Database Management Systems

Lectures 7-8-9*

Parallel Databases

Spatial Databases

Distributed Databases

* 22.04.2022 – Public holiday

Parallel Databases

Parallel Database Systems

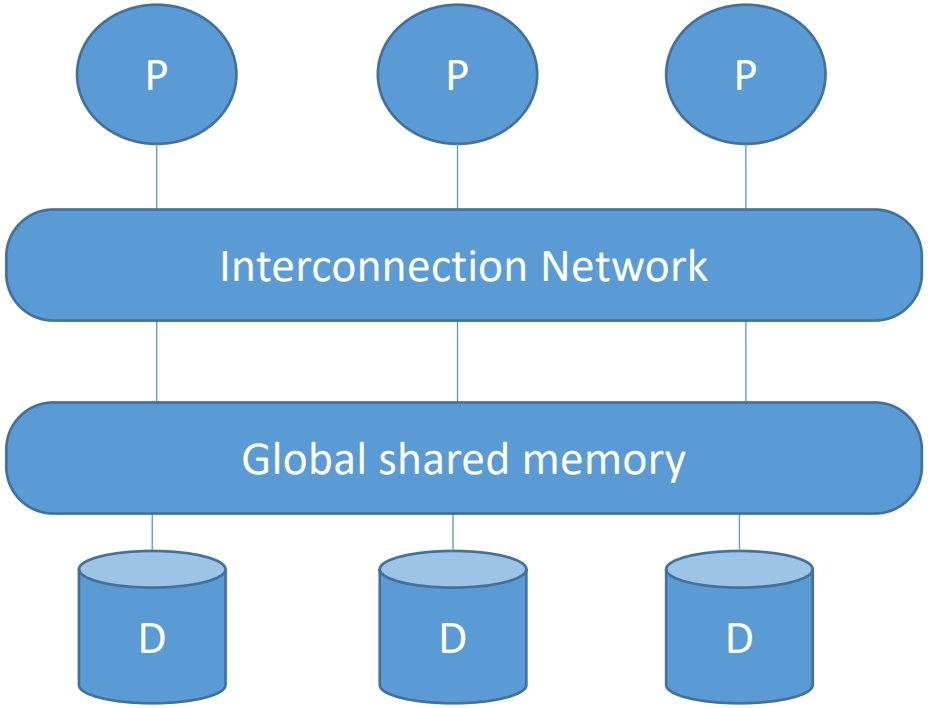
- performance improvement
  - parallelize operations:
    - loading data
    - building indexes
    - query evaluation
  - data can be distributed, but distribution is dictated solely by performance reasons

Parallel Databases - Architectures

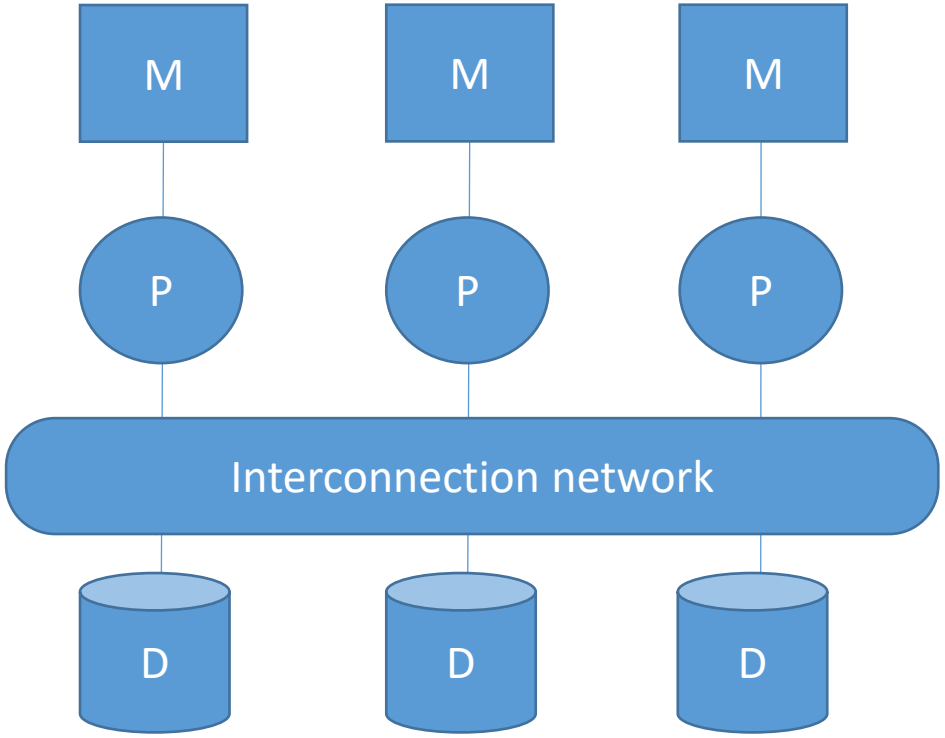
- *shared-memory*
- *shared-disk*
- *shared-nothing*

Parallel Databases - Architectures

- *shared-memory*
  - several CPUs:
    - attached to an interconnection network
    - can access a common region in the main memory

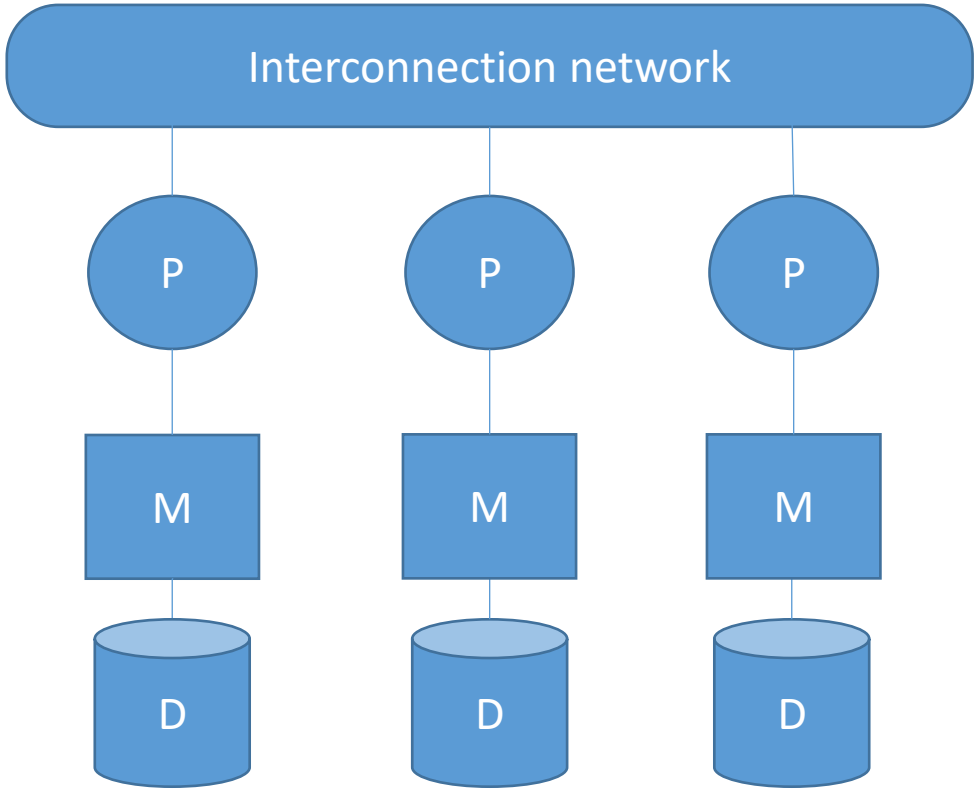


- *shared-disk*
  - a CPU:
    - its own private memory
    - can access all disks through a network



Parallel Databases - Architectures

- *shared-nothing*
  - a CPU:
    - its own local main memory
    - its own disk space
  - 2 different CPUs cannot access the same storage area
  - CPUs communicate through a network



Interference

- specific to shared-memory and shared-disk architectures
- add CPUs:
  - increased contention for memory and network bandwidth

  => existing CPUs are slowing down
- main reason that led to the shared-nothing architecture, currently considered as the best option for large parallel database systems

The Shared-Nothing Architecture

- linear speed-up & linear scale-up

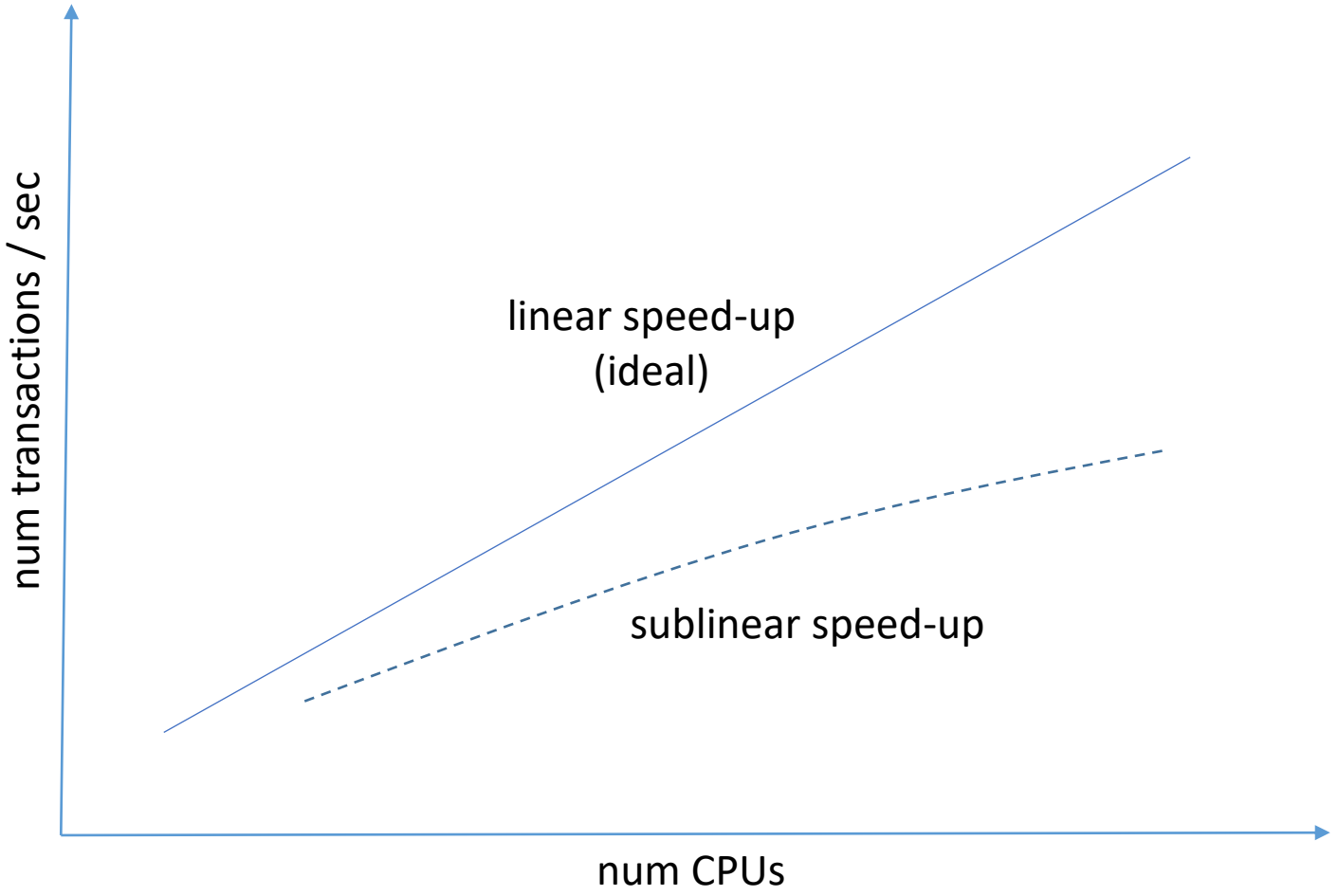
- linear speed-up
  - required processing time for operations decreases proportionally to the increase in the number of CPUs and disks

- linear scale-up
  - num. of CPUs and disks grows proportionally to the amount of data

  => performance is sustained

The Shared-Nothing Architecture

- speed-up

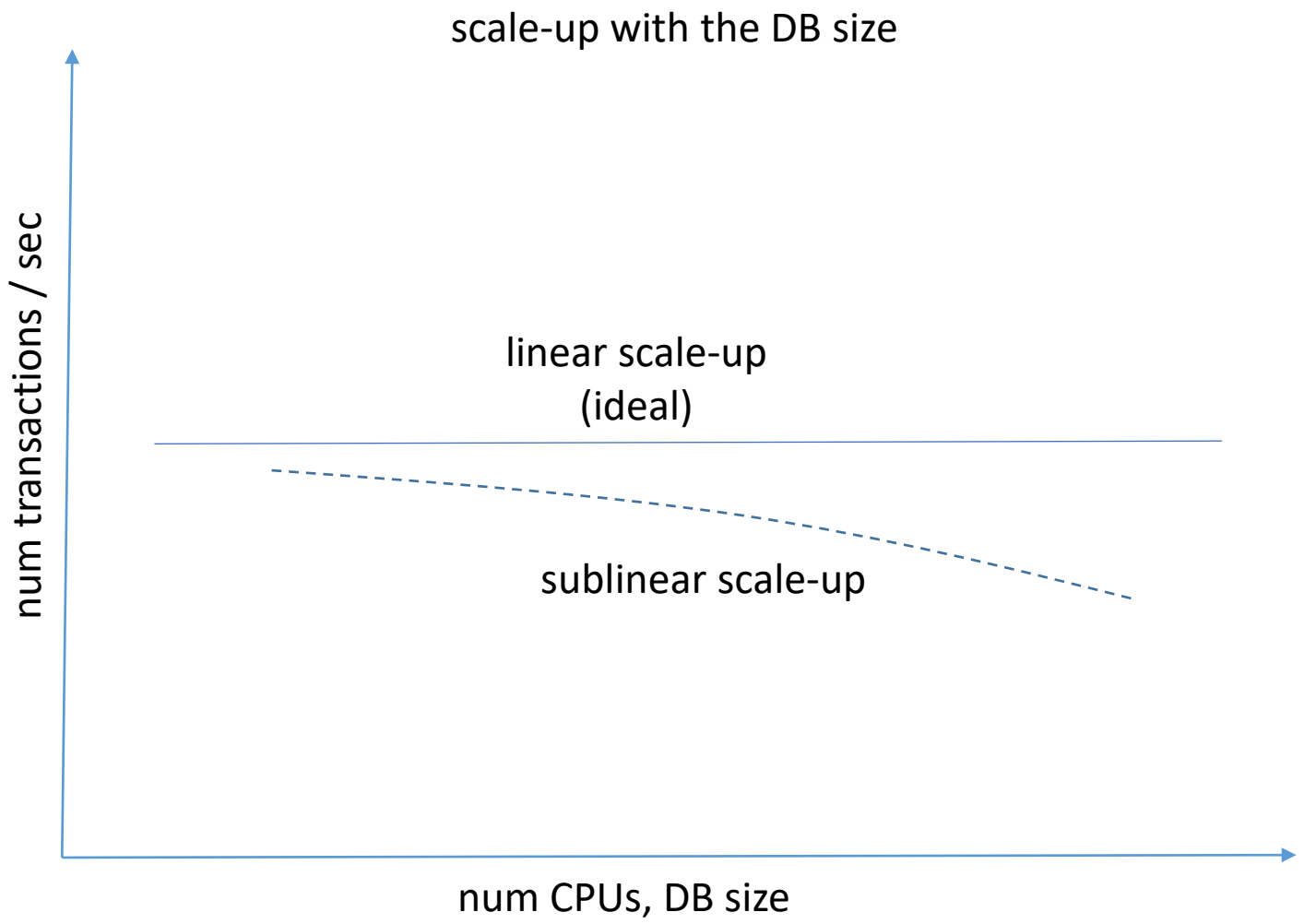


- DB size - fixed
- add CPUs

=> more transactions can be executed per second

The Shared-Nothing Architecture

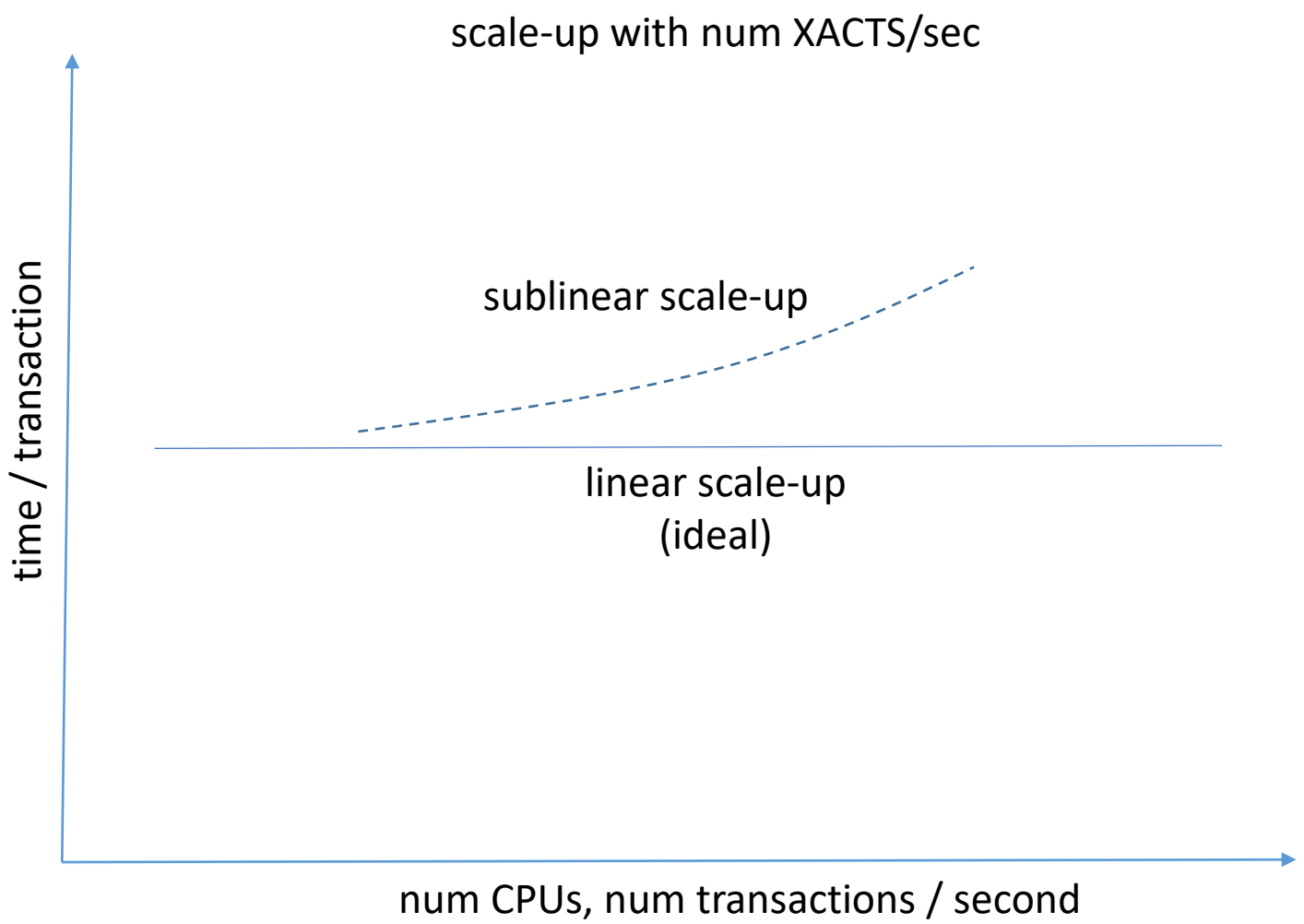
- scale-up



- the number of transactions executed per second, as the DB size and the number of CPUs increase

The Shared-Nothing Architecture

- scale-up



* alternative

- add CPUs as the number of transactions executed per second increases
- evaluate the time required for a transaction

Parallel Query Evaluation

- context
  - DBMS based on a shared-nothing architecture
- evaluate a query in a parallel manner
- operators in an execution plan can be evaluated in parallel
  - 2 operators are evaluated in parallel
  - one operator is evaluated in a parallel manner
- an operator is said to *block* if it doesn't produce results until it consumes all its inputs (e.g., sorting, aggregation)
- pipelined parallelism
  - an operator consumes the output of another operator
  - limited by blocking operators

Parallel Query Evaluation

- parallel evaluation on partitioned data
  - every operator in a plan can be evaluated in a parallel manner by partitioning the input data
  - partitions are processed in parallel, the results are then combined
- processor = CPU + its local disk

Parallel Query Evaluation – Data Partitioning

- horizontally partition a large dataset on several disks
- partitions are then read / written in parallel
- *round-robin* partitioning
  - n processors
  - the i^{th} tuple is assigned to processor i % n
- hash partitioning
  - determine the processor for a tuple t
    - apply a hash function to t ((some of) its attributes)
- range partitioning
  - n processors
  - order tuples conceptually
  - choose n ranges for the sorting key values s.t. each range contains approximately the same number of tuples
  - tuples in range i are assigned to processor i

Parallel Query Evaluation – Data Partitioning

- queries that scan the entire relation
  - round-robin partitioning - suitable
- queries that operate on a subset of tuples
  - equality selection, e.g., age = 30
    - tuples partitioned on the attributes in the selection condition, e.g., age
    - hash and range partitioning are better than round-robing (one can access only the disks containing the desired tuples)
  - range selection, e.g., 20 < age < 30
    - range partitioning is better than hash partitioning (it's likely that the desired tuples are grouped on several processors)

Parallelizing Individual Operations

- context
  - DBMS based on a shared-nothing architecture
- each relation is horizontally partitioned on several disks
- scanning a relation
  - pages can be read in parallel
  - obtained tuples can than be reunited
  - similarly – obtain all tuples that meet a selection condition

Parallelizing Individual Operations

- sorting
  - v1
    - each CPU sorts the relation fragment on its disk
    - subsequently, the sorted tuple sets are merged
  - v2
    - redistribute tuples in the relation using range partitioning
    - each processor sorts its tuples with a sequential sorting algorithm => several sorted runs on the disk
    - merge runs => sorted version of the set of tuples assigned to the current processor
    - obtain the entire sorted relation
      - visit processors in an order corresponding to their assigned ranges and scan the tuples

Parallelizing Individual Operations

- sorting
  - v2
    - challenges
      - range partitioning – assign approximately the same number of tuples to each processor
      - a processor that receives a disproportionately large number of tuples will limit scalability

Spatial Databases

Types of Spatial Data and Queries

- spatial data
  - multidimensional points
  - lines
  - rectangles
  - cubes
  - etc.
- spatial extent (SE)
  - region of space occupied by an object
  - characterized by location + boundary

Types of Spatial Data and Queries

- DBMS
  - point data
  - region data

- point data
  - collection of points in a multidimensional space
  - point
    - SE – only location
    - no space, area, volume
  - direct measurements (e.g., MRI)
  - transforming data objects (e.g., feature vectors)

Types of Spatial Data and Queries

- region data
  - collection of regions
  - region - SE
    - location
      - position of a fixed anchor point for the region
    - boundary (line, surface)
  - geometric approximations to objects, built with line segments, polygons, spheres, cubes, etc.
  - e.g., in geographic apps
    - line segments – roads, railways, rivers, etc.
    - polygons – lakes, counties, countries, etc.

Types of Spatial Data and Queries

- spatial queries
  - spatial range queries
  - nearest neighbor queries
  - spatial join queries

  - spatial range queries
    - associated region
      - location + boundary
    - find all regions that overlap / are contained within the specified range
    - e.g., Find all cities within 150 km of Constanța.
    - e.g., Find all rivers in Bihor.

Types of Spatial Data and Queries

- spatial queries
  - nearest neighbor queries
    - e.g., Find the 5 cities that are nearest to Paris.
    - usually, answers are ordered by proximity
  - spatial join queries
    - e.g., Find all cities near Bucharest.
    - e.g., Find pairs of cities within 150 km of each other.
    - meaning of queries can be determined by the level of detail in the representation of objects
    - e.g., relation in which each tuple is a point that represents a city
      - answer the queries with a self join
      - join condition – distance between 2 tuples in the query result

Types of Spatial Data and Queries

- spatial queries
  - spatial join queries
    - e.g., cities have a boundary

    => meaning of queries and evaluation strategies become more complex
      - cities whose centroids are within 150 km of each other?
      - or cities whose boundaries come within 150 km of each other?

Applications

- relation R with k attributes seen as a collection of k-dimensional points
- Geographic Information Systems (GIS)
  - point & region data
  - e.g., a map that contains the locations of several small objects (points), highways (lines), cities (regions)
  - range, nearest neighbor, join queries
- Computer-aided design (CAD) systems, medical imaging systems
  - store spatial objects
  - point & region data
  - most frequent queries - range & join
  - spatial integrity constraints

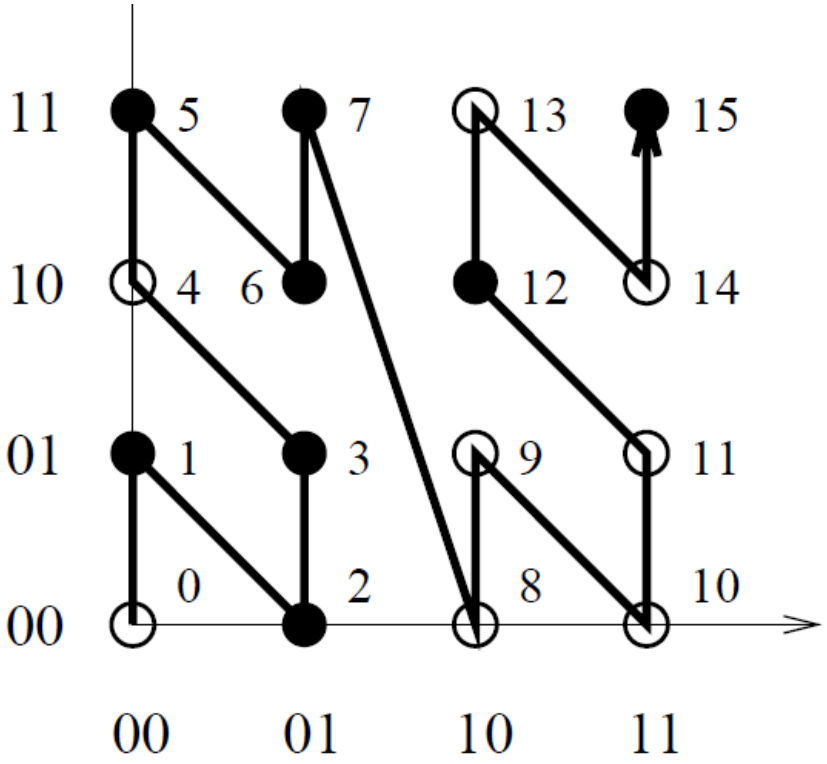
Applications

- multimedia DBs
  - multimedia objects
    - images, text, *time-series* data (e.g., audio)
  - object
    - point in a multidimensional space
  - similarity of 2 multimedia objects
    - distance between the corresponding points
  - similarity queries seen as nearest neighbor queries
  - point data & nearest neighbor queries – most common

Indexing - *space-filling curves*

- assumption
  - any attribute value can be represented with a fixed num. of bits, e.g., k bits

  => max. num. of values per dimension = 2^k
- the point with X = 01, Y = 11 has Z-value = 0111, obtained by interleaving the bits of X and Y
  - the 8^{th} point visited by the space-filling curve, which starts at point X=00, Y=00
- points in the dataset are stored in the order of their Z-values
- Z-ordering curve
  - linear ordering on the domain



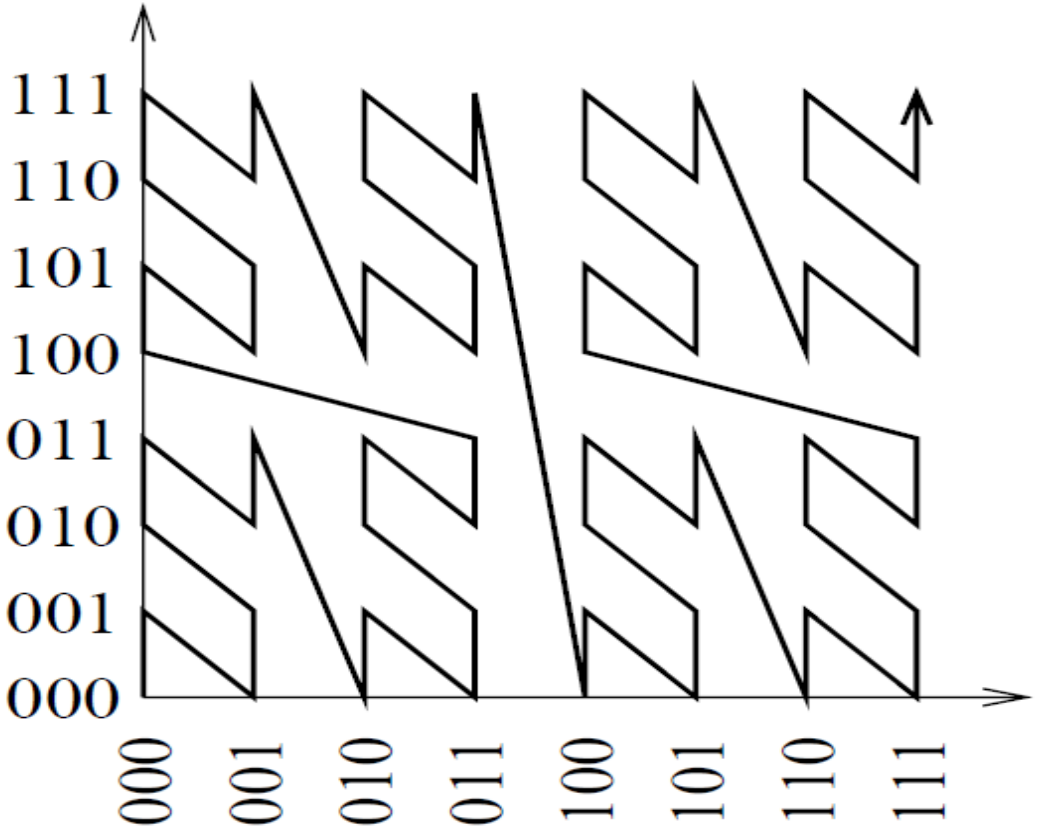
Z-ordering with 2 bits

Indexing - *space-filling curves*

- Z-ordering curve
  - the curve visits all the points in a quadrant before moving on to the next quadrant; all points in a quadrant are stored together
- indexing
  - B+ tree
  - search key: Z-value
  - store the Z-value of the point (along with the point)
  - I / D / search point
    - compute Z-value
    - I / D / search into / from / the B+ tree
  - unlike when using traditional B+ tree-based indexing, points are clustered together by spatial proximity in the X-Y space
  - spatial queries in the X-Y space become linear range queries
    - efficient evaluation on the B+ tree organized by Z-values

Indexing - *space-filling curves*

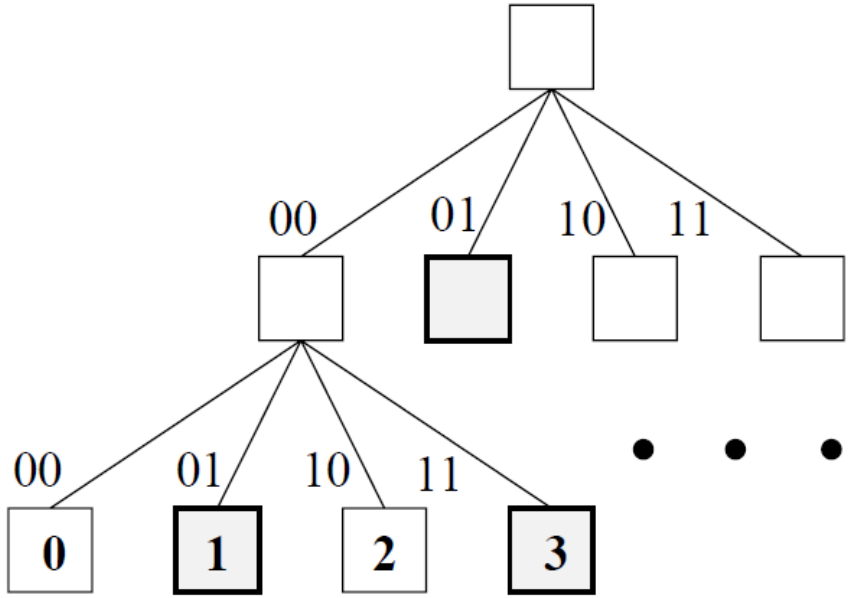
- Z-ordering curve when k = 3

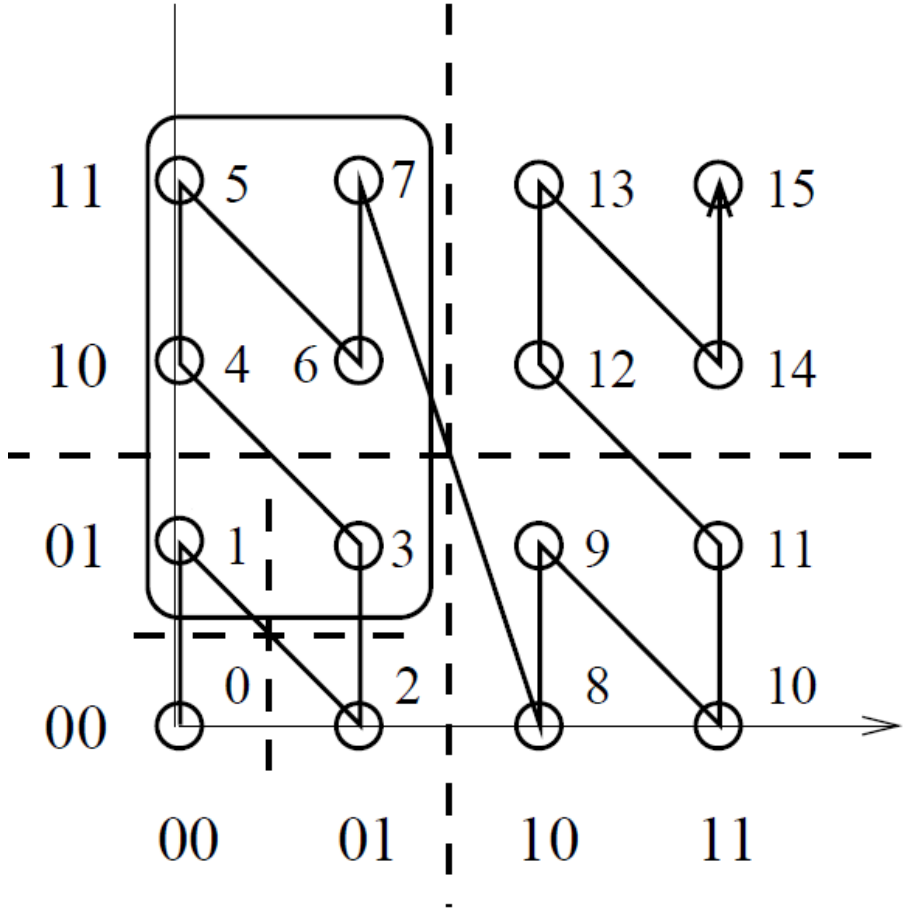


Z-ordering with 3 bits

Region Quad Trees

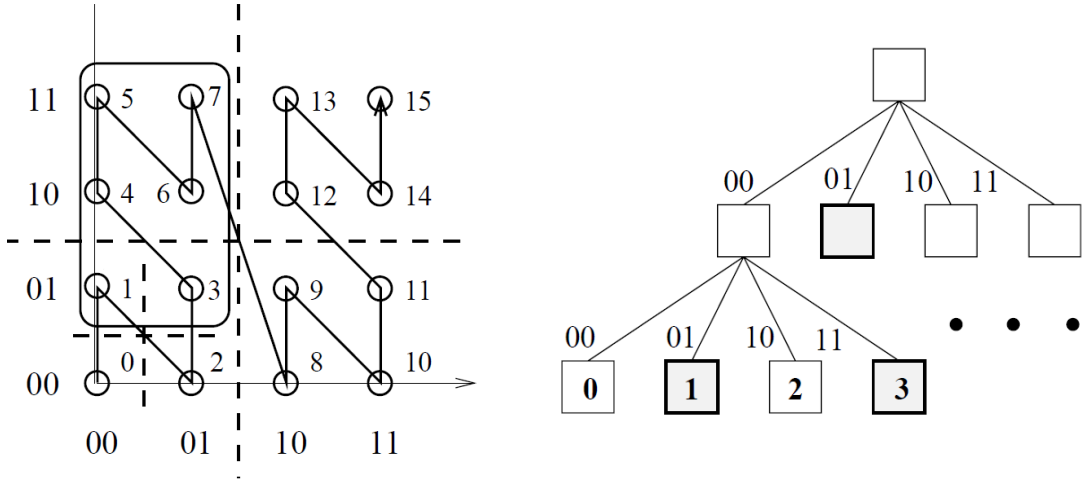
- region data
- Z-ordering recursively decomposes space into quadrants and subquadrants
- the structure of a Region Quad tree directly corresponds to the recursive decomposition of the data space
- each node in the tree corresponds to a square region in the data space





Region Quad Trees

- root – entire data space
- internal node – 4 children
- rectangle object R stored by the DBMS
  - all points in the 01 quadrant of the root and the points with Z-values 1 and 3
  - store 3 records: <0001, R>, <0011, R>, <01, R>
  - records clustered and indexed by the 1st field (the Z-value) in a B+ tree
- use B+ trees to implement Region Quad trees
- generalization - k dimensions => at every node, the space is partitioned into 2^k subregions



Distributed Databases

* centralized DB systems

- all data at a single site
- each transaction is processed sequentially
- centralized lock management
- processor fails => entire system fails

* distributed systems

- the data is stored at several sites
- each site is managed by a DBMS that can run independently; these autonomous components can also be heterogeneous

=> impact on: query processing, query optimization, concurrency control, recovery

Distributed Database Systems

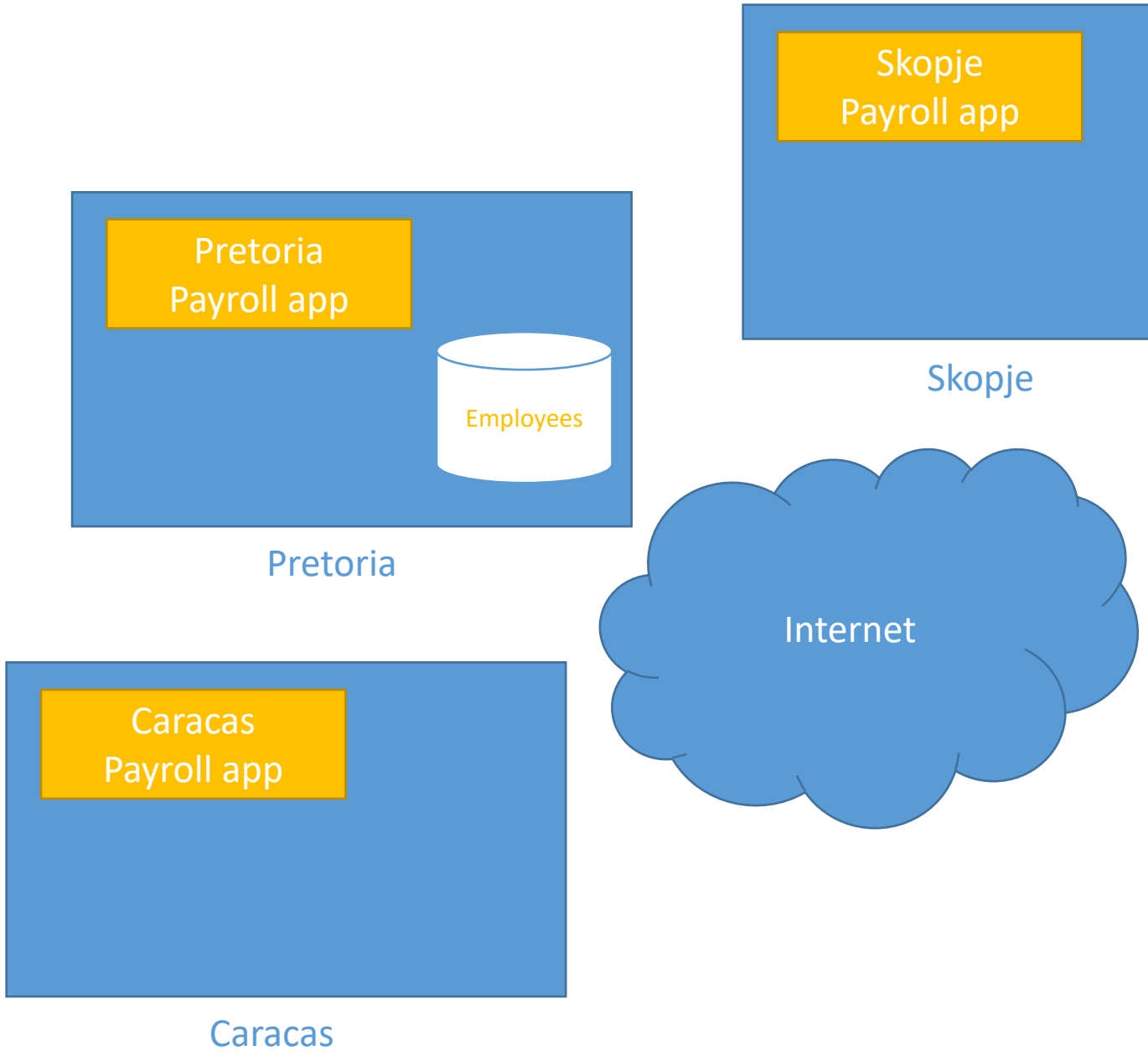
* properties:

- distributed data independence (extension of the physical and logical data independence principles):
  - users can write queries without knowing / specifying the actual location of the data
  - cost-based query optimization that takes into account communication costs & differences in computation costs across sites
- distributed transaction atomicity
  - users can write transactions accessing multiple sites just as they would write local transactions
  - transactions are still atomic (if the transaction commits, all its changes persist; if it aborts, none of its changes persist)

Distributed Databases - Motivating Example

- company with offices in Pretoria, Skopje, Caracas
- in general, an employee's data is managed from the office where the employee works (payroll, benefits, hiring data, etc.)
  - for instance, data about Skopje employees is managed from the Skopje office
- periodically, the company needs access to all employees' data, e.g.:
  - compute the total payroll expenses
  - compute the annual bonus
- where should we store employee data?

Distributed Databases - Motivating Example



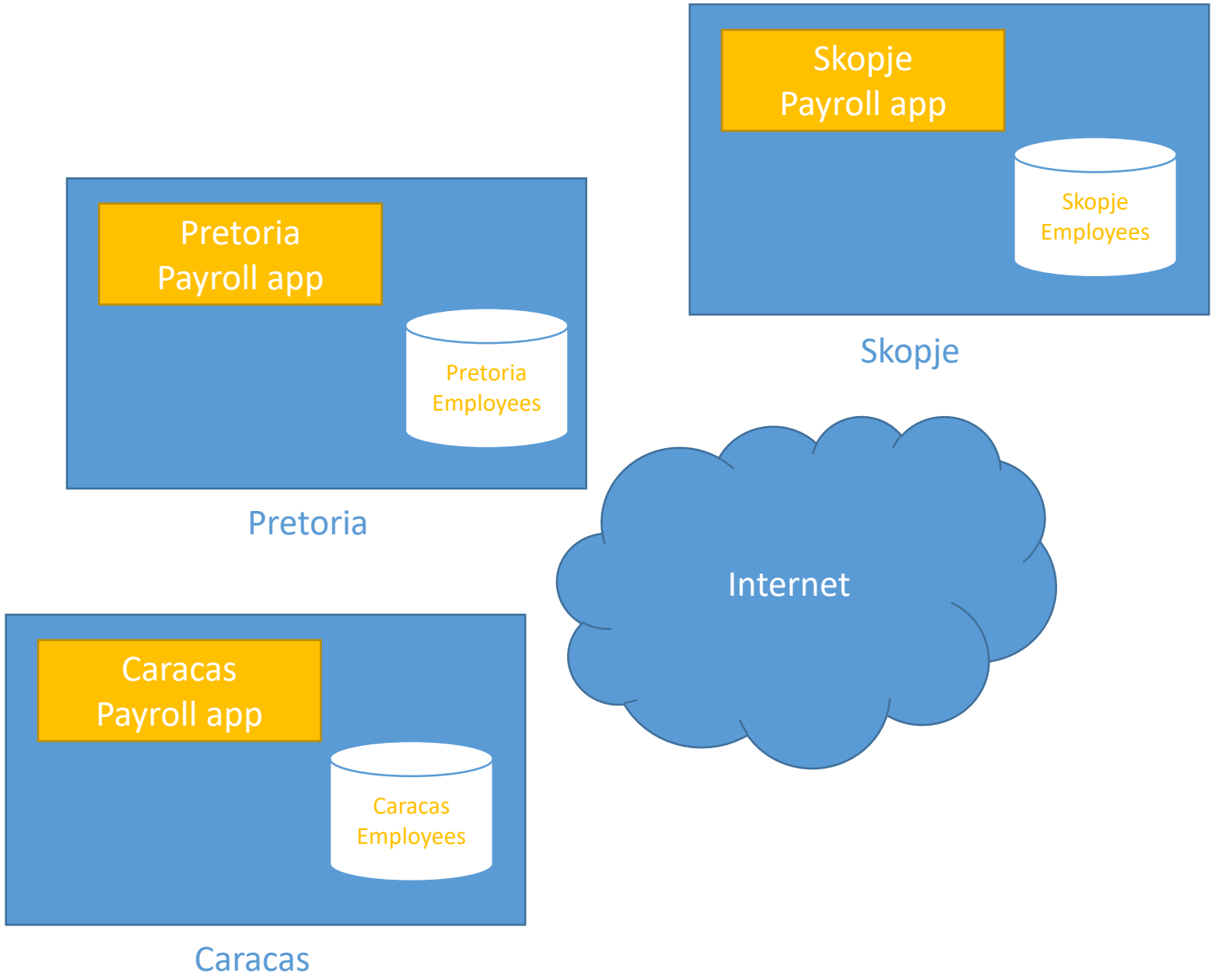
- *SiteX* payroll app – computing payrolls for *SiteX* employees

- suppose the DB is stored at the company's headquarters in Pretoria

=> slow-running payroll apps in Caracas and Skopje

- moreover, if the Pretoria site becomes unavailable, payroll apps in Caracas and Skopje cannot access the required data

Distributed Databases - Motivating Example

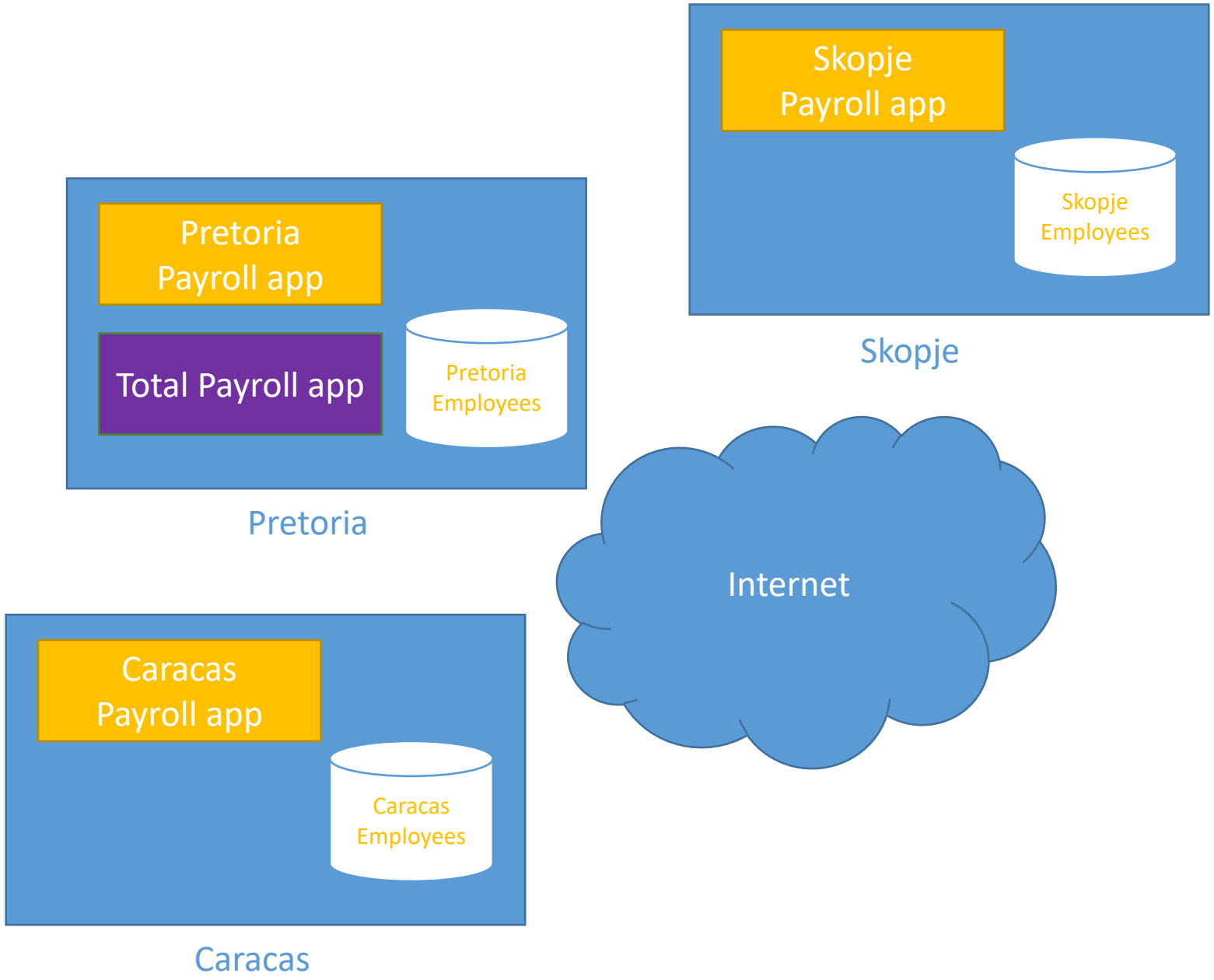


- store data about Pretoria employees in Pretoria, about Skopje employees in Skopje, etc.

=> improved performance for payroll apps in Caracas and Skopje

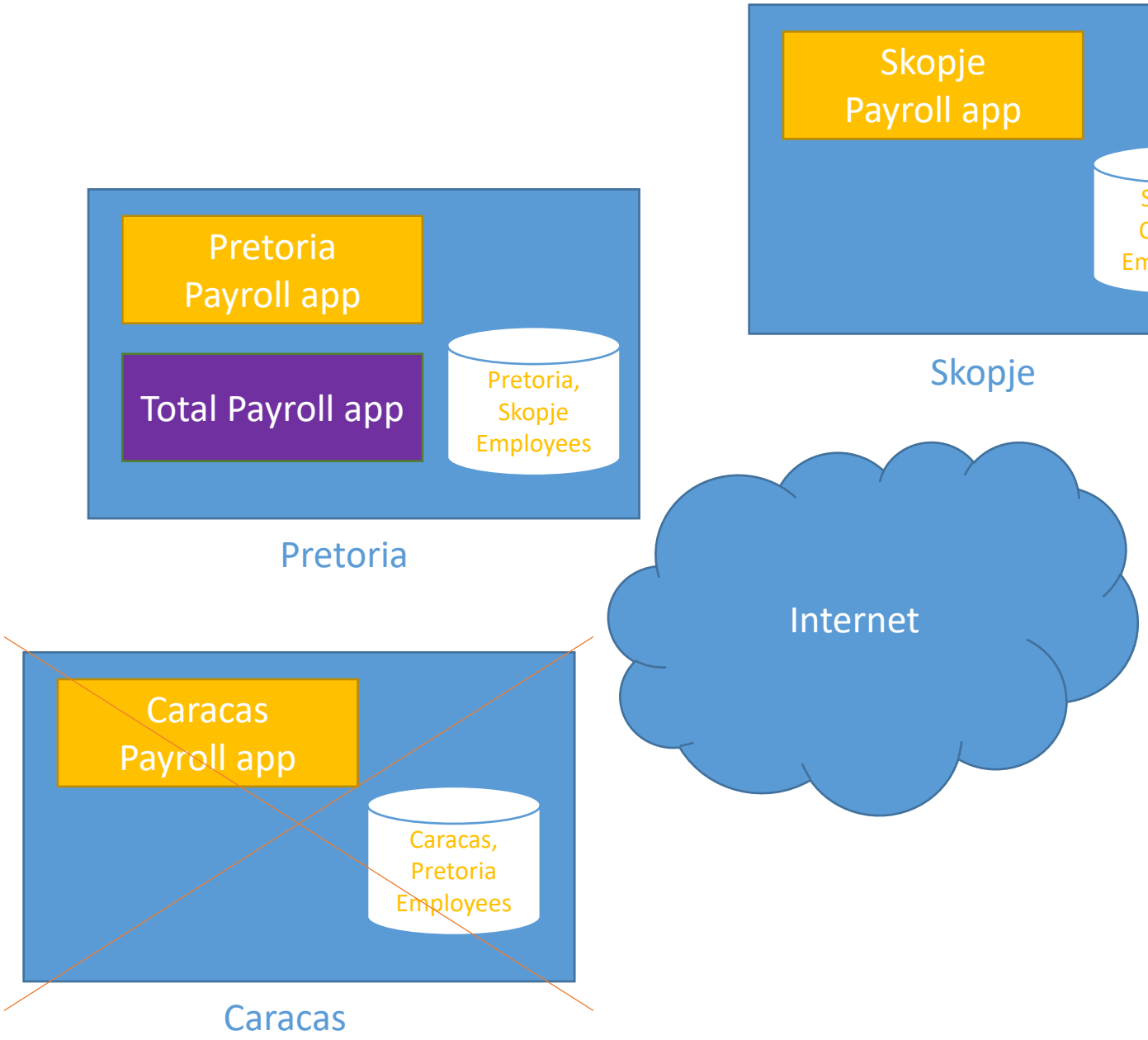
- if the Pretoria site becomes unavailable, payroll apps in Caracas and Skopje are still operational (as these apps only need data about Caracas employees and about Skopje employees, respectively)

Distributed Databases - Motivating Example



- *Total Payroll app* (at the company's headquarters in Pretoria) computing the total payroll expenses
- this app needs to access employee data at all 3 sites, so generating the final results will last a little longer
- if the Caracas site crashes, the *Total Payroll app* cannot access all required data
- opportunities for parallel execution

Distributed Databases - Motivating Example



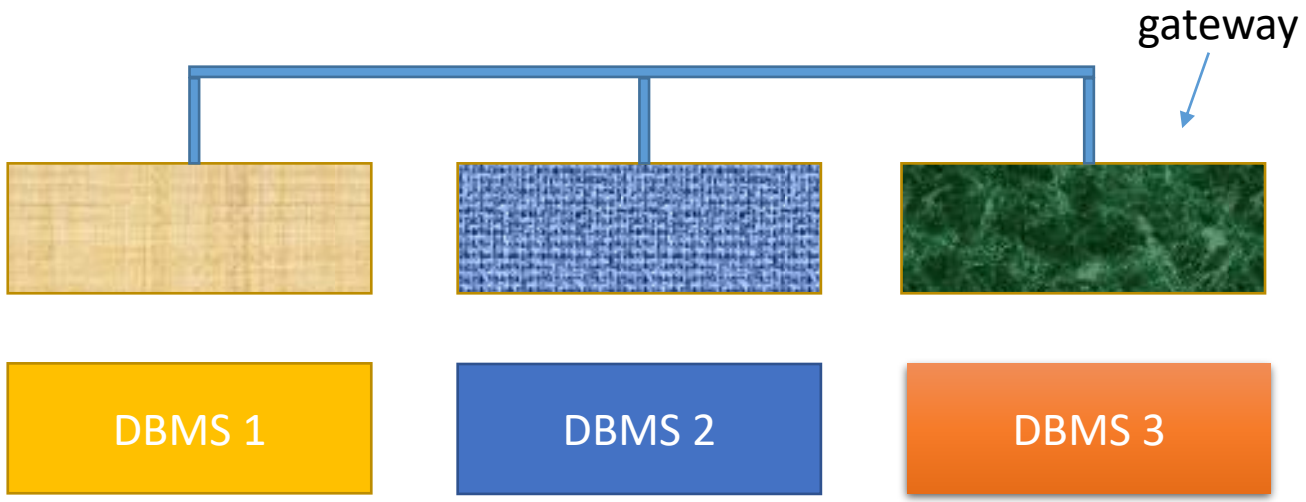
- data replication: at *Site X*, store data about *Site X* employees and data about employees from a different site, e.g., store data about Pretoria employees and Skopje employees in Pretoria (a copy of the Skopje employees data, which is stored in Skopje)
- if the Caracas site crashes, the *Total Payroll app* can continue to work, as it can access all required data, including data about the Caracas employees (a copy of this data being store in Skopje)

Types of Distributed Databases

- homogeneous:
  - the same DBMS software at every site
- heterogeneous (multidatabase system):
  - different DBMSs at different sites

* *gateway protocols*:

- mask differences between different database servers



Distributed Databases - Challenges

* distributed database design:

- deciding where to store the data
- depends on the data access patterns of the most important applications (how are they accessing the data)
- two sub-problems: *fragmentation* and *allocation*

* distributed query processing:

- centralized query plan
  - objective: minimize the number of disk I/Os; execute the query as fast as possible
- distributed context – there are additional factors to consider:
  - communication costs
  - opportunity for parallelism

=> the space of possible query plans for a given query is much larger!

Distributed Databases - Challenges

* distributed concurrency control:

- serializability
- distributed deadlock management (e.g., deadlock involving transactions T1 and T2, with T1 executing at site S1, and T2 executing at site S2)

* reliability of distributed databases:

- transaction failures
  - one or more processors may fail
  - the network may fail
- data must be synchronized

Storing Data in a Distributed DBMS

- accessing relations at remote sites => communication costs
  * example:
  - Pretoria: the site stores the *Employees* relation, holding data about all employees in the company
  - Skopje: a local manager wants to obtain the average salary for Skopje employees; she must access the relation stored in Pretoria
- reduce such costs: *fragmentation / replication*
  - a relation can be partitioned into *fragments*, which are stored across several sites (a fragment is kept where it's most often accessed)
    * example:
    - partition the *Employees* relation into fragments *PretoriaEmployees*, *SkopjeEmployees*, etc.
    - store fragment *PretoriaEmployees* in Pretoria, fragment *SkopjeEmployees* in Skopje, etc.

Storing Data in a Distributed DBMS

- accessing relations at remote sites => communication costs
- reduce such costs: *fragmentation / replication*
  - a relation can be *replicated* at each site where it's needed the most

    * example:
      - suppose the *Employees* relation is frequently needed in Beijing, New York, and Bucharest
      - *Employees* can be replicated at the Bucharest site, the New York one, and in Beijing

Storing Data in a Distributed DBMS

* *fragmentation*: break a relation into smaller relations (fragments); store the fragments instead of the relation itself

- *horizontal / vertical / hybrid*

R
1, Radu, 250, Eroilor 2, Ana, 200, Napoca 3, Ionel, 150, Motilor 4, Maria, 400, Eroilor 5, Andi, 600, Napoca 6, Calin, 250, Eroilor 7, Iulia, 350, Motilor

* example – relation Accounts(accnum, name, balance, branch):

- <u>horizontal fragmentation</u>
  - fragment: subset of rows
  - n selection predicates => n fragments (n record sets)
  - horizontal fragments should be disjoint
  - reconstruct the original relation: take the union of the horizontal fragments
    - $\sigma_{branch='Eroilor'}(Accounts)$, $\sigma_{branch='Napoca'}(Accounts)$, $\sigma_{branch='Motilor'}(Accounts)$ =>

R1	1, Radu, 250, Eroilor 4, Maria, 400, Eroilor 6, Calin, 250, Eroilor
R2	2, Ana, 200, Napoca 5, Andi, 600, Napoca
R3	3, Ionel, 150, Motilor 7, Iulia, 350, Motilor

Storing Data in a Distributed DBMS

- vertical fragmentation
  - fragment: subset of columns
  - performed using projection operators
    - must obtain a good decomposition*
    - reconstruction operator: natural join
      - $\pi_{\{\underline{accnum}, name\}}(Accounts)$
        $\pi_{\{\underline{accnum}, balance, branch\}}(Accounts)$

R1	R2
1, Radu 2, Ana 3, Ionel 4, Maria 5, Andi 6, Calin 7, Iulia	1, 250, Eroilor 2, 200, Napoca 3, 150, Motilor 4, 400, Eroilor 5, 600, Napoca 6, 250, Eroilor 7, 350, Motilor

- hybrid fragmentation
  - horizontal fragmentation + vertical fragmentation

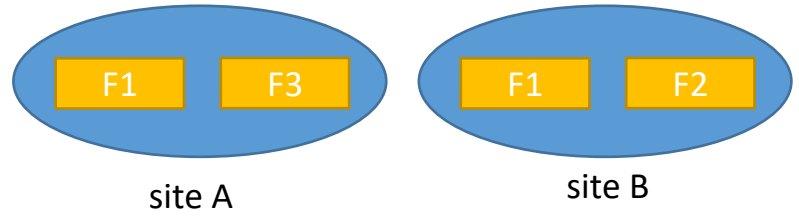
R1	R2
1, Radu 2, Ana 3, Ionel 6, Calin	1, 250, Eroilor 2, 200, Napoca 3, 150, Motilor 6, 250, Eroilor
R3	R4
4, Maria 5, Andi 7, Iulia	4, 400, Eroilor 5, 600, Napoca 7, 350, Motilor

* see *Databases – Normal Forms*

Storing Data in a Distributed DBMS

* *replication*: store multiple copies of a relation or of a relation fragment; an entire relation or one or several fragments of a relation can be replicated at one or several sites

* example: relation R is partitioned into fragments F1, F2, F3; fragment F1 is stored at site A and at site B:



- motivation:
  - increased availability of data: query Q uses fragment F1 from site A; if site A goes down or a communication link fails, the query can use another active server (e.g., site B)
  - faster query evaluation: can use a local copy of the data to avoid communication costs, e.g., query Q at site A can use the local copy of F1, it doesn't need to access the copy of F1 from site B
- types of replication: synchronous versus asynchronous
  - how are the copies of the data kept current when the relation is changed

Updating Distributed Data

- synchronous replication:
  - transaction T modifies relation R
  - before T commits, it synchronizes R’s copies

=> data distribution is transparent to the user

- asynchronous replication:
  - transaction T modifies relation R
  - R's copies are synchronized periodically, i.e., it's possible that some of R's copies are outdated for brief periods of time
  - a transaction T2 reading 2 different copies of R may see different data; but in the end, all copies of R will be synchronized

=> users must be aware of the fact that the data is distributed, i.e., distributed data independence is compromised

  - a lot of current systems are using this approach

Updating Distributed Data – Synchronous Replication

- 2 basic techniques: *voting* and *read-any write-all*

* *voting*

- to modify object O, a transaction T1 must write a majority of its copies
- when reading O, a transaction T2 must read enough copies to make sure it's seeing at least one current copy
- e.g., O has 10 copies; T1 changes O: suppose T1 writes 7 copies of O; T2 reads O: it should read at least 4 copies to make sure one of them is current
- each copy has a version number (the copy that is current has the highest version number)
- not an attractive approach in most cases, because reads are usually much more common than writes (and reads are expensive in this approach)

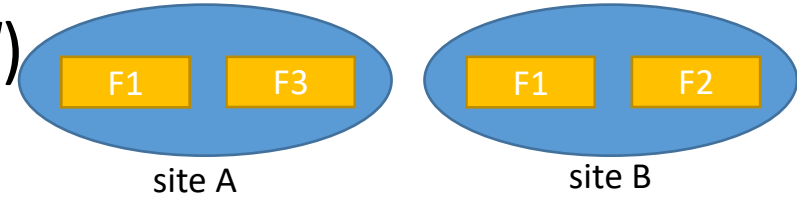
Updating Distributed Data – Synchronous Replication

* *read-any write-all*

- transaction T1 modifies O: T1 must write all copies of O
- transaction T2 reads O: T2 can read any copy of O (no matter which copy T2 reads, it will see current data, as T1 wrote all copies of O)
- fast reads (only one copy is read), slower writes (compared with the voting technique)
- most common approach to synchronous replication

Updating Distributed Data – Synchronous Replication Costs

- before an update transaction T can commit, it must lock all copies of the modified relation / fragment (with *read-any write-all*)
  - suppose relation R is partitioned into fragments F1, F2, F3, stored at sites A and B (with F1 being stored at both sites)
  - if transaction T changes fragment F1 at site A, it must also lock the copy of F1 stored at site B!
  - such a transaction sends lock requests to remote sites; while waiting for the response, the transaction holds on to its other locks



- if there's a site or link failure, transaction T cannot commit until the network / involved sites are back up (if site B becomes unavailable in the example above, transaction T cannot commit until site B comes back up)
- even if there are no failures and locks are immediately obtained, T must follow an expensive commit protocol when committing (with several messages being exchanged) => asynchronous replication is more used

Updating Distributed Data – Asynchronous Replication

- two approaches:
  - *primary site replication*
  - *peer-to-peer replication*

  * difference: number of *updatable copies* (*master copies*)

Updating Distributed Data – Asynchronous Replication

* *peer-to-peer* replication

- several copies of an object O can be *master copies* (i.e., updatable)
- changes to a master copy are propagated to the other copies of object O
- need a conflict resolution strategy for cases when two master copies are changed in a conflicting manner:
  - e.g., master copies at site 1 and site 2; at site 1: Dana’s city is changed to Cluj-Napoca; at site 2: Dana's city is changed to Timișoara; which value is correct?
  - in general - ad hoc approaches to conflict resolution

Updating Distributed Data – Asynchronous Replication

* *peer-to-peer* replication

- best utilized when conflicts do not arise:
  - each master site owns a fragment (usually a horizontal fragment) and any 2 fragments that can be updated by different master sites are disjoint

    * example: store relation *Employees* at Skopje and Caracas
      - data about Skopje employees can only be updated in Skopje
      - data about Caracas employees can only be updated in Caracas

    => no conflicts
  - updating rights are held by only one master site at a time

Updating Distributed Data – Asynchronous Replication

** primary site* replication

- exactly one copy of an object O is chosen as the *primary* (*master*) copy; this copy is published at the *primary* site
- secondary copies of the object (copies of the relation or copies of relation fragments) can be created at other sites (*secondary* sites)
- can subscribe to the primary copy or to fragments of the primary copy
- changes to the primary copy are propagated to the secondary copies in 2 steps:
  - *capture* the changes made by committed transactions
  - *apply* these changes to secondary copies

Updating Distributed Data – Asynchronous Replication

* *primary site* replication

- capture: *log-based capture / procedural capture*
  - log-based capture:
    - the log (kept for recovery purposes) is used to generate the *Change Data Table* (CDT) structure: write log tail to stable storage -> write all log records affecting replicated relations to the CDT
    - changes of aborted transactions must be removed from the CDT
    - in the end, CDT contains only update log records of committed transactions
    - suppose committed transaction T has executed 3 UPDATE operations on (the replicated) relation *Employees*; then the CDT will include 3 update log records, describing the UPDATE operations

Updating Distributed Data – Asynchronous Replication

** primary site* replication

- capture: *log-based capture / procedural capture*
  - procedural capture
    - capture is performed through a procedure that is automatically invoked (a trigger)
    - the procedure takes a snapshot of the primary copy
    - *snapshot*: a copy of the relation

  - log-based capture:
    - smaller overhead and smaller delay, but it depends on proprietary log details

Updating Distributed Data – Asynchronous Replication

* *primary site* replication

- apply
  - applies changes collected in the Capture step (from the Change Data Table or snapshot) to the secondary copies:
    - the primary site can continuously send the CDT
    - or the secondary sites can periodically request a snapshot or (the latest portion of) the CDT from the primary site
    - each secondary site runs a copy of the Apply process

Updating Distributed Data – Asynchronous Replication

* *primary site* replication

- log-based capture + continuous apply
  - minimizes delay in propagating changes
- procedural capture + application-driven apply
  - most flexible way to process changes

Updating Distributed Data – Asynchronous Replication
* *primary site* replication

- the replica could be a view over the modified relation - optional
  - the view is incrementally updated as the relation changes

Distributed Query Processing

Researchers(RID: integer, Name: string, ImpactF: integer, Age: real)
AuthorContribution(RID: integer, PID: integer, Year: integer, Descr: string)

- Researchers
  - 1 tuple - 50 bytes
  - 1 page - 80 tuples
  - 500 pages
- AuthorContribution
  - 1 tuple - 40 bytes
  - 1 page - 100 tuples
  - 1000 pages

- estimate the cost of evaluation strategies:
  - number of I/O operations and number of pages shipped among sites, i.e., take into account communication costs
  - use t_d to denote the time to read / write a page from / to disk
  - use t_s to denote the time to ship a page from one site to another (e.g., from Skopje to Caracas)

Distributed Query Processing

* nonjoin queries in a distributed DBMS

- impact of fragmentation / replication on simple operations
- scanning a relation, selection, projection

- horizontal fragmentation
  - all Researchers tuples with ImpactF < 6 - stored at New York
  - all Researchers tuples with ImpactF >= 6 - stored at Lisbon

```
Q1.
SELECT R.Age
FROM Researchers R
WHERE R.ImpactF > 4 AND R.ImpactF < 10
```

  - the DBMS evaluates the query at New York and Lisbon, then takes the union of the obtained results to produce the result set for query Q1

Distributed Query Processing

* nonjoin queries in a distributed DBMS

- horizontal fragmentation

```
Q2.
SELECT AVG(R.Age)
FROM Researchers R
WHERE R.ImpactF > 4 AND R.ImpactF < 10
```

  - the DBMS computes SUM(Age) and number of Age values at New York and Lisbon; then based on this information, it computes the average age of all researchers with ImpactF in the specified range

```
Q3.
SELECT ...
FROM Researchers R
WHERE R.ImpactF > 7
```

  - in this case, the DBMS evaluates the query only at Lisbon

Distributed Query Processing

* nonjoin queries in a distributed DBMS

- vertical fragmentation
  - RID, ImpactF - stored at New York (for all researchers)
  - Name, Age - stored at Lisbon (for all researchers)
  - the DBMS adds a field that contains the id of the corresponding tuple from Researchers to both fragments and rebuilds the Researchers relation by joining the 2 fragments on the common field (the tuple-id field)
  - the DBMS then evaluates the query over the reconstructed relation

Distributed Query Processing

* nonjoin queries in a distributed DBMS

- replication
  - Researchers relation stored at both New York and Lisbon
  - then Q1, Q2, Q3 can be executed at either New York or Lisbon
  - choosing the execution site - factors to consider:
    - the cost of shipping the result to the query site (e.g., the query site could be New York, Lisbon, or a 3rd, distinct site)
    - local processing costs:
      - can vary from one site to another – check the available indexes on Researchers at New York and Lisbon

Distributed Query Processing

* join queries in a distributed DBMS
    - can be quite expensive if the relations are stored at different sites
    - Researchers stored at New York
    - AuthorContribution stored at Lisbon
    - evaluate *Researchers join AuthorContribution*

->

Distributed Query Processing

* join queries in a distributed DBMS
  - Researchers R - New York, AuthorContribution A - Lisbon, R join A

* fetch as needed
- page-oriented nested loops in New York
  - Researchers - outer relation
  - for each page in Researchers, bring in all the AuthorContribution pages from Lisbon
  - cost
    - scan Researchers: $500t_d$
    - scan AuthorContribution + ship all AuthorContribution pages (for each Researchers page): $1000(t_d + t_s)$

  => total cost: $500t_d + 500{,}000(t_d + t_s)$

Distributed Query Processing

* join queries in a distributed DBMS
  - Researchers R - New York, AuthorContribution A - Lisbon, R join A

* fetch as needed
- page-oriented nested loops in New York
  - obs. bring in all AuthorContribution pages for each Researchers tuple => much higher cost
  - optimization
    - bring in AuthorContribution pages only once from Lisbon to New York
    - cache AuthorContribution pages at New York until the join is complete

Distributed Query Processing

* join queries in a distributed DBMS
  - Researchers R - New York, AuthorContribution A - Lisbon, R join A

* fetch as needed

- page-oriented nested loops in New York
  - query not submitted at New York
    => add the cost of shipping the result to the query site
  - RID - key in Researchers
    => the result has 100,000 tuples (the number of tuples in AuthorContribution)
  - the size of a tuple in the result
    - 40 + 50 = 90 bytes
  - the number of result tuples / page
    - 4000 / 90 = 44

Distributed Query Processing

* join queries in a distributed DBMS
    - Researchers R - New York, AuthorContribution A - Lisbon, R join A

* fetch as needed

- page-oriented nested loops in New York
    - query not submitted at New York
    - number of pages necessary to hold all the result tuples
        - 100,000/44 = 2273 pages
    - the cost of shipping the result to another site (if necessary)
        - 2273 t_s
        - higher than the cost of shipping both Researchers and AuthorContribution to the site (1500 t_s)

Distributed Query Processing

* join queries in a distributed DBMS
  - Researchers R - New York, AuthorContribution A - Lisbon, R join A

* fetch as needed

- index nested loops join in New York
  - AuthorContribution - unclustered hash index on RID
  - 100,000 AuthorContribution tuples, 40,000 Researchers tuples
  - on average, a researcher has 2.5 corresponding tuples in AuthorContribution
  - for each Researchers tuple, retrieve the 2.5 corresponding tuples in AuthorContribution:
    - obtain the index page: 1.2 t_d (on average)

      +
    - read the matching records in AuthorContribution: 2.5 t_d

Distributed Query Processing

* join queries in a distributed DBMS
  - Researchers R - New York, AuthorContribution A - Lisbon, R join A

* fetch as needed

- index nested loops join in New York
  - for each Researchers tuple, retrieve the 2.5 corresponding tuples in AuthorContribution:
    => cost per Researchers tuple: $(1.2 + 2.5)t_d$
    - the pages containing these 2.5 tuples must also be shipped from Lisbon to New York

  => total cost: $500t_d + 40.000(3.7t_d + 2.5t_s)$ (there are 40.000 records in Researchers)

Distributed Query Processing

* join queries in a distributed DBMS
  - Researchers R - New York, AuthorContribution A - Lisbon, R join A

* ship to one site

- ship Researchers to Lisbon, compute the join at Lisbon
  - scan Researchers, ship it to Lisbon, save Researchers at Lisbon:
    - cost: $500(2t_d+t_s)$
  - compute *Researchers join AuthorContribution* at Lisbon
    - example: use improved version of Sort-Merge Join with cost = 3(number of R pages + number of A pages)

      SMJ cost: $3(500 + 1000) = 4500t_d$

  => total cost: $500(2t_d+t_s) + 4500t_d$

Distributed Query Processing

* join queries in a distributed DBMS
    - Researchers R - New York, AuthorContribution A - Lisbon, R join A

* ship to one site

- ship Researchers to Lisbon, compute the join at Lisbon
    - total cost: $500(2t_d+t_s) + 4500t_d$
- ship AuthorContribution to New York, compute the join at New York
    - total cost: $1000(2t_d + t_s) + 4500t_d$

Distributed Query Processing

* join queries in a distributed DBMS
  - Researchers R - New York, AuthorContribution A - Lisbon, R join A

* semijoin

- at New York:
  - project Researchers onto the join columns (RID)
  - ship the projection to Lisbon
- at Lisbon:
  - join the Researchers projection with AuthorContribution

  => the so-called *reduction of AuthorContribution with respect to Researchers*
  - ship the reduction of AuthorContribution to New York
- at New York:
  - join Researchers with the reduction of AuthorContribution

Distributed Query Processing

* join queries in a distributed DBMS
    - Researchers R - New York, AuthorContribution A - Lisbon, R join A

* semijoin
- tradeoff:
    - the cost of computing and shipping the projection
      +
    - the cost of computing and shipping the reduction

  versus
    - the cost of shipping the entire AuthorContribution relation
- very useful if there is a selection on one of the relations

Distributed Query Processing

* join queries in a distributed DBMS
  - Researchers R - New York, AuthorContribution A - Lisbon, R join A

* bloomjoin

- at New York:
  - compute a bit-vector of some size k
    - hash Researchers tuples (using the join column RID) into the range 0 to k-1
    - if some tuple hashes to *i*, set bit *i* to 1 (i from 0 to k-1)
      - otherwise (no tuple hashes to *i*), set bit *i* to 0
    - ship the bit-vector to Lisbon
- at Lisbon:
  - hash each AuthorContribution tuple (using the join column RID) into the range 0 to k-1, with the same hash function

Distributed Query Processing

* join queries in a distributed DBMS
    - Researchers R - New York, AuthorContribution A - Lisbon, R join A

* bloomjoin

- at Lisbon:
    - discard tuples with a hash value *i* that corresponds to a 0 bit in the Researchers bit-vector

    => *reduction of AuthorContribution with respect to Researchers*
    - ship the reduction to New York
- at New York:
    - join Researchers with the reduction

Distributed Catalog Management - optional

- keeping track of data distribution across sites
- one should be able to identify each replica of each fragment for a relation that is fragmented and replicated
- local autonomy should not be compromised
  - solution - names containing several fields:
    - global relation name:
      - <local-name, birth-site>
    - global replica name:
      - <local-name, birth-site, replica_id>

Distributed Catalog Management - optional

- centralized system catalog
  - stored at a single site
  - contains data about all the relations, fragments, replicas
  - vulnerable to single-site failures
  - can overload the server

- global system catalog maintained at each site
  - every copy of the catalog describes all the data
  - not vulnerable to single-site failures (the data can be obtained from a different site)
  - local autonomy is compromised:
    - changes to a local catalog must be propagated to all the other sites

Distributed Catalog Management - optional

- local catalog maintained at each site
  - each site keeps a catalog that describes local data, i.e., copies of data stored at the site
  - the catalog at the birth-site for a relation keeps track of all the fragments / replicas of the relation
  - create a new replica / move a replica to another site:
    - must update the catalog at the birth-site
  - not vulnerable to single-site failures & doesn't compromise local autonomy

Distributed Transaction Management

- a transaction submitted at a site S could ask for data stored at several other sites
- *subtransaction* - the activity of a transaction at a given site
- context: Strict 2PL with deadlock detection
- problems:
  - distributed concurrency control
    - lock management when objects are stored across several sites
    - deadlock detection
  - distributed recovery
    - transaction atomicity
      - all the effects of a committed transaction (across all the sites it executes at) are permanent
      - none of the actions of an aborted transaction are allowed to persist

Distributed Transaction Management

- distributed concurrency control
  - lock management
    - techniques – synchronous / asynchronous replication
      - which objects will be locked
    - concurrency control protocols
      - when are locks acquired / released
    - lock management
      - *centralized*
      - *primary copy*
      - *fully distributed*

Distributed Transaction Management

- distributed concurrency control
  - lock management
    - centralized:
      - one site does all the locking for all the objects
      - vulnerable to single-site failures
    - primary copy:
      - object O, primary copy PC of O stored at site S with lock manager L
      - all requests to lock / unlock some copy of O are handled by L
      - not vulnerable to single-site failures
      - read some copy C of O stored at site S2:
        => communicate with both S and S2

Distributed Transaction Management

- distributed concurrency control
  - lock management
    - fully distributed:
      - object O, some copy C of O stored at site S with Lock Manager L
        - requests to lock / unlock C are handled by L (the site where the copy is stored)
        - one doesn't need to access 2 sites when reading some copy of O

Distributed Transaction Management

- distributed concurrency control
  - detect and resolve deadlocks
  - each site maintains a local waits-for graph
  - a cycle in such a graph indicates a deadlock
  - but a global deadlock can exist even if none of the local graphs contains a cycle

->

Distributed Transaction Management

- distributed concurrency control – distributed deadlock
  - e.g., using *read-any write-all*

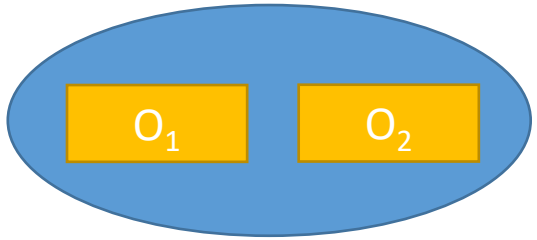
- T_1 wants to read O_1 and write O_2
- T_2 wants to read O_2 and write O_1

- T_1 acquires a S lock on O_1 and an X lock on O_2 at site S1
- T_2 obtains a S lock on O_2 and an X lock on O_1 at site S2

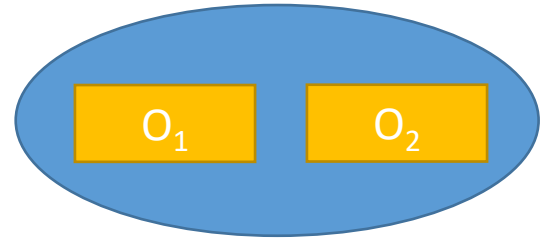
- T_1 asks for an X lock on O_2 at site S2
- T_2 asks for an X lock on O_1 at site S1

Distributed Transaction Management

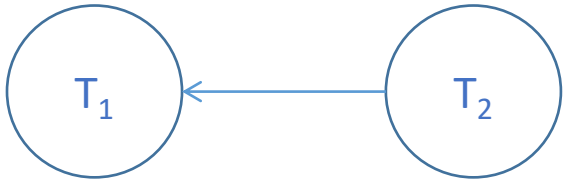
- distributed concurrency control – distributed deadlock
  - e.g., using *read-any write-all*



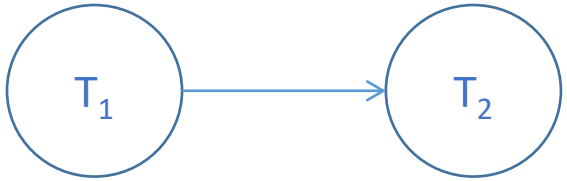
site S1



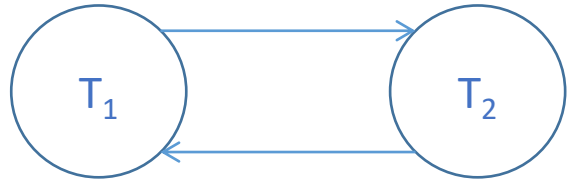
site S2



at site S1



at site S2



global waits-for graph

Distributed Transaction Management

- distributed concurrency control – distributed deadlock
  - distributed deadlock detection algorithms
    - *centralized*
    - *hierarchical*
    - based on a *timeout* mechanism

Distributed Transaction Management

- distributed concurrency control – distributed deadlock
  - distributed deadlock detection algorithms
    - centralized:
      - all the local waits-for graphs are periodically sent to a single site S
      - S - responsible for global deadlock detection
      - the global waits-for graph is generated at site S
        - nodes
          - the union of the nodes in the local graphs
        - edges
          - there is an edge from node N1 to node N2 if such an edge exists in one of the local graphs

Distributed Transaction Management

- distributed concurrency control – distributed deadlock
  - distributed deadlock detection algorithms
    - hierarchical:
      - sites are organized into a hierarchy, e.g., grouped by city, county, country, etc.
      - each site periodically sends its local waits-for graph to its parent site
      - assumption: more deadlocks are likely across related sites
      - all the deadlocks are detected in the end

Distributed Transaction Management

- distributed concurrency control – distributed deadlock
  - distributed deadlock detection algorithms
    - hierarchical:
      - example:

RO (CJ (Cluj-Napoca, Dej, Turda), BN (Bistrita, Beclean))

Cluj-Napoca : T1 -> T2
Dej: T2 -> T3
Turda: T3 -> T4 <- T7
Bistrita: T5 -> T6
Beclean: T4 -> T7 -> T6 -> T5

CJ: T1 -> T2 -> T3 -> T4 <- T7
BN: T5 <-> T6 <- T7 <- T4 (*)

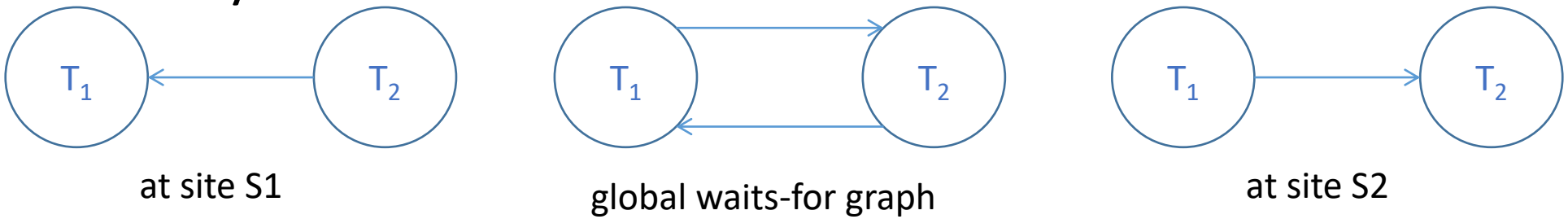
RO: T1 -> T2 -> T3 -> T4 <-> T7 -> *T6* <-> *T5*
Obs RO: T5 or T6 was aborted at (*)

Distributed Transaction Management

- distributed concurrency control – distributed deadlock
  - distributed deadlock detection algorithms
    - based on a timeout mechanism:
      - a transaction is aborted if it lasts longer than a specified interval
      - can lead to unnecessary restarts
      - however, the deadlock detection overhead is low
      - could be the only available option in a heterogeneous system (if the participating sites cannot cooperate, i.e., they cannot share their local waits-for graphs)

Distributed Transaction Management

- distributed concurrency control – distributed deadlock
- <u>phantom deadlocks</u>
  - "deadlocks" that don't exist, but are detected due to delays in propagating local information
  - lead to unnecessary aborts
  - example:



at site S1 | global waits-for graph | at site S2

  - generate local waits-for graphs, send them to the site responsible for global deadlock detection
  - T2 aborts (not because of the deadlock) => local waits-for graphs are changed, there is no cycle in the "real" global waits-for graph
  - but the built waits-for graph does have a cycle, T1 could be chosen as a victim

Distributed Transaction Management

- distributed recovery
  - more complex than in a centralized DBMS
  - new types of failure
    - network failure
    - site failure
  - commit protocol
    - either all the subtransactions of a transaction commit, or none of them does
  - normal execution
    - ensure all the necessary information is provided to recover from failures
    - a log is maintained at each site; it contains:
      - data logged in a centralized DBMS
      - actions carried out as part of the commit protocol

Distributed Transaction Management

- distributed recovery
  - transaction T
    - coordinator
      - the Transaction Manager at the site where T originated
    - subordinates
      - the Transaction Managers at the sites where T's subtransactions execute

Distributed Transaction Management

- distributed recovery
  - two-phase commit protocol (2PC)
    - exchanged messages + records written in the log
    - 2 rounds of messages, both initiated by the coordinator
      - *voting phase*
      - *termination phase*
    - any Transaction Manager can abort a transaction
    - however, for a transaction to commit, all Transaction Managers must decide to commit

Distributed Transaction Management

- distributed recovery
  - two-phase commit protocol
    - the user decides to commit transaction T

    => the commit command is sent to T's coordinator, 2PC is initiated

    1. the coordinator sends a *prepare* message to each subordinate
    2. upon receiving a *prepare* message, a subordinate decides whether to commit / abort its subtransaction
       - the subordinate force-writes an *abort* or a *prepare** log record
       - then it sends a *no* or *yes* message to the coordinator

* *prepare* log records are specific to the commit protocol, they are not used in centralized DBMSs

Distributed Transaction Management

- distributed recovery
  - two-phase commit protocol

    3.
    - if the coordinator receives *yes* messages from all subordinates:
      - it force-writes a *commit* log record
      - it then sends *commit* messages to all subordinates
    - otherwise (i.e., if it receives at least one *no* message or if it doesn't receive any message from a subordinate for a predetermined timeout interval)
      - if force-writes an *abort* log record
      - it then sends an *abort* message to each subordinate

Distributed Transaction Management

- distributed recovery
  - two-phase commit protocol

    4.
    - upon receiving an *abort* message, a subordinate:
      - force-writes an *abort* log record
      - sends an *ack* message to the coordinator
      - aborts the subtransaction
    - upon receiving a *commit* message, a subordinate:
      - force-writes a *commit* log record
      - sends an *ack* message to the coordinator
      - commits the subtransaction

Distributed Transaction Management

- distributed recovery
  - two-phase commit protocol

    5. after it receives *ack* messages from all subordinates, the coordinator writes an *end* log record for the transaction

    * obs. sending a message – the sender has made a decision
    - the message is sent only after the corresponding log record has been forced to stable storage (to ensure the corresponding decision can survive a crash)

    * obs. T is a committed transaction if the commit log record of T's coordinator has been forced to stable storage

Distributed Transaction Management

- distributed recovery
  - two-phase commit protocol
    - log records for the commit protocol
      - record type
      - transaction id
      - coordinator's identity
    - the commit / abort log record for the coordinator also contains the identities of the subordinates

Distributed Transaction Management

- distributed recovery
  - restart after a failure – site S comes back up after a crash
    - if there is a *commit* or an *abort* log record for transaction T:
      - must redo / undo T
      - if S is T's coordinator:
        - periodically send *commit* / *abort* messages to subordinates until *ack* messages are received
        - write an *end* log record after receiving all *ack* messages
    - if there is a *prepare* log record for transaction T, but no *commit* / *abort*, S is one of T's subordinates
      - contact T's coordinator repeatedly until T's status is obtained
      - write a *commit* / an *abort* log record
      - redo / undo T

Distributed Transaction Management

- distributed recovery
  - restart after a failure – site S
    - if there are no *commit / abort / prepare* log records for T:
      - abort T, undo T
      - if S is T's coordinator, T's subordinates may subsequently contact S

Distributed Transaction Management

- distributed recovery
  - link and remote site failures
    - current site S, remote site R, transaction T
    - if R doesn't respond during the commit protocol for T, either because R failed or the link failed:
      - if S is T's coordinator:
        - S should abort T
      - if S is one of T's subordinates, and has not voted yet:
        - S should abort T
      - if S is one of T's subordinates and has voted *yes:*
        - S is blocked until T's coordinator responds

Distributed Transaction Management

- distributed recovery
  - 2PC – observations
    - *ack* messages
      - used to determine when can a coordinator C “forget” about a transaction T
      - C must keep T in the transaction table until it receives all *ack* messages
    - C fails after sending *prepare* messages, but before writing a *commit* / an *abort* log record
      - when C comes back up it aborts T
      - i.e., absence of information => T is presumed to have aborted
    - if a subtransaction doesn't change any data, its commit / abort status is irrelevant

Distributed Transaction Management

- distributed recovery
  - 2PC with Presumed Abort
    - coordinator C, transaction T, some subordinate S, some subtransaction t
    - C aborts T
      - T is undone
      - C immediately removes T from the Transaction Table, i.e., it doesn’t wait for *ack* messages
    - subordinates' names need not be recorded in C's abort log record
    - S doesn't send an *ack* message when it receives an *abort* message

Distributed Transaction Management

- distributed recovery
  - 2PC with Presumed Abort
    - coordinator C, transaction T, some subordinate S, some subtransaction t
    - t doesn't change any data
      - t responds to *prepare* messages with a *reader* message, instead of *yes / no*
    - C subsequently ignores readers
    - if all subtransactions are readers, the 2nd phase of the protocol is not needed

References

- [Ra02] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (3rd Edition), McGraw-Hill, 2002
- [Da03] DATE, C.J., An Introduction to Database Systems (8th Edition), Addison-Wesley, 2003
- [Ga09] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book (2nd Edition), Pearson Education, 2009
- [Ra02S] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, Slides for the 3rd Edition, http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html
- [Si11] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts (6th Edition), McGraw-Hill, 2011
- [Si19S] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, Slides for the 7th Edition, http://codex.cs.yale.edu/avi/db-book/
- [Ul11] ULLMAN, J., WIDOM, J., A First Course in Database Systems, http://infolab.stanford.edu/~ullman/fcdb.html