Final Review

May 9, 2017

SQL

A Basic SQL Query

(optional) keyword indicating that the answer should **not** contain duplicates

`SELECT [DISTINCT] target-list`

A list of attributes of relations in relation-list

`FROM relation-list`

A list of relation names
(possibly with a range-variable after each name)

`WHERE condition`

Comparisons (‘=’,‘<>’,‘<‘,‘>’,‘<=’,‘>=’) and other boolean predicates,
combined using AND, OR, and NOT
(a boolean formula)

Integrity Constraints

- Domain Constraints
  - Limitations on valid values of a field.
- Key Constraints
  - A field(s) that must be unique for each row.
- Foreign Key Constraints
  - A field referencing a key of another relation.
  - Can also encode participation/1-many/many-1/1-1.
- Table Constraints
  - More general constraints based on queries.

Algorithms

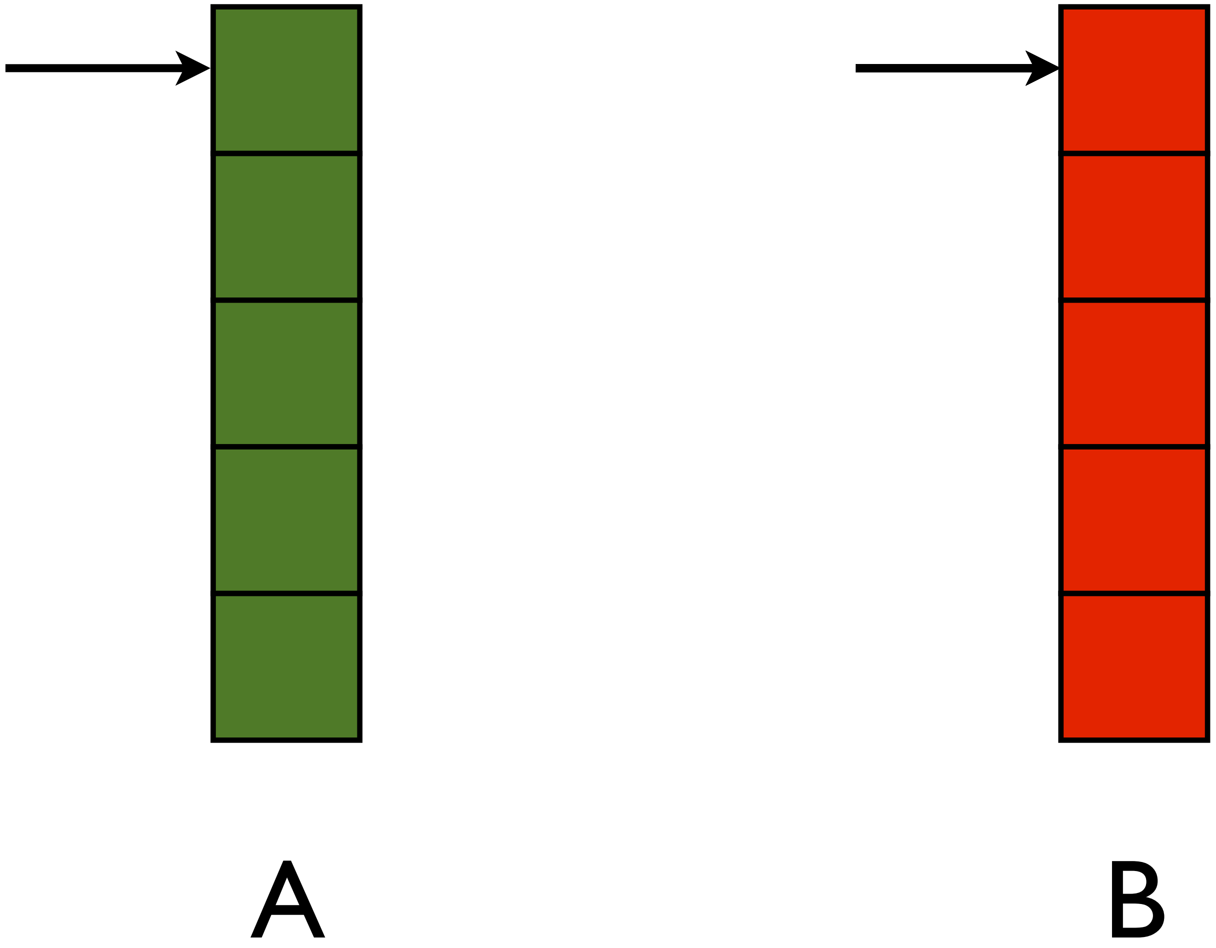
Memory Conscious Algorithms

- Join
  - NLJ has a small working set (but is slow)
- GB Aggregate
  - Working Set ~ # of Groups
- Sort
  - Working Set ~ Size of Relation

Implementing: Joins

Solution 1 (Nested-Loop)

For Each (a in A) { For Each (b in B) { emit (a, b); }}

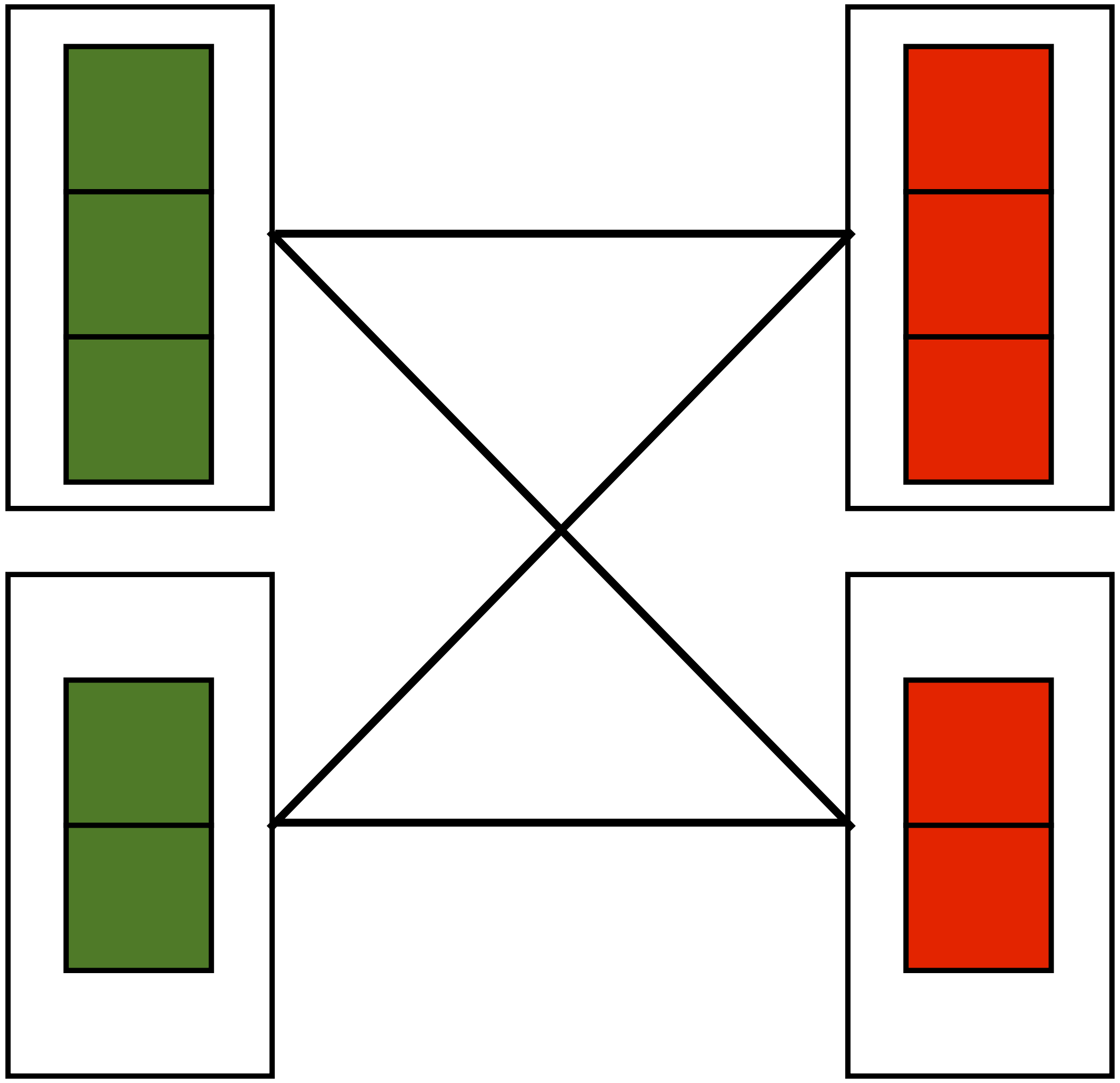


Implementing: Joins

Solution 2 (Block-Nested-Loop)

1) Partition into Blocks

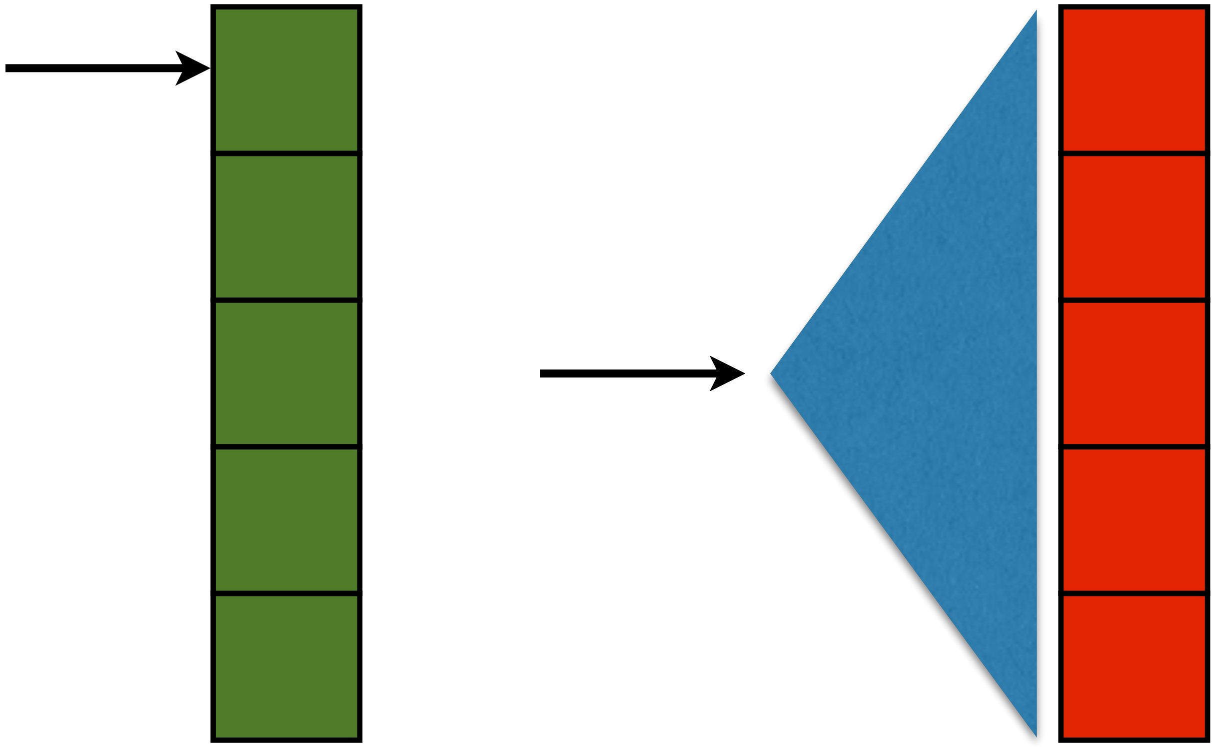
2) NLJ on each pair of blocks



Implementing: Joins

Solution 3 (Index-Nested-Loop)

Like nested-loop, but use an index to make the inner loop much faster!

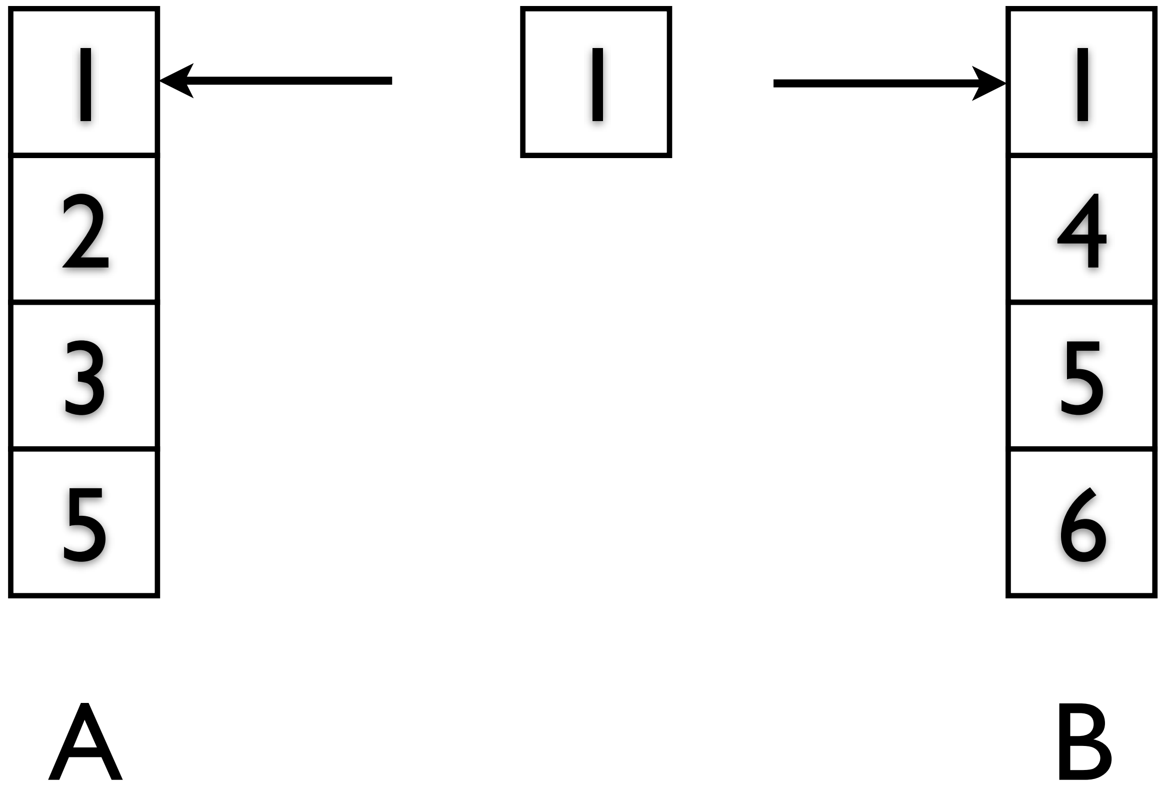


Implementing: Joins

Solution 4 (Sort-Merge Join)

Keep iterating on the set with the lowest value.

When you hit two that match, emit, then iterate both

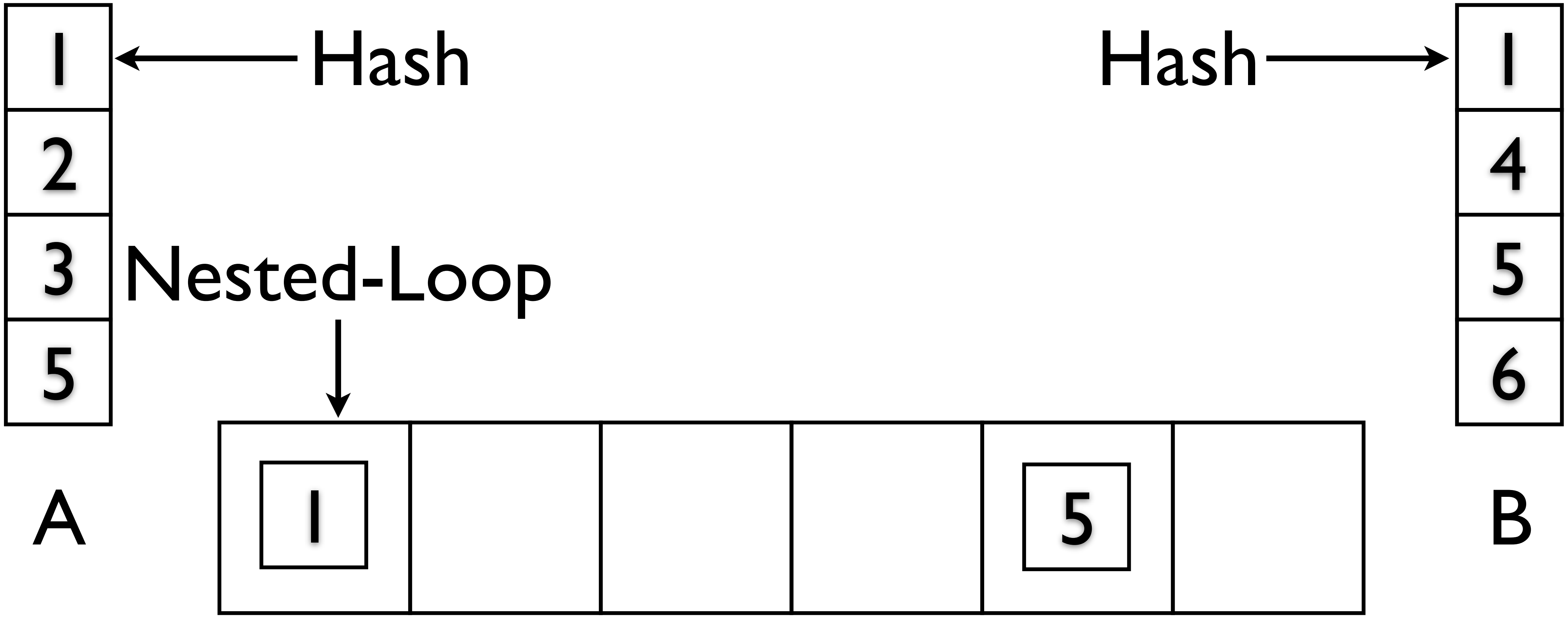


Implementing: Joins

Solution 5 (2-pass Hash)

1) Build a hash table on both relations

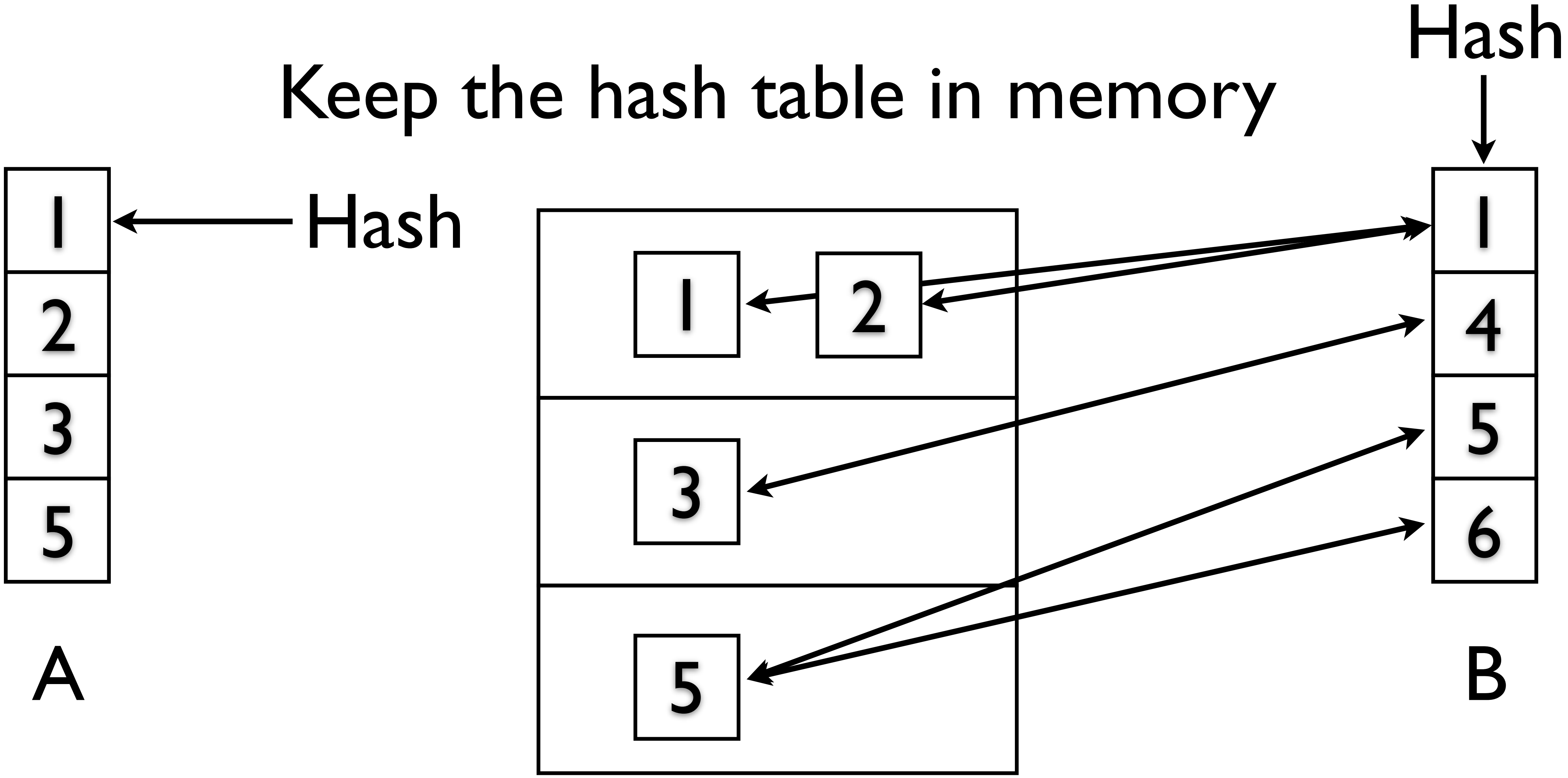
2) In-Memory Nested-Loop Join on each hash bucket



Implementing: Joins

Solution 6 (1-pass Hash)

Keep the hash table in memory



(Essentially a more efficient nested loop join)

Implementing: Joins

Tradeoffs

	Pipelined?	Memory Requirements?	Predicate Limitation?
Nested Loop	1/2	1 Table	No
Block-Nested Loop	No	2 'Blocks'	No
Index-Nested Loop	1/2	1 Tuple (+Index)	Single Comparison
Sort-Merge	If Data Sorted	Same as reqs. of Sorting Inputs	Equality Only
2-pass Hash	No	Max of 1 Page per Bucket and All Pages in Any Bucket	Equality Only
1-pass Hash	1/2	Hash Table	Equality Only

Relational Algebra

RA Equivalencies

Selection

$\sigma_{c_1 \wedge c_2}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(R))$ (Decomposable)

$\sigma_{c_1 \vee c_2}(R) \equiv \delta(\sigma_{c_1}(R) \cup \sigma_{c_2}(R))$ (Decomposable)

$\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$ (Commutative)

Projection

$\pi_a(R) \equiv \pi_a(\pi_{a \cup b}(R))$ (Idempotent)

Cross Product (and Join)

$R \times (S \times T) \equiv (R \times S) \times T$ (Associative)

$(R \times S) \equiv (S \times R)$ (Commutative)

Selection and Projection

$$\pi_a(\sigma_c(R)) \equiv \sigma_c(\pi_a(R))$$

Selection <u>commutes</u> with Projection
(but only if attribute set **a** and condition **c** are *compatible*)

a must include all columns referenced by **c**

Join

$$\sigma_c(R \times S) \equiv R \bowtie_c S$$

Selection combines with Cross Product
to form a Join as per the definition of Join
(Note: This only helps if we have a join algorithm for conditions like **c**)

Selection and Cross Product

$$\sigma_c(R \times S) \equiv (\sigma_c(R) \times S)$$

Selection commutes with Cross Product
(but only if condition **c** references attributes of R exclusively)

Projection and Cross Product

$$\pi_a(R \times S) \equiv (\pi_{a_1}(R)) \times (\pi_{a_2}(S))$$

Projection commutes (distributes) over Cross Product
(where $\mathbf{a_1}$ and $\mathbf{a_2}$ are the attributes in $\mathbf{a}$ from R and S respectively)

RA Equivalencies

Union and Intersections are Commutative and Associative

Selection and Projection both commute with both Union and Intersection

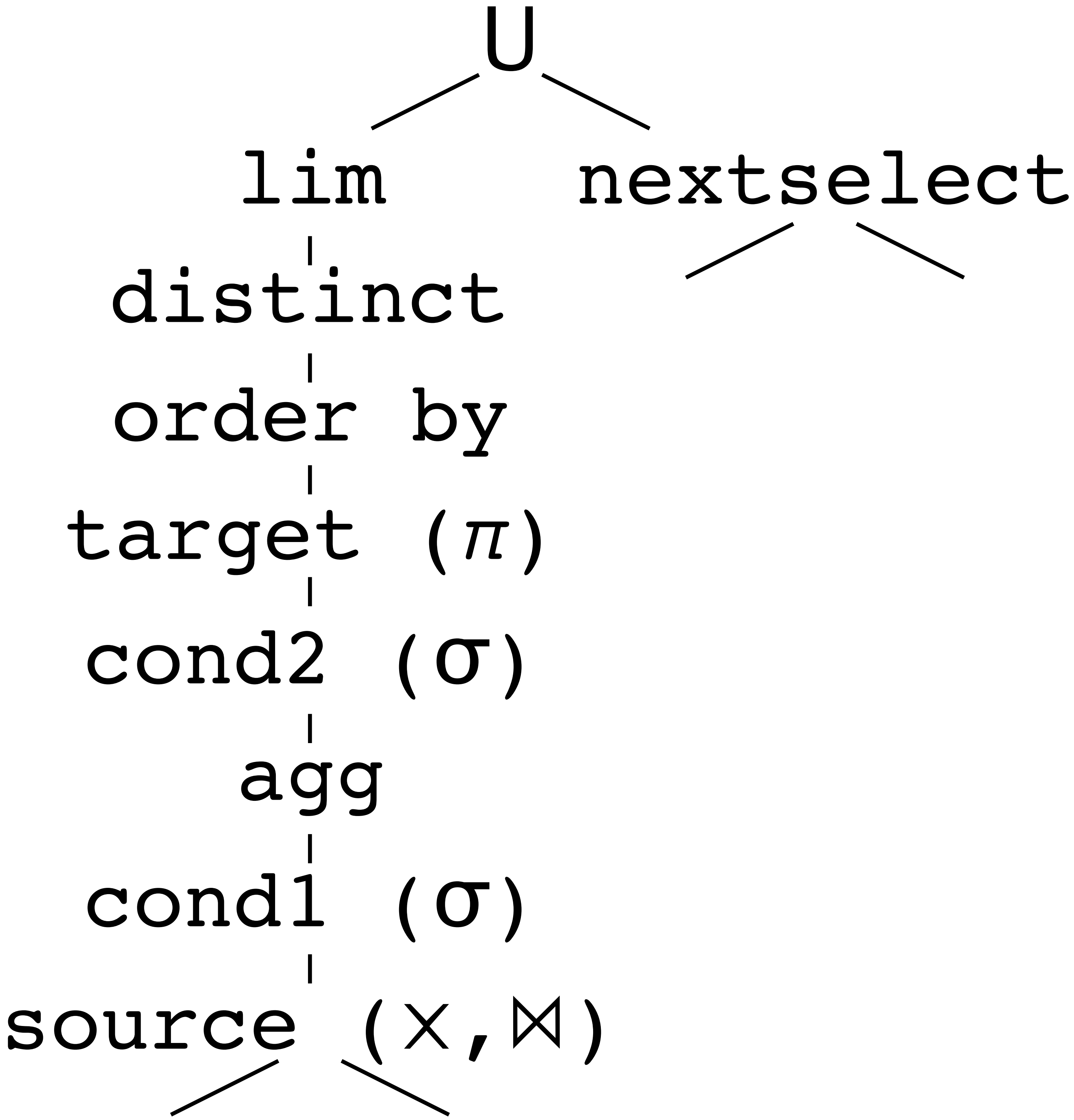
Relational Algebra

Operation	**Sym**	**Meaning**
Selection	σ	Select a subset of the input rows
Projection	π	Delete unwanted columns
Cross-product	x	Combine two relations
Set-difference	-	Tuples in Rel 1, but not Rel 2
Union	U	Tuples either in Rel 1 or in Rel 2

Also: Intersection, **Join**, Division, Renaming (Not essential, but very useful)

SQL to RA

```
SELECT [DISTINCT]
       target
FROM source
WHERE cond1
GROUP BY …
HAVING cond2
ORDER BY order
LIMIT lim
UNION nextselect
```



Transactions

Transaction

What does it mean for a ~~database operation~~ to be correct?

What could go wrong?

Reading uncommitted data

(write-read/WR conflicts; aka “Dirty Reads”)

```
T1: R(A),W(A),              R(B),W(B),ABRT
T2:           R(A),W(A),CMT,
```

Unrepeatable Reads

(read-write/RW conflicts)

```
T1: R(A),              R(A),W(A),CMT
T2:      R(A),W(A),CMT,
```

What could go wrong?

Overwriting Uncommitted Data

(write-write/WW conflicts)

```
T1: W(A),              W(B),CMT
T2:      W(A),W(B),CMT,
```

Schedule

An ordering of read and write operations.

Serial Schedule

No interleaving between transactions **at all**

Serializable Schedule

Guaranteed to produce equivalent output
to a serial schedule

Conflict Equivalence

Possible Solution: Look at read/write, etc… conflicts!

Allow operations to be reordered as long as conflicts are ordered the same way

Conflict Equivalence: Can reorder one schedule into another without reordering conflicts.

Conflict Serializability: Conflict Equivalent to a serial schedule.

Conflict Serializability

- **Step 1:** Serial Schedules are <u>Always Correct</u>
- **Step 2:** Schedules with the same operations and the same conflict ordering are <u>conflict-equivalent</u>.
- **Step 3:** Schedules <u>conflict-equivalent to</u> an always correct schedule are also correct.
  - … or <u>conflict serializable</u>

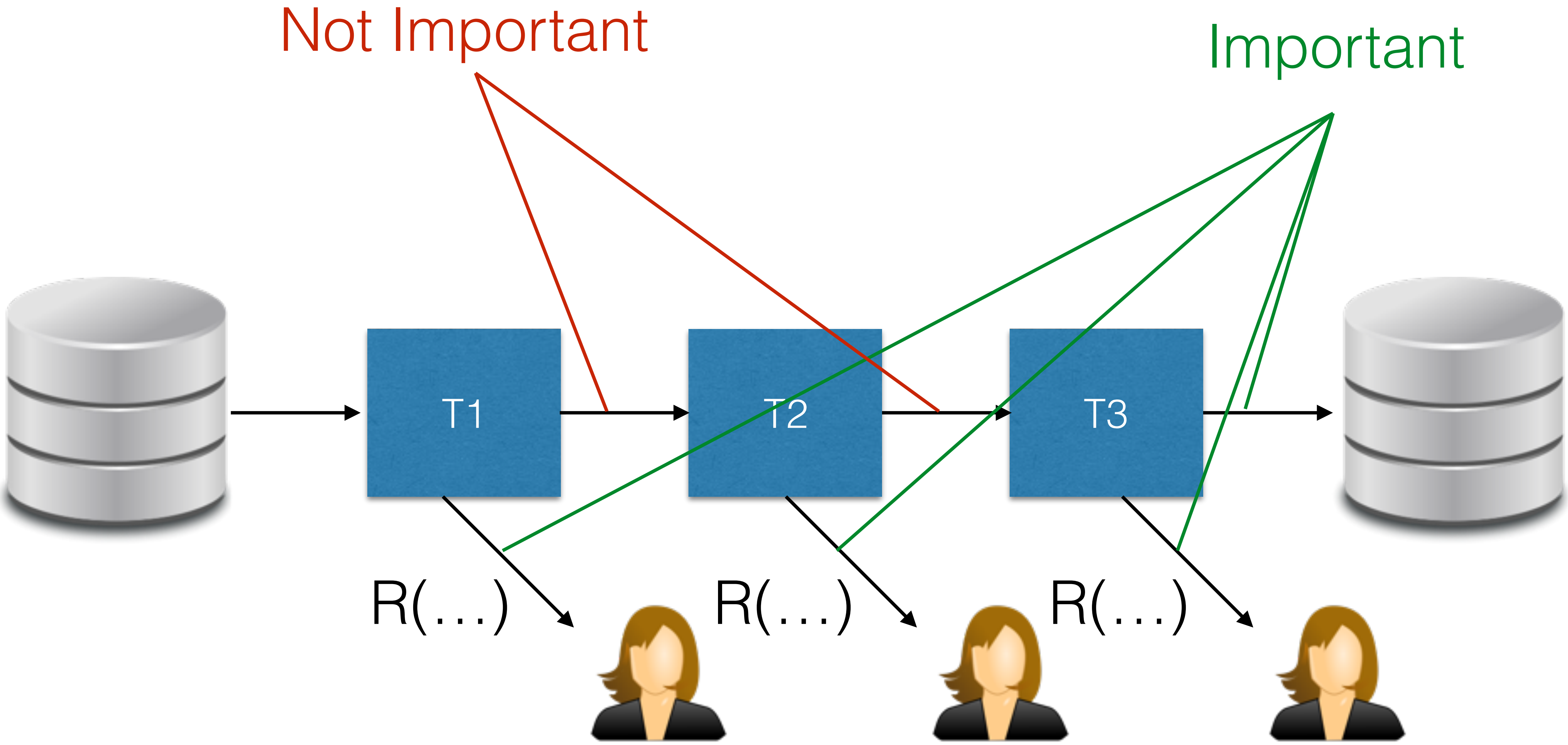
View Serializability

Possible Solution: Look at data flow!

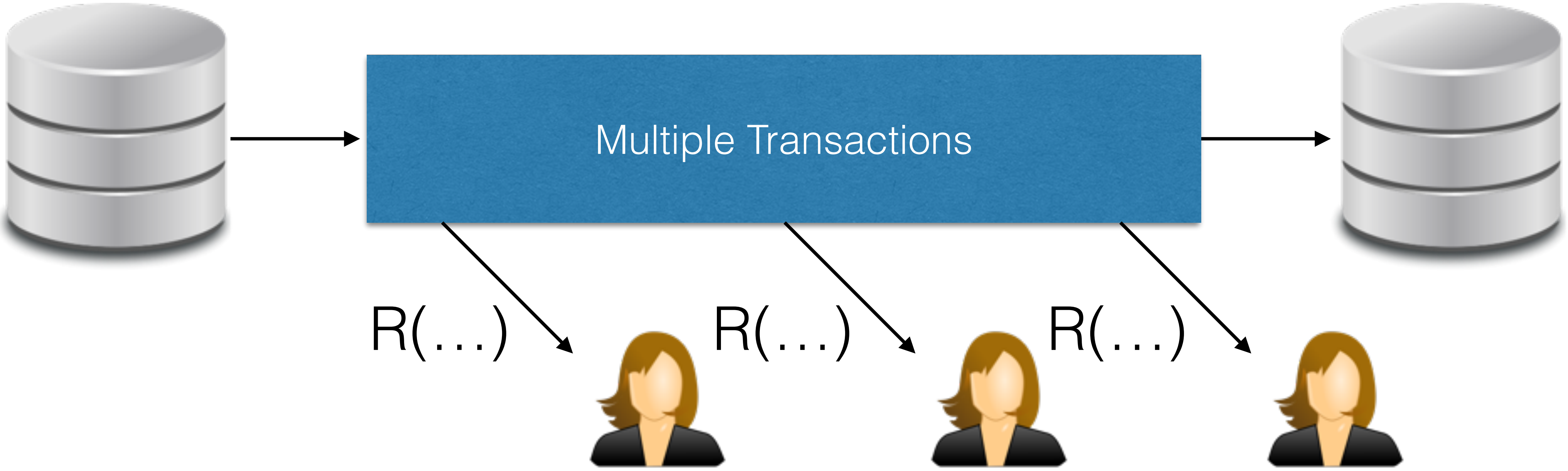
View Equivalence: All reads read from the same writer
Final write in a batch comes from the same writer

View Serializability: Conflict Equivalent to a serial schedule.

Information Flow



Information Flow



Multiple Transactions

R(…)

R(…)

R(…)

View Serializability

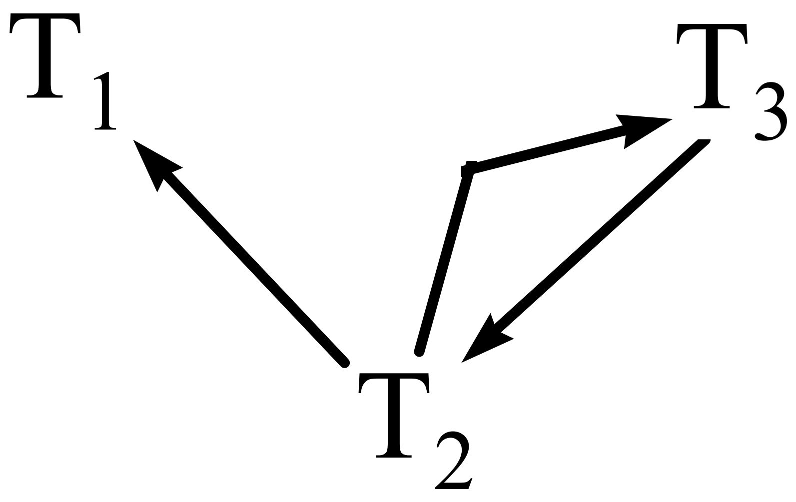
- **Step 1:** Serial Schedules are Always Correct
- **Step 2:** Schedules with the same information flow are view-equivalent.
- **Step 3:** Schedules view-equivalent to an always correct schedule are also correct.
  - … or view serializable

Enforcing Serializability

- Conflict Serializability:
  - Does locking enforce conflict serializability?
- View Serializability
  - Is view serializability stronger, weaker, or incomparable to conflict serializability?
- What do we need to enforce either fully?

How to detect conflict serializable schedule?

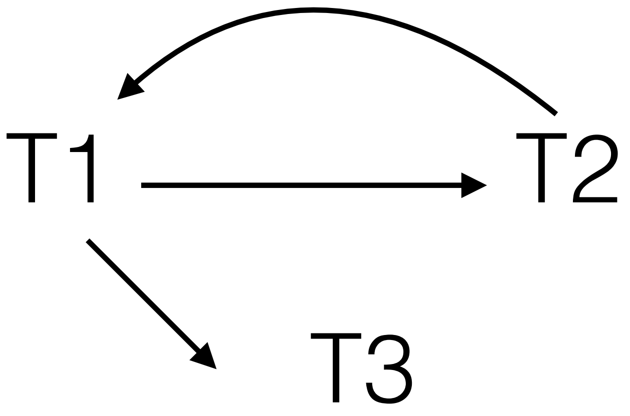
T1	T2	T3
W(a)		
	R(b)	
		W(d)
W(b)		
	R(d)	
		W(d)



Precedence Graph

Cycle!
Not Conflict serializable

Not conflict serializable but view serializable



T1	T2	T3
W(y)		
	W(y)	
	W(x)	
W(x)		
		W(x)

Satisfies 3 conditions of view serializability

Every view serializable schedule which is not conflict serializable has blind writes.

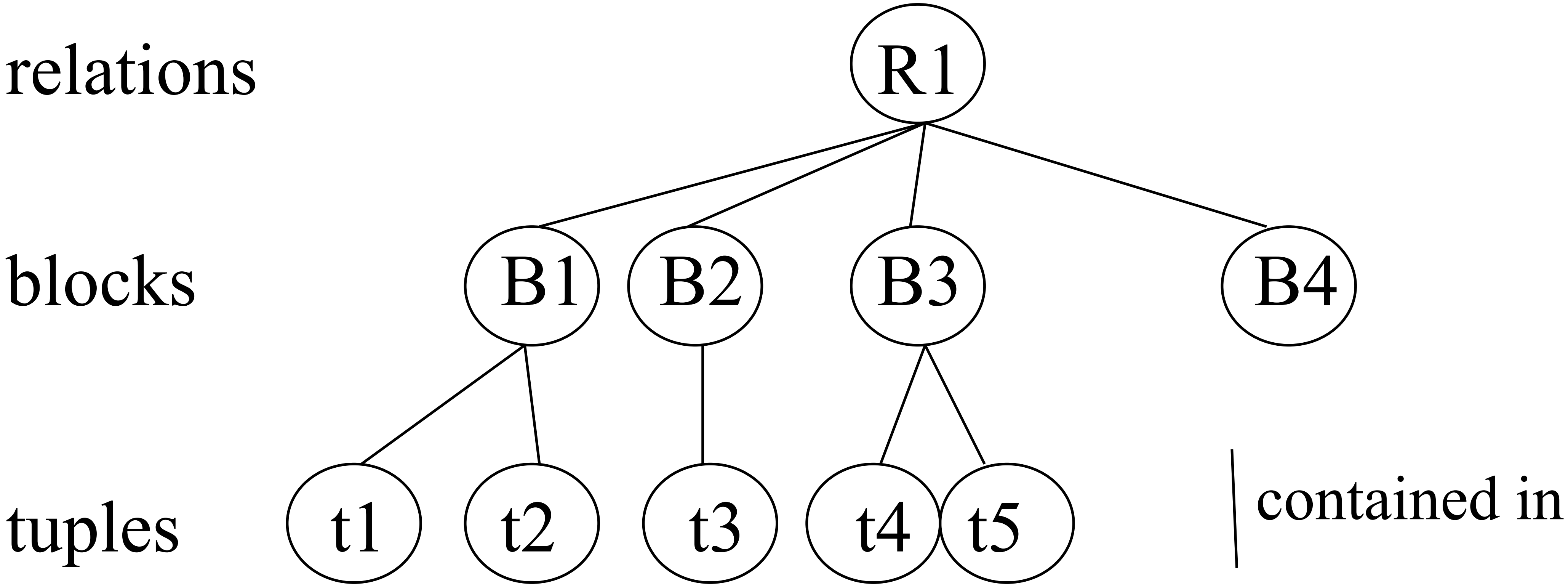
Two-Phase Locking

- Phase 1: Acquire (do not release) locks.
    - *Typically happens as objects are needed.*
- Phase 2: Release (do not acquire) locks.
    - *Typically happens as part of commit.*

Reader/Writer (S/X)

- When accessing a DB Entity…
  - Table, Row, Column, Cell, etc…
- Before reading: Acquire a Shared (S) lock.
  - Any number of transactions can hold S.
- Before writing: Acquire an Exclusive (X) lock.
  - If a transaction holds an X, no other transaction can hold an S or X.

New Lock Modes



Hierarchical Locks

- Lock Objects Top-Down
  - Before acquiring a lock on an object, an xact must have at least an intention lock on its parent!
- For example:
  - To acquire a S on an object, an xact must have an IS, IX on the object’s parent (why not S, SIX, or X?)
  - To acquire an X (or SIX) on an object, an xact must have a SIX, or IX on the object’s parent.

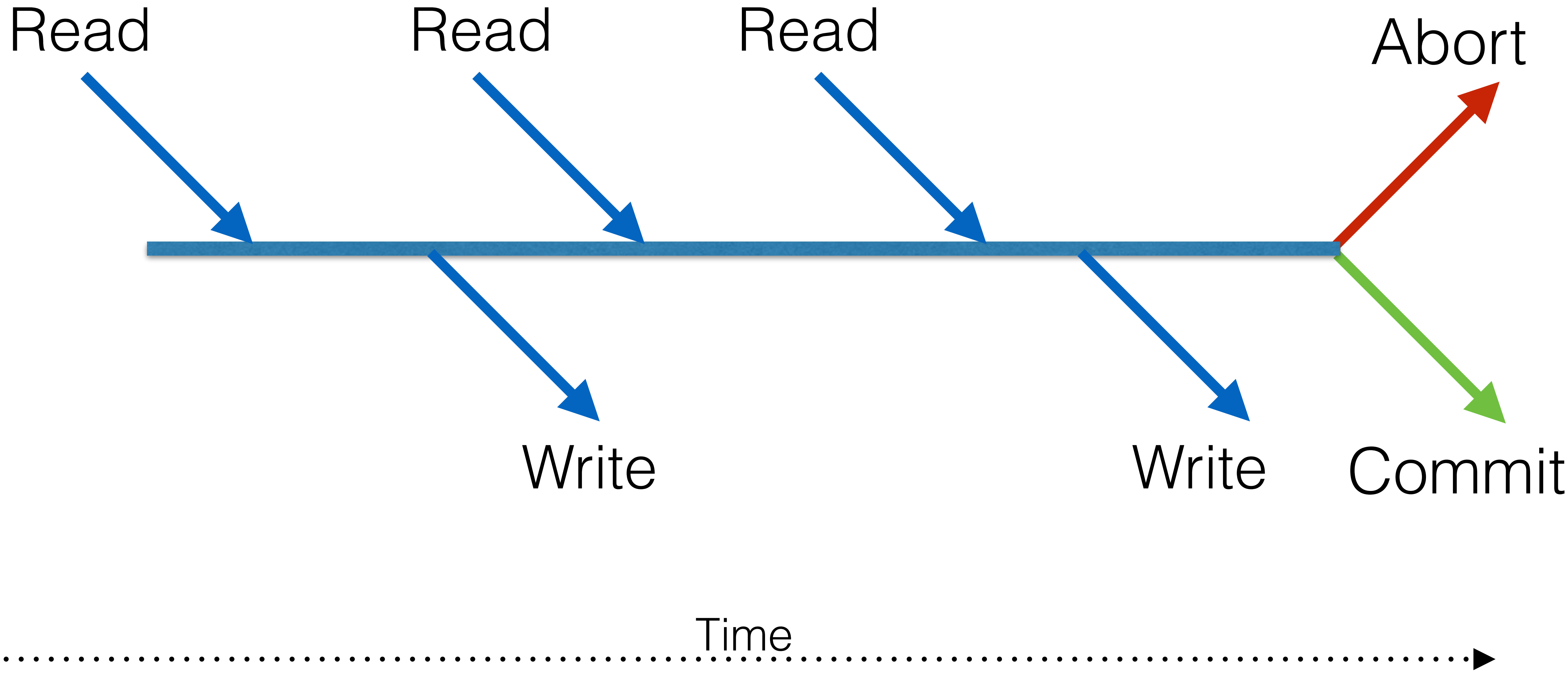
New Lock Modes

Lock Mode(s) Currently Held By Other Xacts

Lock Mode Desired

	None	IS	IX	S	X
None	valid	valid	valid	valid	valid
IS	valid	valid	valid	valid	fail
IX	valid	valid	valid	fail	fail
S	valid	valid	fail	valid	fail
X	valid	fail	fail	fail	fail

Serializability



Read

Read

Read

Abort

Write

Write

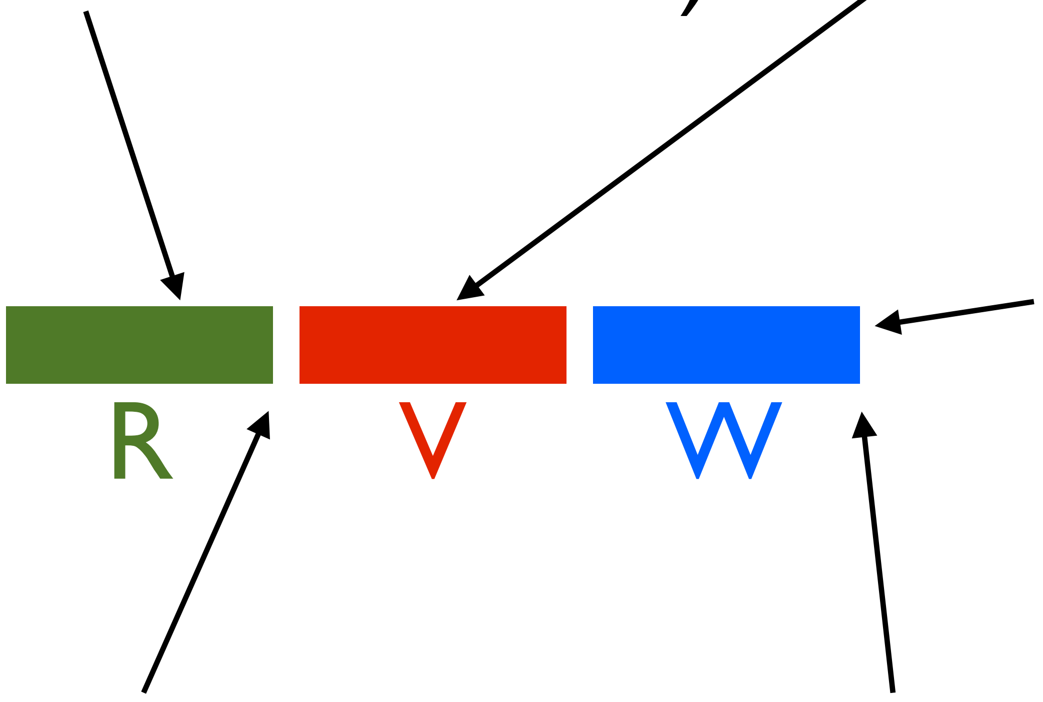
Commit

Time

Optimistic CC

- **Read Phase**: Transaction executes on a <u>private copy</u> of all accessed objects.
- **Validate Phase**: Check for conflicts.
- **Write Phase**: Make the transaction’s changes to updated objects <u>public</u>.

Read, Validate, Write



(1) Transaction executes on a **private copy** of the DB (writes are buffered)

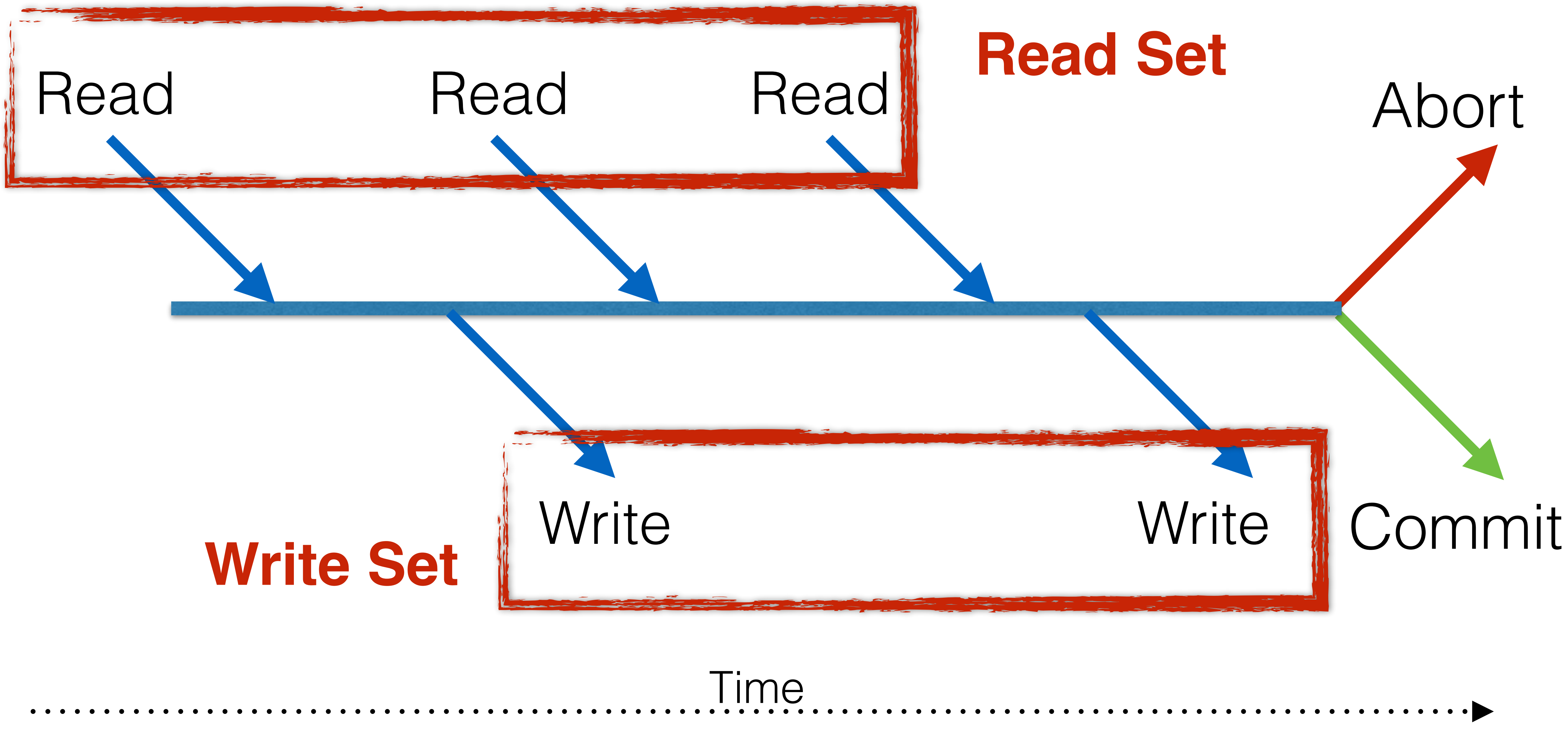
(2) Transaction checks for conflicts

(3) Buffered writes written to main Database

COMMIT Called (user ok with commit)

COMMIT Returns (Commit complete)

Read Phase



Read Read Read

Read Set

Abort

Write Write

Write Set

Commit

Time

Read Phase

ReadSet(T_i): Set of objects read by T_i.

WriteSet(T_i): Set of objects written by T_i.

Validation Phase

Pick a serial order for the transactions
(e.g., assign id #s or timestamps)

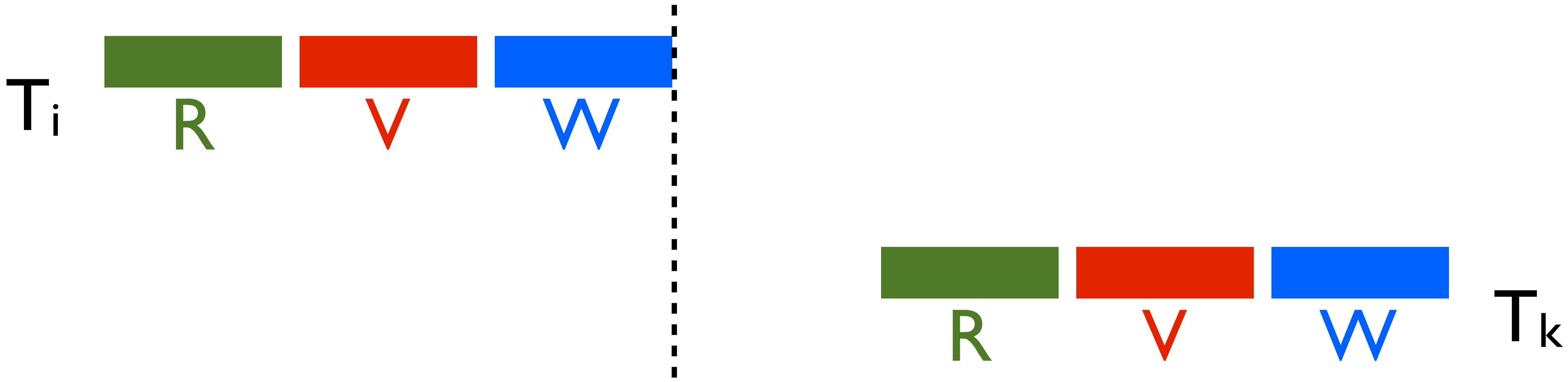
When should we assign Transaction IDs? (Why?)

Validation Phase

What tests are needed?

Simple Test

For all i and k for which i < k,
check that Ti completes before Tk begins.

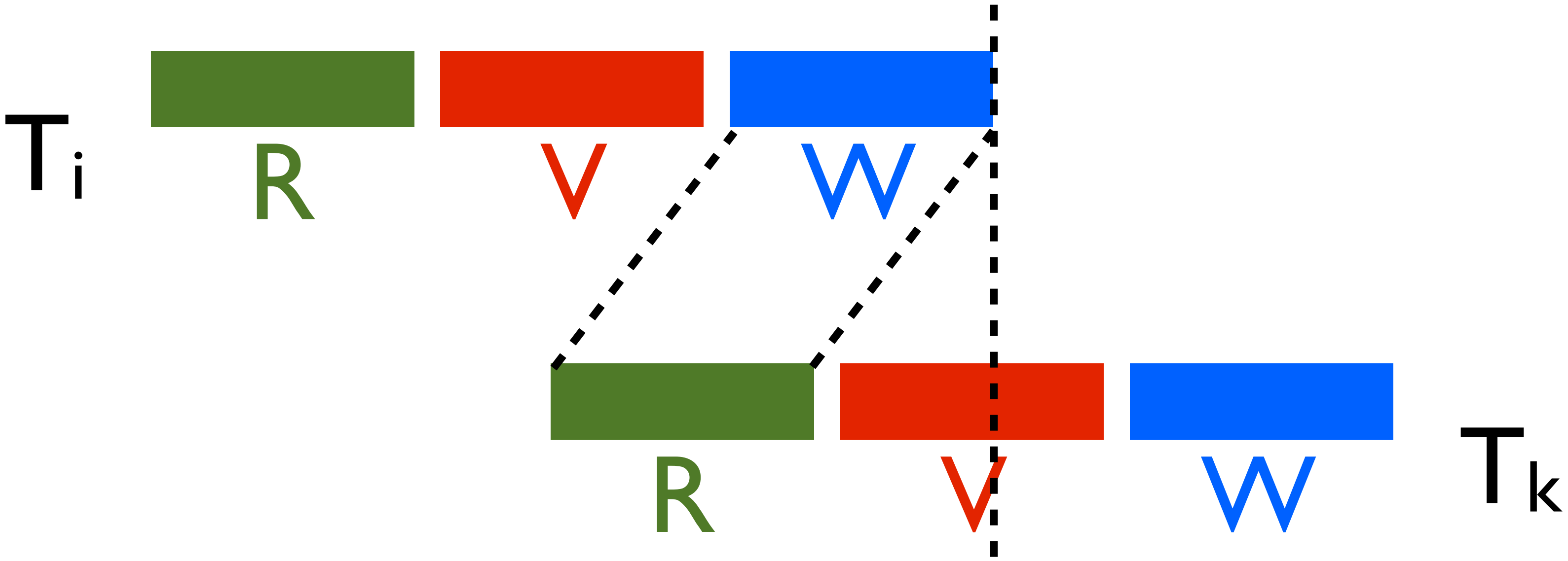


Is this sufficient?

Is this efficient?

Test 2

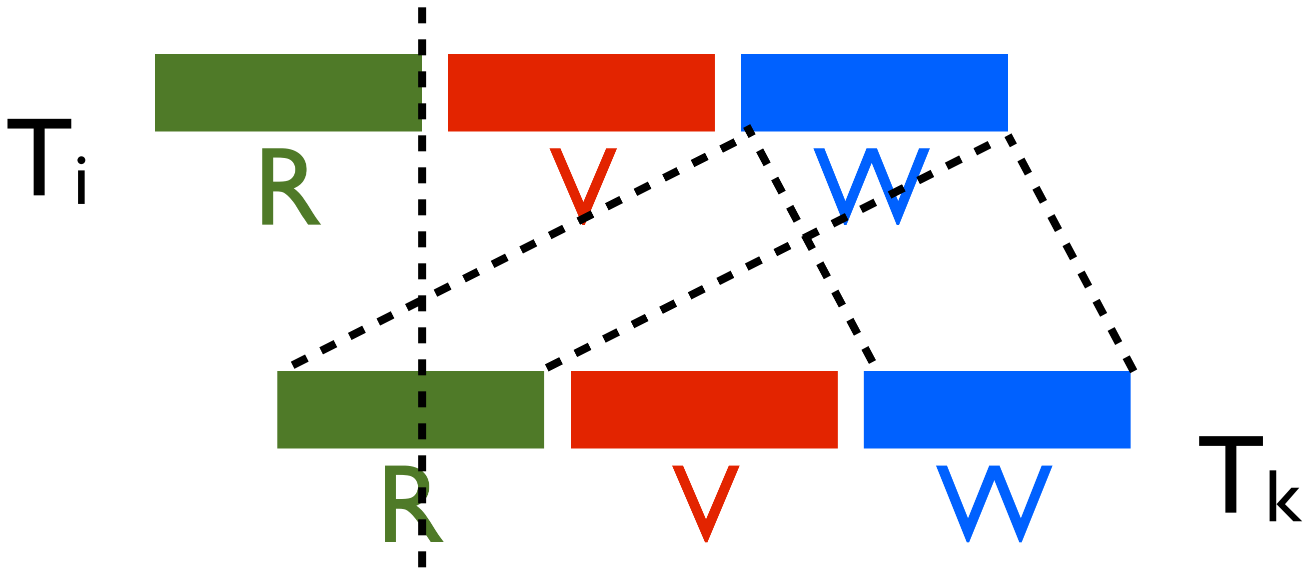
For all i and k for which i < k,
check that Ti completes before Tk begins its write phase
AND WriteSet(Ti) ∩ ReadSet(Tk) is empty



How do these two conditions help?

Test 3

For all i and k for which i < k,
check that Ti completes its read phase first
AND WriteSet(Ti) ∩ ReadSet(Tk) is empty
AND WriteSet(Ti) ∩ WriteSet(Tk) is empty



How do these three conditions help?

Timestamp CC

- Give each object a read timestamp (RTS) and a write timestamp (WTS)
- Give each transaction a timestamp (TS) at the start.
- Use RTS/WTS to track previous operations on the object.
    - Compare with TS to ensure ordering is preserved.

Timestamp CC

- When T_i reads from object O:
  - If WTS(O) > TS(T_i), T_i is reading from a ‘later’ version.
    - Abort Ti and restart with a new timestamp.
  - If WTS(O) < TS(T_i), T_i’s read is safe.
    - Set RTS(O) to MAX(RTS(O), TS(T_i))

Timestamp CC

- When T_i writes to object O:
  - If RTS(O) > TS(T_i), T_i would cause a dirty read.
    - Abort T_i and restart it.
  - If WTS(O) > TS(T_i), T_i would overwrite a ‘later’ value.
    - Don’t need to restart, just ignore the write.
  - Otherwise, allow the write and update WTS(O).

Logging

Write-Ahead Logging

Before writing to the database, first write what you plan to write to a log file…

A	8
B	12
C	5
D	18
E	16

Log

`W(A:10)`

Image copyright: OpenClipart (rg1024)

Write-Ahead Logging

Once the log is safely on disk
you can write the database

A	~~8~~ 10
B	12
C	5
D	18
E	16

Log

`W(A:10)`

Image copyright: OpenClipart (rg1024)

Write-Ahead Logging

Log is append-only,
so writes are always
efficient

Log

```
W(A:10)
W(C:8)
W(E:9)
```

A	~~8~~ 10
B	12
C	5
D	18
E	16

Image copyright: OpenClipart (rg1024)

Write-Ahead Logging

…allowing random writes
to be safely batched

A	~~3~~ 10
B	12
C	~~5~~ 8
D	18
E	~~16~~ 9

Log

```
W(A:10)
W(C:8)
W(E:9)
```

Image copyright: OpenClipart (rg1024)

UNDO Logging

Store both the “old” and the “new” values of the record being replaced

A	~~8~~ 10
B	12
C	~~5~~ 8
D	18
E	~~16~~ 9

Log

```
W(A:8→10)
W(C:5→8)
W(E:16→9)
```

Image copyright: OpenClipart (rg1024)

UNDO Logging

A	~~8~~ 10
B	12
C	~~5~~ 8
D	18
E	~~16~~ 9

Active Xacts

Xact:1, Log: 45

Xact:2, Log: 32

Log

```
43:W(A:8→10)
44:W(C:5→8)
45:W(E:16→9)
```

Image copyright: OpenClipart (rg1024)

UNDO Logging

A	~~8~~ 10
B	12
C	~~5~~ 8
D	18
E	~~16~~ 9

Active Xacts

Xact:1, Log: 45

ABORT

Xact:2, Log: 32

Log

```
43:W(A:8→10)
44:W(C:5→8)
45:W(E:16→9)
```

Image copyright: OpenClipart (rg1024)

UNDO Logging

A	~~8~~ 10
B	12
C	~~5~~ 8
D	18
E	16

Active Xacts

Xact:1, Log: 45

ABORT

Xact:2, Log: 32

Log

```
43: W(A:8→10)
44: W(C:5→8)
45: W(E:16→9)
```

Image copyright: OpenClipart (rg1024)

UNDO Logging

A	~~8~~ 10
B	12
C	5
D	18
E	16

Active Xacts

Xact:3, Log: 45

ABORT

Xact:2, Log: 32

Log

```
43: W(A:8→10)
44: W(C:5→8)
45: W(E:16→9)
```

Image copyright: OpenClipart (rg1024)

UNDO Logging

A	8
B	12
C	5
D	18
E	16

Active Xacts

Xact:1, Log: 45 **ABORT**

Xact:2, Log: 32

Log

```
43: W(A:8→10)
44: W(C:5→8)
45: W(E:16→9)
```

Image copyright: OpenClipart (rg1024)

ACID

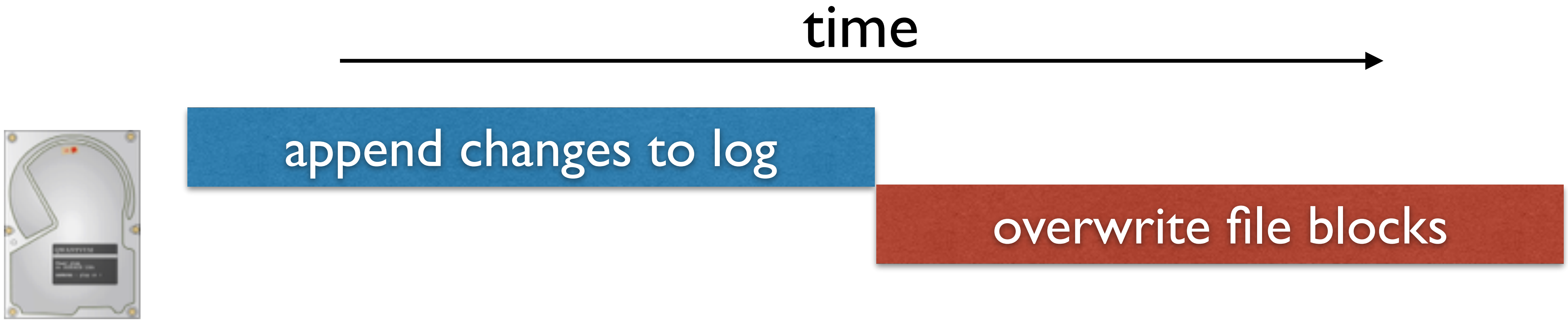
- **Isolation**: Already addressed.
- **Atomicity**: Need writes to get *flushed* in a single step.
  - IOs are only atomic at the page level.

- **Durability**: Need to *buffer* some writes until commit.
  - May need to free up memory for another xact.

- **Consistency**: Need to roll back incomplete xacts.
  - May have already paged back to disk.

Atomicity

- **Problem**: IOs are only atomic for 1 page.
    - What if we crash in between writes?
- **Solution**: Logging (e.g., Journaling Filesystem)
    - Log everything first before you do it.



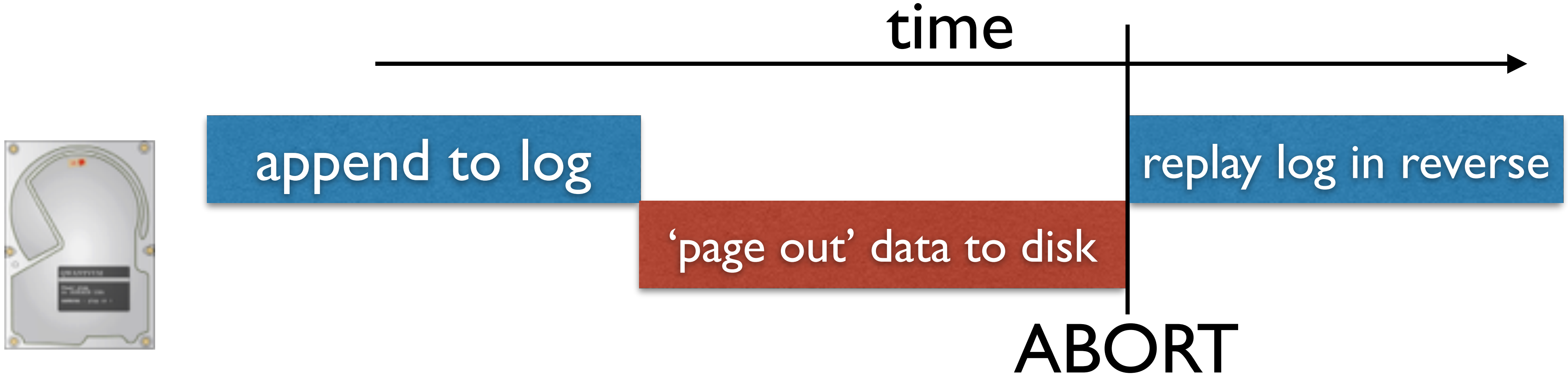
time

append changes to log

overwrite file blocks

Durability / Consistency

- **Problem**: Buffer memory is limited
  - What if we need to ‘page out’ some data?
- **Solution**: Use log (or similar) to recover buffer
  - *Problem*: Commits more expensive
- **Solution**: Modify DB in place, use log to ‘undo’ on abort
  - *Problem*: Aborts more expensive



time

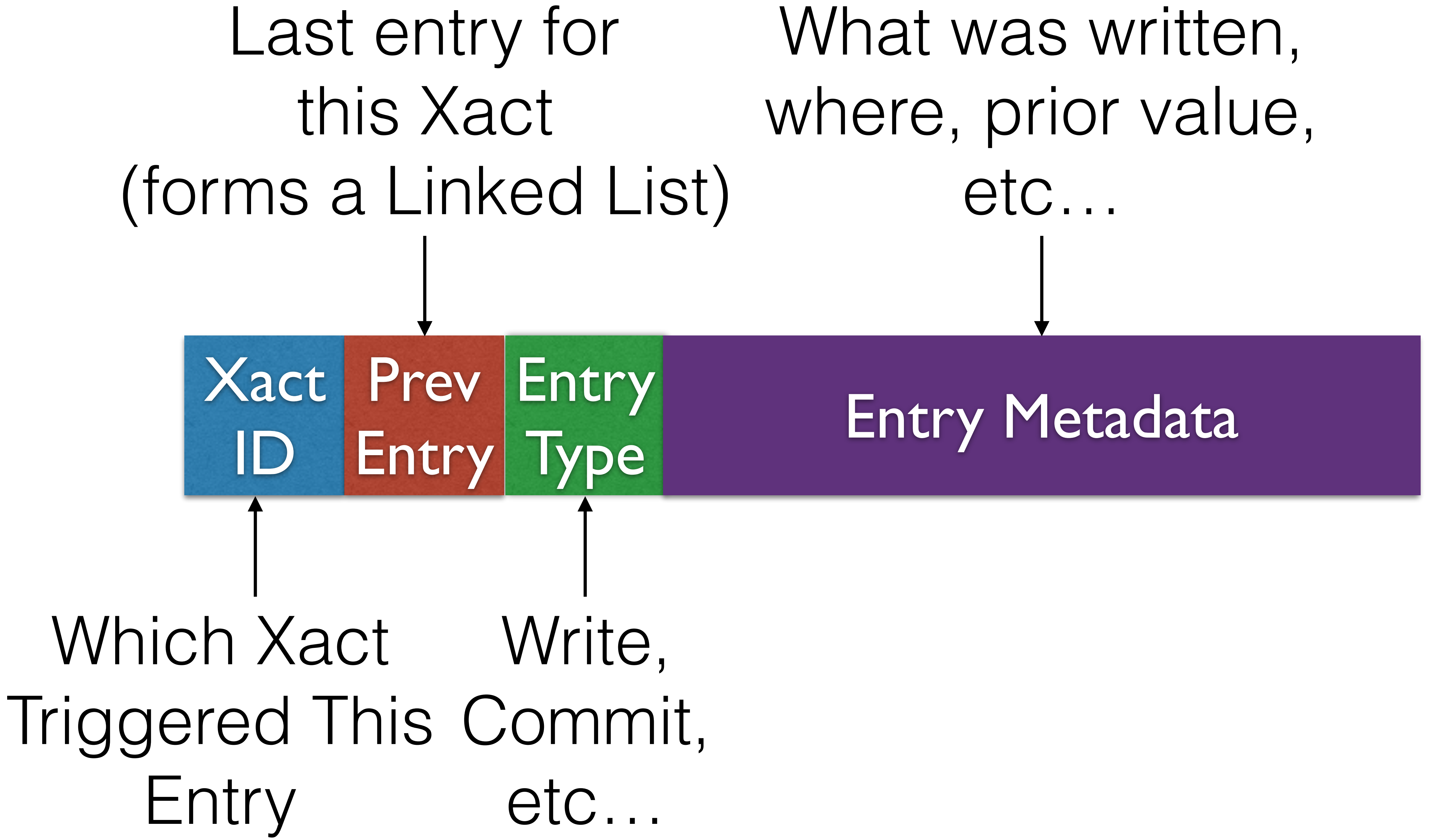
append to log

‘page out’ data to disk

replay log in reverse

ABORT

Anatomy of a log entry



Last entry for this Xact (forms a Linked List)

What was written, where, prior value, etc…

Which Xact Triggered This Entry

Write, Commit, etc…

Transaction Table

Transaction	Status	Last Log Entry
Transaction 24	VALIDATING	99
Transaction 38	COMMITTING	85
Transaction 42	ABORTING	87
Transaction 56	ACTIVE	100

Buffer Manager

Page	Status	Last Log Entry	Data
24	DIRTY	47	01011010…
30	CLEAN	n/a	11001101…
52	DIRTY	107	10100010…
57	DIRTY	87	01001101…
66	CLEAN	n/a	01001011…

Transaction Table

Step 1: Recover Xact State

- **Problem**: We might need to scan to the very beginning of the log to recover the full state of the Xact table (& Buffer Manager)

- **Solution**: Periodically save (checkpoint) the Xact table to the log.

  - Only need to scan the log up to the last (successful) checkpoint.

Checkpointing

- **begin_checkpoint** record indicates when the checkpoint began.
  - Checkpoint covers all log entries before this entry.
- **end_checkpoint** record contains the current transaction table and the dirty page table.
  - Signifies that the checkpoint is now stable.

Buffer Manager

Step 2: Recover Buffered Data

- Where do we get the buffered data from?

Consistency

Step 3: Undo incomplete xacts

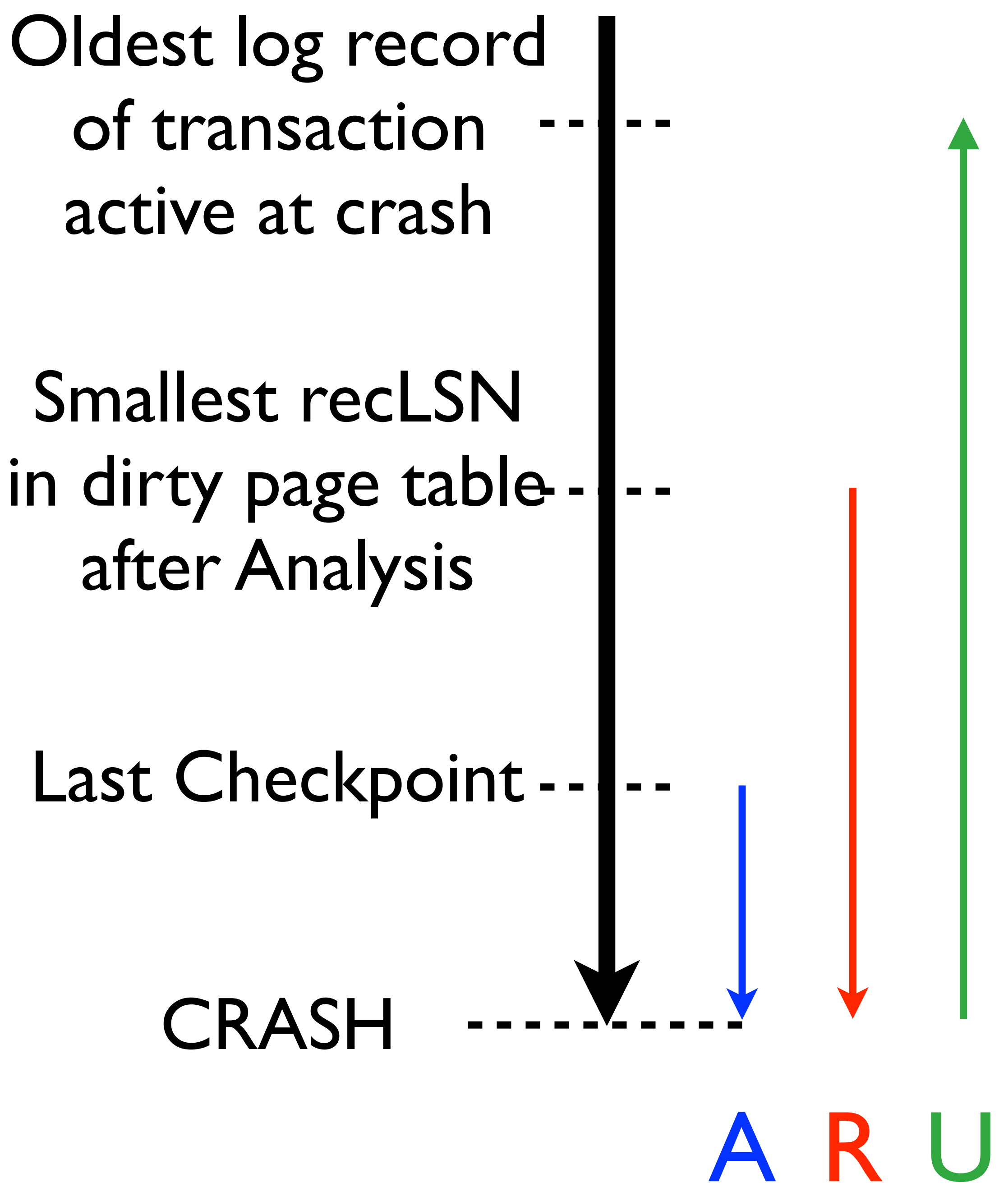
- Record *previous values* with log entries
- Replay log in reverse (linked list of entries)
  - Which Xacts do we undo?
  - Which log entries do we undo?
  - How far in the log do we need to go?

Compensation Log Records

- **Problem**: Step 3 is expensive!
    - What if we crash during step 3?
- **Optimization**: Log undos as writes as they are performed (CLRs).
    - Less repeat computation if we crash during recovery
    - Shifts effort to step 2 (replay)
    - CLRs don’t need to be undone!

ARIES Crash Recovery

- Start from checkpoint stored in master record.
- Analysis: Rebuild the Xact Table
- Redo: Replay operations from all live Xacts (even uncommitted ones).
- Undo: Revert operations from all uncommitted/aborted Xacts.



Materialized Views

Materialized Views

When the base data changes, the view needs to be updated

View Maintenance

`VIEW ← Q(D)`

View Maintenance

```
WHEN D ← D+ΔD DO:
  VIEW ← Q(D+ΔD)
```

Re-evaluating the query from scratch is expensive!

View Maintenance

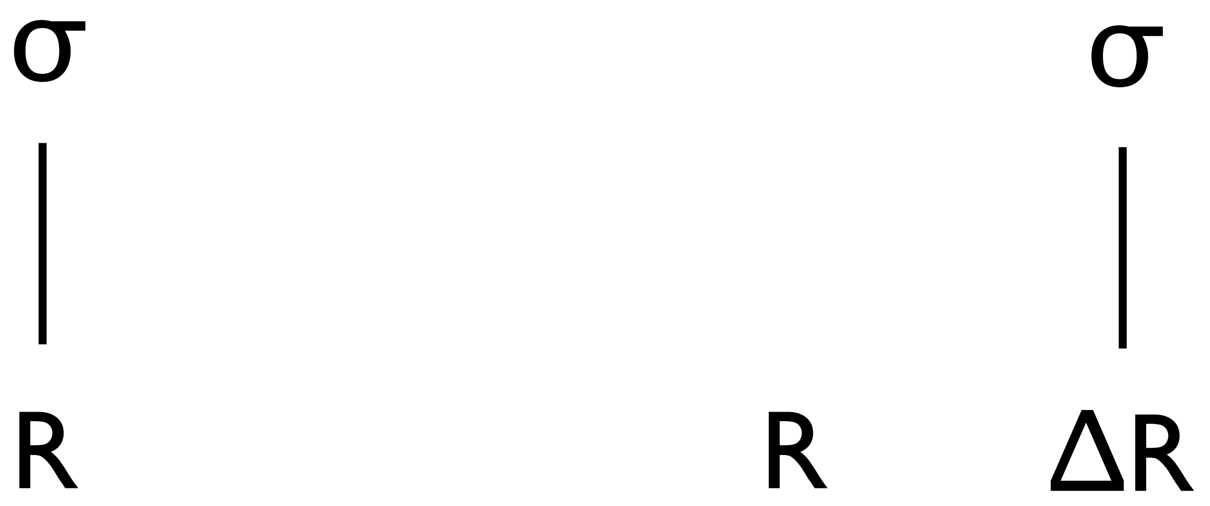
(ideally) Smaller & Faster Query

```
WHEN D ← D+$\Delta$D DO:
  VIEW ← VIEW+$\Delta$Q(D,$\Delta$D)
```

(ideally) Fast “merge” operation.

Delta Queries

$$\Delta(\sigma(R))$$



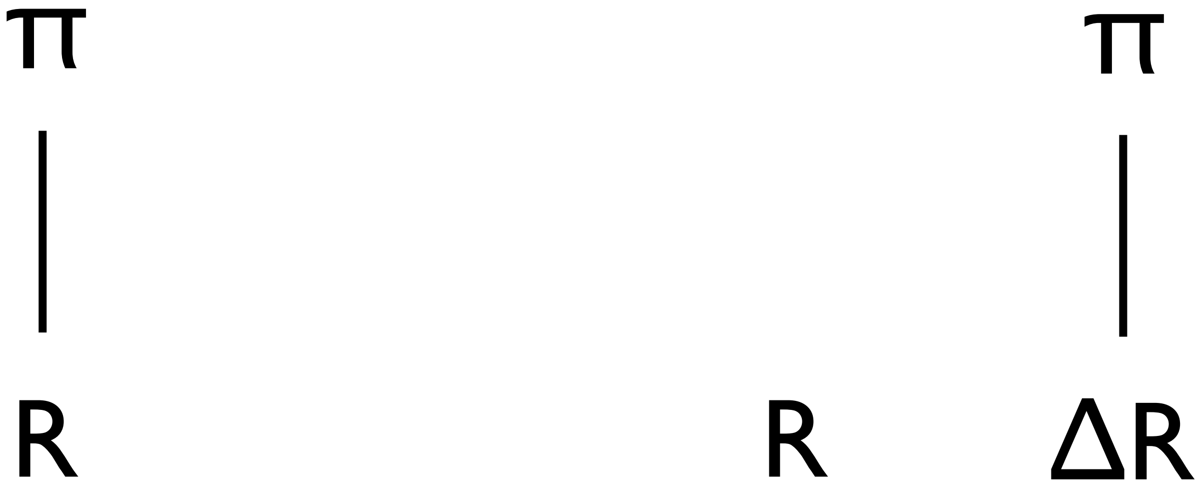
Original R

Inserted Tuples of R

Does this work for deleted tuples?

Delta Queries

$$\Delta(\pi(R)) = \pi(\Delta R)$$

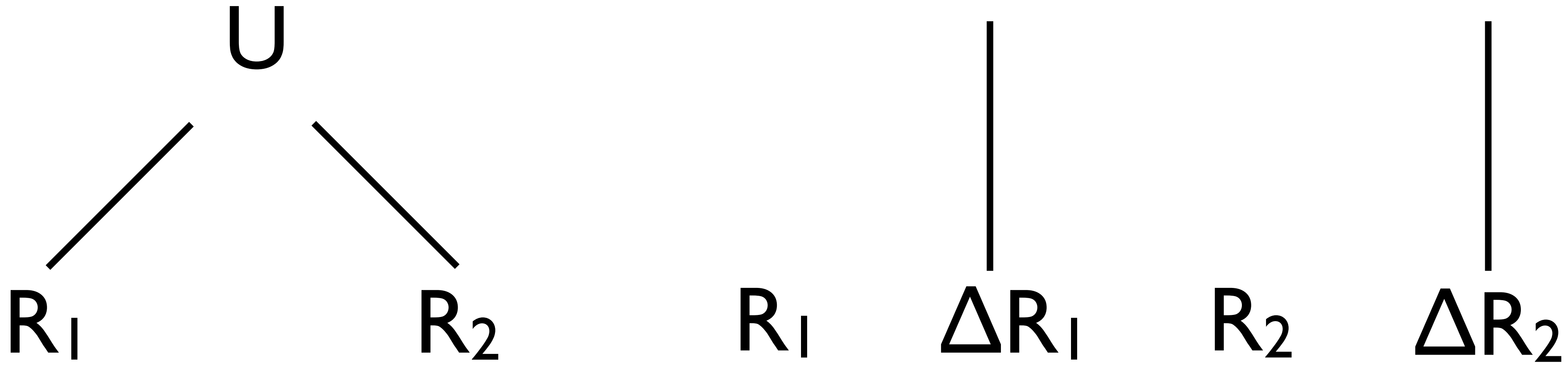

π π

R R ΔR

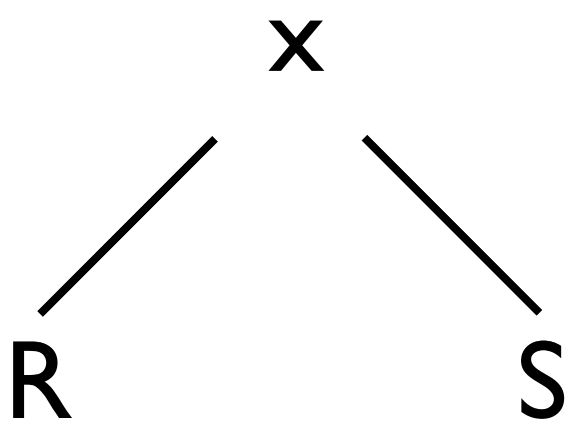
Does this work (completely) under set semantics?

Delta Queries

$$\Delta(R_1 \cup R_2)$$



Delta Queries



R ΔR S

Delta Queries

R : { 1, 2, 3 }   S : { 5, 6}

R x S = { <1,5>, <1, 6>, <2,5>, <2,6>, <3,5>, <3,6> }

$\Delta R_{\text{inserted}} = \{ 4 \}$

$\Delta R_{\text{deleted}} = \{ 3,2 \}$

(R+ΔR) x S = { <1,5>, <1, 6>, **<4,5>, <4,6>** }

$\Delta_{\text{inserted}}(R \times S) = \Delta R_{\text{inserted}} \times S$

$\Delta_{\text{deleted}}(R \times S) = \Delta R_{\text{deleted}} \times S$

What if R and S <u>both</u> change?

Delta Queries

$$(R_1 \cup \Delta R_1) \times (R_2 \cup \Delta R_2)$$

$$\boxed{(R_1 \times R_2)} \cup \boxed{(R_1 \times \Delta R_2) \cup (\Delta R_1 \times R_2) \cup (\Delta R_1 \times \Delta R_2)}$$

The original query

The delta query