Unit 12
NoSQL: Not (Only) SQL Concepts

Characteristics of Some Applications

- A typical application: security trading system

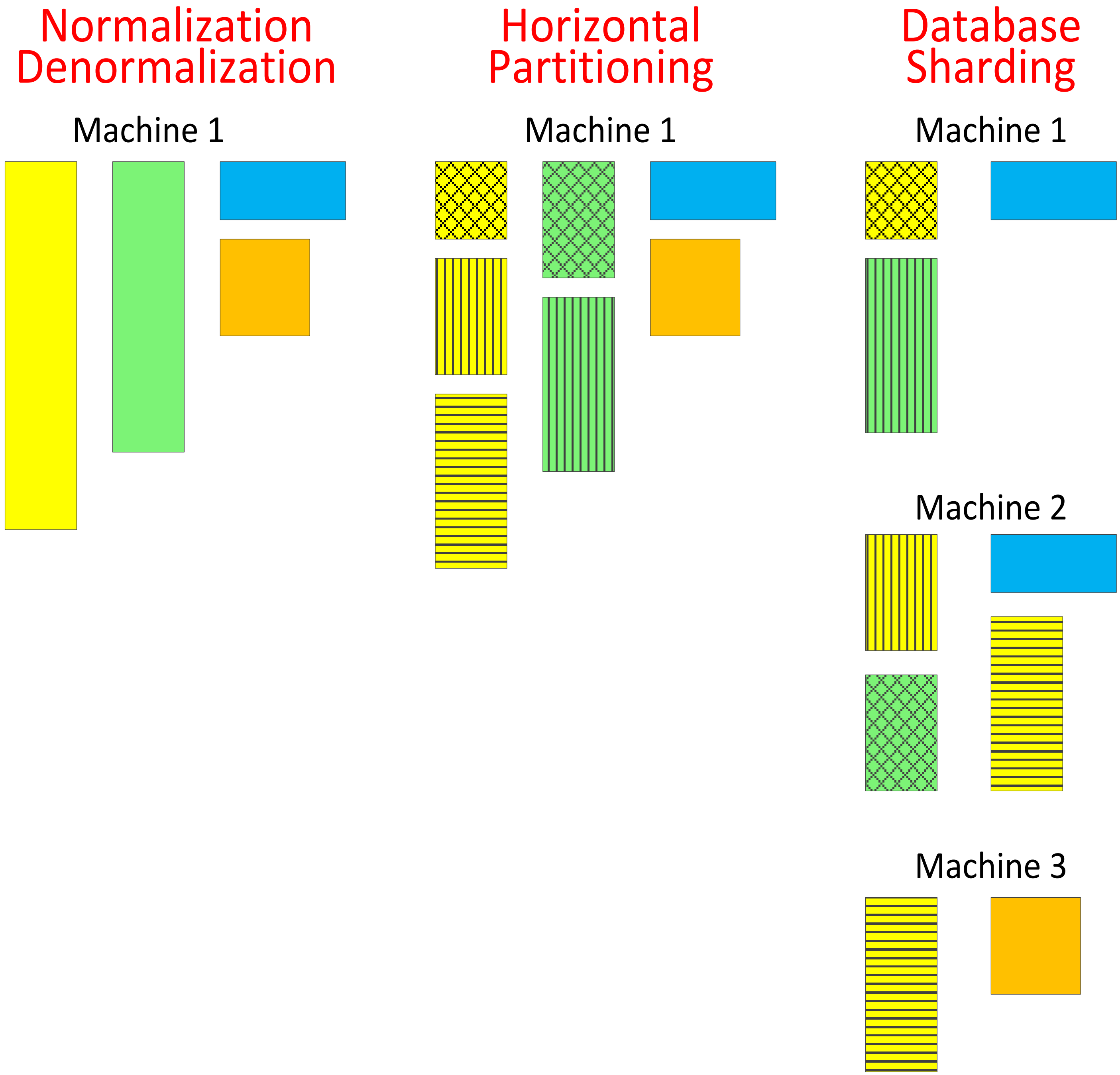
- Fast response
- Fault tolerance
- Fast application development

- Correctness less important for decision making (not execution)

- Run on clusters of machines, so really a distributed database + trading algorithms
- Do not use relational databases: too heavy weight

- We will look at some concepts of distributed databases

Distributing The Data



Collection of Machines Each Running a DBMS

- Each machine runs some DBMS, not necessarily a relational database system
- But each has some version of
  - Physical Implementation: file system, indexes, …
  - Query Processor
  - Recovery Mechanism
  - Concurrency Mechanism
- The new issue: ***coordinate the concurrent execution of several machines***

Issues to Revisit

- ACID properties
- Query execution planning

- We will talk very briefly about
  - Recovery
  - Concurrency
  - Query execution planning

Recovery

Global Recovery

- We have a local recovery manager on each machine
- It is able to guarantee
  - A: Atomicity
  - C: Consistency
  - D: Durability

  for transactions executing on the machine

- We need to guarantee ACD for transactions that run on more than one machine
- So for example, such a ***transaction must be either committed or aborted globally***, that is the work on each machine must be either committed or aborted (rolled back)

Our Old Example: Money Transfer

- Items a and b are stored on ***a disk attached to some machine*** running a DBMS
- Transfer $5 from account a to b

1. transaction starts
2. read a into xa (local variable in RAM)
3. xa := xa − 5
4. write xa onto a
5. read b into xb (local variable in RAM)
6. xb := xb + 5
7. write xb onto b
8. transaction ends

- If initial values are a = 8 and b = 1

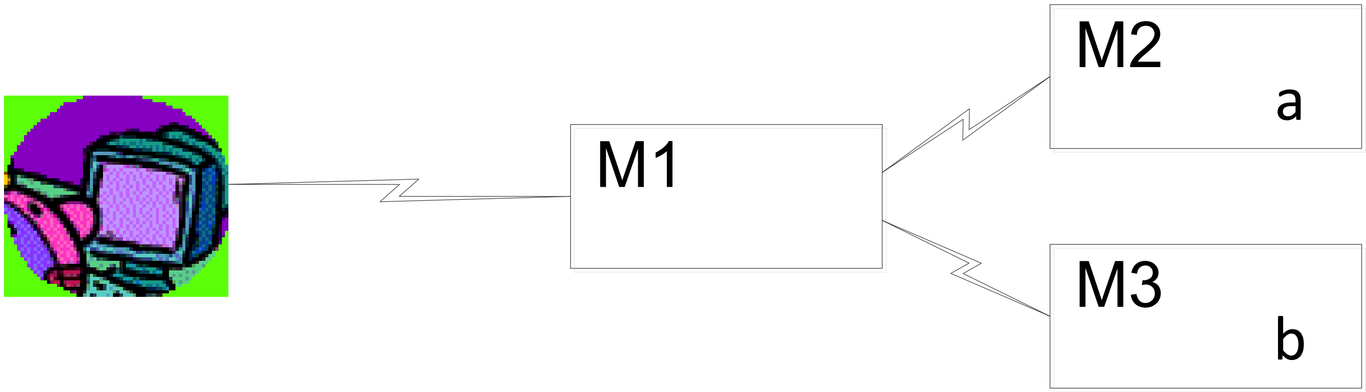
  then after the execution a = 3 and b = 6

Old Example: New Scenario

- There are 3 DBMS machines: nodes in a cluster

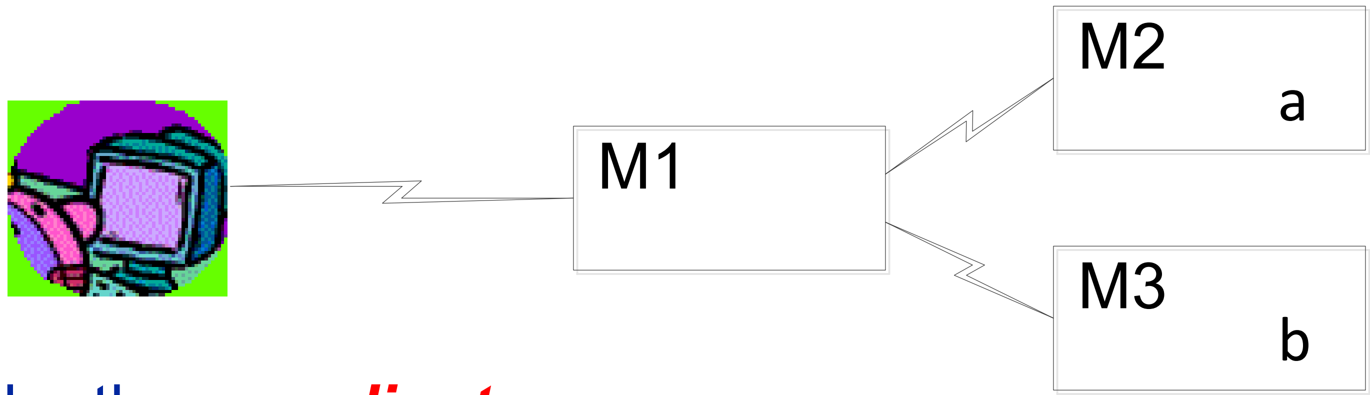
- There is M1 that is the coordinator
- There is M2 that is a participant
- There is M3 that is a participant

- User interacts with M1
- M2 stores a on its local disk
- M3 stores b on its local disk



Our New Example: Money Transfer

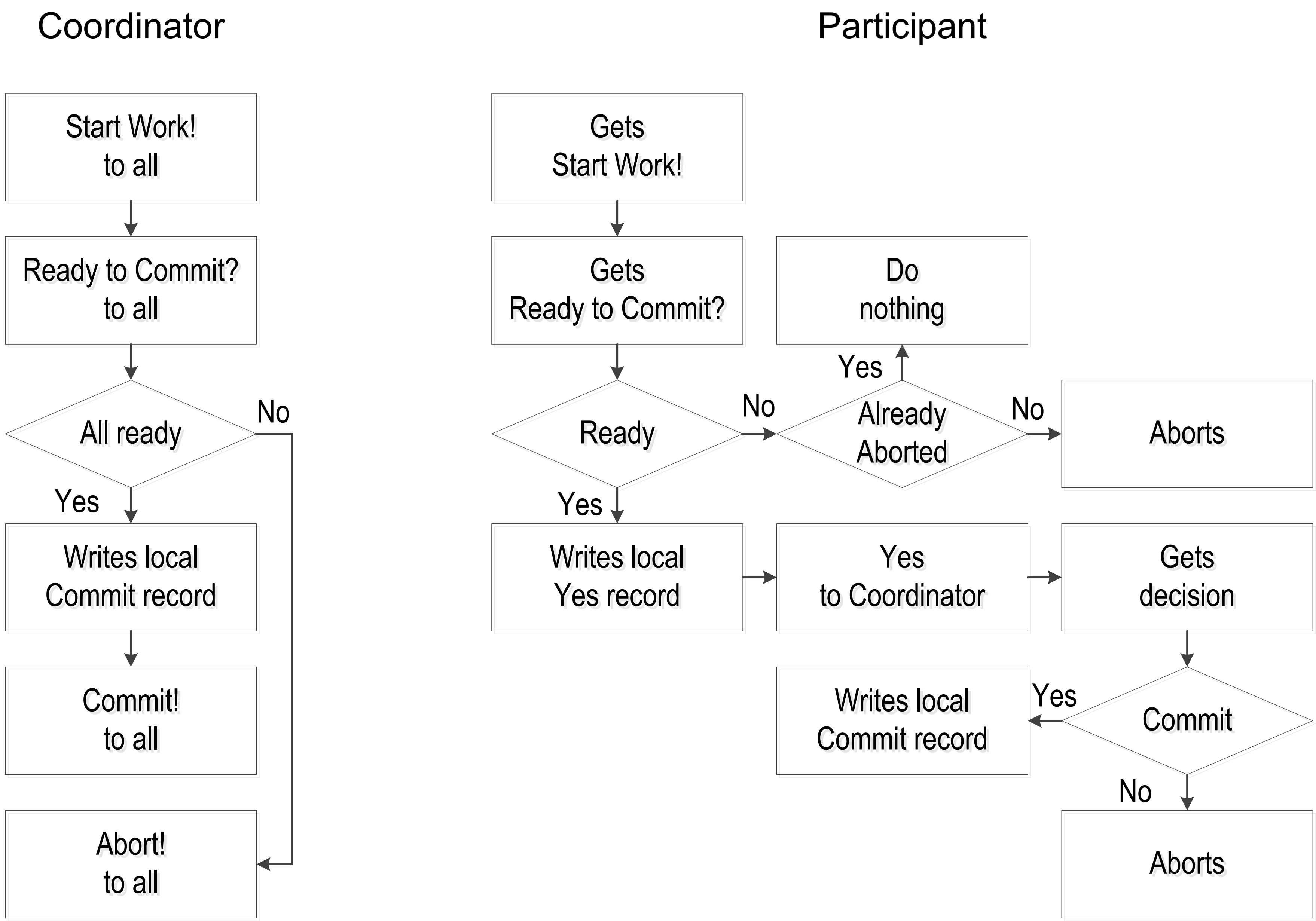
- User asks to transfer $5 from account a to b



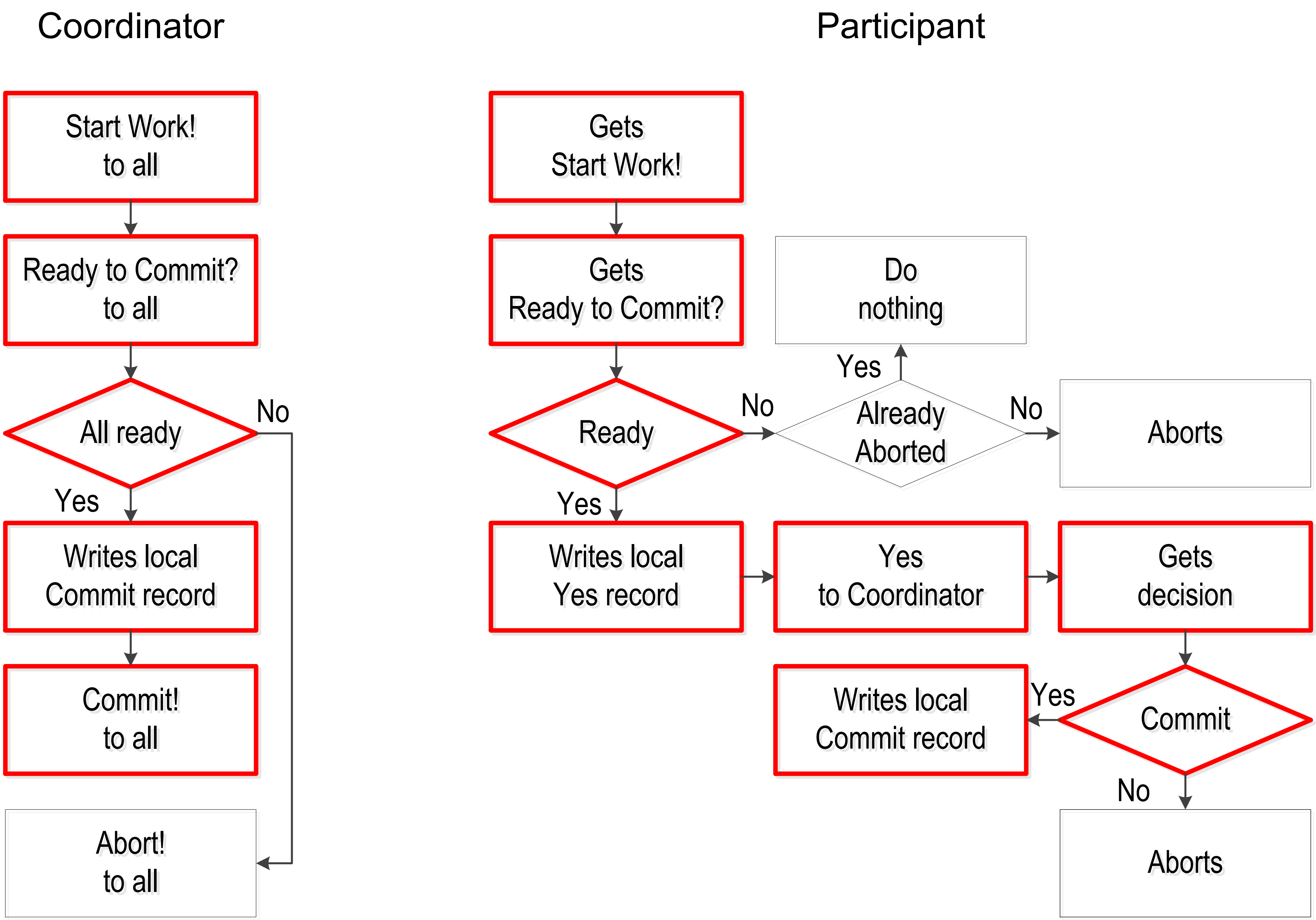
- M1 will be the ***coordinator***
- M2 + M3 will be the ***participants***
- ***Very rough sketch of execution***

1. M1 starts a ***global*** transaction
2. M1 tells M2 to subtract 5 from a
3. M1 tells M3 to add 5 to b
4. M2 starts a ***local*** transaction to subtract 5 from a
5. M3 starts a ***local*** transaction to add 5 to b
6. M1 + M2 + M3 cooperate so “everything” is atomically committed or aborted: all transactions commit or abort

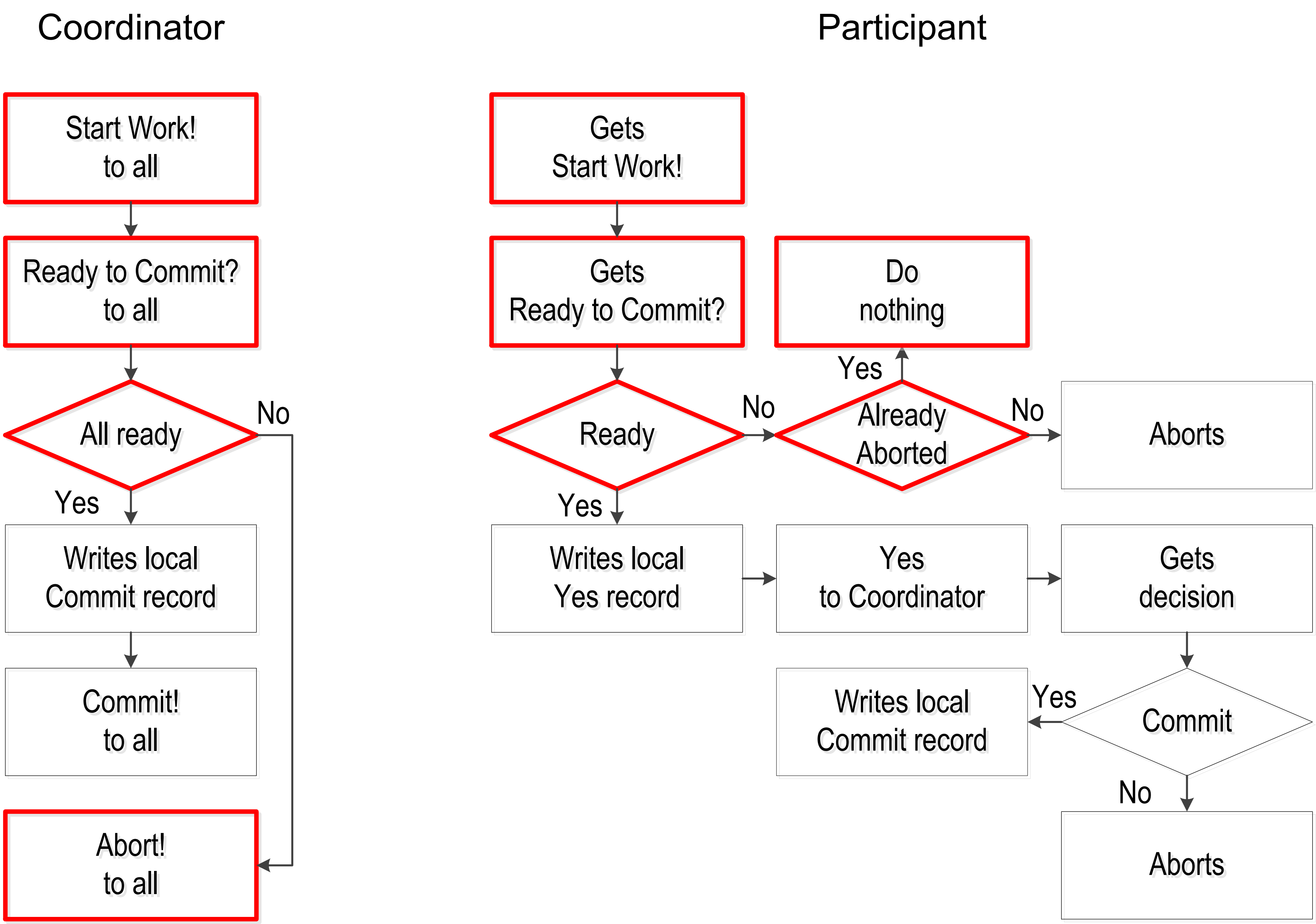
Two-Phase Commit Protocol General Flowchart (Simplified)



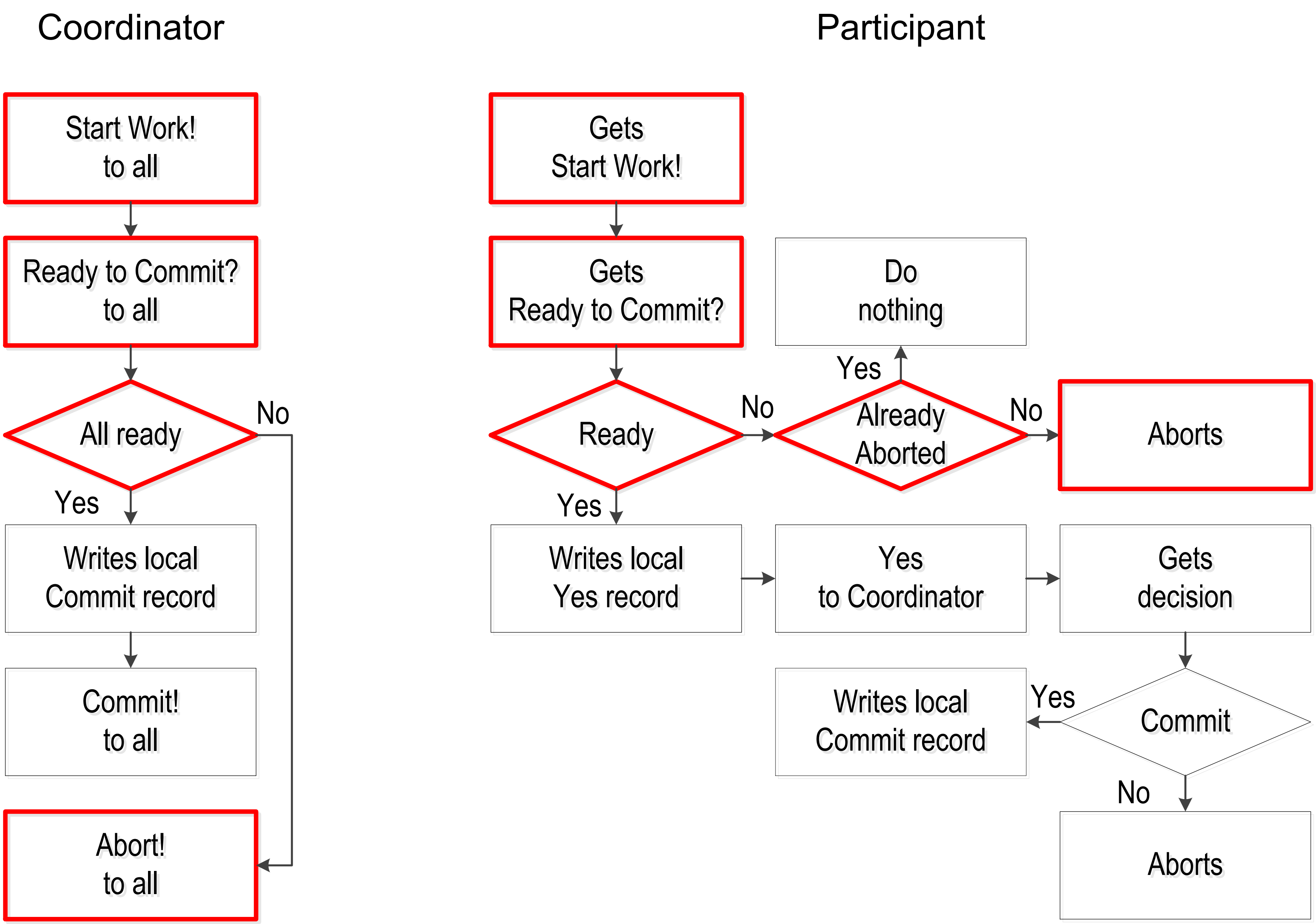
Two-Phase Commit Protocol
All Commit



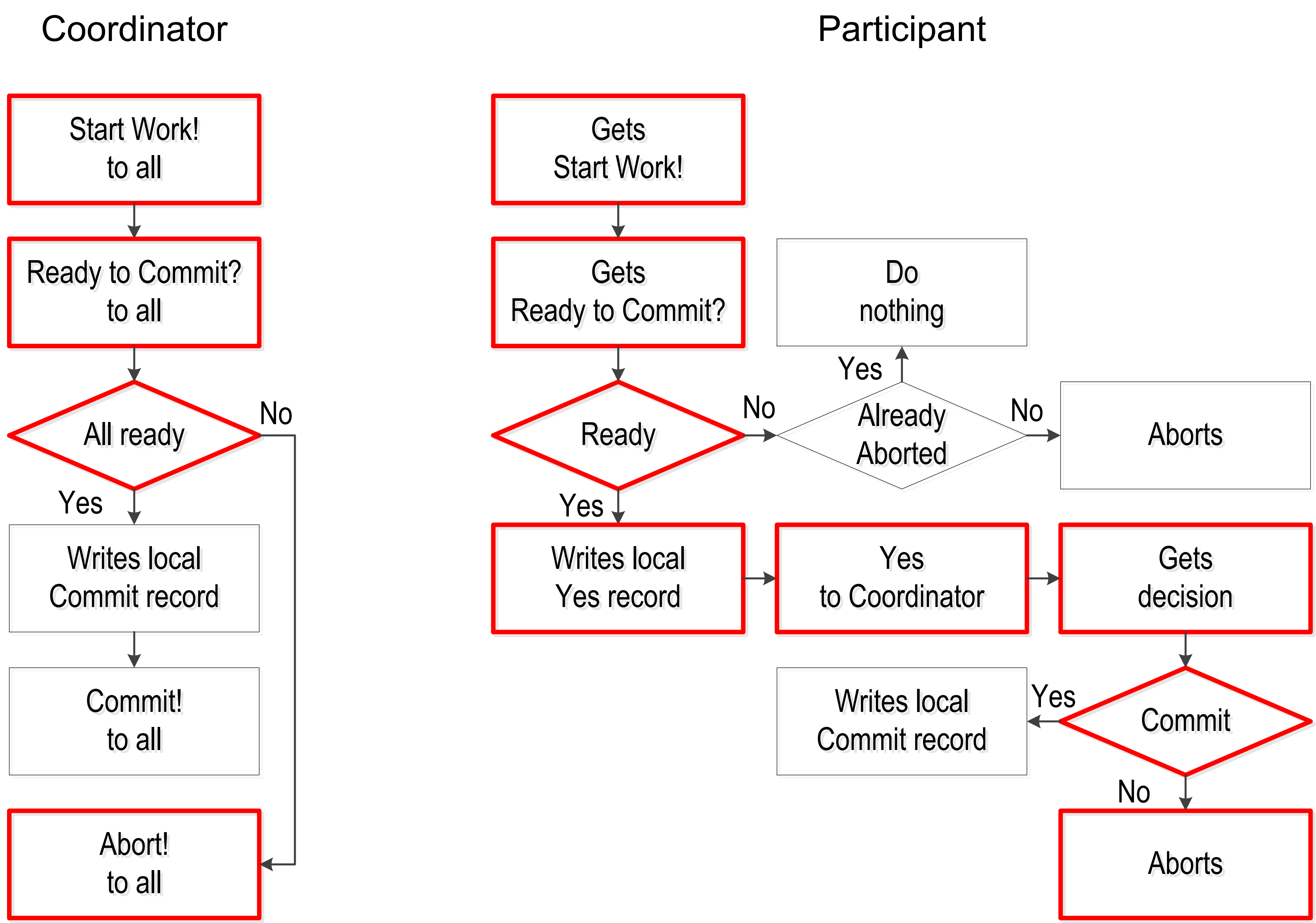
Two-Phase Commit Protocol
A Participant Aborts ⇒ All Abort



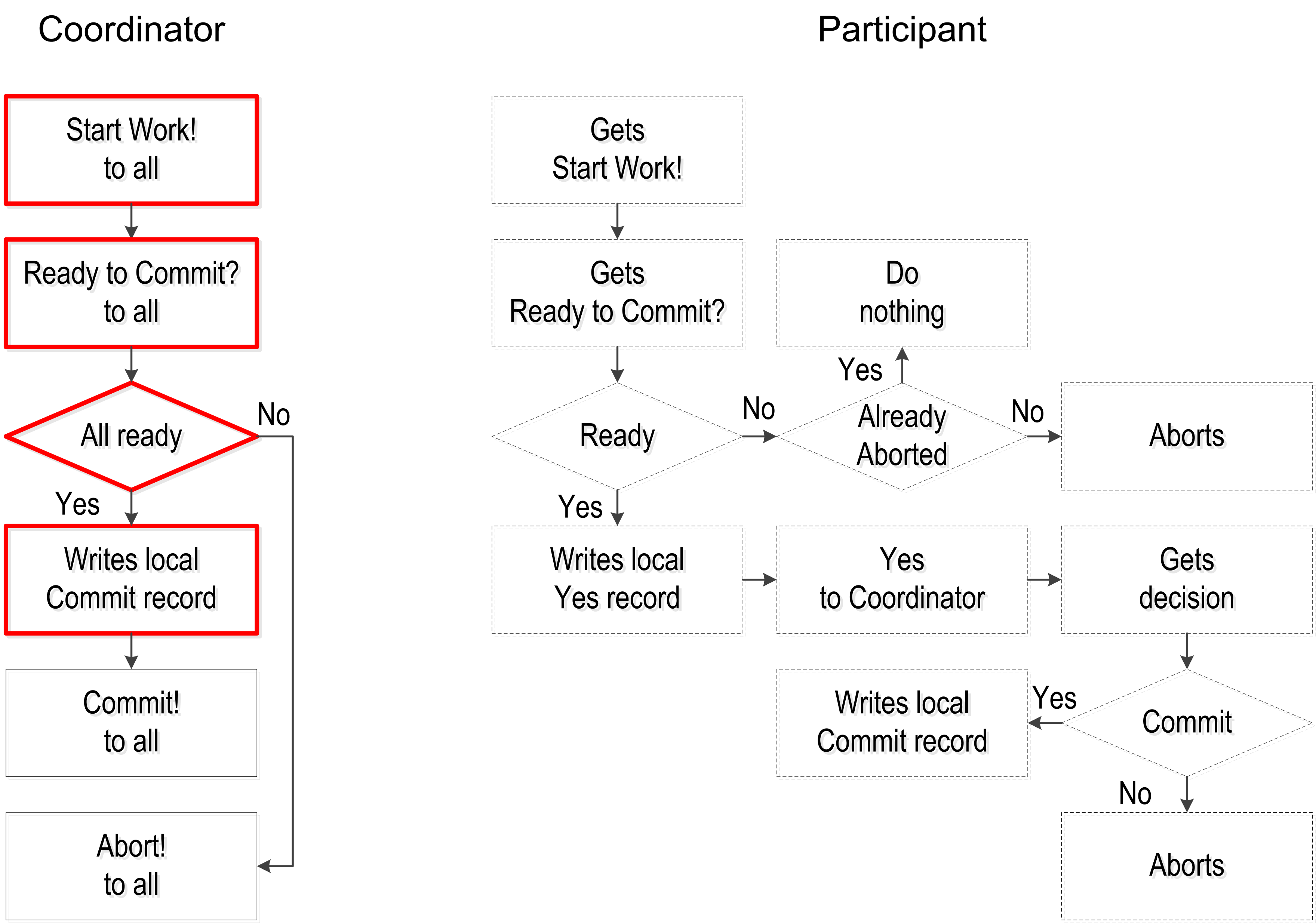
Two-Phase Commit Protocol
A Participant Not Ready ⇒ All Abort



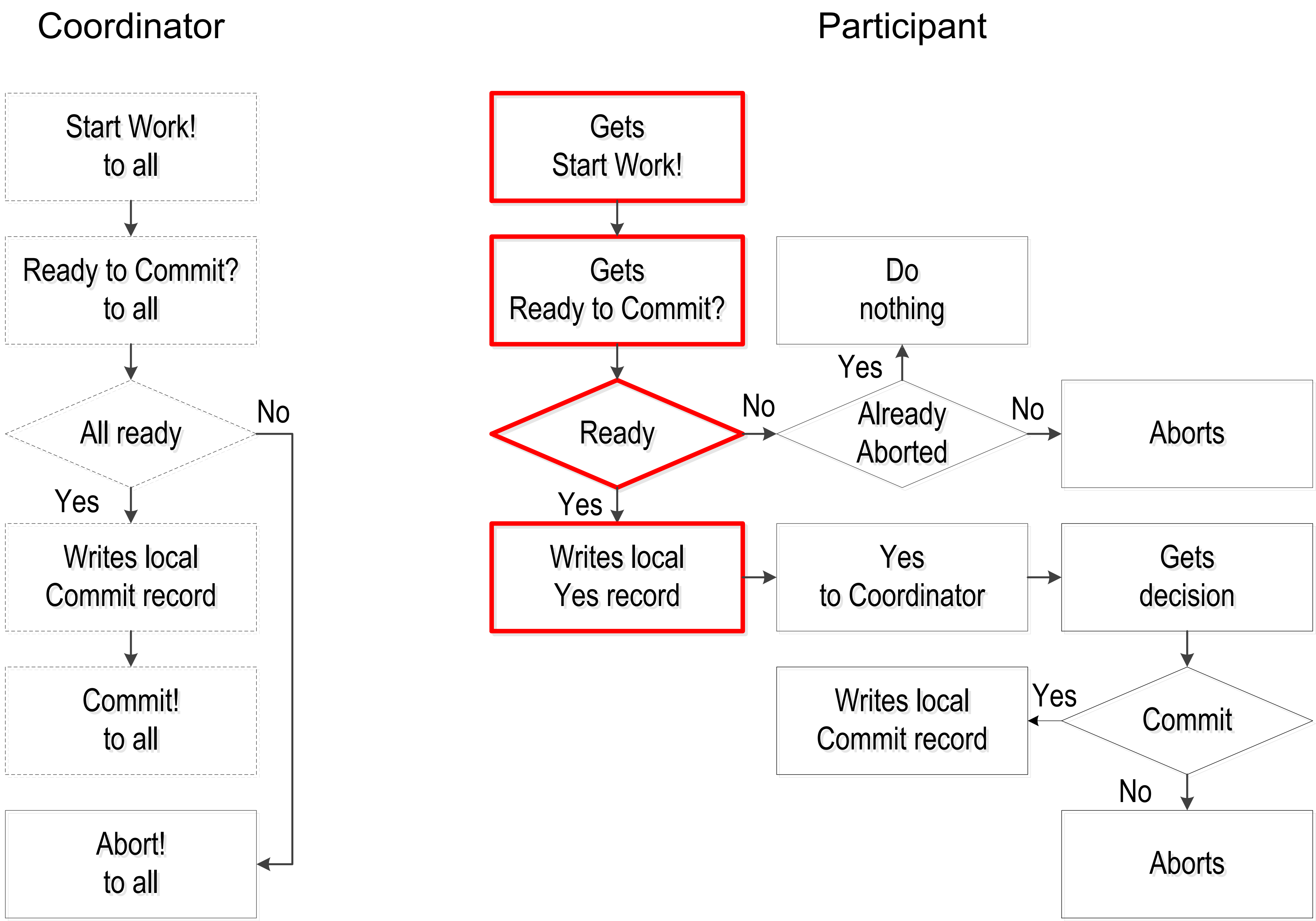
Two-Phase Commit Protocol
Some Participant Cannot Commit ⇒ All Abort



Two-Phase Commit Protocol
Coordinator Decides: Global Commit



Two-Phase Commit Protocol
A Participant Is Uncertain ⇒ It Must Wait



Two-Phase Commit
Many Optimizations Possible

- A participant can report it is ready on its own initiative
- A participant can report that it must abort on its own initiative
- If a participant crashes while uncertain it can ask other participants if they know what the decision was
- …

Another Issue: Global Deadlock Handling

- Assume a system with strict two-phase locking (locked held until after commit)
- The system uses two-phase commit
- M1 “spawned” two transactions
  - T[1,1] executing at site S1
  - T[1,2] executing at site S2
- Only after global commit of M1, T[1,1], T[1,2] can their locks be released
- Only after global commit of M2, T[2,1], T[2,2] can their locks be released
- M2 “spawned” two transactions
  - T[2,1] executing at site S1
  - T[2,2] executing at site S2
- S1 contains items *a* and *b*
- S2 contains items *c* and *d*

Another Issue: Global Deadlock Handling

S1	S2
T[1,1] locks *a*	
T[2,1] locks *b*	
T[1,1] waits to lock *b*	
	T[1,2] locks *c*
	T[2,2] locks *d*
	T[2,2] waits to lock *c*

- For T[1,1] to continue, T[2,1] has to release a lock
- Can only happen after M2, T[2,1], T[2,2] committed

- For T[2,2] to continue, T[1,2] has to release a lock
- Can only happen after M1, T[1,1], T[1,2] committed

Another Issue: Global Deadlock Handling

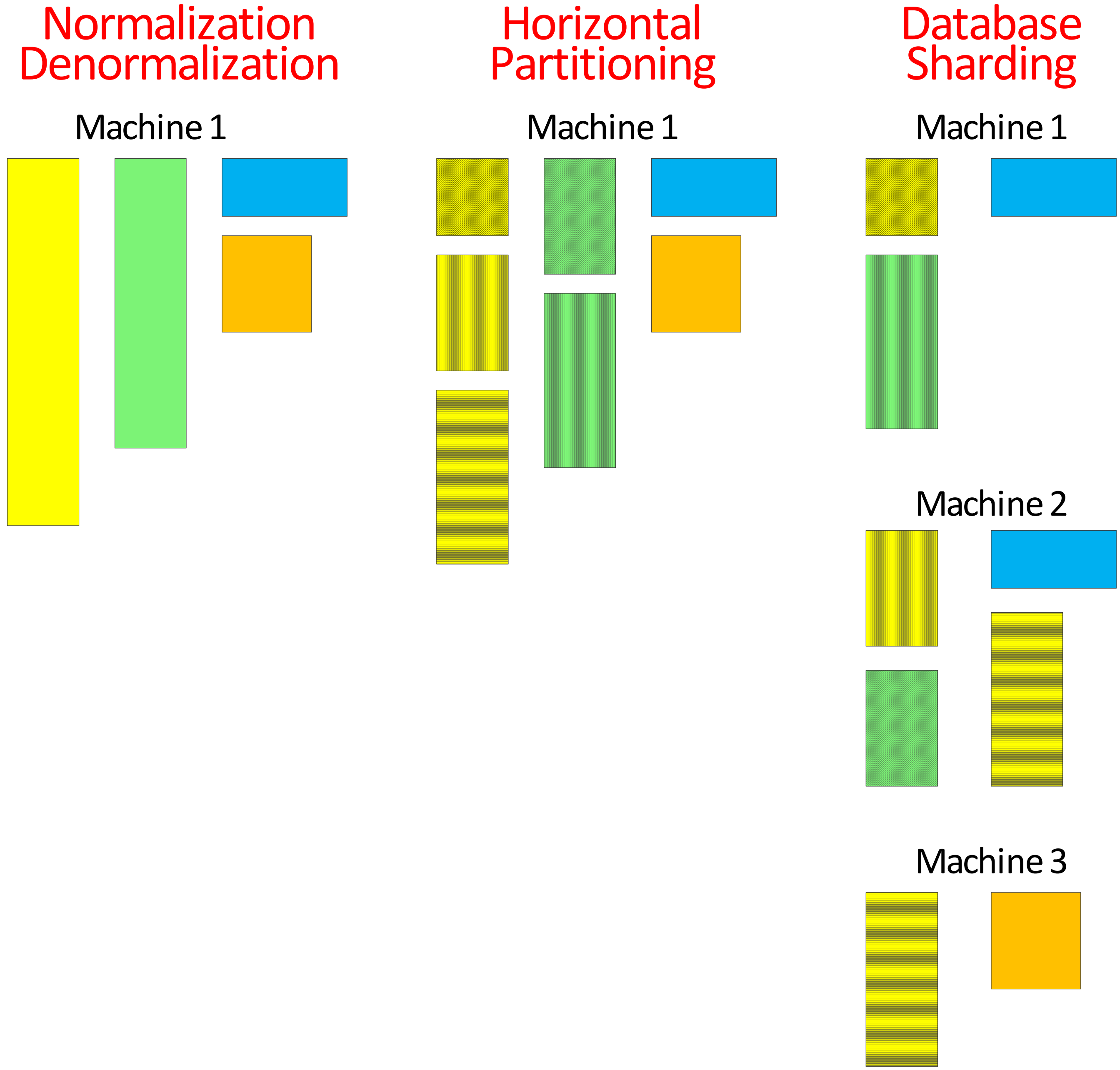
- We have a global deadlock
- There is no local deadlock anywhere
- Difficult to detect

Concurrency

Global Concurrency Management

- We assume that know how to manage recovery, that is ***a distributed transaction either commits or aborts at all sites on which it executes***
- ACD is guaranteed
- ***We need to guarantee I (Isolation) also for transactions that run on more than one machine***
- Each machine is running a local concurrency manager, which we assume operates using rigorous locking
- All locks are held until after local commit or abort on each machine
- In case of global commit, all the locks are held until after global commit decision: ***the coordinator writes commit record on its log***
- This guarantees global serializability

Extension to Multiple Copies (Replication) One Machine vs. Two Machines



Advantages of Data Replication

- It may be useful to replicate some data

- To improve fault-tolerance

  If Machine 1 crashes, we can still access "the blue data" on Machine 2

- To improve efficiency

  Both Machine 1 and Machine 2 can access "the blue data" locally

  So they do not have to use the network to access that data and can access it fast

Problems With Data Replication

- We need to keep the replicated data consistent
- "The blue data" has to be the same on Machine 1 and on Machine 2

- So, if some transaction running on Machine 1 modifies "the blue data", we must make sure that the same modification is made (preferably transparently by the system) to "the blue data" on Machine 2

- So perhaps we could use the following protocol

  If a transaction ***wants to modify*** "the blue data" on one machine, we must make sure transparently that it is modified in the same way on both machines

  A transaction ***wants to read*** "the blue data", it can read it from any machine

A Nightmare Scenario: Network Partition

- The network partitions into two sets that cannot communicate with each other

1. Machine 1
2. Machine 2 and Machine 3

- No transaction can modify “the blue data”

- Because if this is possible, it can only do it on one of the machines
- Then “the blue data” is not consistent
- A transaction that reads “the blue data” on Machine 1 will get a different results than a transaction that reads “the blue data” on Machine 2

Thomas Majority Rule (Example: Sufficient For Understanding)

- There is a data item **X** that is replicated on 5 machines, M1, M2, M3, M4, M5
- The majority of these machines is 3
- The data item is stored as a pair **(X,T)**, where **T** is the timestamp it was last written, assuming the existence of a global clock known to everybody (easy to implement, e.g., atomic clock broadcasting on radio from Colorado)
- To write **X**, access a majority (at least 3) sites and replace the existing **(X,T)** with **(Xnew,Tcurrent)**
- To read **X**, access a majority (= 3) sites and, read the three pairs of **(X,T).** Find the one in which with **T** is **the largest** and return the corresponding **X**

Thomas Majority Rule (Example: Sufficiently General)

- The value of (X,T) in the majority of sites used will be red
- Initial state in the 5 sites
  (10,0) (10,0) (10,0) (10,0) (10,0) (10,0)
- Majority used to write 20 into X at time 1: M1, M2, M3
  (20,1) (20,1) (20,1) (10,0) (10,0)
- Majority used to write 30 into X at time 3: M2, M3, M4
  (20,1) (30,3) (30,3) (30,3) (10,0)
- Majority used to read X at time 6: M3, M4, M5
  Retrieved: (30,3) (30,3) (10,0)
- Since the largest timestamp is 3, the correct value for X is 30
- The protocol works since any two sets of at least 3 machines contain at least one common machine with the latest timestamp

Thomas Majority Rule
General Network Partitioning

- Machines that are in a partition that does not include the majority of the copies cannot act on these copies
  - Cannot read
  - Cannot write
- So this does not solve the problem of “the blue data” as we always need to access both copies

Query Execution Planning

New Issue: Movement of Data

- We now have another cost to consider: moving data among machines
- We will look at one example where we will try just to decrease the cost of moving data

- We have two machines: M1 and M2
- In M1 we have a relation R(<u>A</u>,B)
- In M2 we have a relation S(<u>C</u>,D)
- Assume for simplicity that R and S are of the same size

- We want to compute
  SELECT A, C
  FROM R, S
  WHERE R.B = S.D;

  and have the result at M2

An Execution Plan

A choice

- Copy S to M1
- Compute the result
- Send the result to M2

A better choice?

- Copy R to M2
- Compute the result

But if S is small and R large this may be better

- Copy S to M1
- Compute the result
- Send the result to M2

Even Better Execution Plan If The Parameters Are Right

- On M2 compute
  INSERT INTO TEMP1 SELECT DISTINCT D
  FROM S;
- Copy TEMP1 to M1
- On M1 compute
  INSERT INTO TEMP2 SELECT A, B
  FROM R, TEMP1
  WHERE B = D;
- Copy TEMP2 to M2
- On M2 compute
  INSERT INTO ANSWER SELECT A, C
  FROM TEMP2, S
  WHERE B = D;

- Very Good if TEMP1 and TEMP2 are relatively small

We Used a Semijoin

- Out TEMP2 was ***left semijoin*** of R and S, that is the set of all the tuples of R for which there is a “matching” tuple in S (under the WHERE equality condition)
- Notation: R$\downarrow$S
- Similarly, we can define a right semijoin, denoted by $\uparrow$

NoSQL Has To Compromise

CAP Theorem

- Without defining precisely, if we have more than one machine and replicate the data

- You can get only 2 of the following 3 properties

1. **Consistency** (you will always see a consistent state when accessing data)
2. **Availability** (if you can access a machine, it can read and write items it stores)
3. **Partition Tolerance** (you can work in the presence of partitions)

- So, to get **A** and **B** you may be willing to sacrifice **C**

Key Ideas

- NoSQL databases and Distributed Database
- Two-phase commit
- Global Deadlocks
- Concurrency control with distributed data
- Query processing with distributed data
- The CAP theorem