南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

DBGroup

Distributed Databases I

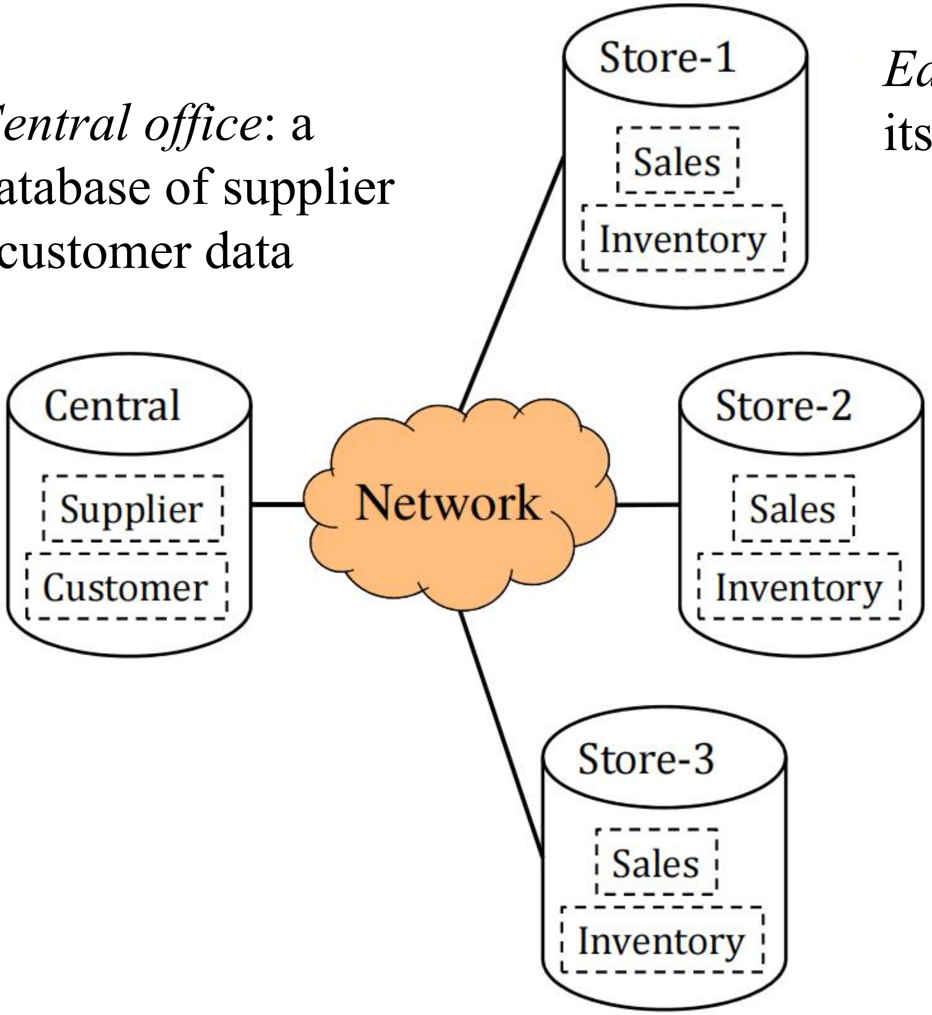
南方科技大学
唐　博
tangb3@sustech.edu.cn

Lecture Objectives

- An overview of the distributed database
- Data replication and fragmentation
- The two-phase commit protocol

Supermarket Chain

Central office: a database of supplier / customer data



Each store: a database of its sales and inventory

Global "inventory" table: the union of all inventory tables at stores

Global transaction: may access local databases (e.g., count the total stock of "coca cola")

What is a distributed database?

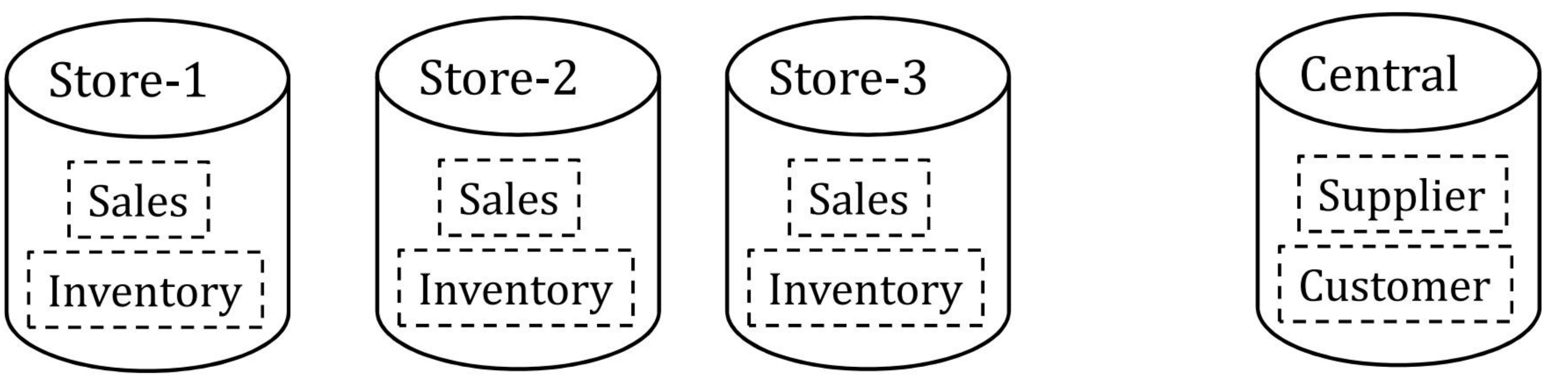
- A collection of data with
  - *Distribution*: data are spread over different sites (of a network)
  - *Logical correlation*: data belong to the same system; some properties tie them together

Support *global transactions*:
accesses data at more than one site

Why distributed databases?

Management perspectives

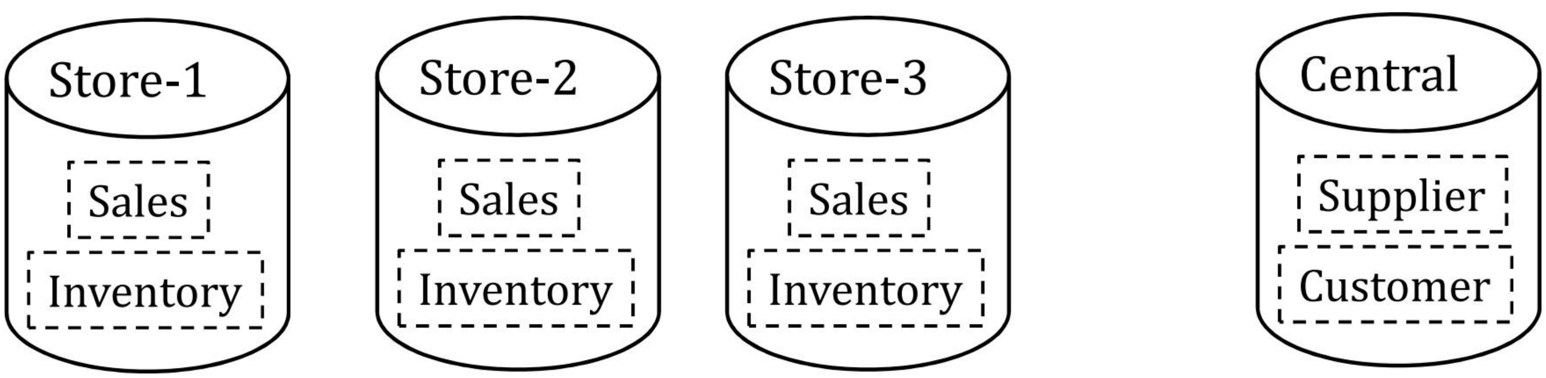
- *Organizational requirement*: each division / branch (of the organization) may want to maintain its own DB
- *Interconnect existing DB's*: when multiple DBs already exist in an organization & need for global applications
- *Incremental growth*: support smooth incremental growth (e.g., adding a branch) with small impact on existing DBs



Why distributed databases?

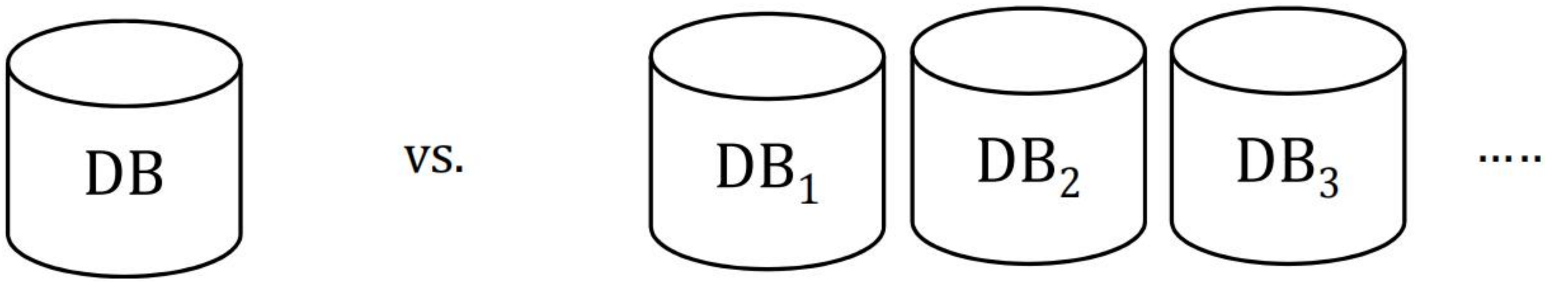
Technical perspectives

- *Reduced communication overhead*: run sub-transactions at different sites ➡ sites transfer intermediate results (small) rather than entire tables
- *Parallel executions*: can execute some transactions in parallel at the participating sites
- *Reliability and availability*: can still run transactions despite failures of some sites



Recovery Manager (RM)

- Centralized DB vs. Distributed DB



- Revisit DBMS issues, how to:
  - Store data? *[study today]*
    - By fragmentation, replication
- Ensure ACID properties?
  - Recovery for A, D *[study today]*
    - By the two-phase commit protocol
  - Concurrency for I *[next lecture]*
- Process a query fast? *[next lecture]*

Lecture Objectives

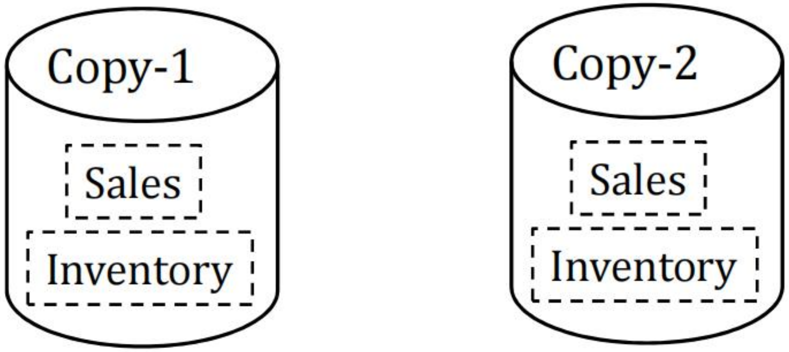
- An overview of the distributed database
- Data replication and fragmentation
- The two-phase commit protocol

Distributed Data Storage

- Assume relational data model
- *Fragmentation*
  - Relation is partitioned into several fragments stored at different sites
- *Replication*
  - System stores multiple copies of data at different sites
  - For faster retrieval and fault tolerance
- Replication and fragmentation can be combined
  - Relation is partitioned into several fragments
  - System stores several identical copies of each such fragment

Data Replication

- A relation (or fragment of a relation) is **replicated** if it is stored redundantly in two or more sites
- How to process queries?
  - Query either copy, OR
  - Query in parallel
- How to process updates?
  - Must update both copies?
    - We'll discuss more about this in the next lecture



Data Replication (Cont.)

- Advantages of Replication
  - **Availability**: even when a site has failure, we can access copies at other sites
  - **Parallelism**: queries on *r* may be processed by several nodes in parallel
  - **Reduced data transfer**: relation *r* is available locally at each site containing a replica of *r*

Data Replication (Cont.)

- Disadvantages of Replication
  - Expensive **updates**: each replica of relation r must be updated
  - **Complex concurrency control**: updates to different replicas may cause inconsistent data
    - Need a concurrency control protocol for distributed DBs: E.g., choose one copy as **primary copy** and apply concurrency control operations on primary copy

Data Fragmentation

- Divide relation r into fragments $r_1, r_2, \ldots, r_n$ which contain ***sufficient*** information to reconstruct relation r

Example : relation account with schema

Account = (branch_name, customer_number, account_number, balance)

branch_name	*customer_name*	*account_number*	*balance*
Hillside	Lowman	A-305	500
Hillside	Camp	A-226	336
Valleyview	Camp	A-177	205
Valleyview	Kahn	A-402	10000
Hillside	Kahn	A-155	62
Valleyview	Kahn	A-408	1123
Valleyview	Green	A-639	750

Partition by Rows

Assign each tuple of *r* to one fragment

branch_name	*customer_name*	*account_number*	*balance*
Hillside	Lowman	A-305	500
Hillside	Camp	A-226	336
Hillside	Kahn	A-155	62

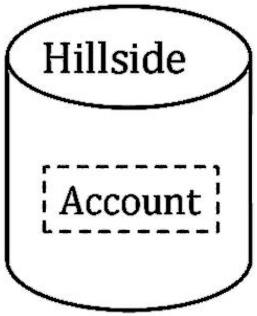
$$account_1 = \sigma_{branch\_name=\text{“Hillside”}} (account)$$

branch_name	*customer_name*	*account_number*	*balance*
Valleyview	Camp	A-177	205
Valleyview	Kahn	A-402	10000
Valleyview	Kahn	A-408	1123
Valleyview	Green	A-639	750

$$account_2 = \sigma_{branch\_name=\text{“Valleyview”}} (account)$$

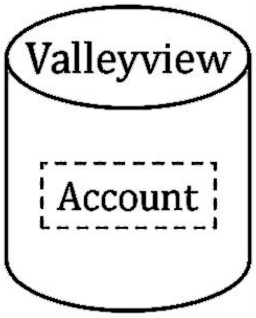
Horizontal Fragmentation

branch_name	*customer_name*	*account_number*	*balance*
Hillside	Lowman	A-305	500
Hillside	Camp	A-226	336
Hillside	Kahn	A-155	62



$$account_1 = \sigma_{branch\_name = \text{"Hillside"}}(account)$$

branch_name	*customer_name*	*account_number*	*balance*
Valleyview	Camp	A-177	205
Valleyview	Kahn	A-402	10000
Valleyview	Kahn	A-408	1123
Valleyview	Green	A-639	750



$$account_2 = \sigma_{branch\_name = \text{"Valleyview"}}(account)$$

How to find out the sum of balance efficiently?

Partition by Columns

branch_name	*customer_name*
Hillside	Lowman
Hillside	Camp
Valleyview	Camp
Valleyview	Kahn
Hillside	Kahn
Valleyview	Kahn
Valleyview	Green

$deposit_1 = \Pi_{branch\_name,\ customer\_name}(account)$

account_number	*balance*
A-305	500
A-226	336
A-177	205
A-402	10000
A-155	62
A-408	1123
A-639	750

$deposit_2 = \Pi_{account\_number,\ balance}(account)$

Account = (*branch_name*, *customer_number*, *account_number*, *balance*)

Do we have ***sufficient*** information to reconstruct the original table?

Vertical Fragmentation

- **Vertical fragmentation**: split the schema for relation *r* into several smaller schemas
  - All schemas must contain a common candidate key (or superkey) to ensure lossless join property
  - May need to add a special attribute (tuple-id) to each schema as a candidate key

$deposit_1 = \Pi_{branch\_name,\ customer\_name,\ tuple\_id}(account)$

branch_name	*customer_name*	*tuple_id*
Hillside	Lowman	1
Hillside	Camp	2
Valleyview	Camp	3
Valleyview	Kahn	4
Hillside	Kahn	5
Valleyview	Kahn	6
Valleyview	Green	7

$deposit_2 = \Pi_{account\_number,\ balance,\ tuple\_id}(account)$

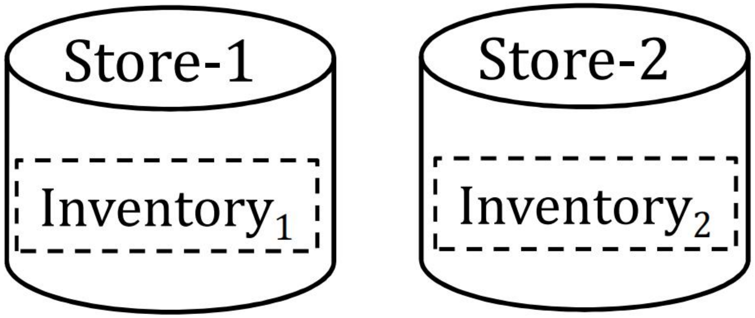
account_number	*balance*	*tuple_id*
A-305	500	1
A-226	336	2
A-177	205	3
A-402	10000	4
A-155	62	5
A-408	1123	6
A-639	750	7

Another Example

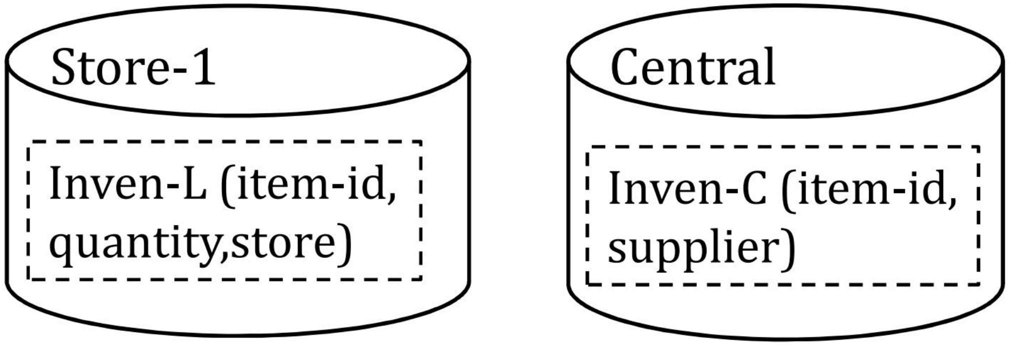
- Example: schema of an inventory relation

  inventory(*item-id, quantity, supplier, store*)

- Horizontal Fragmentation



- Vertical Fragmentation



How to Improve Query Performance?

- Store together the tuples that are frequently accessed together
  - E.g., likely to access tuples at the branch "Hillside" together

branch_name	*customer_name*	*account_number*	*balance*
Hillside	Lowman	A-305	500
Hillside	Camp	A-226	336
Hillside	Kahn	A-155	62

- Store together the attributes that are frequently accessed together
  - E.g., likely to access the attributes account number and balance together

account_number	*balance*	*tuple_id*
A-305	500	1
A-226	336	2
A-177	205	3
……	……	……

How to Improve Query Performance?

- Different transactions may access data with different access patterns
- Difficult to decide the fragmentation manually (by DB administrator)
- Any automatic method for this problem?
  - First, extract access patterns from transactions
  - Then, design the fragmentation accordingly

Case study

Use attribute usage to derive a good vertical fragmentation

Attributes / Transactions	1	2	3	4	5	6	7	8	9	10	Type	Number of accesses per time period
T1	1	0	0	0	1	0	1	0	0	0	R	Acc 1 = 25
T2	0	1	1	0	0	0	0	1	1	0	R	Acc 2 = 50
T3	0	0	0	1	0	1	0	0	0	1	R	Acc 3 = 25
T4	0	1	0	0	0	0	1	1	0	0	R	Acc 4 = 35
T5	1	1	1	0	1	0	1	1	1	0	U	Acc 5 = 25
T6	1	0	0	0	1	0	0	0	0	0	U	Acc 6 = 25
T7	0	0	1	0	0	0	0	0	1	0	U	Acc 7 = 25
T8	0	0	1	1	0	1	0	0	1	1	U	Acc 8 = 15

Fig.1 Attribute usage matrix

Attributes	1	2	3	4	5	6	7	8	9	10
1	75	25	25	0	75	0	50	25	25	0
2	25	110	75	0	25	0	60	110	75	0
3	25	75	115	15	25	15	25	75	115	15
4	0	0	15	40	0	40	0	0	15	40
5	75	25	25	0	75	0	50	25	25	0
6	0	0	15	40	0	40	0	0	15	40
7	50	60	25	0	50	0	85	60	25	0
8	25	110	75	0	25	0	60	110	75	0
9	25	75	115	15	25	15	25	75	115	15
10	0	0	15	40	0	40	0	0	15	40

Fig.2 Attribute affinity (AA) matrix

Example adapted from the paper: "*Vertical Partioninig for Database Design: A Graphical Algorithm*". SIGMOD 1989.

Advantages of Fragmentation

- Horizontal:
  - Allows parallel processing (on fragments with different tuples)
- Vertical:
  - Allows parallel processing (on fragments with different attributes)
  - Tuple-id attribute allows efficient joining of vertical fragments
- Vertical and horizontal fragmentation can be mixed
  - Fragments may be further fragmented to an arbitrary depth

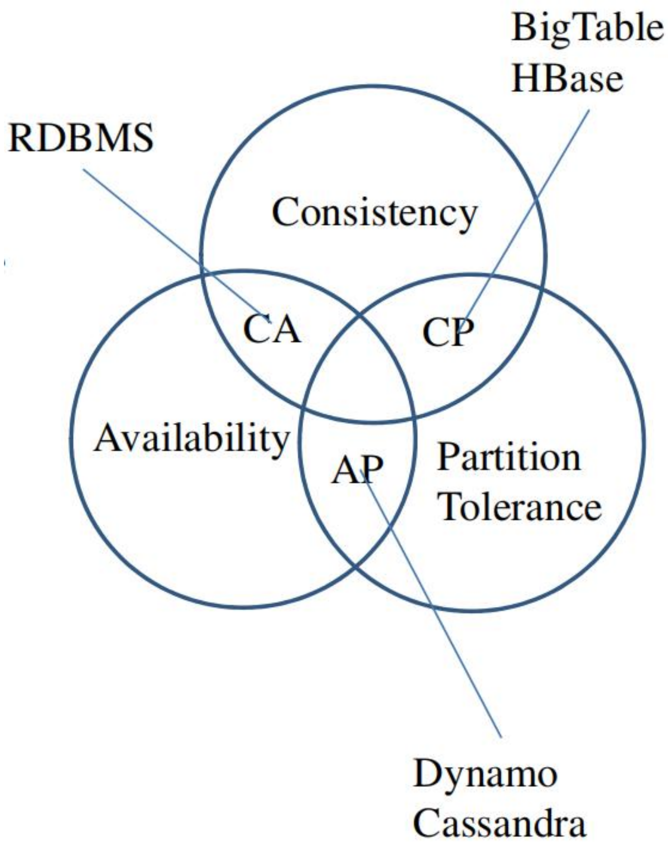
Lecture Objectives

- An overview of the distributed database
- Data replication and fragmentation
- The two-phase commit protocol

  This protocol aims to achieve the properties 'A' and 'D'
  Don't confuse it with the 2PL protocol (for property 'I')

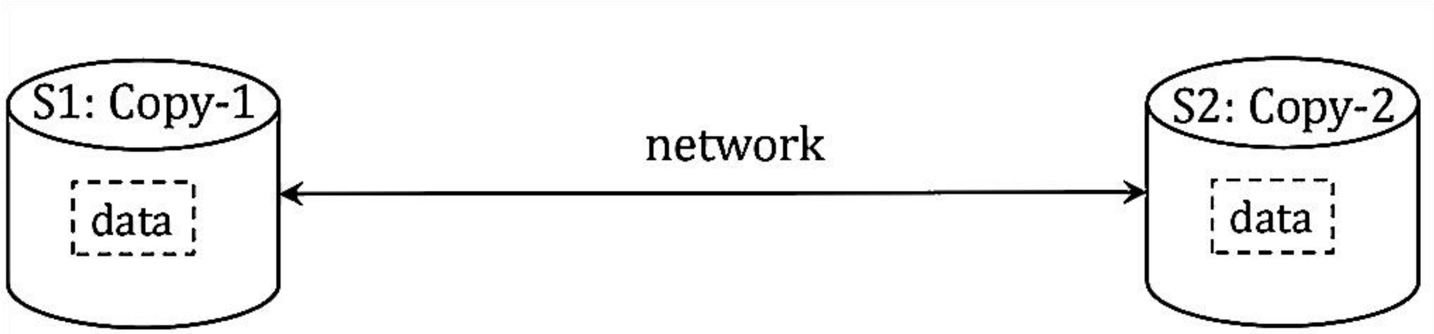
CAP Theorem

- Hard to achieve all three properties together
  - C: Consistency
    - All users can access the up-to-date copy of the data
  - A: Availability
    - The system can work properly even with node failures
  - P: Partitioning tolerance
    - The system can work properly even with network/message failures

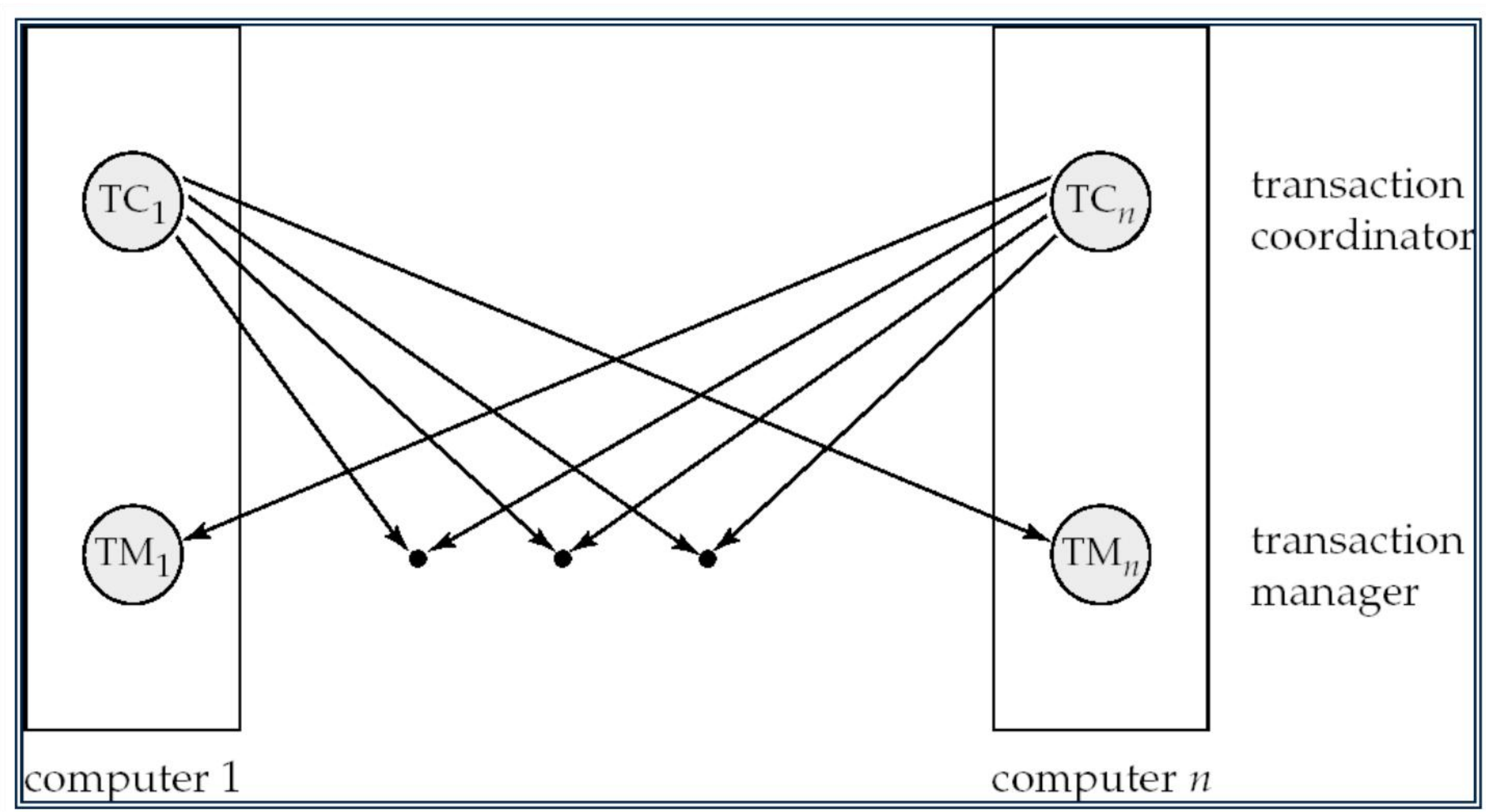


CAP Theorem

- Example: consider replicated data at two sites
  - Consistency: When we update data in S1, need to replicate this update in S2
  - Availability: When a site is running, we can query/update data from it (via network)
  - Partition tolerance: If the network fails, we can still query/update the local site
- When the network fails
  - Allow both sites available ➡ data may not be up-to-date
  - Keep consistency ➡ cannot make both sites available



Transaction System Architecture



Distributed Transactions

- Transaction may access data at several sites
- Each site has a local transaction manager to:
  - Maintain a **log** for recovery purposes
  - Participate in the concurrent execution of transactions at that site
- Each site has a transaction coordinator to:
  - Start the execution of **global transactions** that originate at the site
  - Distribute **sub-transactions** to appropriate sites
  - Coordinate the termination of each transaction that originates at the site, which may result in:

    commit the transaction at all sites / abort …… at all sites

System Failure Modes

- Failures unique to distributed systems:
  - Message loss
    - Handled by network protocols (e.g., TCP-IP)
- Communication link failure
  - Handled by network protocols, by routing messages via alternative links
- Site failure
- **Network partition**
  - It happens when the network is split into subsystems that lack connection
    - Note: a subsystem may consist of a single node

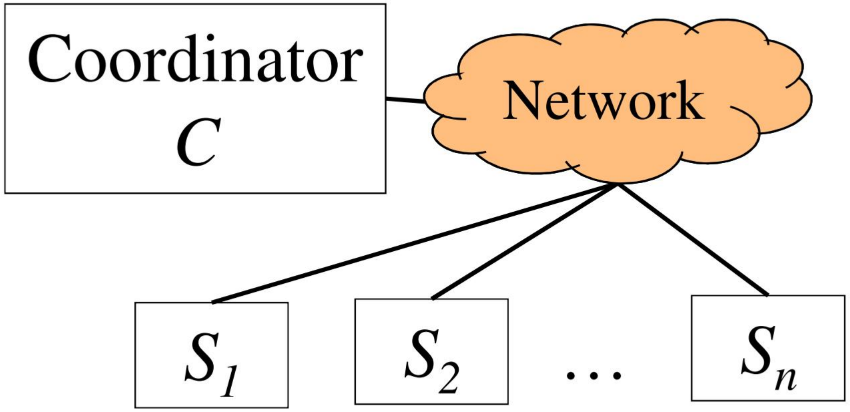
Commit Protocols

- Commit protocols ensure **atomicity** across sites
  - a transaction (which executes at multiple sites) must either
    be committed at all sites, OR
    aborted at all sites.

not acceptable to have a transaction
committed at one site and aborted at another

- The *two-phase commit* (2PC) protocol is widely used
  - It ensures atomicity property despite network / site failures
  - Suppose that each site uses recovery protocol to ensure sub-transaction atomicity

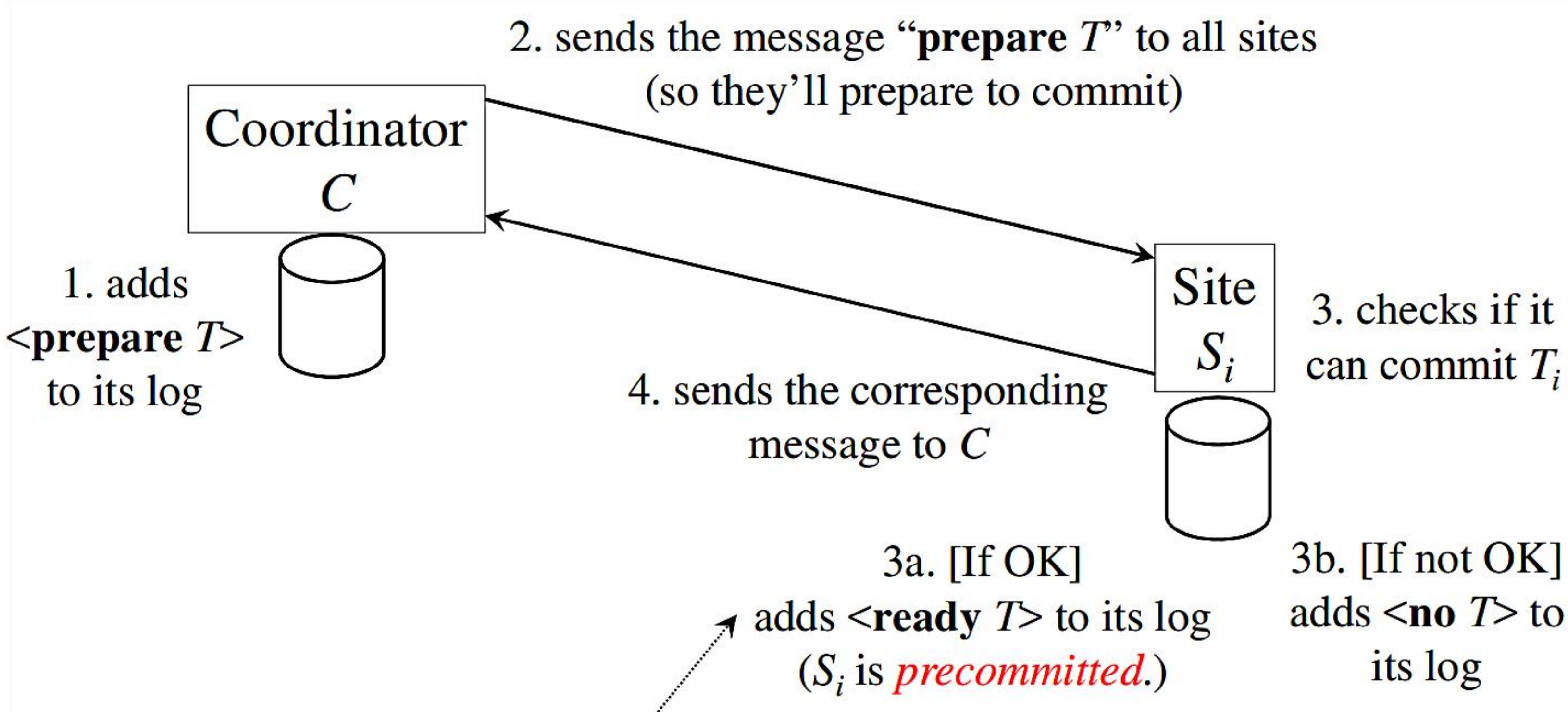
Two-phase commit: Notations



- T: a global transaction
- T_1, T_2, …, Tn: sub-transactions of T
  - Ti will be executed at participant sites S_i
- C: the coordinator of T
  - This site monitors the sub-transactions and decides whether T should commit or not

Phase I: Obtain a Decision

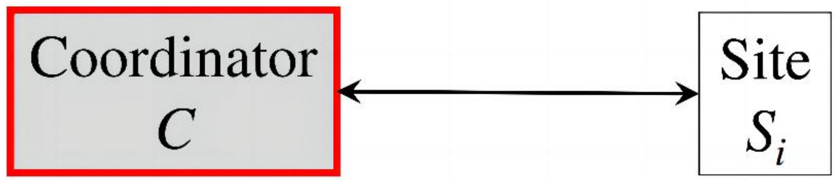
(when all sub-transactions finish)



Q: Why S_i needs to write <ready T> to its stable log?

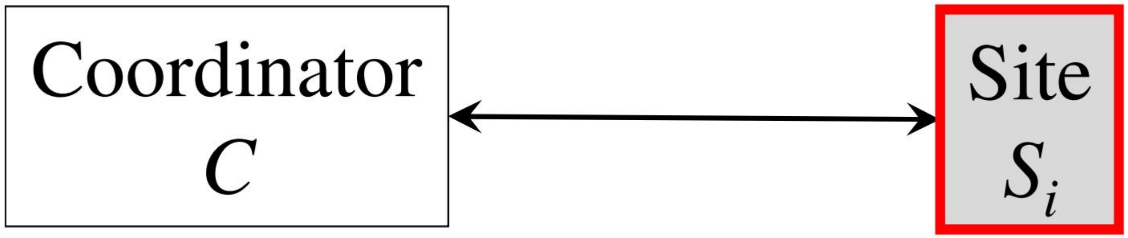
A: If failures occur later (before the protocol finishes), then it can commit/abort T_i

Phase II: Record the Decision



<table>
<tr><th>Cases for the coordinator C</th><th>Actions for C</th></tr>
<tr><td>(1): received a “ready <i>T</i>” from all sites</td><td><u><i>decides to commit T</i></u>
• adds <commit <i>T</i>> to its log
• sends the message “commit <i>T</i>” to all sites</td></tr>
<tr><td>(2): received an “abort <i>T</i>” from some sites</td><td rowspan="2"><u><i>decides to abort T</i></u>
• adds <abort <i>T</i>> to its log
• sends the message “abort <i>T</i>” to all sites</td></tr>
<tr><td>(3): not heard from some site S_i after a certain timeout period [C assumes S_i is down]</td></tr>
</table>

Phase II: Record the Decision

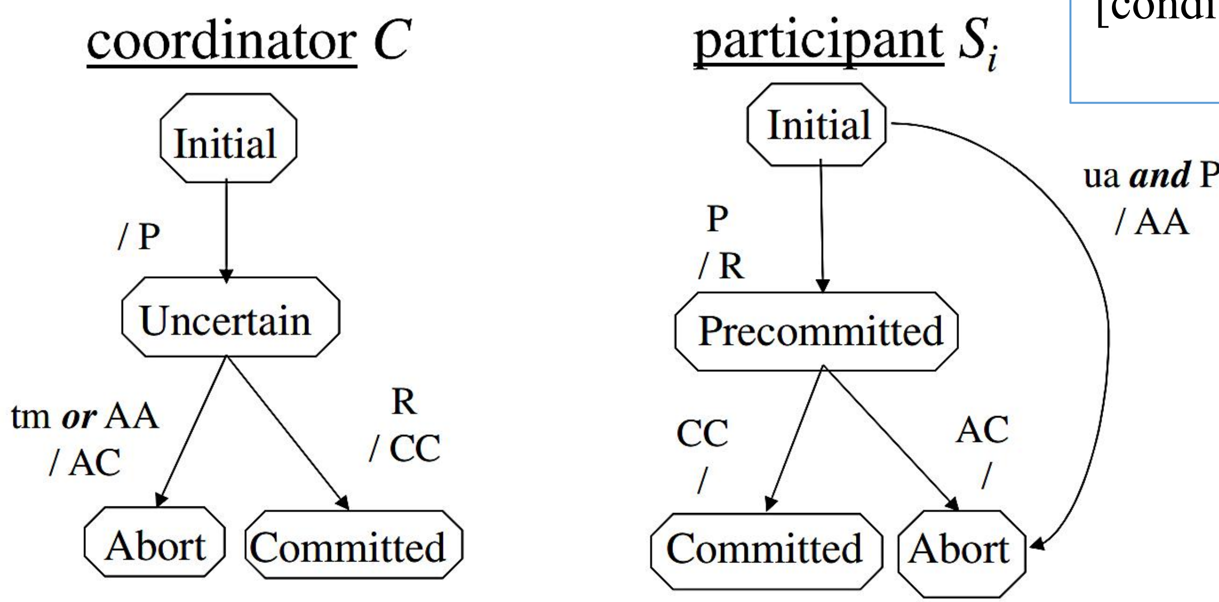


For each participant S_i

- Received "**commit** T":
  - it adds <**commit** T> to its log and commits T_i locally
- Received "**abort** T":
  - it adds <**abort** T> to its log and aborts T_i locally

State Diagram

Format
[condition, message received]
/ [message sent]



- Conditions:
  - ua = unilateral abort
  - tm = timeout
- Messages:
  - P = “prepare T”
  - R = “ready T”
  - AA = “abort T” answer
  - AC = “abort T” command
  - CC = “commit T” comman

Recovery

- No failure ➡ 2PC ensures that
  either all sub-transactions commit or
  all of them abort

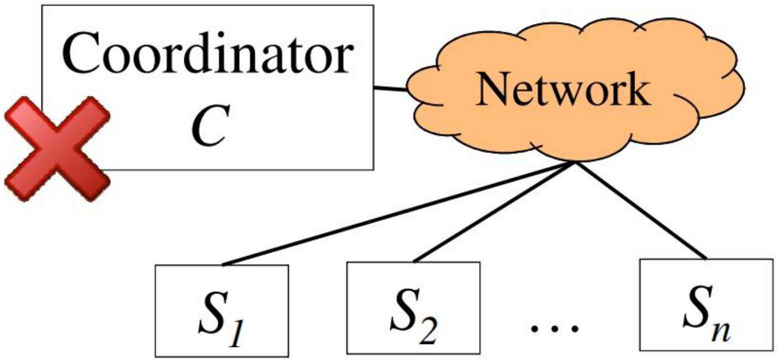
- Site / network failure ➡ need to make sure that the site's recovery is consistent with the global decision for *T*

- Types of failure:
  - (1) a site failure
  - (2) a coordinator failure
  - (3) network partition failure

(1) Site Failure

- When a site S_i recovers after a failure, it checks its log for entries for T. If it finds:
  - <**commit** *T*>: T has committed, **redo** T_i
  - <**abort** *T*>: T has been aborted, **undo** T_i
  - <**no** *T*>: S_i has not received the decision from C yet, but the decision must be to abort T, so **undo** T_i
  - <**ready** *T*>: S_i does not know the decision. It asks C to determine whether T has committed or aborted.
  - None of the above: C could not have decided to commit T. It would be safe to abort T_i (so **undo** T_i)

(2) Coordinator Failure

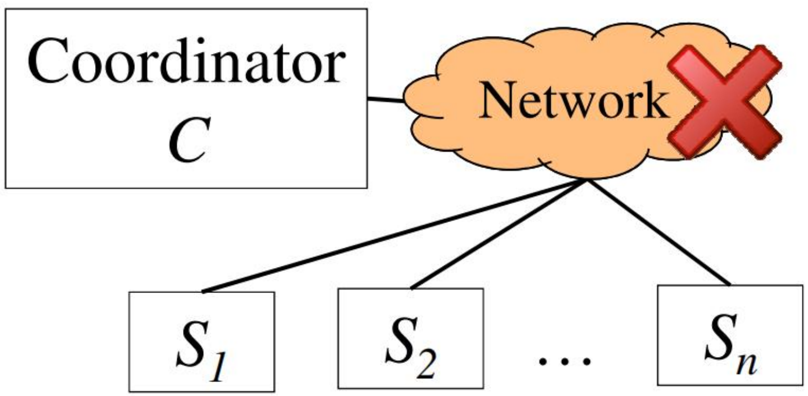


- If the coordinator fails and all the "living" participants are precommitted (<**ready** *T*> logged), no one knows the final decision until the coordinator recovers.
  - All "living" participants are blocked
  - This is the *blocking problem* of 2PC

- The blocking problem of 2PC is undesirable
  - sub-transactions are holding locks on data items
  - cause severe blocking to other transactions in the system

(3) Network Partition Failure

- When a network partition occurs, a participant cannot communicate with the coordinator
  - In that case, the separated participant assumes the coordinator fails, and
  - The coordinator assumes that the separated participant fails
- So all sites execute the same 2PC protocol



Summary

- An overview of the distributed database
- Data replication and fragmentation
- The two-phase commit protocol

Readings after the class

- Chapters 3 and 12.4 in the book

  Ozsu, and Valuriez. Principles of Distributed Database System, 3rd Ed, Springer, 2011 (free online)

南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

DBGroup

谢谢!

DBGroup @ SUSTech
Dr. Bo Tang (唐博)
tangb3@sustech.edu.cn