Electronics and Computer Science

UNIVERSITY OF Southampton

Distributed Databases

COMP3211 Advanced Databases

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

2014-2015

Overview

Fragmentation

- Horizontal (primary and derived), vertical, hybrid

Query processing

- Localisation, optimisation (semijoins)

Concurrency control

- Centralised 2PL, Distributed 2PL, deadlock

Reliability

- Two Phase Commit (2PC)

The CAP Theorem

What is a distributed database?

A collection of sites connected by a communications network

Each site is a database system in its own right, but the sites have agreed to work together

A user at any site can access data anywhere as if data were all at the user's own site

DDBMS Principles

Local autonomy

The sites in a distributed database system should be autonomous or independent of each other

Each site should provide its own security, locking, logging, integrity, and recovery. Local operations use and affect only local resources and do not depend on other sites

No reliance on a central site

A distributed database system should not rely on a central site, which may be a single point of failure or a bottleneck

Each site of a distributed database system provides its own security, locking, logging, integrity, and recovery, and handles its own data dictionary. No central site must be involved in every distributed transaction.

Continuous operation

A distributed database system should never require downtime

A distributed database system should provide on-line backup and recovery, and a full and incremental archiving facility. The backup and recovery should be fast enough to be performed online without noticeable detrimental affect on the entire system performance.

Location independence

Applications should not know, or even be aware of, where the data are physically stored; applications should behave as if all data were stored locally

Location independence allows applications and data to be migrated easily from one site to another without modifications.

Fragmentation independence

Relations can be divided into fragments and stored at different sites

Applications should not be aware of the fact that some data may be stored in a fragment of a table at a site different from the site where the table itself is stored.

Replication independence

Relations and fragments can be stored as many distinct copies on different sites

Applications should not be aware that replicas of the data are maintained and synchronized automatically.

Distributed query processing

Queries are broken down into component transactions to be executed at the distributed sites

Distributed transaction management

A distributed database system should support atomic transactions

Critical to database integrity; a distributed database system must be able to handle concurrency, deadlocks and recovery.

Hardware independence

A distributed database system should be able to operate and access data spread across a wide variety of hardware platforms

A truly distributed DBMS system should not rely on a particular hardware feature, nor should it be limited to a certain hardware architecture.

Operating system independence

A distributed database system should be able to run on different operating systems

Network independence

A distributed database system should be designed to run regardless of the communication protocols and network topology used to interconnect sites

DBMS independence

An *ideal* distributed database system must be able to support interoperability between DBMS systems running on different nodes, even if these DBMS systems are unalike

All sites in a distributed database system should use common standard interfaces in order to interoperate with each other.

Distributed Databases vs. Parallel Databases

Distributed Databases

- Local autonomy
- No central site
- Continuous operation
- Location independence
- Fragmentation independence
- Replication independence
- Distributed query processing
- Distributed transactions
- Hardware independence
- Operating system independence
- Network independence
- DBMS independence

Distributed Databases vs. Parallel Databases

Parallel Databases

- ~~Local autonomy~~
- No central site
- Continuous operation
- Location independence
- Fragmentation independence
- Replication independence
- Distributed query processing
- Distributed transactions
- ~~Hardware independence~~
- ~~Operating system independence~~
- ~~Network independence~~
- ~~DBMS independence~~

Fragmentation

Why Fragment?

Fragmentation allows:

- localisation of the accesses of relations by applications
- parallel execution (increases concurrency and throughput)

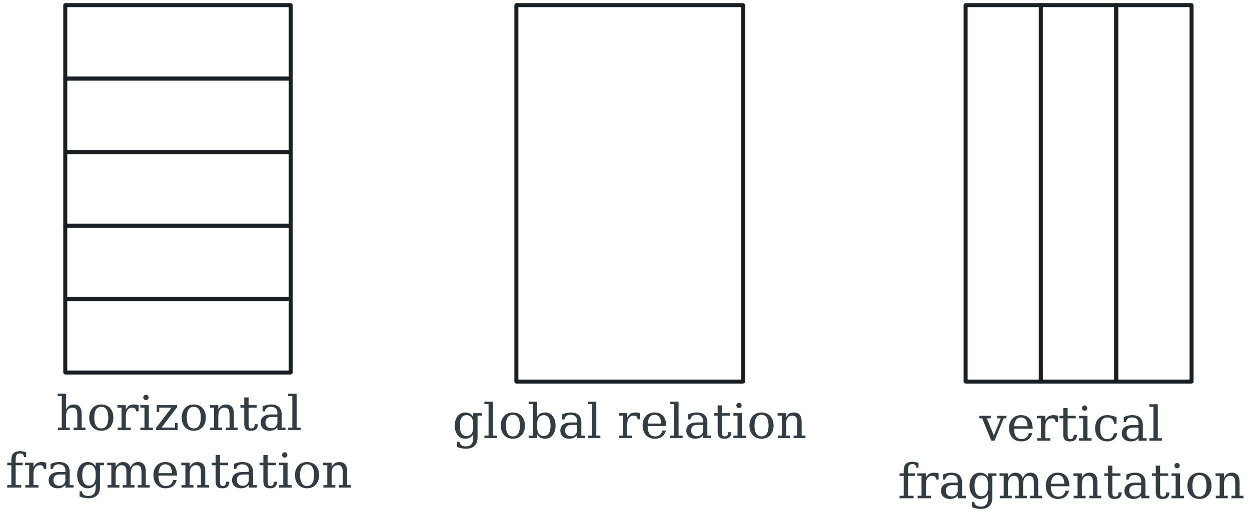
Fragmentation Approaches

Horizontal fragmentation

Each fragment contains a subset of the tuples of the global relation

Vertical fragmentation

Each fragment contains a subset of the attributes of the global relation



Decomposition

Relation R is *decomposed* into fragments $F_R = \{R_1, R_2, \ldots, R_n\}$

Decomposition (horizontal or vertical) can be expressed in terms of relational algebra expressions

Completeness

F_R is *complete* if each data item d_i in R is found in some R_j

Reconstruction

R can be *reconstructed* if it is possible to define a relational operator $\triangledown$ such that $R = \triangledown R_i$, for all $R_i \in F_R$

Note that $\triangledown$ will be different for different types of fragmentation

Disjointness

F_R is *disjoint* if every data item d_i in each R_j is not in any R_k
where $k \neq j$

Note that this is only strictly true for horizontal decomposition

For vertical decomposition, primary key attributes are typically repeated in all fragments to allow reconstruction; disjointness is defined on non-primary key attributes

Horizontal Fragmentation

Each fragment contains a subset of the tuples of the global relation

Two versions:

- *Primary horizontal fragmentation*
  performed using a predicate defined on the relation being partitioned
- *Derived horizontal fragmentation*
  performed using a predicate defined on another relation

Primary Horizontal Fragmentation

Decomposition

$F_R = \{ R_i : R_i = \sigma_{fi}(R) \}$

where f_i is the *fragmentation predicate* for R_i

Reconstruction

$R = \cup R_i$ for all $R_i \in F_R$

Disjointness

F_R is disjoint if the simple predicates used in f_i are mutually exclusive

Completeness for primary horizontal fragmentation is beyond the scope of this lecture...

Derived Horizontal Fragmentation

Decomposition

$F_R = \{ R_i : R_i = R \triangleright S_i \}$

where $F_S = \{S_i : S_i = \sigma_{fi}(S) \}$
and f_i is the fragmentation predicate for the primary horizontal fragmentation of S

Reconstruction

$R = \cup R_i$ for all $R_i \in F_R$

Completeness and disjointness for derived horizontal fragmentation is beyond the scope of this lecture…

Vertical Fragmentation

Decomposition

$F_R = \{ R_i : R_i = \pi_{ai}(R) \}$, where a_i is a subset of the attributes of R

Completeness

F_R is complete if each attribute of R appears in some a_i

Reconstruction

$R = \bowtie_K R_i$ for all $R_i \in F_R$

where K is the set of primary key attributes of R

Disjointness

F_R is disjoint if each non-primary key attribute of R appears in at most one a

Hybrid Fragmentation

Horizontal and vertical fragmentation may be combined

- Vertical fragmentation of horizontal fragments
- Horizontal fragmentation of vertical fragments

Query Processing

Localisation

Fragmentation expressed as relational algebra expressions

Global relations can be reconstructed using these expressions

- a *localisation program*

Naively, generate distributed query plan by substituting localisation programs for relations

- use reduction techniques to optimise queries

Reduction for Horizontal Fragmentation

Given a relation R fragmented as $F_R = \{R_1, R_2, \ldots, R_n\}$

Localisation program is $R = R_1 \cup R_2 \cup \ldots \cup R_n$

Reduce by identifying fragments of localised query that give empty relations

Two cases to consider:

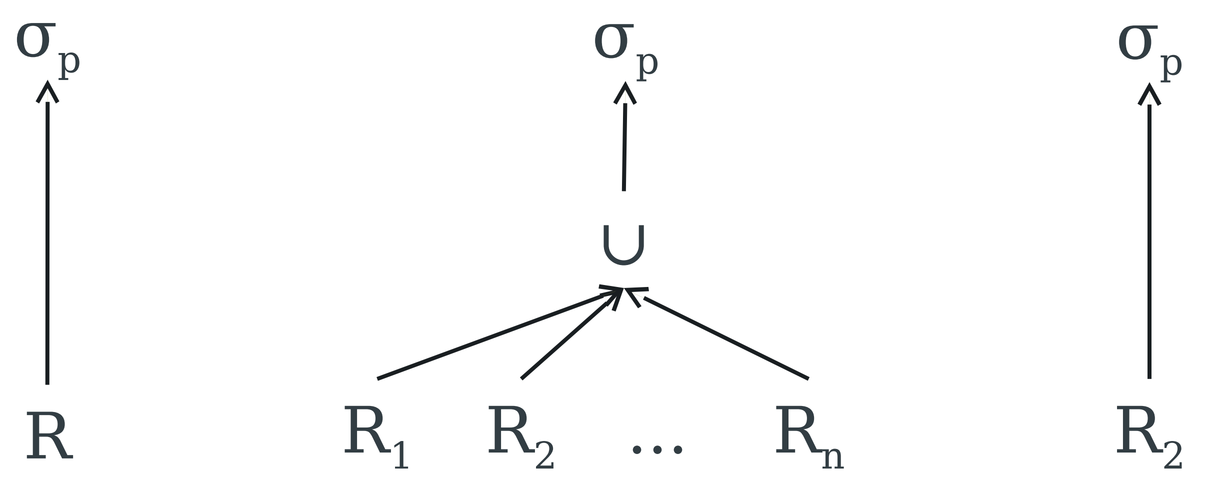
- reduction with selection
- reduction with join

Horizontal Selection Reduction

Given horizontal fragmentation of R such that $R_j = \sigma_{pj}(R)$:

$\sigma_p(R_j) = \varnothing$ if $\forall x \in R, \neg(p(x) \wedge p_j(x))$

where p_j is the fragmentation predicate for R_j



query

localised query

reduced query

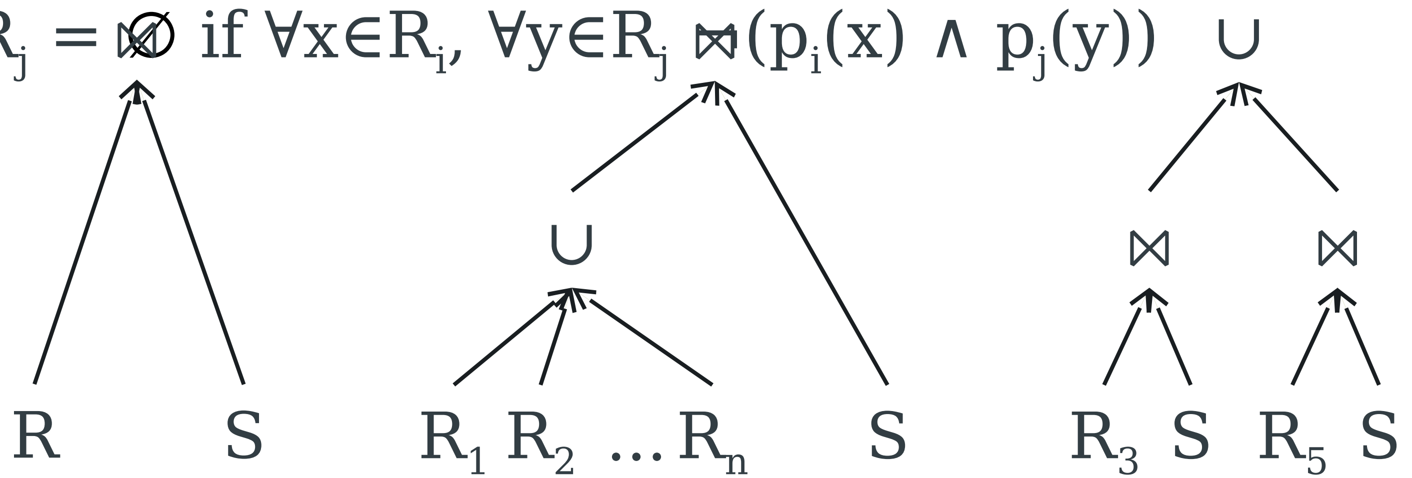
Horizontal Join Reduction

Recall that joins distribute over unions:

$$(R_1 \cup R_2) \bowtie S \equiv (R_1 \bowtie S) \cup (R_2 \bowtie S)$$

Given fragments R_i and R_j defined with predicates p_i and p_j :

$$R_i \bowtie R_j = \emptyset \text{ if } \forall x \in R_i, \forall y \in R_j\ \neg(p_i(x) \wedge p_j(y))$$



query

localised query

reduced query

Reduction for Vertical Fragmentation

Given a relation R fragmented as $F_R = \{R_1, R_2, \ldots, R_n\}$

Localisation program is $R = R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$

Reduce by identifying useless intermediate relations

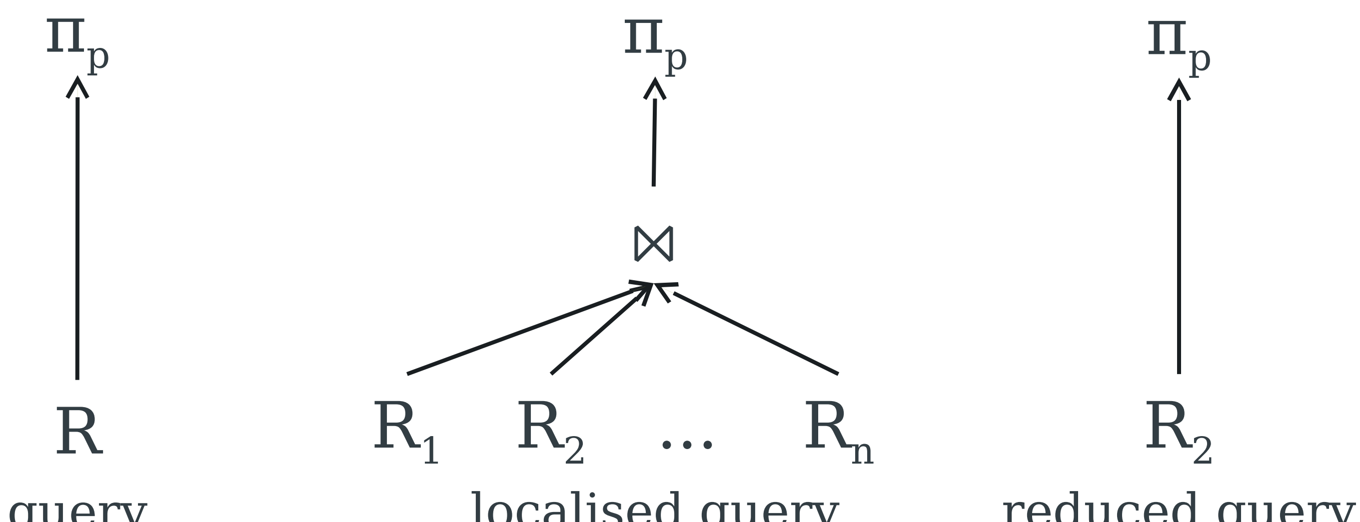
One case to consider:

- reduction with projection

Vertical Projection Reduction

Given a relation R with attributes $A = \{a_1, a_2, \ldots, a_n\}$ vertically fragmented as $R_i = \pi_{A_i}(R)$ where $A_i \subseteq A$

$\pi_{D,K}(R_i)$ is useless if $D \not\subseteq A_i$



The Distributed Join Problem

We have two relations, R and S, each stored at a different site

Where do we perform the join $R \bowtie S$?

The Distributed Join Problem

We can move one relation to the other site and perform the join there

- CPU cost of performing the join is the same regardless of site
- Communications cost depends on the size of the relation being moved

The Distributed Join Problem

$Cost_{COM} = size(R) = cardinality(R) * length(R)$

if size(R) < size(S)   then move R to site 2,
otherwise move S to site 1

Semijoin Reduction

We can further reduce the communications cost by only moving that part of a relation that will be used in the join

Use a semijoin…

Semijoins

Recall that $R \rhd_p S \equiv \pi_R(R \bowtie_p S)$

where p is a predicate defined over R and S

π_R projects out only those attributes from R

$\text{size}(R \rhd_p S) < \text{size}(R \bowtie_p S)$

$$\begin{aligned} R \bowtie_p S &\equiv (R \rhd_p S) \bowtie_p S \\ &\equiv R \bowtie_p (R \lhd_p S) \\ &\equiv (R \rhd_p S) \bowtie_p (R \lhd_p S) \end{aligned}$$

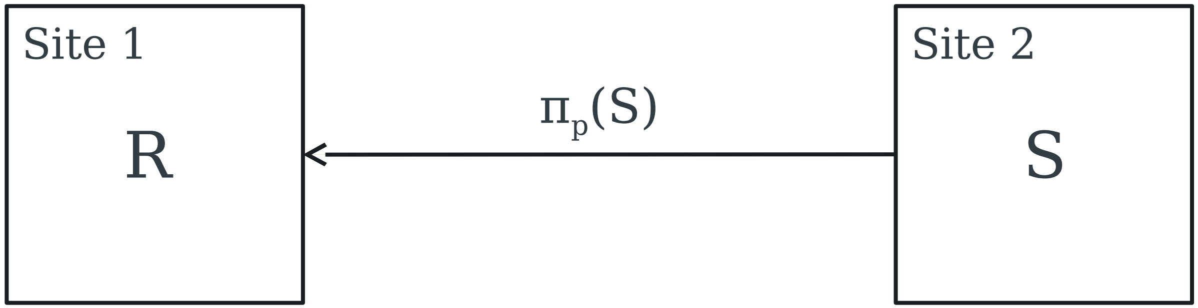
Semijoin Reduction

$$R \triangleright_p S \equiv \pi_R(R \bowtie_p S)$$
$$\equiv \pi_R(R \bowtie_p \pi_p(S))$$

where $\pi_p(S)$ projects out from S only the attributes used in predicate p

Semijoin Reduction, step 1

Site 2 sends $\pi_p(S)$ to site 1



Semijoin Reduction, step 2

Site 1 calculates $R \triangleright_p S \equiv \pi_R(R \bowtie_p \pi_p(S))$

Site 1

R

$R \triangleright_p S$

Site 2

S

Semijoin Reduction, step 3

Site 1 sends $R \rhd_p S$ to site 2

Semijoin Reduction, step 4

Site 2 calculates $R \bowtie_p S \equiv (R \triangleright_p S) \bowtie_p S$

Site 1

R

$R \triangleright_p S$

Site 2

S

$R \bowtie_p S$

Semijoin Reduction

$Cost_{COM} = size(\pi_p(S)) + size(R \rhd_p S)$

This approach is better if $size(\pi_p(S)) + size(R \rhd_p S) < size(R)$

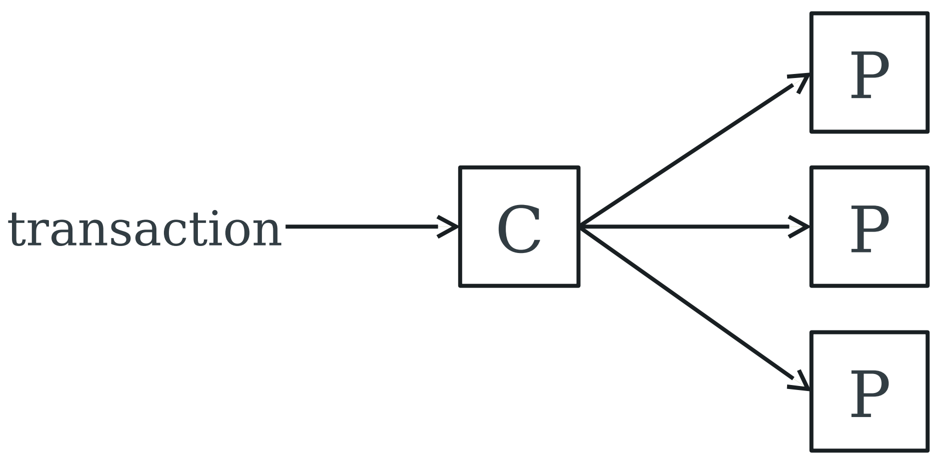
Site 1	Site 2
R	S
$R \rhd_p S$	$R \bowtie_p S$

Concurrency Control

Distributed Transactions

Transaction processing may be spread across several sites in the distributed database

- The site from which the transaction originated is known as the *coordinator*
- The sites on which the transaction is executed are known as the *participants*



Distribution and ACID

Non-distributed databases aim to maintain isolation

- Isolation: A transaction should not make updates externally visible until committed

Distributed databases commonly use two-phase locking (2PL) to preserve isolation

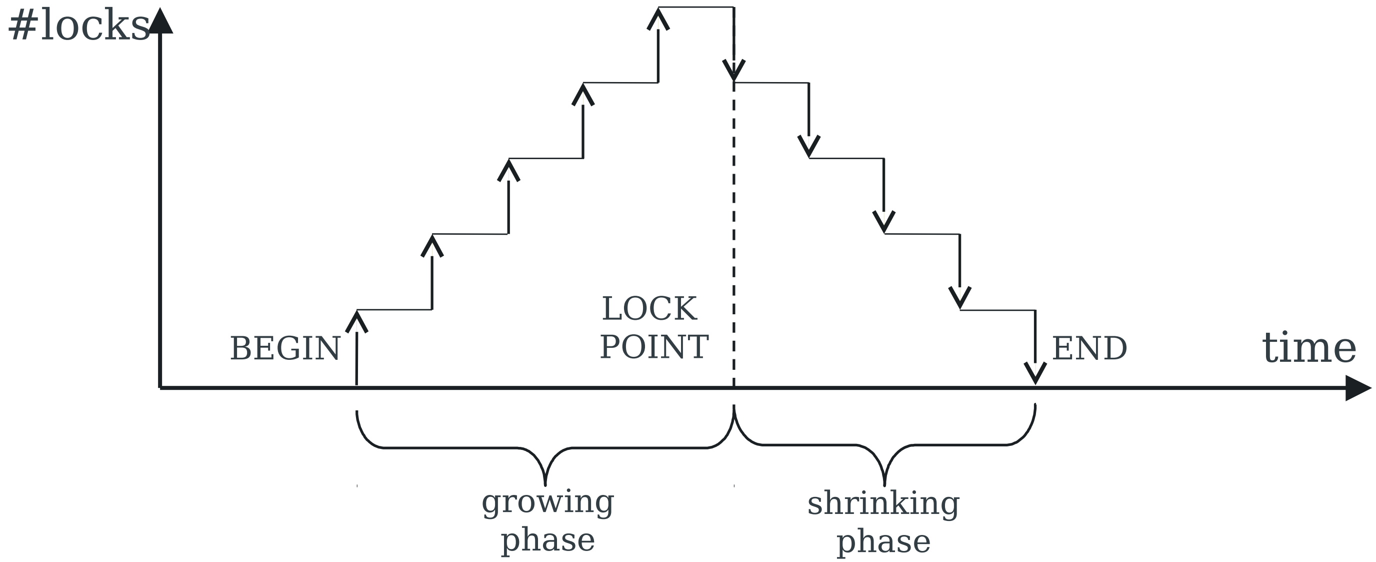
- 2PL ensures serialisability, the highest isolation level

Two-Phase Locking

Two phases:

- Growing phase: obtain locks, access data items
- Shrinking phase: release locks

Guarantees serialisable transactions



Distribution and Two-Phase Locking

In a non-distributed database, locking is controlled by *a lock manager*

Two main approaches to implementing two-phase locking in a distributed database:

- Centralised 2PL (C2PL)
  Responsibility for lock management lies with a single site
- Distributed 2PL (D2PL)
  Each site has its own lock manager

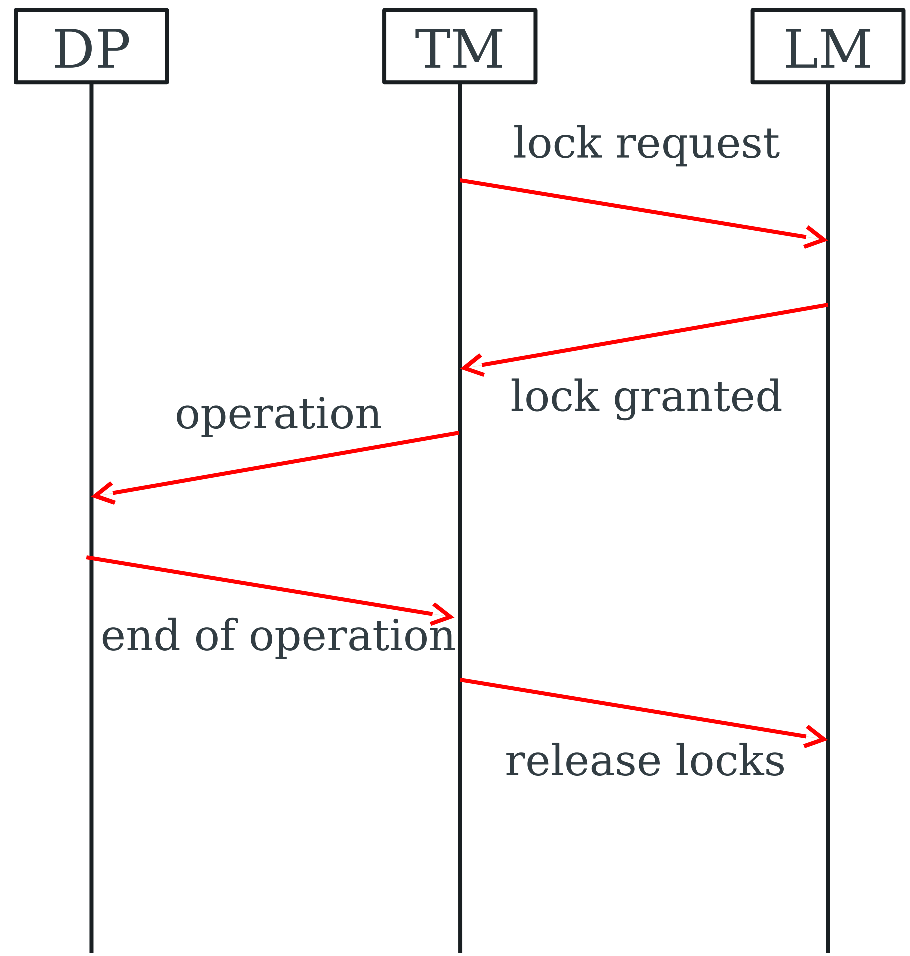
Centralised Two-Phase Locking (C2PL)

Coordinating site runs *transaction manager* TM

Participant sites run *data processors* DP

Lock manager LM runs on central site

1. TM requests locks from LM
2. If granted, TM submits operations to processors DP
3. When DPs finish, TM sends message to LM to release



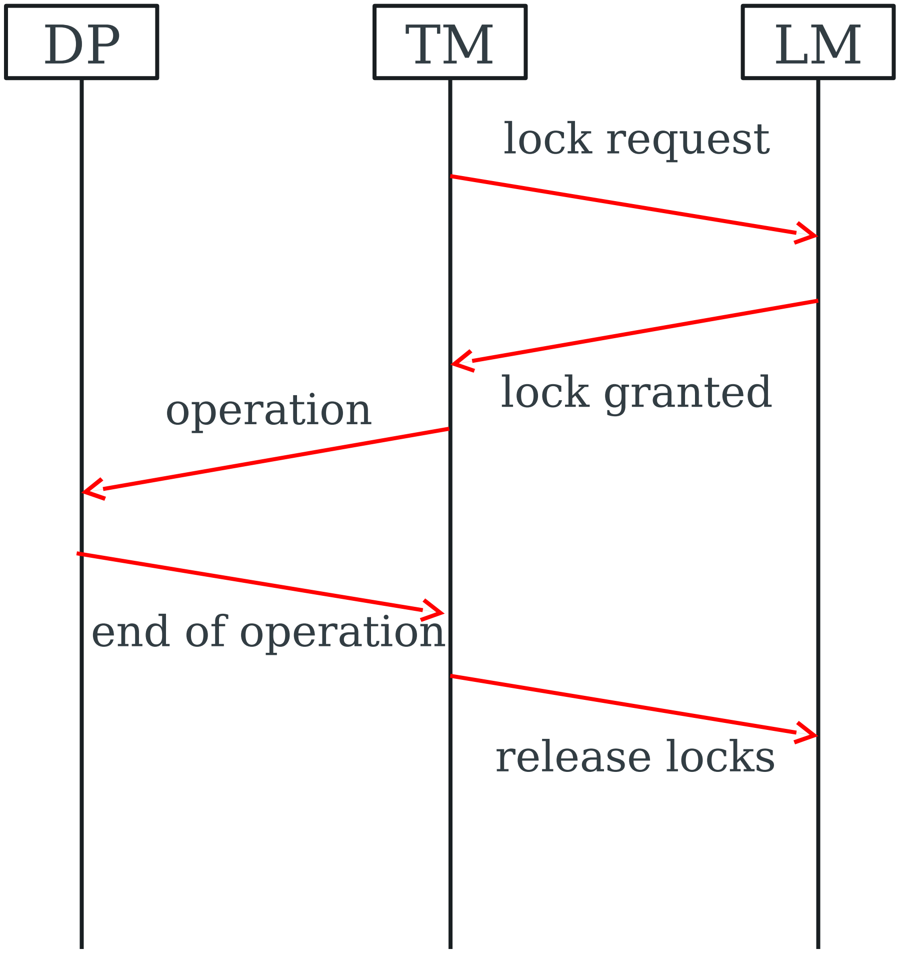
Centralised Two-Phase Locking (C2PL)

LM is a single point of failure

- less reliable

LM is a bottleneck

- affects transaction throughput

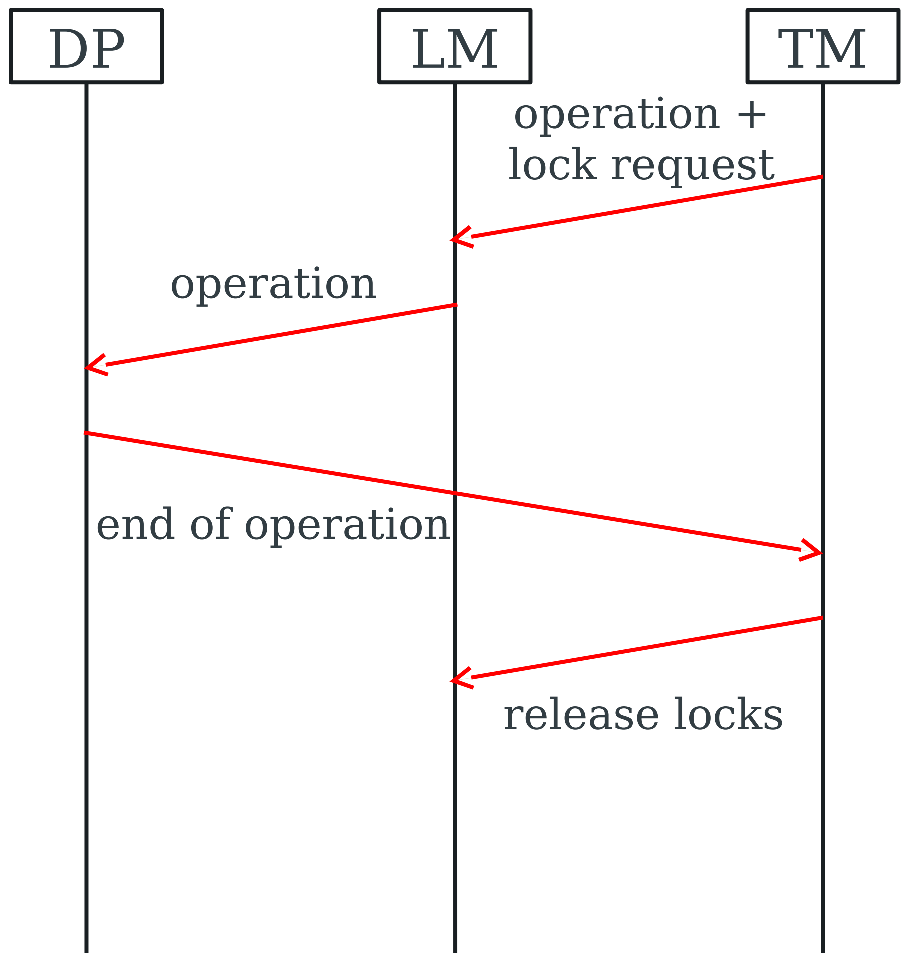


Distributed Two-Phase Locking (D2PL)

Coordinating site C runs TM

Each participant runs both an LM and a DP

1. TM sends operations and lock requests to each LM
2. If lock can be granted, LM forwards operation to local DP
3. DP sends “end of operation” to TM
4. TM sends message to LM to release locks

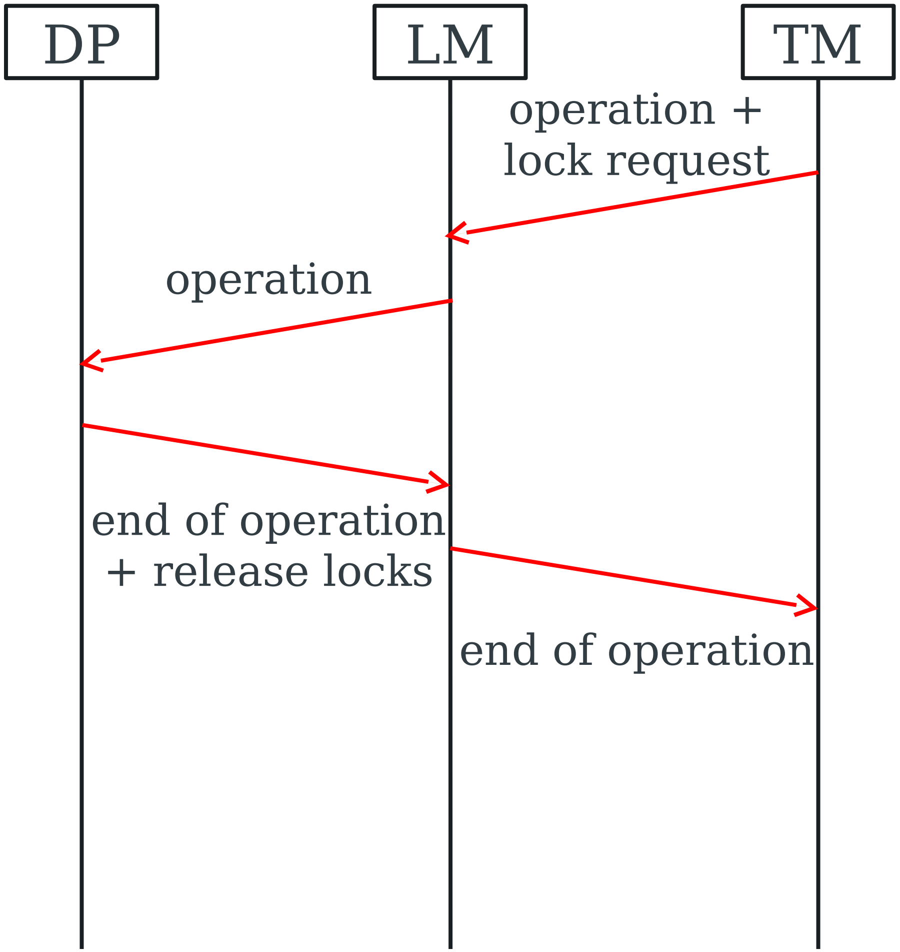


Distributed Two-Phase Locking (D2PL)

Variant:

DPs may send “end of operation” to their own LM

LM releases lock and informs TM



Deadlock

Deadlock exists when two or more transactions are waiting for each other to release a lock on an item

Three conditions must be satisfied for deadlock to occur:

- Concurrency: two transactions claim exclusive control of one resource
- Hold: one transaction continues to hold exclusively controlled resources until its need is satisfied
- Wait: transactions wait in queues for additional resources while holding resources already allocated

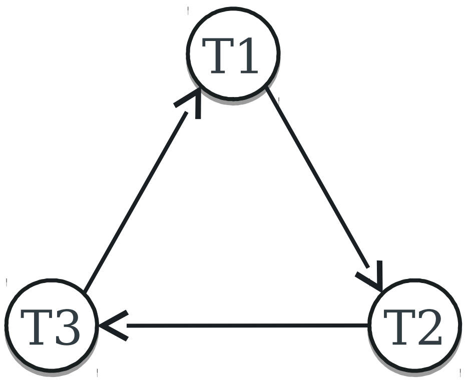
Wait-For Graph

Representation of interactions between transactions

Directed graph containing:

- A vertex for each transaction that is currently executing
- An edge from T1 to T2 if T1 is waiting to lock an item that is currently locked by T2

Deadlock exists iff the WFG contains a cycle



Distributed Deadlock

Two types of Wait-For Graph

- Local WFG
  (one per site, only considers transactions on that site)
- Global WFG
  (union of all LWFGs)

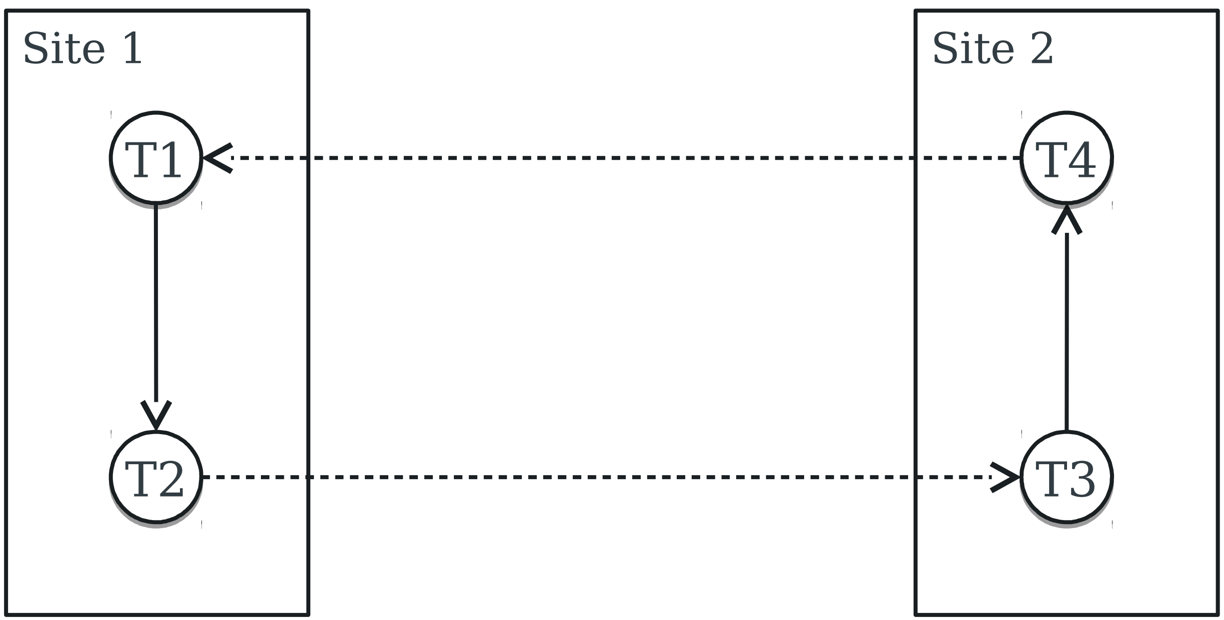
Deadlock may occur

- on a single site
  (within its LWFG)
- between sites
  (within the GWFG)

Distributed Deadlock Example

Consider the wait-for relationship T1→T2→T3→T4→T1

with T1, T2 on site 1 and T3, T4 on site 2



Managing Distributed Deadlock

Three main approaches:

1. Prevention
   - pre-declaration
2. Avoidance
   - resource ordering
   - transaction prioritisation
3. Detection and Resolution

Prevention

Guarantees that deadlocks cannot occur in the first place

1. Transaction pre-declares all data items that it will access
2. TM checks that locking data items will not cause deadlock
3. Proceed (to lock) only if all data items are available (unlocked)

Avoidance

Two main sub-approaches:

1. Resource ordering
   - Concurrency controlled such that deadlocks won’t happen
2. Transaction prioritisation
   - Potential deadlocks detected and avoided

Resource Ordering

All resources (data items) are ordered

Transactions always access resources in this order

Example:

- Data item A comes before item B
- All transactions must get a lock on A before trying for a lock on B
- No transaction will ever be left with a lock on B and waiting for a lock on A

Transaction Prioritisation

Each transaction has a timestamp that corresponds to the time it was started: ts(T)

- Transactions can be prioritised using these timestamps

When a lock request is denied, use priorities to choose a transaction to abort

- WAIT-DIE and WOUND-WAIT rules

WAIT-DIE and WOUND-WAIT

T_i requests a lock on a data item that is already locked by T_j

The WAIT-DIE rule:

if $ts(T_i) < ts(T_j)$
then T_i waits
else T_i dies (aborts and restarts with same timestamp)

The WOUND-WAIT rule:

if $ts(T_i) < ts(T_j)$
then T_j is wounded (aborts and restarts with same timestamp)
else T_i waits

note: WOUND-WAIT pre-empts active transactions

Detection and Resolution

1. Study the GWFG for cycles (detection)
2. Break cycles by aborting transactions (resolution)

Selecting minimum total cost sets of transactions to abort is NP-complete

Three main approaches to deadlock detection:

- centralised
- hierarchical
- distributed

Centralised Deadlock Detection

One site is designated as the deadlock detector (DD) for the system

Each site sends its LWFG (or changes to its LWFG) to the DD at intervals

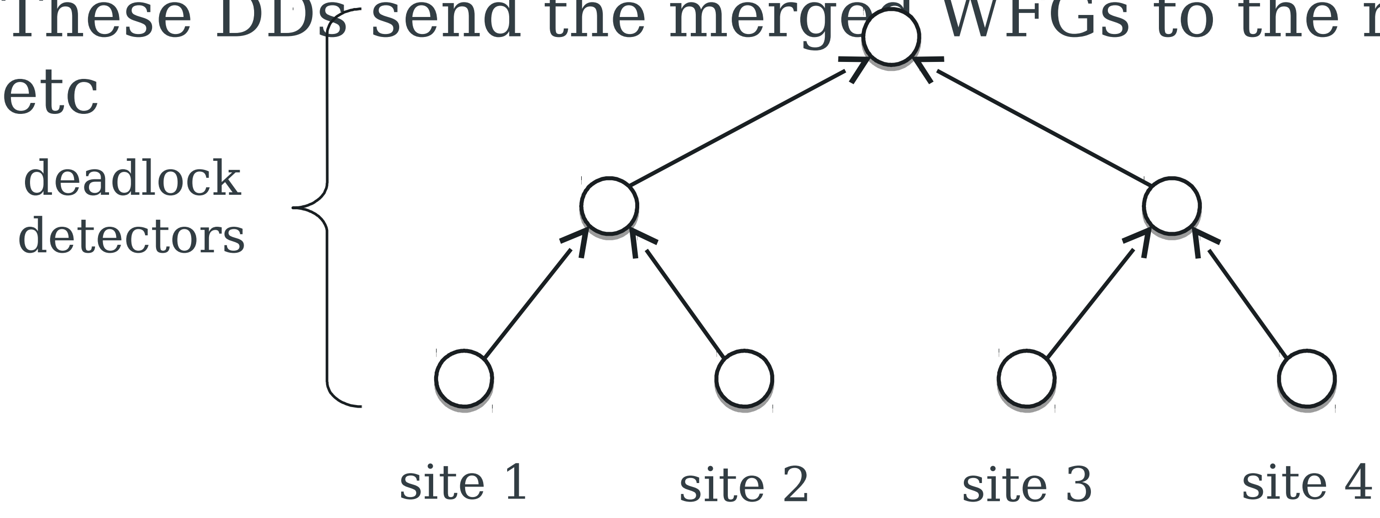
DD constructs the GWFG and looks for cycles

Hierarchical Deadlock Detection

Each site has a DD, which looks in the site's LWFG for cycles

Each site sends its LWFG to the DD at the next level, which merges the LWFGs sent to it and looks for cycles

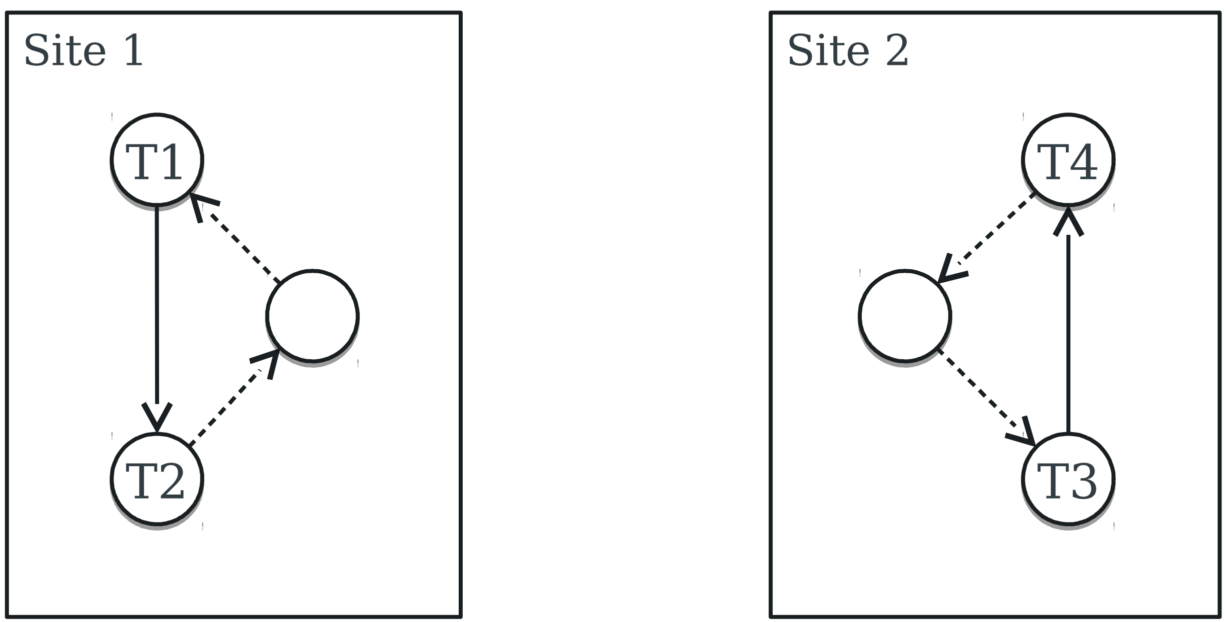
These DDs send the merged WFGs to the next level, etc



Distributed Deadlock Detection

Responsibility for detecting deadlocks is delegated to sites

LWFGs are modified to show relationships between local transactions and remote transactions



Distributed Deadlock Detection

LWFG contains a cycle not involving external edges

- Local deadlock, resolve locally

LWFG contains a cycle involving external edges

- Potential deadlock – communicate to other sites
- Sites must then agree on a victim transaction to abort

Reliability

Distribution and ACID

Non-distributed databases aim to maintain atomicity and durability of transactions

- Atomicity: A transaction is either performed completely or not at all
- Durability: Once a transaction has been committed, changes should not be lost because of failure

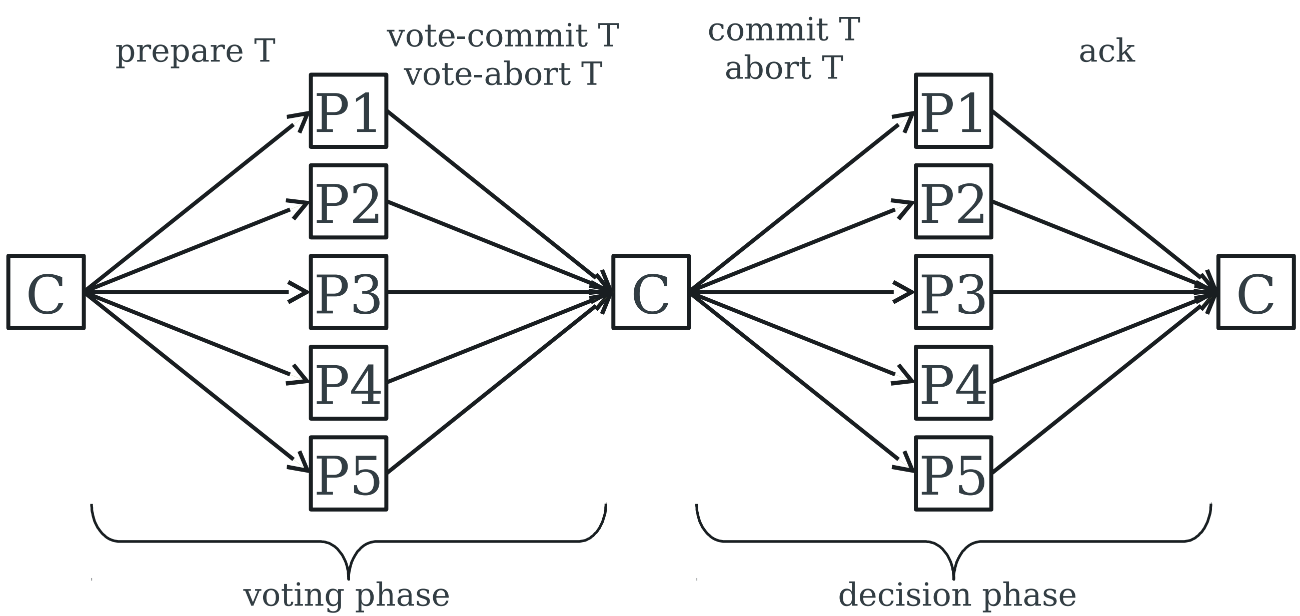
As with parallel databases, distributed databases use the two-phase commit protocol (2PC) to preserve atomicity

- Increased cost of communication may require a variant approach

Centralised 2PC

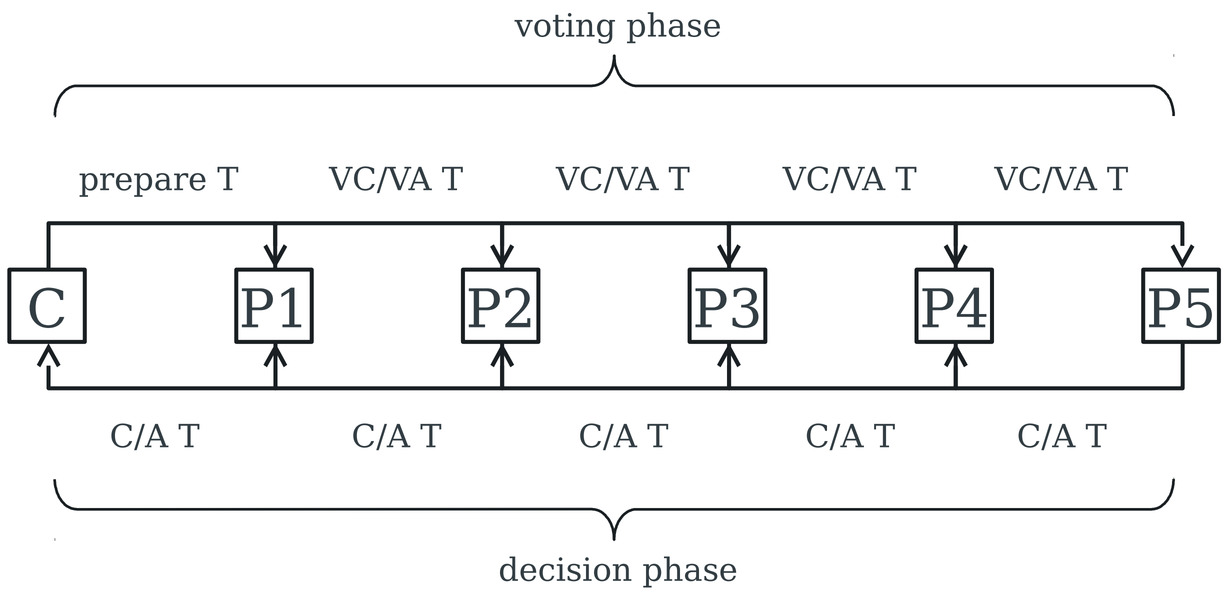
Communication only between the coordinator and the participants

- No inter-participant communication



Linear 2PC

- First phase from the coordinator to the participants
- Second phase from the participants to the coordinator
- Participants may unilaterally abort



Centralised versus Linear 2PC

- Linear 2PC involves fewer messages
- Centralised 2PC provides opportunities for parallelism
- Linear 2PC has worse response time performance

The CAP Theorem

The CAP Theorem

In any distributed system, there is a trade-off between:

- Consistency

  Each server always returns the correct response to each request

- Availability

  Each request eventually receives a response

- Partition Tolerance

  Communication may be unreliable (messages delayed, messages lost, servers partitioned into groups that cannot communicate with each other), but the system as a whole should continue to function

The CAP Theorem

CAP is an example of the trade-off between safety and liveness in an unreliable system

- Safety: nothing bad ever happens
- Liveness: eventually something good happens

We can only manage two of three from C, A, P

- Typically we sacrifice either availability (liveness) or consistency (safety)