - QUAs can represent simple shared memory
  - QUAs can represent abstract data types
  - QUAs can represent entire databases
- QUAs provide what we need to define EC
  - Clean separation of queries (no side effects) and updates (only side effects, no return value)
  - Updates contain information about intent, and are total functions (can apply to different state)
- QUAs can enable optimized implementations
  - Bounded metadata, not unbounded logs

A fork-join QUA

- Rather than maintaining update logs or walking graphs, we can use an FJ-QUA to represent the state of a revisions
- A fork-join QUA is a tuple (Σ, σ_0, f, j, #)
  - An initial state $\sigma_0 \in \Sigma$
  - For each q in Q, an interpretation $q^{\sharp}$: $S \rightarrow V$
  - For each u in U, an interpretation $u^{\sharp}$: $S \rightarrow S$
  - A fork operation f: $\Sigma \rightarrow \Sigma \times \Sigma$
  - A join operation j: $\Sigma \times \Sigma \rightarrow \Sigma$

Programming with FJ-QUAs
[ECOOP'12, to appear]

- No need to develop custom FJ-QUAs for each application: programmer can compose them
- Support FJ-QUAs for basic types
  - Cloud Integers
  - Cloud Strings
- Support FJ-QUAs for certain collection types
  - Cloud Entities
  - Cloud Arrays

System Models

- Revision diagrams are a visualization tool; actual system can use a variety of implementations
  - Paper gives several system models.
  - Since revisions can be nested, we can use server pools to scale to large numbers of clients.

THE AXIOMATIC MODEL

Sequential Consistency

- Histories are sequentially consistent if the query results are consistent with a single order of the transactions

Client 1

```
store(17, 1)
load(17) → 1
load(19) → 0
```

```
load(17) → 2
```

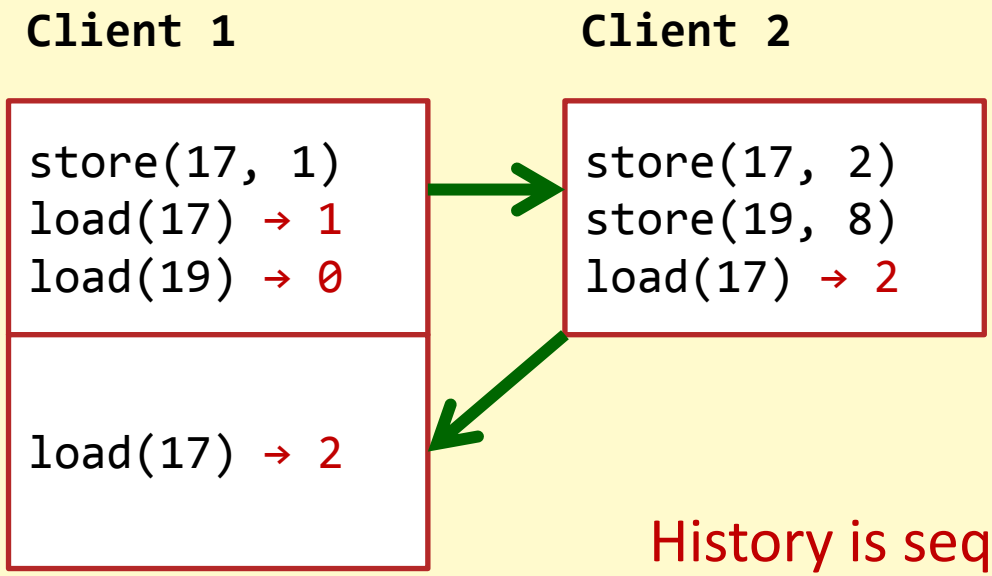
Client 2

```
store(17, 2)
store(19, 8)
load(17) → 2
```

Is this history sequentially consistent?

Sequential Consistency

- Histories are sequentially consistent if the query results are consistent with a single order of the transactions



History is sequentially consistent.

Sequential Consistency, Formally

- A history H is sequentially consistent if there exists a partial order $<$ on E_H such that
  - $<$ extends program order $<_p$
  - (atomicity) $<$ factors over transactions
  - $<$ is a total order on past events
    - $(e_1 < e) \wedge (e_2 < e) \Rightarrow (e_1 < e_2) \vee (e_2 < e_1) \vee (e_2 = e_1)$
  - All query results $(q, v) \in E_H$ are consistent with $<$
    - $v = q^{\sharp}(u_k{}^{\sharp}(u_{k-1}{}^{\sharp} \ldots {}^{\sharp}(u_1{}^{\sharp}(s_0))))$
    - $u_1 < u_2 < \ldots < u_k < (q, v)$
  - (isolation) if $e_1 < e_2$ and e_1 is not committed then $e_1 <_p e_2$
  - (eventual delivery) For all committed transactions t there exists only finitely many transactions t' such that not t $<$ t'

Eventual Consistency

- Histories are eventually consistent if the query results are consistent with **two orders**:
  - A visibility order (partial order) that determines what updates a query can see
  - An arbitration order that orders all updates
- The value returned by a query is determined by those two orders (apply all visible updates in arbitration order to the initial state)

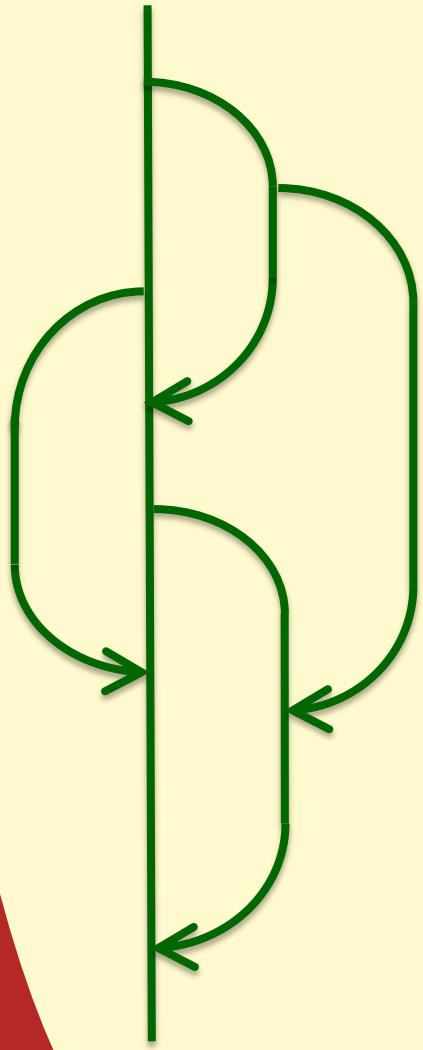
Eventual Consistency, Formally

- A history H is eventually consistent if there exist partial orders $<_v$ (visibility order) $<_a$ (arbitration order) on E_H such that
  - $<_v$ extends program order $<_p$
  - $<_a$ extends $<_v$
  - (atomicity) Both $<_v$ and $<_a$ factor over transactions
  - (total order on past events)
    - $(e_1 <_v e) \wedge (e_2 <_v e) \Rightarrow (e_1 <_a e_2) \vee (e_2 <_a e_1) \vee (e_2 = e_1)$
  - All visible query results $(q, v) \in E_H$ are consistent with $<_a$
    - $v = q^{\sharp}(u_k{}^{\sharp} u_{k-1}{}^{\sharp} \ldots u_1{}^{\sharp}(s_0))$
    - $u_1 <_a u_2 <_a \ldots <_a u_k$
    - $u_i <_v (q, v)$
  - (isolation) if $e_1 <_v e_2$ and e_1 is not committed then $e_1 <_p e_2$
  - (eventual delivery) For all committed transactions t there exists only finitely many transactions t' such that not $t <_v t'$

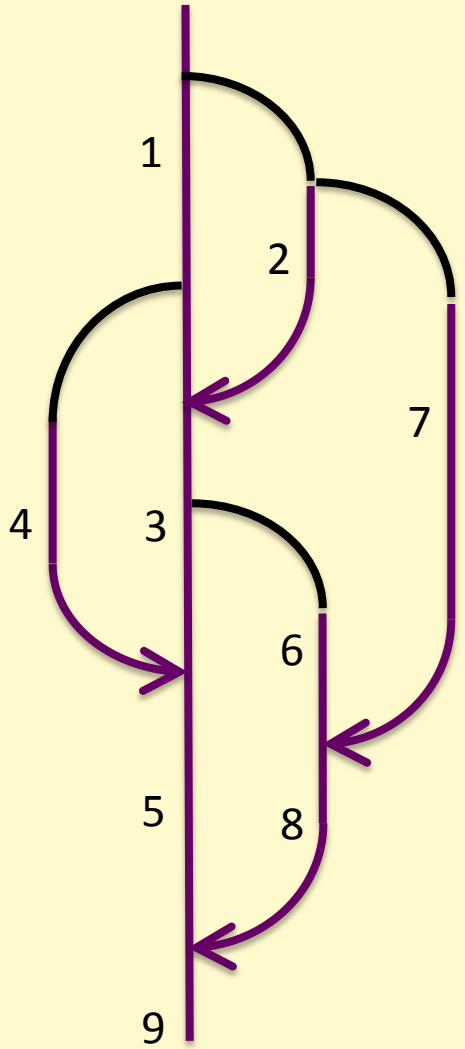
Proved in Paper

- Sequentially consistent histories are eventually consistent.
  (easy, direct from definition)
- Revision-consistent histories are eventually consistent.
  (some nontrivial parts, related to join condition)
- There exist some eventually consistent histories that are not revision-consistent.
  (give counterexample)

See Visibility & Arbitration Order in Revision Diagrams



- Visibility = Reachability
- Arbitration = Cactus Walk



Related Work

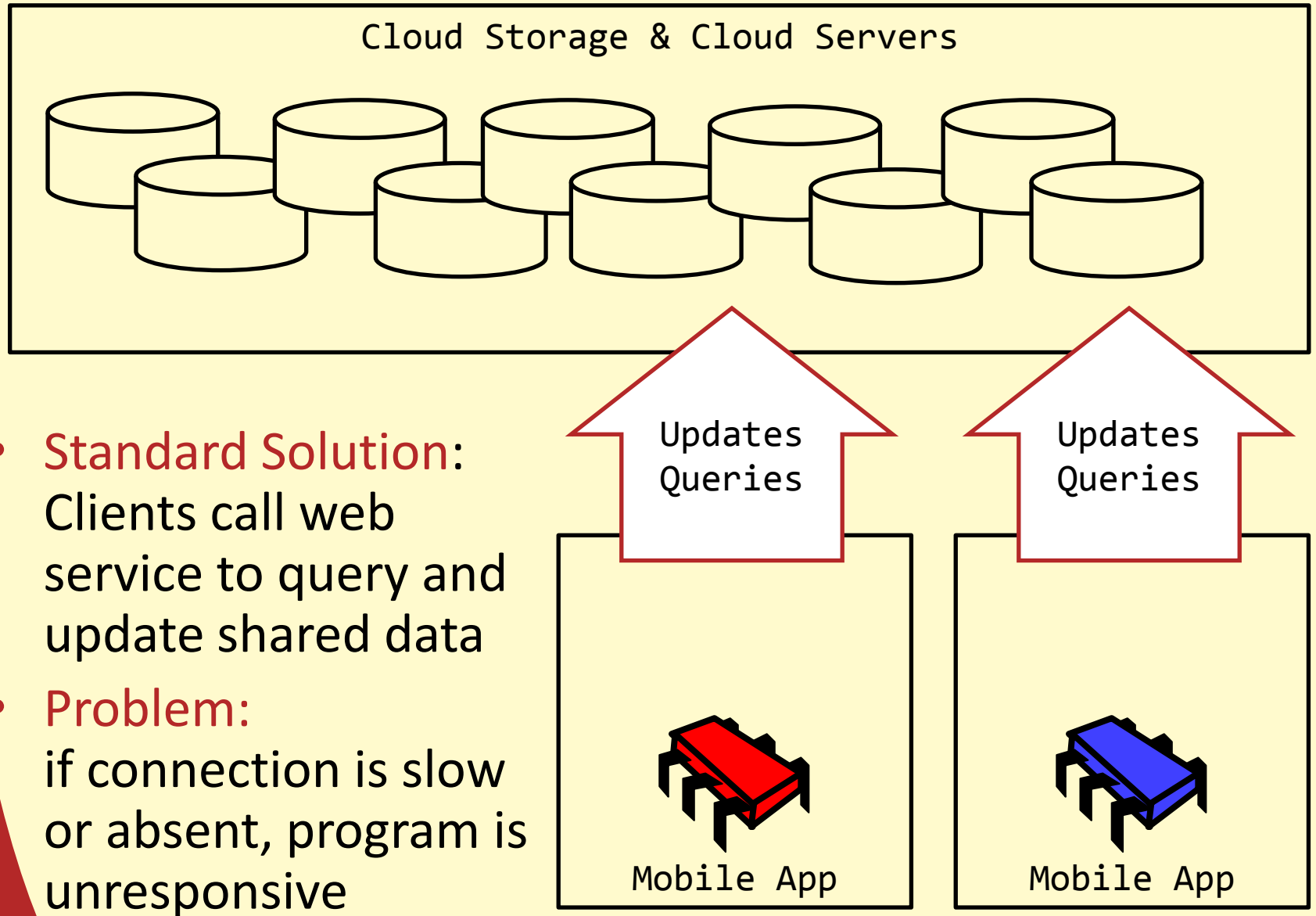
- Marc Shapiro et al.
  - Same motivation and similar techniques
  - CRDTs (Conflict-Free Replicated Data Types) require operations to commute, unlike QUAs which support non-commuting updates.
- Transactional Memory Research
  - Provided many of the insights used in this work
- Relaxed Memory Model Research
  - Provided insights on axiomatic & operational memory models

PART III: CLOUD TYPES

Sharing Data Across Mobile Devices

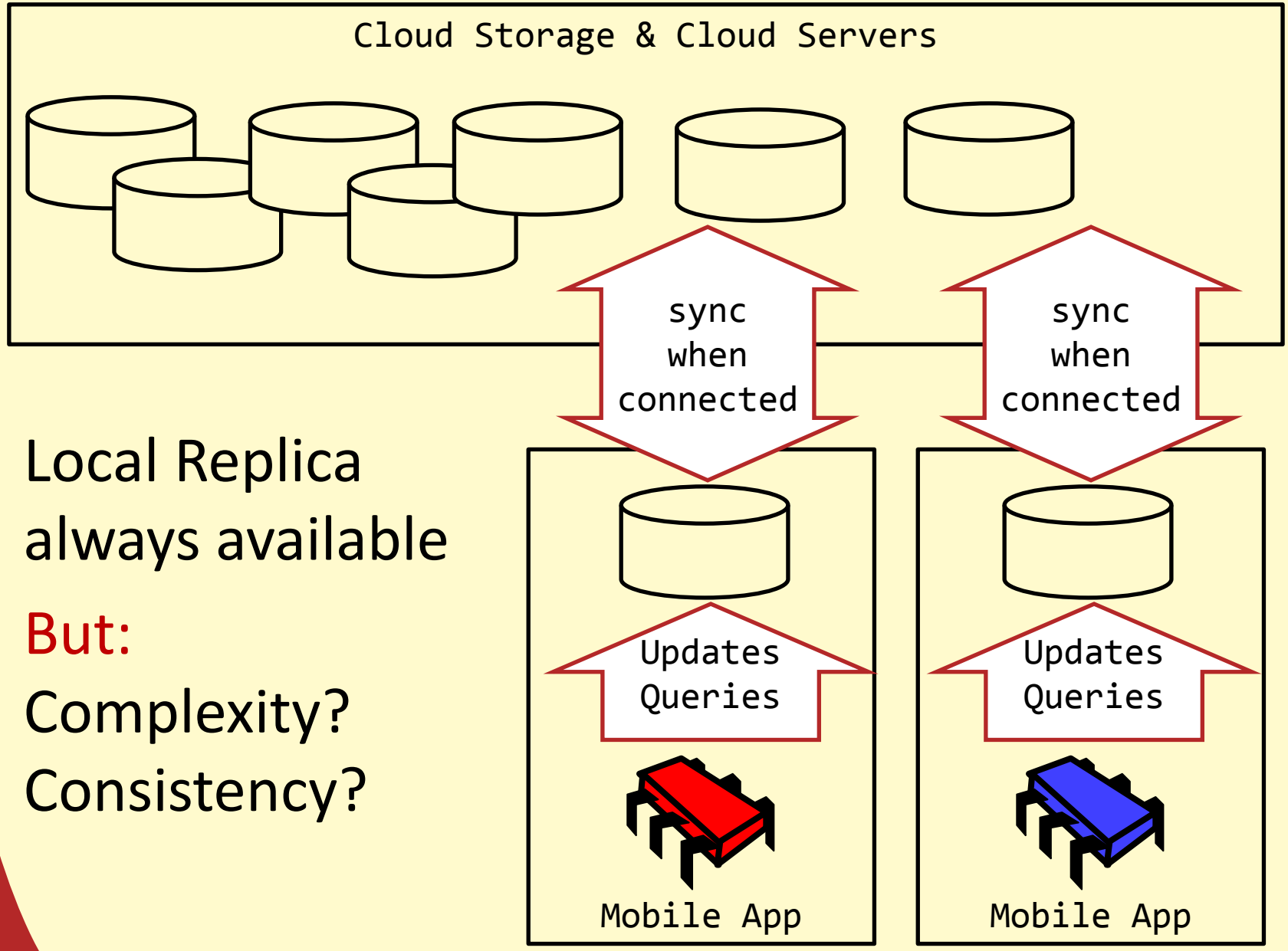
- Sharing data in the cloud makes apps more social, fun, and convenient.
- Examples: Games, Settings, Chat, Favorites, Ratings, Comments, Grocery List…
- But implementation is challenging.

Sharing Data Across Mobile Devices



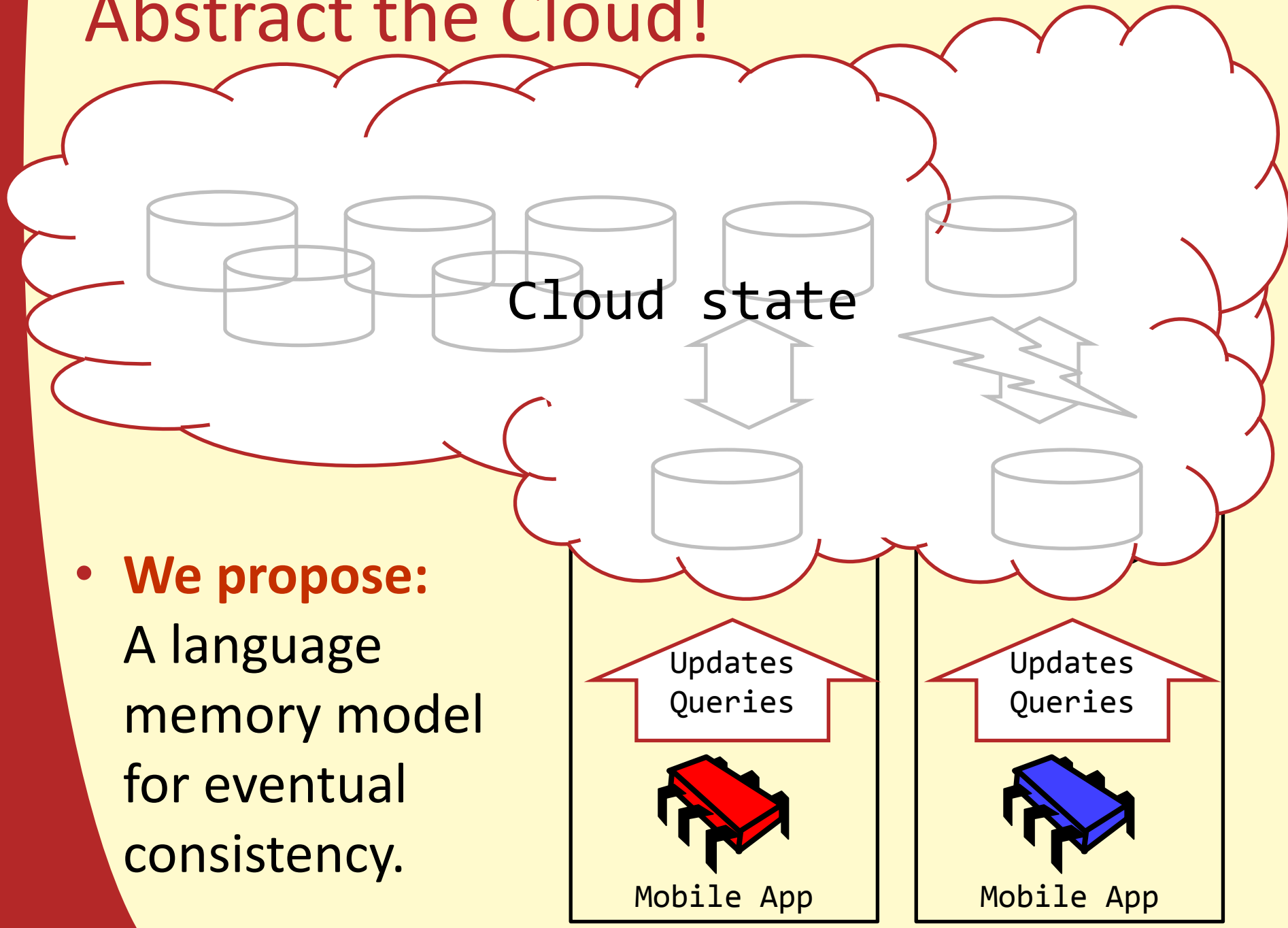
- Standard Solution: Clients call web service to query and update shared data
- Problem: if connection is slow or absent, program is unresponsive

Sharing Data w/ Offline Support



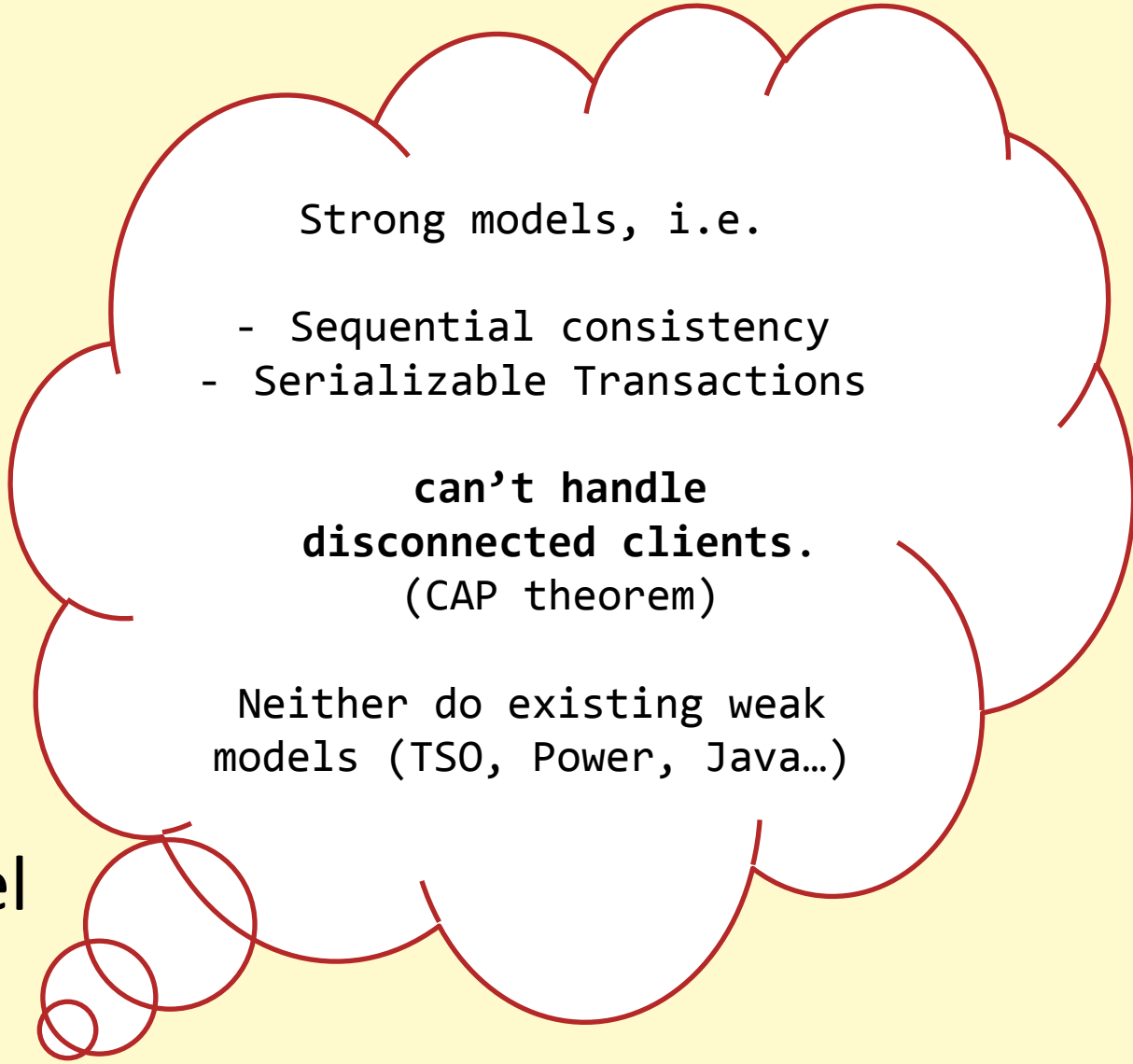
- Local Replica always available
- But:
  Complexity?
  Consistency?

Abstract the Cloud!



- **We propose:**
  A language memory model for eventual consistency.

Abstract the Cloud!



- **We propose:**
  A language memory model for eventual consistency.

How do we define this memory model?

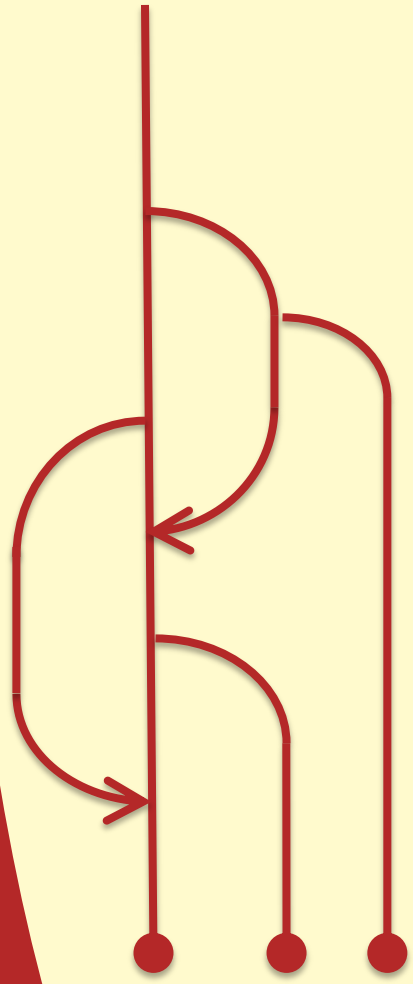
- Informal operational model

  We will give you a quick intro on the next couple slides

- Formal operational model
- 2 Example Implementations (single server, server pool)
- Formal axiomatic model

Beyond the scope of this talk, see papers [ESOP2012, ECOOP2012]

Powered By **Concurrent Revisions**



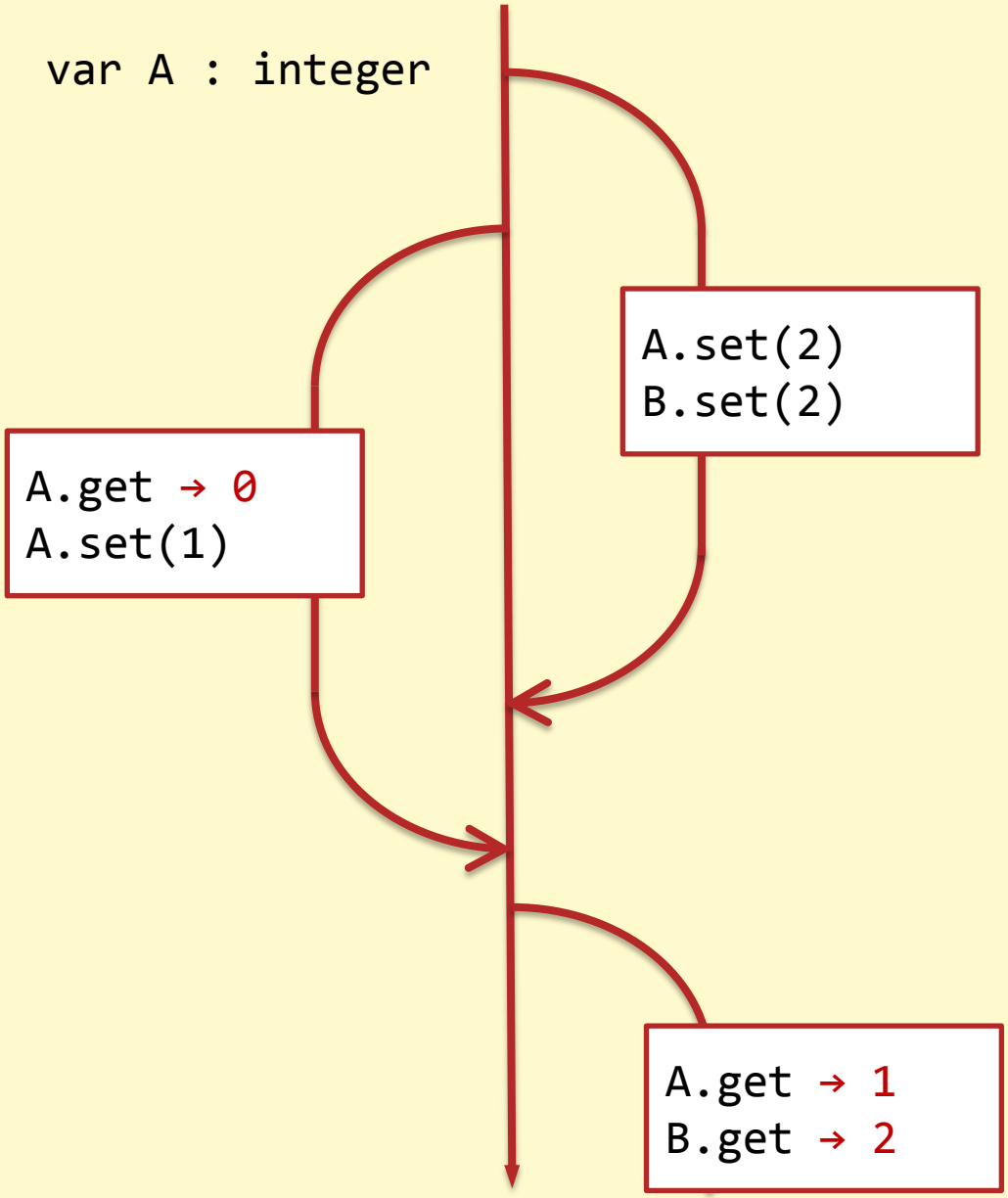
[OOPSLA'10] [WoDet'11] [ESOP'11] [OOSPLA'11] [ESOP'12] [ECOOP'12]

- reminiscent of source control systems
- but: about application state, not source code

1. Models state as a *revision diagram*
   - *Fork:* creates revision (snapshot)
   - Queries/Updates target specific revision
   - *Join:* apply updates to joining revision
2. Raises data abstraction level
   - Record operations, not just states

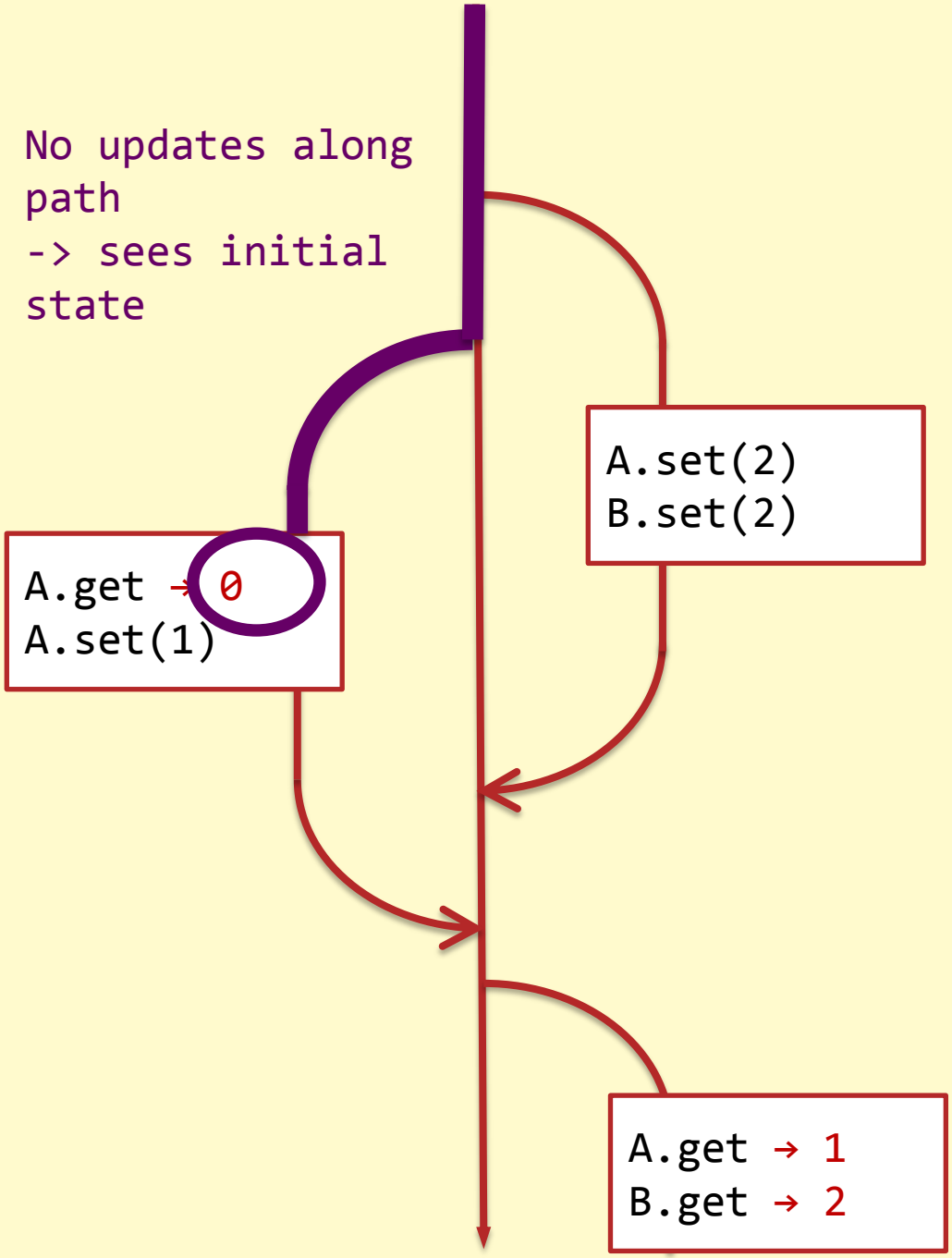
Semantics of Concurrent Revisions

- State determined by sequence of updates along path from root
- Inserts updates at tip of arrow.



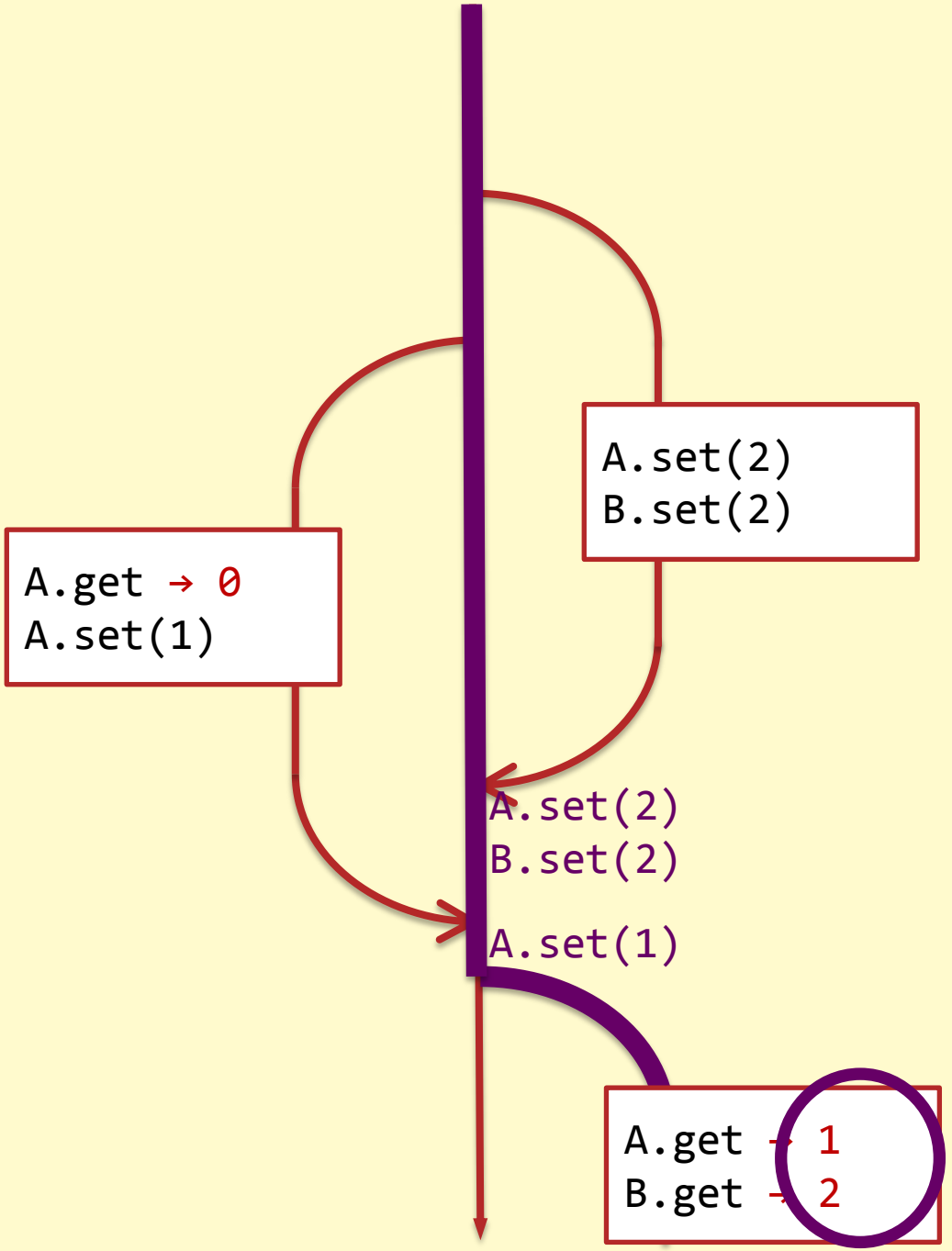
Semantics

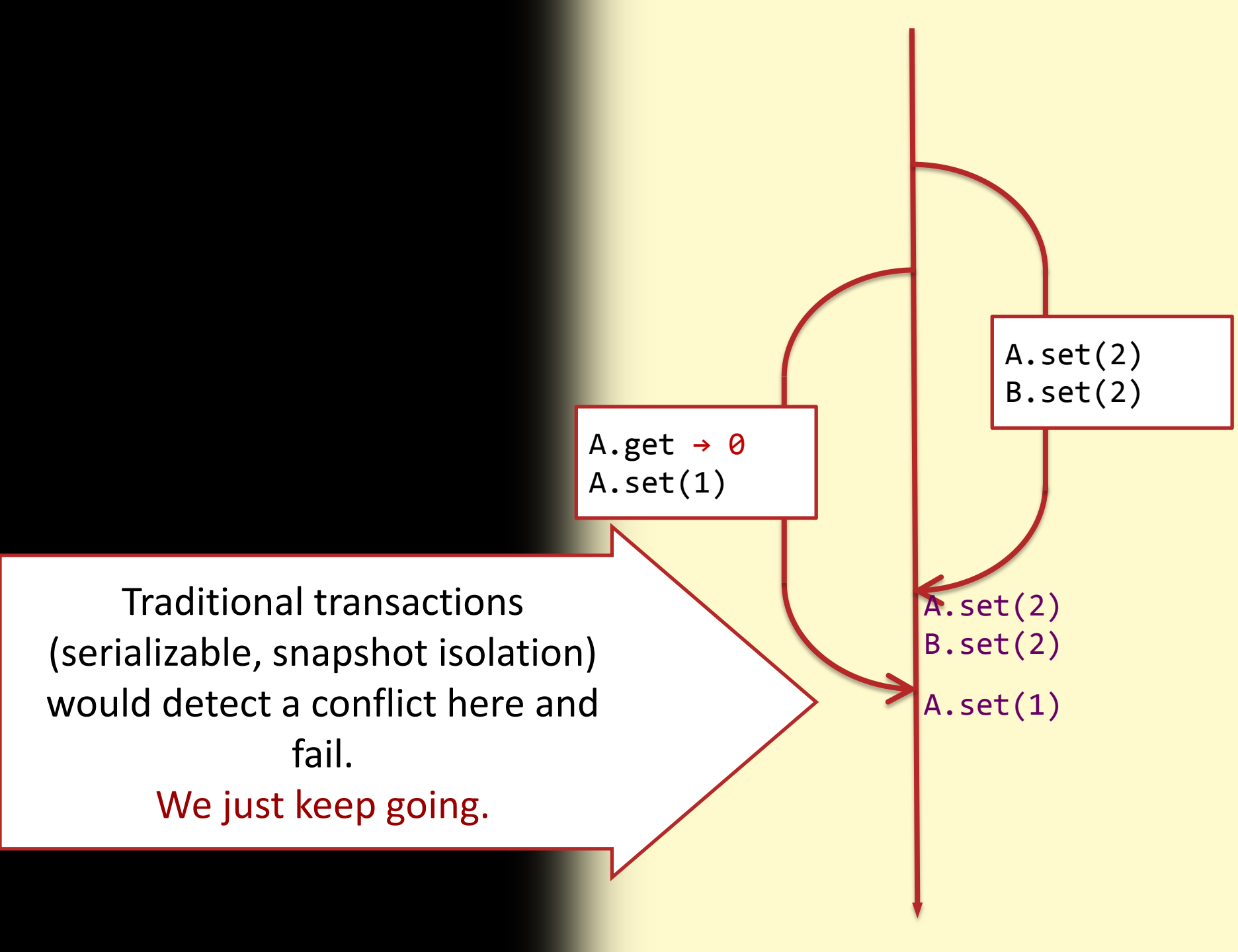
- State determined by sequence of updates along path from root
- Inserts updates at tip of arrow.



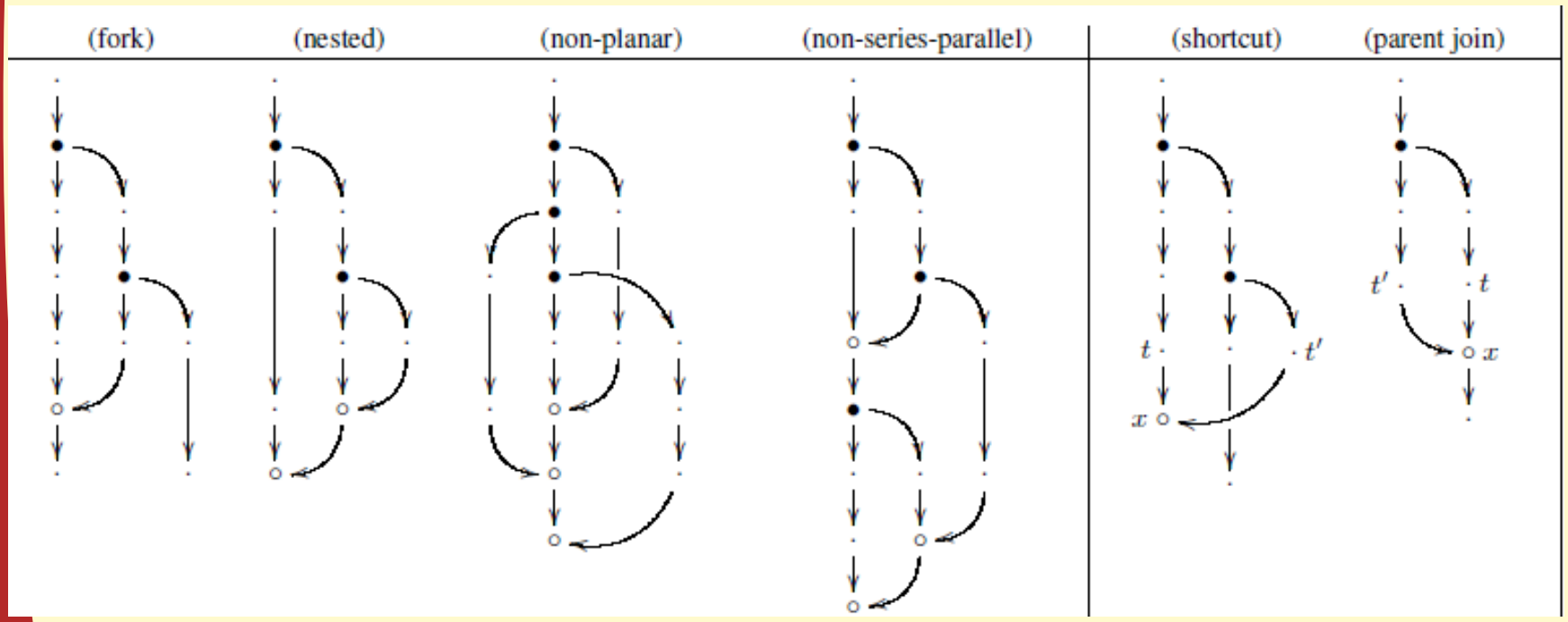
Semantics

- State determined by sequence of updates along path from root
- Inserts updates at tip of arrow.




A.get → 0
A.set(1)
A.set(2)
B.set(2)
A.set(2)
B.set(2)
A.set(1)
Traditional transactions (serializable, snapshot isolation) would detect a conflict here and fail.
We just keep going.

Revision Diagrams



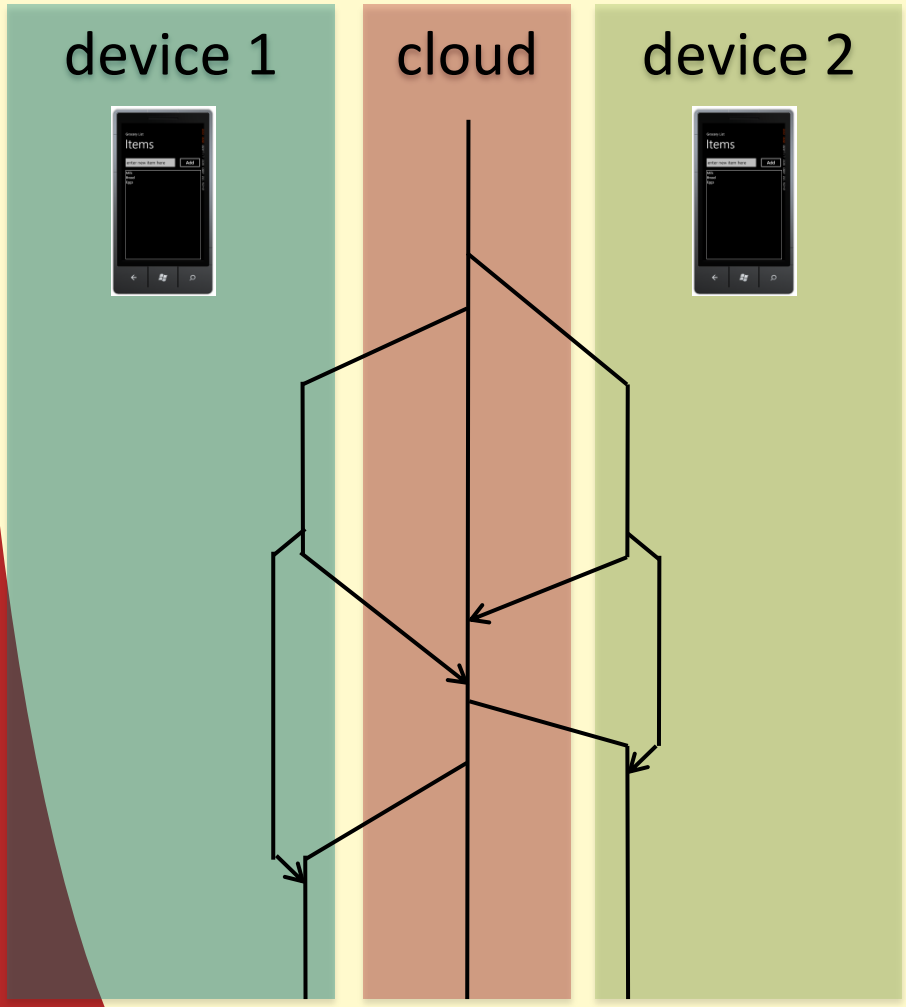
These are revision diagrams

These are not revision diagrams

- Less general than DAGs, more general than SP-graphs

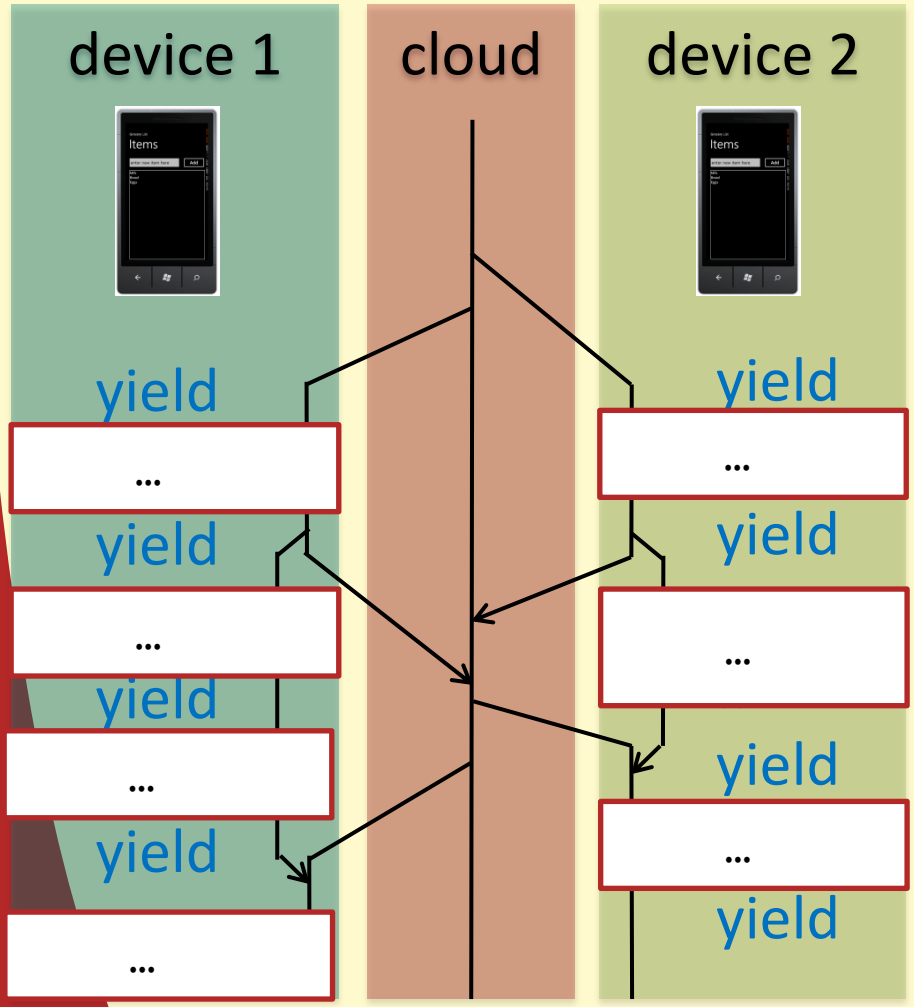
  See [ESOP11], [ESOP10] for formal definitions

Cloud State = Revision Diagram



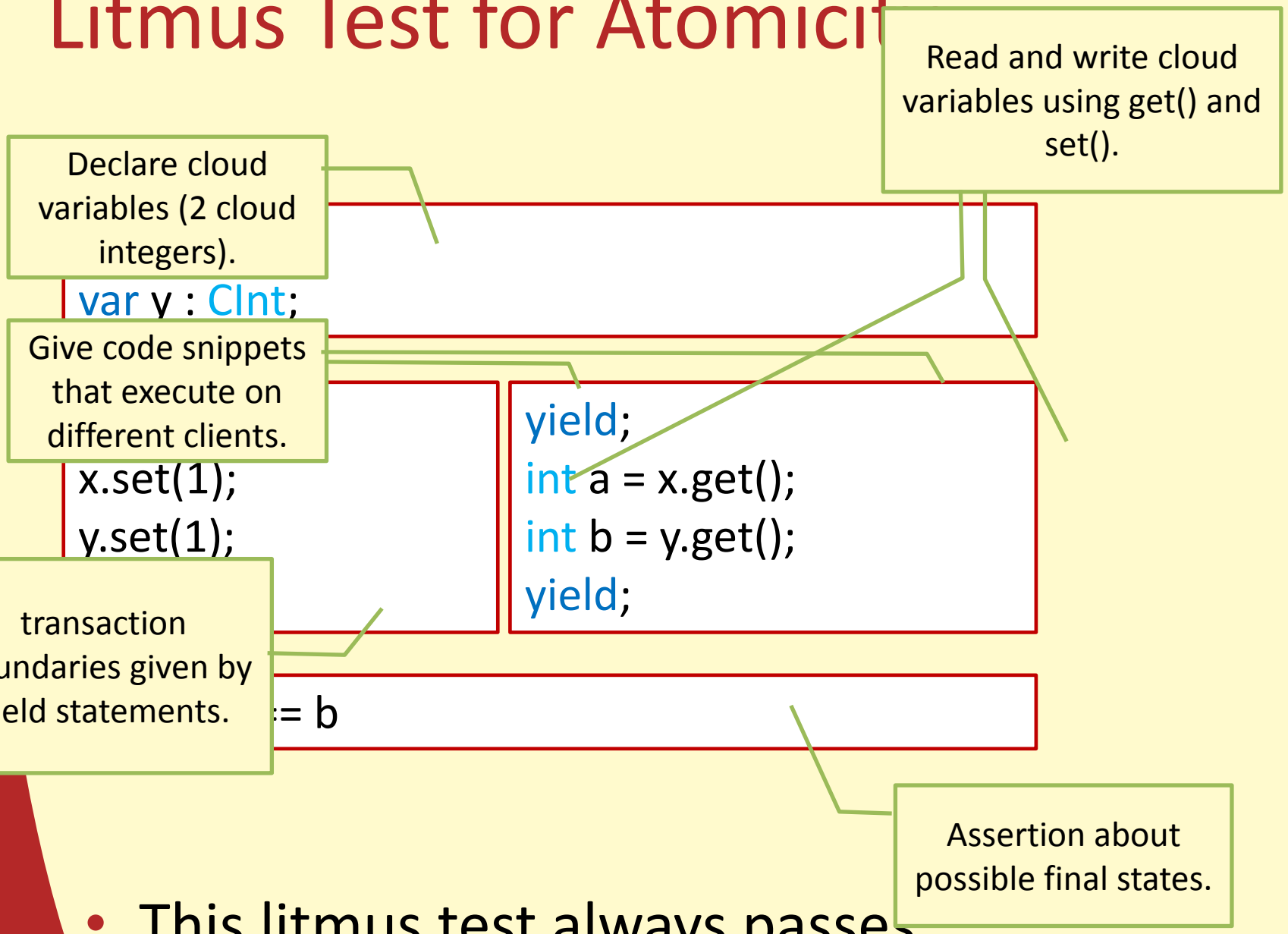
- Client code:
  - reads/modifies data
  - yields
- Runtime:
  - Applies operations to local revision
  - Asynchronous sends/receive at yield points

Yield marks transaction boundaries



- **At yield**
  Runtime has permission to send or receive updates
- **In between yields**
  Runtime is not allowed to send or receive updates

Litmus Test for Atomicit



- This litmus test always passes.

Another simple Litmus Test

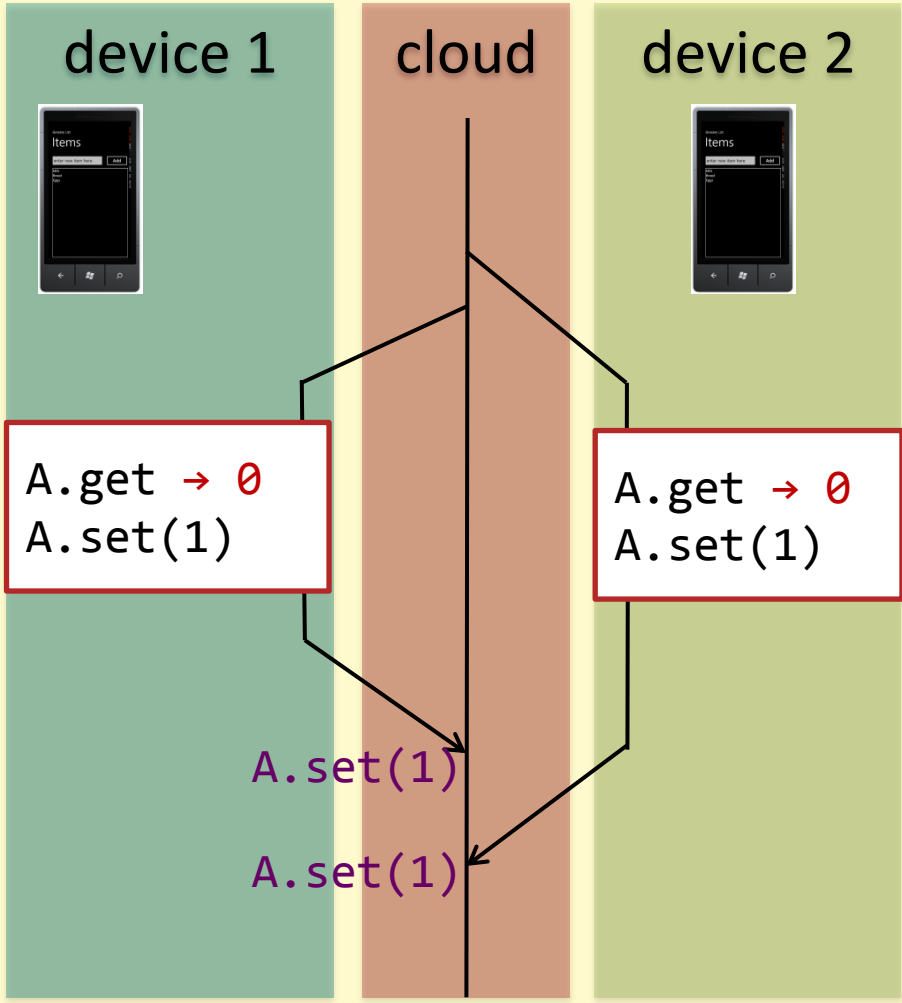
```
var x : CInt;
```

```
yield;
x.set(x.get() + 1));
yield;
```

```
yield;
x.set(x.get() + 1));
yield;
```

```
always x == 2
```

- This litmus test fails!
  Final value x == 1 possible.
- Because devices operate on local snapshots which may be stale.

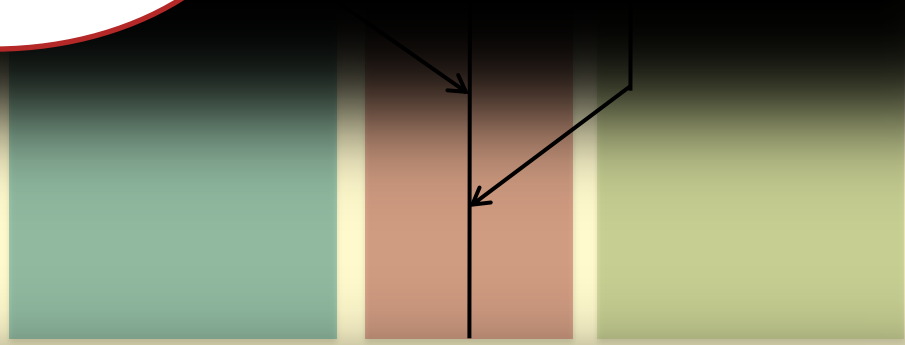


How can we write sensible programs under these conditions?

Idea: Raise Abstraction Level of Data

Use **Cloud Types** to capture more semantic information about updates.

- Because devices operate on local snapshots which may be stale.



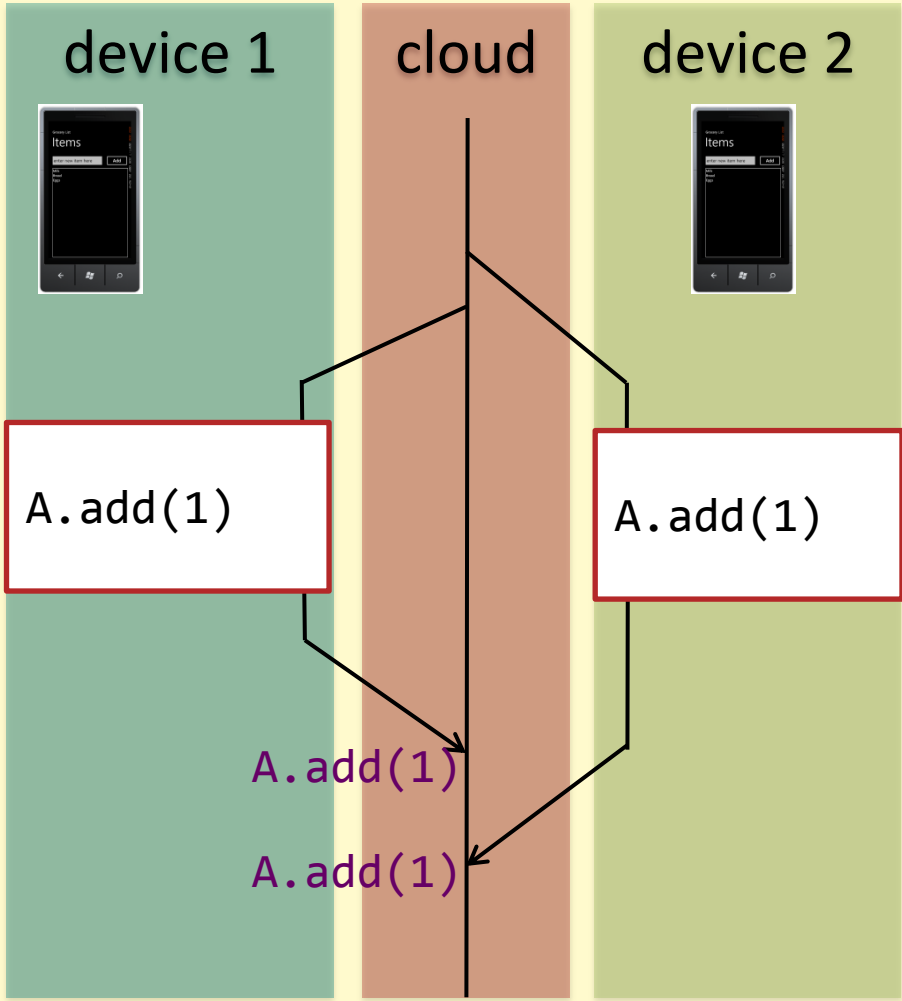
It works if we add instead of set

```
var x : CInt;

yield;
x.add(1);
yield;

yield;
x.add(1);
yield;

always x == 2
```



- Final value is determined by serialization of updates in main revision.
- Effect of adds is cumulative!

  Final value is always 2.

What is a cloud type?

- An abstract data type with
  - Initial value — e.g. { 0 }
  - Query operations — e.g. { get }
    - No side effects
  - Update operations — e.g. { set(x), add(x) }
    - Total (no preconditions)
- Good cloud types minimize programmer surprises.

Our goals for finding cloud types…

- to select only a few
  - But ensure many others can be derived

- to choose types with minimal anomalies
  - Updates should make sense even if state changes

Forces us to rethink basic data structuring.

- objects&pointers fail the second criterion
- entities&relations do better

Our Collection of Cloud Types

Primitive cloud types

- **Cloud Integers**
  { get } { set(x), add(x) }
- **Cloud Strings**
  { get } { set(s), set-if-empty(s) }

Structured cloud types

- **Cloud Tables**
  (cf. entities, tables with implicit primary key)
- **Cloud Arrays**
  (cf. key-value stores, relations)

Cloud Tables

- Declares
  - Fixed columns
  - Regular columns
- Initial value: empty
- Operations:
  - new E(f_1,f_2) add new row (at end)
  - all E return all rows (top to bottom)
  - delete e delete row
  - e.f_1
  - e.col_i.op perform operation on cell
    - If e deleted: queries return initial value, updates have no effect

```
cloud table E
(
   f1: index_type1;
   f2: index_type1;
)
{
   col1: cloud_type1;
   col2: cloud_type2;
}
```

Cloud Arrays

- Example:

```
cloud array A
[
   idx1: index_type1;
   idx2: index_type2;
]
{
   val1: cloud_type1;
   val2: cloud_type2;
}
```

- Initial value:
  for all keys, fields have initial value
- Operations:
  - $A[i_1,i_2].val_i.op$ perform operation on value
  - entries $A.val_i$ return entries for which val_i is not initial value

Index types

- Used for keys in arrays
- Used for fixed columns in tables

- Can be
  - Integer
  - String
  - Table entry
  - Array entry

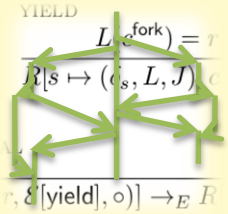
Example App: Birdwatching

- An app for a birdwatching family.

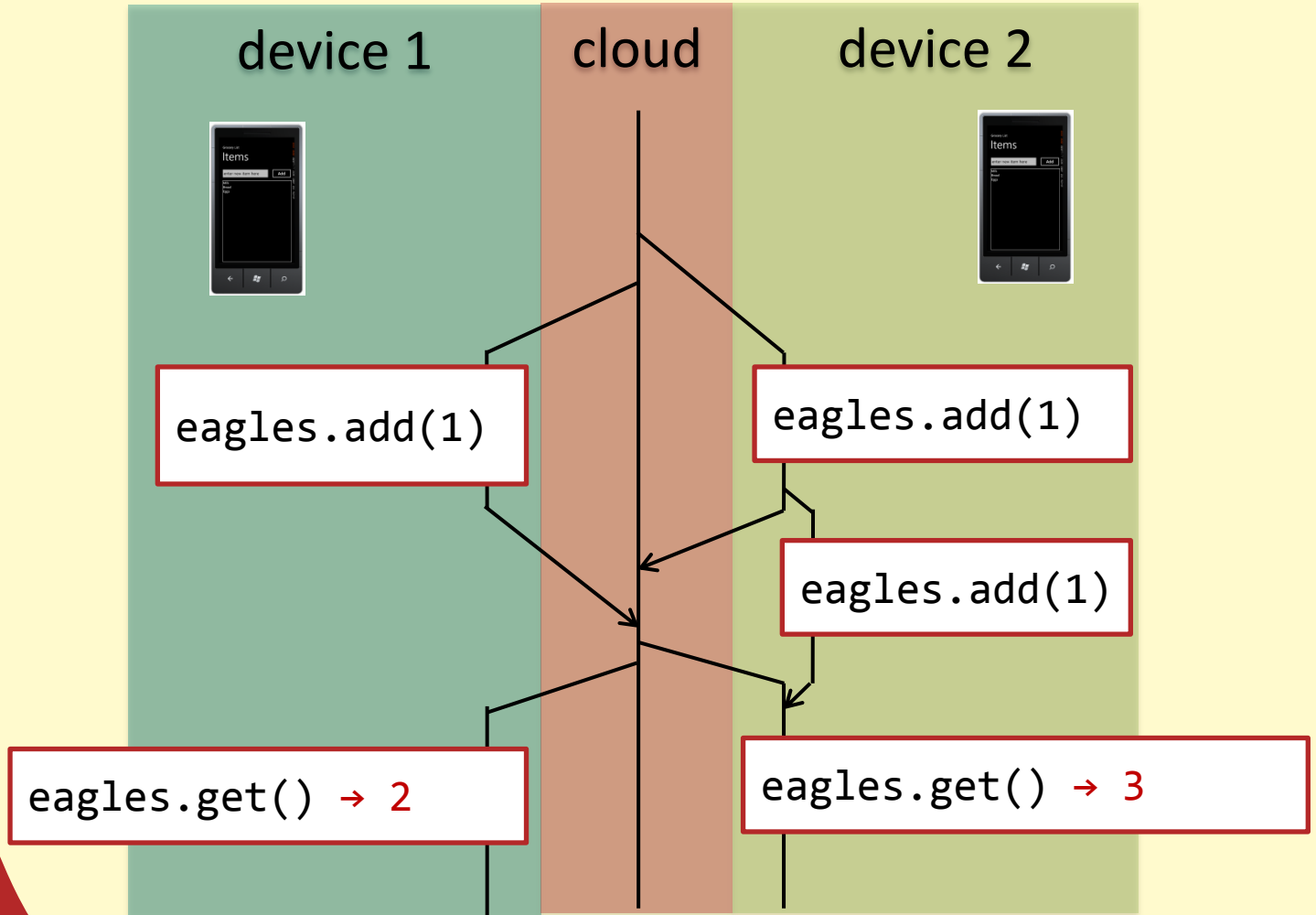
- Start simple:
  let's count the number of eagles seen.

```
var eagles : cloud integer;
```

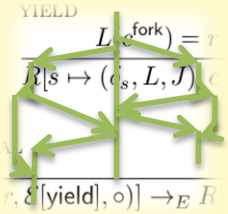
Eventually consistent counting



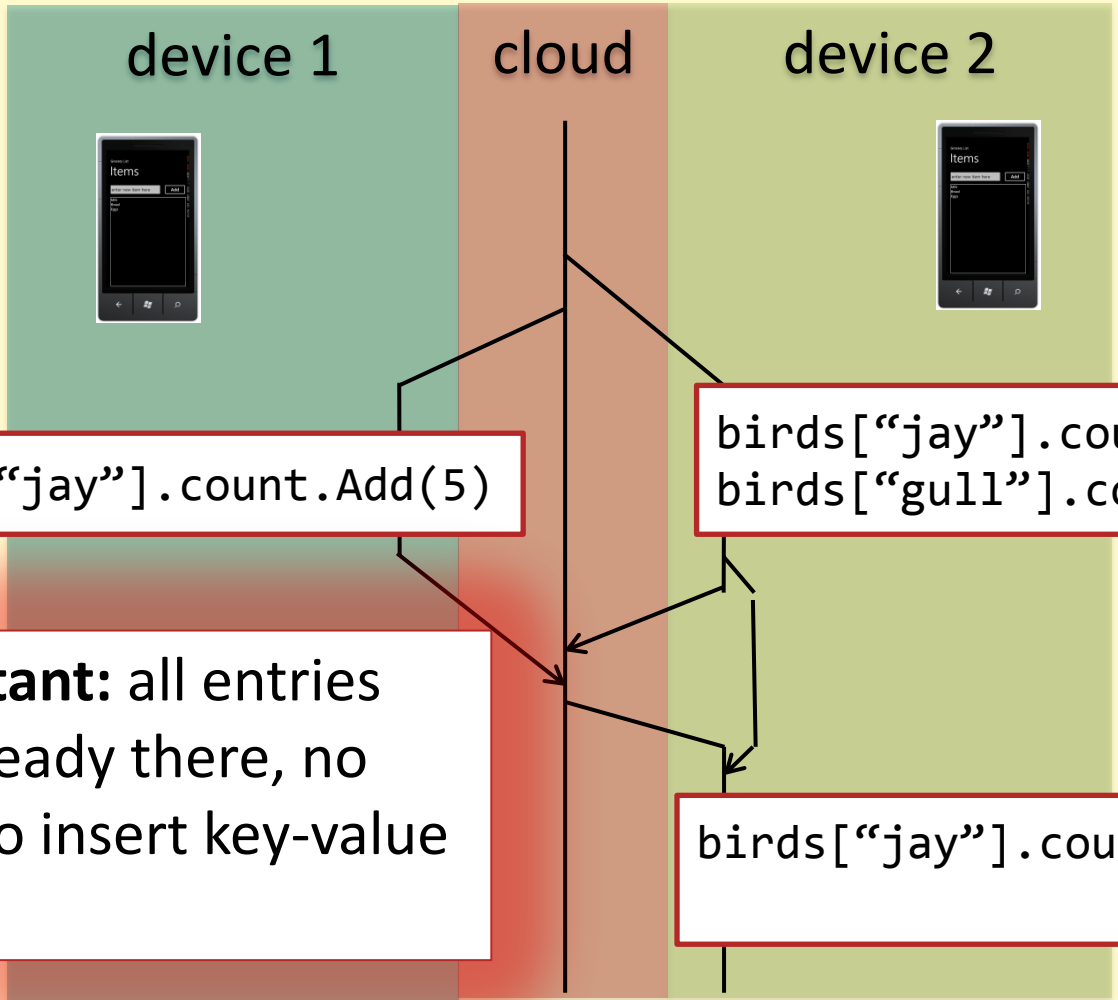
```
var eagles : cloud integer;
```



Counting by bird

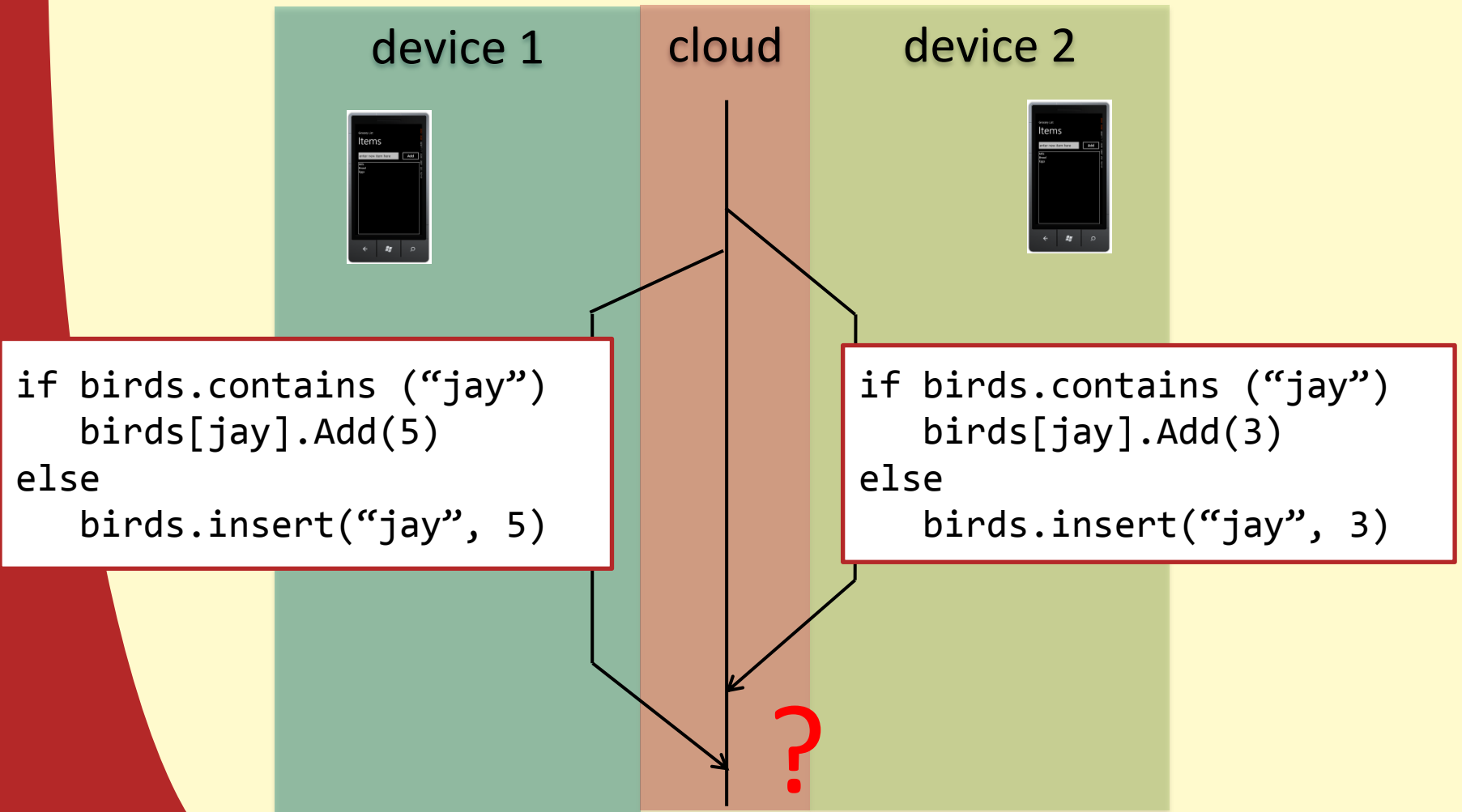


```
var birds: cloud array
           [name: string]
           {count : cloud integer}
```



Important: all entries are already there, no need to insert key-value pairs.

Standard Map Semantics Would not Work!



Arrays + Tables = Relational Data

- Tables
  - Define entities
  - Row identity = Invisible primary key
- Arrays
  - Define relations
- Code can access data using queries
  - For example, LINQ queries

Arrays + Tables = Relational Data

- Example: shopping cart

```
cloud table Customer
{
   name: cloud string;
}

cloud table Product
{
   description: cloud string;
}
```

```
cloud array ShoppingCart
[
   customer: Customer;
   product: Product;
]
{
   quantity: cloud integer;
}
```

Arrays + Tables = Relational Data

- Example: binary relation

```
cloud table User
{
   name: cloud string;
}

cloud array friends
(
   user1 : User;
   user2 : User;
)
{
   value: cloud boolean;
}
```

Standard math: { relations AxBxC } = { functions AxBxC -> bool }

Arrays + Tables = Relational Data

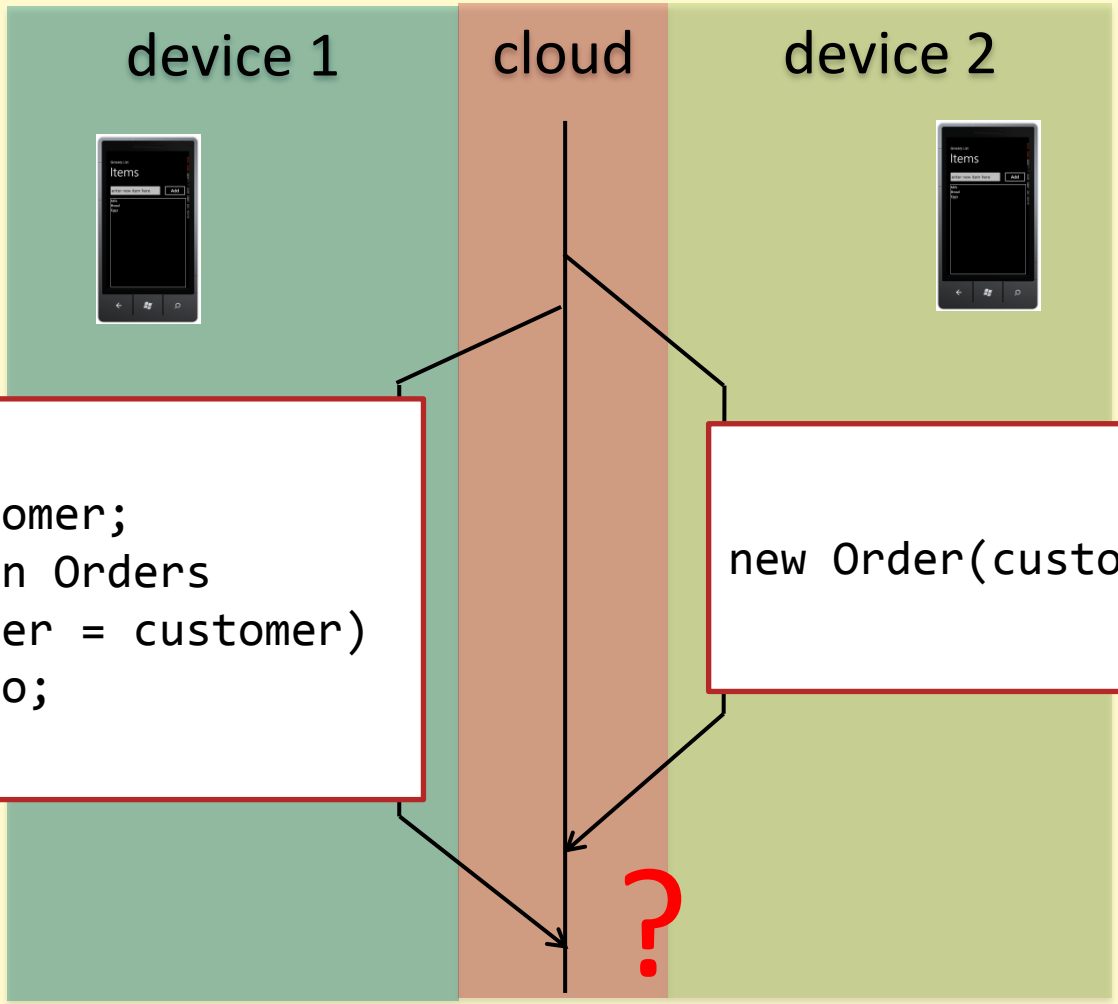
- Example: linked tables

```
cloud table Customer
{
   name: cloud string;
}

cloud table Order
[
   owner: Customer
]
{
   description: cloud string;
}
```

  - Cascading delete: Order is deleted automatically when owning customer is deleted

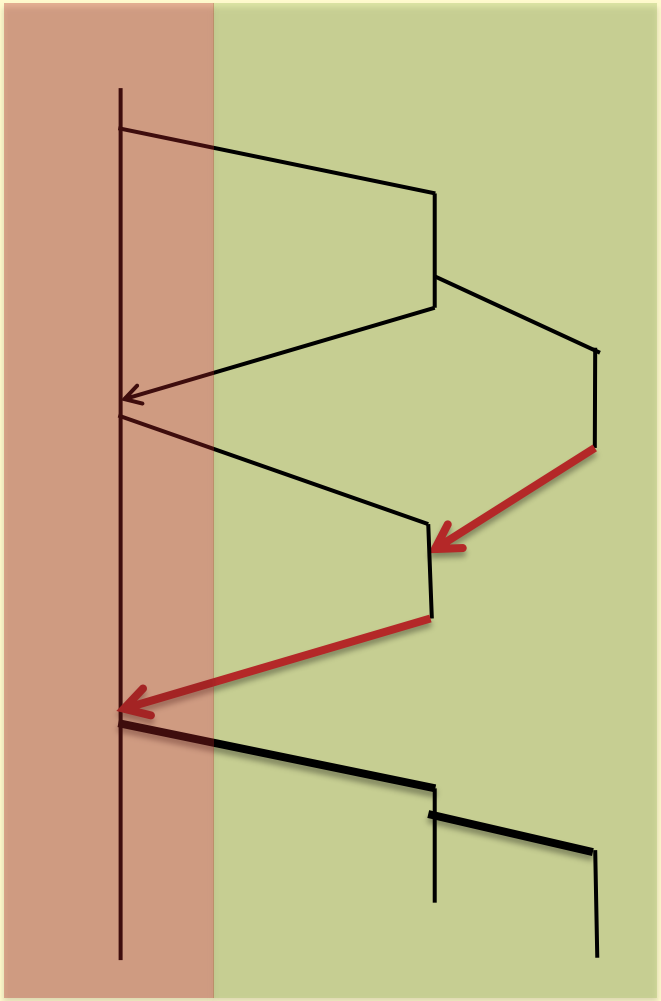
Linked tables solve following problem:




```
delete customer;
foreach o in Orders
  if (o.owner = customer)
    delete o;
```

```
new Order(customer);
```

Recovering stronger consistency



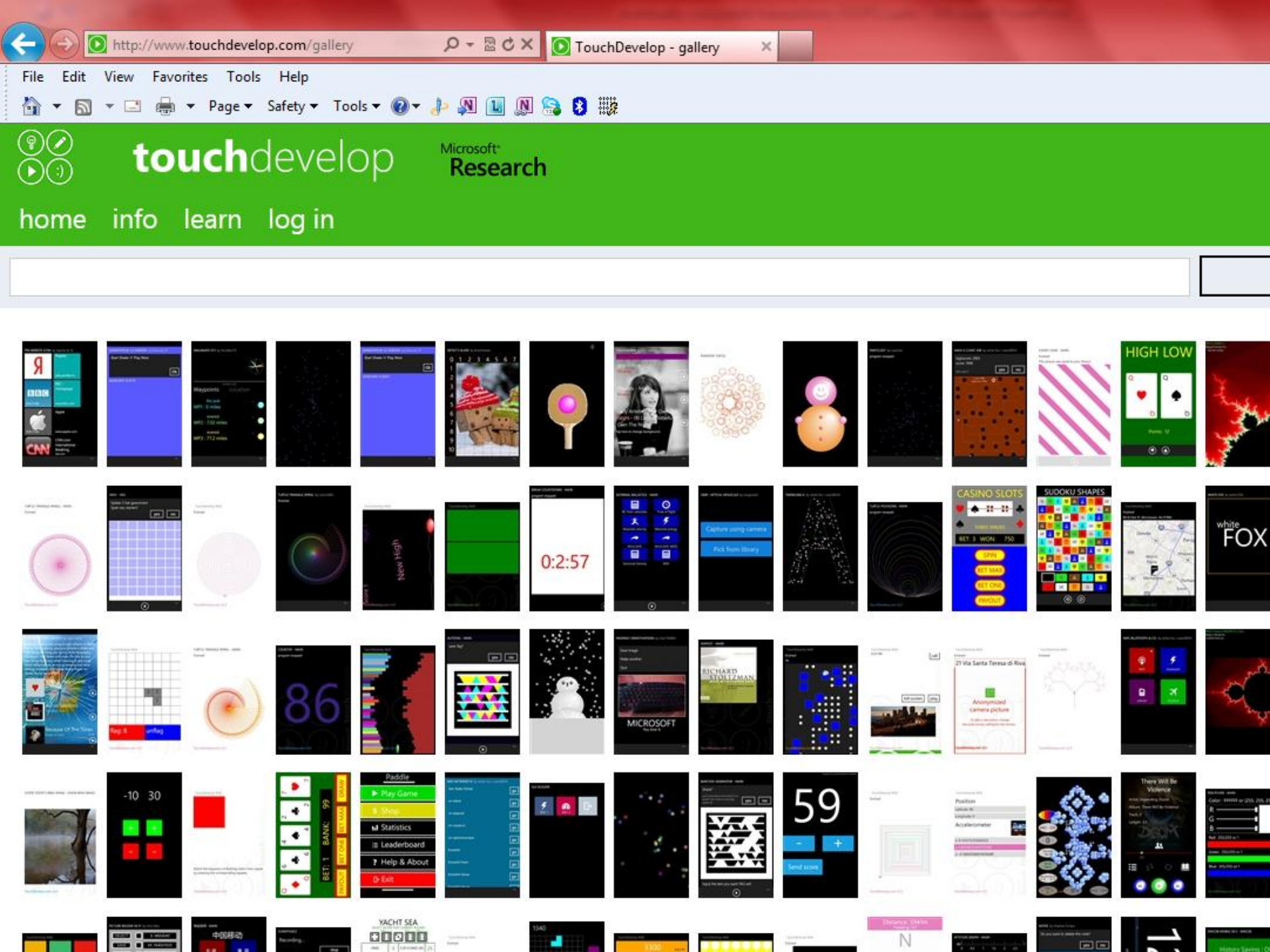
- While connected to server, we may want more certainty
- flush primitive blocks until local state has reached main revision and result has come back to device
- Sufficient to implement strong consistency

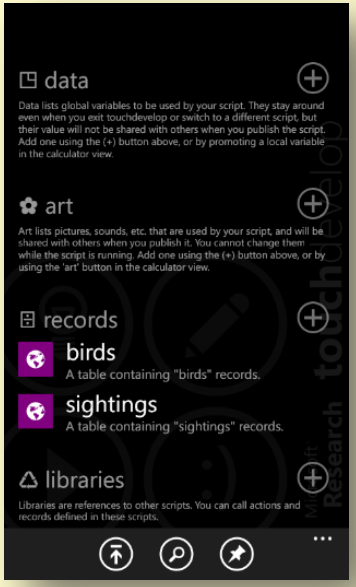
- Claim: this is not too hard. Developers can write correct programs using these primitives.

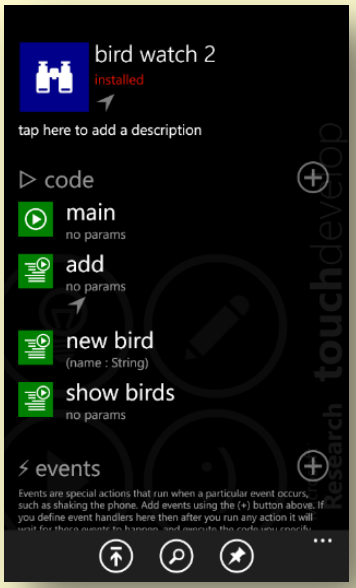
- Future work: evidence?

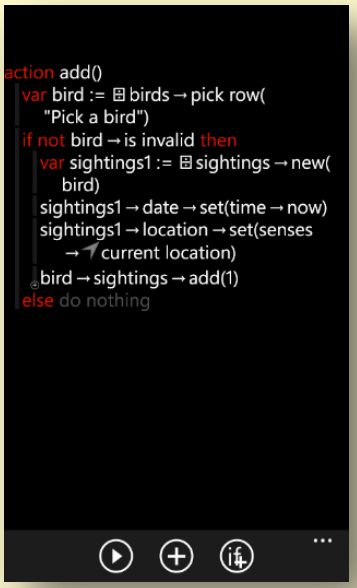
Implementation for TouchDevelop

- Currently working on integration into TouchDevelop Phone-Scripting IDE.
- TouchDevelop: Free app for Windows Phone, with a complete IDE, scripting language, and bazaar.


http://www.touchdevelop.com/gallery
TouchDevelop - gallery
File Edit View Favorites Tools Help
Page Safety Tools
touchdevelop
Microsoft Research
home info learn log in
HIGH LOW
CASINO SLOTS
SUDOKU SHAPES
white FOX
0:2:57
Capture using camera
Pick from library
MICROSOFT
Paddle
Play Game
Shop
Statistics
Leaderboard
Help & About
Exit
There Will Be Violence
YACHT SEA







- Declare cloud types in graphical editor
- Automatic yield
  - Before and after each script execution
  - Between iterations of the event loop

Related Work

- CRDTs (Conflict-Free Replicated Data Types)
  - [Shapiro, Preguica, Baquero, Zawirski]
  - Similar motivation and similar techniques
  - use commutative operations only
  - not clear how to do composition
- Bayou
  - user-defined conflict resolution (merge fcts.)
- Transactional Memory
- Relaxed Memory Models

Conclusion

- eventually consistent shared state is difficult to implement and reason about on traditional platforms.
- revision diagrams [ESOP11],[ESOP12] provide a natural and formally grounded intuition.
- cloud types [ECOOP12] provide a general way to declare eventually consistent storage.