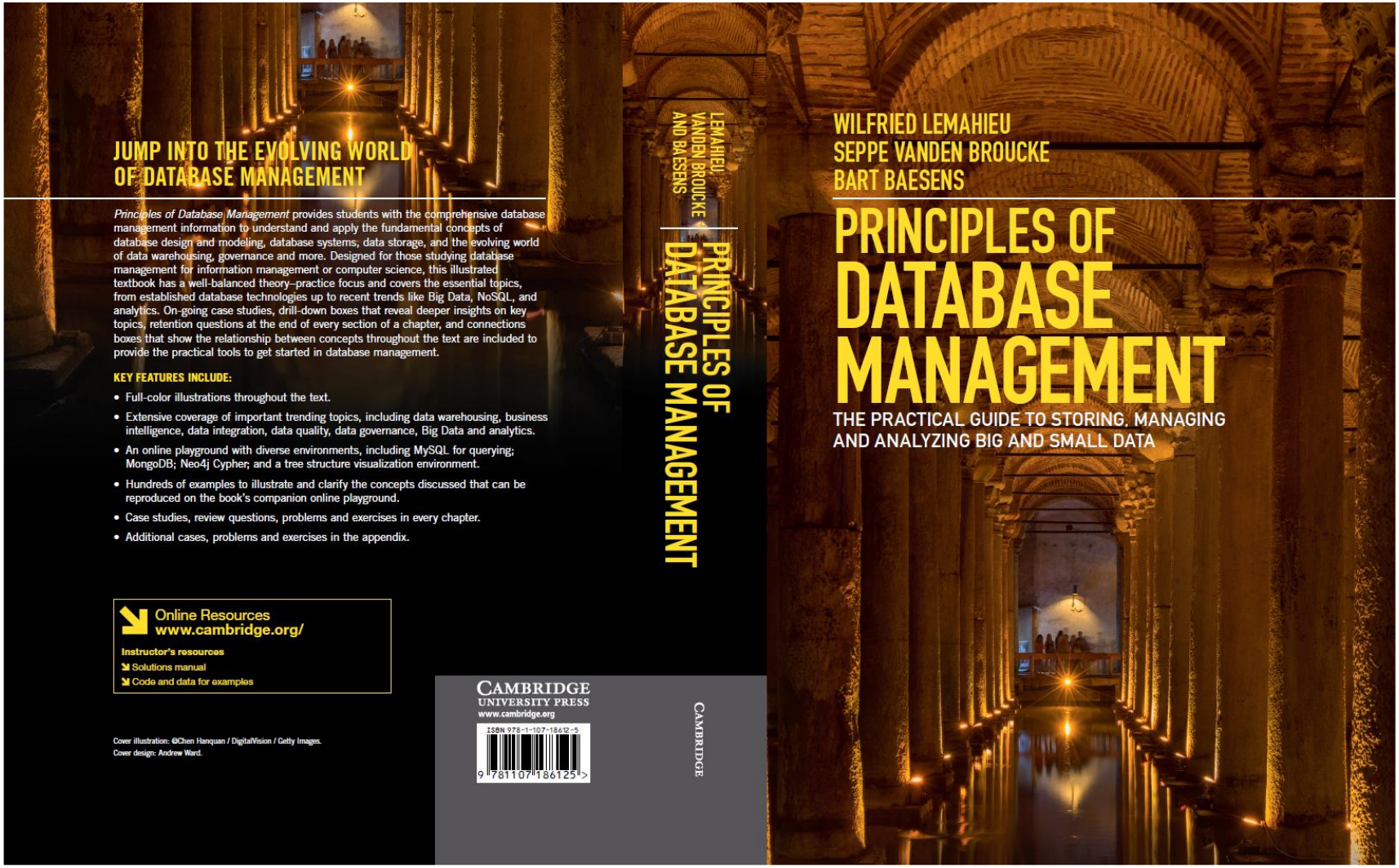
Data Distribution and Distributed Transaction Management



www.pdbmbook.com

Introduction

- Distributed Systems and Distributed Databases
- Architectural Implications of Distributed Databases
- Fragmentation, Allocation and Replication
- Transparency
- Distributed Query Processing
- Distributed Transaction Management and Concurrency Control
- Eventual Consistency and BASE Transactions

Distributed Systems and Distributed Databases

- Distributed computing system consists of several processing units (nodes) with a certain level of autonomy, which are interconnected by a network and which cooperatively perform complex tasks
- By dividing and distributing a complex problem into smaller, more manageable units of work, the complexity becomes manageable
- Distributed database systems distribute data and data retrieval functionality over multiple data sources and/or locations
- Goal is to provide an integrated view of the distributed data and transparency

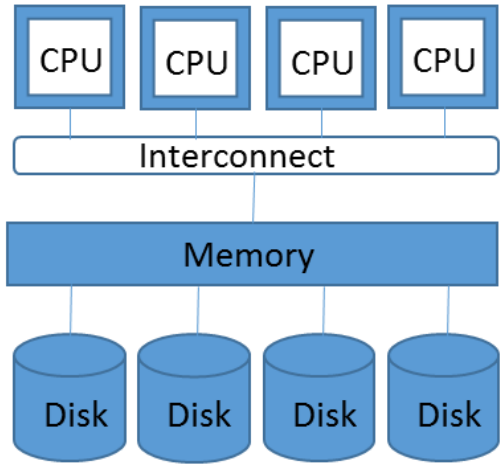
Distributed Systems and Distributed Databases

- Distributed database environment
  - deliberate choice
  - merger or acquisition
  - consecutive investments in different technologies
- Distribution aspects
  - performance
  - local autonomy
  - availability
  - degree of transparency
  - distributed query processing\transaction management\concurrency control and recovery

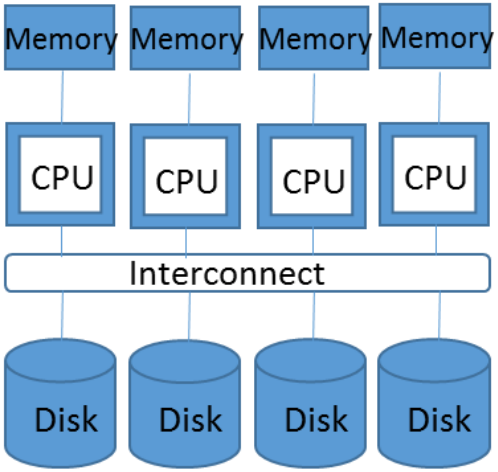
Architectural Implications of Distributed Databases

- Shared memory architecture
  - multiple interconnected processors that run the DBMS software share the same central storage and secondary storage
- Shared disk architecture
  - each processor has its own central storage but shares secondary storage with other processors (using e.g. a SAN or NAS)
- Shared-nothing architecture
  - each processor has its own central storage and hard disk units
  - data sharing occurs through the processors communicating with one another over the network

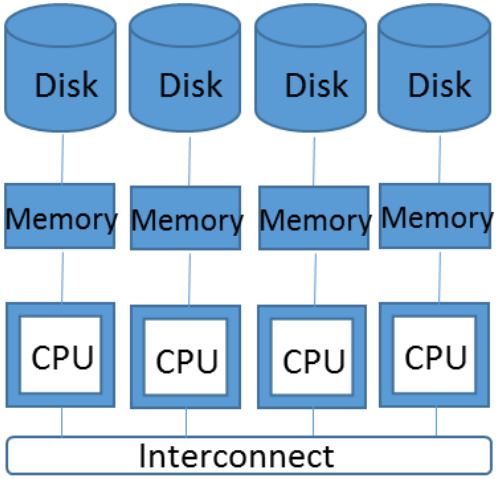
Architectural Implications of Distributed Databases



Shared memory architecture



Shared disk architecture



Shared nothing architecture

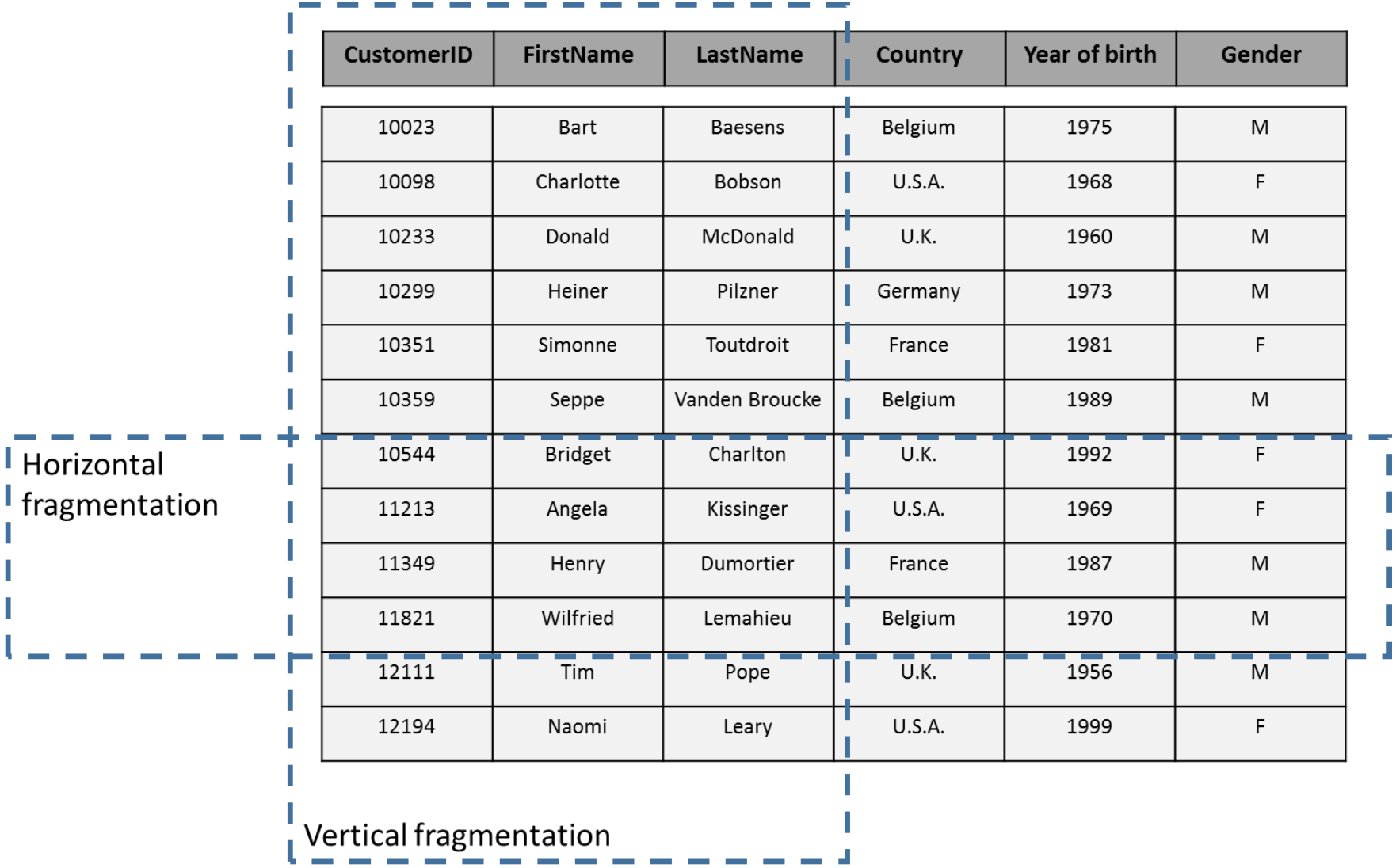
Architectural Implications of Distributed Databases

- Scalability can be achieved in 2 ways
  - capacity of nodes can be increased (vertical scalability)
  - more nodes can be added (horizontal scalability)
- Notes
  - parallel databases
    - focus on data distribution for performance
    - intra-query versus inter-query parallelism
  - federated databases
    - nodes in a shared-nothing architecture each run an independent DBMS instance with horizontal data fragmentation

Fragmentation, Allocation and Replication

- Fragmentation: partitioning of data into subsets (fragments) based on performance, local autonomy and availability
- Vertical fragmentation
  - fragment consists of subset of columns of data
  - global view with JOIN query
  - useful if only some of a tuple's attributes are relevant to a certain node
- Horizontal fragmentation (Sharding)
  - fragment consists of rows that satisfy a query predicate
  - global view with UNION query
  - common in NoSQL databases
- Mixed fragmentation
  - combines horizontal and vertical fragmentation
  - global view with JOIN + UNION query

Fragmentation, Allocation and Replication



CustomerID	FirstName	LastName	Country	Year of birth	Gender
10023	Bart	Baesens	Belgium	1975	M
10098	Charlotte	Bobson	U.S.A.	1968	F
10233	Donald	McDonald	U.K.	1960	M
10299	Heiner	Pilzner	Germany	1973	M
10351	Simonne	Toutdroit	France	1981	F
10359	Seppe	Vanden Broucke	Belgium	1989	M
10544	Bridget	Charlton	U.K.	1992	F
11213	Angela	Kissinger	U.S.A.	1969	F
11349	Henry	Dumortier	France	1987	M
11821	Wilfried	Lemahieu	Belgium	1970	M
12111	Tim	Pope	U.K.	1956	M
12194	Naomi	Leary	U.S.A.	1999	F

Fragmentation, Allocation and Replication

CustomerID	FirstName	LastName
10023	Bart	Baesens
10098	Charlotte	Bobson
10233	Donald	McDonald
10299	Heiner	Pilzner
10351	Simonne	Toutdroit
10359	Seppe	Vanden Broucke
10544	Bridget	Charlton
11213	Angela	Kissinger
11349	Henry	Dumortier
11821	Wilfried	Lemahieu
12111	Tim	Pope
12194	Naomi	Leary

CustomerID	Country	Year of birth	Gender
10023	Belgium	1975	M
10098	U.S.A.	1968	F
10233	U.K.	1960	M
10299	Germany	1973	M
10351	France	1981	F
10359	Belgium	1989	M
10544	U.K.	1992	F
11213	U.S.A.	1969	F
11349	France	1987	M
11821	Belgium	1970	M
12111	U.K.	1956	M
12194	U.S.A.	1999	F

Fragmentation, Allocation and Replication

CustomerID	FirstName	LastName	Country	Year of birth	Gender
10023	Bart	Baesens	Belgium	1975	M
10359	Seppe	Vanden Broucke	Belgium	1989	M
11821	Wilfried	Lemahieu	Belgium	1970	M
10351	Simonne	Toutdroit	France	1981	F
11349	Henry	Dumortier	France	1987	M

CustomerID	FirstName	LastName	Country	Year of birth	Gender
10299	Heiner	Pilzner	Germany	1973	M

CustomerID	FirstName	LastName	Country	Year of birth	Gender
10544	Bridget	Charlton	U.K.	1992	F
10233	Donald	McDonald	U.K.	1960	M
12111	Tim	Pope	U.K.	1956	M

CustomerID	FirstName	LastName	Country	Year of birth	Gender
11213	Angela	Kissinger	U.S.A.	1969	F
10098	Charlotte	Bobson	U.S.A.	1968	F
12194	Naomi	Leary	U.S.A.	1999	F

Fragmentation, Allocation and Replication

CustomerID	FirstName	LastName
10023	Bart	Baesens
10359	Seppe	Vanden Broucke
11821	Wilfried	Lemahieu
10351	Simonne	Toutdroit
11349	Henry	Dumortier

CustomerID	FirstName	LastName
10299	Heiner	Pilzner

CustomerID	FirstName	LastName
10544	Bridget	Charlton
10233	Donald	McDonald
12111	Tim	Pope

CustomerID	FirstName	LastName
11213	Angela	Kissinger
10098	Charlotte	Bobson
12194	Naomi	Leary

CustomerID	Country	Year of birth	Gender
10023	Belgium	1975	M
10098	U.S.A.	1968	F
10233	U.K.	1960	M
10299	Germany	1973	M
10351	France	1981	F
10359	Belgium	1989	M
10544	U.K.	1992	F
11213	U.S.A.	1969	F
11349	France	1987	M
11821	Belgium	1970	M
12111	U.K.	1956	M
12194	U.S.A.	1999	F

Fragmentation, Allocation and Replication

- Derived fragmentation
  - fragmentation criteria belong to another table
- Data replication occurs when the fragments overlap, or if there are multiple identical fragments allocated to different nodes so as to ensure
  - local autonomy
  - performance and scalability
  - reliability and availability
- Note: replication induces additional overhead and complexity to keep replicas consistent!

Fragmentation, Allocation and Replication

- Also metadata can be distributed and replicated
- Local versus global catalog

Transparency

- Transparency: application and users confronted with only a single logical database and are insulated from the complexities of the distribution
  - extension to logical and physical data independence
- Location transparency
  - users don’t know where required data resides
- Fragmentation transparency
  - users can execute global queries, without being concerned with the fact that distributed fragments will be involved, and need to be combined

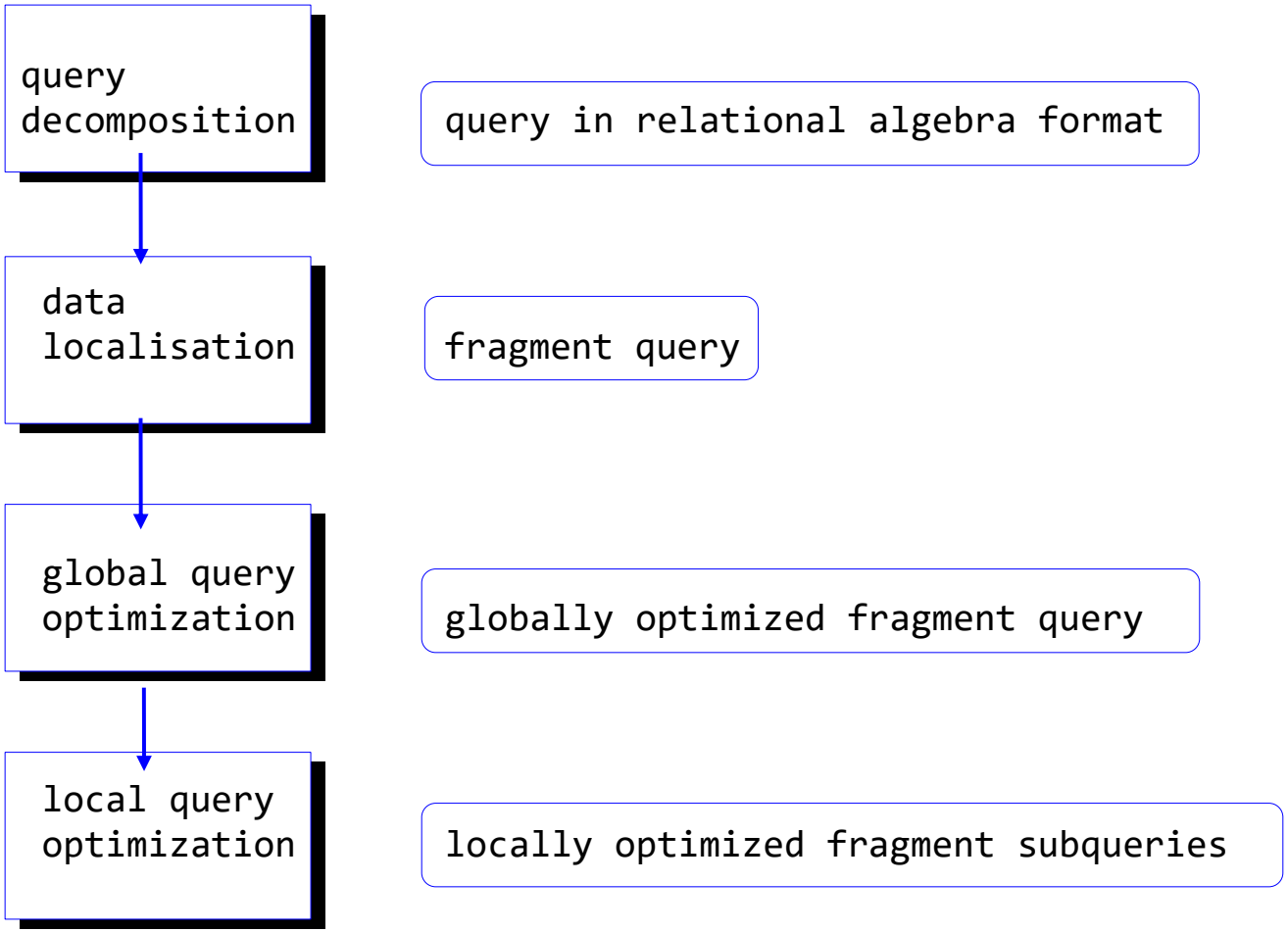
Transparency

- Replication transparency
  - different replicas will be kept consistent and updates to one replica will be propagated transparently to others
- Access transparency
  - distributed database can be accessed and queried uniformly, regardless of different database systems and APIs
- Transaction transparency
  - DBMS transparently performs distributed transactions as if they were transactions in a standalone system

Distributed Query Processing

- Optimizer should not only consider the elements of a standalone setting but also properties of respective fragments, communication costs, and location of the data in the network
- Also metadata may be distributed
- Both global (across all nodes) and local (within a single node) query optimization are needed

Distributed Query Processing



Distributed Query Processing

- Decomposition
  - query first analysed for correctness (syntax, etc.)
  - query represented in relational algebra and transformed into canonical form
- Data localization
  - transformation of query into a fragment query
  - database fragments and locations are identified
- Global query optimization
  - cost model is used to evaluate different global strategies
- Local query optimization
  - optimal strategy for local execution

Distributed Query Processing

SUPPLIER (SUPNR, SUPNAME, SUPADDRESS, SUPSTATUS)
PURCHASEORDER (PONR, PODATE, *SUPPLIER*)
PRODUCT (PNR, PNAME, PCOLOR, PWEIGHT, WAREHOUSE, STOCK)
…

Location 1:

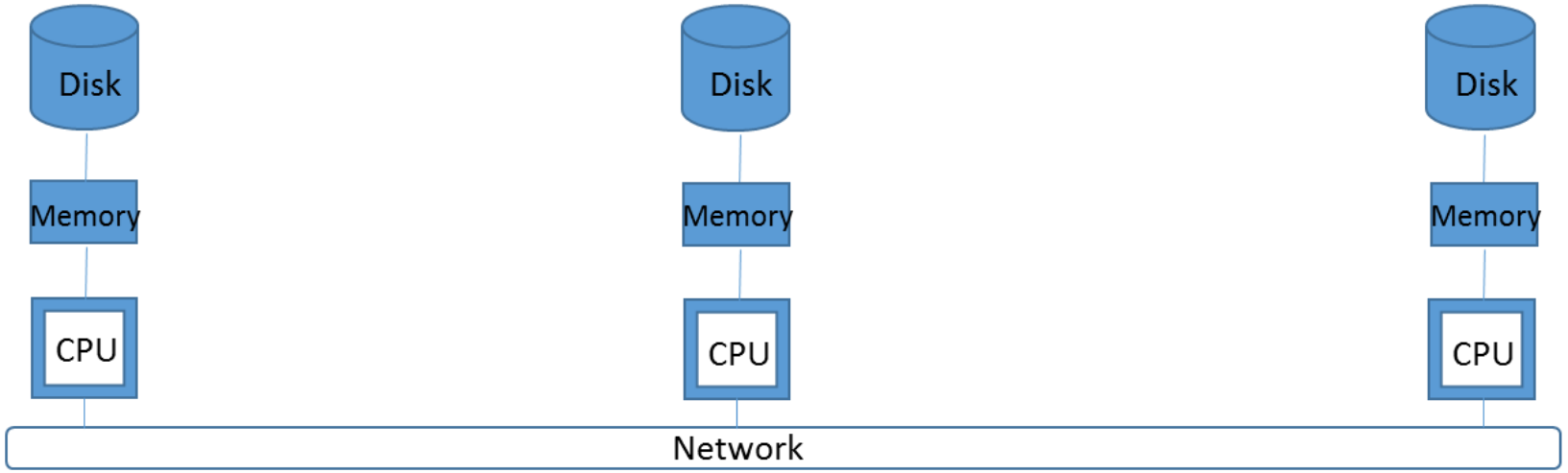
SUPPLIER table
SUPNR: 4 bytes
SUPNAME: 30 bytes
Entire row: 84 bytes
Number of rows: 1000

Location 2:

PURCHASEORDER table
PONR: 6 bytes
SUPPLIER: 4 bytes
Entire row: 16 bytes
Number of rows: 3000
On average, there are 200 suppliers with outstanding purchase orders

Location 3:

Query:
SELECT PONR, SUPNAME
FROM PURCHASEORDER PO, SUPPLIER S
WHERE PO.SUPPLIER = S.SUPNR



Distributed Query Processing

- Strategy 1
  - all tables are copied to location 3, which is also the location where all querying is performed
  - data transport amounts to (1000 x 84) + (3000 x 16) bytes = 132,000 bytes
- Strategy 2
  - SUPPLIER table is copied to location 2, where it is joined with the PURCHASEORDER table
  - query result is then sent to location 3
  - data transport amounts to (1000 x 84) + (3000 x (6 + 30)) bytes = 192,000 bytes

Distributed Query Processing

- Strategy 3
  - PURCHASEORDER table is copied to location 1, where it is joined with the SUPPLIER table
  - query result is sent to location 3
  - data transport amounts to (3000 x 16) bytes + (3000 x (6 + 30)) bytes = 156.000 bytes

Distributed Query Processing

- Strategy 4
  - use semi-join technique
  - in location 2, a projection operation is executed, to yield all numbers of suppliers associated with a purchase order (say around 200 supplier numbers)
  - numbers are copied to location 1, where a join operation is executed to combine supplier numbers with names
  - intermediate result is copied to location 2 for further processing
  - final result is sent to location 3
  - data transport amounts to only (200 x 4) + (200 x (4 + 30)) + (3000 x (6 + 30)) bytes = 115.600 bytes
  - Lowest network traffic but higher processing cost!

Distributed Transaction Management and Concurrency Control

- Tightly versus Loosely Coupled Setting
- Primary Site and Primary Copy 2PL
- Distributed 2PL
- Two-Phase Commit Protocol (2PC)
- Optimistic Concurrency and Loosely Coupled Systems
- Compensation-Based Transaction Models

Tightly versus Loosely Coupled Setting

- Global transaction coordinator versus local participants
- Tightly coupled setting
  - interdependence between participants and central control are substantial
  - distributed transaction required to have ‘ACID’ properties
  - typically paired with synchronous communication
- Loosely coupled setting
  - mobile devices, web services and programming models such as .NET
  - interactions based on asynchronous messaging and locally replicated data
  - cached local updates only synchronized periodically with global system
  - often apply some form of optimistic concurrency

Primary Site and Primary Copy 2PL

- Primary site 2PL
  - centralized 2PL protocol in distributed environment
  - single lock manager applies 2PL rules
  - lock manager informs coordinator when locks can be granted
  - participant that has completed processing will notify coordinator who, then instructs central lock manager to release locks
  - biggest advantage is relative simplicity (no global deadlocks!)
  - lock manager may become bottleneck + no location autonomy and limited reliability/availability

Primary Site and Primary Copy 2PL

- Primary Copy 2PL
  - lock managers implemented at different locations and maintain locking information pertaining to a predefined subset of the data
  - requests for granting and releasing locks directed to the lock manager responsible for that subset
  - impact of particular location going down will be less severe than with primary site 2PL

Distributed 2PL

- Every site has its own lock manager, which manages all locking data pertaining to the fragments stored on that site
- For global transactions that involve updates at n sites
  - n locking requests
  - n confirmations about whether the locks are granted or not
  - n notifications of local operations having completed
  - n requests to release the locks
- Location autonomy is respected but deadlocks may occur

Distributed 2PL

- If database has no replicated data, applying 2PL protocol guarantees serializability
  - serializable global schedule is union of local serializable schedules
- In case of replication, 2PL protocol must be extended

Distributed 2PL

	Location 1 (L_1)		Location 2 (L_2)	
time	$T_{1.1}$	$T_{2.1}$	$T_{1.2}$	$T_{2.2}$
t_1		begin transaction	begin transaction	
t_2	begin transaction	x-lock($amount_x$)	x-lock($amount_x$)	begin transaction
t_3	x-lock($amount_x$)	read($amount_x$)	read($amount_x$)	x- lock($amount_x$)
t_4	wait	$amount_x$= $amount_x$ x 2	$amount_x$= $amount_x$-50	wait
t_5	wait	write($amount_x$)	write($amount_x$)	wait
t_6	wait	commit	commit	wait
t_7	wait	unlock($amount_x$)	unlock($amount_x$)	wait
t_8	read($amount_x$)			read($amount_x$)
t_9	$amount_x$= $amount_x$-50			$amount_x$= $amount_x$ x2
t_{10}	write($amount_x$)			write($amount_x$)
t_{11}	commit			commit
t_{12}	unlock($amount_x$)			unlock($amount_x$)

Distributed 2PL

- 2PL can give rise to global deadlocks
  - can not be detected by local lock managers
- Deadlock detection in distributed 2PL requires the construction of a global wait-for graph
  - schedule is only deadlock free if not only the local graphs, but also the global graph contains no cycles

Distributed 2PL

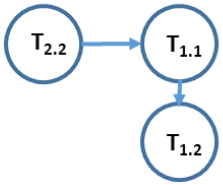
Transaction 1 (T_1)
begin transaction x-lock($account_x$) read($account_x$) $account_x$ = $account_x$ – 50 write($account_x$) x-lock($account_y$) read($account_y$) $account_y$ = $account_y$ + 50 write($account_y$) commit unlock($account_x$, $account_y$)

Transaction 2 (T_2)
begin transaction x-lock($account_y$) read($account_y$) $account_y$ = $account_y$ – 30 write($account_y$) x-lock($account_x$) read($account_x$) $account_x$ = $account_x$ + 30 write($account_x$) commit unlock($account_y$, $account_x$)

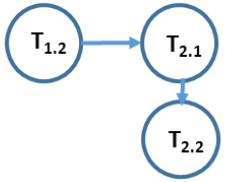
$Account_x$ is stored on Location 1
$Account_y$ is stored on Location 2

	Location 1 (L_1)		Location 2 (L_2)	
Time	$T_{1.1}$	$T_{2.2}$	$T_{2.1}$	$T_{1.2}$
t_1	begin transaction			
t_2	x-lock($account_x$)		begin transaction	
t_3	read($account_x$)		x-lock($account_y$)	
t_4	accountx = $account_x$ – 50		read($account_y$)	
t_5	write($account_x$)		$account_y$ = $account_y$ – 30	
t_6			write($account_y$)	x-lock($account_y$)
t_7		x-lock($account_x$)		wait
t_8		wait		wait

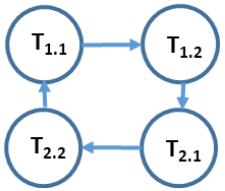
Local wait for graph for L_1



Local wait for graph for L_2



Global wait for graph



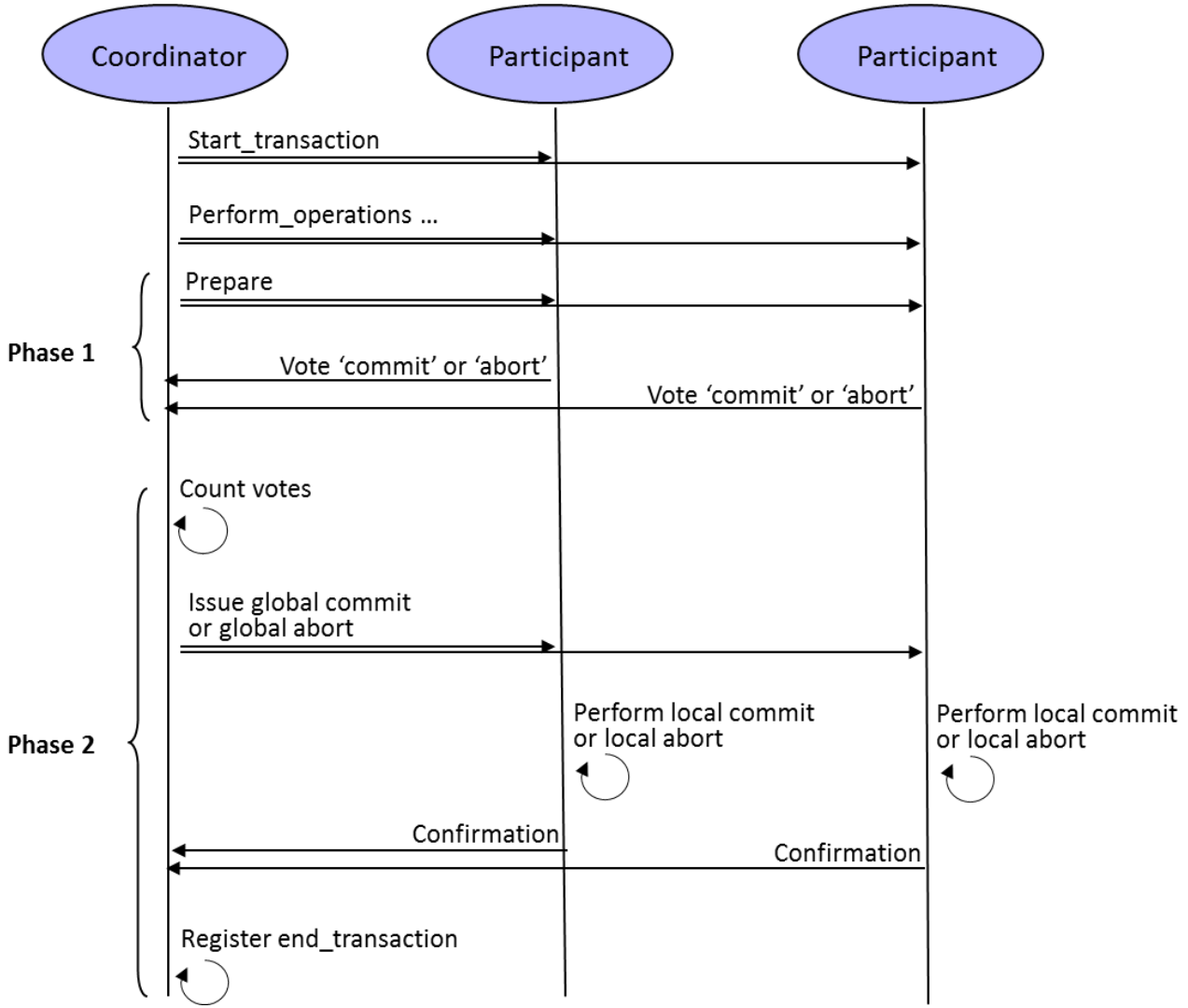
Distributed 2PL

- Example approach to detect global deadlocks
  - central site maintains global wait-for graph
  - all local lock managers periodically inform central site of changes in their local wait-for graphs
  - if one or more cycles are detected, local lock managers will be informed accordingly and victim selection will determine which transaction(s) to abort and roll back

Two-Phase Commit Protocol (2PC)

- Global transaction only attains ‘committed’ or ‘aborted’ state if all subtransactions have attained same state
- Two-Phase Commit (2PC) Protocol supports transaction recovery in distributed environment
- Global transaction completion involves 2 steps
  - voting phase: all participants ‘vote’ about transaction outcome
  - decision phase: transaction coordinator makes final decision about outcome

Two-Phase Commit Protocol (2PC)



Two-Phase Commit Protocol (2PC)

- 2PC protocol supplemented by
  - termination protocol: describes how to react to a timeout
  - recovery protocol: describes how faulty site should correctly resume operation after malfunction

Optimistic Concurrency and Loosely Coupled Systems

- Optimistic protocols resolve conflict before transaction commit, typically with abort and rollback of conflicting transaction
  - advantageous in distributed setting
- Tightly coupled distributed systems often apply pessimistic concurrency protocol
  - conflicting operations of other transactions postponed until locks released
  - transactions are typically sufficiently short-lived so as not to hold any locks for longer periods

Optimistic Concurrency and Loosely Coupled Systems

- Loosely coupled distributed systems often apply optimistic concurrency protocol
  - locks only held during brief period when database connection is open, to exchange data between the database and, e.g. an ADO.NET DataSet
- Example downside of optimistic concurrency
  - suppose application A_1 reads data from database into a DataSet and then closes the database connection
  - data in the disconnected dataset is updated locally
  - new connection is opened to propagate to database
  - no guarantee that data in the database was not altered by other application A_2!

Optimistic Concurrency and Loosely Coupled Systems

- Detecting conflicting updates in optimistic concurrency setting
  - Timestamps
    - 'timestamp' column added to any table that is open to disconnected access indicating time of most recent update
    - if A_1 retrieves rows and then disconnects, timestamp column is copied.
    - when application attempts to propagate its updates to the database, the timestamp associated with updated row is compared to timestamp of corresponding row in database
    - if both timestamps don't match, the update by A_1 will be refused. Otherwise, updated row can be propagated safely and new timestamp value is stored.

Optimistic Concurrency and Loosely Coupled Systems

- Detecting conflicting updates in an optimistic concurrency setting (contd.)
  - store 2 versions of each row in the disconnected entity: ‘current’ version and ‘original’ version
    - default in ADO.NET DataSet
    - ‘original’ version contains values read from the database
    - ‘current’ version contains values affected by local updates
    - when updates are propagated to the database, for each locally updated row, the ‘original’ values are compared to values of the corresponding row in the database. If the values are identical, the row is updated, otherwise, update is rejected.

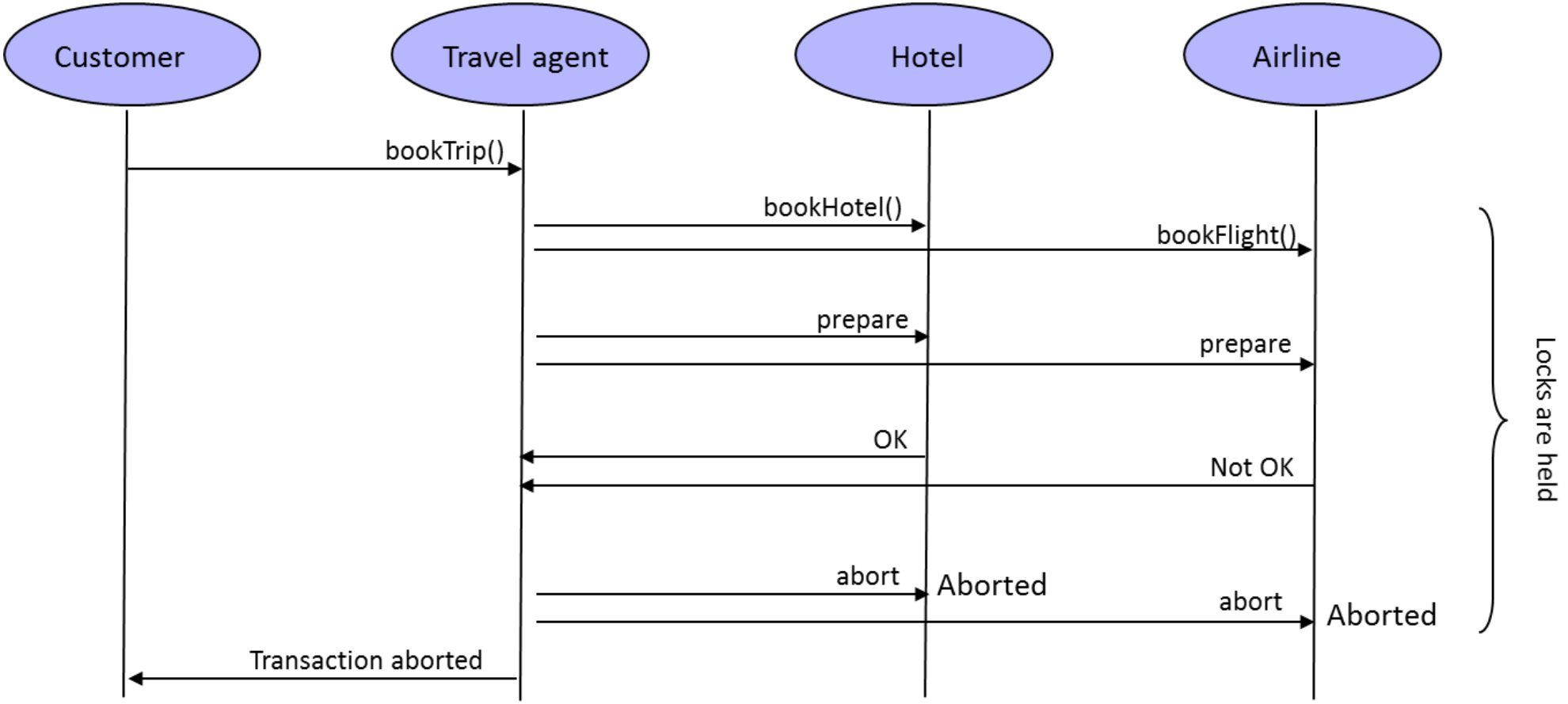
Optimistic Concurrency and Loosely Coupled Systems

```
UPDATE MYTABLE
SET    column1 = @currentValue1,
       column2 = @currentValue2,
       column3 = @currentValue3
WHERE column1 = @originalValue1
AND    column2 = @originalValue2
AND    column3 = @originalValue3
```

Compensation-Based Transaction Models

- Loosely coupled settings (e.g., web service environments) characterized by long running transactions
- Example: travel agent web service
  - pessimistic concurrency protocol not advised because locks might be held too long
  - optimistic concurrency not advised because coordinated rollback across participants infeasible

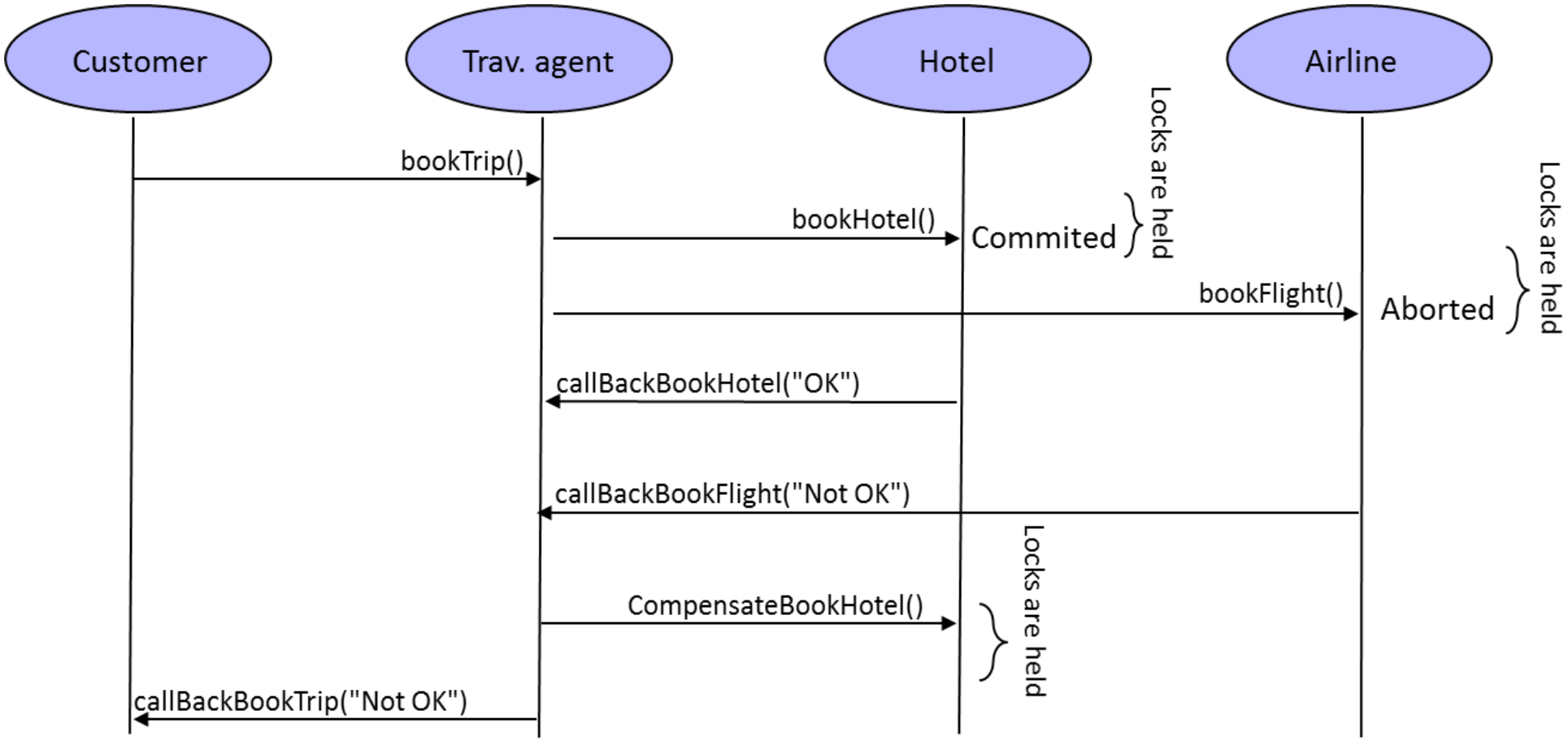
Compensation-Based Transaction Models



Compensation-Based Transaction Models

- Compensation-based transaction model
  - undo local effects of transaction if global long running transaction is unsuccessful
  - abandons ‘atomicity’ property of long running transaction
  - local subtransactions within single participant remain atomic and are committed asap, without waiting for global commit notification
  - requires each transaction participant to define its transaction sensitive operations in pairs, with the second operation specifying a new local transaction that cancels out the effects of the first (‘compensation’)

Compensation-Based Transaction Models



Compensation-Based Transaction Models

- Compensation-based transaction model (contd.)
  - requires for each transaction sensitive operation O_i, a compensating, hand-coded operation C_i
  - if global transaction that invoked O_i is aborted, C_i will be invoked by transaction coordinator
  - default transaction mechanism of, e.g. WS-BPEL, which define long running processes as orchestrations of individual web services
  - does not guarantee transaction isolation

Compensation-Based Transaction Models

- Closing thoughts
  - pessimistic concurrency preferable if many concurrent updates to same data subsets are expected, and data consistency is important
  - optimistic concurrency and/or compensation if concurrent access to same data subsets is limited or if read operations outnumber write operations
  - trade-off between consistency and performance also pertinent to divide between RDBMSs and NoSQL DBMSs

Eventual Consistency and BASE Transactions

- Horizontal Fragmentation and Consistent Hashing
- CAP Theorem
- BASE Transactions
- Multi-Version Concurrency Control and Vector Clocks
- Quorum-Based Consistency

Horizontal Fragmentation and Consistent Hashing

- NoSQL databases apply horizontal fragmentation (sharding)
- Shards allocated to different nodes in cluster with consistent hashing mechanism applied to data items' key
- Sharding usually entails replication and allows for parallel access
- NoSQL DBMSs approach linear horizontal scalability
- Sharding and replication yields high availability
- Not possible with traditional RDBMS and ACID transactions

CAP Theorem

- Distributed system has at most 2 of 3 properties
  - Consistency: all nodes see same data, and same versions of these data, at the same time
  - Availability: every request receives a response indicating success or failure
  - Partition tolerance: system continues to work even if nodes go down or are added

CAP Theorem

- Standalone system can provide both data consistency and availability with ACID transactions
- Tightly coupled distributed DBMSs often sacrifice availability for consistency and partition tolerance
- NoSQL DBMSs give up on consistency
  - in big data settings, unavailability is costlier than (temporary) data inconsistency
  - overhead of locking has severe impact on performance

CAP Theorem

- Critical Notes
  - performance degradation induced by overhead of mechanisms that enforce transactional consistency under normal operation, even in absence of network partitions is reason to abandon perpetual consistency
  - availability and performance same concepts, with unavailability an extreme case of high latency and low performance
  - real trade-off is between consistency and performance

BASE Transactions

- NoSQL DBMSs do not give up on consistency
- BASE transactions
  - Basically Available: measures in place to guarantee availability under all circumstances, if necessary at cost of consistency
  - Soft State: state may evolve, even without external input, due to asynchronous propagation of updates
  - Eventually consistent: database will become consistent over time, but may not be consistent at any moment and especially not at transaction commit

BASE Transactions

- Write operations performed on one or a few replicas of data item
- Updates to other replicas propagated asynchronously in background
- Read operations performed on one or few replicas
- If read of multiple replicas gives inconsistent results
  - DBMS uses timestamps (e.g., ‘last write wins’) and returns most recent version
  - applications determines how conflicting replicas are reconciled

Multi-Version Concurrency Control and Vector Clocks

- With BASE transactions, conflict resolution does not necessarily happen at moment of writing data but may be postponed until data is read
- Rather than postponing write operation, new version of data item is created with updated value
- DBMS may contain multiple inconsistent versions of a data item
  - conflict resolution is postponed until data is retrieved

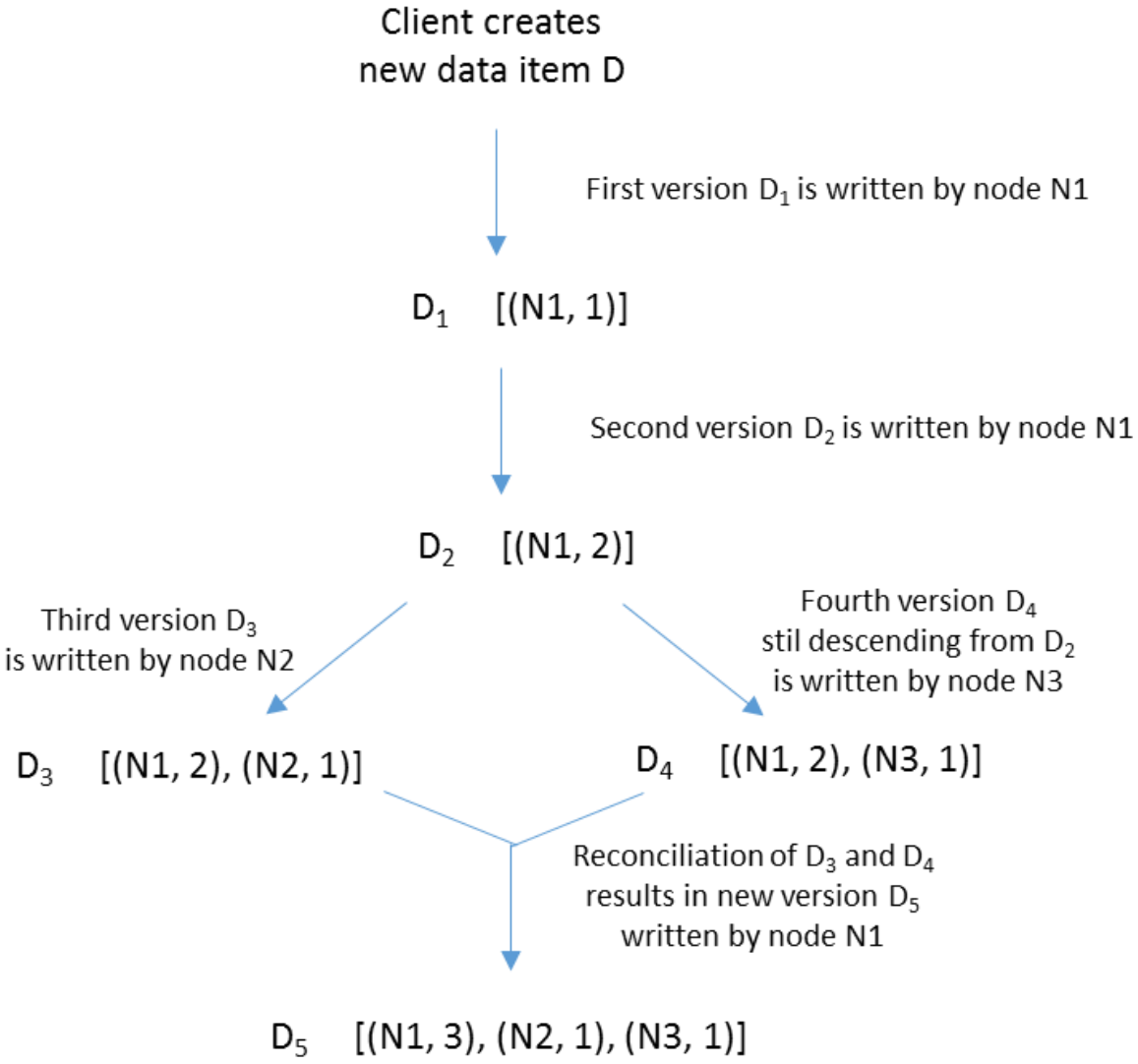
Multi-Version Concurrency Control and Vector Clocks

- MVCC (Multi-Version Concurrency Control) protocol
  - read operation returns one or more versions of data item; conflicts resolved by DBMS or application
  - write operation creates new version of data item
  - vector clocks used to discriminate between data item versions and to trace their origin
  - obsolete versions of data item garbage collected
- Vector clock consists of list of [node, counter] pairs
  - node refers to node that handled write of that version
  - counter denotes version number of writes by that node
- Entirety of vector clocks associated with versions of data item represents lines of descendance of respective versions

Multi-Version Concurrency Control and Vector Clocks

- Read operation retrieves all conflicting versions of data item, with the versions' vector clocks
- Write operation creates new version of a data item with corresponding vector clock
- If all counters in a version's vector clock are less-than-or-equal-to all counters in another version's clock, then the first version is an ancestor of the second one and can safely be garbage collected
  - otherwise, both versions represent conflicting versions and should be retained (may be reconciled later)

Multi-Version Concurrency Control and Vector Clocks



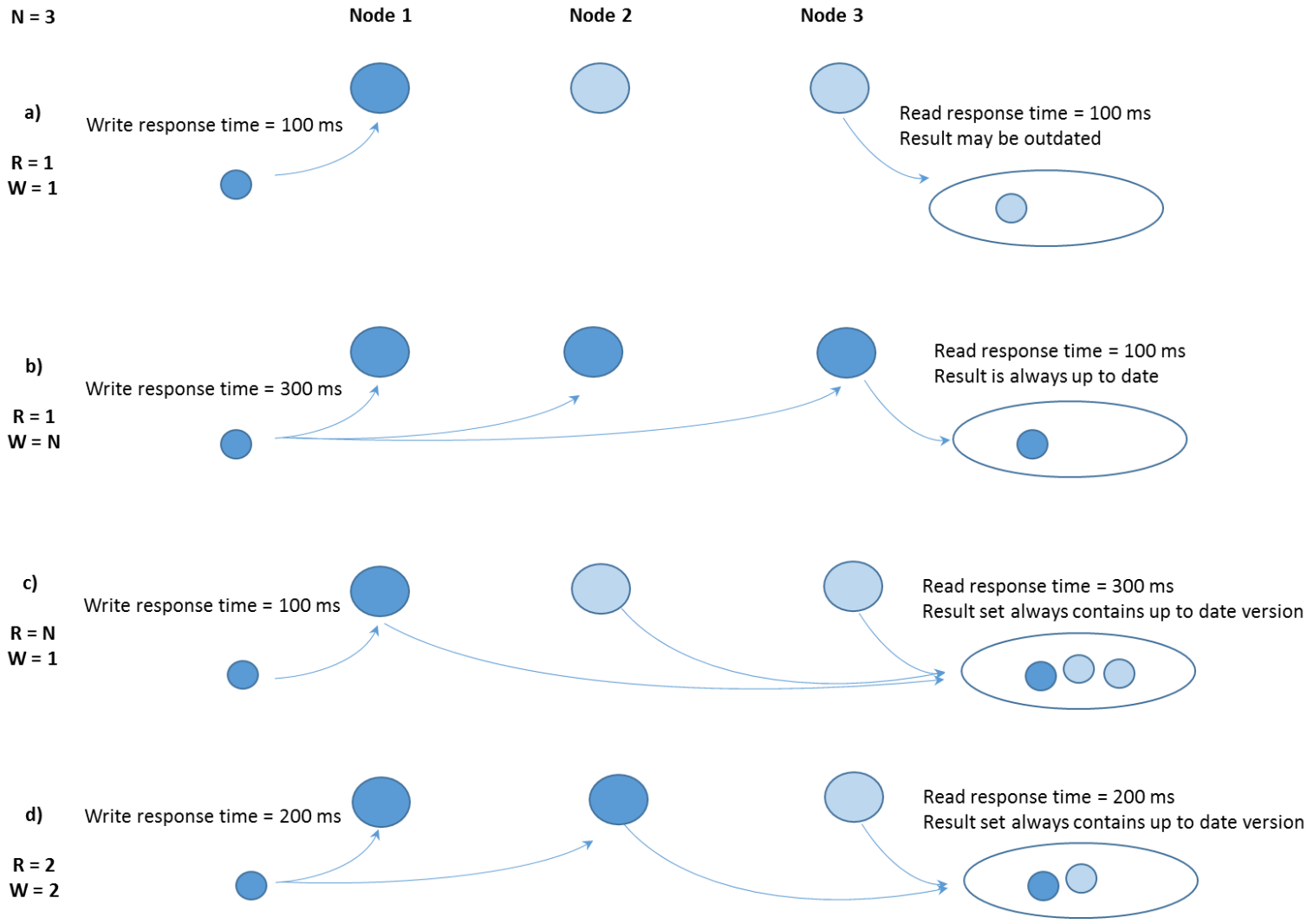
Quorum-Based Consistency

- Administrator can position NoSQL DBMS on continuum between high availability and permanent consistency using quorum-based protocol
- Quorum based protocols enforce consistency using 3 parameters N, R and W, with $R \leq N$ and $W \leq N$
  - N represents number of nodes to which data item is replicated
  - R is minimum number of nodes that should respond before a read operation can be considered as completed
  - W is the minimum number of nodes that should receive updated value before write operation can be considered as completed

Quorum-Based Consistency

- By manipulating R and W, administrator can decide on trade-off between performance and consistency , but also on trade-off between read and write performance
- Configuration with $R + W > N$ is guaranteed to provide at least one up to date replica with each read operation
- Configurations with $R + W \leq N$ have better read and/or write performance, but it cannot be guaranteed that the result set of each read operation will contain an up to date replica

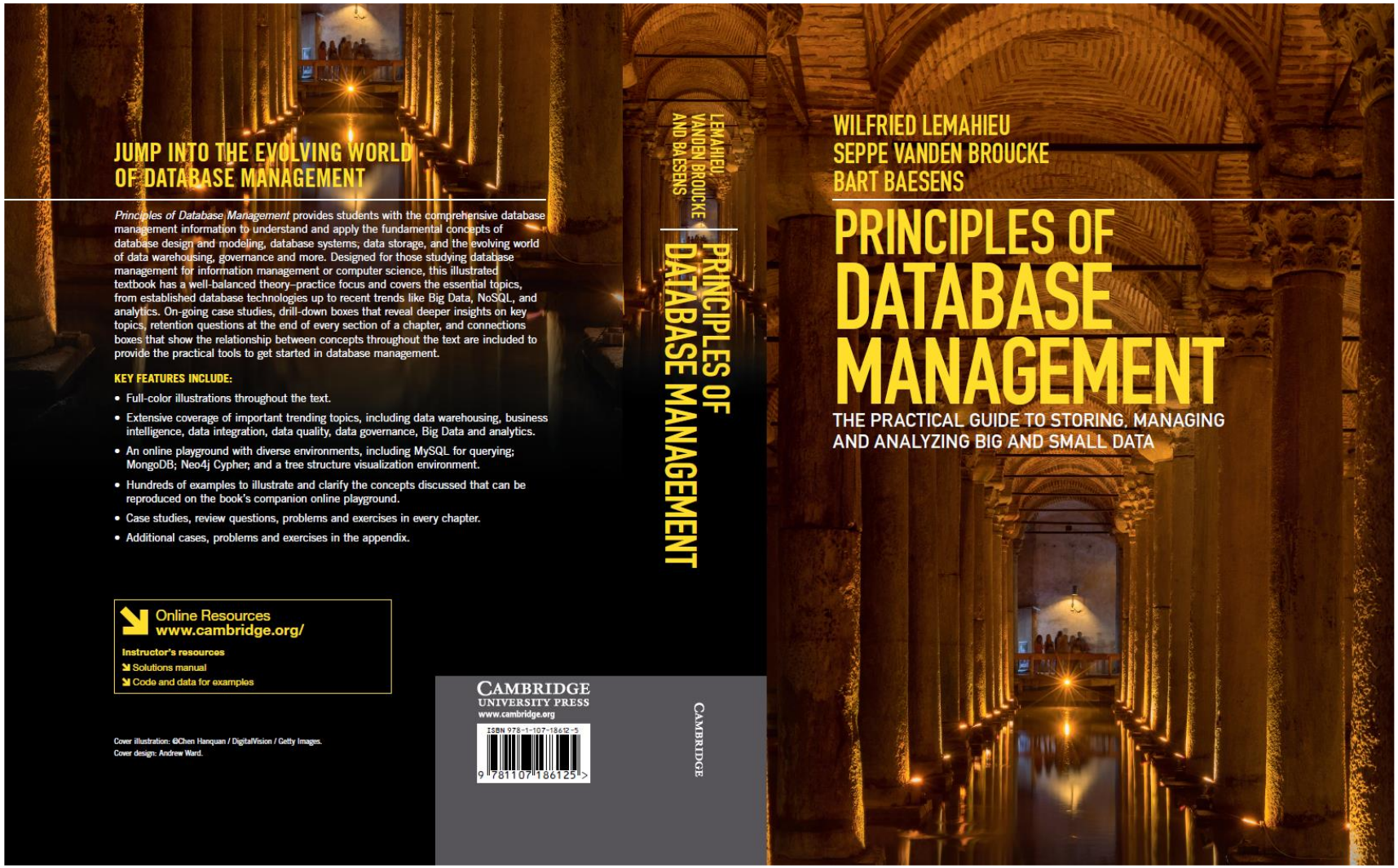
Quorum-Based Consistency



Conclusions

- Distributed Systems and Distributed Databases
- Architectural Implications of Distributed Databases
- Fragmentation, Allocation and Replication
- Transparency
- Distributed Query Processing
- Distributed Transaction Management and Concurrency Control
- Eventual Consistency and BASE Transactions

More information?



www.pdbmbook.com