
University of
Southampton

University of Southampton

Parallel Databases

COMP3211 Advanced Databases

Dr Nicholas Gibbins - nmg@ecs.soton.ac.uk

University of Southampton

Overview

- The I/O bottleneck
- Parallel architectures
- Parallel query processing
  - Inter-operator parallelism
  - Intra-operator parallelism
  - Bushy parallelism
- Concurrency control
- Reliability

University of Southampton

The I/O Bottleneck

University of Southampton

The Memory Hierarchy, Revisited

Type	Capacity	Latency
Registers	10^1 bytes	1 cycle
L1	10^4 bytes	<5 cycles
L2	10^5 bytes	5-10 cycles
RAM	10^9-10^{10} bytes	20-30 cycles (10^{-8} s)
Hard Disk	10^{11}-10^{12} bytes	10^6 cycles (10^{-3} s)

University of Southampton

The I/O Bottleneck

Access time to secondary storage (hard disks) dominates performance of DBMSes

Two approaches to addressing this:

- Main memory databases (expensive!)
- Parallel databases (cheaper!)

Increase I/O bandwidth by spreading data across a number of disks

University of Southampton

Definitions

Parallelism

- An arrangement or state that permits several operations or tasks to be performed simultaneously rather than consecutively

Parallel Databases

- have the ability to split:
  - processing of data
  - access to data
- across multiple processors, multiple disks

University of Southampton

Why Parallel Databases?

- Hardware trends
- Reduced elapsed time for queries
- Increased transaction throughput
- Increased scalability
- Better price/performance
- Improved application availability
- Access to more data

- in short, for better performance

University of Southampton

Parallel Architectures

Monolithic Architecture

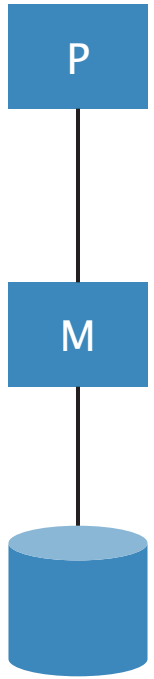
Single processor (P)

- Tasks may be interleaved
- Not true parallelism

Single bank of memory (M)

- Used by transactions
- Buffer pool for staging data to/from disc

Single disc

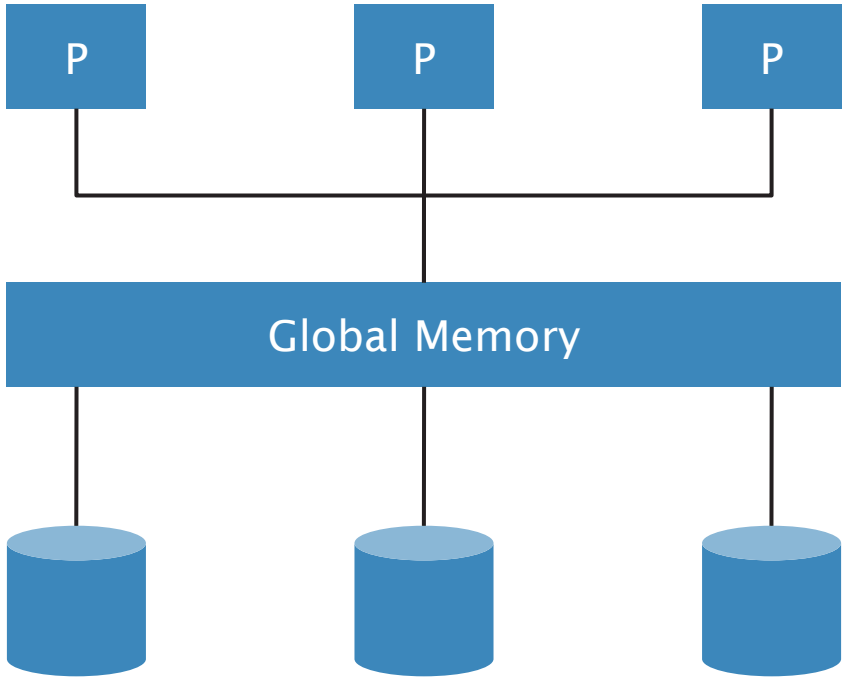


Shared Memory Architecture

- Tightly coupled
- Symmetric Multiprocessor (SMP)

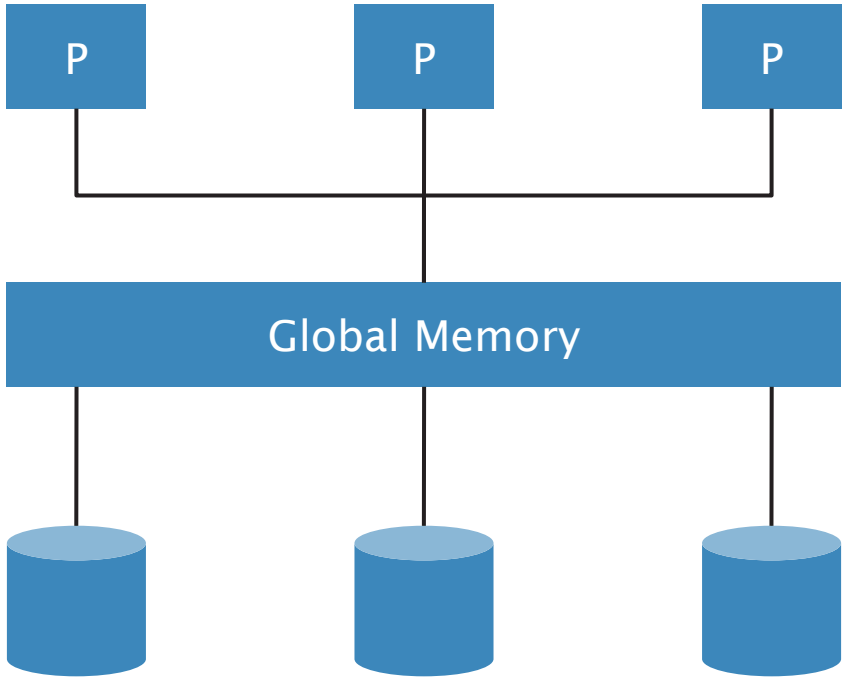
P = processor

M = memory (for buffer pool)



Software – Shared Memory

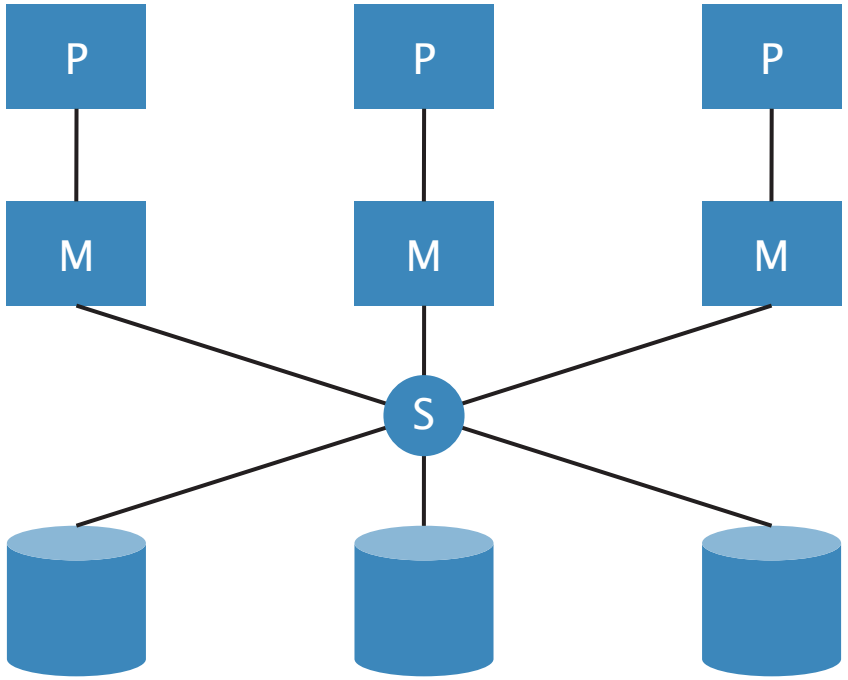
- Less complex database software
- Limited scalability
- Single buffer
- Single database storage



Shared Disc Architecture

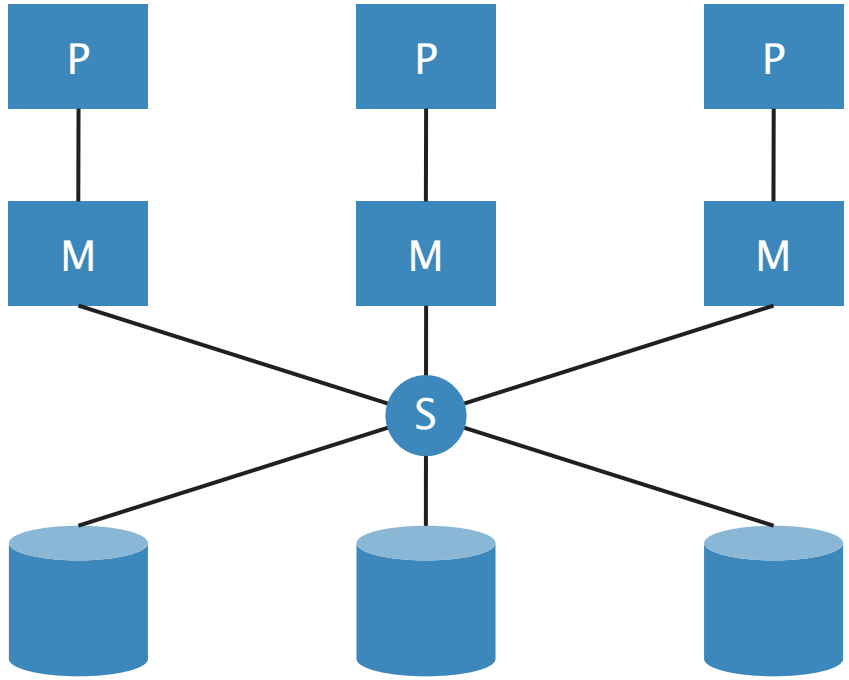
- Loosely coupled
- Distributed Memory

S = switch



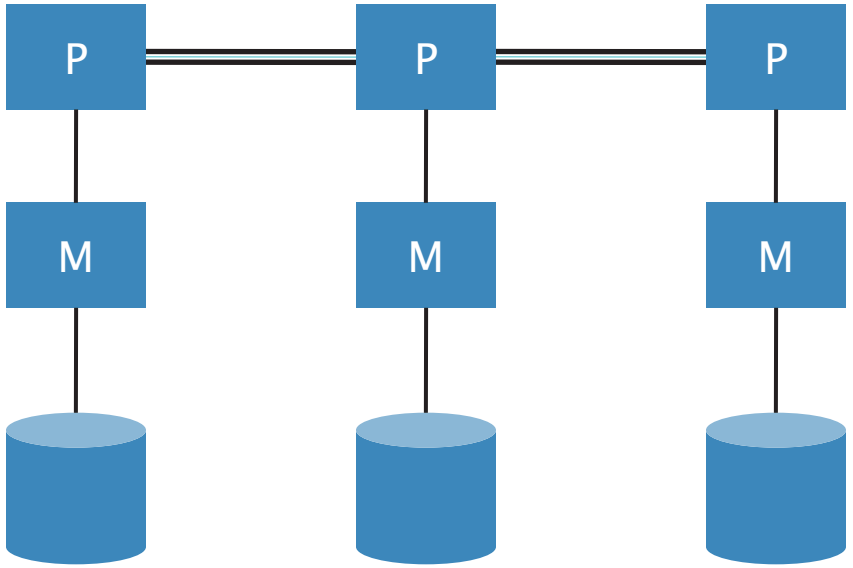
Software – Shared Disc

- Avoids memory bottleneck
- Same page may be in more than one buffer at once – can lead to incoherence
- Needs global locking mechanism
- Single logical database storage
- Each processor has its own database buffer



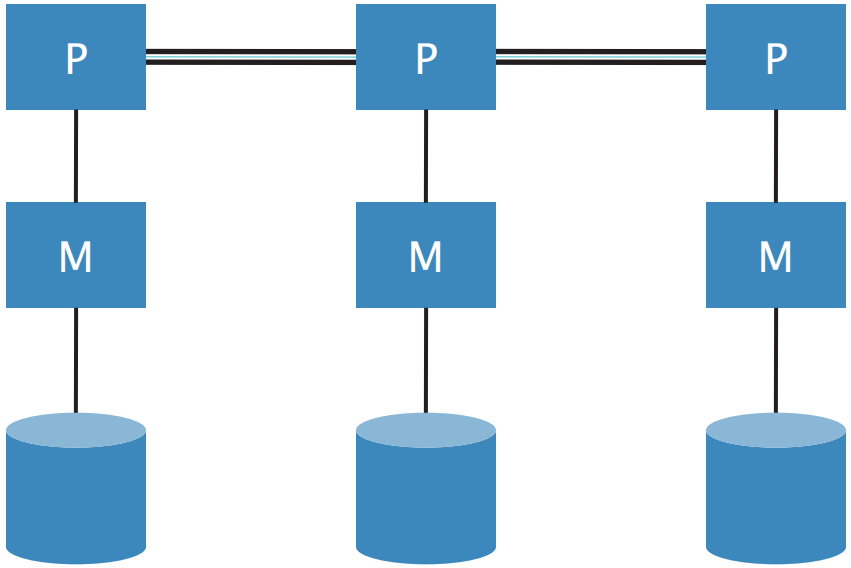
Shared Nothing Architecture

- Massively Parallel
- Loosely Coupled
- High Speed Interconnect (between processors)



Software - Shared Nothing

- Each processor owns part of the data
- Each processor has its own database buffer
- One page is only in one local buffer – no buffer incoherence
- Needs distributed deadlock detection
- Needs multiphase commit protocol
- Needs to break SQL requests into multiple sub-requests



University of Southampton

Hardware vs. Software Architecture

- It is possible to use one software strategy on a different hardware arrangement
- Also possible to simulate one hardware configuration on another
  - Virtual Shared Disk (VSD) makes an IBM SP shared nothing system look like a shared disc setup (for Oracle)
- From this point on, we deal only with shared nothing

University of Southampton

Shared Nothing Challenges

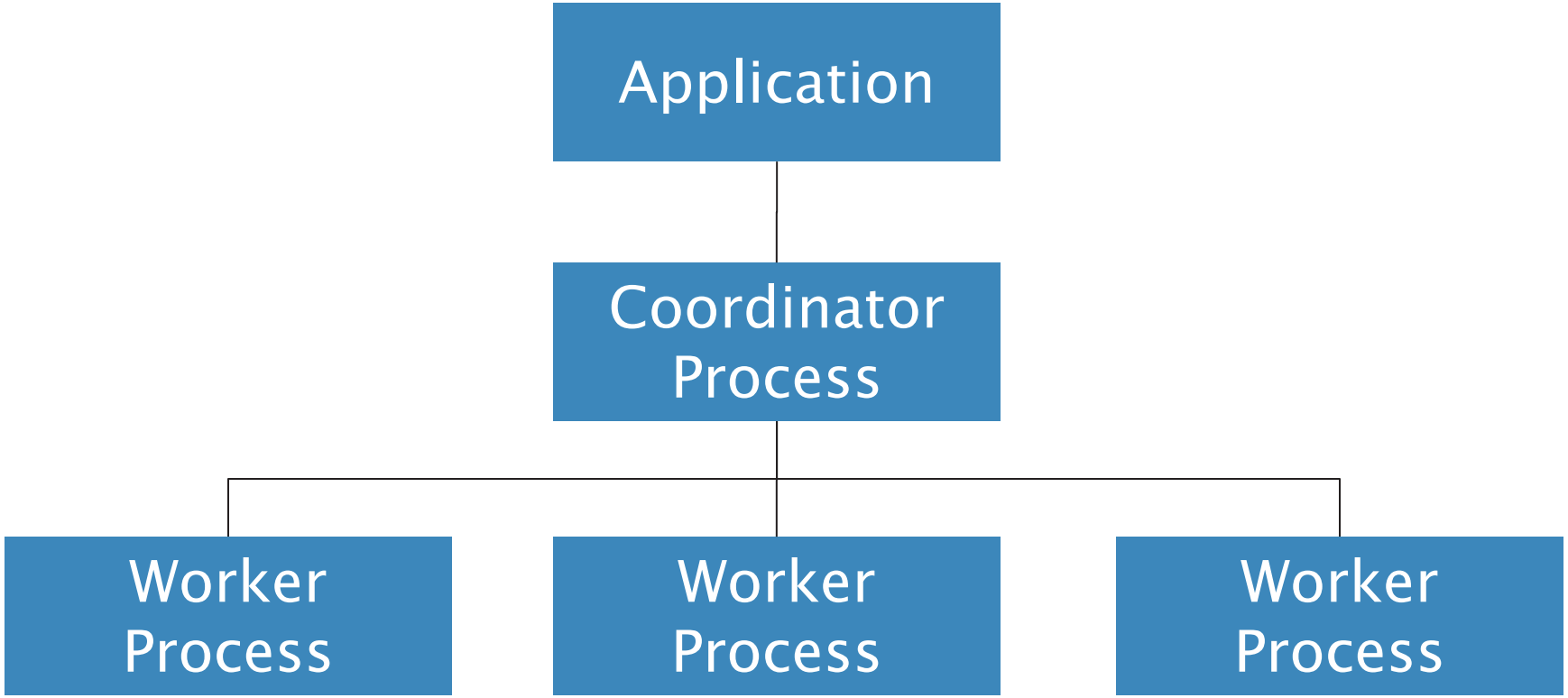
- Partitioning the data
- Keeping the partitioned data balanced
- Splitting up queries to get the work done
- Concurrency control and avoiding distributed deadlock
- Dealing with node failure

University of Southampton

Parallel Query Processing

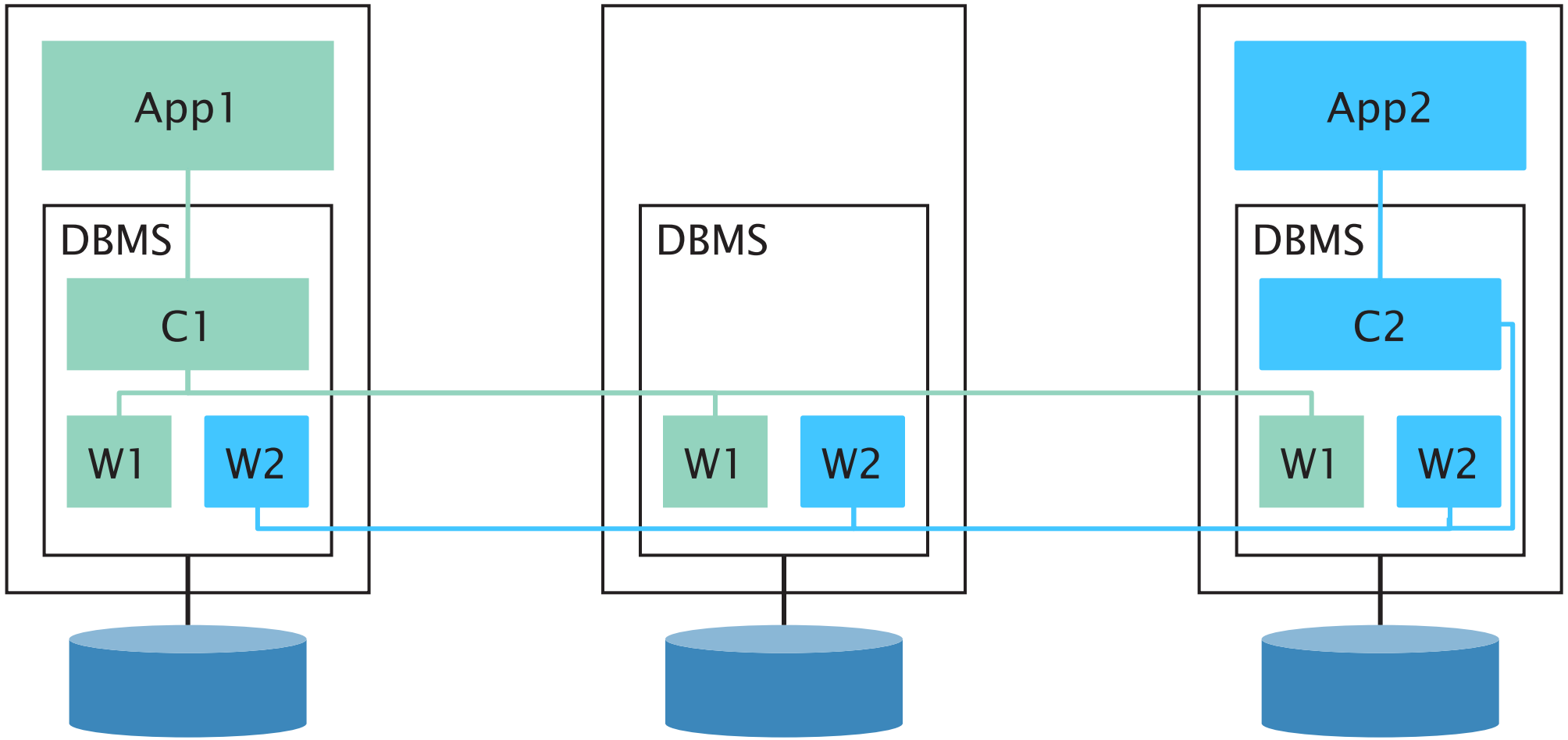
University of Southampton

Dividing up the Work



University of Southampton

Database Software on each node



University of Southampton

Inter-Query Parallelism

Improves throughput

Different queries/transactions execute on different processors

- (largely equivalent to material in lectures on concurrency)

University of Southampton

Intra-Query Parallelism

Improves response times (lower latency)

Intra-operator (horizontal) parallelism

- Operators decomposed into independent operator instances, which perform the same operation on different subsets of data

Inter-operator (vertical) parallelism

- Operations are overlapped
- Pipeline data from one stage to the next without materialisation

Bushy (independent) parallelism

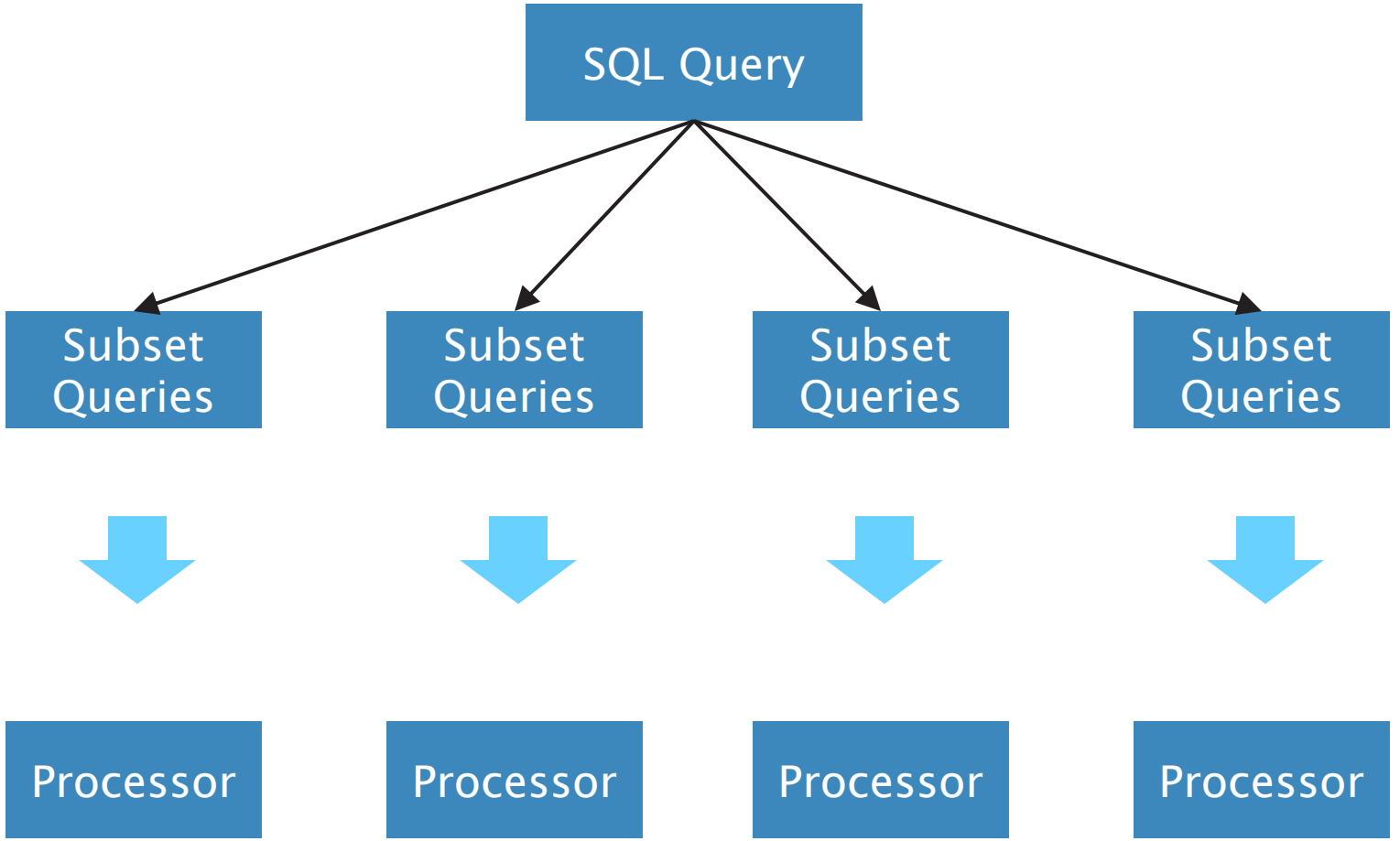
- Subtrees in query plan executed concurrently

University of Southampton

Intra-Operator Parallelism

University of Southampton

Intra-Operator Parallelism



University of Southampton

Partitioning

Decomposition of operators relies on data being partitioned across the servers that comprise the parallel database

- Access data in parallel to mitigate the I/O bottleneck

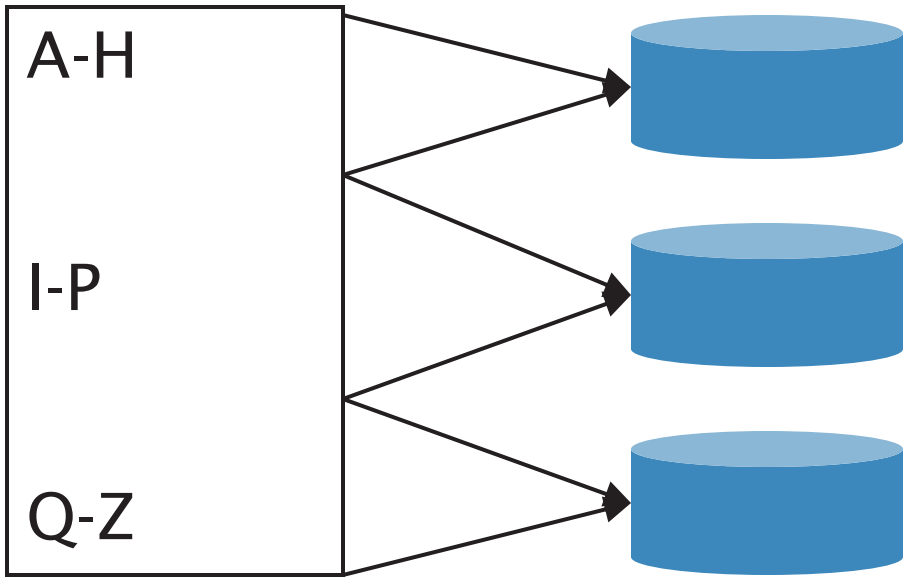
Partitions should aim to spread I/O load evenly across servers

Choice of partitions affords different parallel query processing approaches:

- Range partitioning
- Hash partitioning
- Schema partitioning

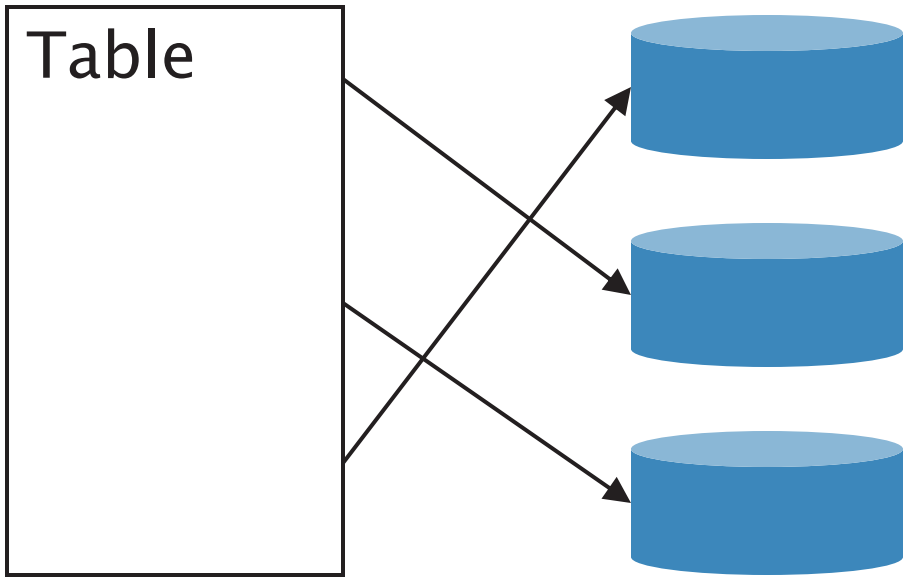
University of Southampton

Range Partitioning



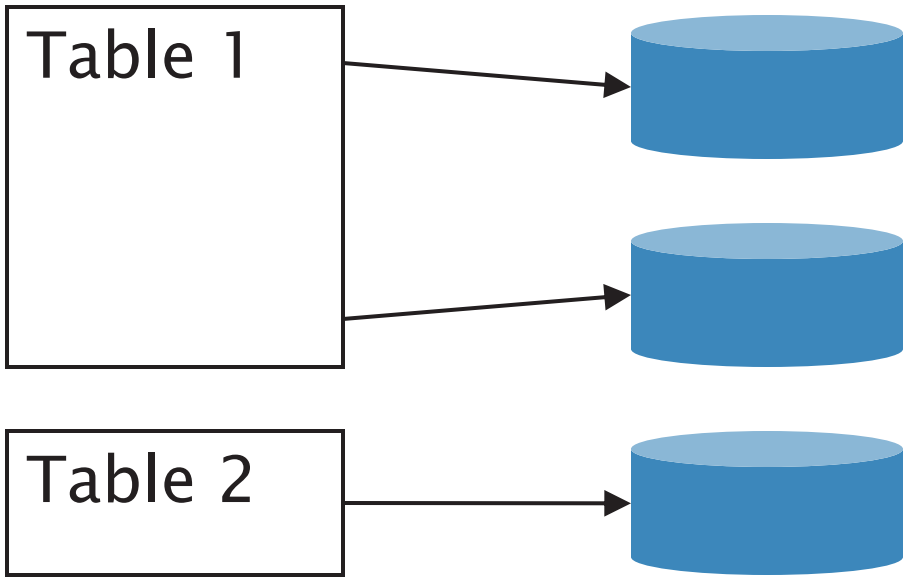
University of Southampton

Hash Partitioning



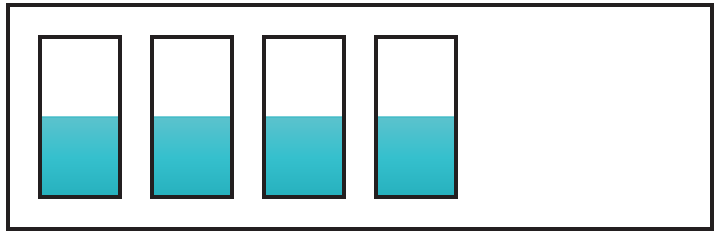
University of Southampton

Schema Partitioning



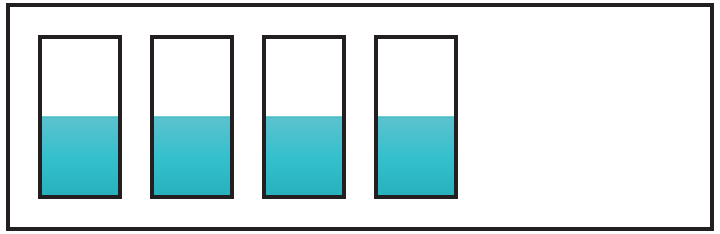
Rebalancing Data

Data in proper balance

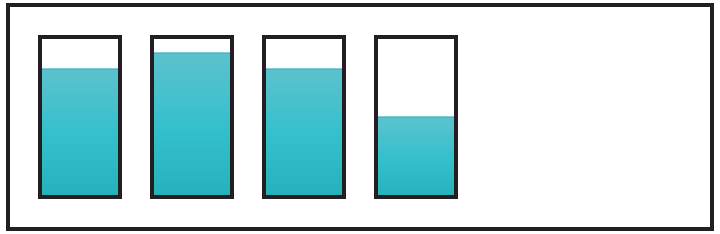


Rebalancing Data

Data in proper balance



Data grows, performance drops

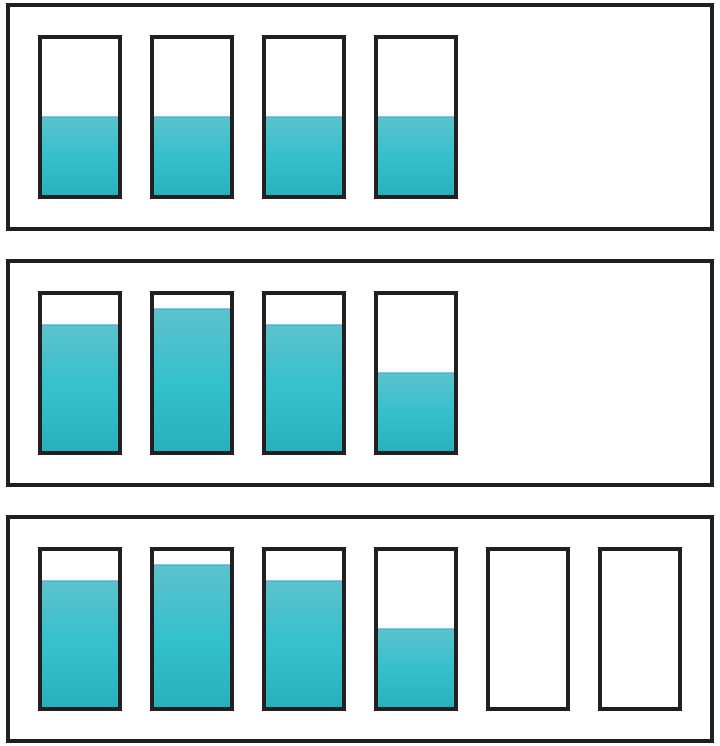


Rebalancing Data

Data in proper balance

Data grows, performance drops

Add new nodes and disc



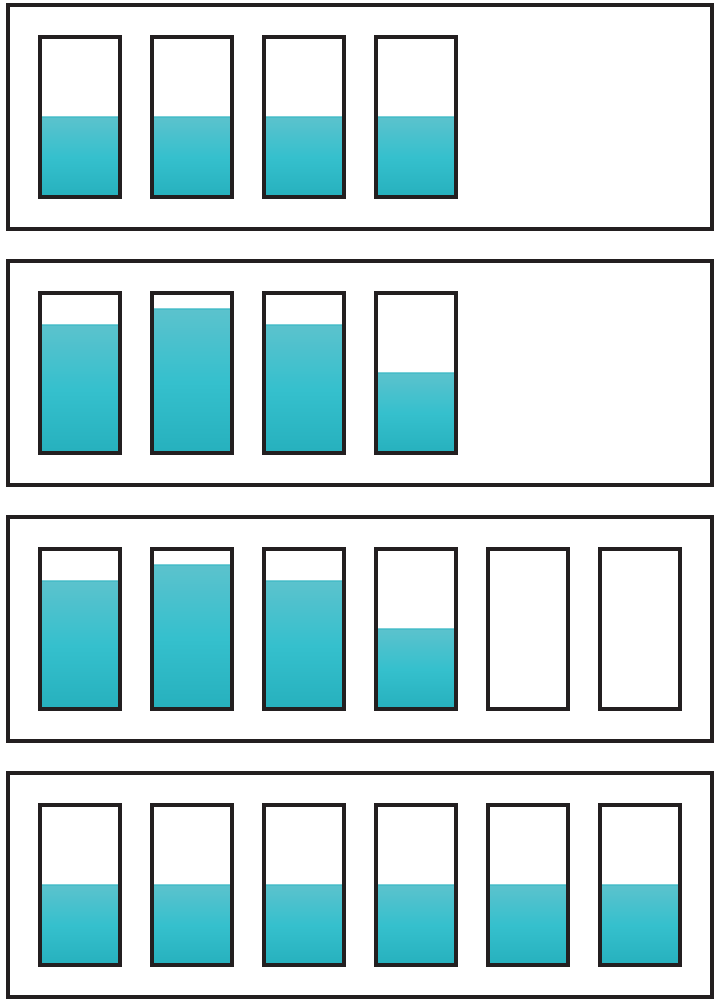
Rebalancing Data

Data in proper balance

Data grows, performance drops

Add new nodes and disc

Redistribute data to new nodes



Intra-Operator Parallelism

Example query:

```
SELECT c1,c2 FROM t WHERE c1>5.5
```

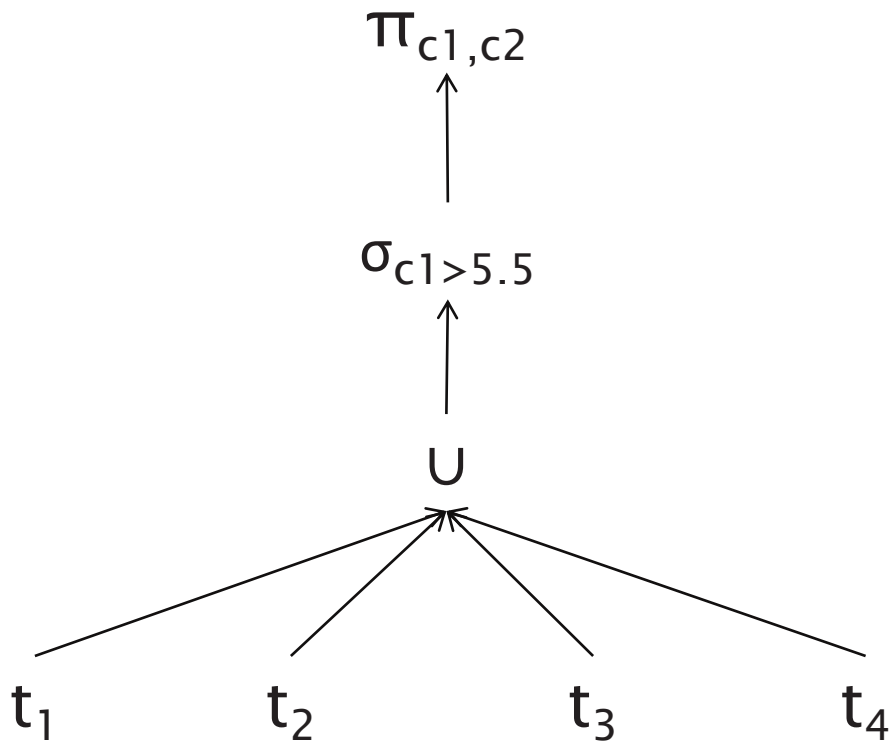
Assumptions:

- 100,000 rows
- Predicates eliminate 90% of the rows

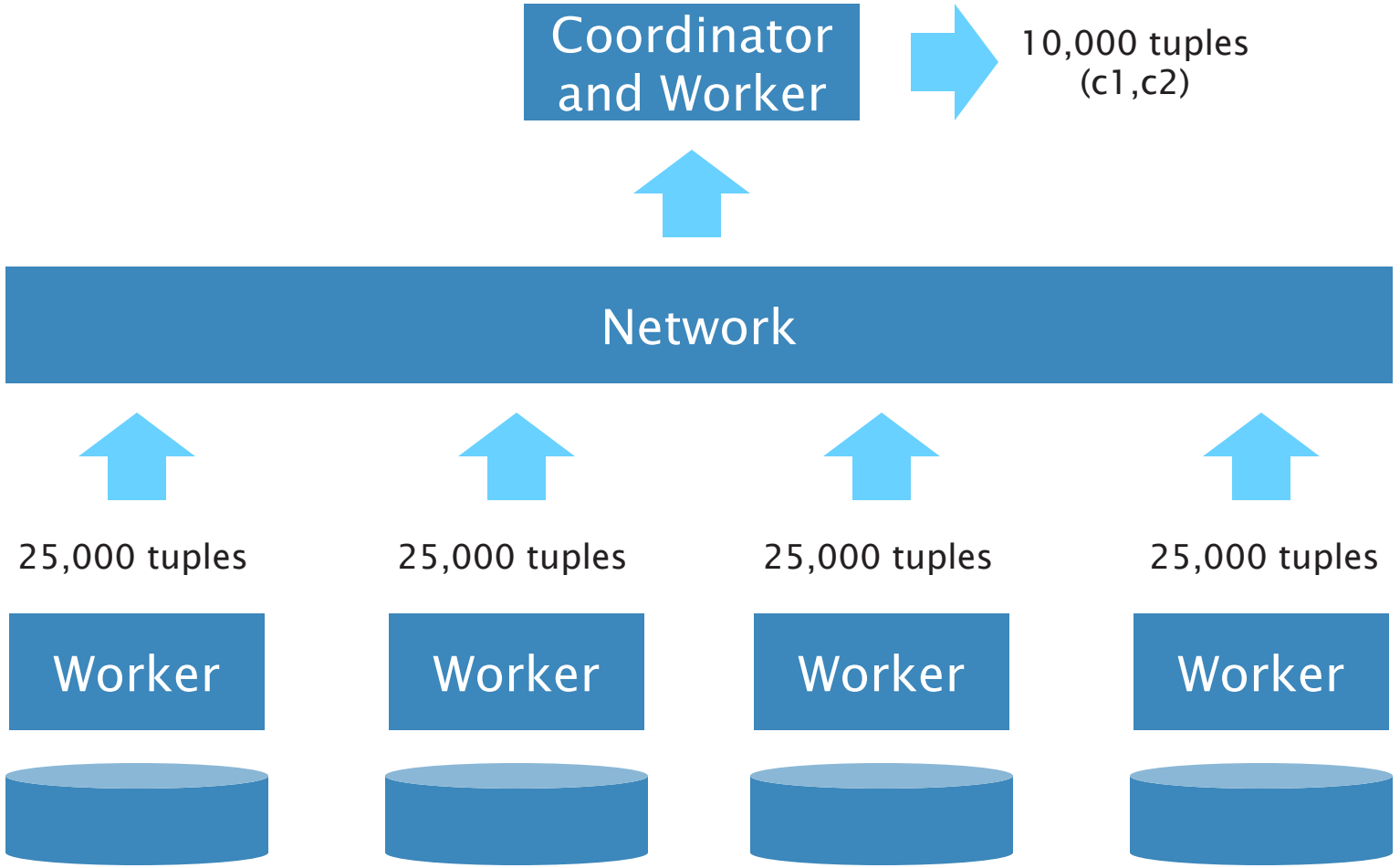
Considerations for query plans:

- Data shipping
- Query shipping

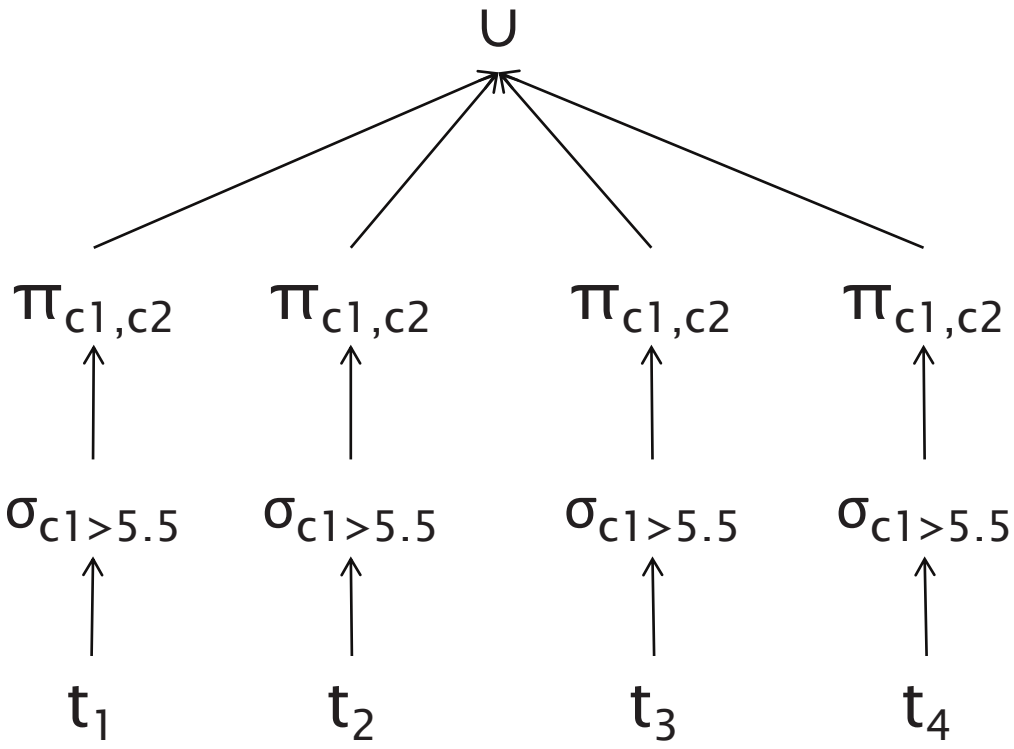
Data Shipping



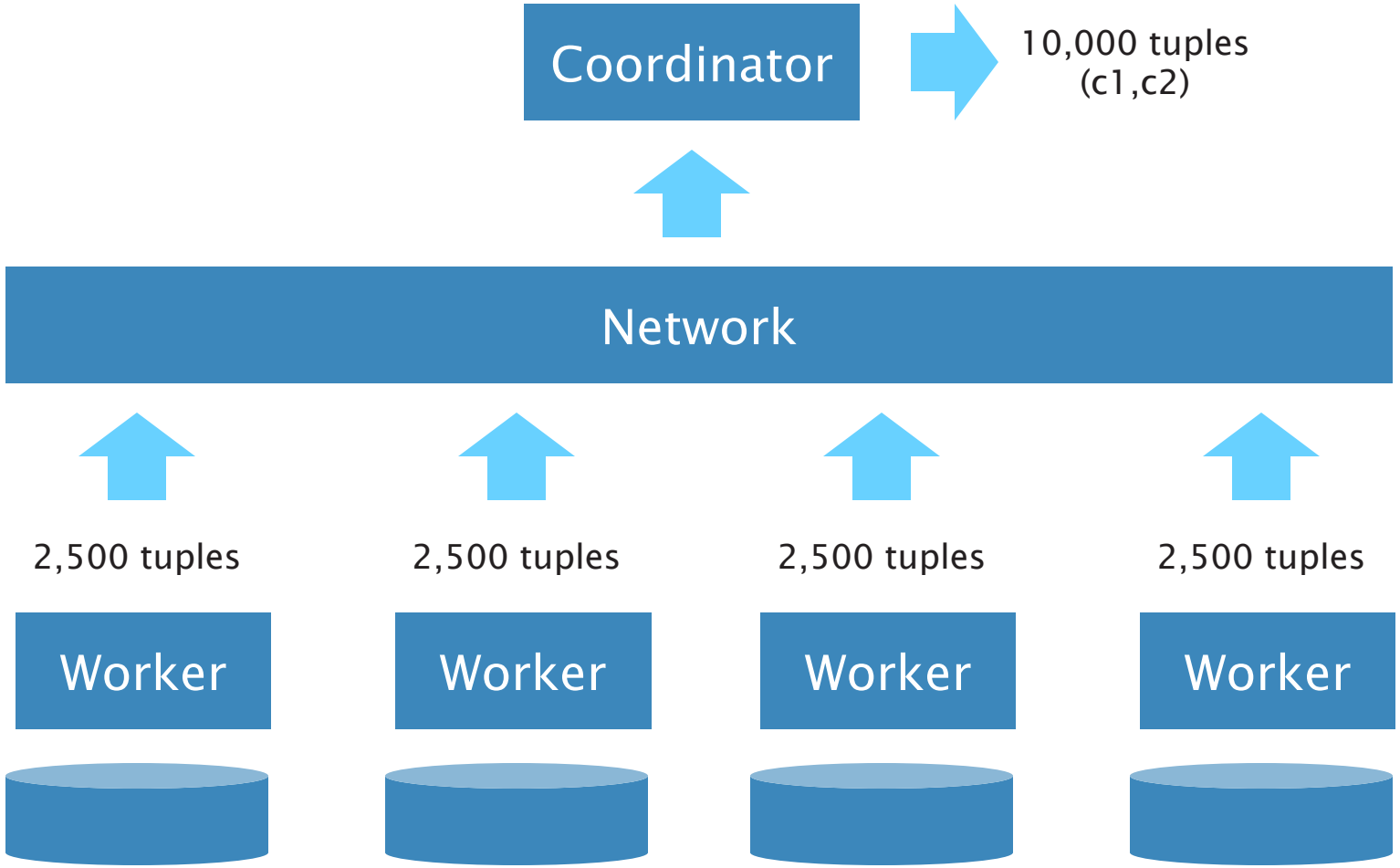
Data Shipping



Query Shipping



Query Shipping



University of Southampton

Query Shipping Benefits

- Database operations are performed where the data are, as far as possible
- Network traffic is minimised
- For basic database operators, code developed for serial implementations can be reused
- In practice, mixture of query shipping and data shipping has to be employed

University of Southampton

Inter-Operator Parallelism

University of Southampton

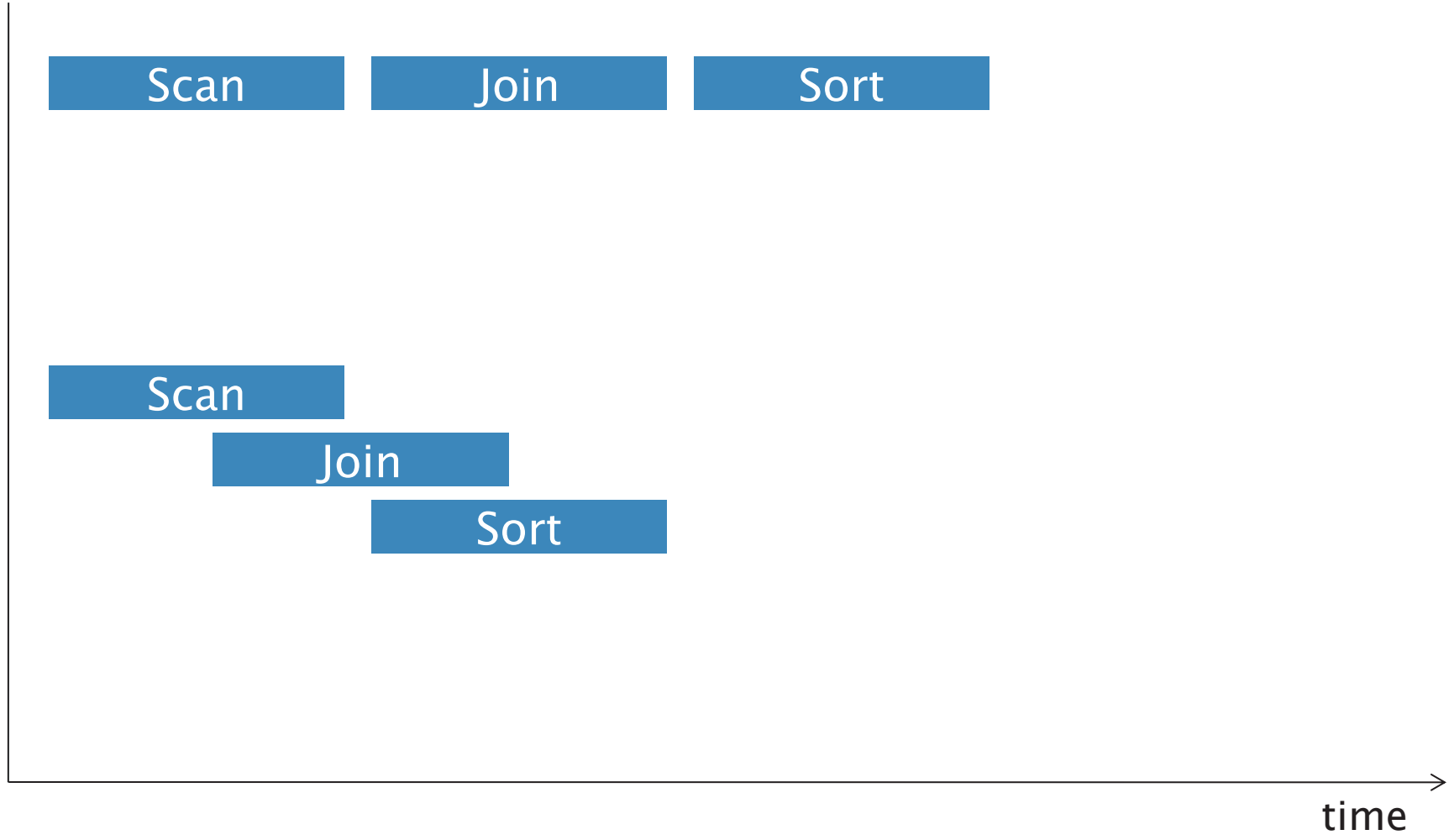
Inter-Operator Parallelism

Allows operators with a producer-consumer dependency to be executed concurrently

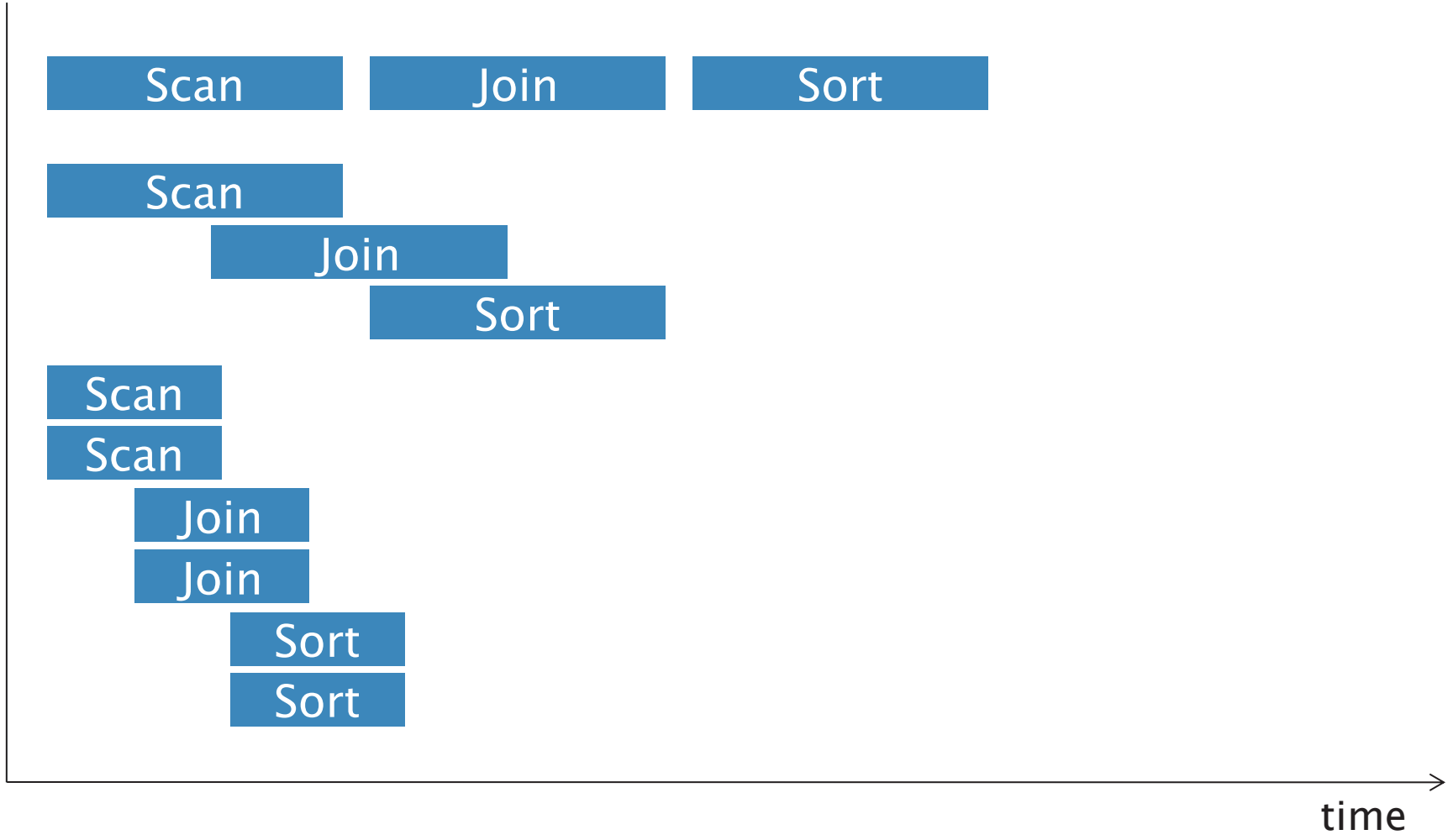
- Results produced by producer are pipelined directly to consumer
- Consumer can start before producer has produced all results
- No need to materialise intermediate relations on disk (although available buffer memory is a constraint)
- Best suited to single-pass operators

University of Southampton

Inter-Operator Parallelism



Intra- + Inter-Operator Parallelism



The Volcano Architecture

Basic operators as usual:

- scan, join, sort, aggregate (sum, count, average, etc)

The Exchange operator

- Inserted between the steps of a query to:
  - Pipeline results
  - Direct streams of data to the next step(s), redistributing as necessary

Provides mechanism to support both vertical and horizontal parallelism

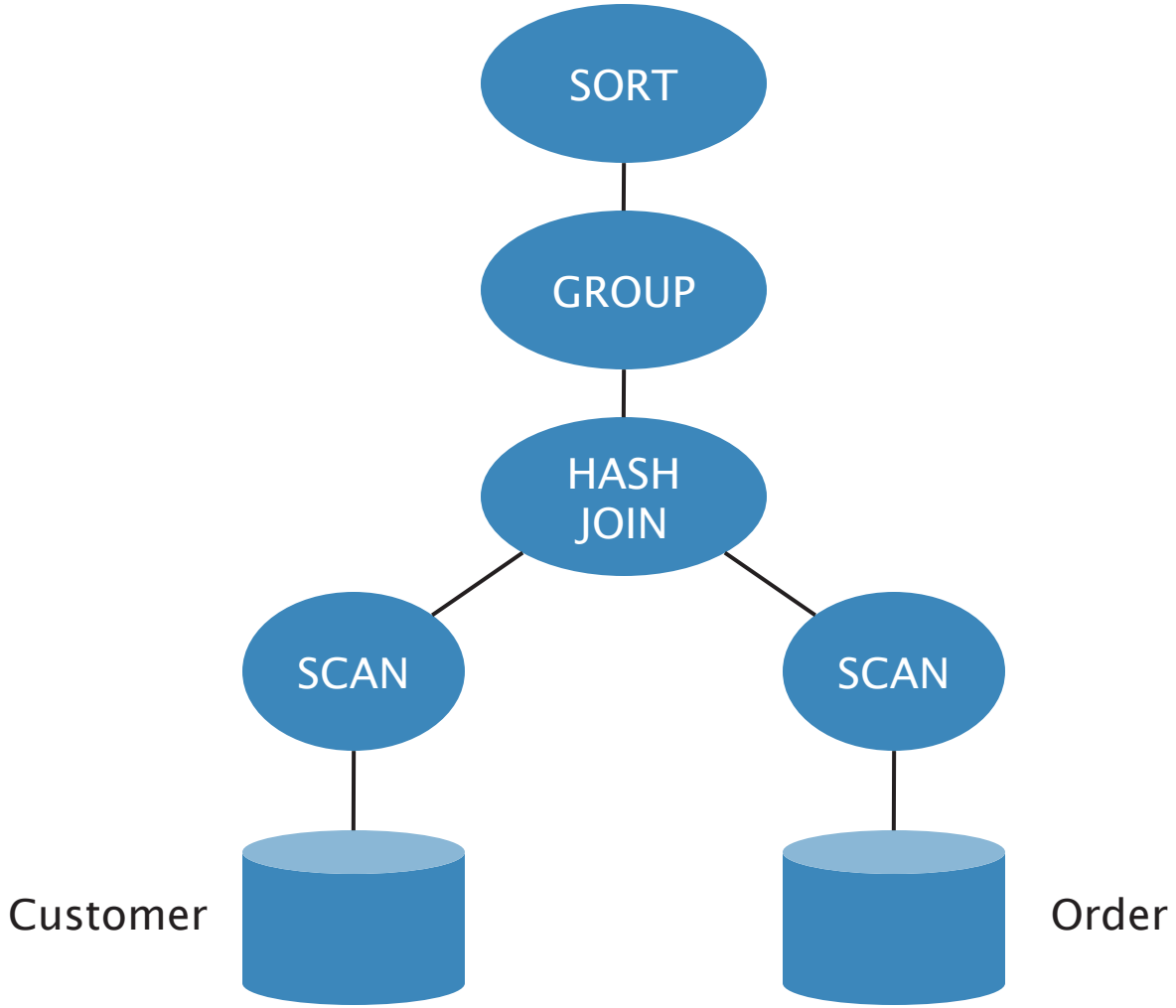
Exchange Operators

Example query:

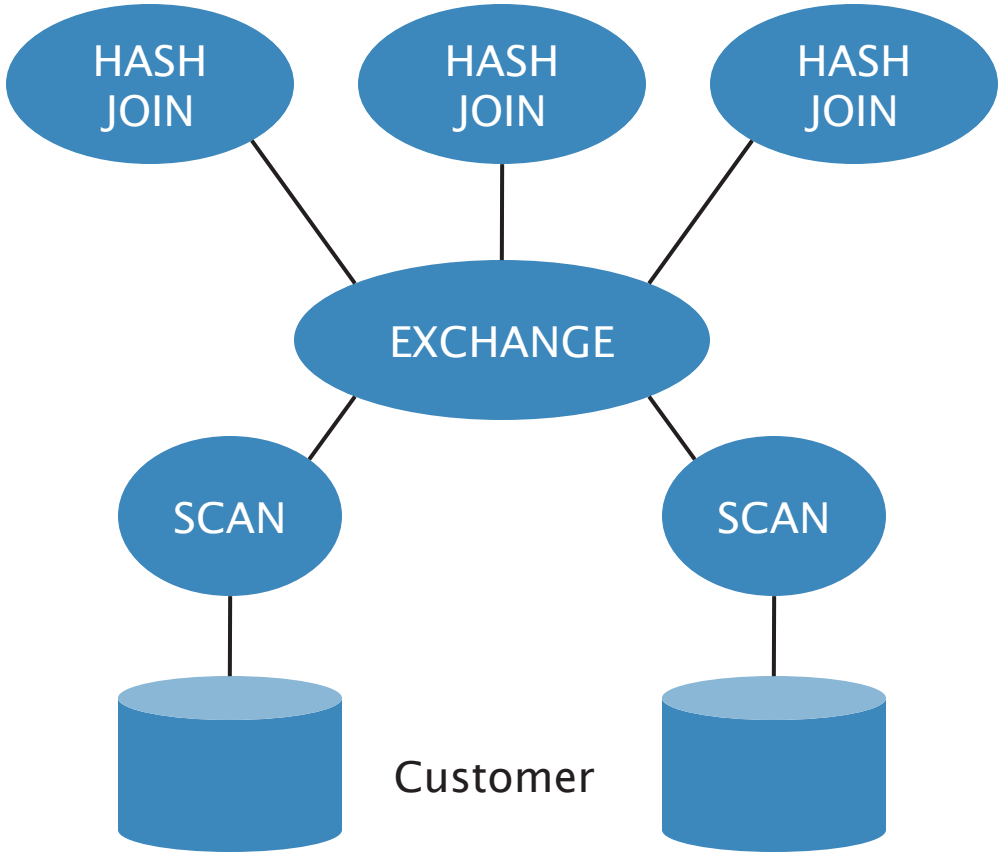
```
SELECT county, SUM(order_item)
FROM customer, order
WHERE order.customer_id=customer_id
GROUP BY county
ORDER BY SUM(order_item)
```

University of Southampton

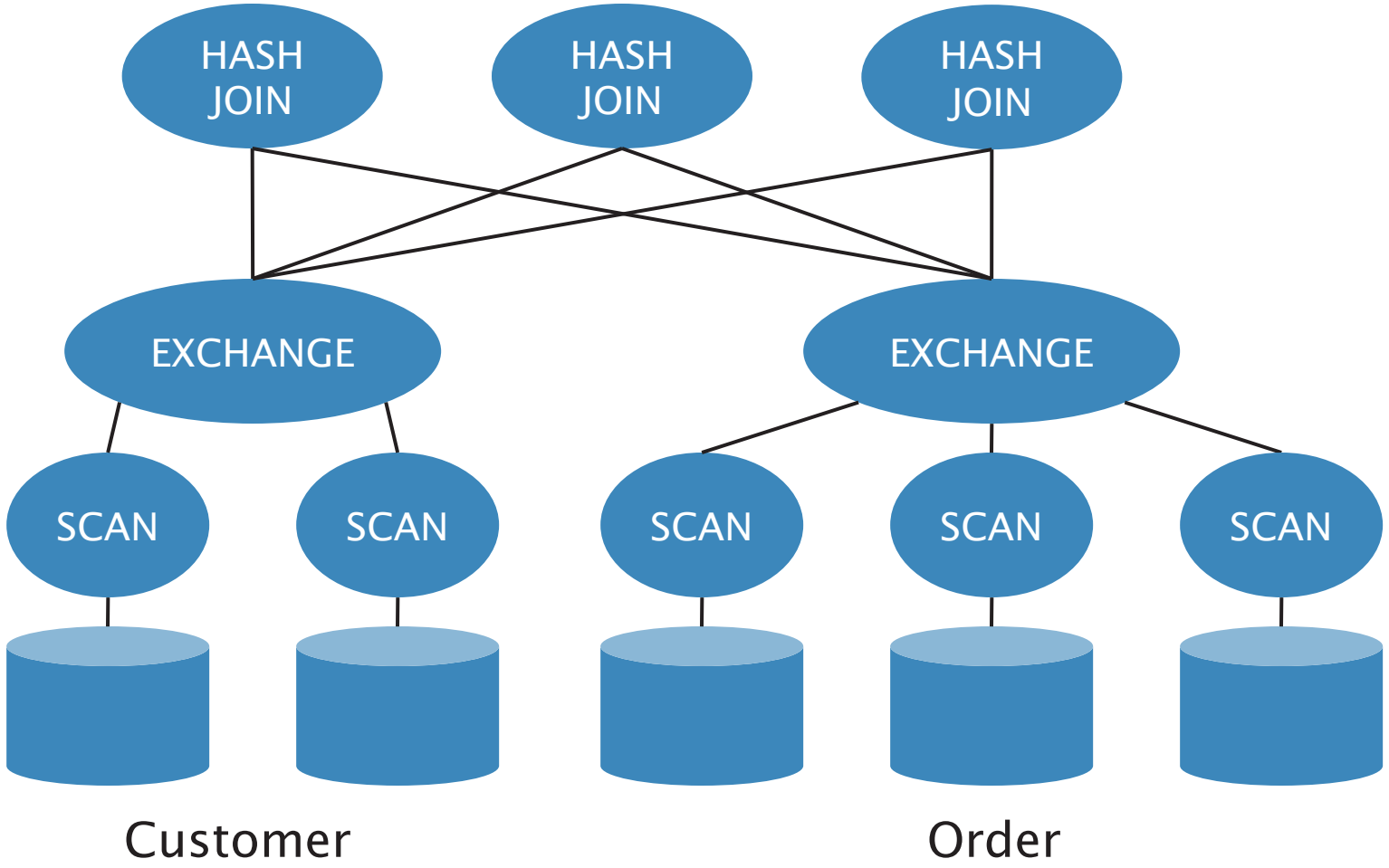
Exchange Operators

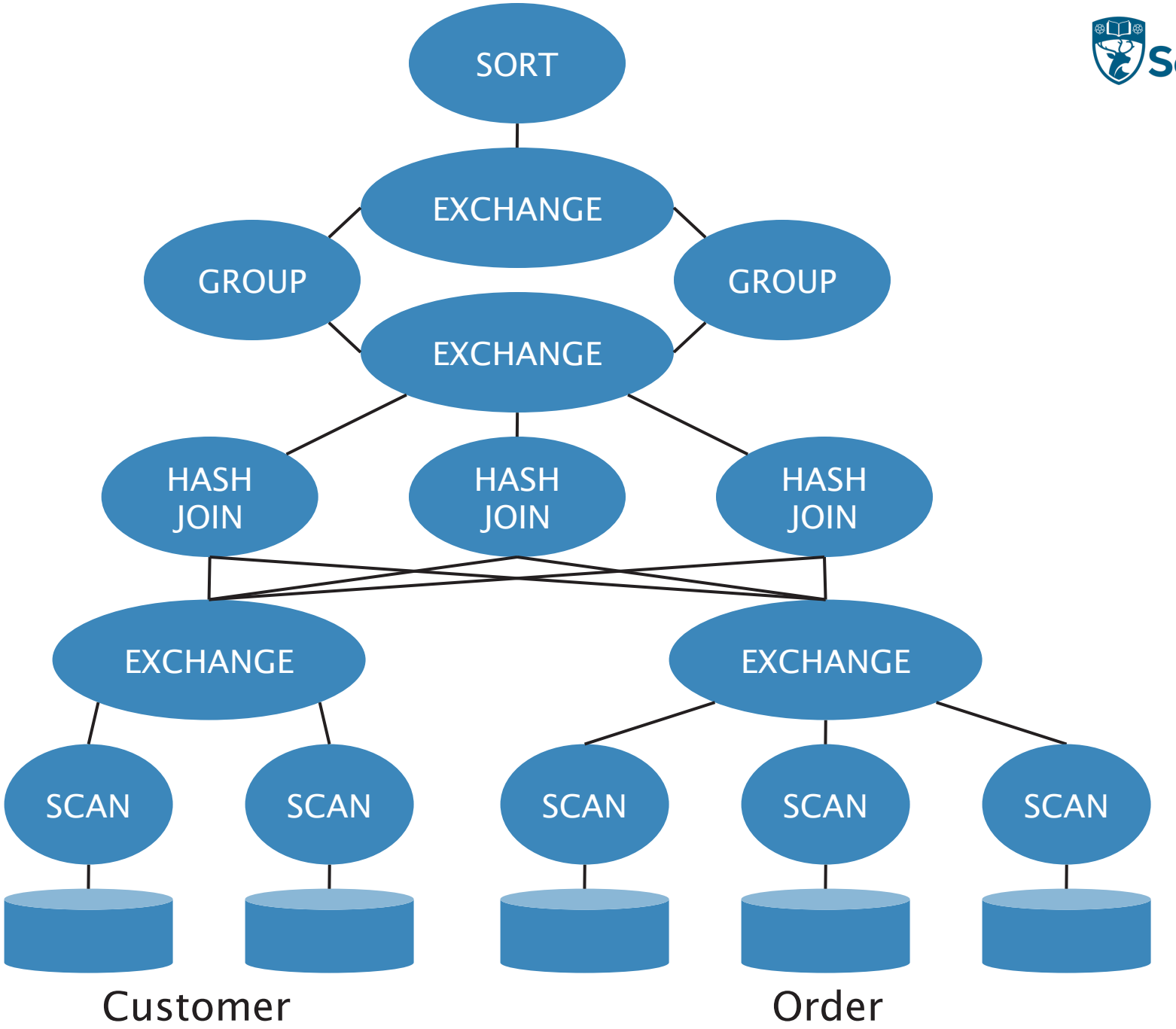


Exchange Operators



Exchange Operators



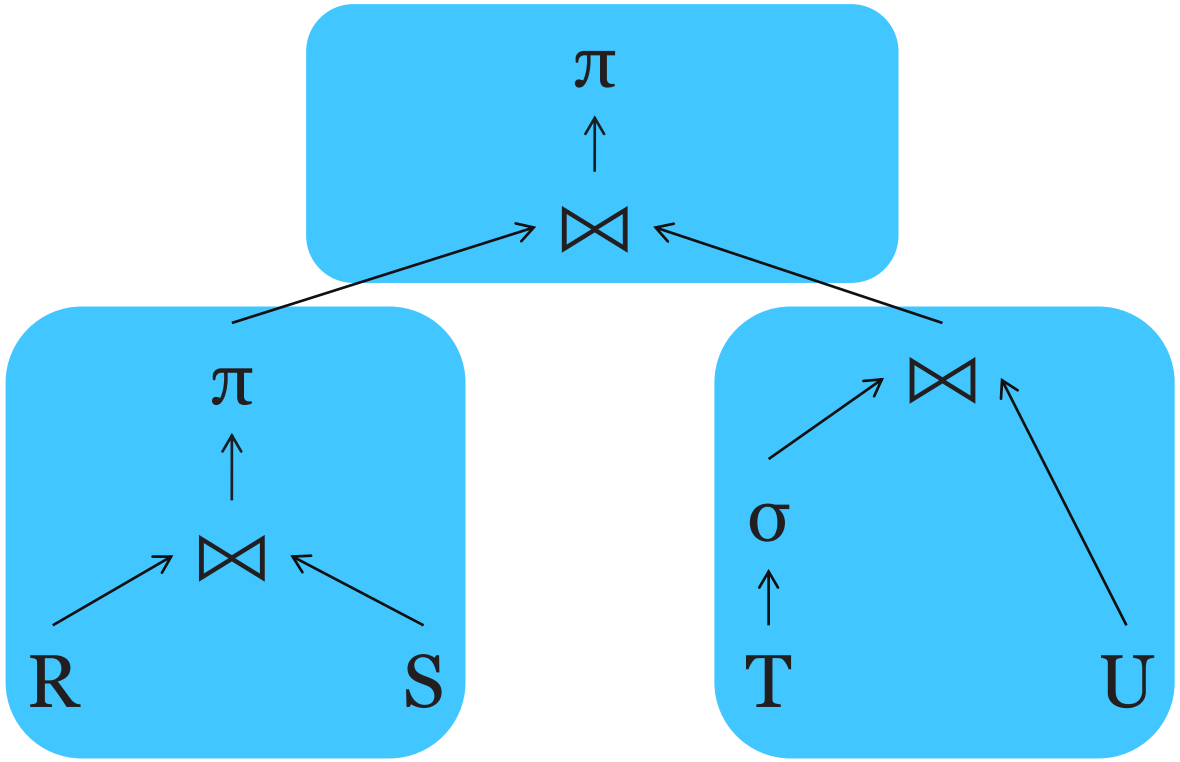

SORT
EXCHANGE
GROUP
GROUP
EXCHANGE
HASH JOIN
HASH JOIN
HASH JOIN
EXCHANGE
EXCHANGE
SCAN
SCAN
SCAN
SCAN
SCAN
Customer
Order

University of Southampton

Bushy Parallelism

Bushy Parallelism

Execute subtrees concurrently



University of Southampton

Parallel Query Processing

University of Southampton

Some Parallel Queries

- Enquiry
- Co-located Join
- Directed Join
- Broadcast Join
- Repartitioned Join

Combine aspects of intra-operator and bushy parallelism

Orders Database

CUSTOMER

CKEY	CNAME	…	CNATION	…

ORDER

OKEY	DATE	…	CKEY	…	SKEY	…

SUPPLIER

SKEY	SNAME	…	SNATION	…

University of Southampton

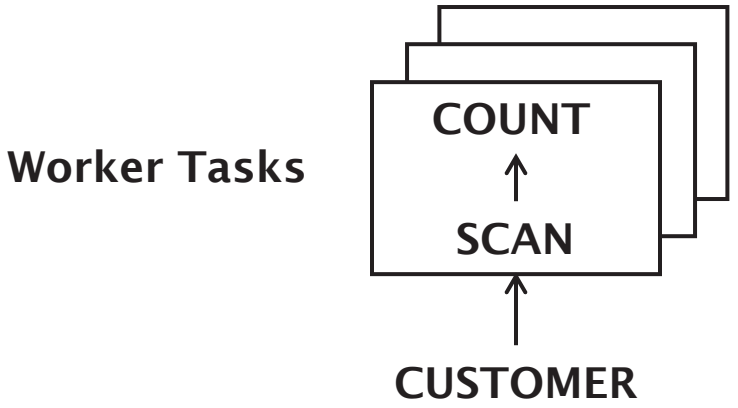
Enquiry/Query without join

"How many customers live in the UK?"

Enquiry/Query without join

"How many customers live in the UK?"

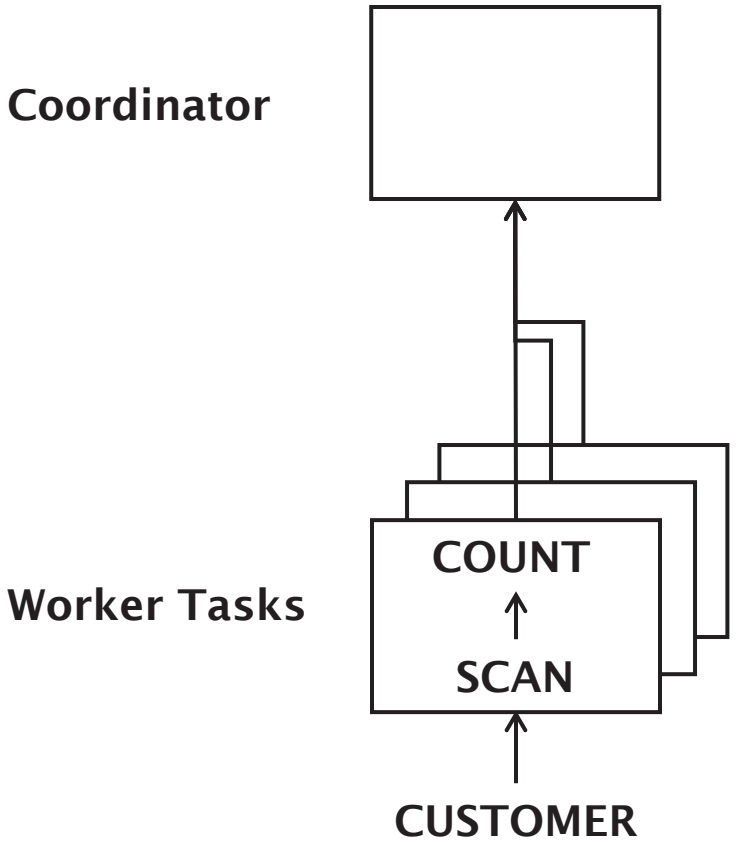
1. Count matching tuples in each partition of CUSTOMER



Enquiry/Query without join

"How many customers live in the UK?"

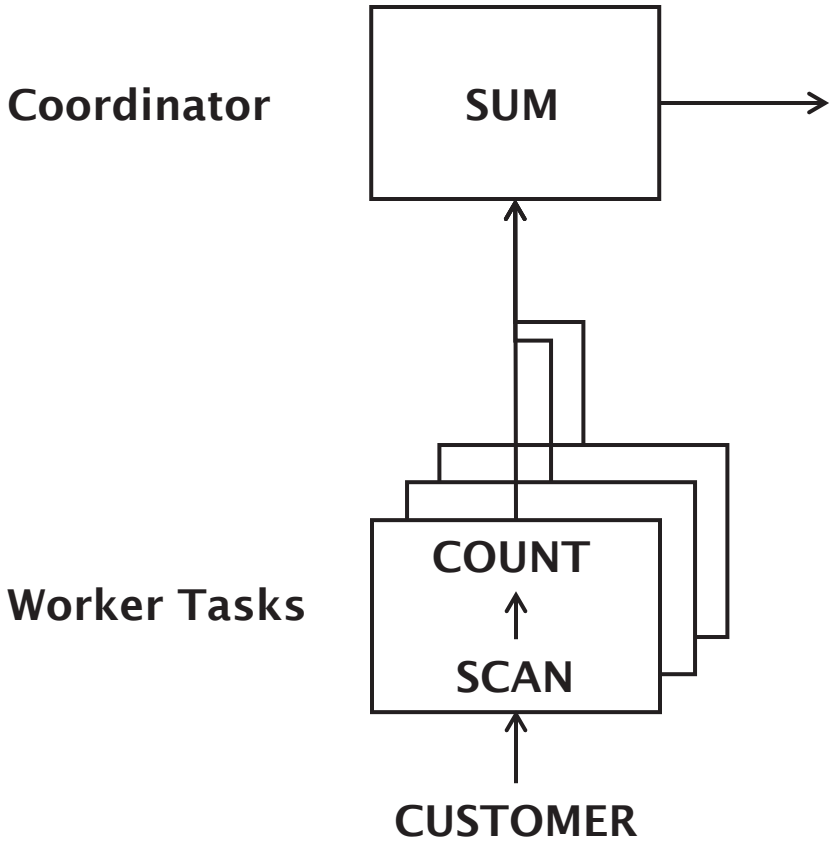
1. Count matching tuples in each partition of CUSTOMER
2. Pass counts to coordinator



Enquiry/Query without join

"How many customers live in the UK?"

1. Count matching tuples in each partition of CUSTOMER
2. Pass counts to coordinator
3. Sum counts and return



University of Southampton

Co-located join

“Which customers placed orders in July?”

University of Southampton

Co-located join

“Which customers placed orders in July?”

ORDER, CUSTOMER partitioned on CKEY

Therefore, corresponding entries are on the same node

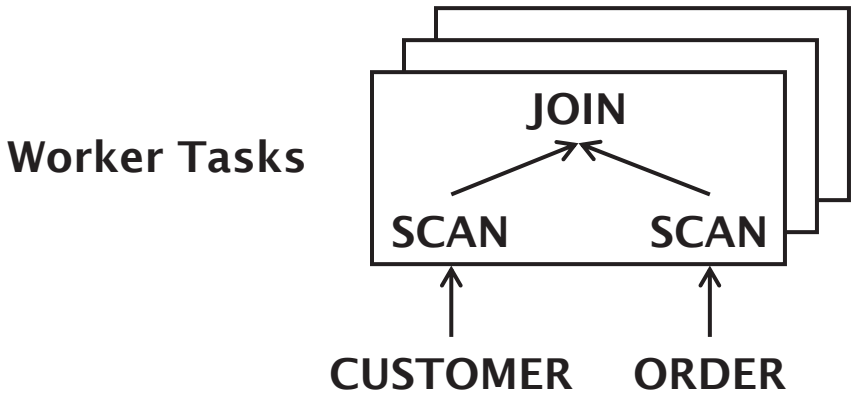
Co-located join

"Which customers placed orders in July?"

ORDER, CUSTOMER partitioned on CKEY

Therefore, corresponding entries are on the same node

1. Join CUSTOMER and ORDER on each partition



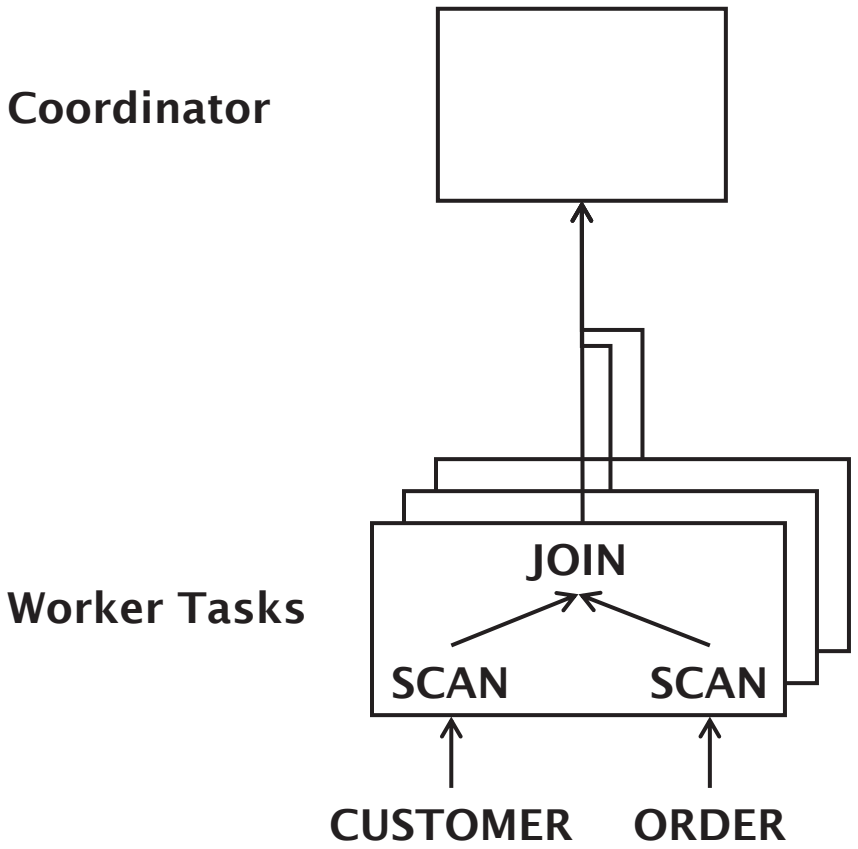
Co-located join

"Which customers placed orders in July?"

ORDER, CUSTOMER partitioned on CKEY

Therefore, corresponding entries are on the same node

1. Join CUSTOMER and ORDER on each partition
2. Pass joined relations to coordinator



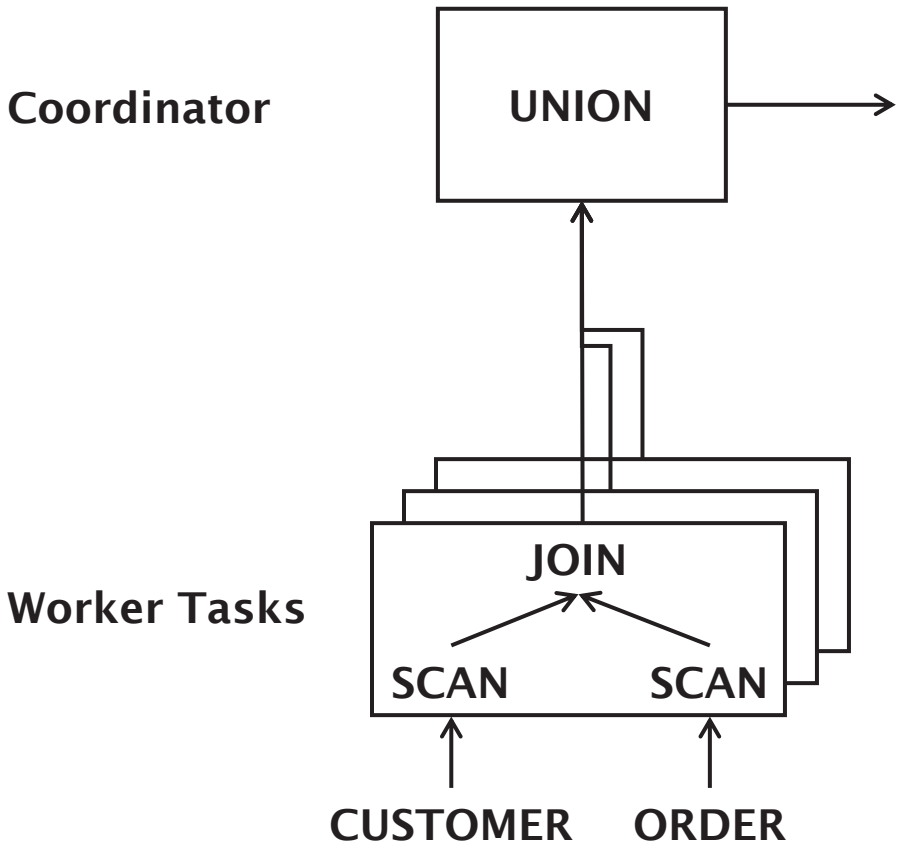
Co-located join

"Which customers placed orders in July?"

ORDER, CUSTOMER partitioned on CKEY

Therefore, corresponding entries are on the same node

1. Join CUSTOMER and ORDER on each partition
2. Pass joined relations to coordinator
3. Take union and return



University of Southampton

Directed join (Parallel associative join)

“Which customers placed orders in July?”

Directed join (Parallel associative join)

“Which customers placed orders in July?”

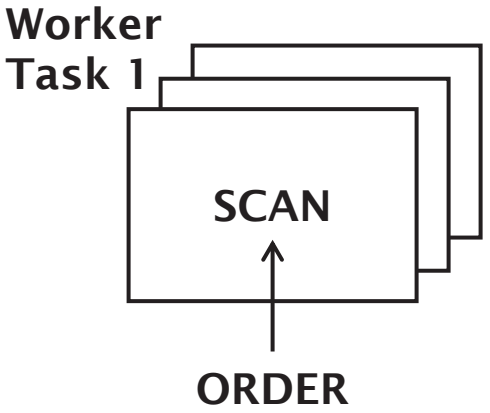
ORDER partitioned on OKEY
CUSTOMER partitioned on CKEY

Directed join (Parallel associative join)

"Which customers placed orders in July?"

ORDER partitioned on OKEY
CUSTOMER partitioned on CKEY

1. Scan ORDER on each partition

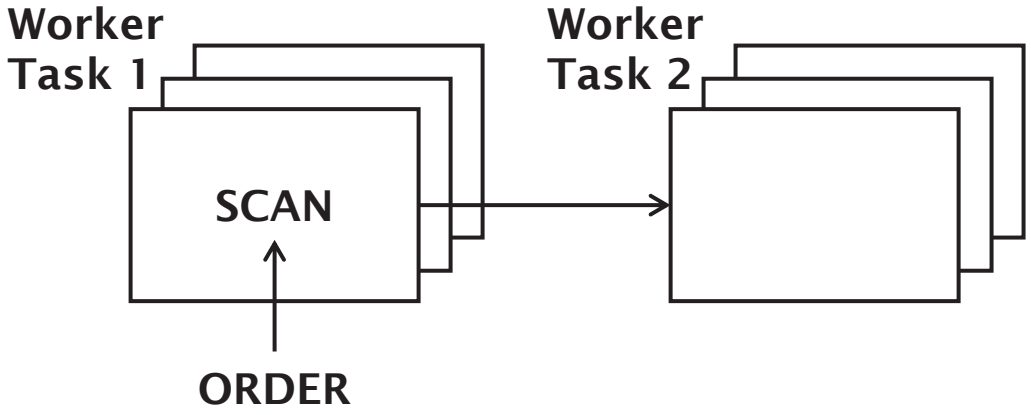


Directed join (Parallel associative join)

"Which customers placed orders in July?"

ORDER partitioned on OKEY
CUSTOMER partitioned on CKEY

1. Scan ORDER on each partition
2. Send tuples to appropriate CUSTOMER node based on ORDER.CKEY

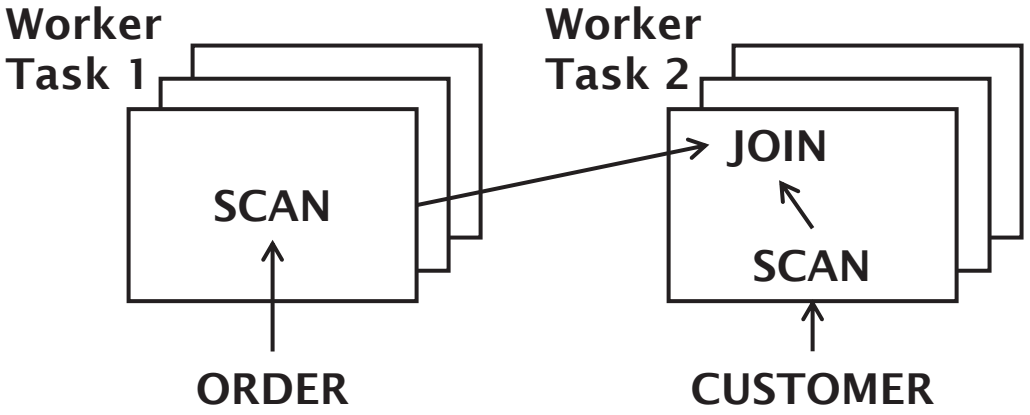


Directed join (Parallel associative join)

"Which customers placed orders in July?"

ORDER partitioned on OKEY
CUSTOMER partitioned on CKEY

1. Scan ORDER on each partition
2. Send tuples to appropriate CUSTOMER node based on ORDER.CKEY
3. Join ORDER tuples with each CUSTOMER fragment

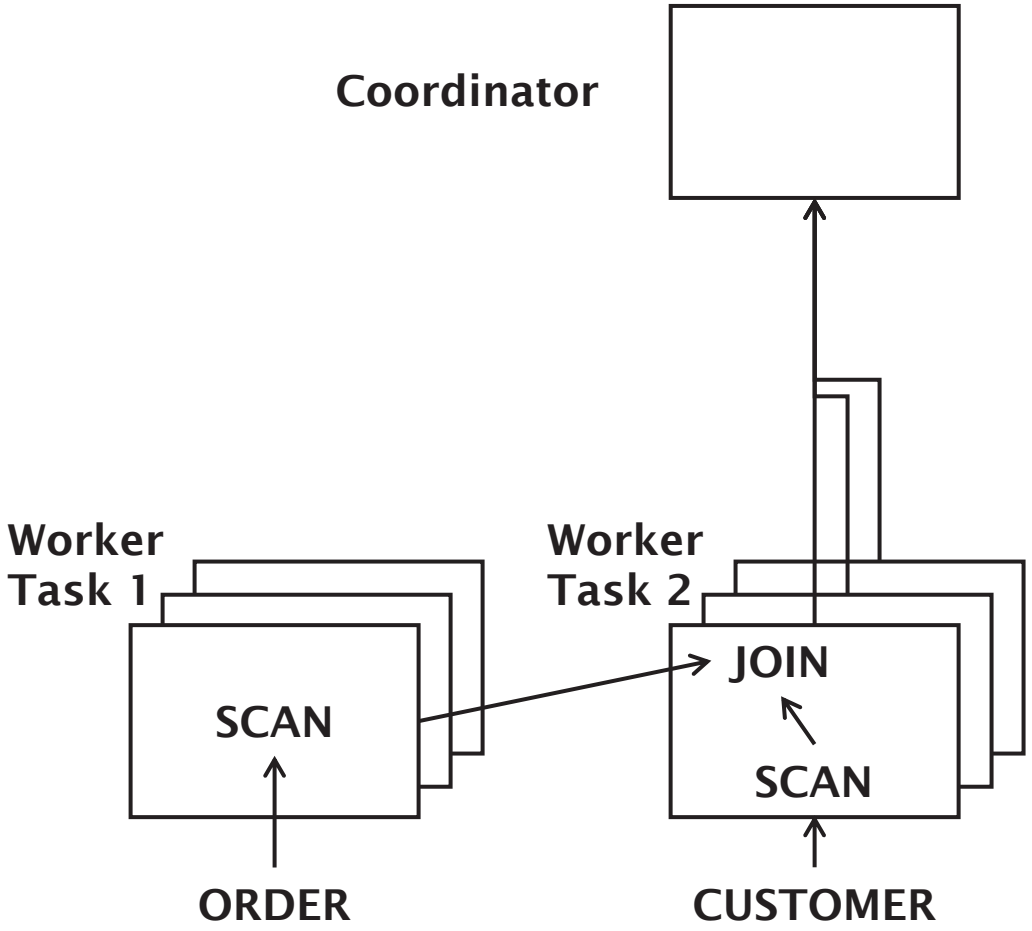


Directed join (Parallel associative join)

"Which customers placed orders in July?"

ORDER partitioned on OKEY
CUSTOMER partitioned on CKEY

1. Scan ORDER on each partition
2. Send tuples to appropriate CUSTOMER node based on ORDER.CKEY
3. Join ORDER tuples with each CUSTOMER fragment
4. Send joined relations to coordinator

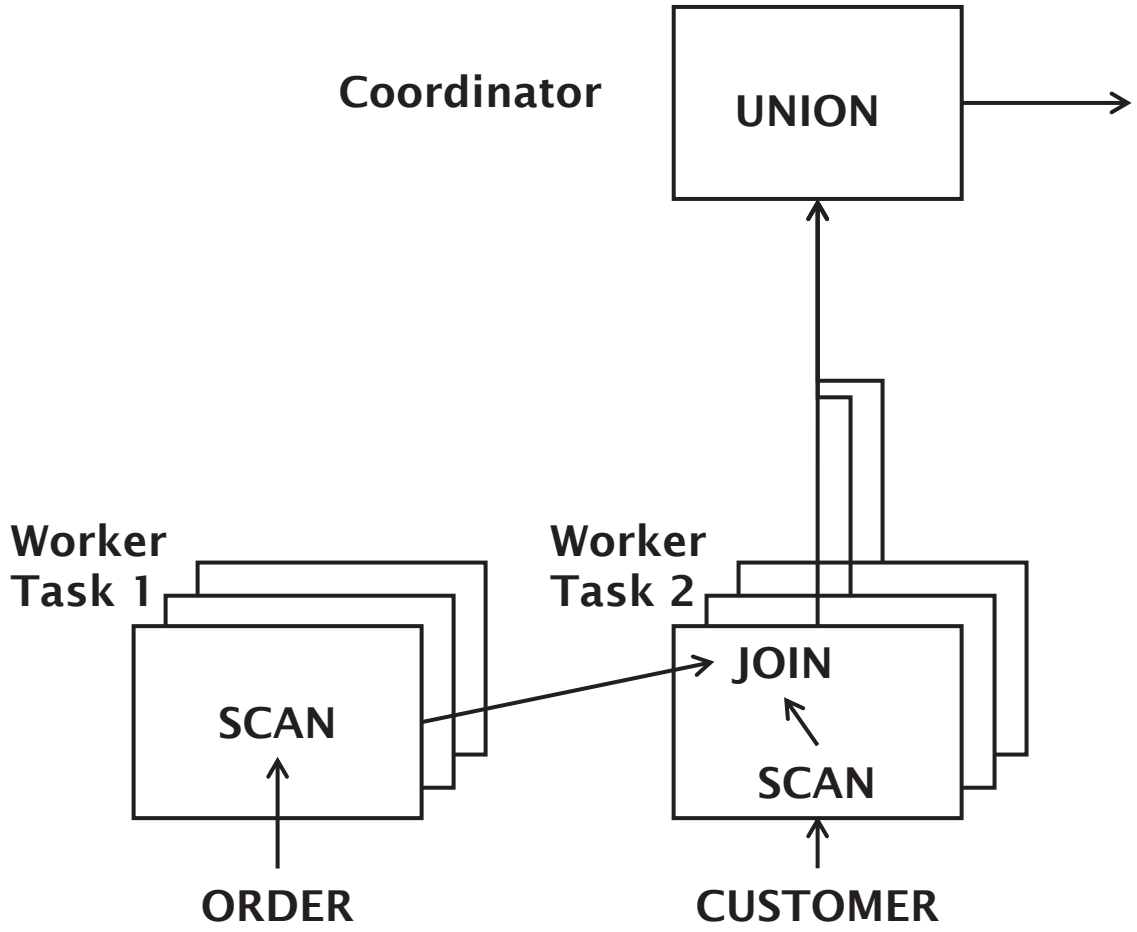


Directed join (Parallel associative join)

"Which customers placed orders in July?"

ORDER partitioned on OKEY
CUSTOMER partitioned on CKEY

1. Scan ORDER on each partition
2. Send tuples to appropriate CUSTOMER node based on ORDER.CKEY
3. Join ORDER tuples with each CUSTOMER fragment
4. Send joined relations to coordinator
5. Take union and return



Broadcast join (Parallel nested loop join)

“Which customers and suppliers are in the same country?”

Broadcast join (Parallel nested loop join)

"Which customers and suppliers are in the same country?"

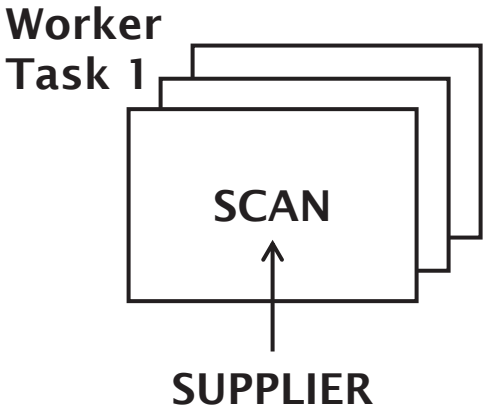
SUPPLIER partitioned on SKEY
CUSTOMER partitioned on CKEY
Join on CNATION=SNATION

Broadcast join (Parallel nested loop join)

"Which customers and suppliers are in the same country?"

SUPPLIER partitioned on SKEY
CUSTOMER partitioned on CKEY
Join on CNATION=SNATION

1. Scan SUPPLIER on each partition

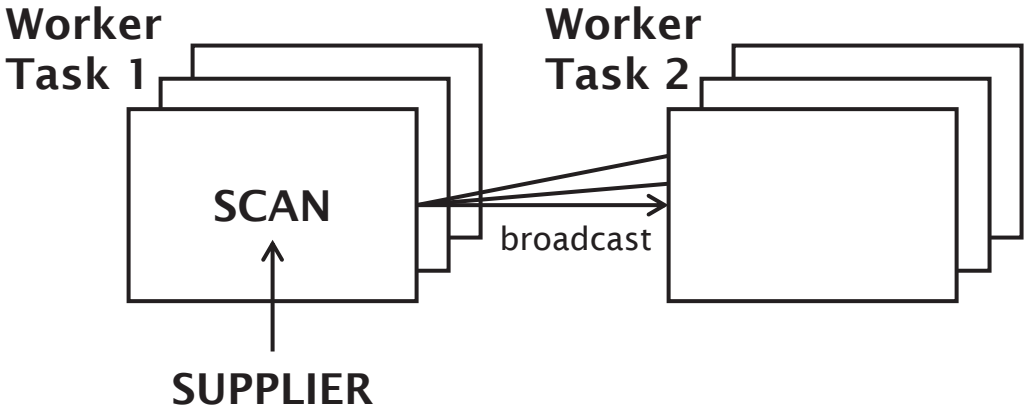


Broadcast join (Parallel nested loop join)

"Which customers and suppliers are in the same country?"

SUPPLIER partitioned on SKEY
CUSTOMER partitioned on CKEY
Join on CNATION=SNATION

1. Scan SUPPLIER on each partition
2. Send tuples to all CUSTOMER nodes

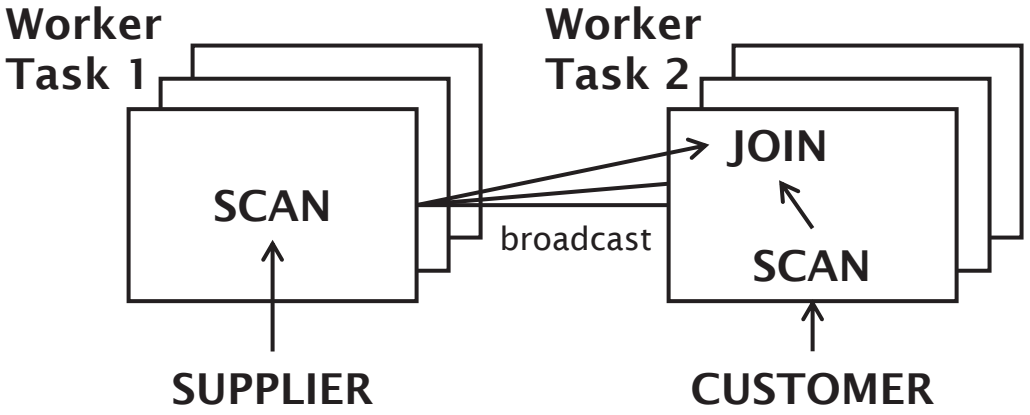


Broadcast join (Parallel nested loop join)

"Which customers and suppliers are in the same country?"

SUPPLIER partitioned on SKEY
CUSTOMER partitioned on CKEY
Join on CNATION=SNATION

1. Scan SUPPLIER on each partition
2. Send tuples to all CUSTOMER nodes
3. Join SUPPLIER tuples with each CUSTOMER fragment

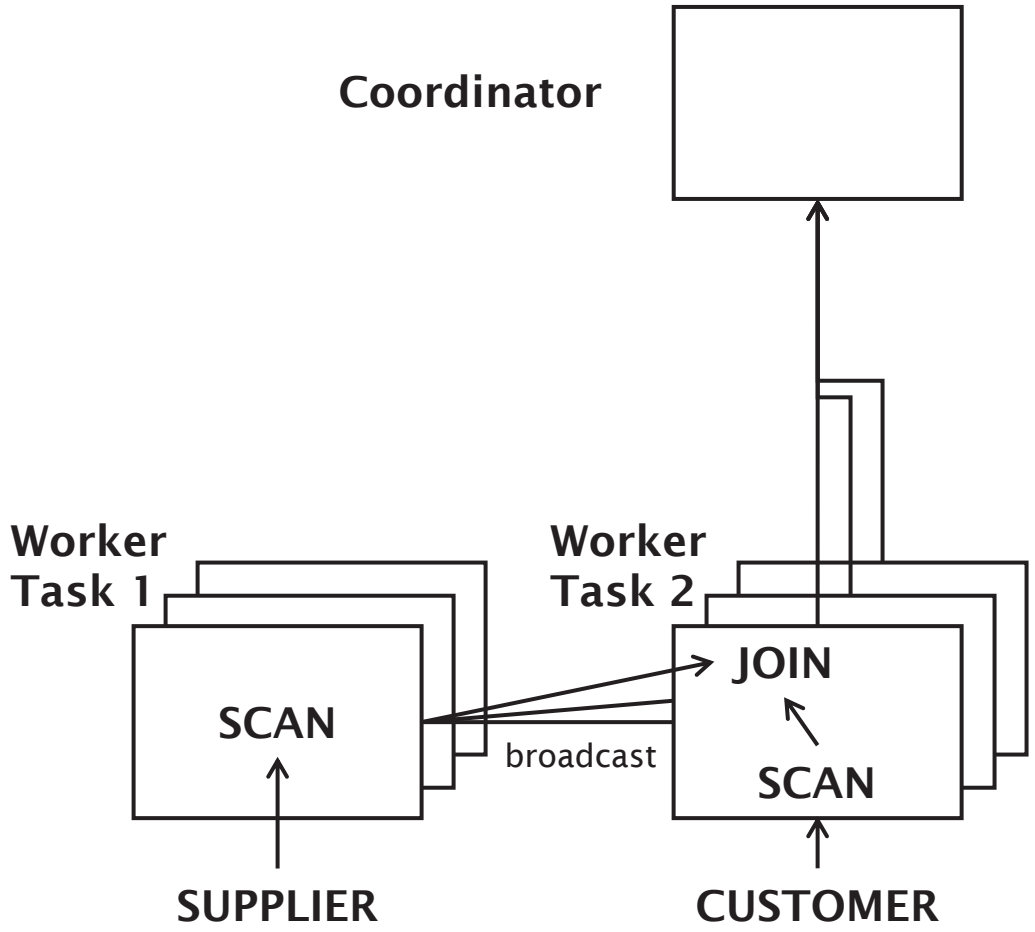


Broadcast join (Parallel nested loop join)

"Which customers and suppliers are in the same country?"

SUPPLIER partitioned on SKEY
CUSTOMER partitioned on CKEY
Join on CNATION=SNATION

1. Scan SUPPLIER on each partition
2. Send tuples to all CUSTOMER nodes
3. Join SUPPLIER tuples with each CUSTOMER fragment
4. Send joined relations to coordinator

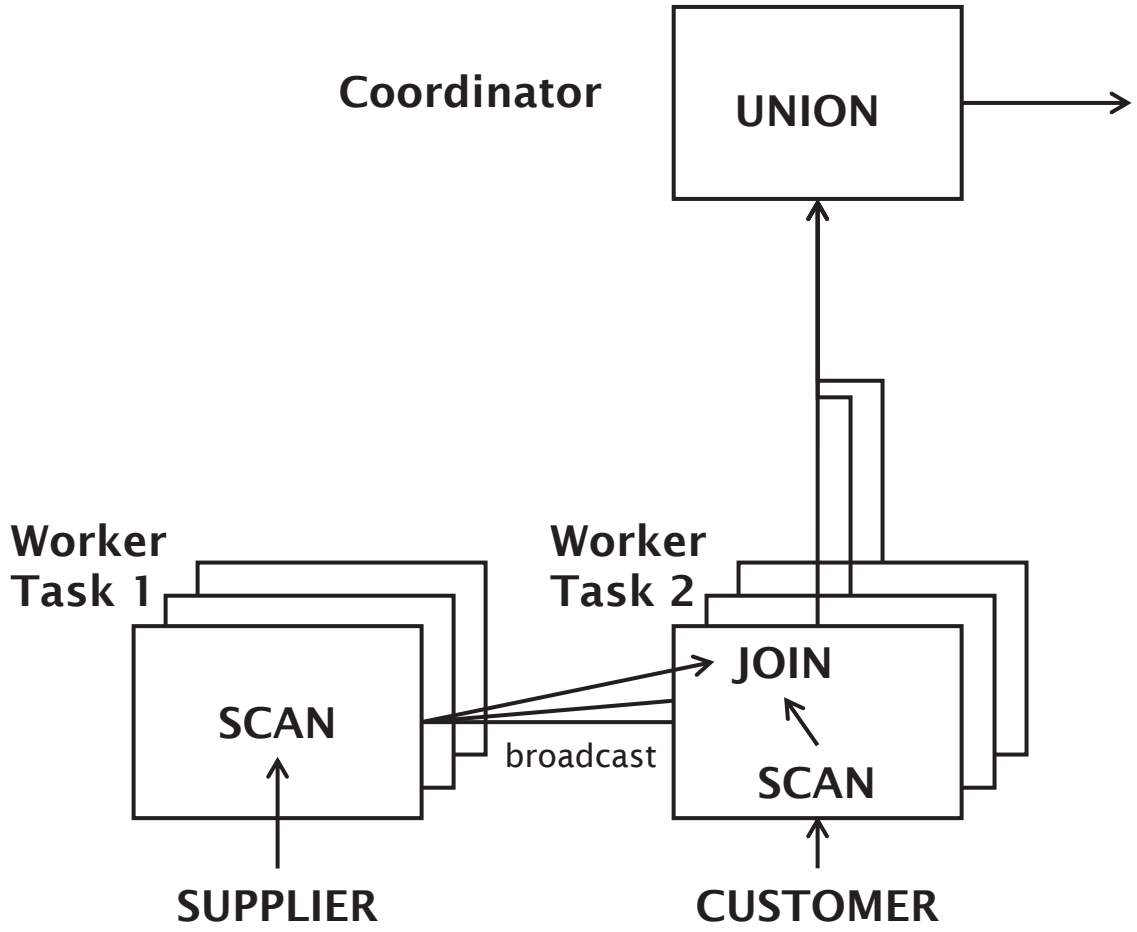


Broadcast join (Parallel nested loop join)

"Which customers and suppliers are in the same country?"

SUPPLIER partitioned on SKEY
CUSTOMER partitioned on CKEY
Join on CNATION=SNATION

1. Scan SUPPLIER on each partition
2. Send tuples to all CUSTOMER nodes
3. Join SUPPLIER tuples with each CUSTOMER fragment
4. Send joined relations to coordinator
5. Take union and return



Repartitioned join (Parallel hash join)

“Which customers and suppliers are in the same country?”

Repartitioned join (Parallel hash join)

"Which customers and suppliers are in the same country?"

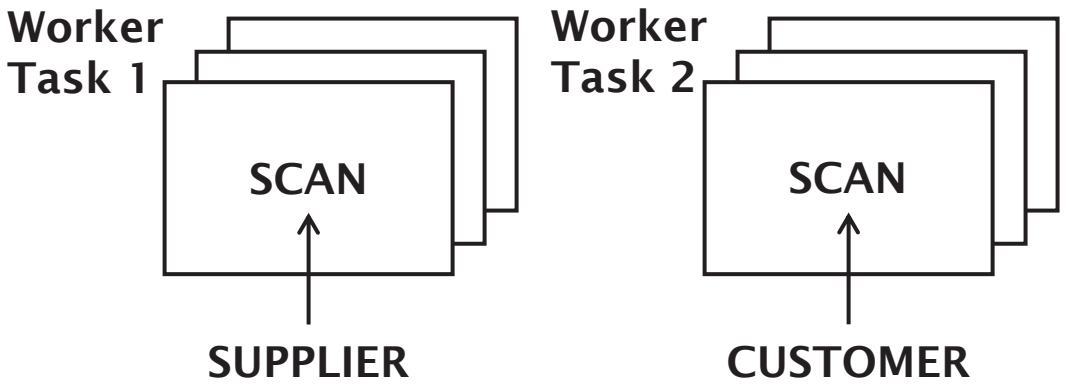
SUPPLIER partitioned on SKEY
CUSTOMER partitioned on CKEY
Join on CNATION=SNATION

Repartitioned join (Parallel hash join)

"Which customers and suppliers are in the same country?"

SUPPLIER partitioned on SKEY
CUSTOMER partitioned on CKEY
Join on CNATION=SNATION

1. Scan SUPPLIER, CUSTOMER

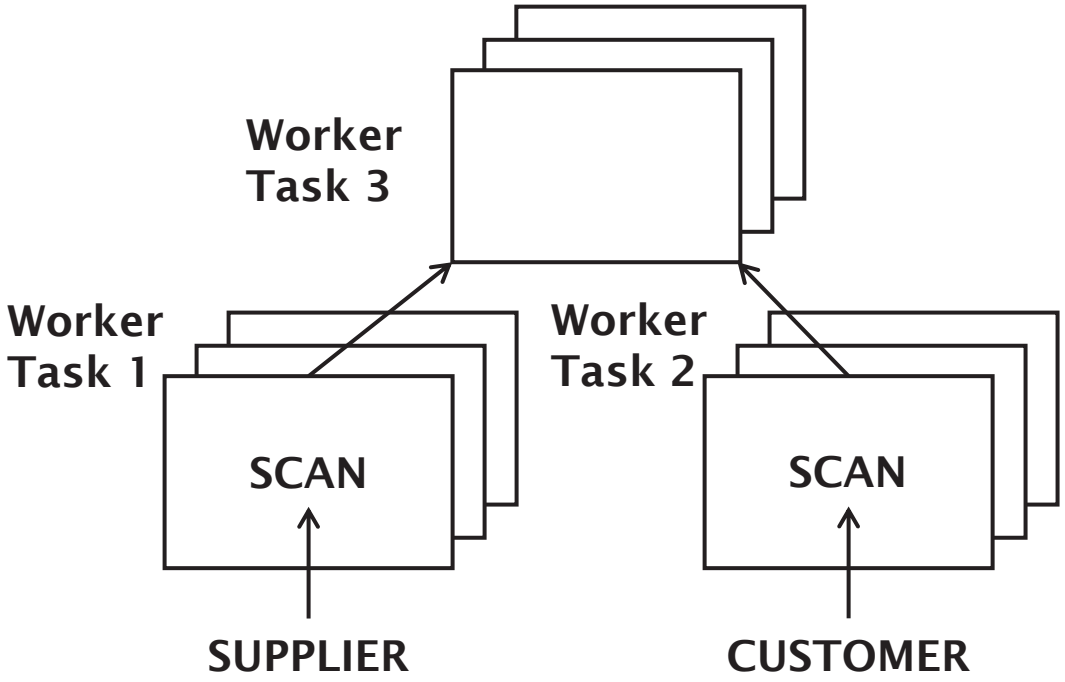


Repartitioned join (Parallel hash join)

"Which customers and suppliers are in the same country?"

SUPPLIER partitioned on SKEY
CUSTOMER partitioned on CKEY
Join on CNATION=SNATION

1. Scan SUPPLIER, CUSTOMER
2. Repartition on *NATION and send to appropriate worker for Task 3

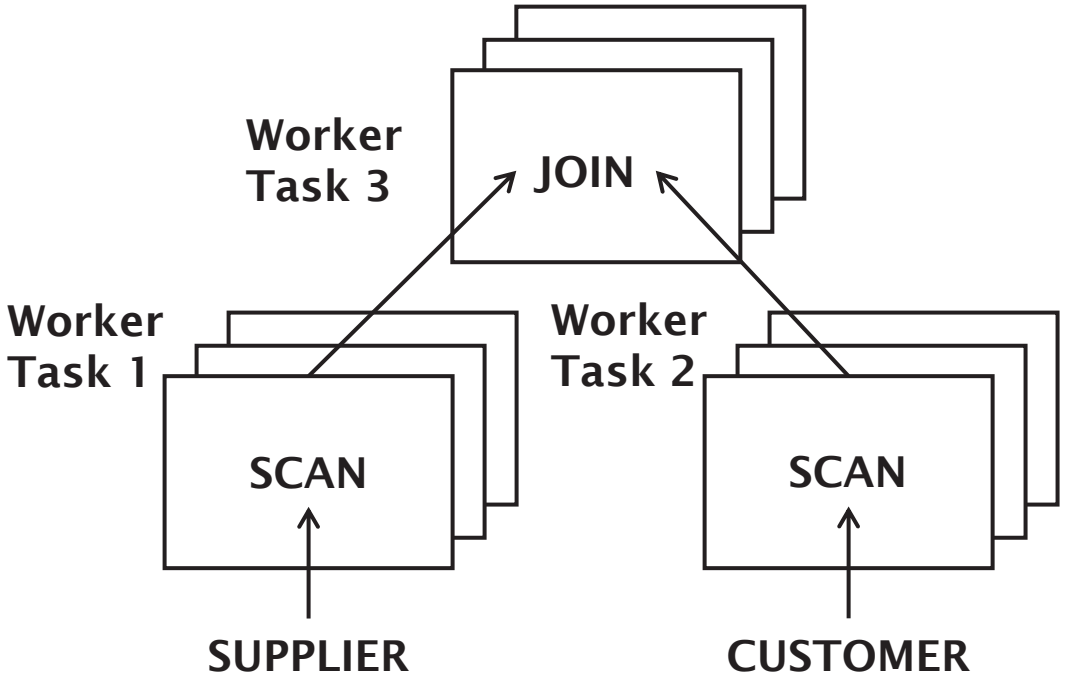


Repartitioned join (Parallel hash join)

"Which customers and suppliers are in the same country?"

SUPPLIER partitioned on SKEY
CUSTOMER partitioned on CKEY
Join on CNATION=SNATION

1. Scan SUPPLIER, CUSTOMER
2. Repartition on *NATION and send to appropriate worker for Task 3
3. Join SUPPLIER and CUSTOMER tuples

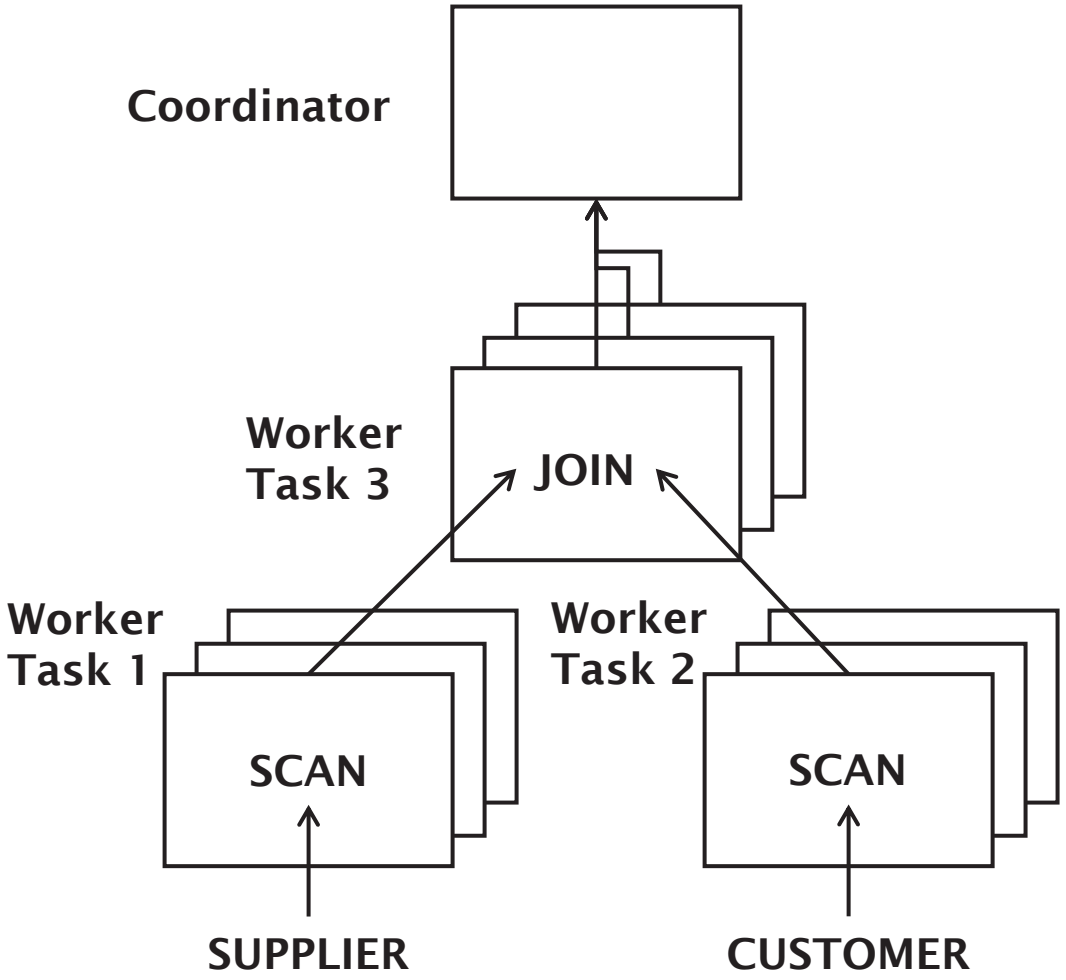


Repartitioned join (Parallel hash join)

"Which customers and suppliers are in the same country?"

SUPPLIER partitioned on SKEY
CUSTOMER partitioned on CKEY
Join on CNATION=SNATION

1. Scan SUPPLIER, CUSTOMER
2. Repartition on *NATION and send to appropriate worker for Task 3
3. Join SUPPLIER and CUSTOMER tuples
4. Send joined relations to coordinator

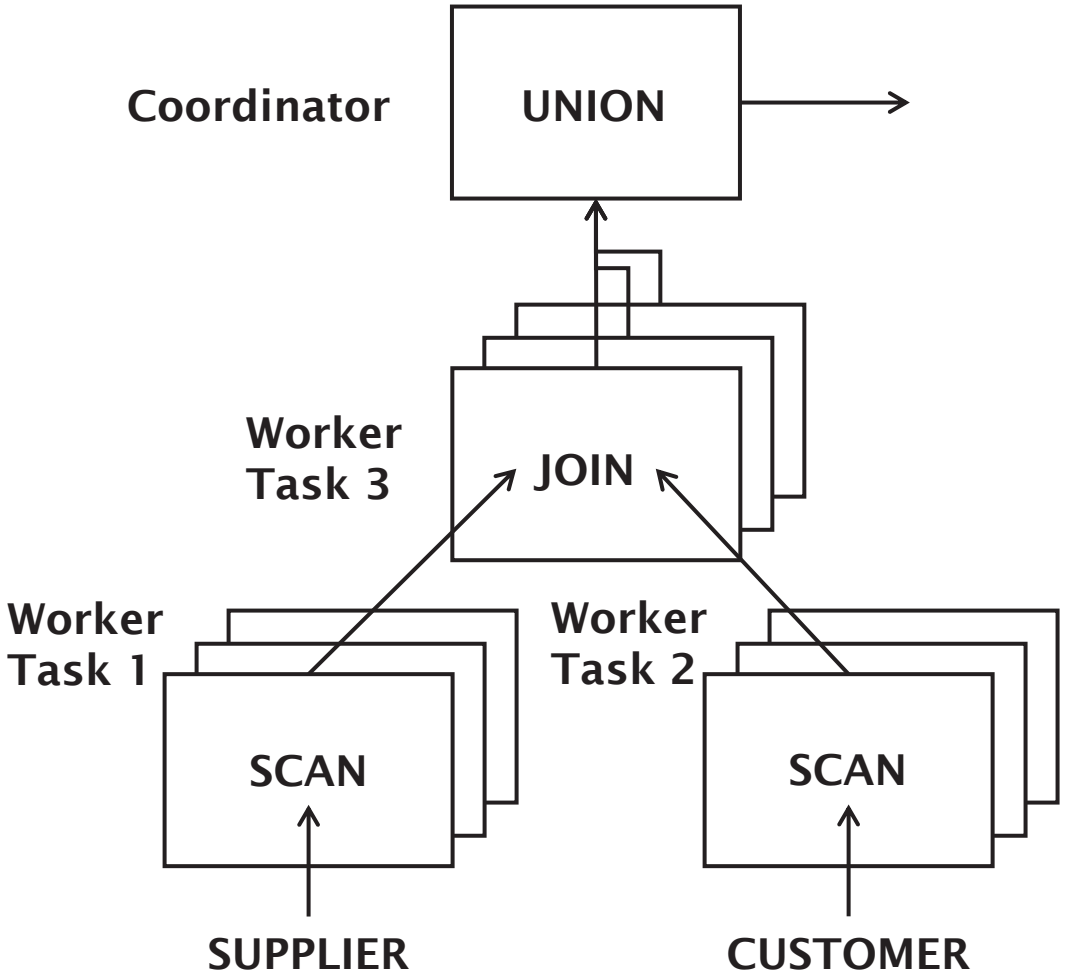


Repartitioned join (Parallel hash join)

"Which customers and suppliers are in the same country?"

SUPPLIER partitioned on SKEY
CUSTOMER partitioned on CKEY
Join on CNATION=SNATION

1. Scan SUPPLIER, CUSTOMER
2. Repartition on *NATION and send to appropriate worker for Task 3
3. Join SUPPLIER tuples CUSTOMER tuples
4. Send joined relations to coordinator
5. Take union and return



University of Southampton

Concurrency Control

University of Southampton

Concurrency and Parallelism

- A single transaction may update data in several different places
- Multiple transactions may be using the same (distributed) tables simultaneously
- One or several nodes could fail
- Requires concurrency control and recovery across multiple nodes for:
  - Locking and deadlock detection
  - Two-phase commit to ensure ‘all or nothing’

University of Southampton

Locking and Deadlocks

- With Shared Nothing architecture, each node is responsible for locking its own data
- No global locking mechanism
- However:
  - T1 locks item A on Node 1 and wants item B on Node 2
  - T2 locks item B on Node 2 and wants item A on Node 1
  - Distributed Deadlock

Resolving Deadlocks

Simple approach – Timeouts

1. Timeout T2, after wait exceeds a certain interval
   - Interval may need random element to avoid ‘chatter’
     i.e. both transactions give up at the same time and then try again
2. Rollback T2 to let T1 to proceed
3. Restart T2, which can now complete

University of Southampton

Resolving Deadlocks

More sophisticated approach (used by DB2)

- Each node maintains a local ‘wait-for’ graph
- Distributed deadlock detector (DDD) runs at the catalogue node for each database
- Periodically, all nodes send their graphs to the DDD
- DDD records all locks found in wait state
- Transaction becomes a candidate for termination if found in same lock wait state on two successive iterations

University of Southampton

Reliability

University of Southampton

Reliability

We wish to preserve the ACID properties for parallelised transactions

- Isolation is taken care of by 2PL protocol
- Isolation implies Consistency
- Durability can be taken care of node-by-node, with proper logging and recovery routines
- Atomicity is the hard part. We need to commit all parts of a transaction, or abort all parts

Two-phase commit protocol (2PC) is used to ensure that Atomicity is preserved

University of Southampton

Two-Phase Commit (2PC)

Distinguish between:

- The global transaction
- The local transactions into which the global transaction is decomposed

Global transaction is managed by a single site, known as the *coordinator*

Local transactions may be executed on separate sites, known as the *participants*

University of Southampton

Phase 1: Voting

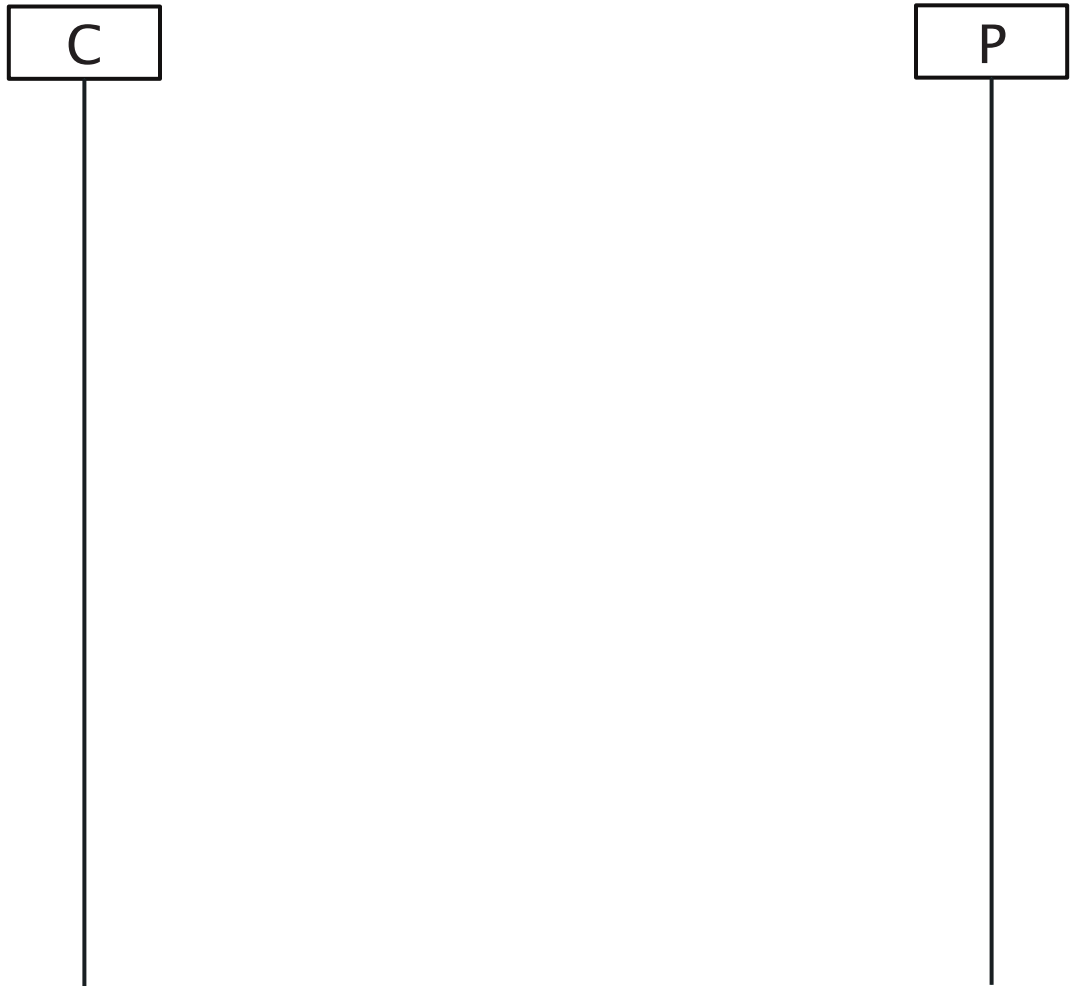
- Coordinator sends “prepare T” message to all participants
- Participants respond with either “vote-commit T” or “vote-abort T”
- Coordinator waits for participants to respond within a timeout period

University of Southampton

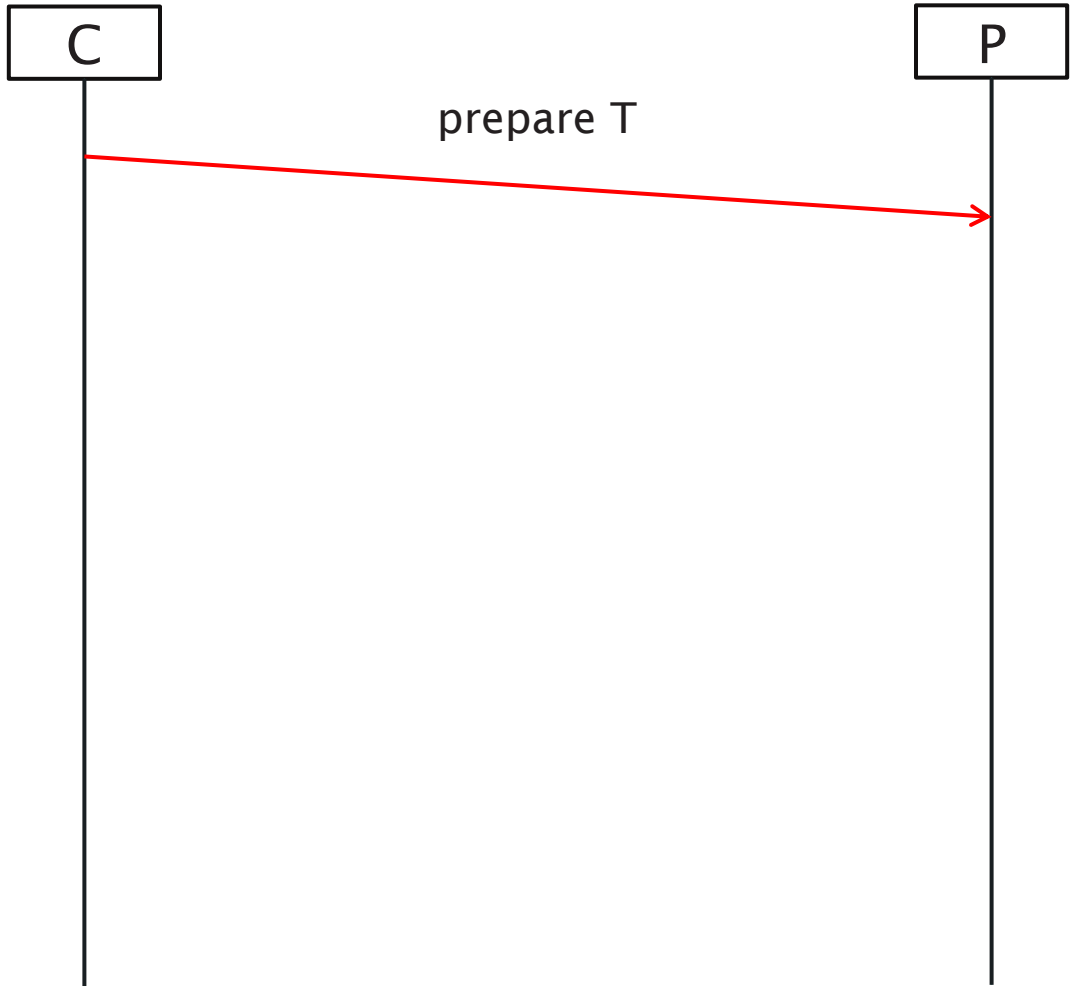
Phase 2: Decision

- If all participants return “vote-commit T” (to commit), send “commit T” to all participants. Wait for acknowledgements within timeout period.
- If any participant returns “vote-abort T”, send “abort T” to all participants. Wait for acknowledgements within timeout period.
- When all acknowledgements received, transaction is completed.
- If a site does not acknowledge, resend global decision until it is acknowledged.

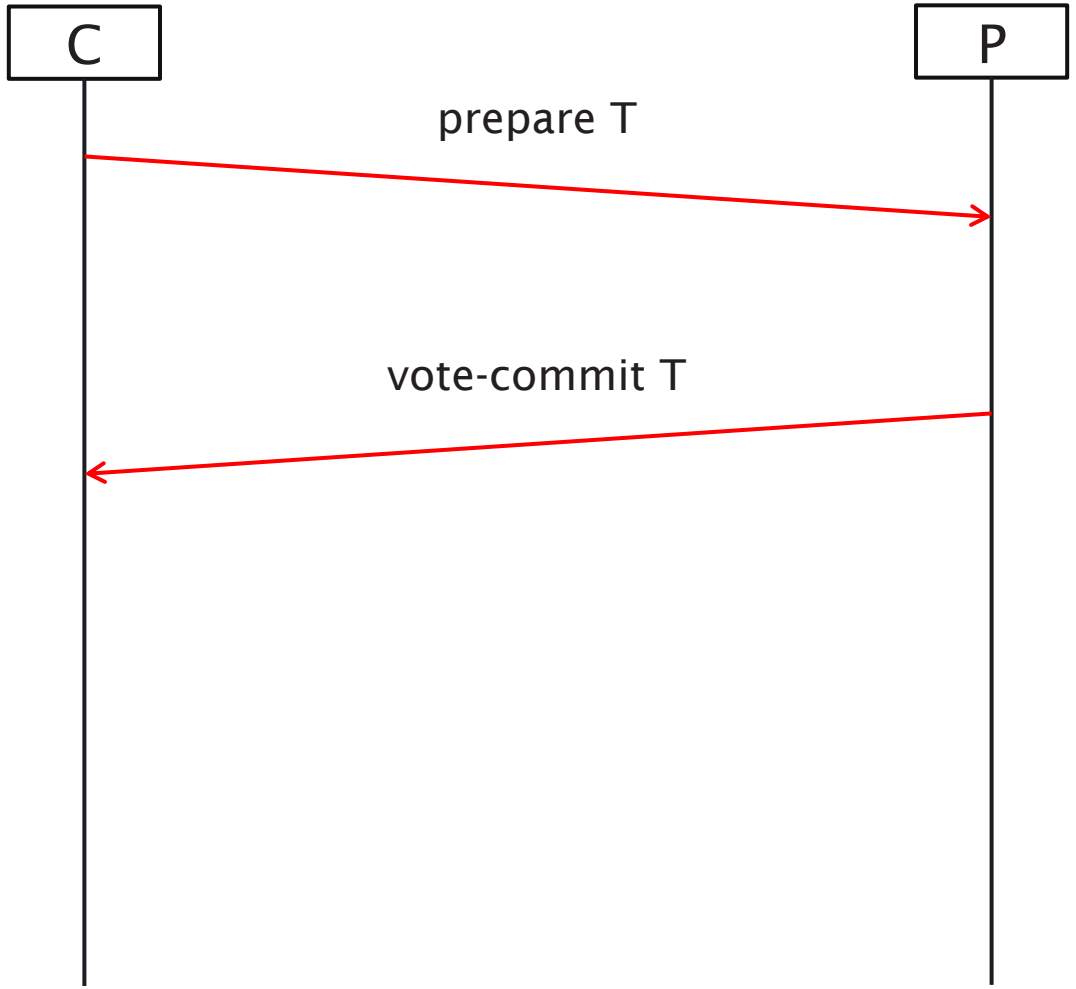
Normal Operation



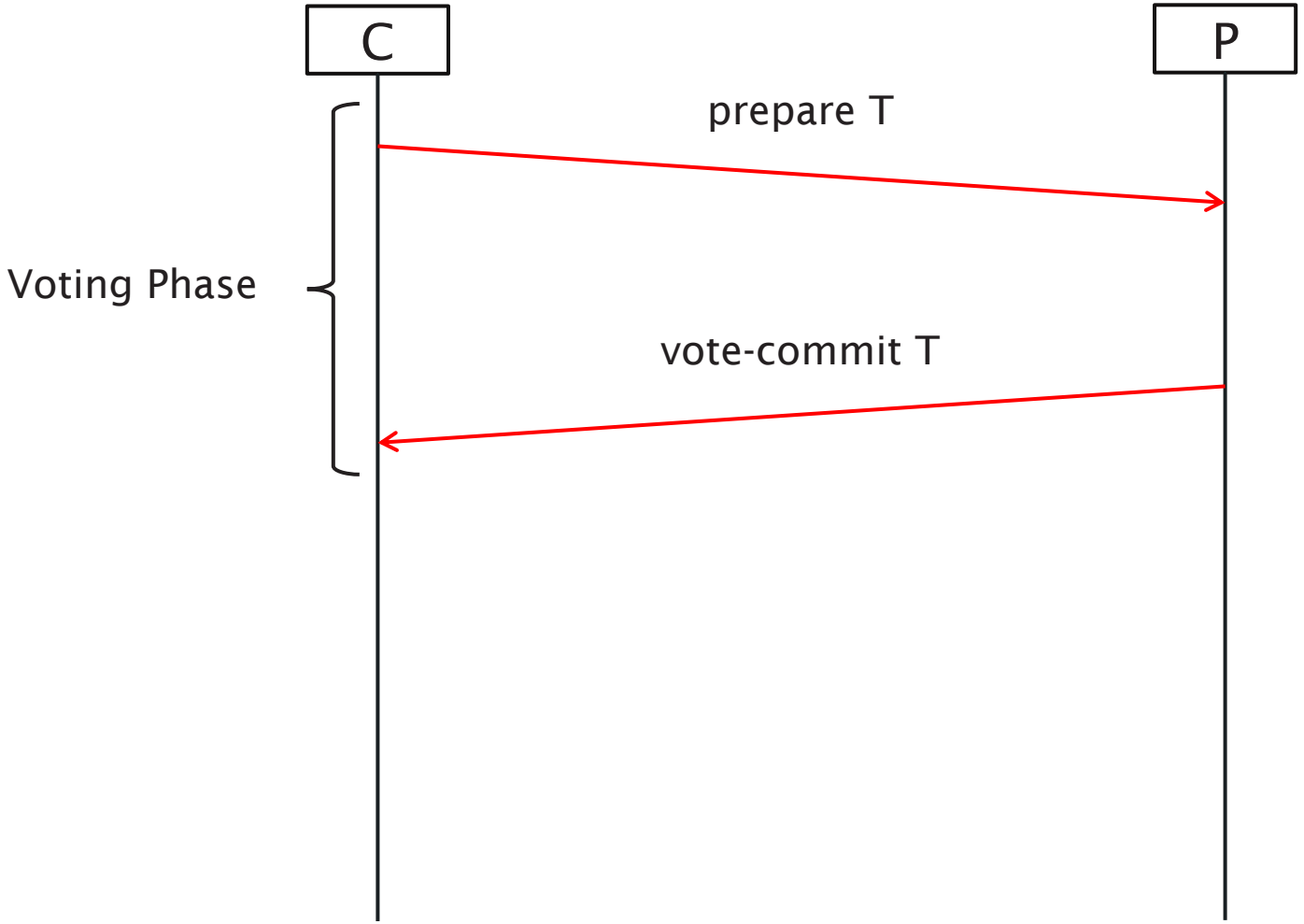
Normal Operation



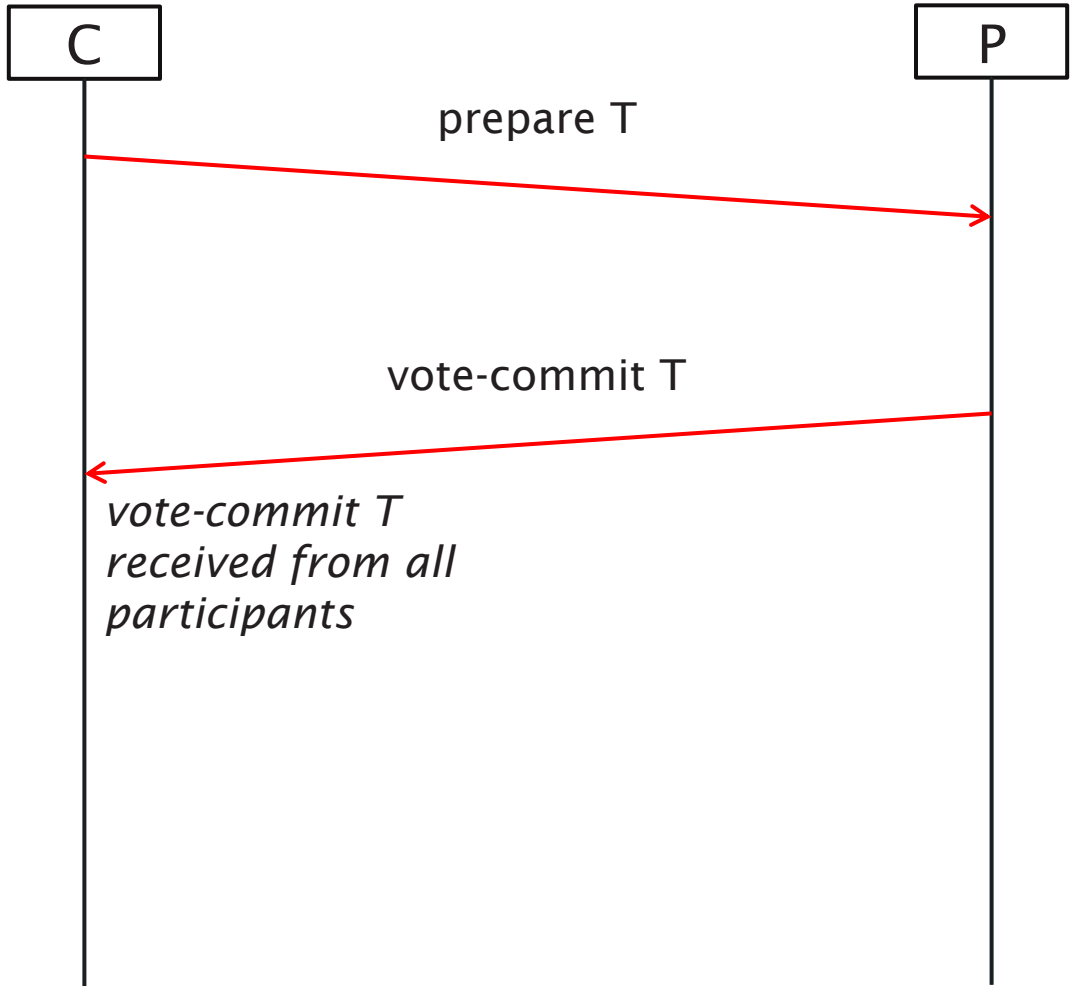
Normal Operation



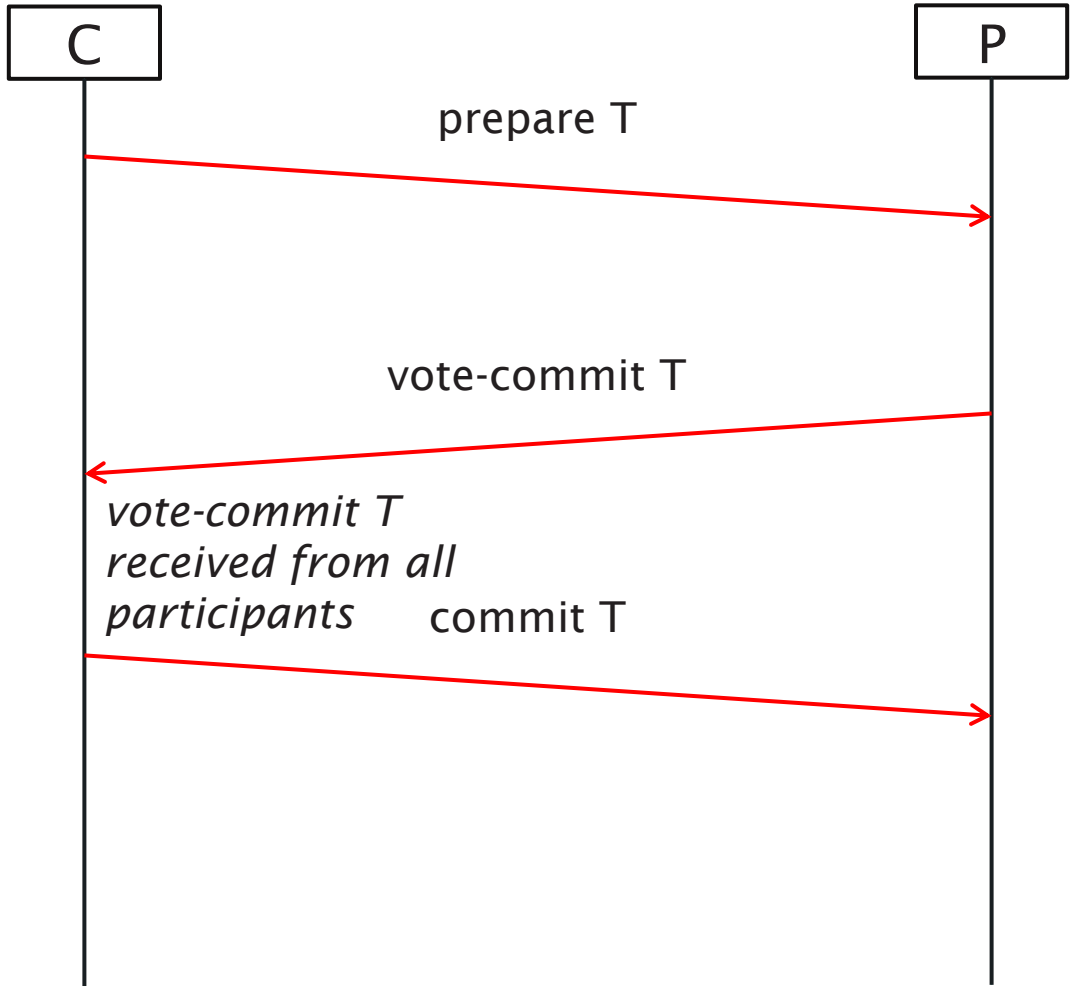
Normal Operation



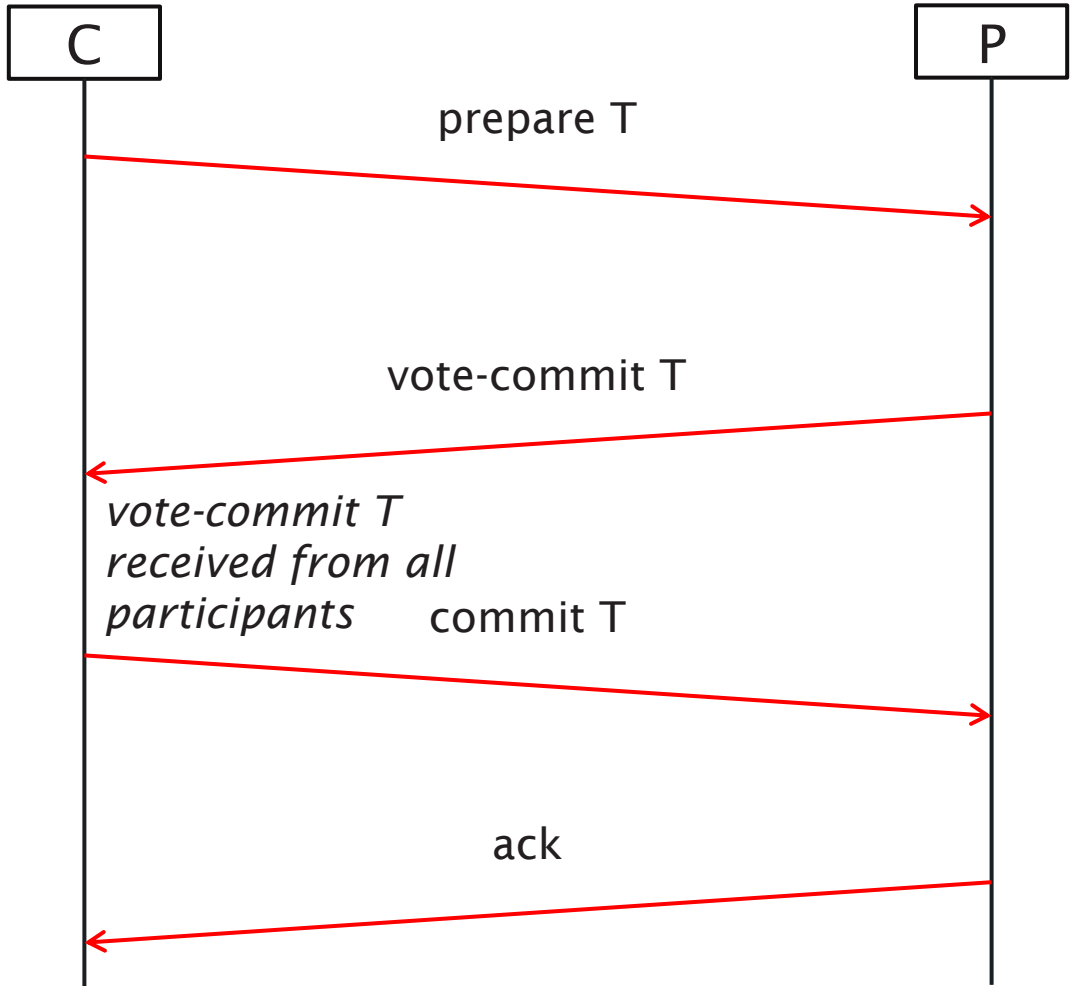
Normal Operation



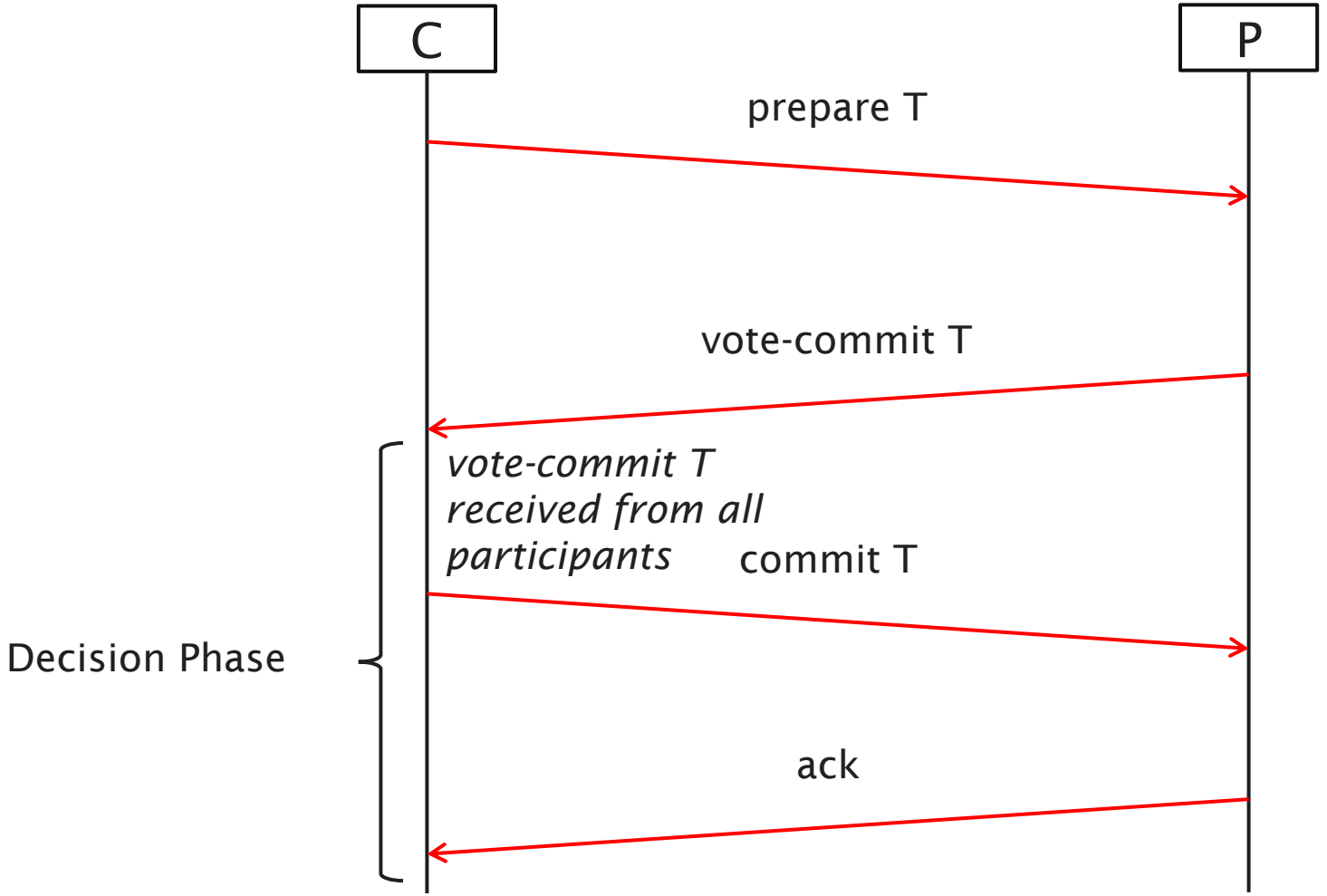
Normal Operation



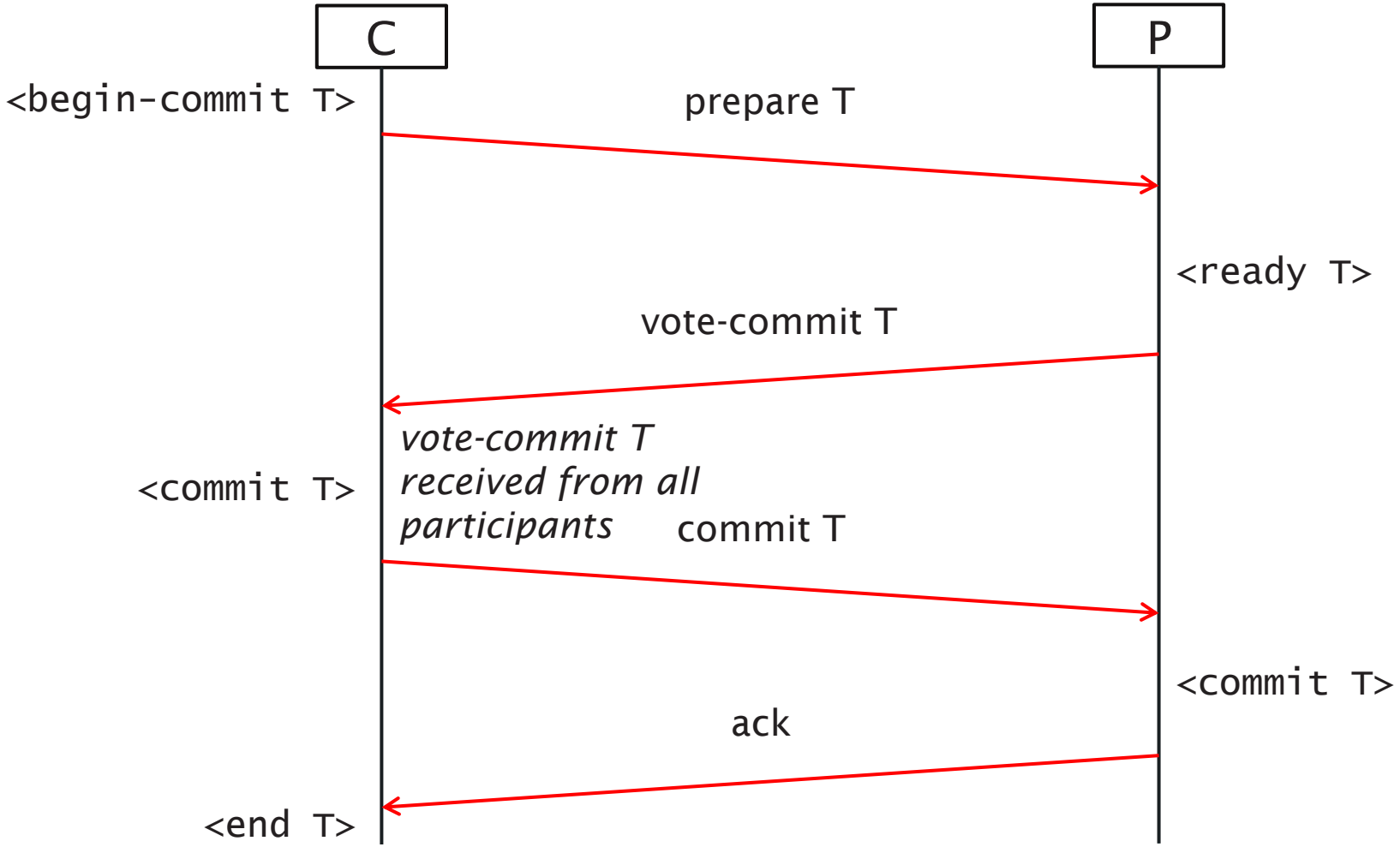
Normal Operation



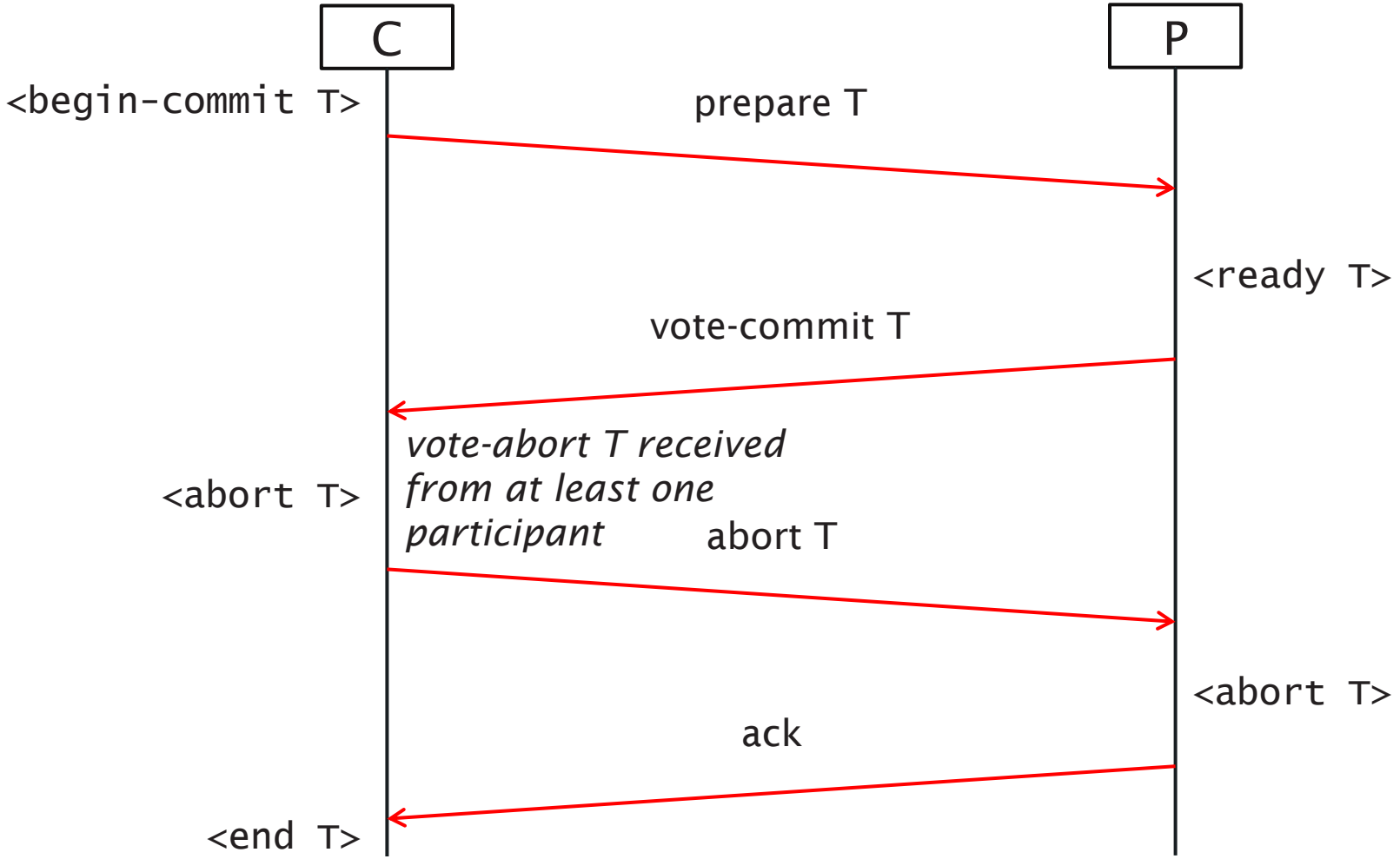
Normal Operation



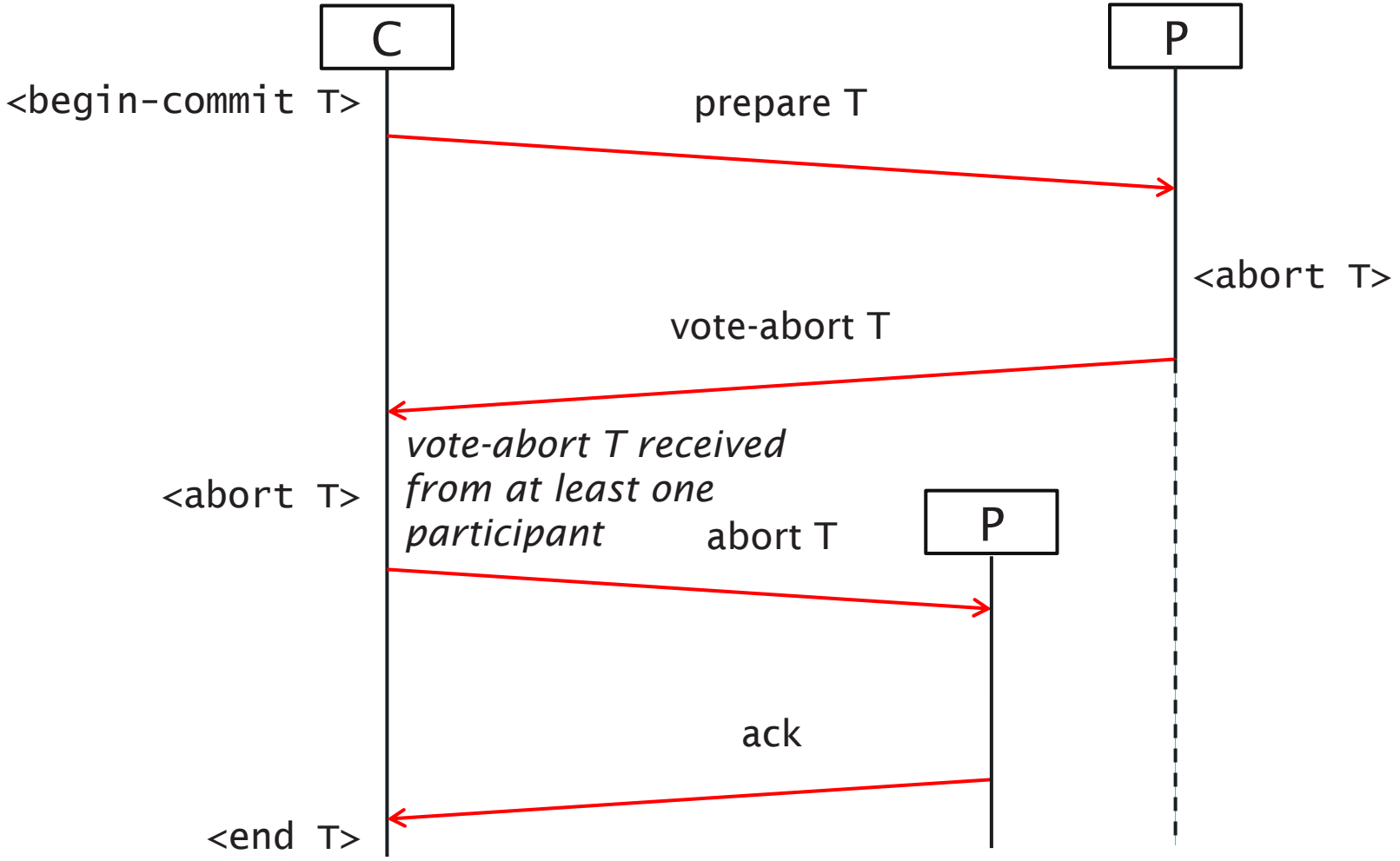
Logging


C
P
<begin-commit T>
prepare T
<ready T>
vote-commit T
vote-commit T
received from all
participants
<commit T>
commit T
<commit T>
ack
<end T>

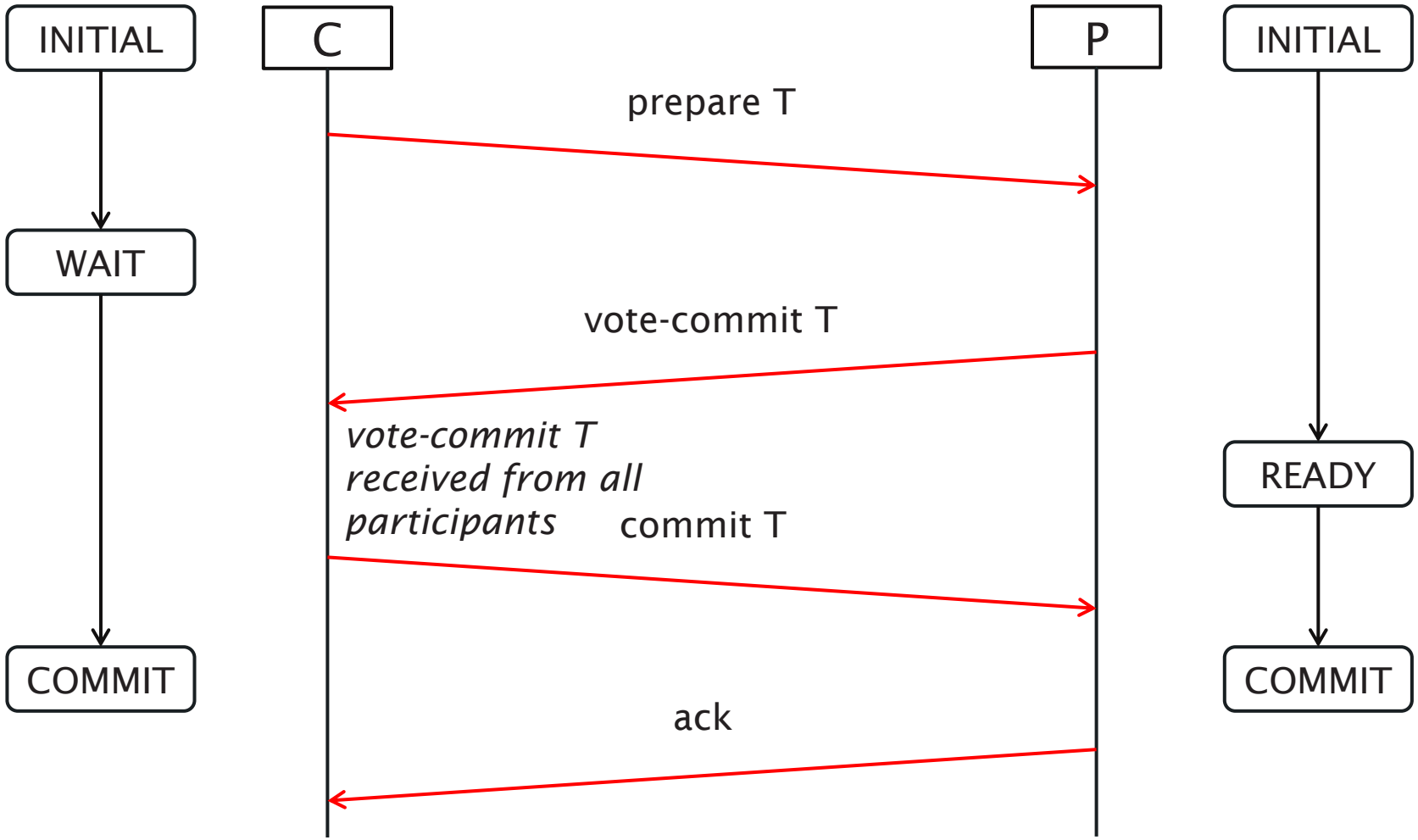
Aborted Transaction



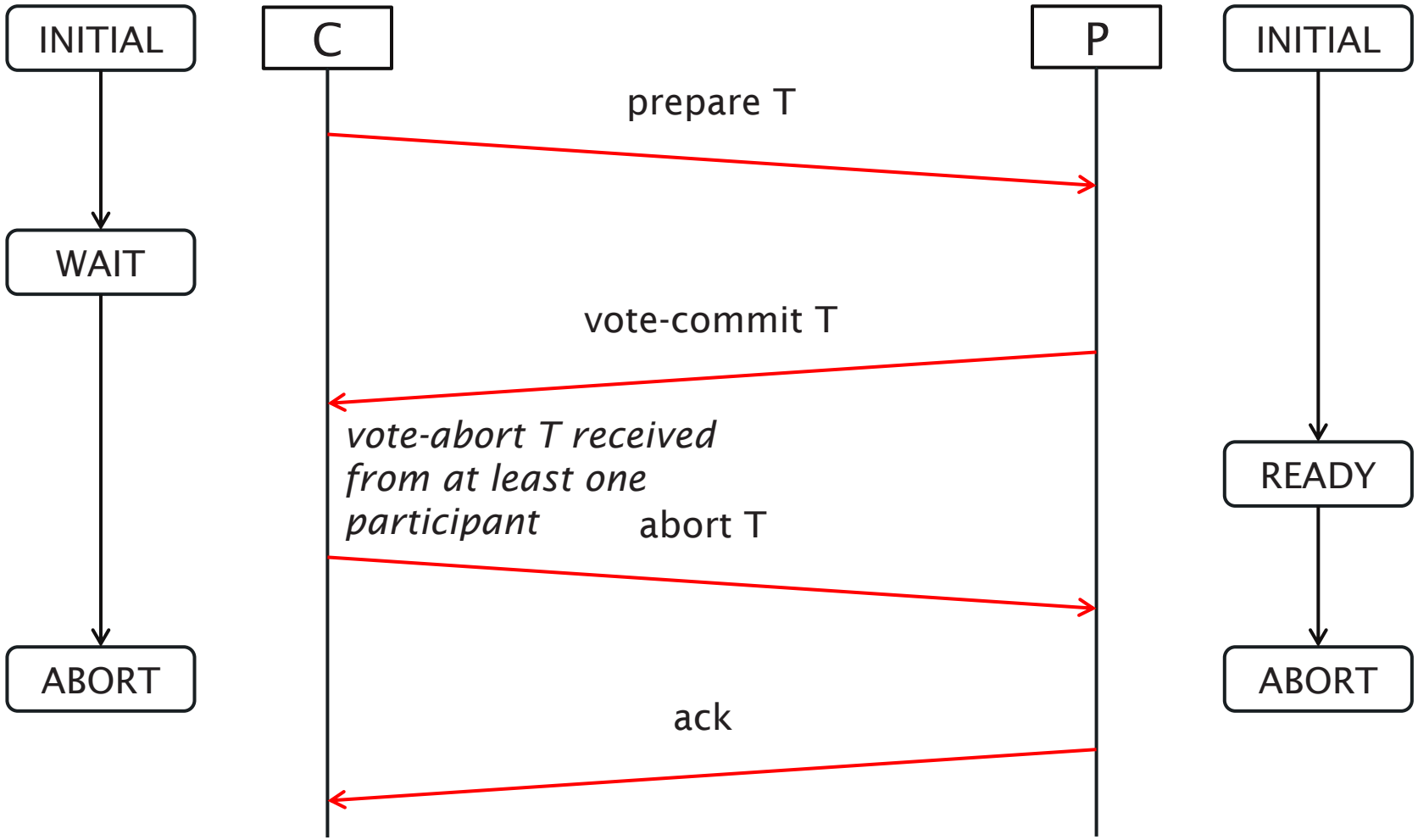
Aborted Transaction



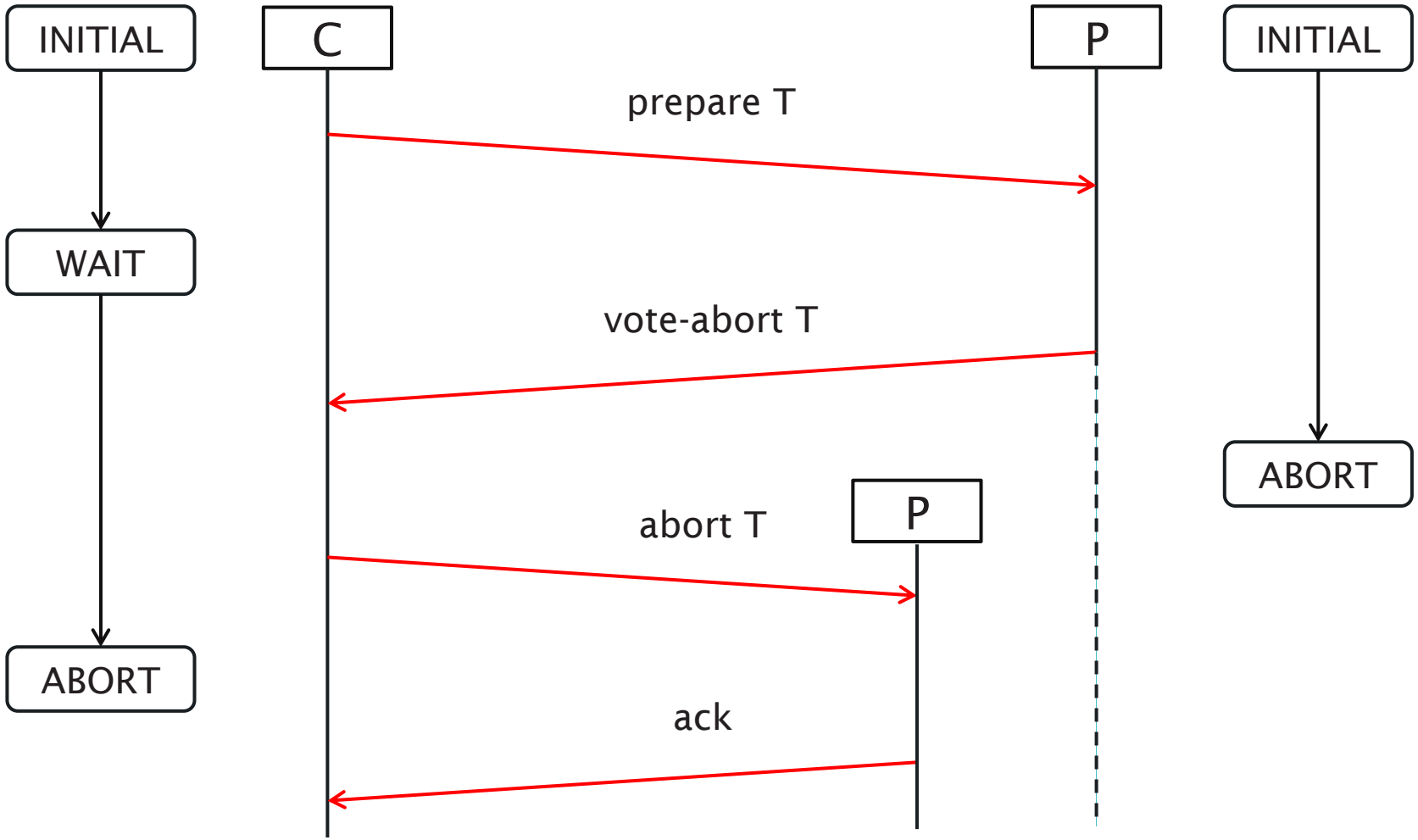
State Transitions



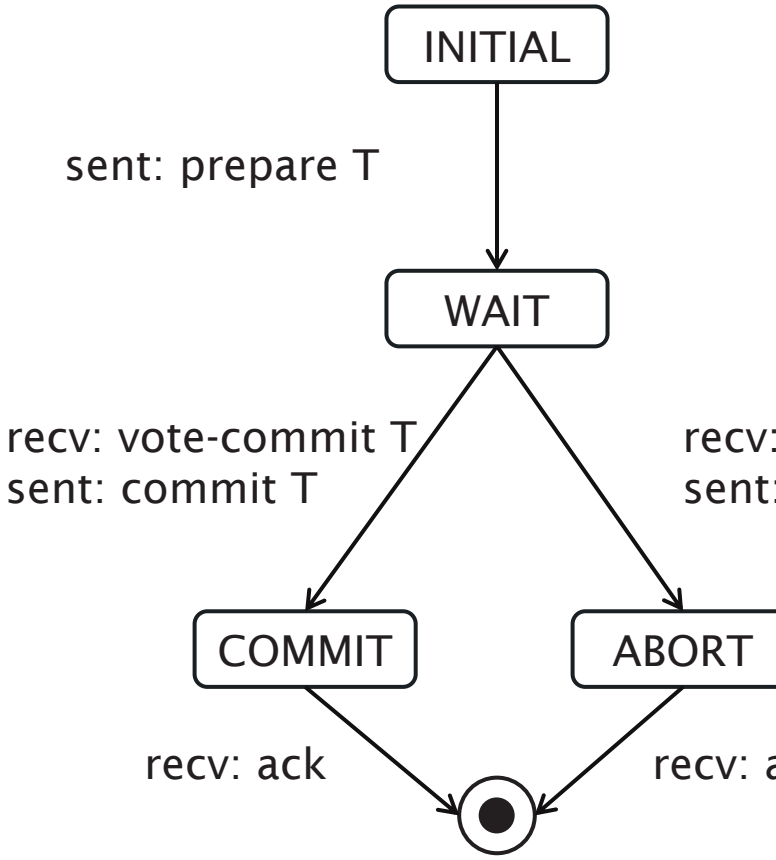
State Transitions



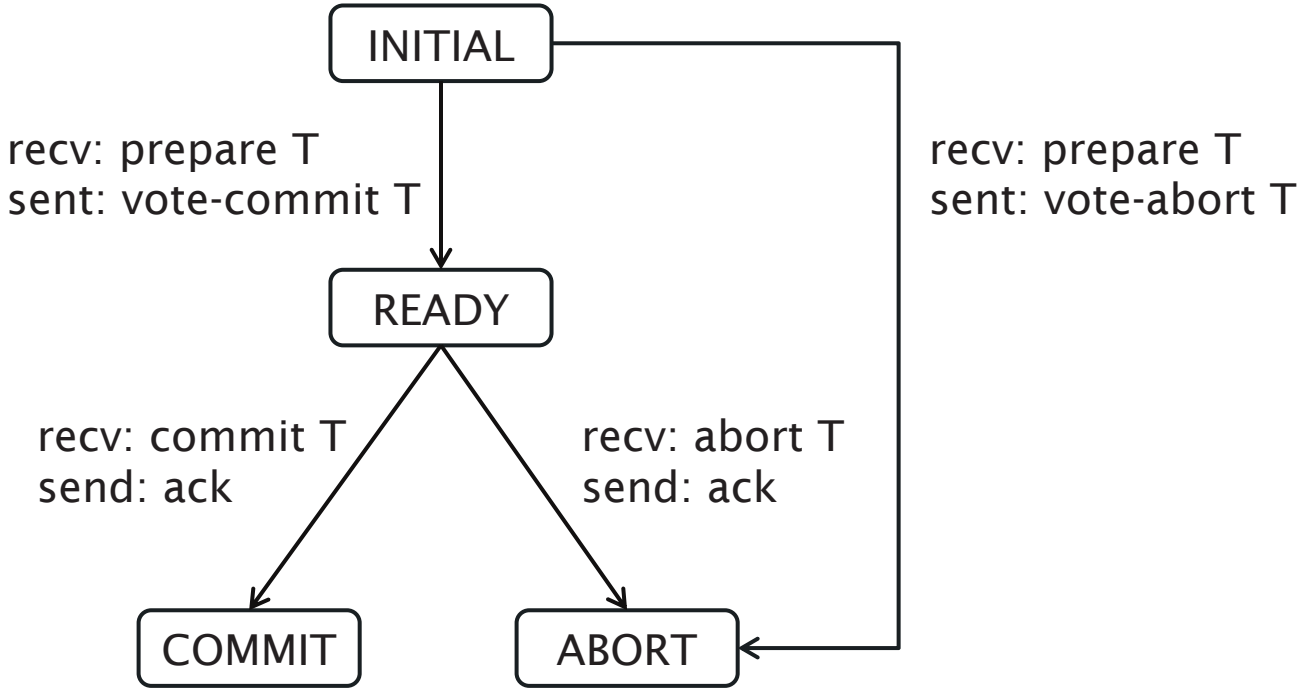
State Transitions



Coordinator State Diagram



Participant State Diagram



Dealing with failures

If the coordinator or a participant fails during the commit, two things happen:

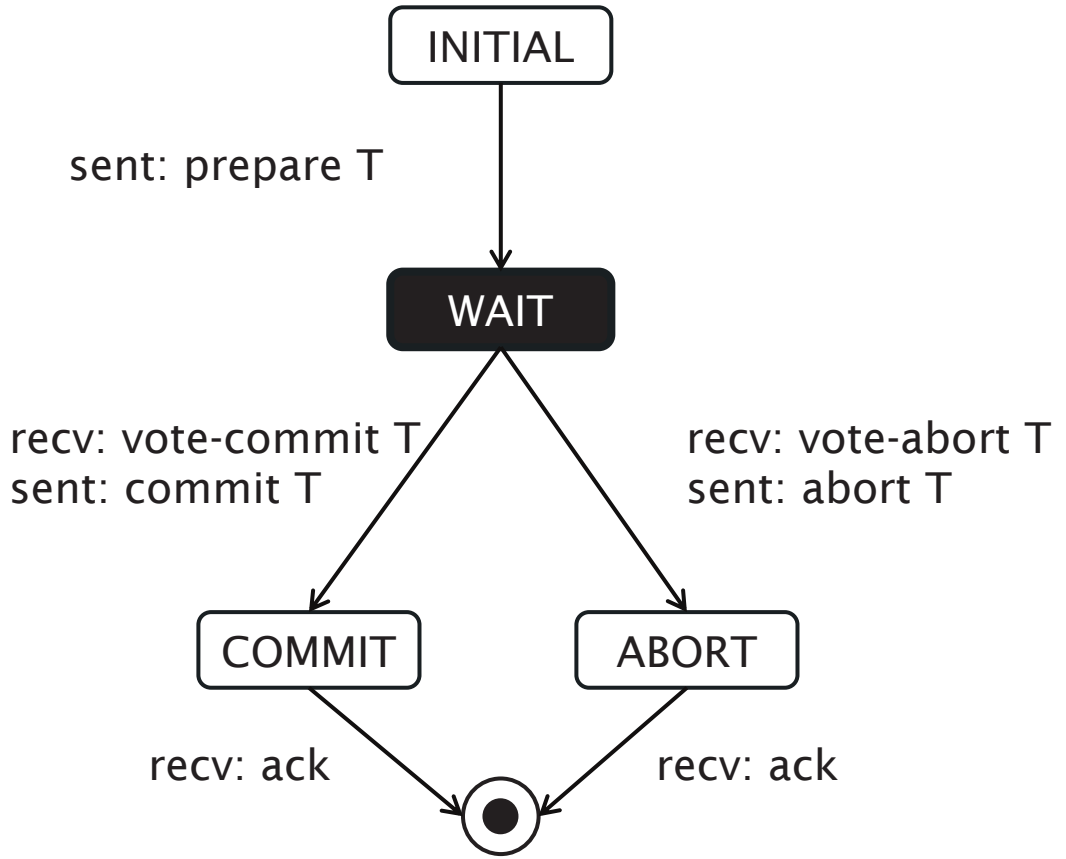
- The other sites will time out while waiting for the next message from the failed site and invoke a *termination protocol*
- When the failed site restarts, it tries to work out the state of the commit by invoking a *recovery protocol*

The behaviour of the sites under these protocols depends on the state they were in when the site failed

Termination Protocol: Coordinator

Timeout in WAIT

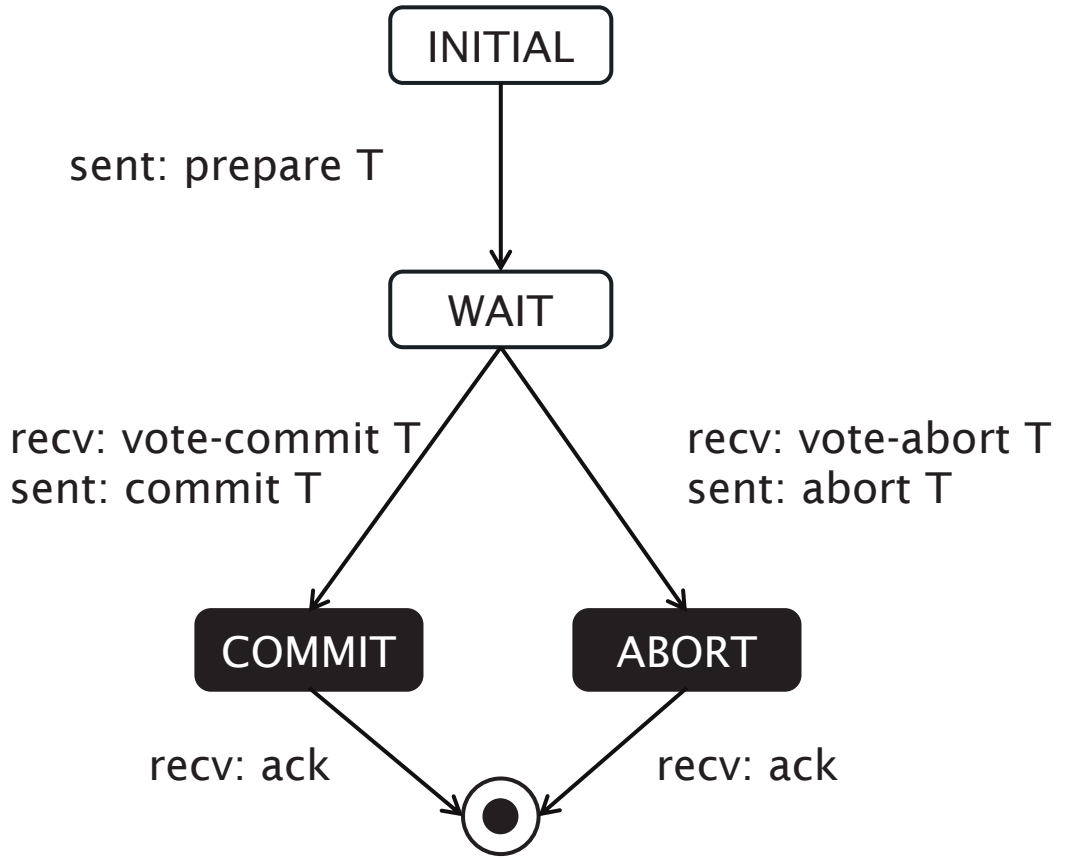
- Coordinator is waiting for participants to vote on whether they're going to commit or abort
- A missing vote means that the coordinator cannot commit the global transaction
- Coordinator may abort the global transaction



Termination Protocol: Coordinator

Timeout in COMMIT/ABORT

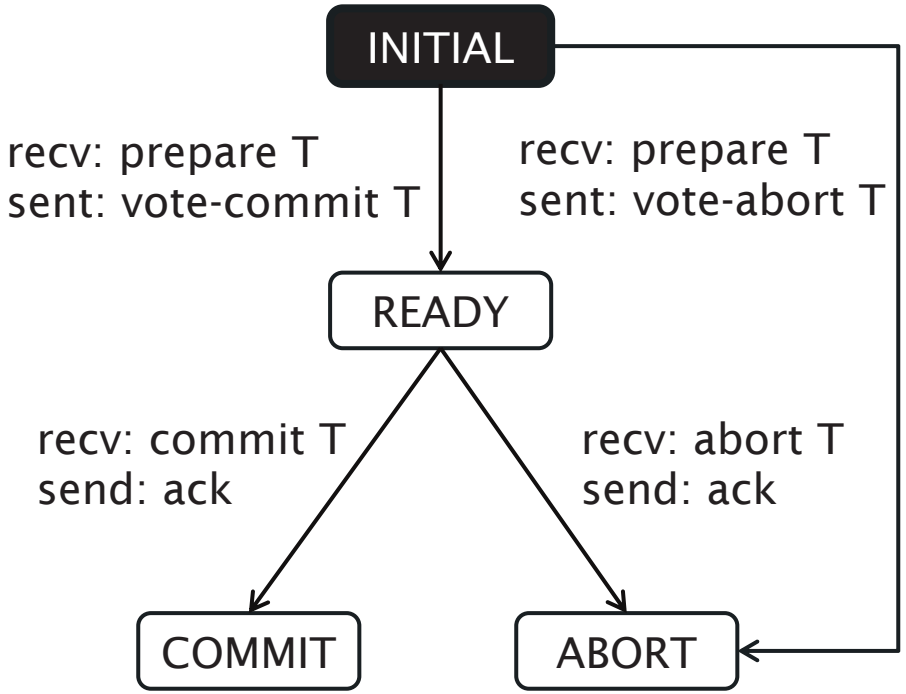
- Coordinator is waiting for participants to acknowledge successful commit or abort
- Coordinator resends global decision to participants who have not acknowledged



Termination Protocol: Participant

Timeout in INITIAL

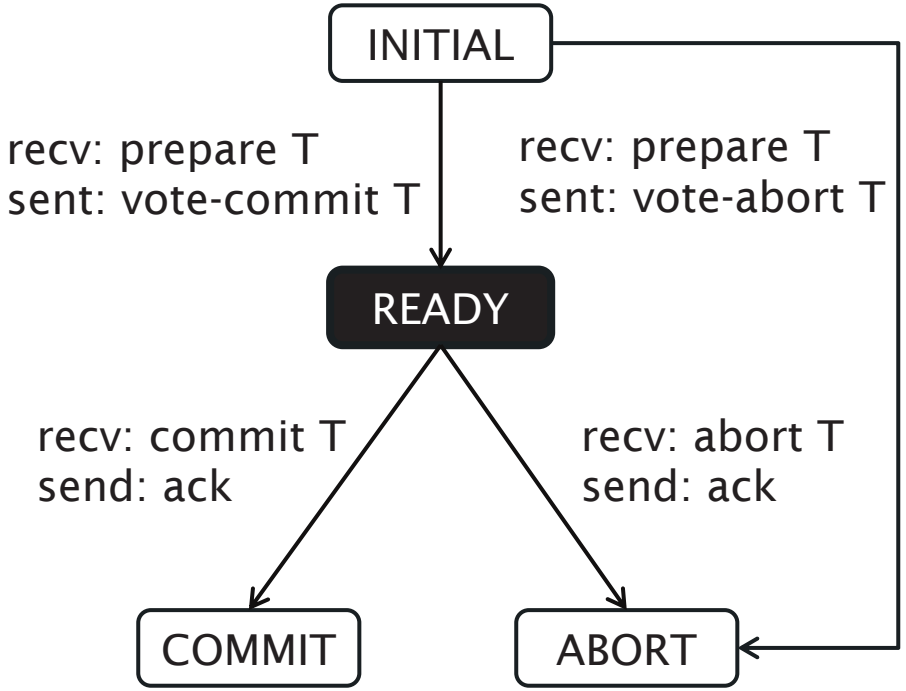
- Participant is waiting for a “prepare T”
- May unilaterally abort the transaction after a timeout
- If “prepare T” arrives after unilateral abort, either:
  - resend the “vote-abort T” message or
  - ignore (coordinator then times out in WAIT)



Termination Protocol: Participant

Timeout in READY

- Participant is waiting for the instruction to commit or abort – blocked without further information
- Alternatively, use cooperative termination protocol – contact other participants to find one who knows the decision



University of Southampton

Cooperative Termination Protocol

Assumes that participants are aware of each other

- Coordinator sends list of participants with "prepare T"

If a participant P times out while waiting for the global decision, it contacts the other participants to see if they know it

Response from the other participant depends on their state and any vote they've sent:

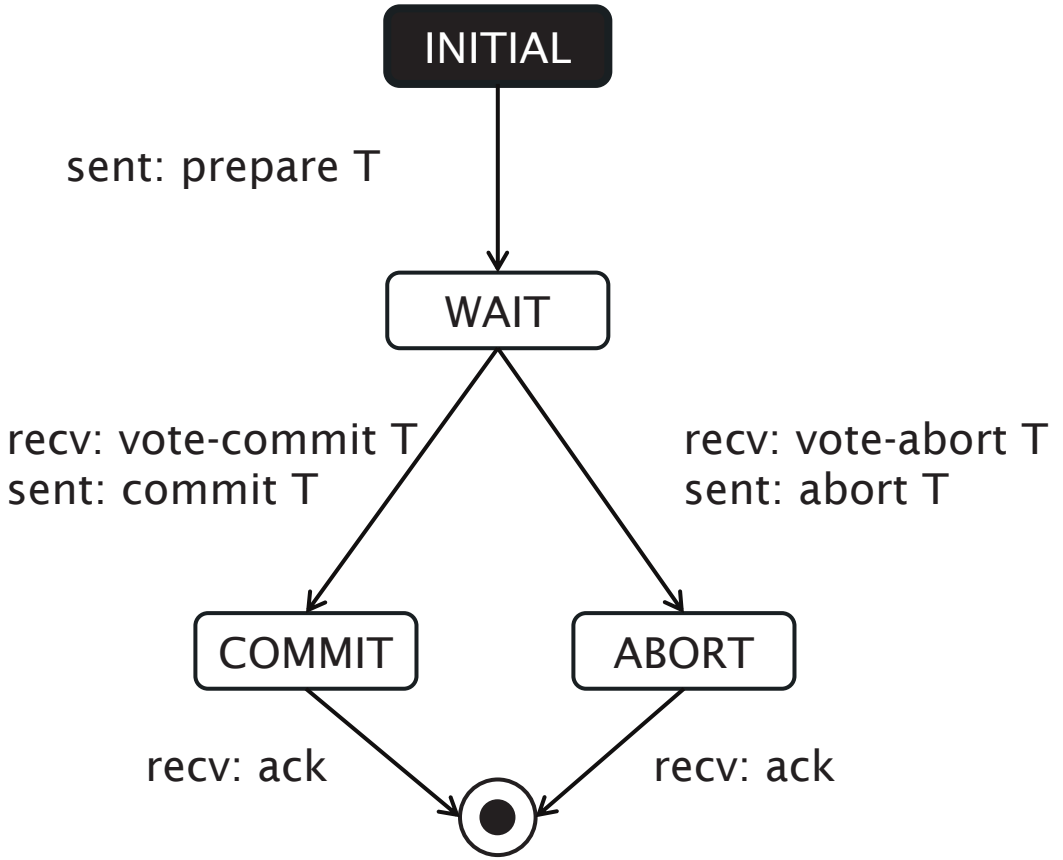
- INITIAL – hasn't yet voted, so unilaterally aborts by sending "abort T"
- READY – voted to abort, so sends "abort T"
- READY – voted to commit, but doesn't know the global decision, so sends "uncertain T"
- ABORT/COMMIT – knows the global decision, so sends "commit T" or "abort T"

If all participants return "uncertain T", then P remains blocked

Recovery Protocol: Coordinator

Failure in INITIAL

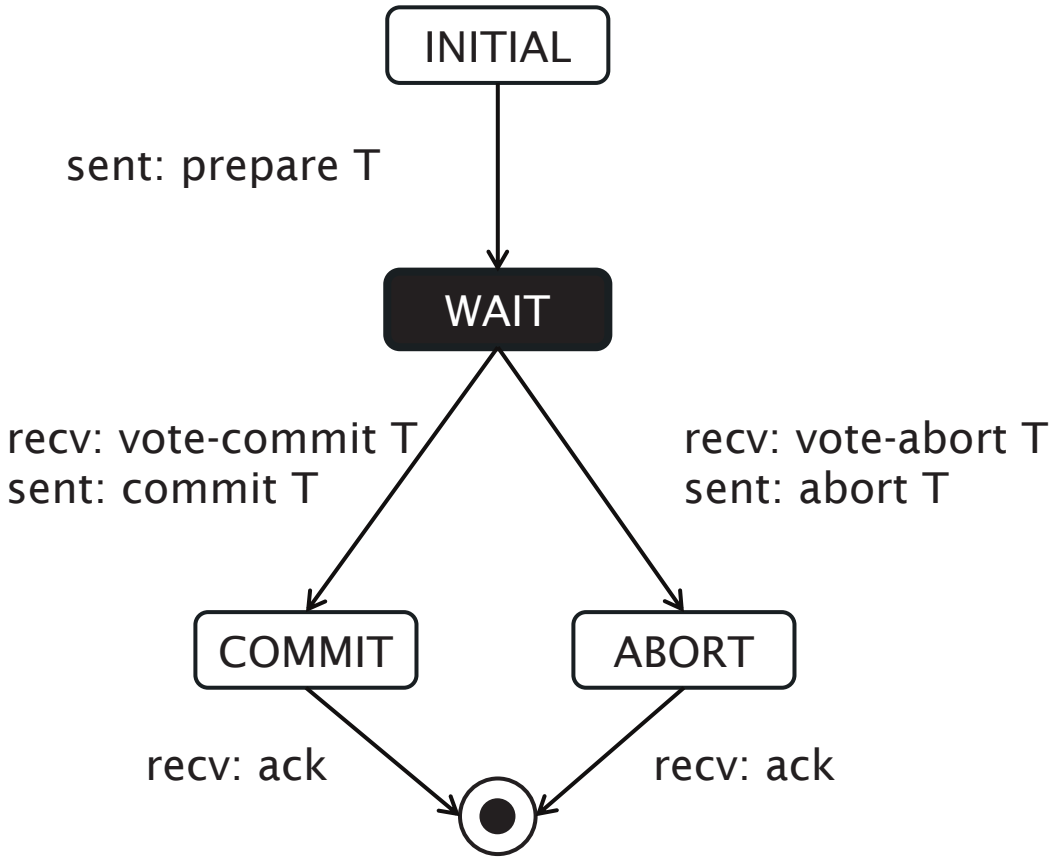
- Commit not yet begun, restart commit procedure



Recovery Protocol: Coordinator

Failure in WAIT

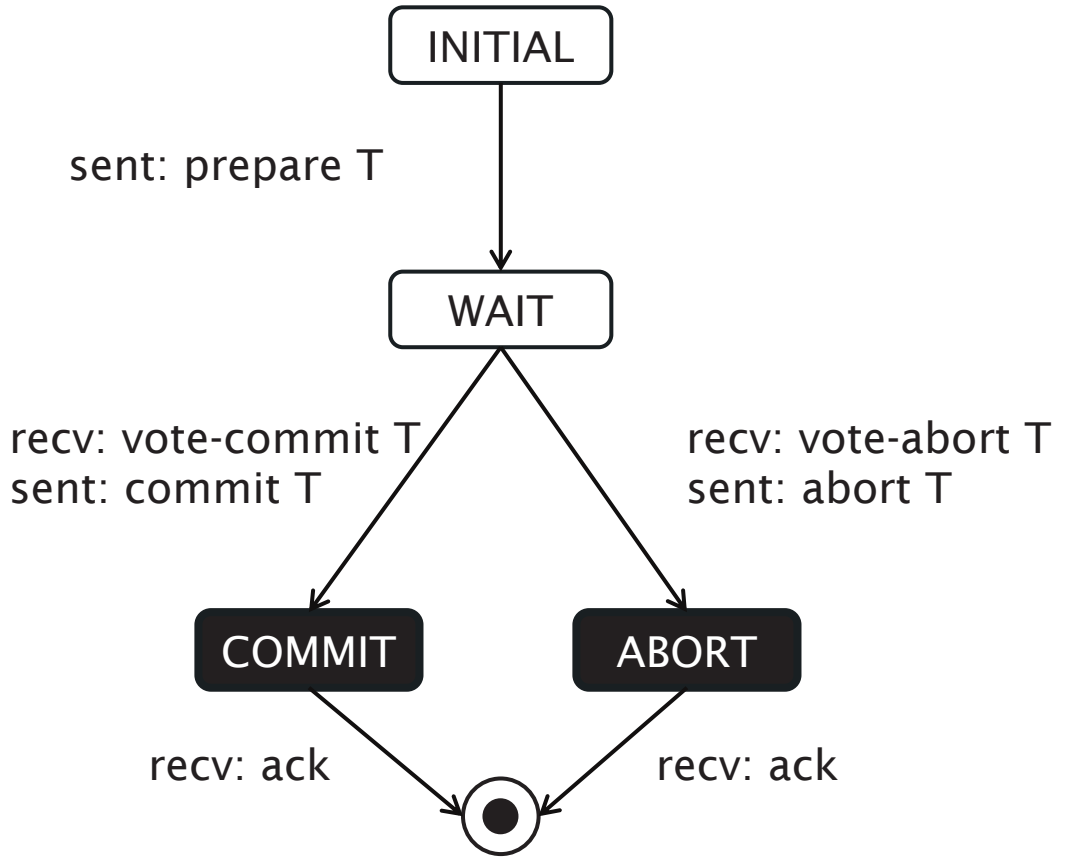
- Coordinator has sent “prepare T”, but has not yet received all vote-commit/vote-abort messages from participants
- Recovery restarts commit procedure by resending “prepare T”



Recovery Protocol: Coordinator

Failure in COMMIT/ABORT

- If coordinator has received all “ack” messages, complete successfully
- Otherwise, invoke terminate protocol (i.e. resend global decision)

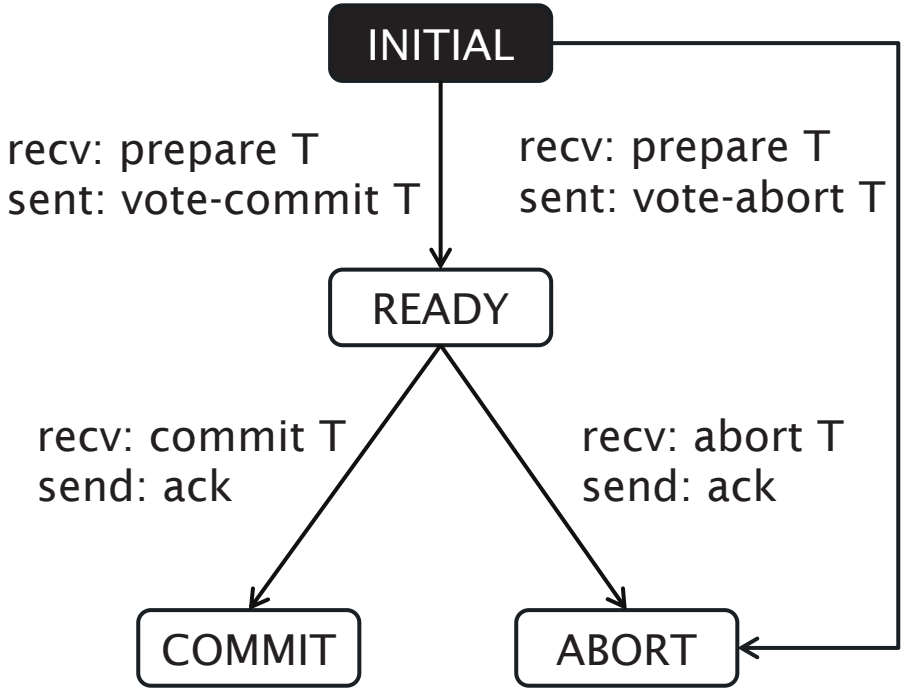


Recovery Protocol: Participant

Failure in INITIAL

- Participant has not yet voted
- Coordinator cannot have reached a decision
- Participant should unilaterally abort by sending “vote-abort T”

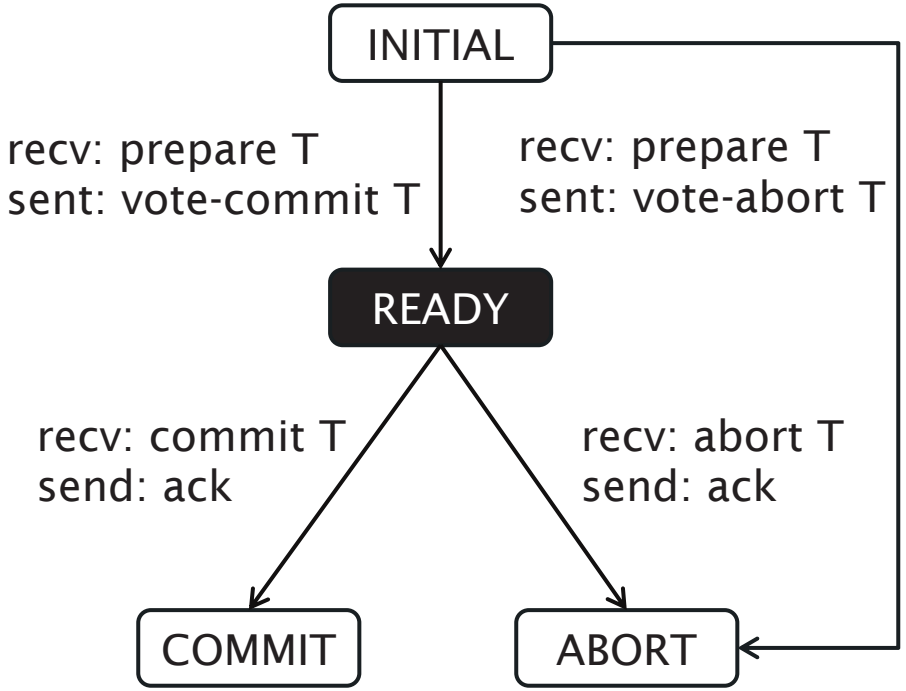
(what was the coordinator doing while the participant was down?)



Recovery Protocol: Participant

Failure in READY

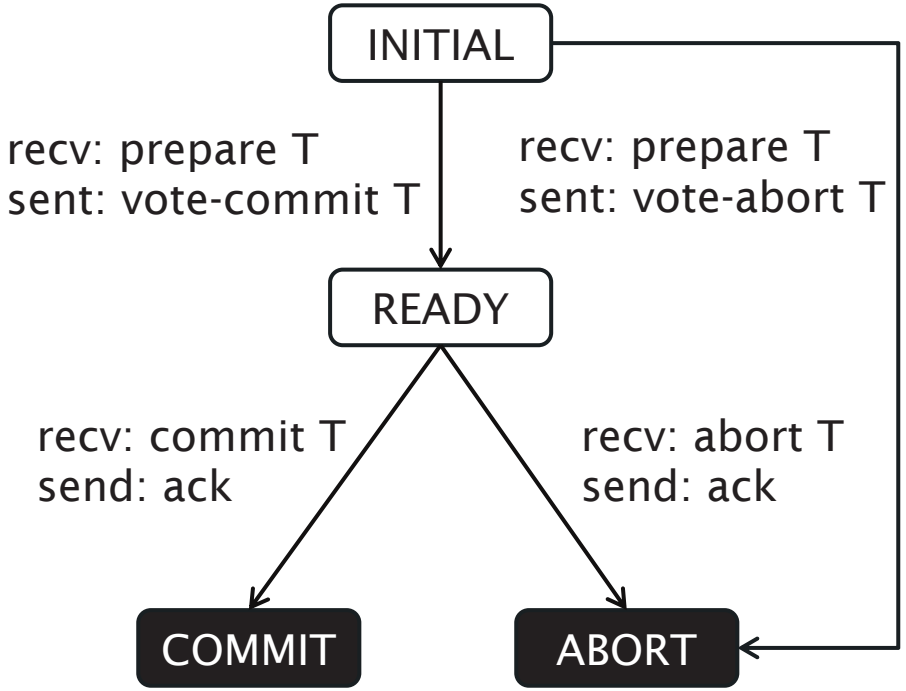
- Participant has voted, but doesn't know what the global decision was
- Treat as a timeout in READY
  (use cooperative termination protocol)



Recovery Protocol: Participant

Failure in COMMIT/ABORT

- “ack” message has been sent
- Participant need take no action



University of Southampton

2PC Variants

University of Southampton

2PC Performance

Costs associated with 2PC:

- Number of messages transmitted between coordinator and participants
- Number of times that logs are accessed

We can improve the performance of 2PC if we can reduce either of these

- Coordinator keeps state information about current transactions in memory (doesn't need to consult logs)

Two proposed approaches:

- Presumed-Abort
- Presumed-Commit

University of Southampton

Presumed-Abort

Improves performance by letting the coordinator forget about transactions (remove them from memory) in certain circumstances

If the global decision was to abort the transaction, write <abort T> to log and forget T

- If a participant asks the coordinator about the global decision and it isn't in memory, tell the participant that the transaction was aborted
- Coordinator doesn't need to write <end T>

If the global decision was to commit the transaction, only forget it and write <commit T> and <end T> to log once all "ack" messages have been received from participants

Presumed-Commit

Assumes that, if no information about a transaction is in memory, it must have been committed

If the global decision is to commit, coordinator writes <commit T> to log, sends "commit T" and forgets the transaction

If the global decision is to abort, coordinator writes <abort T> to log and sends "abort T"

Only writes <end T> and forgets T when all "ack" messages have been received

University of Southampton

Three-Phase Commit

University of Southampton

Three-Phase Commit

As we saw earlier, 2PC can still block in certain circumstances

- Participant times out in READY and is unable to find out the global decision

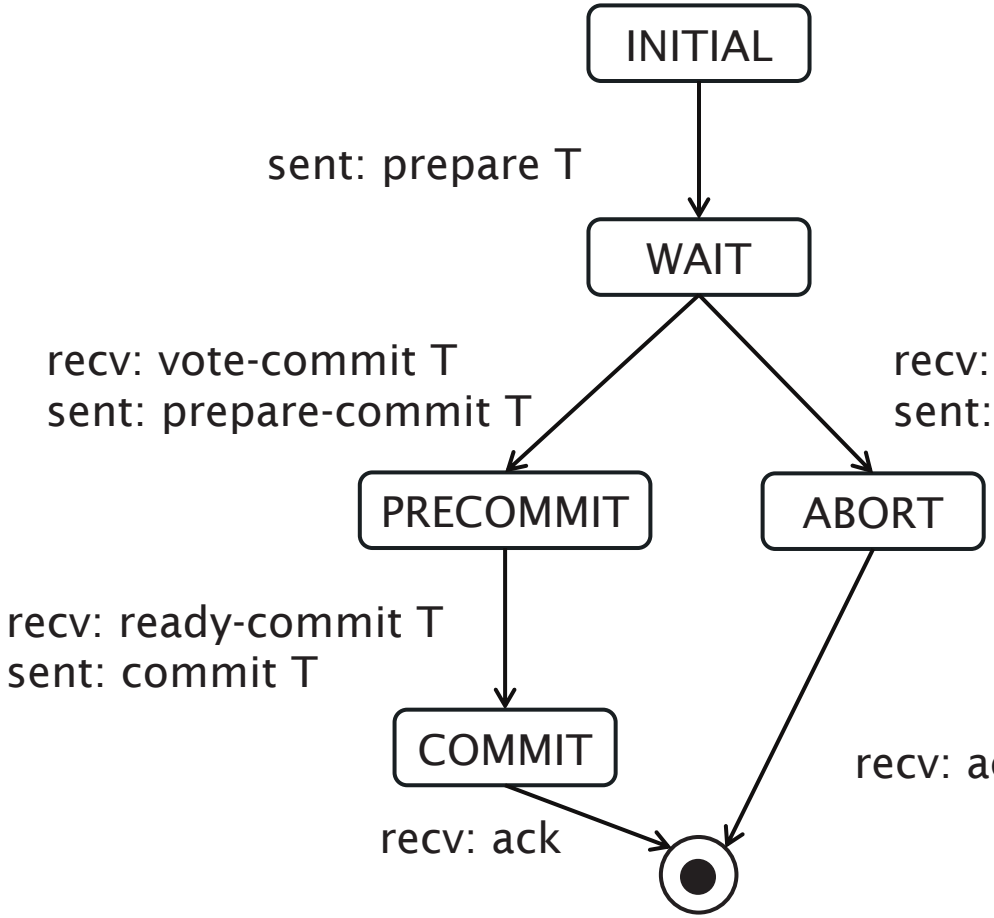
3PC is non-blocking in the event of site failure (but not network partition)

Adds an additional state between WAIT/READY and COMMIT

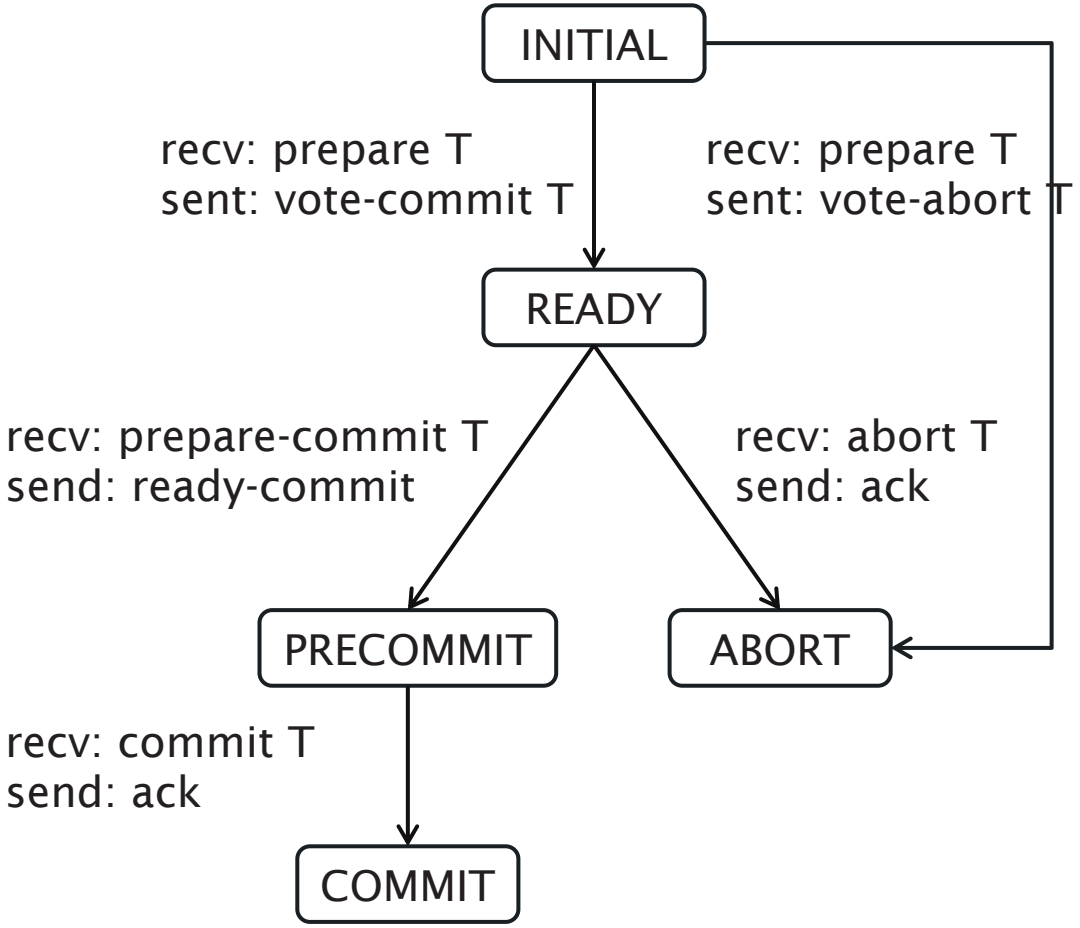
- PRECOMMIT – process is ready to commit but has not yet committed

Some changes to termination and recovery protocols from 2PC

Coordinator State Diagram



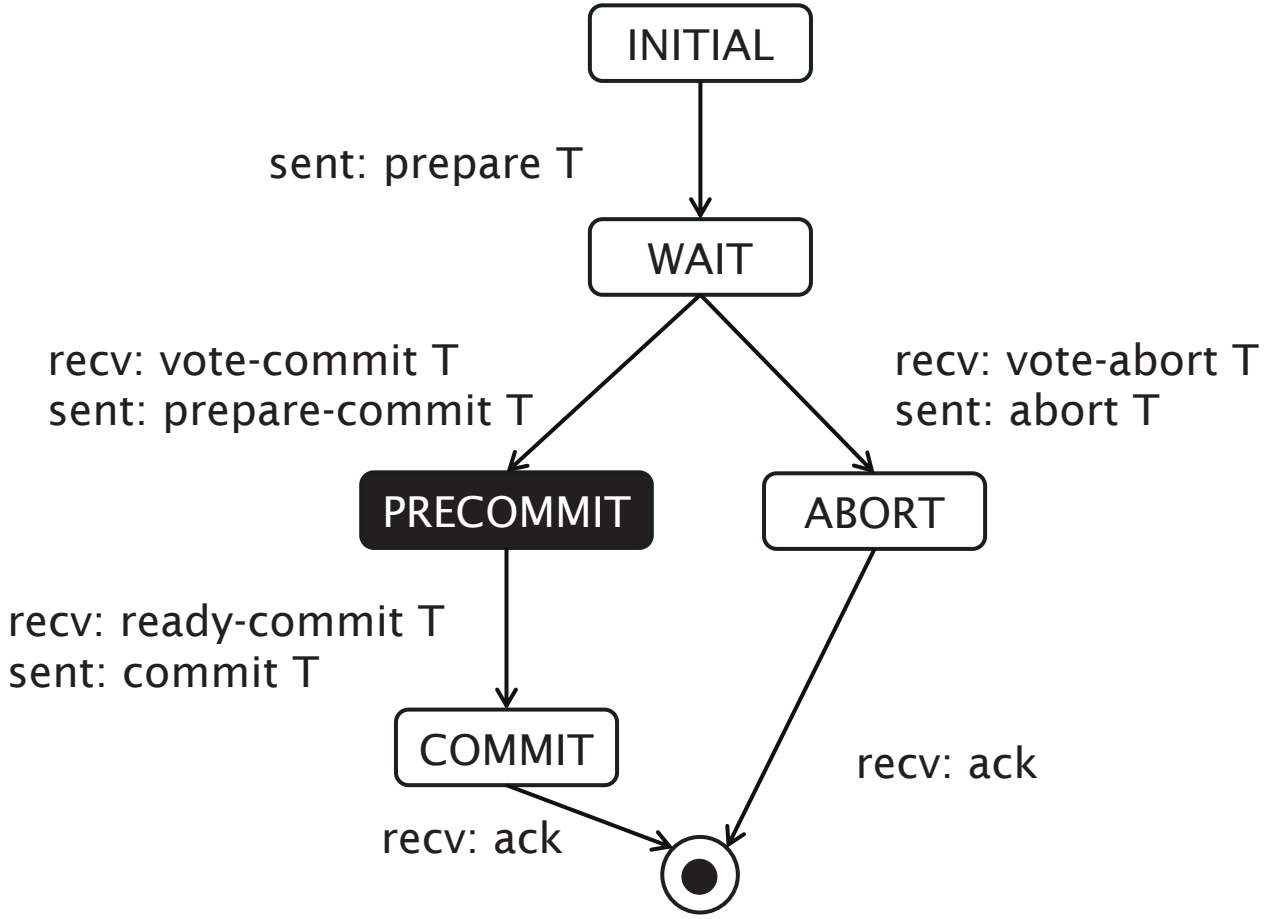
Participant State Diagram



3PC Termination Protocol: Coordinator

Timeout in PRECOMMIT

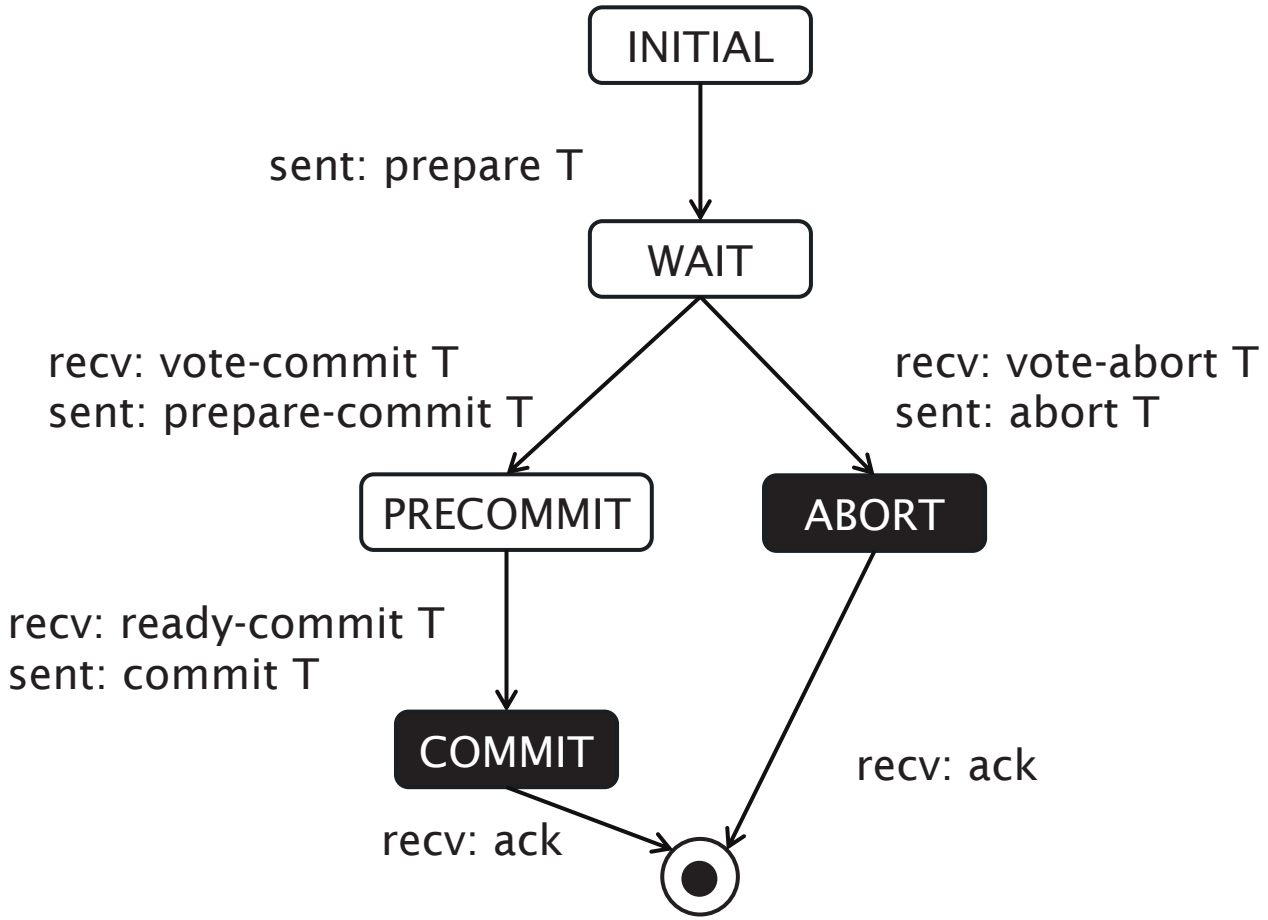
- Coordinator does not if non-responding participants have moved to PRECOMMIT, but it does know that they're all in READY at least (so have all voted to commit)
- Move all participants to PRECOMMIT by sending "prepare-commit T", then send "commit T"



3PC Termination Protocol: Coordinator

Timeout in COMMIT/ABORT

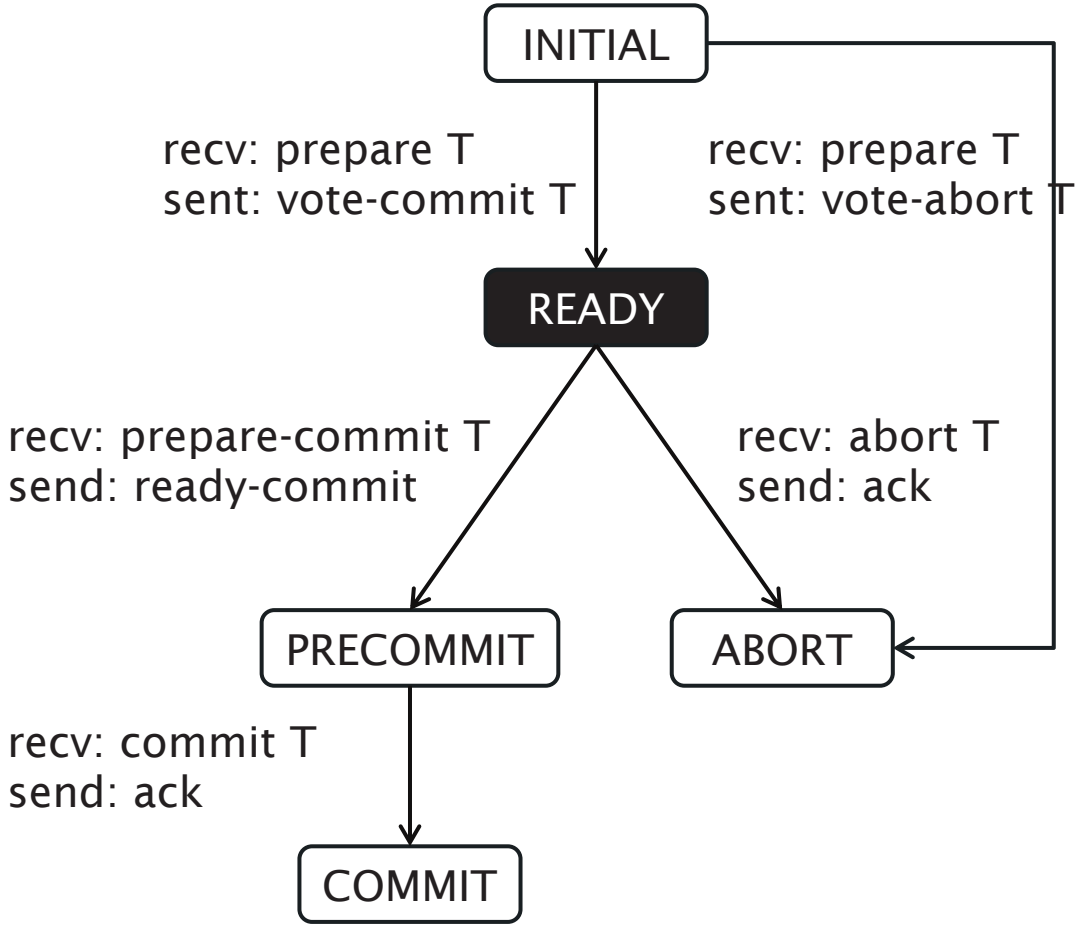
- Coordinator does not know if participants have performed the commit or abort, but knows that they are in either PRECOMMIT or READY
- Participants follow their own recovery protocols



3PC Termination Protocol: Participant

Timeout in READY

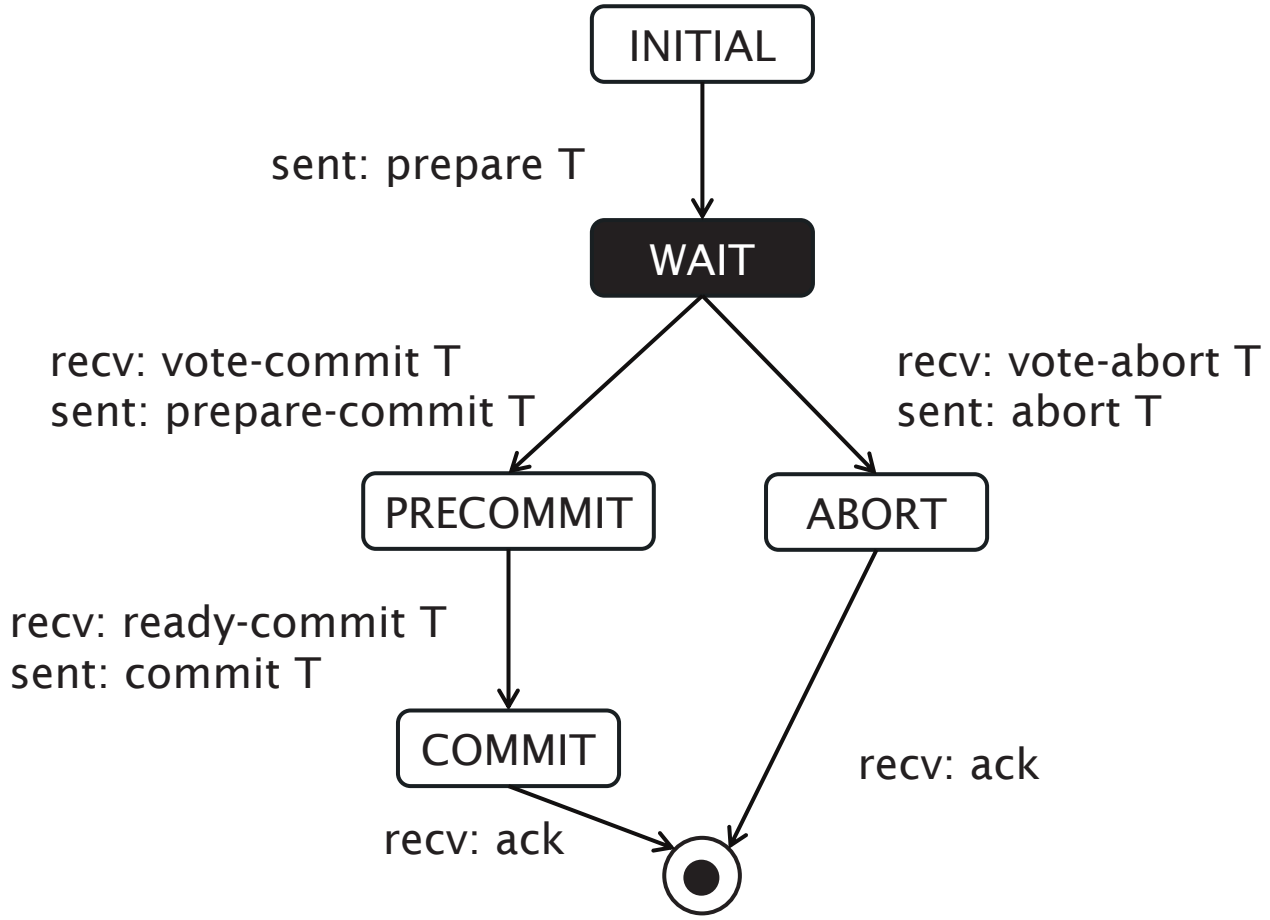
- Participant has voted to commit, but does not know the global decision
- Elects a new coordinator, and proceeds according to its state:
  - WAIT – new coordinator globally aborts
  - PRECOMMIT – new coordinator globally commits
  - ABORT – all participants will also move into ABORT



3PC Recovery Protocol: Coordinator

Failure in WAIT

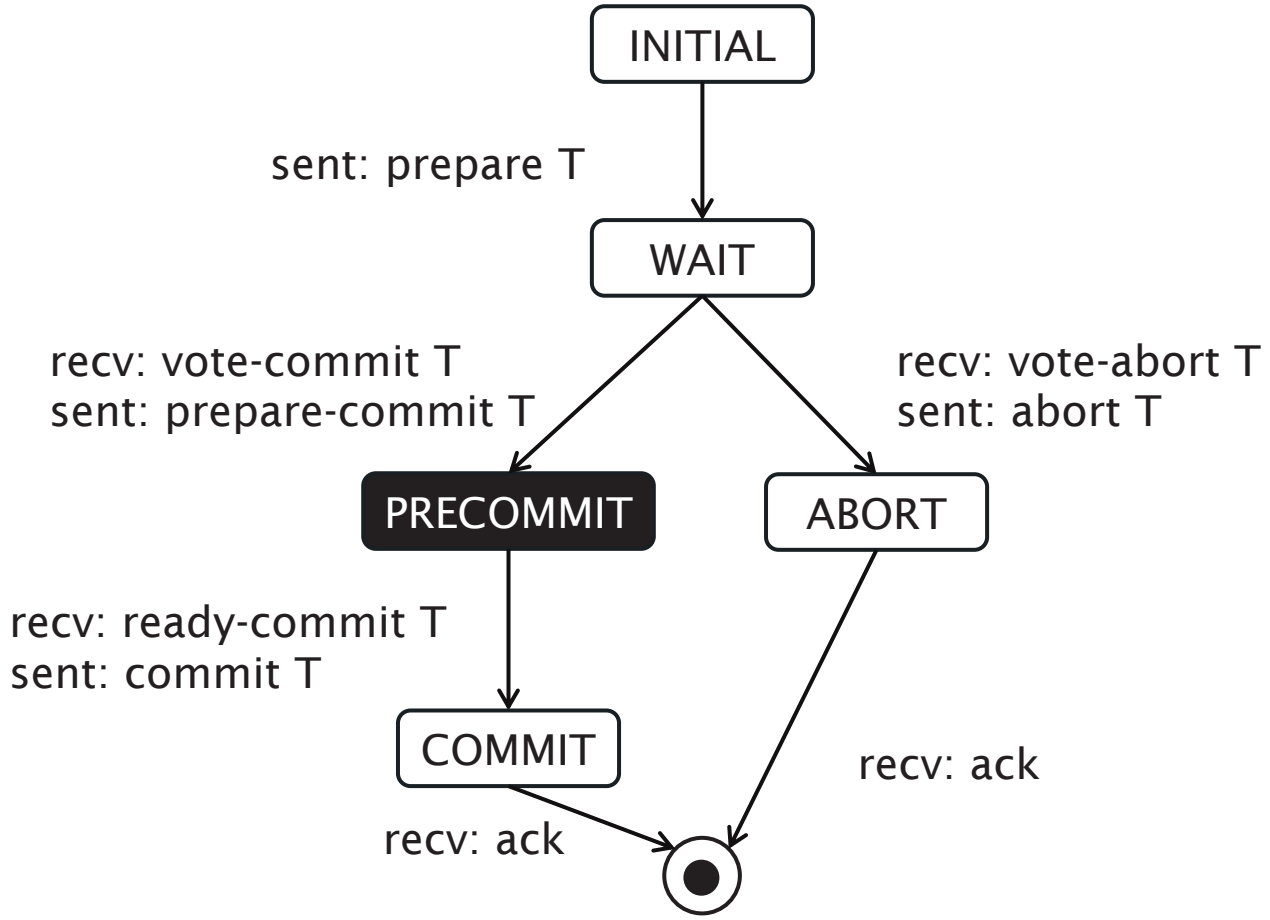
- Participants will have already terminated the transaction due to termination protocol
- Coordinator needs to ask participants for outcome



3PC Recovery Protocol: Coordinator

Failure in PRECOMMIT

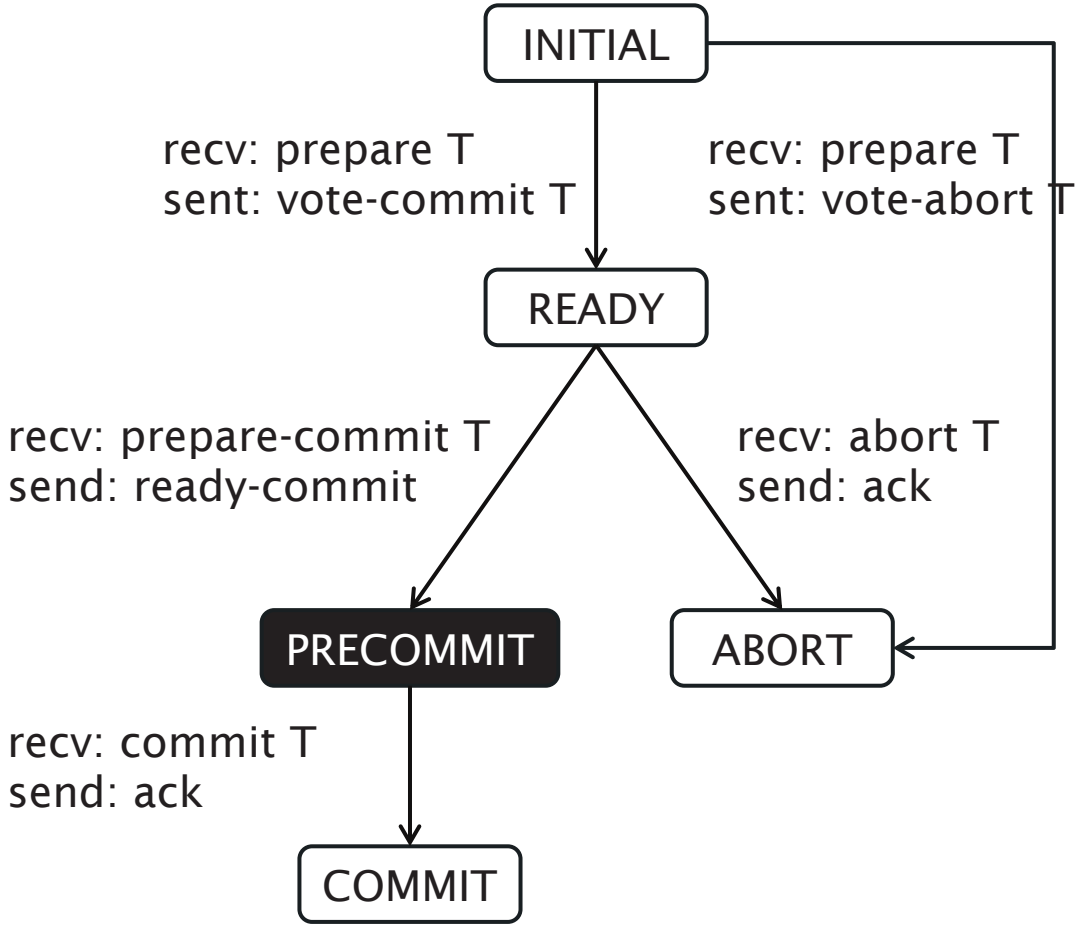
- Participants will have already terminated the transaction due to termination protocol
- Coordinator needs to ask participants for outcome



3PC Recover Protocol: Participant

Failure in PRECOMMIT

- Participant must ask to determine how other participants have terminated the transaction



University of Southampton

Parallel Utilities

Parallel Utilities

Ancillary operations can also exploit the parallel hardware

- Parallel Data Loading/Import/Export
- Parallel Index Creation
- Parallel Rebalancing
- Parallel Backup
- Parallel Recovery

University of Southampton

Next Lecture: Distributed Databases