

Parallel Databases

COMP3211 Advanced Databases

Dr Nicholas Gibbins - nmg@ecs.soton.ac.uk
2014-2015

Overview

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

The I/O Bottleneck

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)

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

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

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

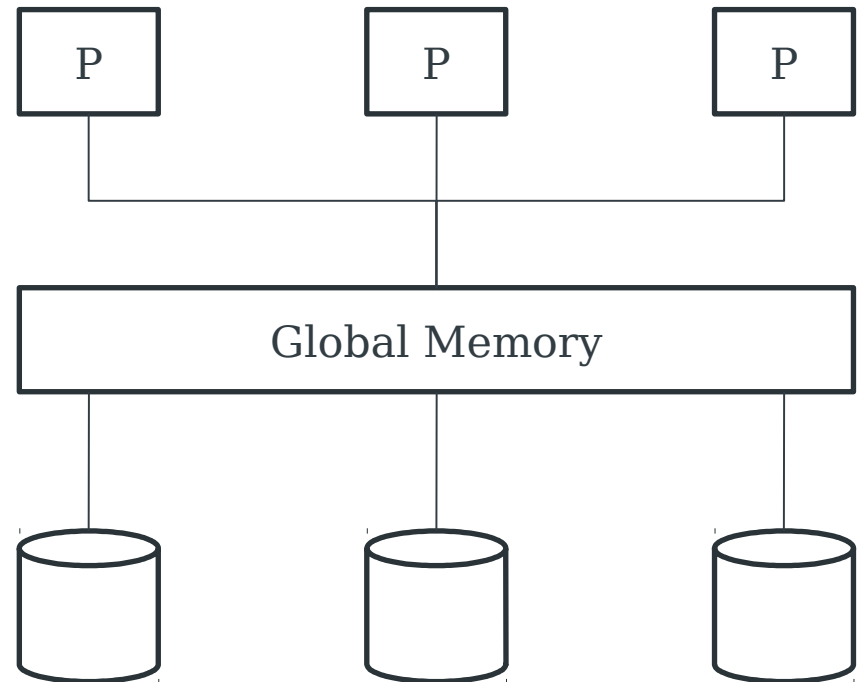
Parallel Architectures

Shared Memory Architecture

- Tightly coupled
- Symmetric Multiprocessor (SMP)

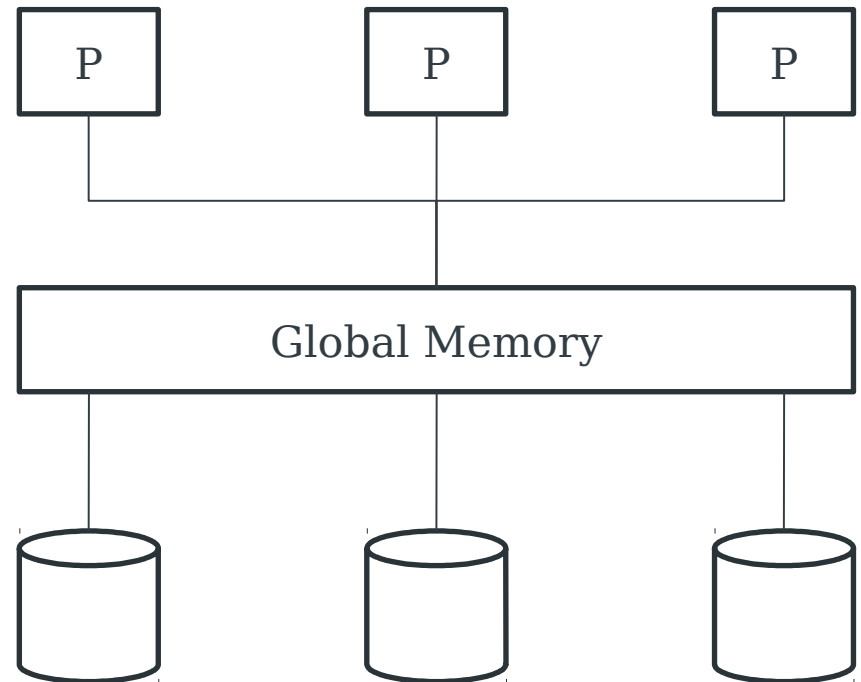
P = processor

M = memory



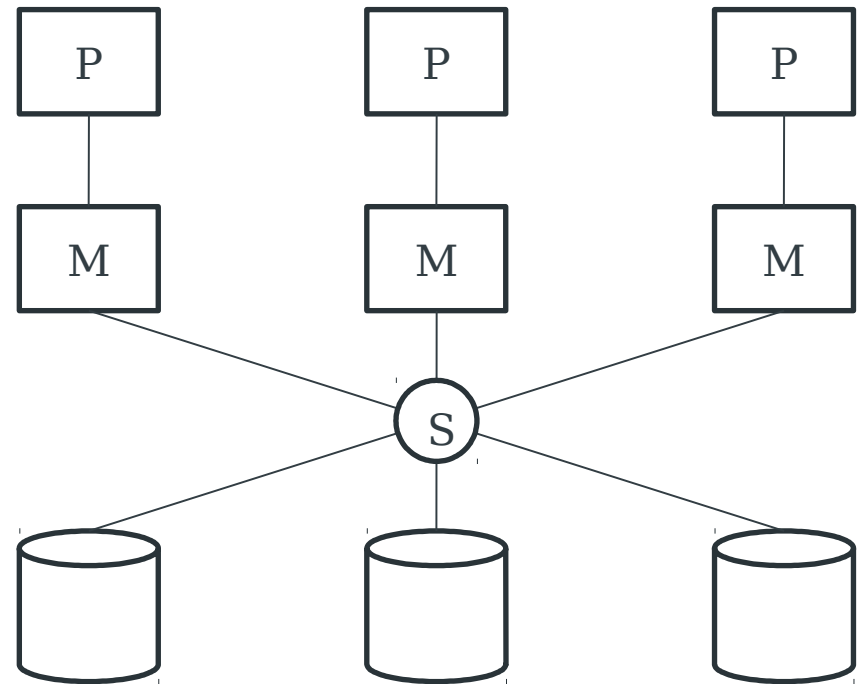
Software – Shared Memory

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



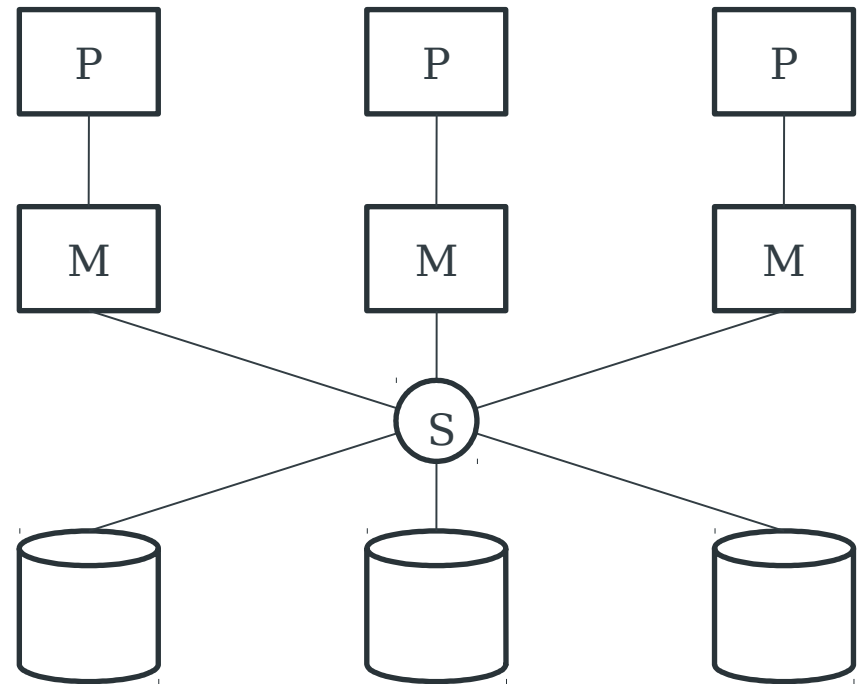
Shared Disc Architecture

- Loosely coupled
- Distributed Memory



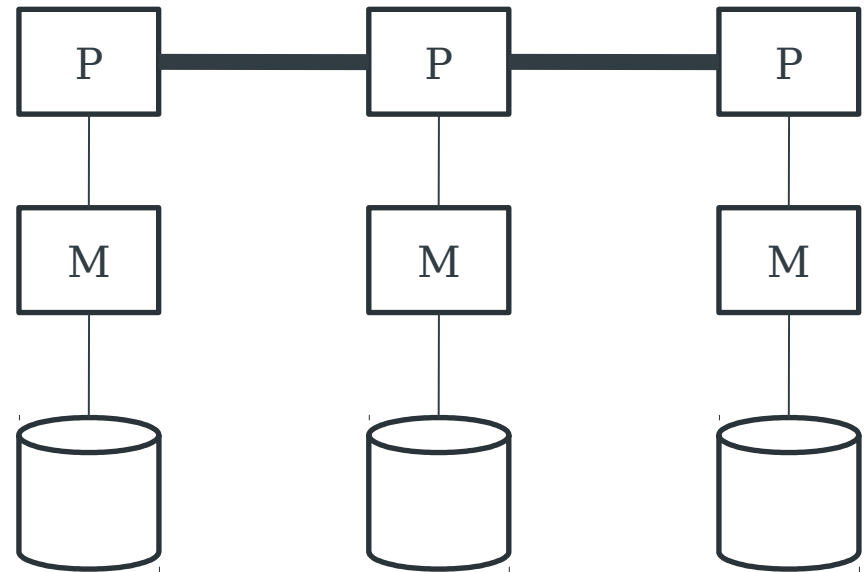
Software – Shared Disc

- 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



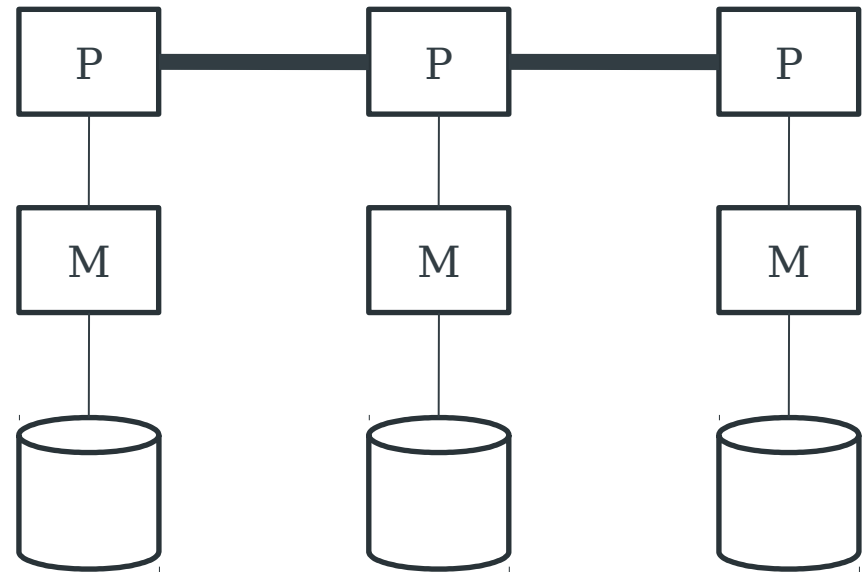
Shared Nothing Architecture

- 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



Hardware vs. Software Architecture

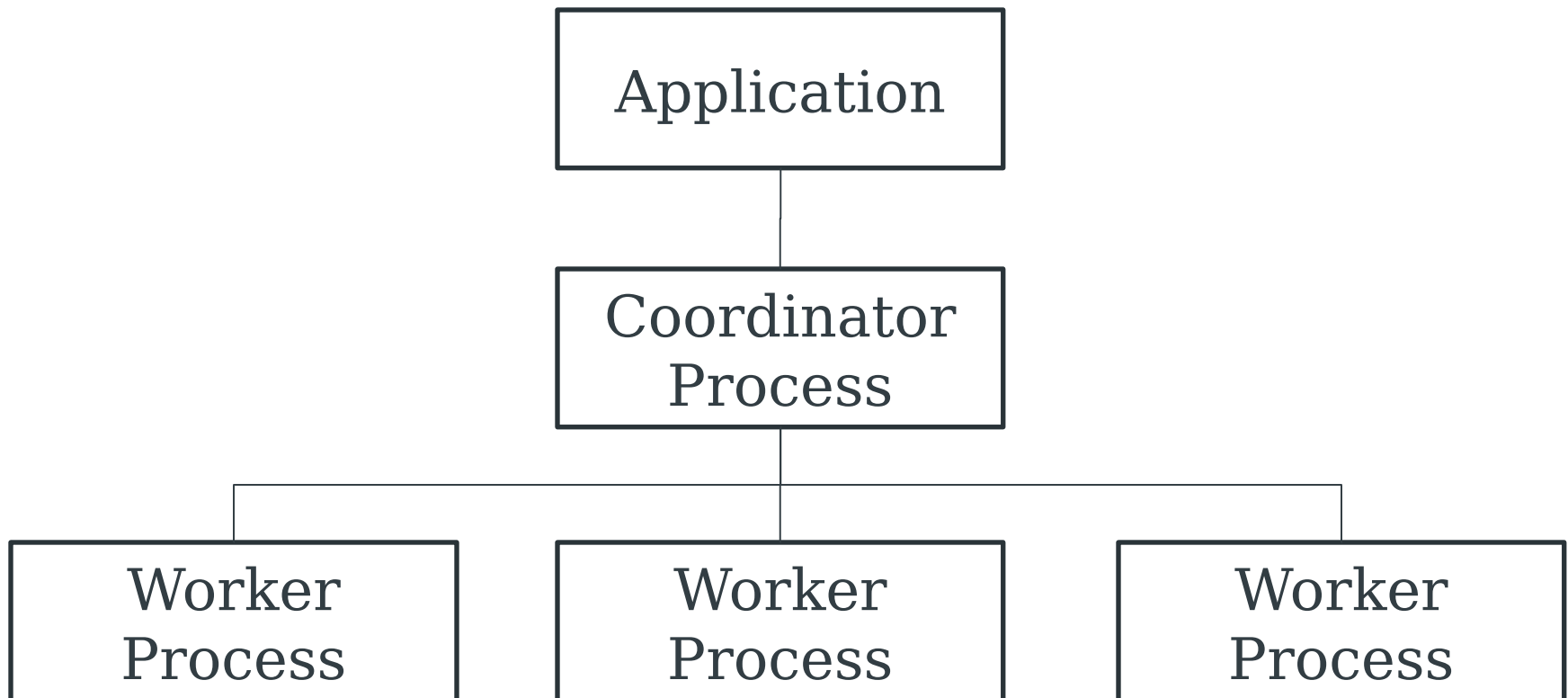
- It is possible to use one software strategy on a different hardware arrangement
- Also possible to simulate one hardware configuration on another
 - Virtual Shared Disk (VSD) makes an IBM SP shared nothing system look like a shared disc setup (for Oracle)
- From this point on, we deal only with shared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

Shared Nothing Challenges

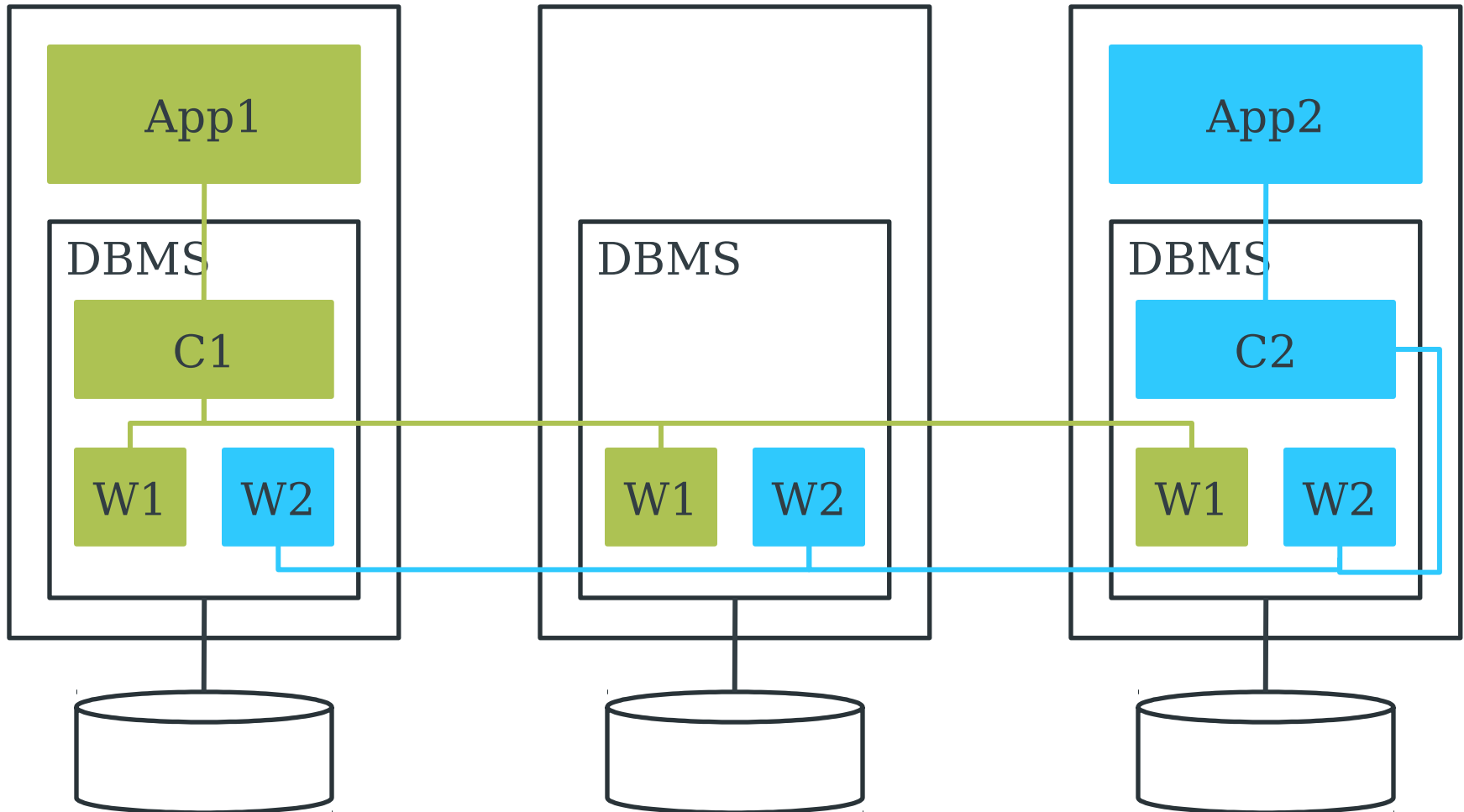
- Partitioning the data
- Keeping the partitioned data balanced
- Splitting up queries to get the work done
- Avoiding distributed deadlock
- Concurrency control
- Dealing with node failure

Parallel Query Processing

Dividing up the Work



Database Software on each node



Inter-Query Parallelism

Improves throughput

Different queries/transactions execute on different processors

- (largely equivalent to material in lectures on concurrency)

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

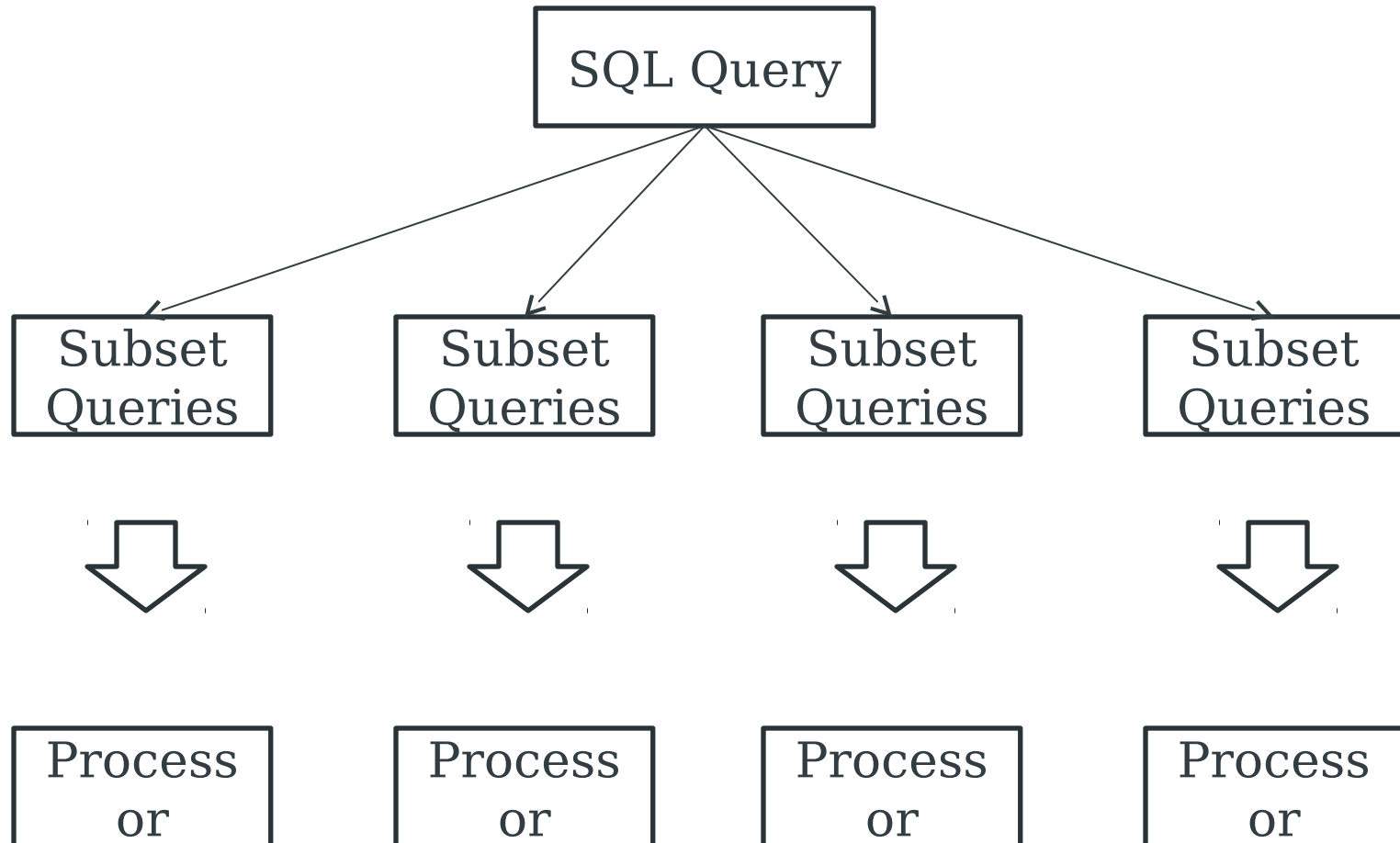
- Operations are overlapped
- Pipeline data from one stage to the next without materialisation

Bushy (independent) parallelism

- Subtrees in query plan executed concurrently

Intra-Operator Parallelism

Intra-Operator Parallelism



Partitioning

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

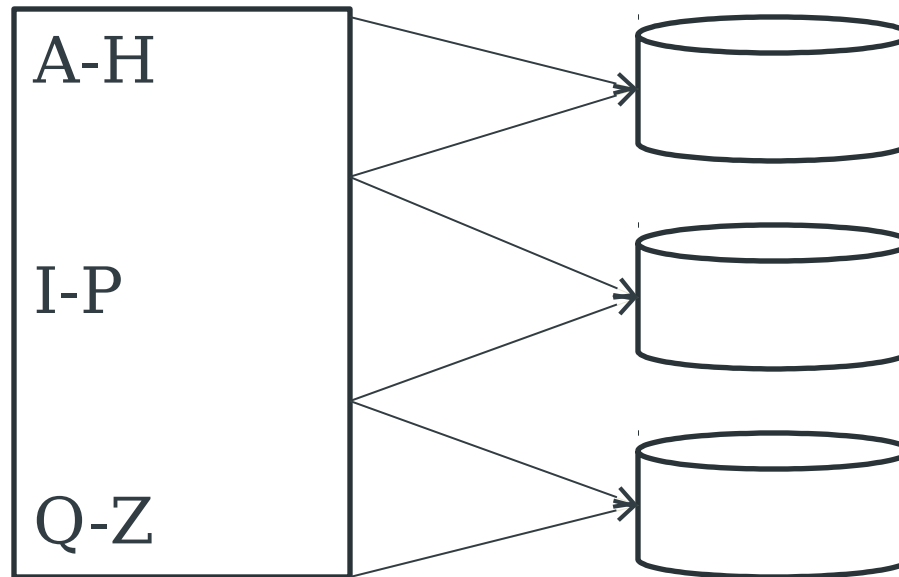
- Access data in parallel to mitigate the I/O bottleneck

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

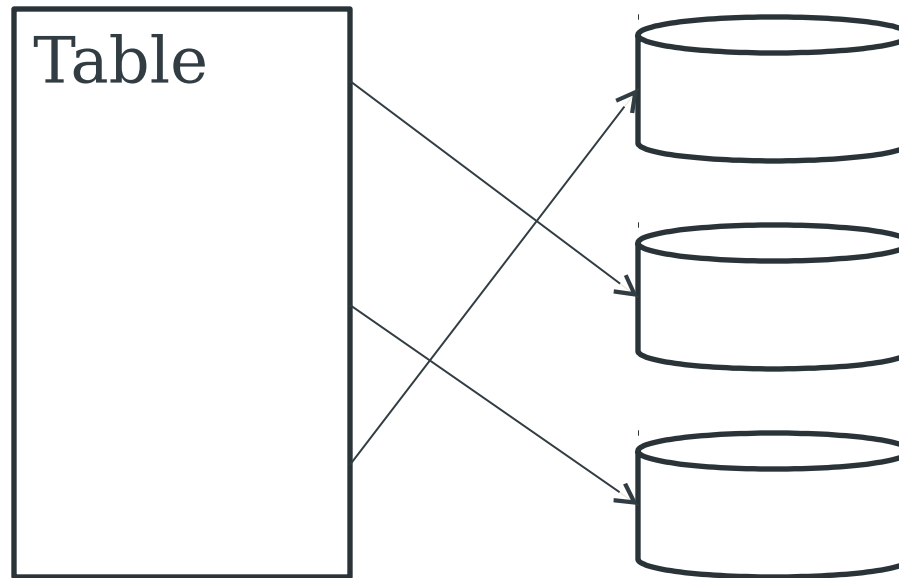
Choice of partitions affords different parallel query processing approaches:

- Range partitioning
- Hash partitioning
- Schema partitioning

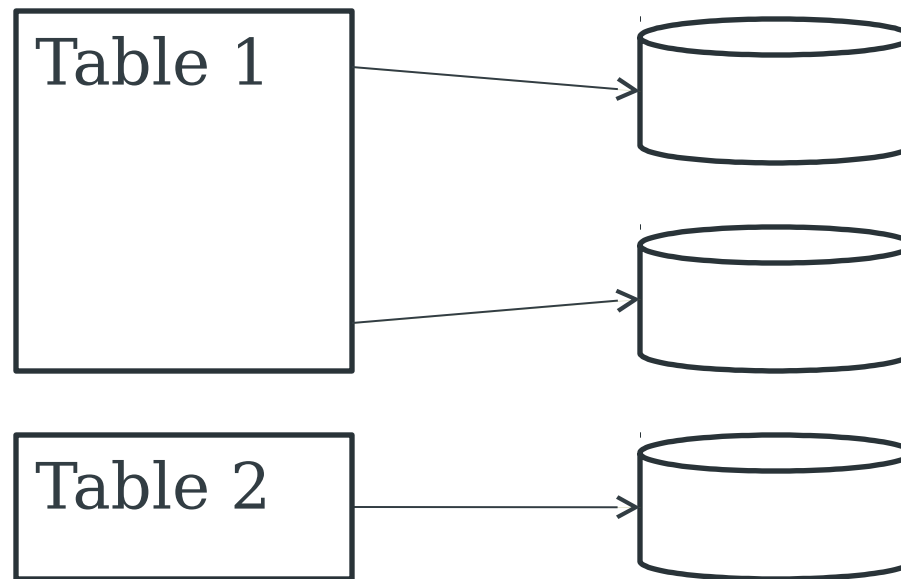
Range Partitioning



Hash Partitioning

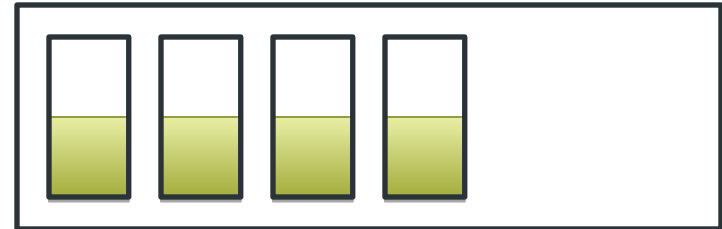


Schema Partitioning

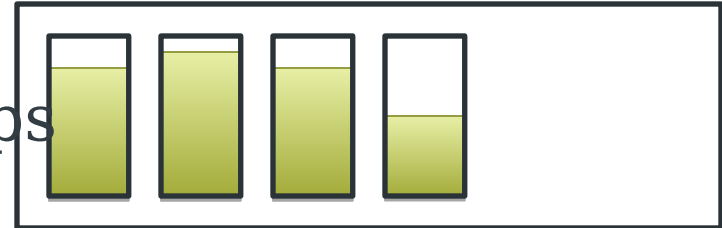


Rebalancing Data

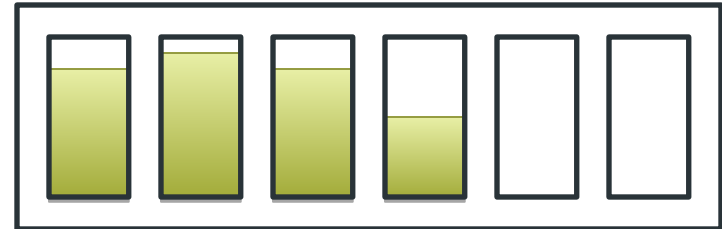
Data in proper balance



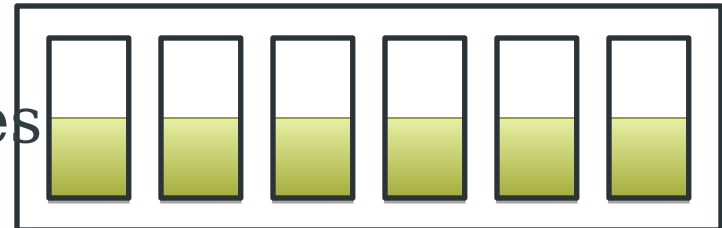
Data grows, performance drops



Add new nodes and disc



Redistribute data to new nodes



Intra-Operator Parallelism

Example query:

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

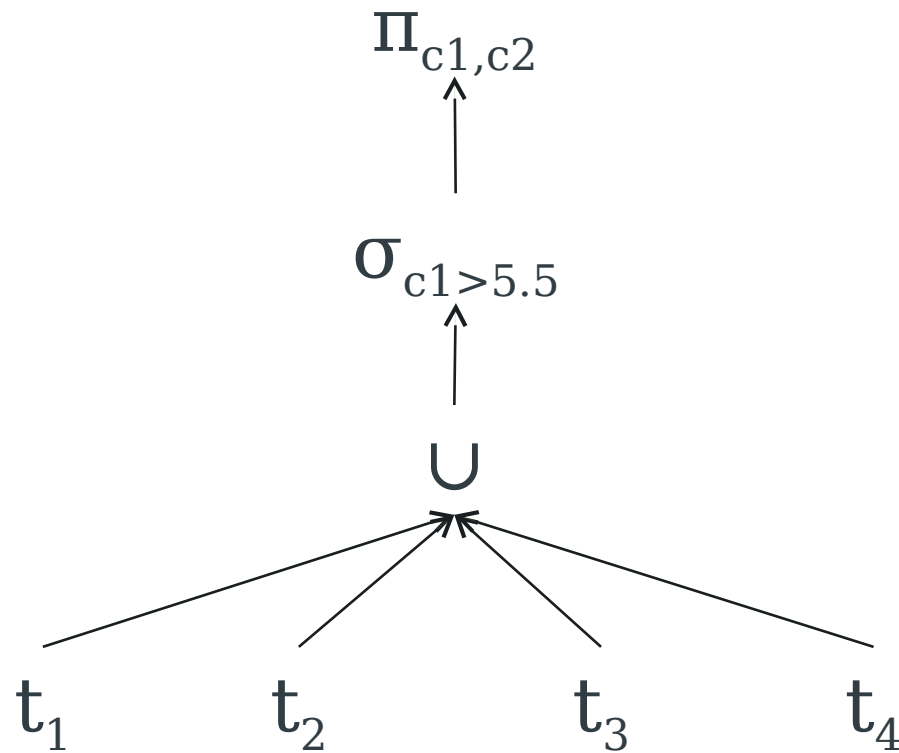
Assumptions:

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

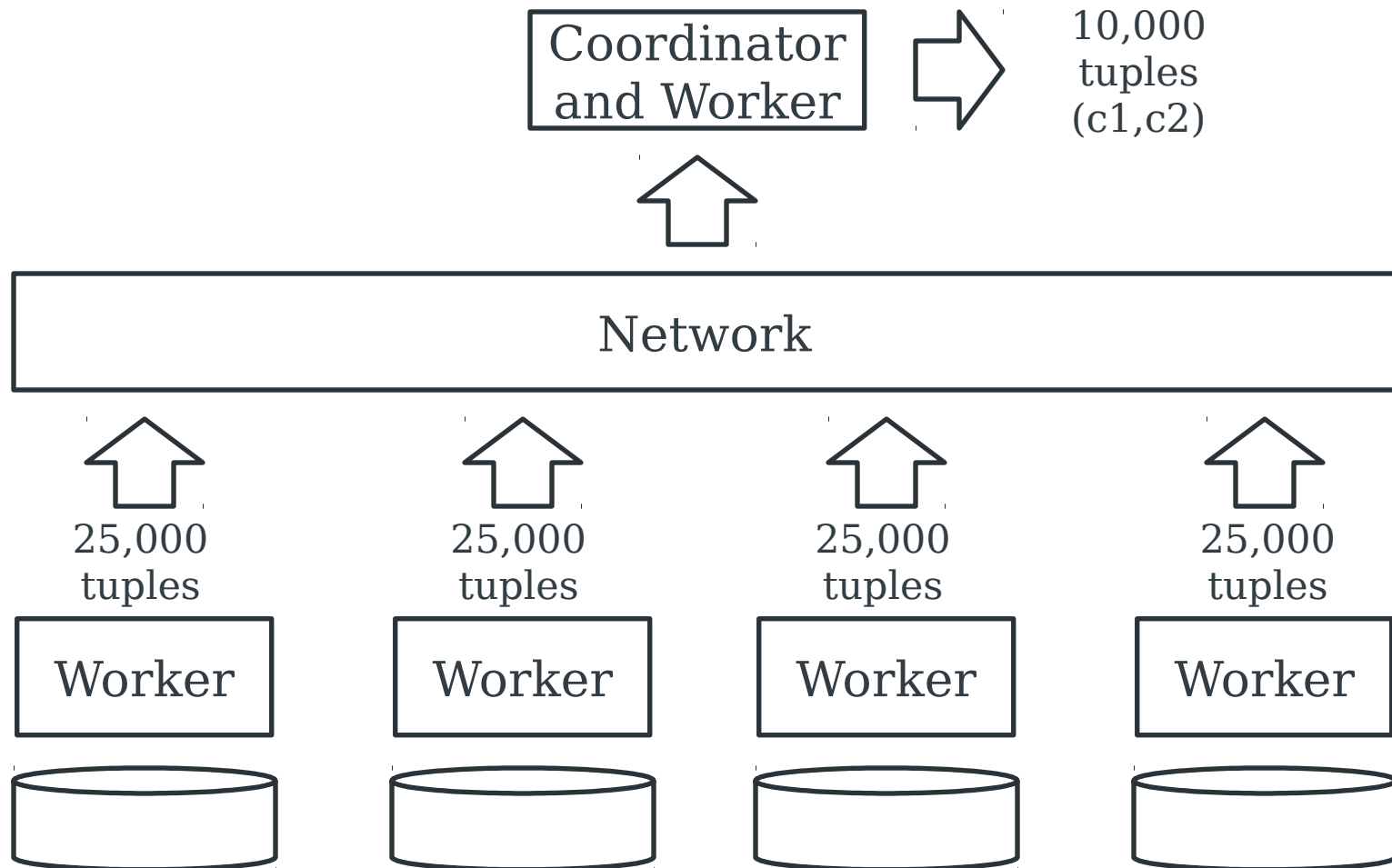
Considerations for query plans:

- Data shipping
- Query shipping

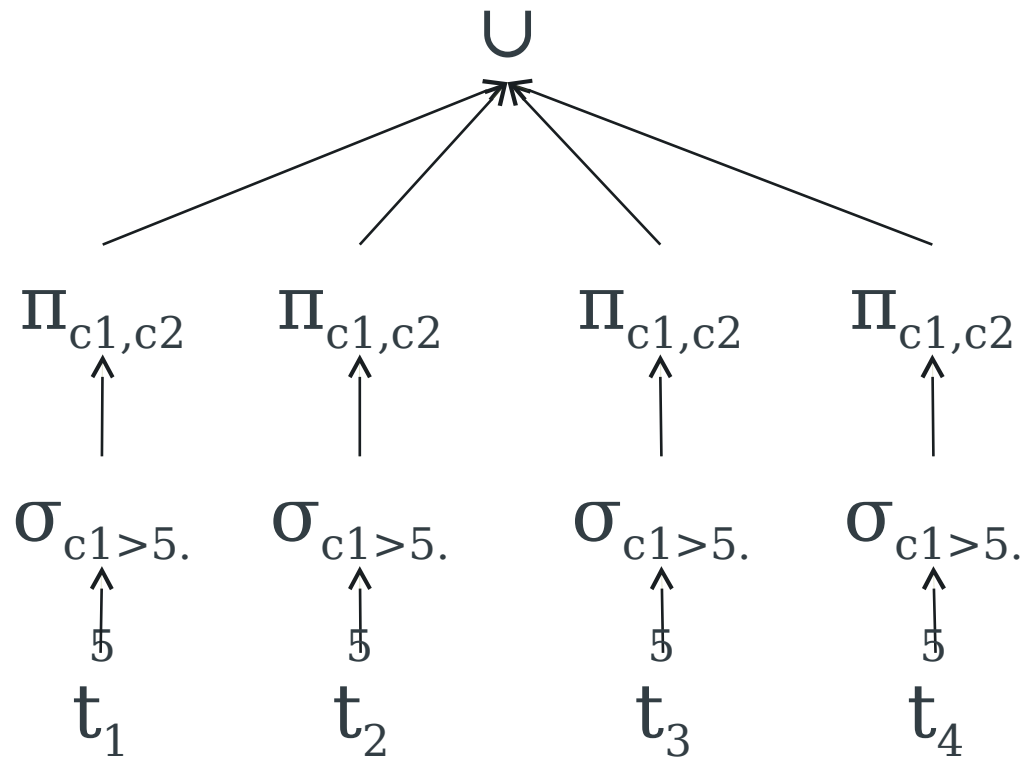
Data Shipping



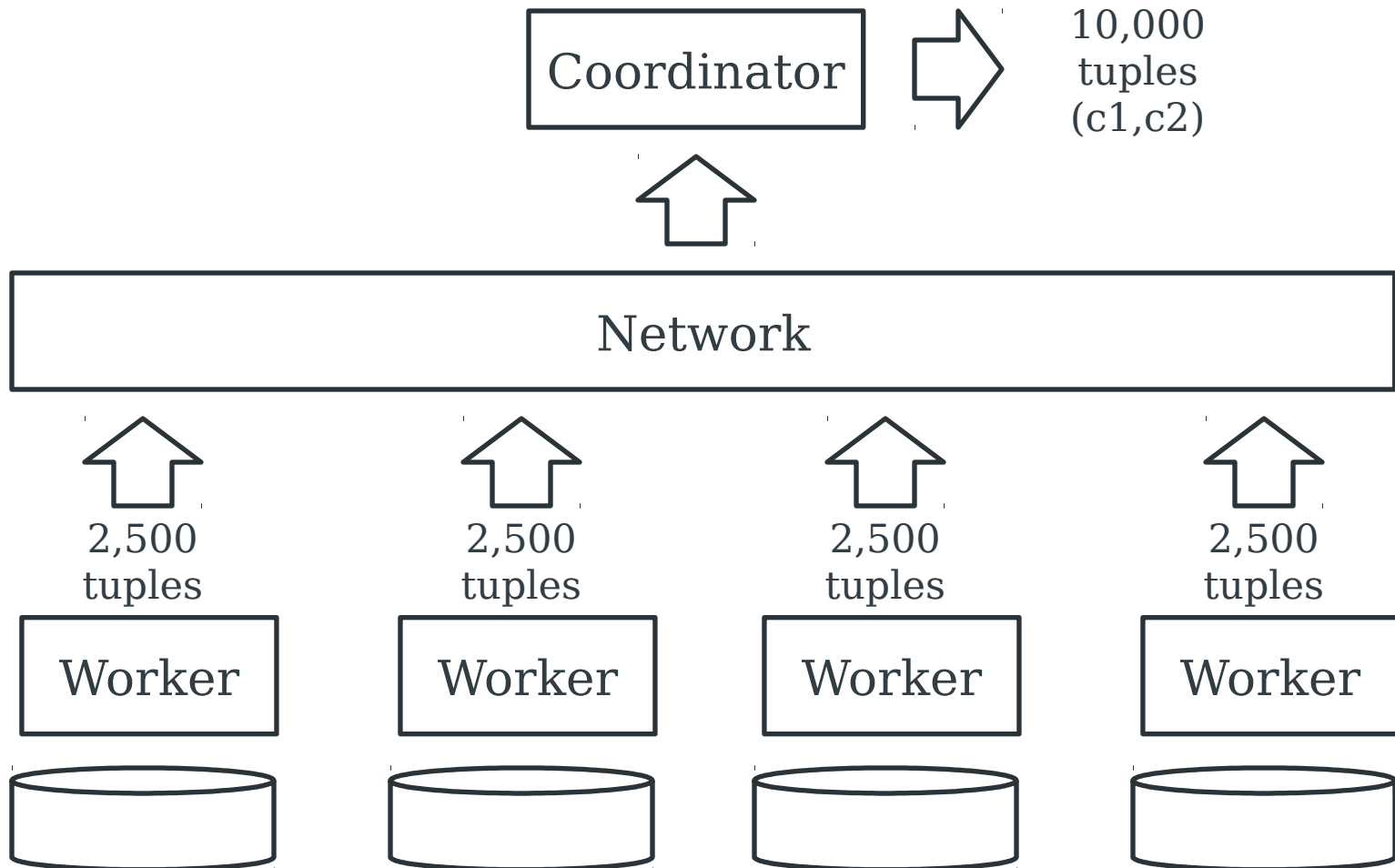
Data Shipping



Query Shipping



Query Shipping



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

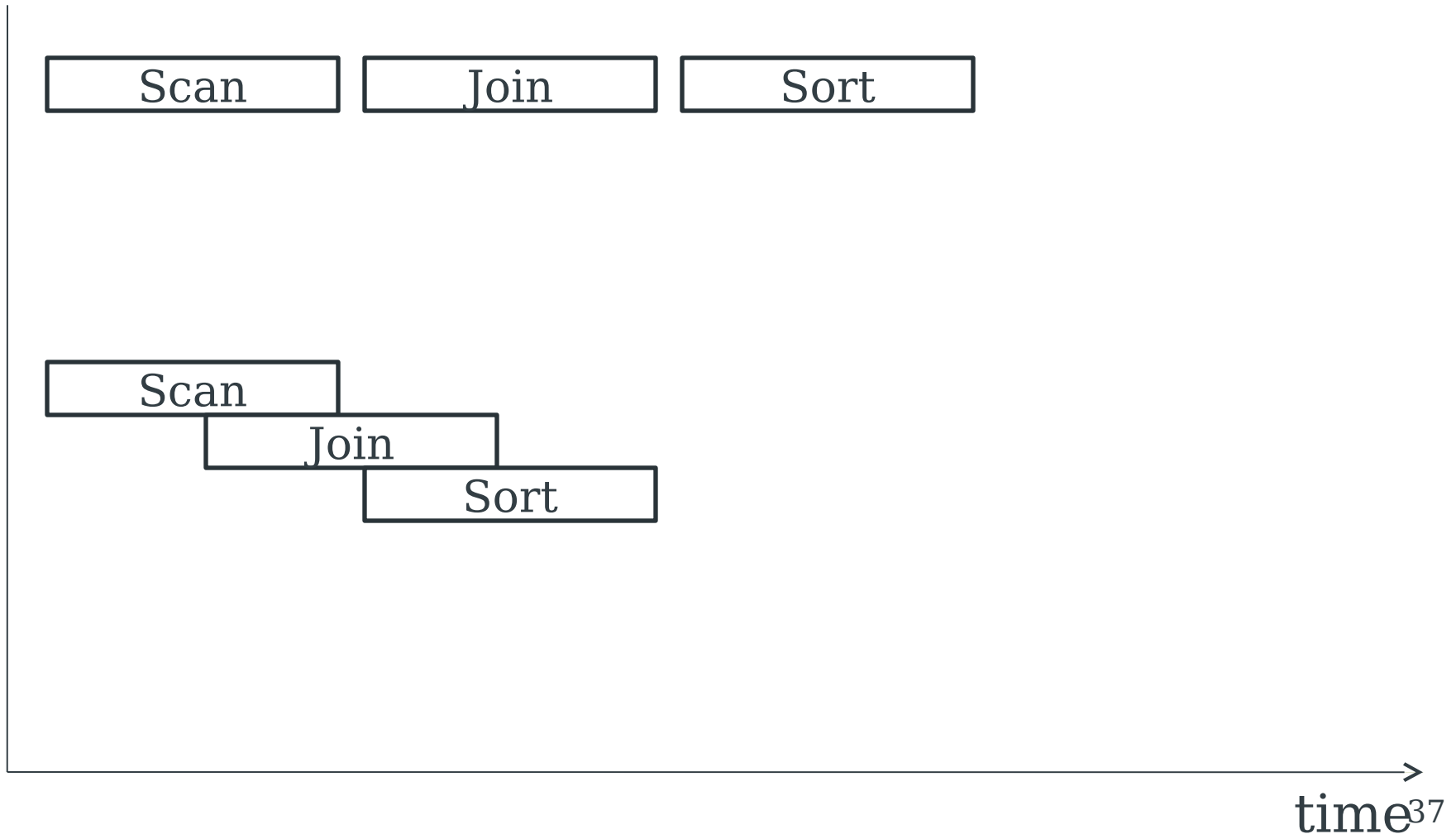
Inter-Operator Parallelism

Inter-Operator Parallelism

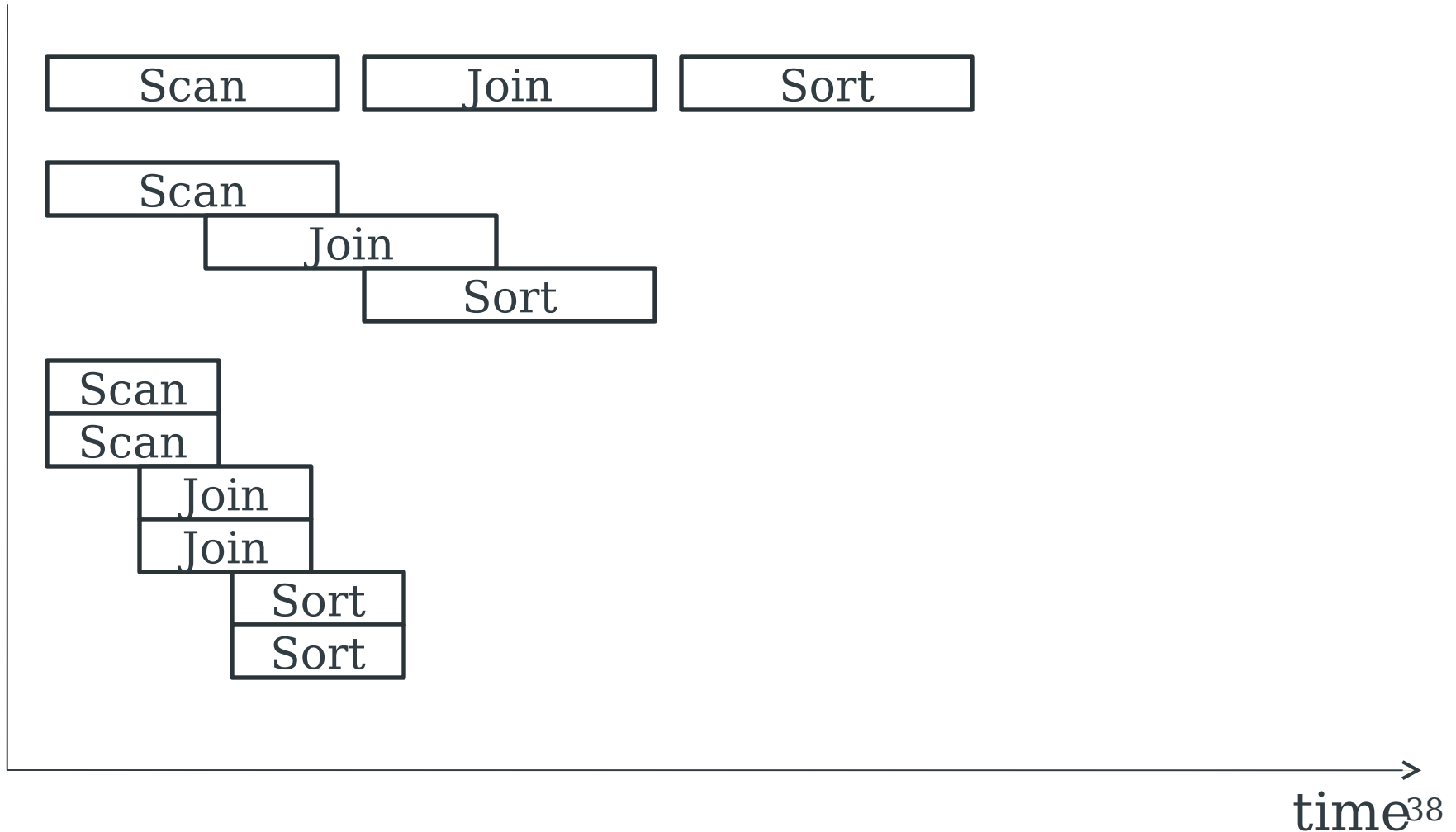
Allows operators with a producer-consumer dependency to be executed concurrently

- 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

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

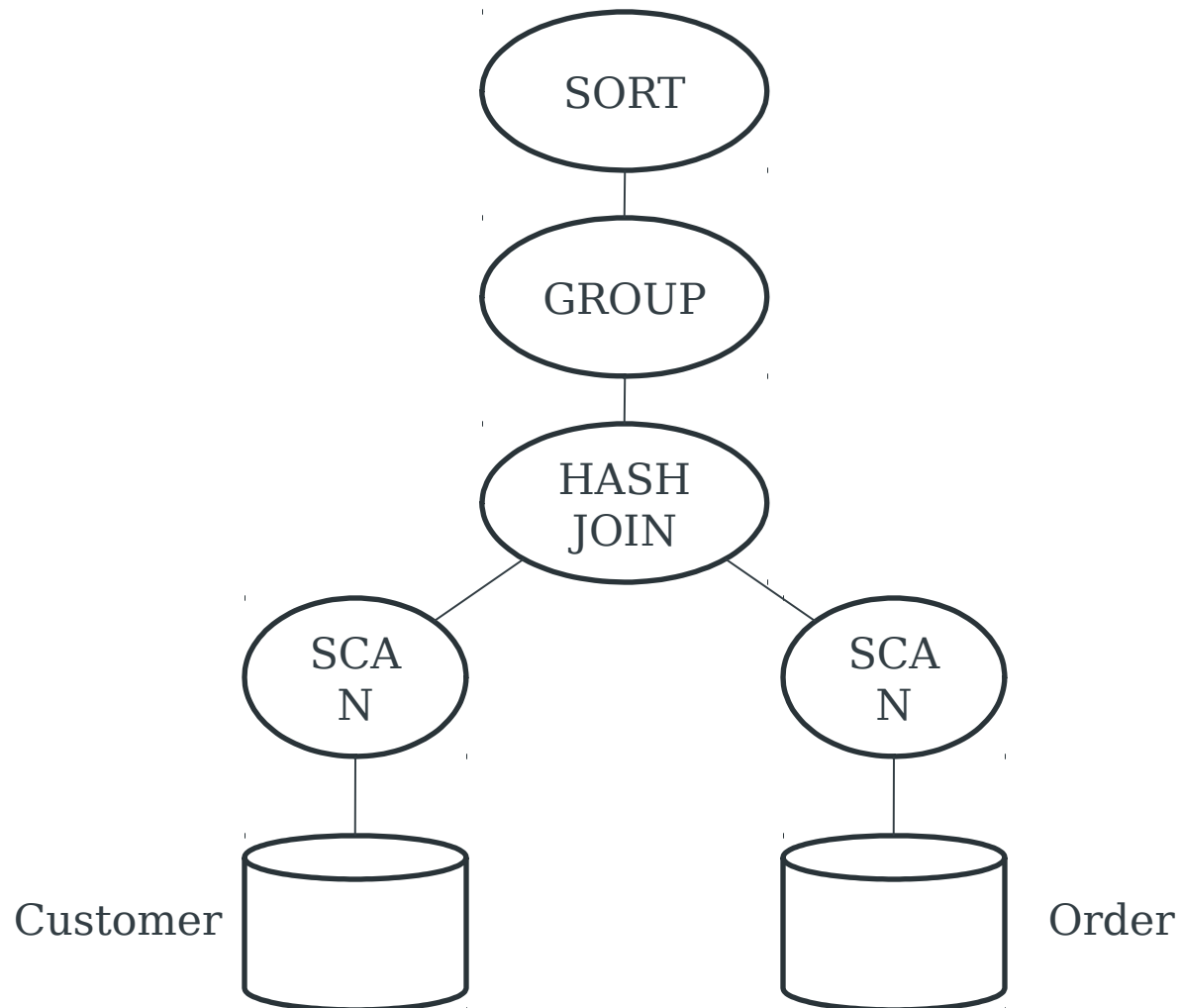
Provides mechanism to support both vertical and horizontal parallelism

Exchange Operators

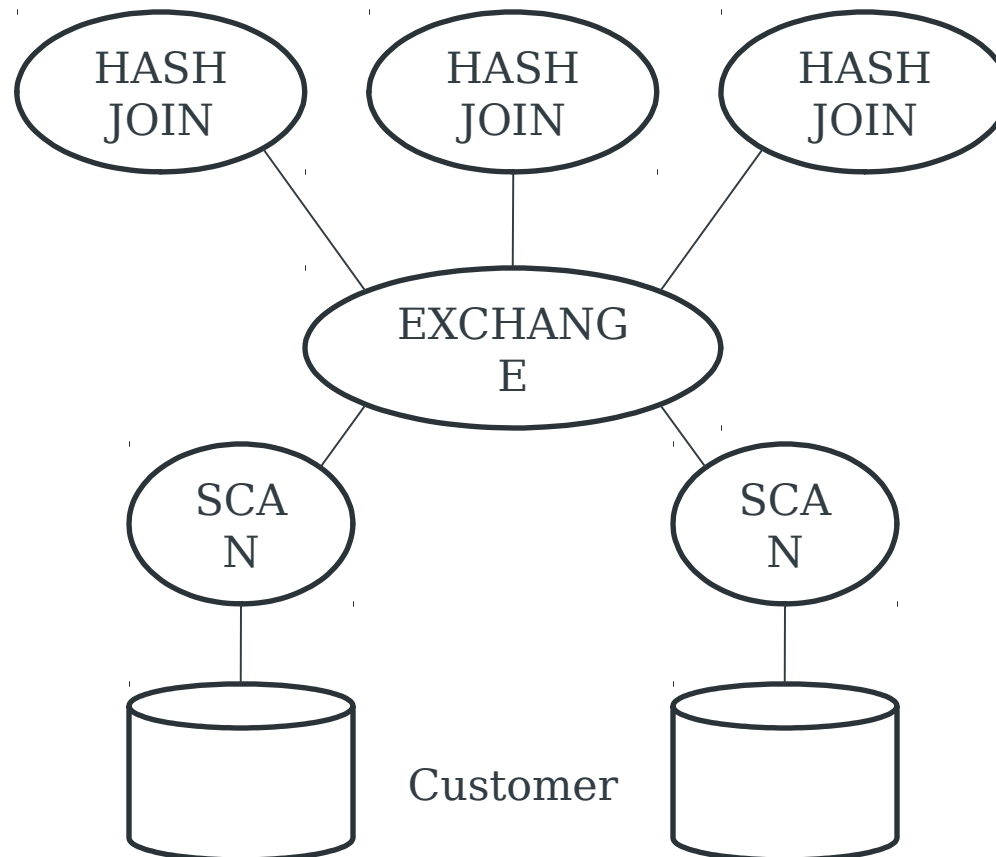
Example query:

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

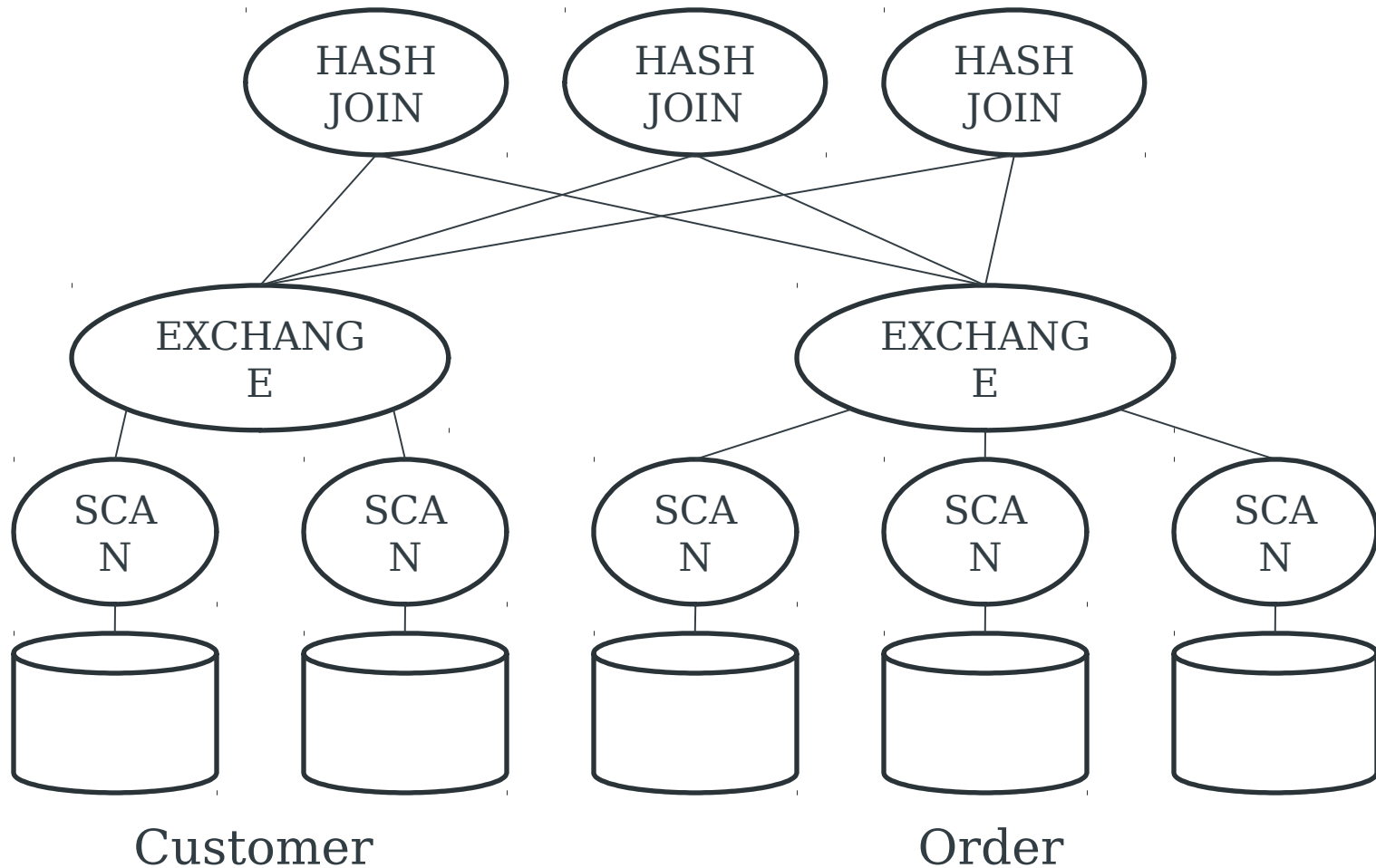

Exchange Operators

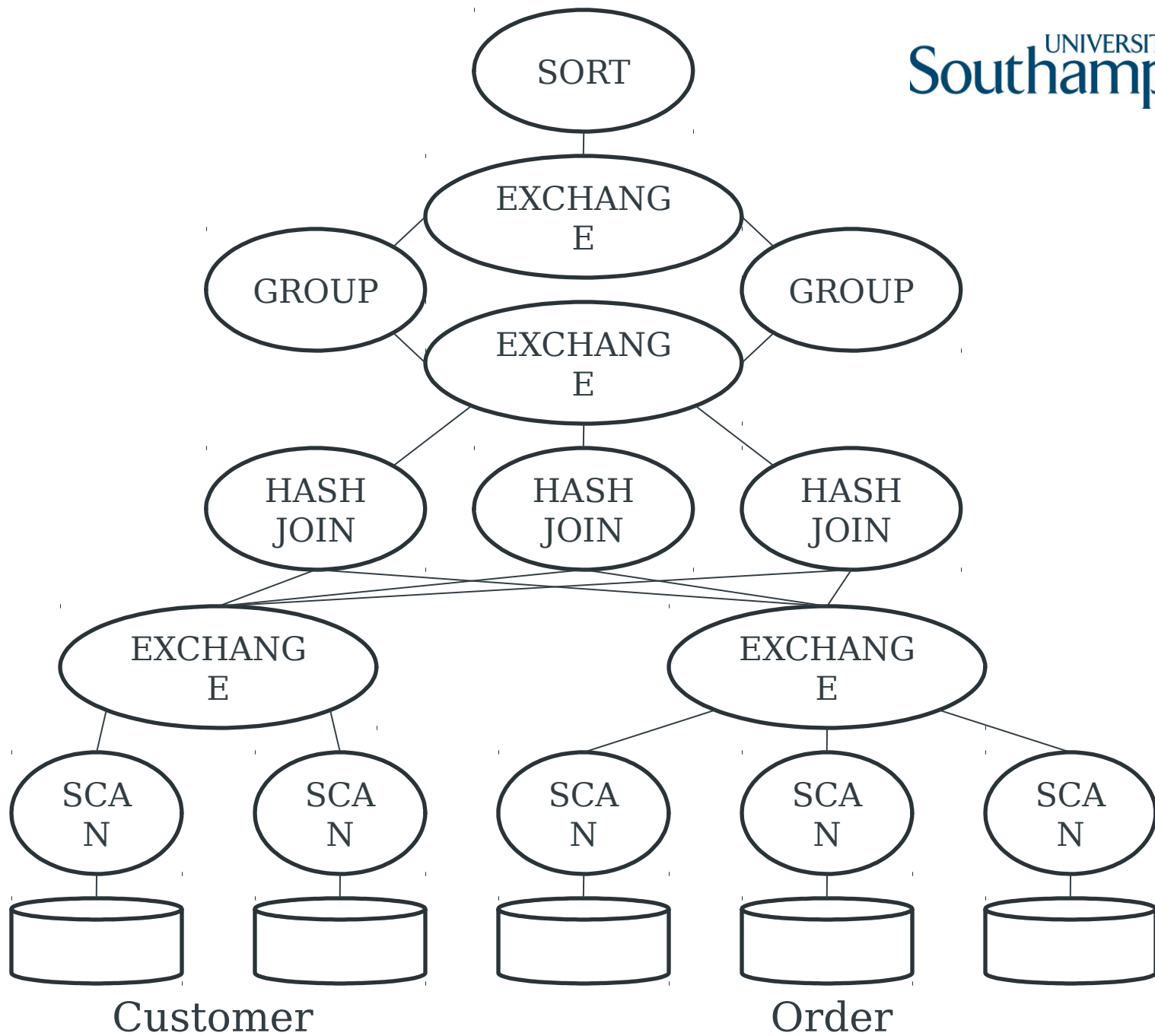


Exchange Operators



Exchange Operators

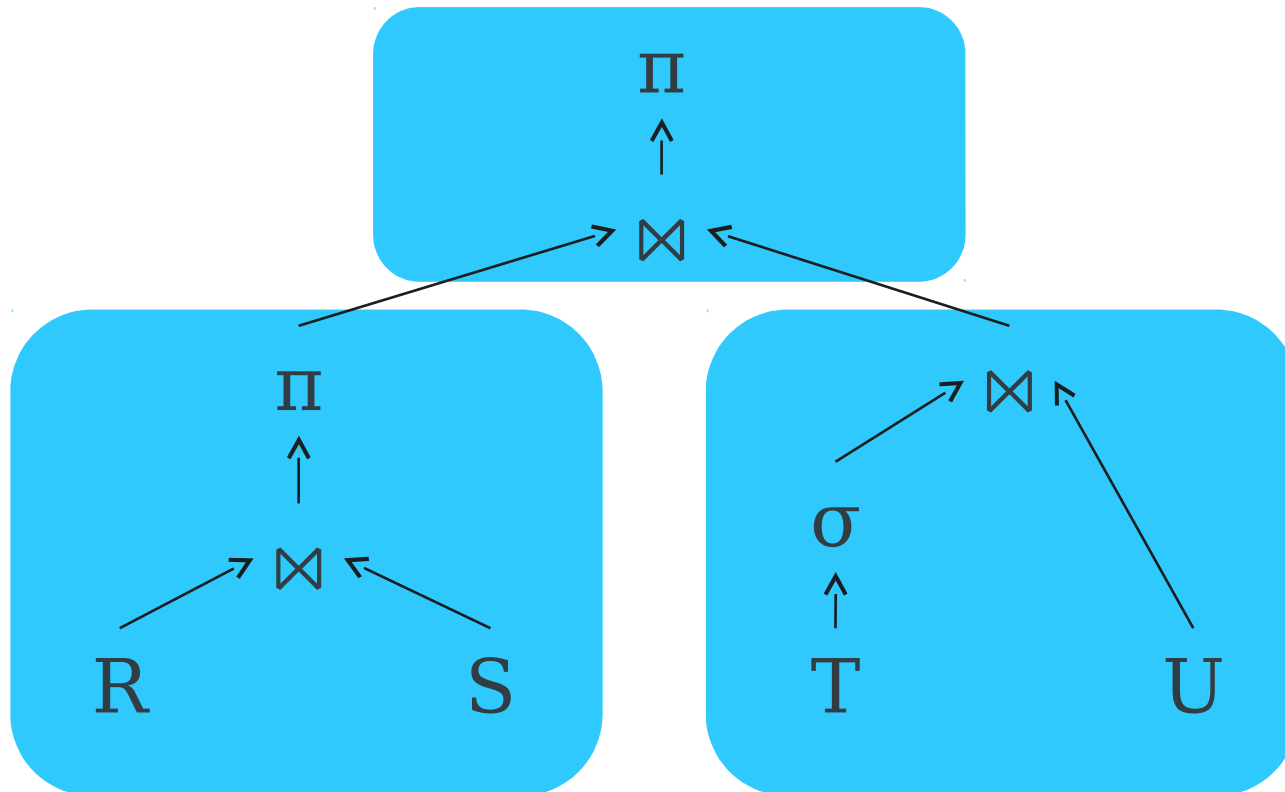




Bushy Parallelism

Bushy Parallelism

Execute subtrees concurrently



Parallel Query Processing

Some Parallel Queries

- Enquiry
- Collocated Join
- Directed Join
- Broadcast Join
- Repartitioned Join

Combine aspects of intra-operator and bushy parallelism

Orders Database

CUSTOMER

<u>CUSTKE</u> Y	C_NAME	...	C_NATION	...
--------------------	--------	-----	----------	-----

ORDER

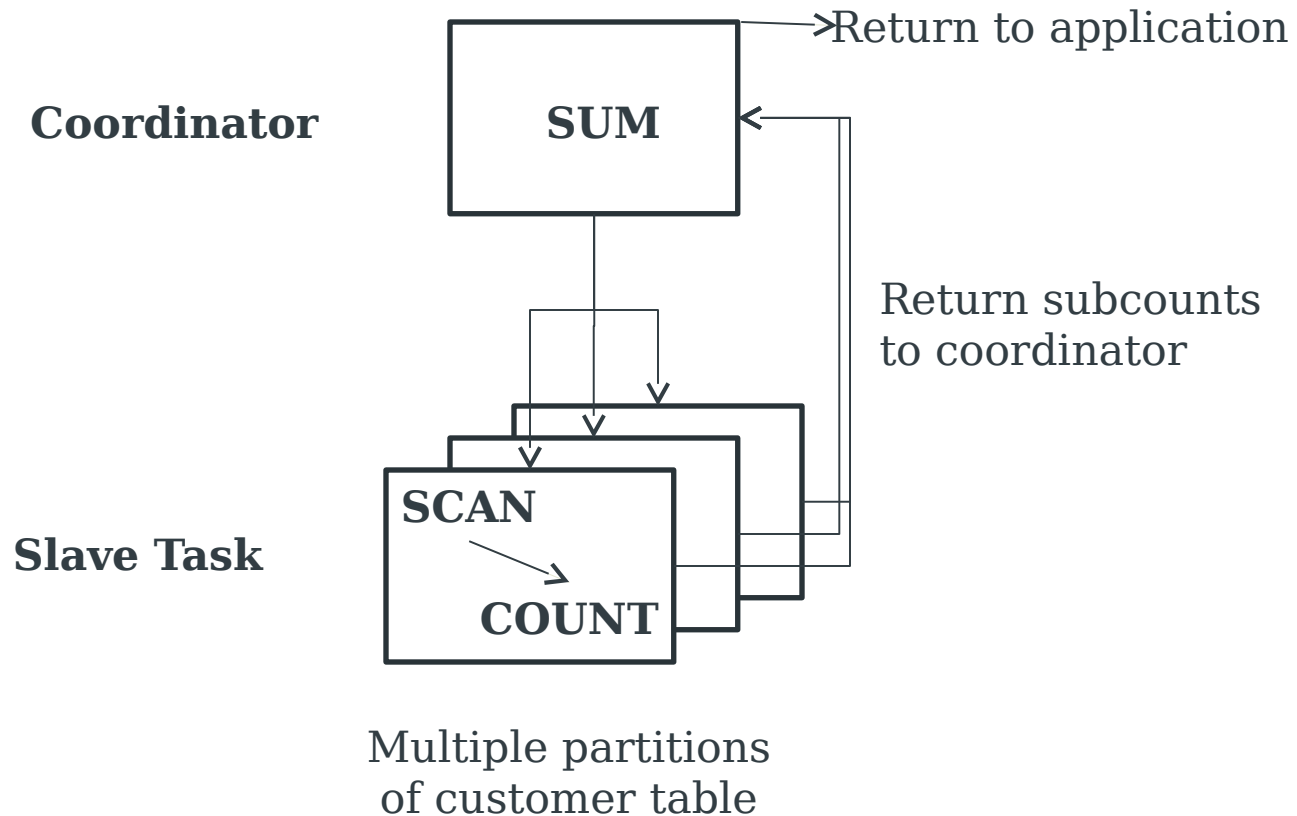
<u>ORDERKE</u> Y	DATE	...	CUSTKE Y	...	SUPPKE Y	...
---------------------	------	-----	-------------	-----	-------------	-----

SUPPLIER

<u>SUPPKE</u> Y	S_NAME	...	S_NATION	...
--------------------	--------	-----	----------	-----

Enquiry/Query

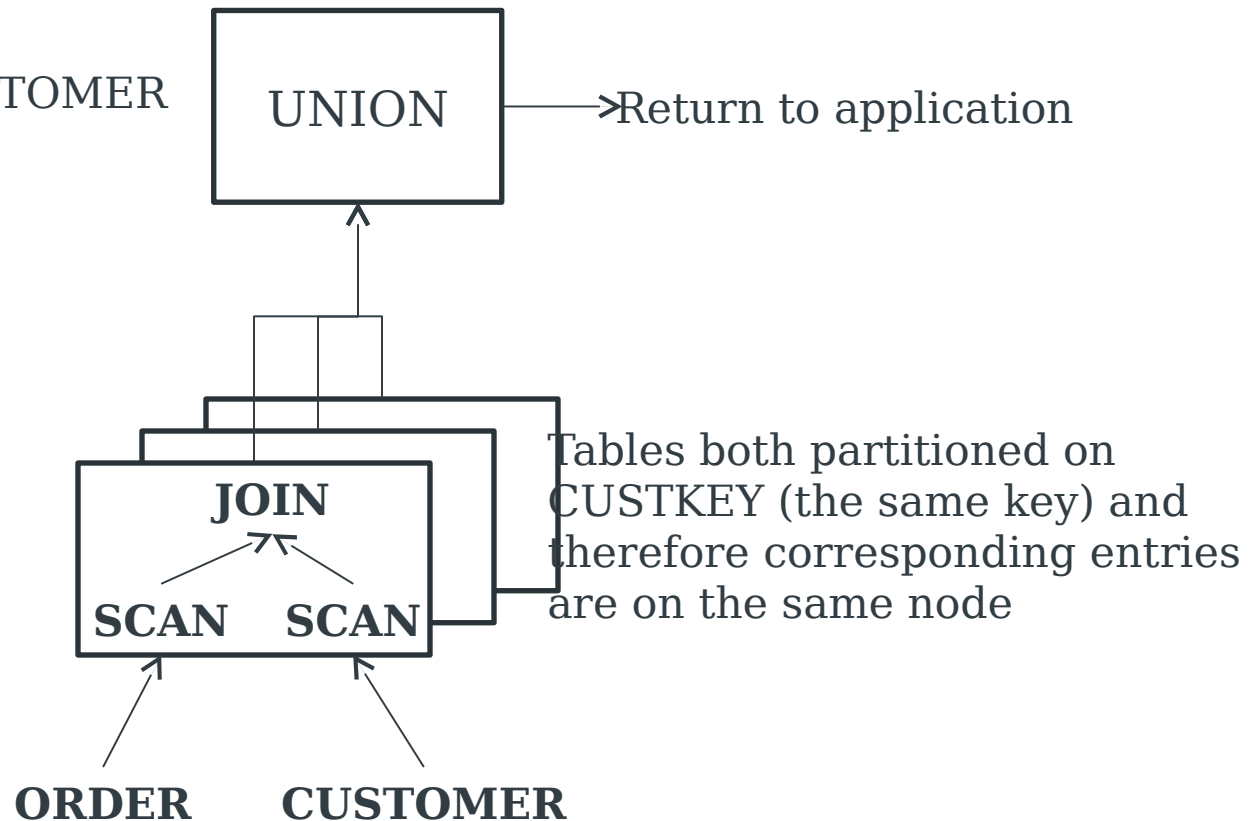
“How many customers live in the UK?”



Collocated Join

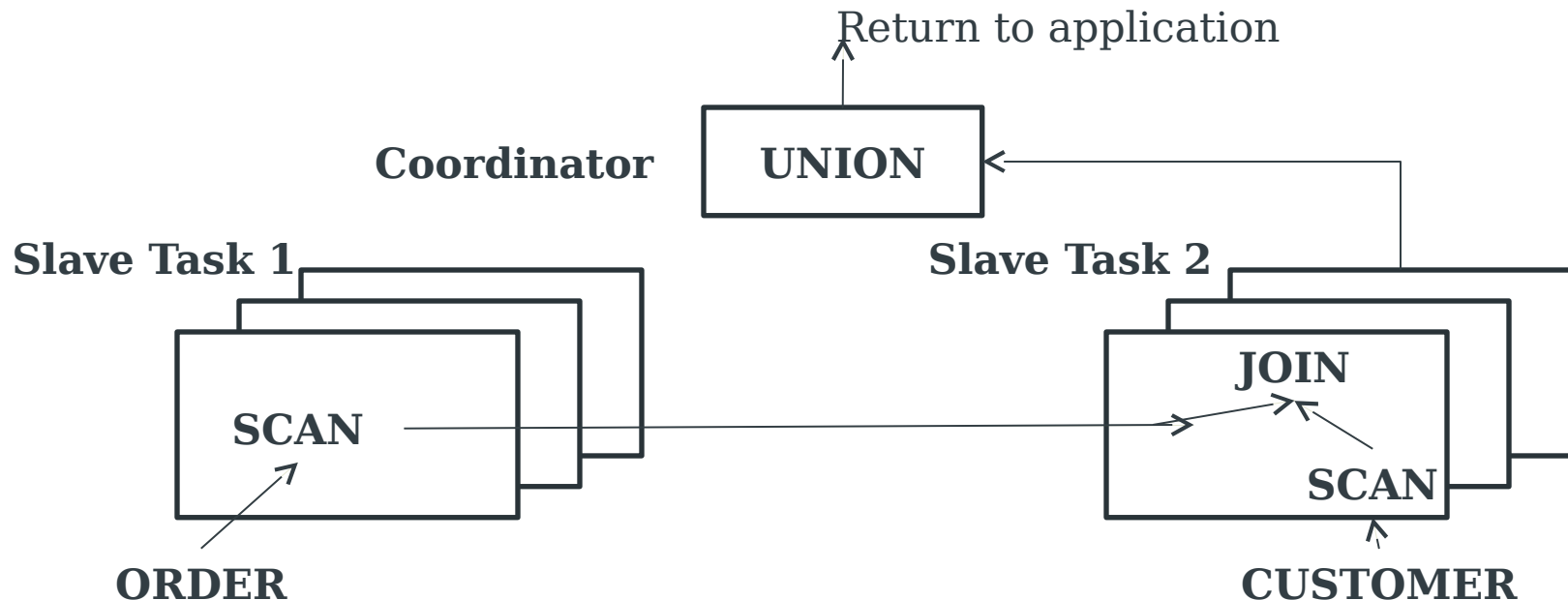
“Which customers placed orders in July?”

Requires a JOIN of CUSTOMER and ORDER



Directed Join

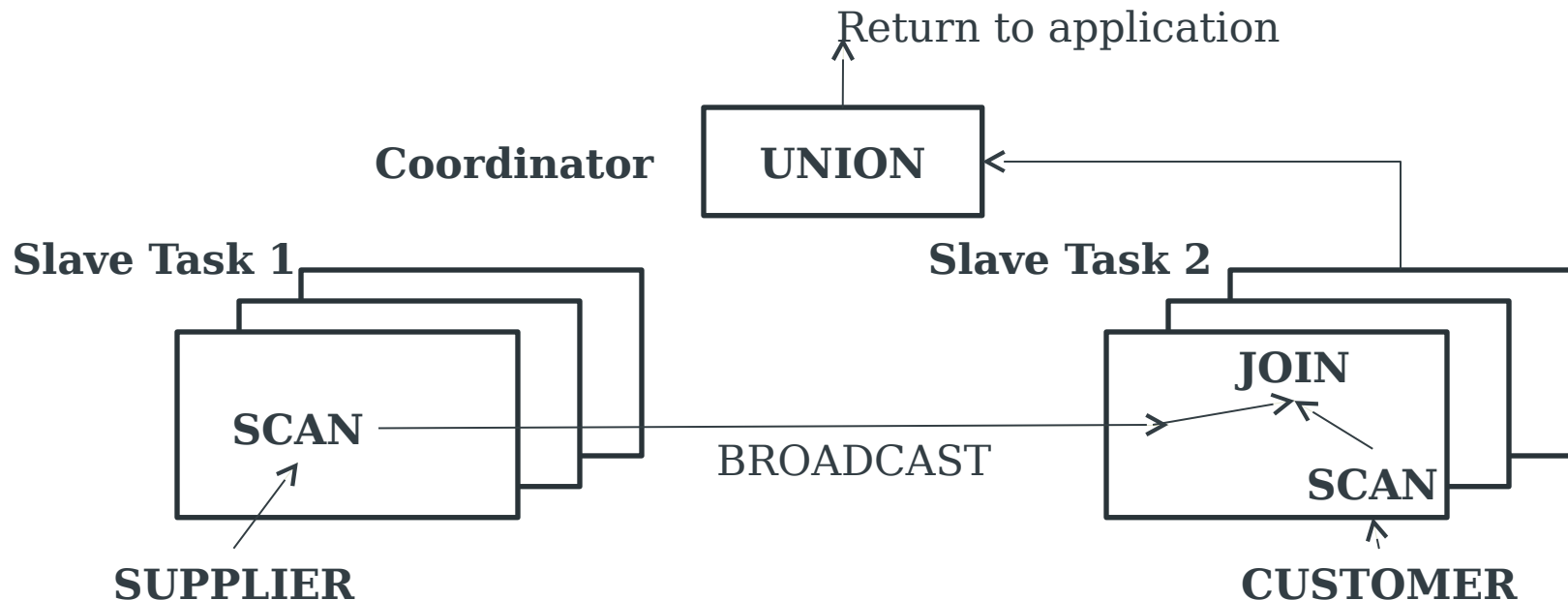
“Which customers placed orders in July?”
(tables have different keys)



ORDER partitioned on ORDERKEY, CUSTOMER partitioned on CUSTKEY
Retrieve rows from ORDER, then use ORDER.CUSTKEY to direct
appropriate rows to nodes with CUSTOMER

Broadcast Join

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



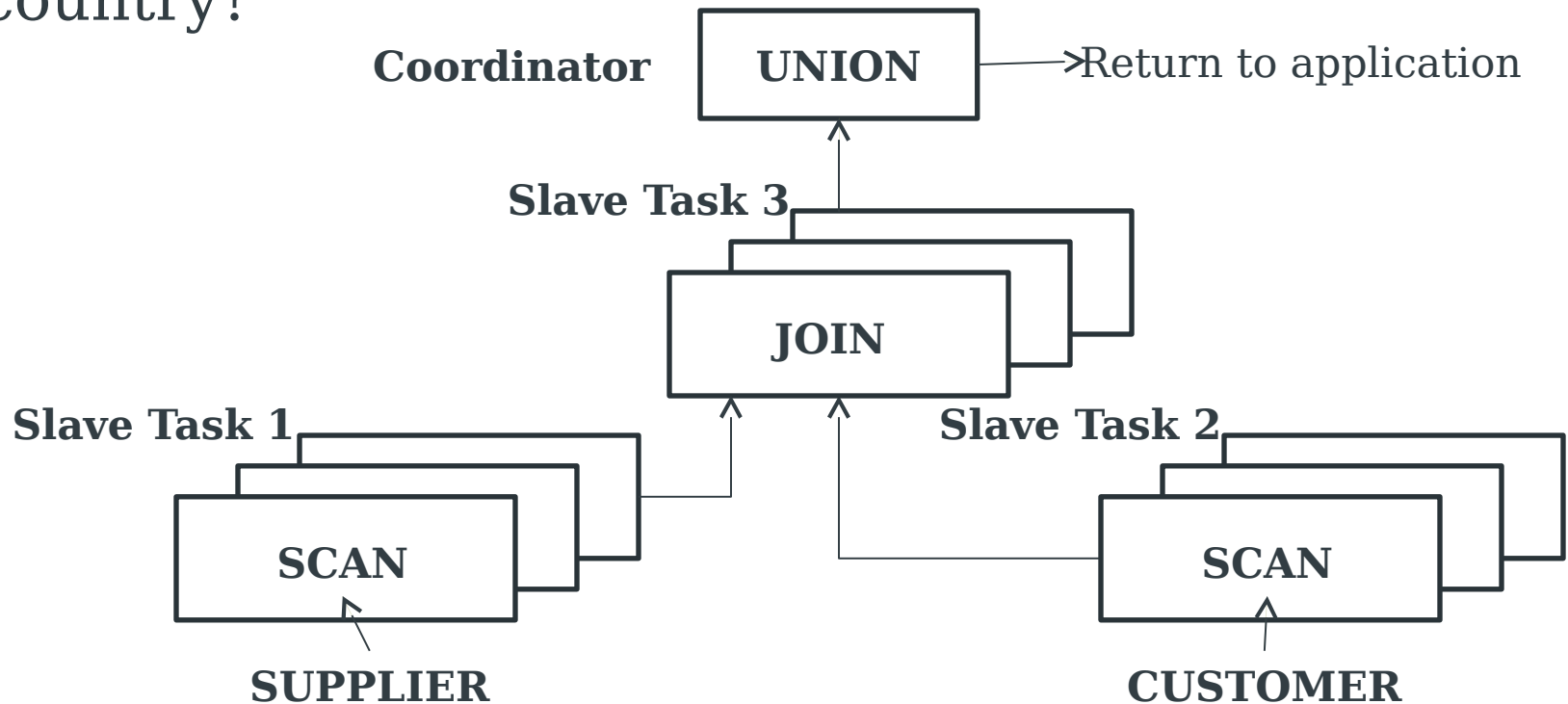
SUPPLIER partitioned on SUPPKEY, CUSTOMER on CUSTKEY.

Join required on *_NATION

Send all SUPPLIER to each CUSTOMER node

Repartitioned Join

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



SUPPLIER partitioned on SUPPKEY, CUSTOMER on CUSTKEY.
Join required on *_NATION. Repartition both tables on *_NATION to
localise and minimise the join effort

Concurrency Control

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’

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

- One approach – Timeouts
- 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
- Rollback T2 to let T1 to proceed
- Restart T2, which can now complete

Resolving Deadlocks

- More sophisticated approach (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

Reliability

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

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*

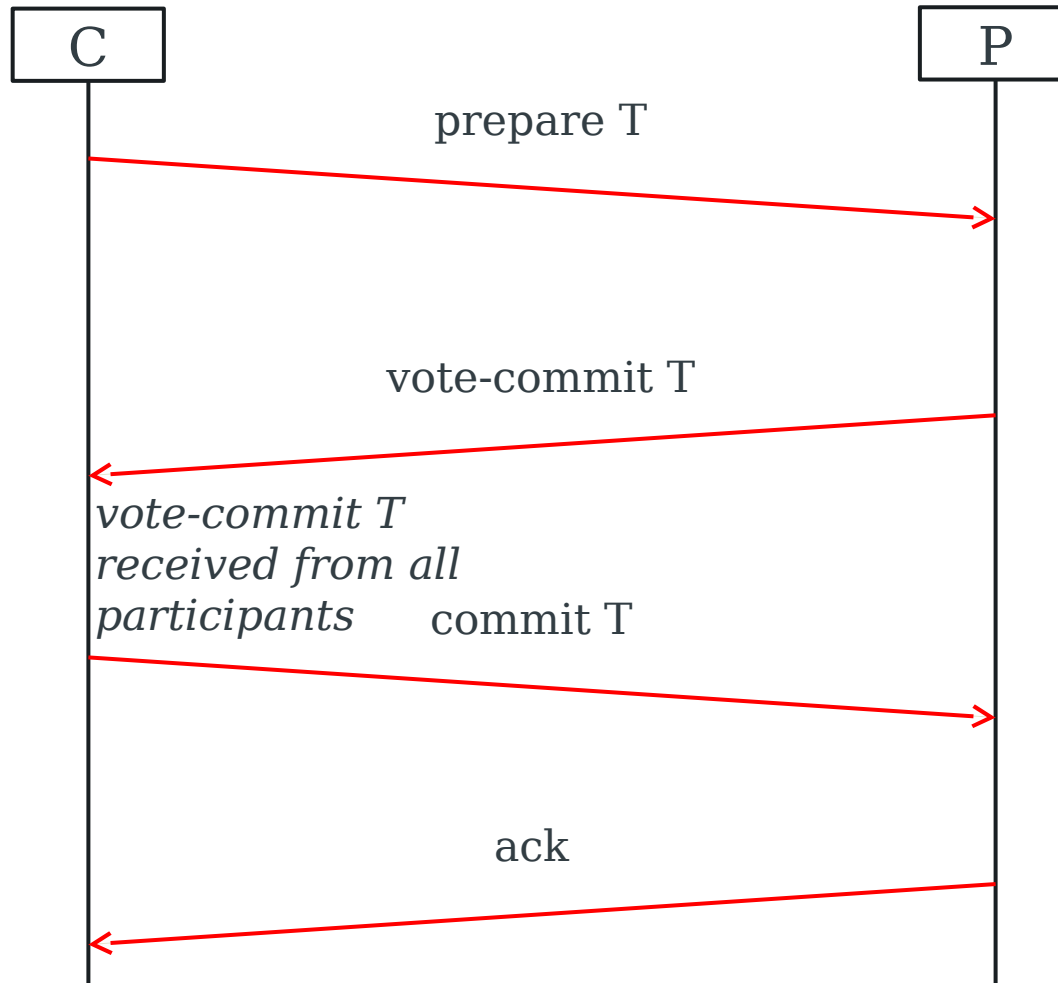
Phase 1: Voting

- 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

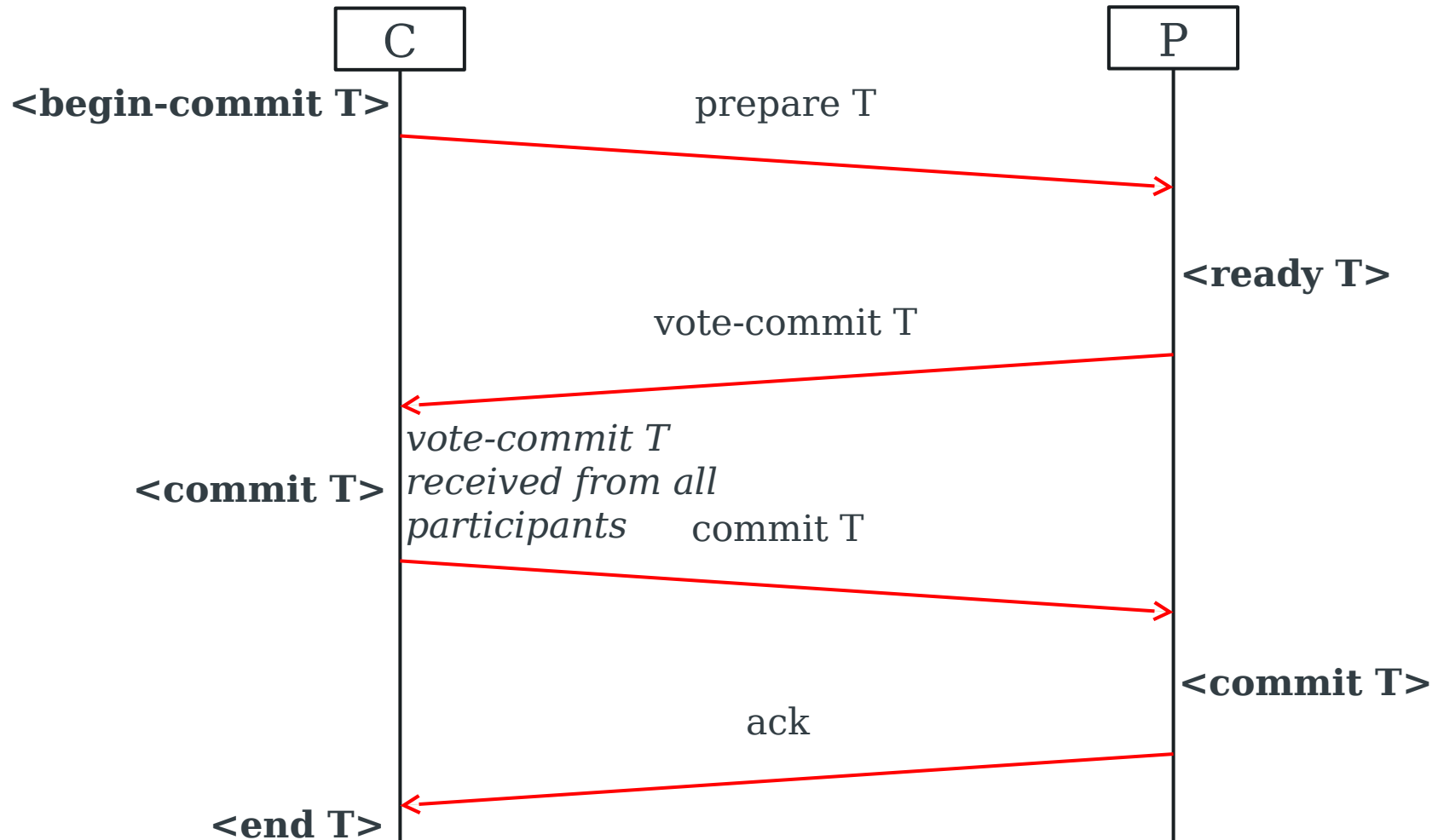
Phase 2: Decision

- 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.

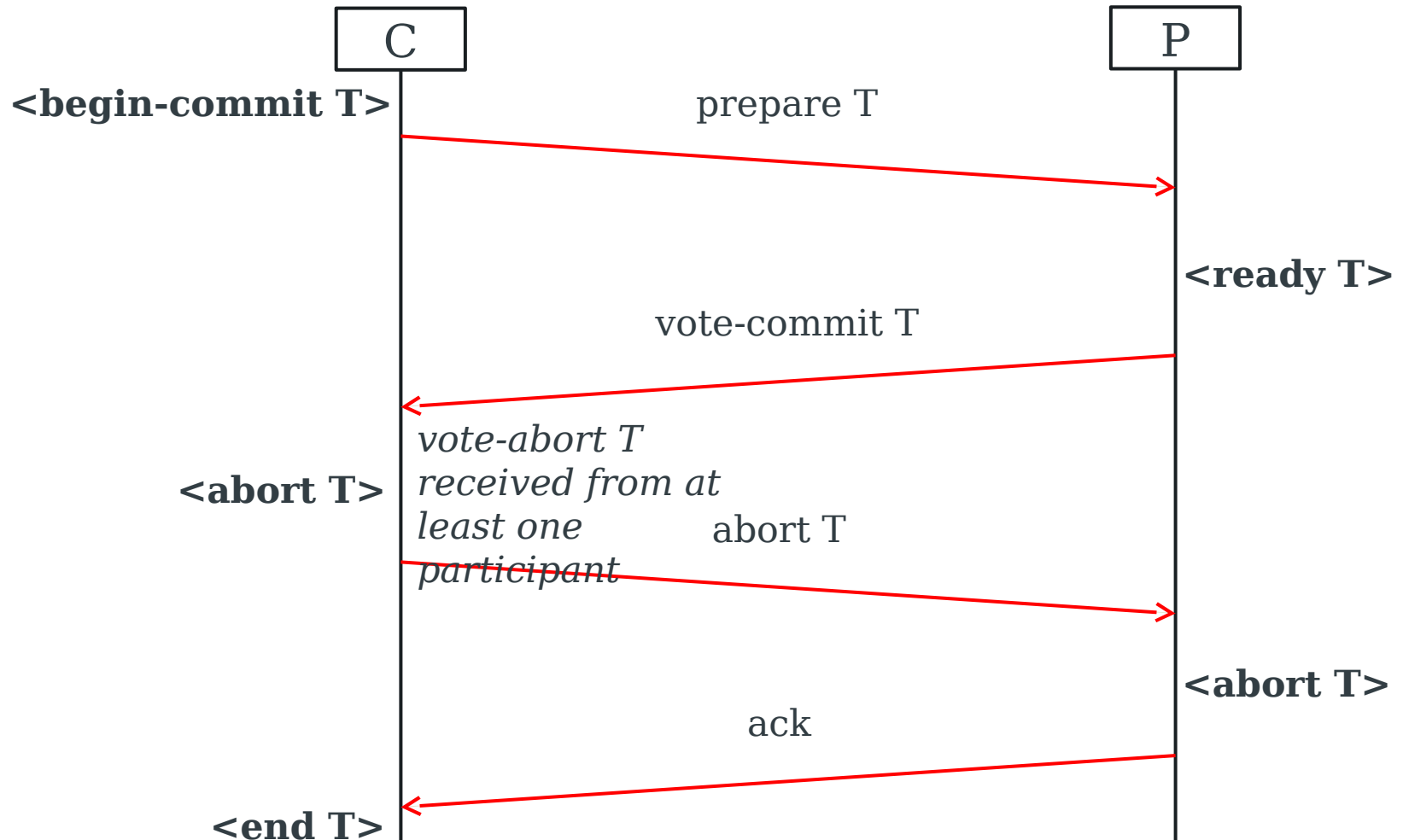
Normal Operation



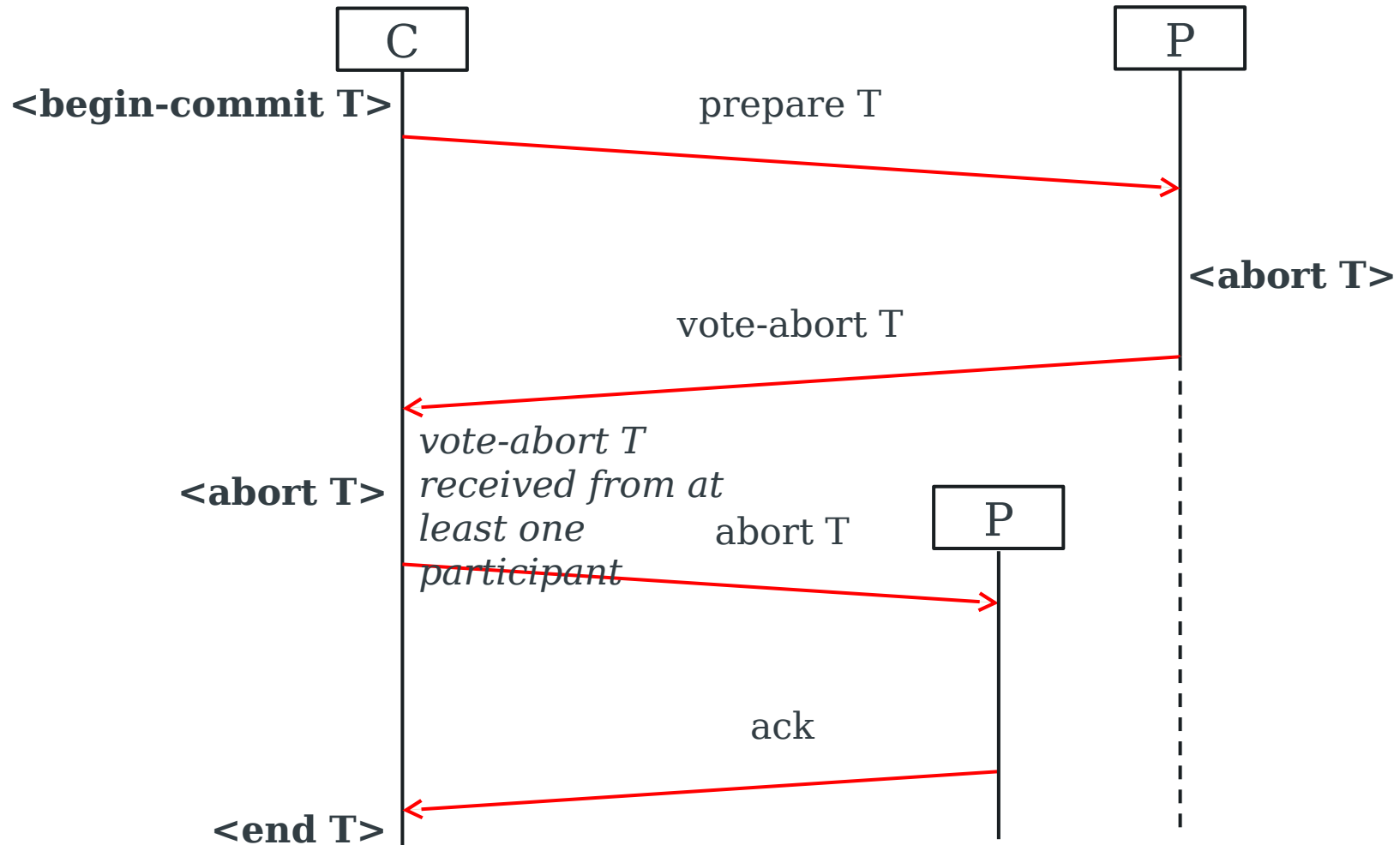
Logging



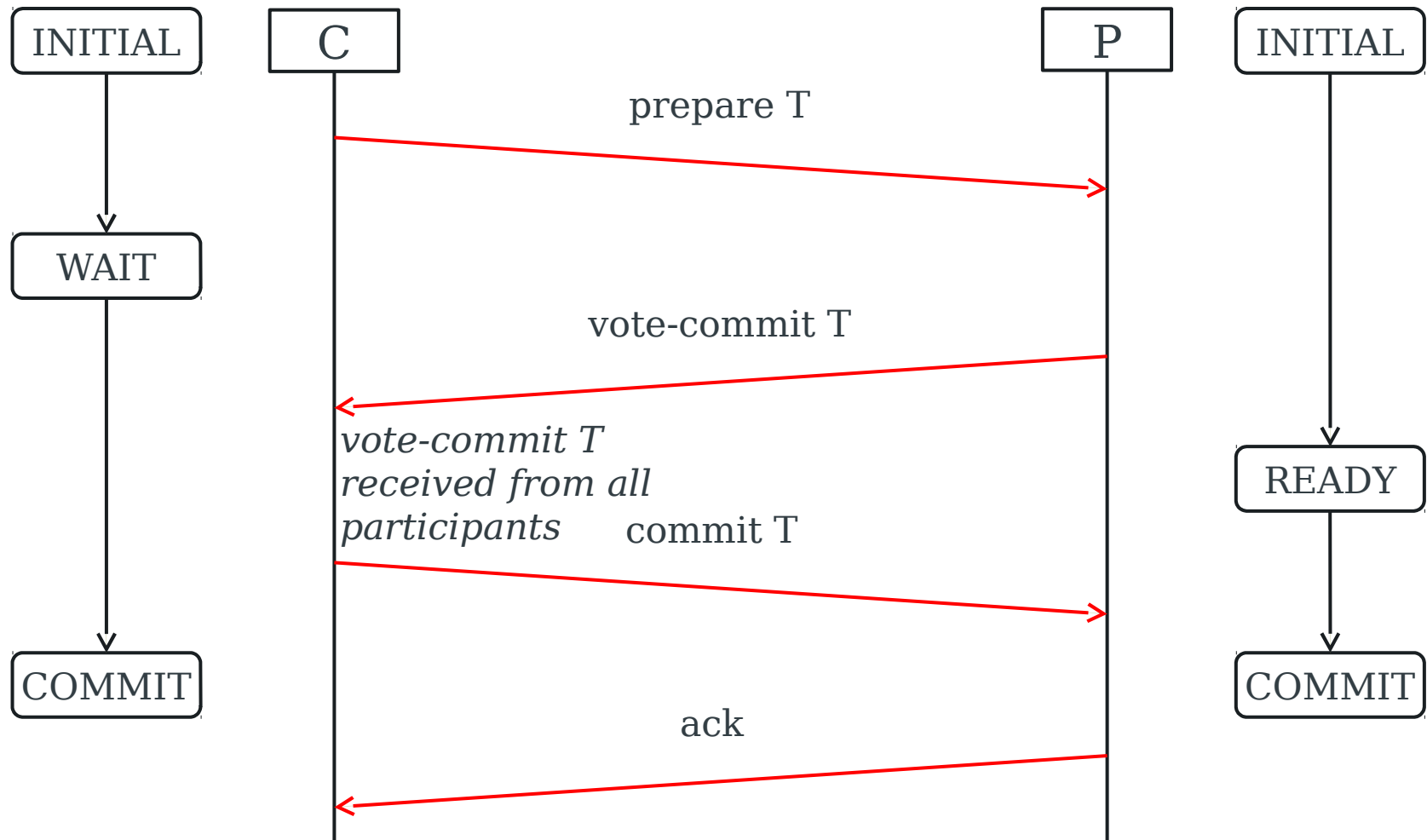
Aborted Transaction



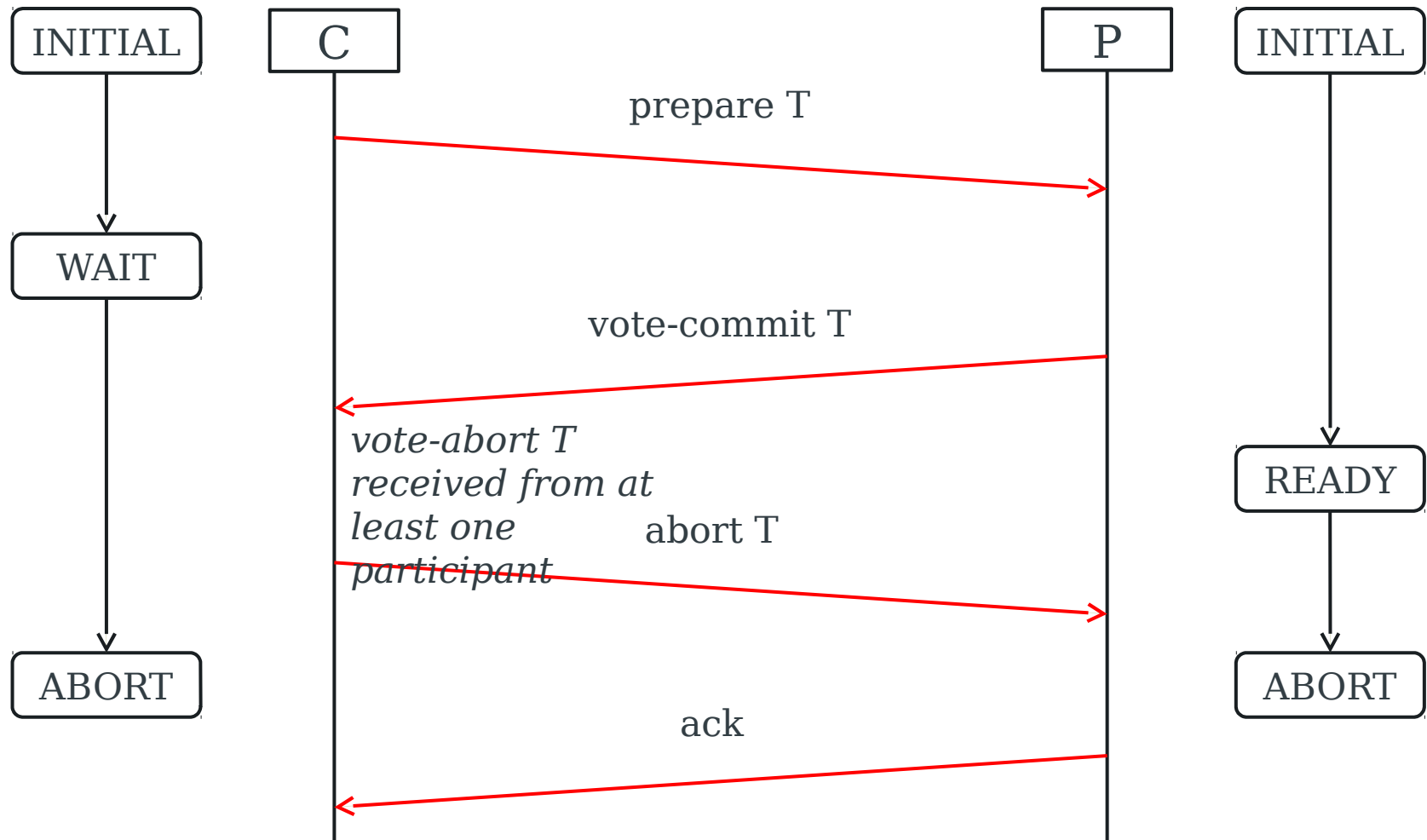
Aborted Transaction



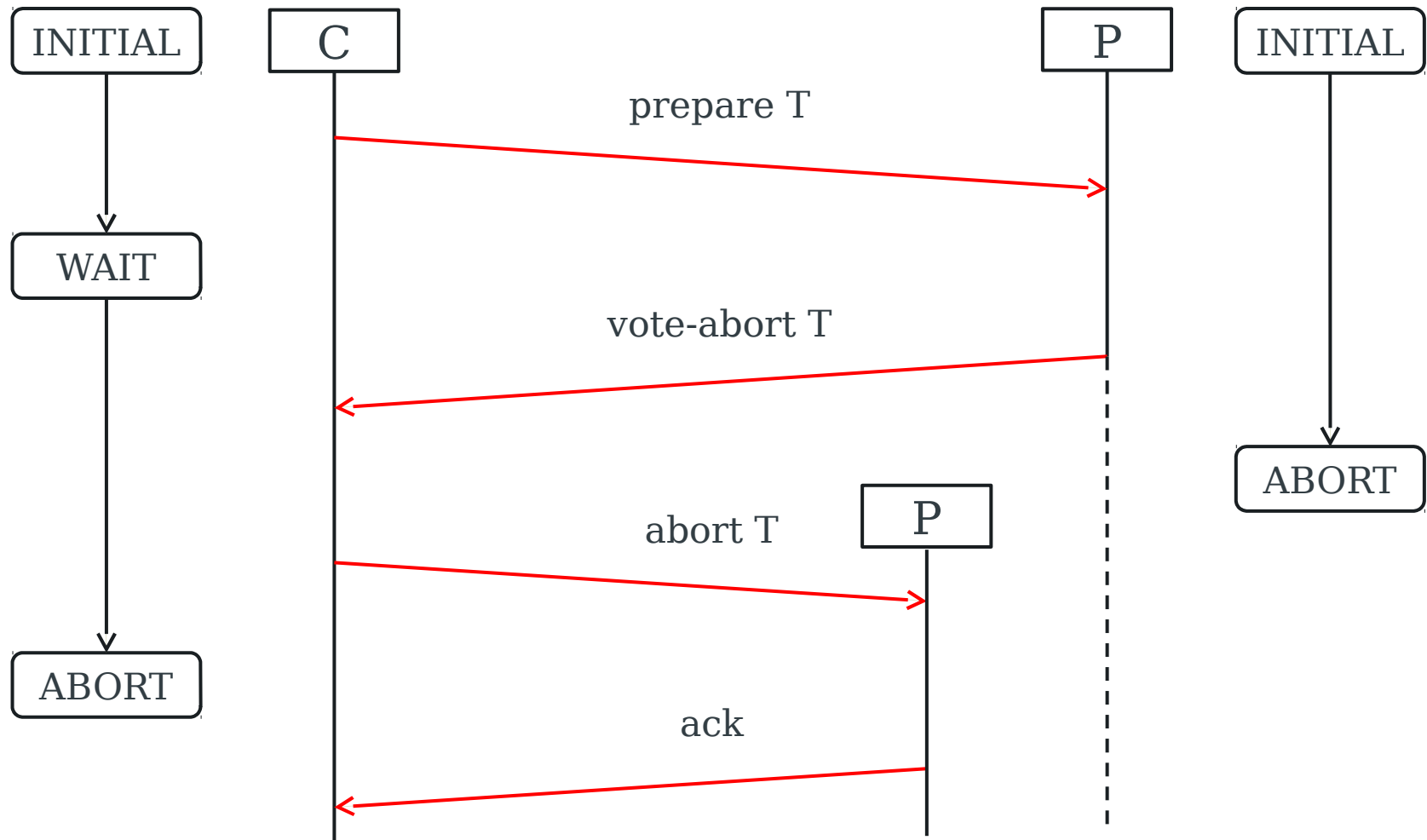
State Transitions



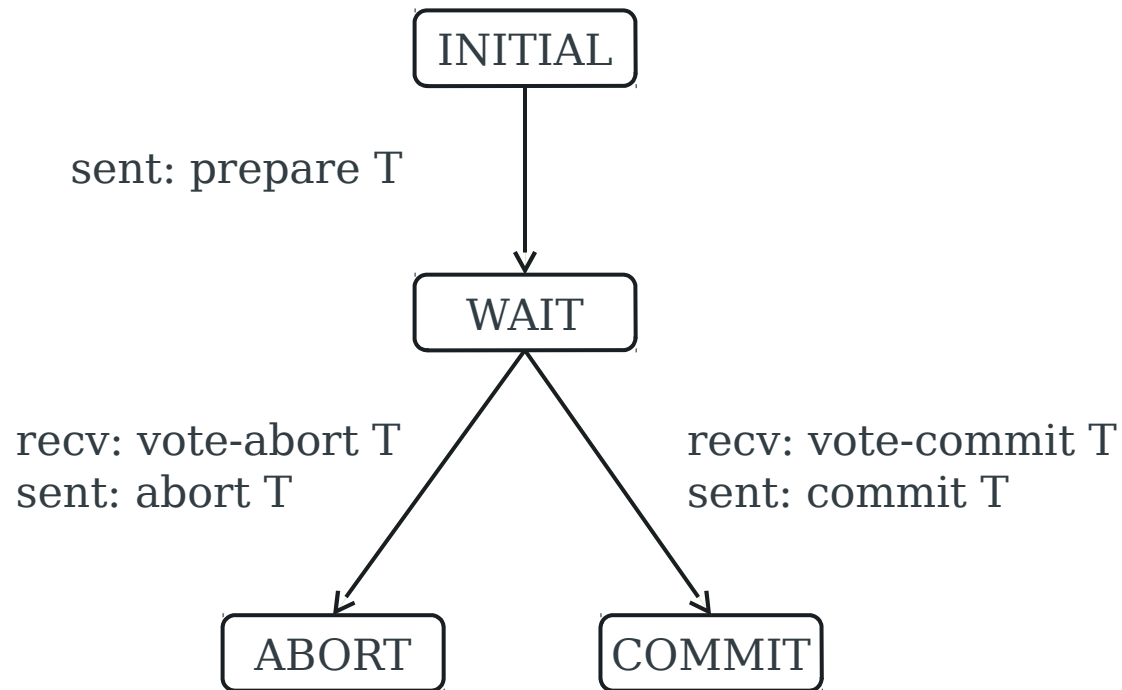
State Transitions



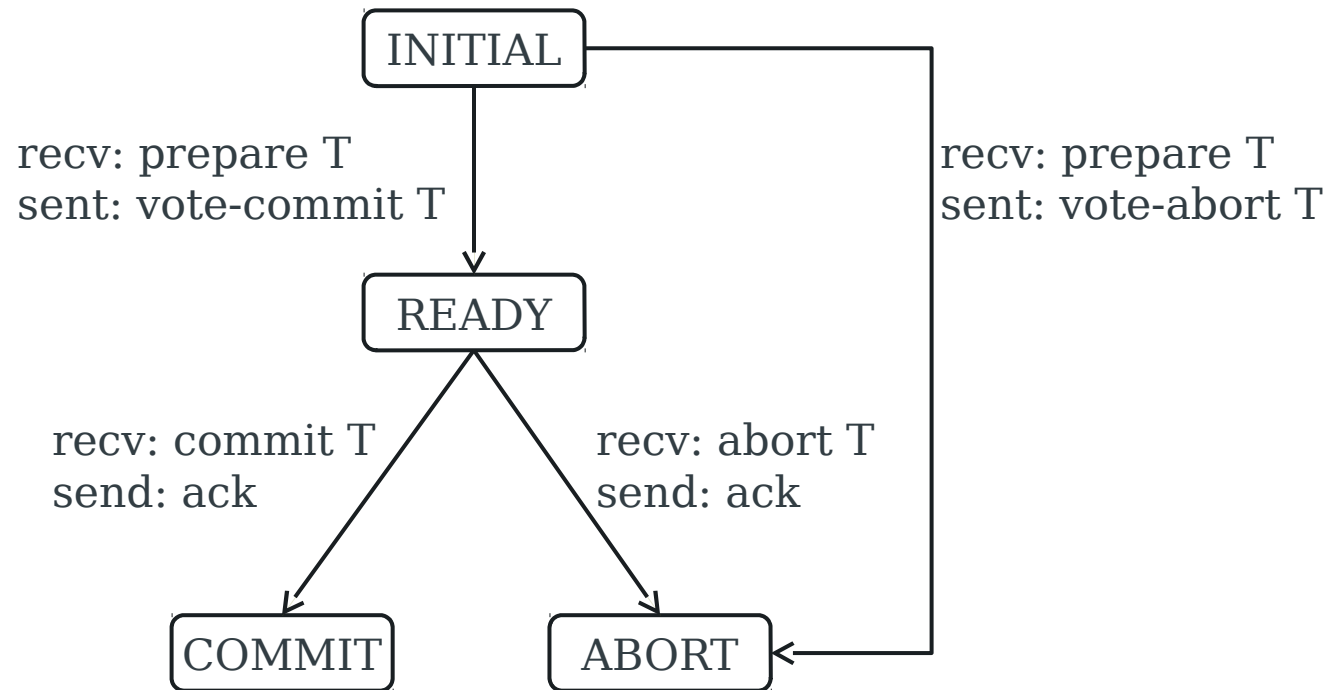
State Transitions



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

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)

Timeout in READY

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

Recovery Protocol: Coordinator

Failure in INITIAL

- Commit not yet begun, restart commit procedure

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”

Failure in COMMIT/ABORT

- If coordinator has received all “ack” messages, complete successfully
- Otherwise, terminate

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”

Failure in READY

- Participant has voted, but doesn't know what the global decision was
- Cooperative termination protocol

Failure in COMMIT/ABORT

- Resend “ack” message

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