Database Management Systems

Lecture 11

Distributed Databases

Sabina S. CS

* centralized DB systems

- all data at a single site
- assumed 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

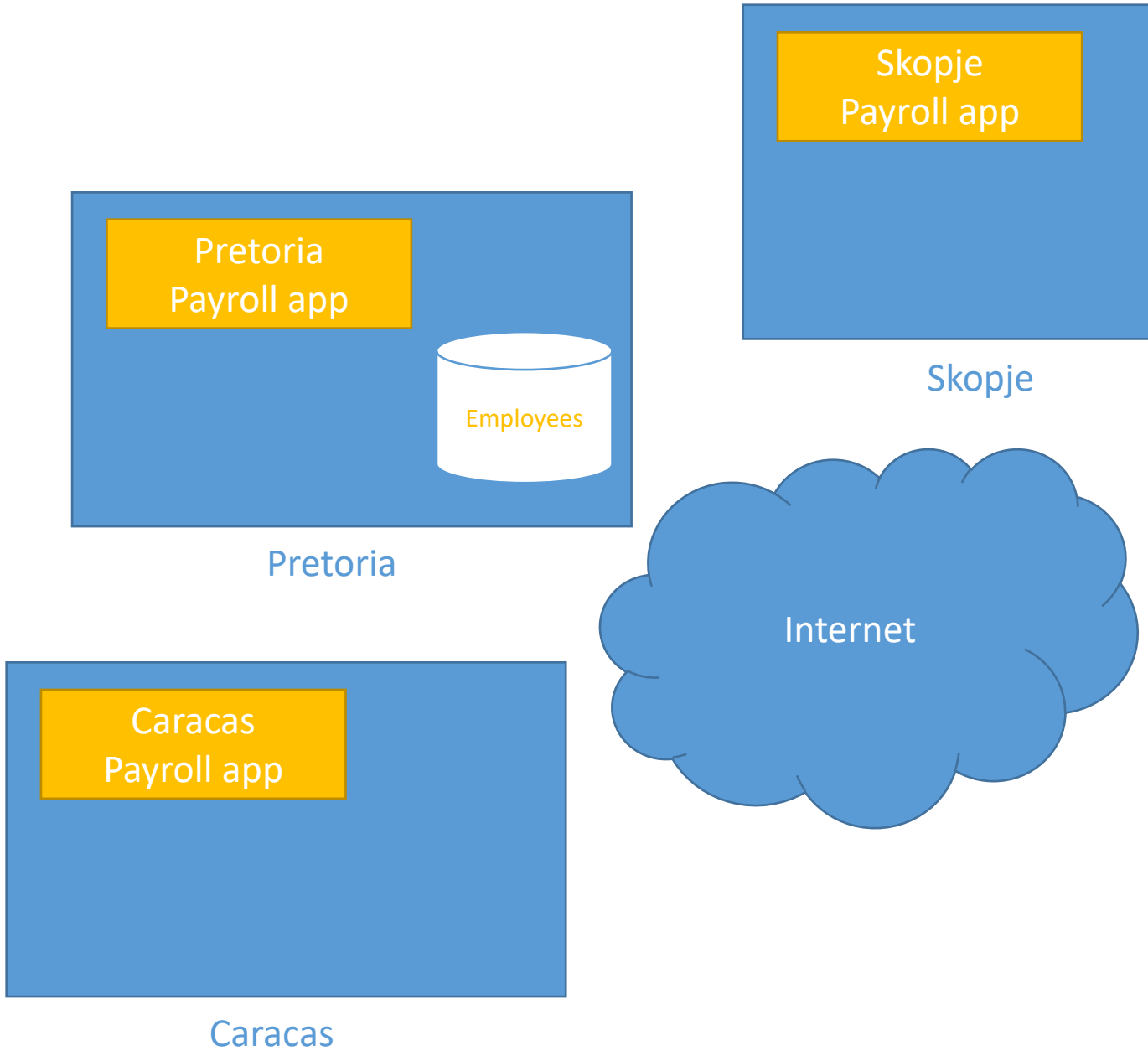
* properties:

- 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, which depends on the global net profit
- where should we store employee data?

Distributed Databases - Motivating Example



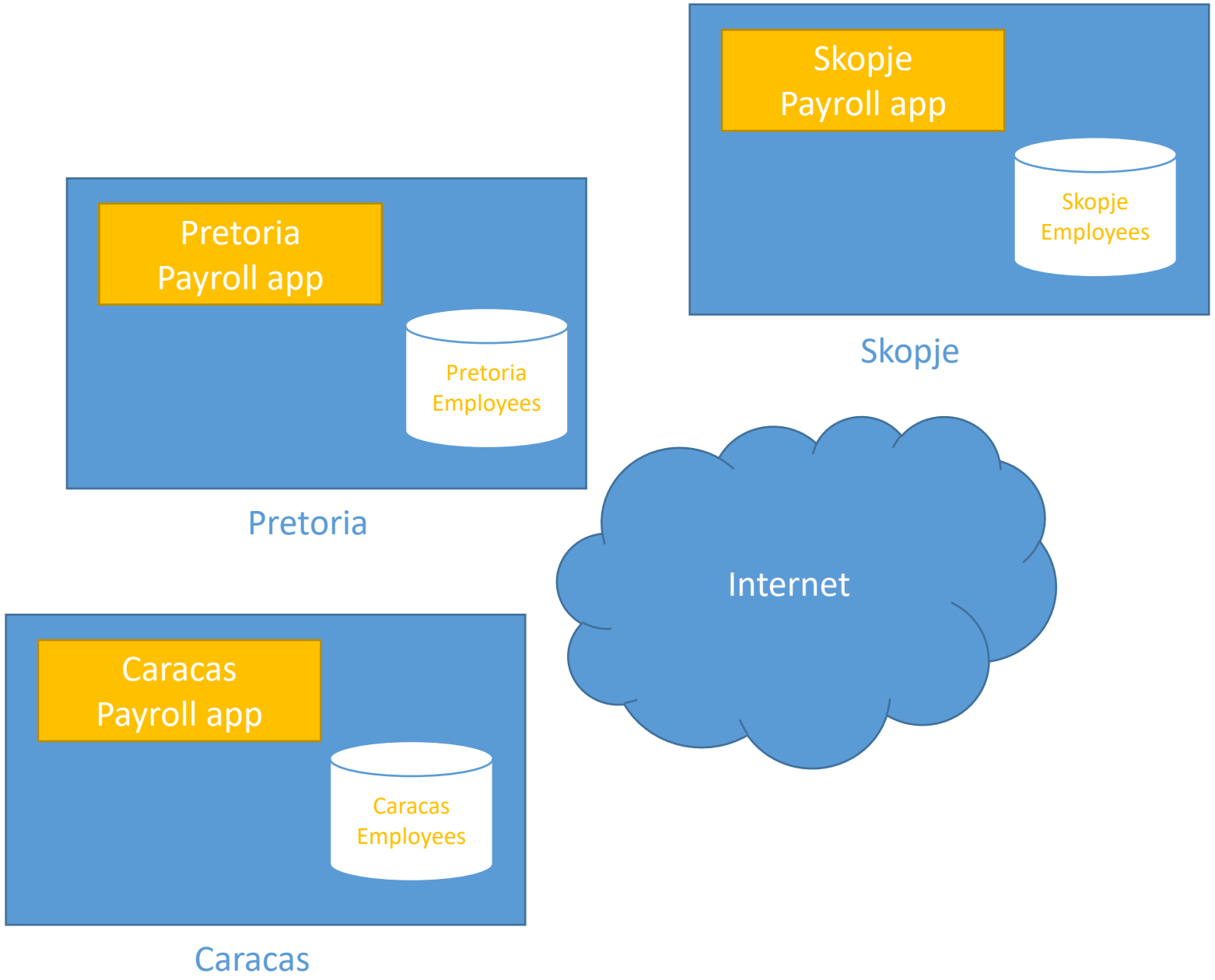
- *SiteX* payroll app – computing payrolls for *SiteX* employees

- 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

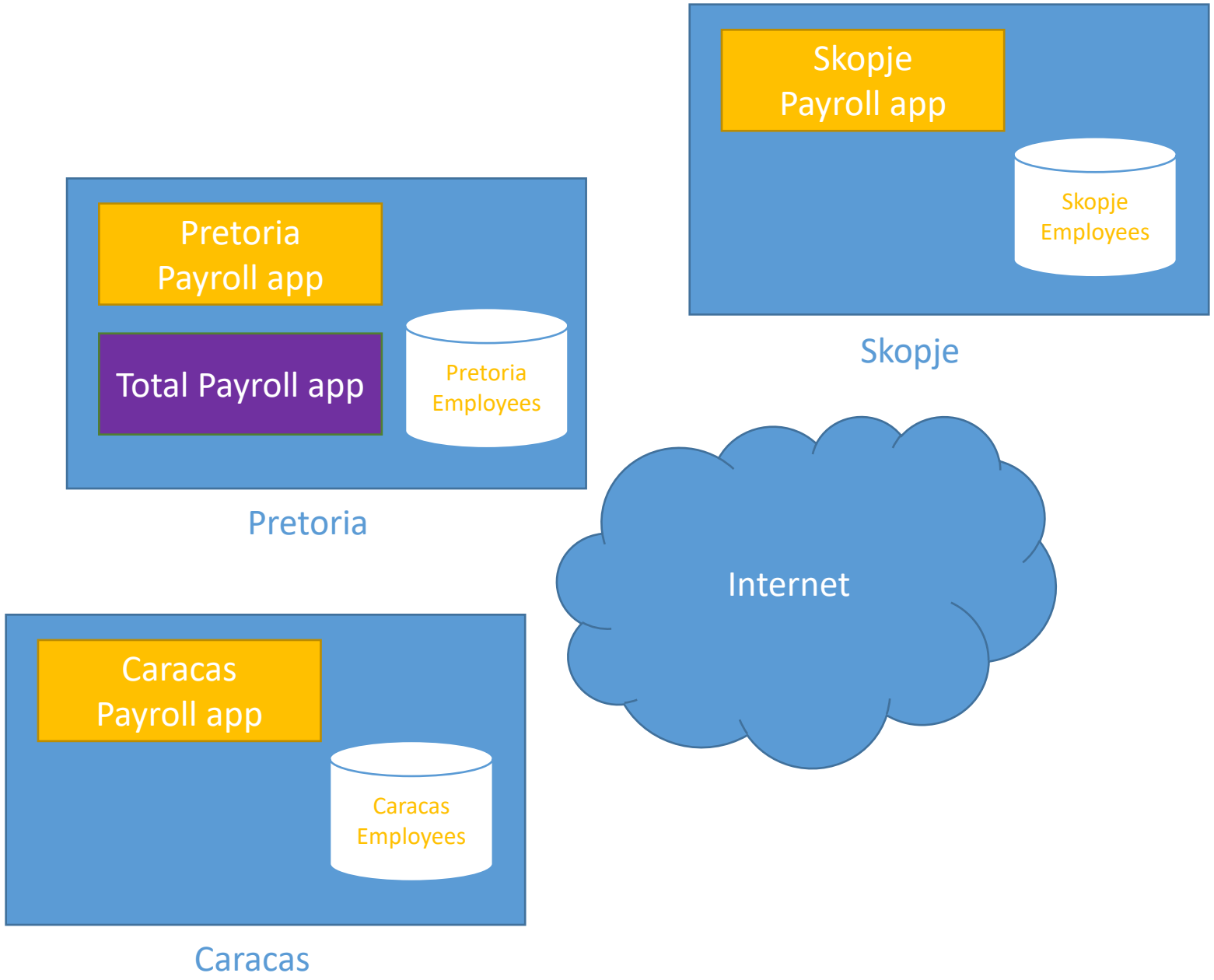


- store data about Pretoria employees in Pretoria, about Skopje employees in Skopje, etc

=> improved performance for payroll apps in Caracas and Skopje

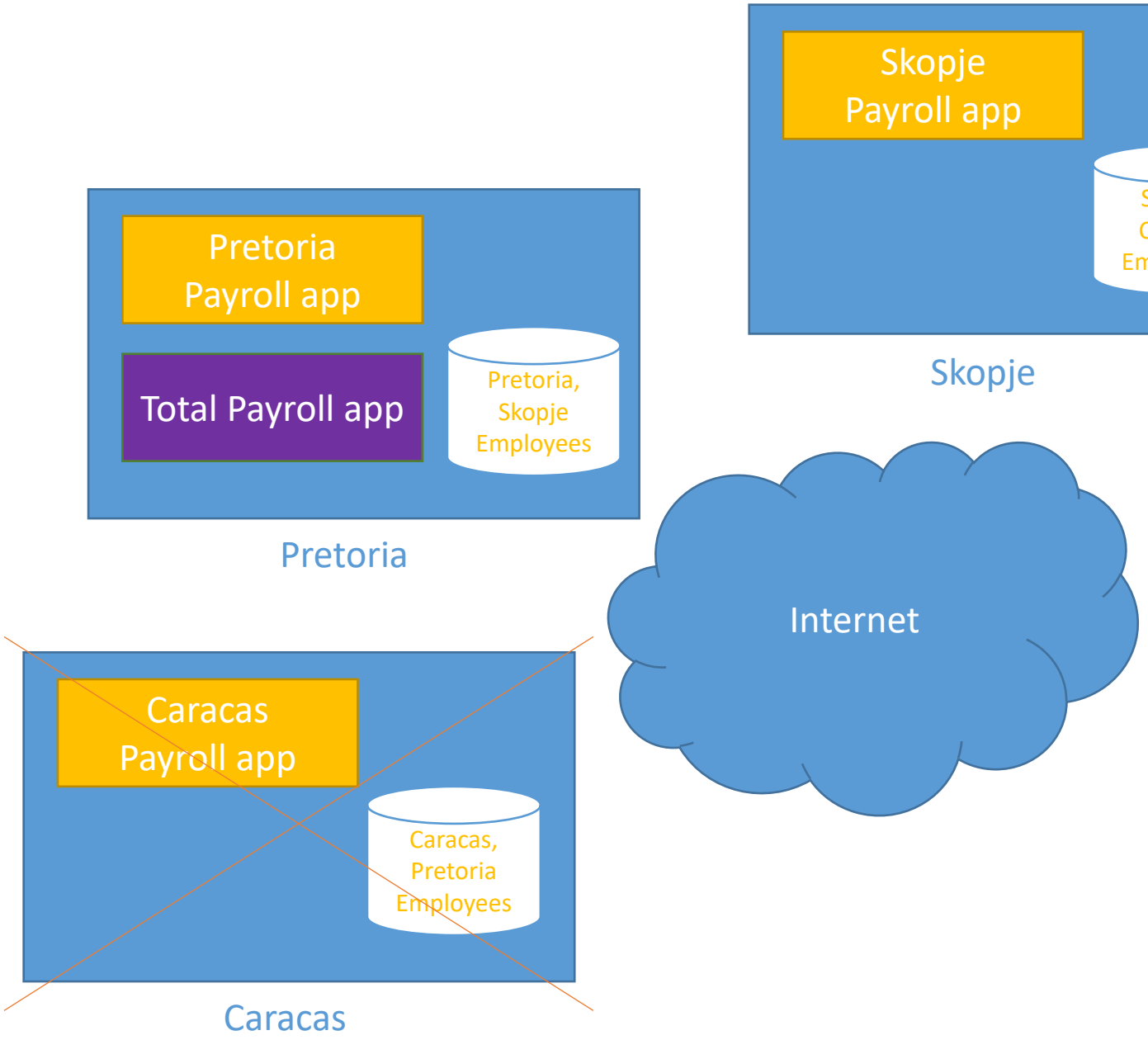
- if the Pretoria site becomes unavailable, payroll apps in Caracas and Skopje are still operational (as these apps only need data about Caracas employees and about Skopje employees, respectively)

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



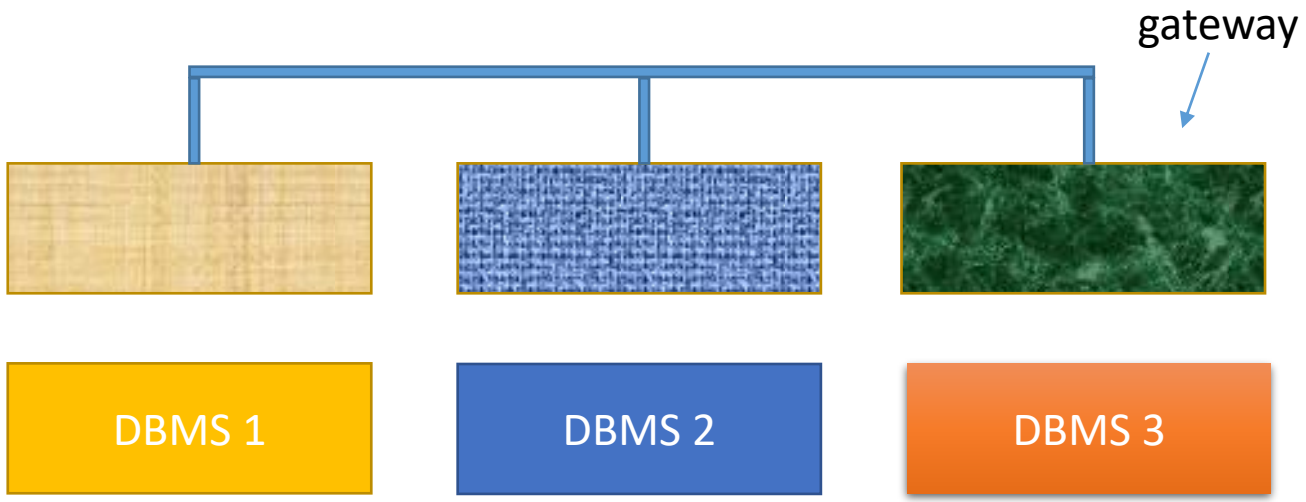
- data replication: at *Site X*, store data about *Site X* employees and data about employees from a different site, e.g., store data about Pretoria employees and Skopje employees in Pretoria (a copy of the Skopje employees data, which is stored in Skopje)
- if the Caracas site crashes, the *Total Payroll 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

* <u>distributed database design:</u>

- 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*

* <u>distributed query processing:</u>

- centralized query plan (treated in previous lectures)
  - 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

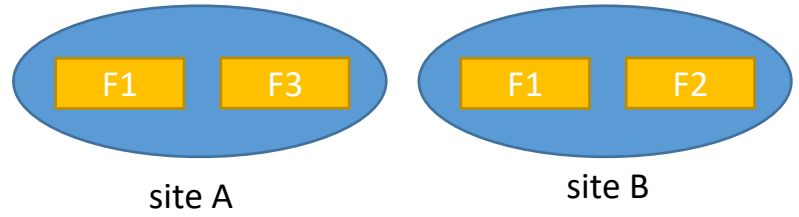
R1	R2
1, Radu 2, Ana 3, Ionel 6, Calin	1, 250, Eroilor 2, 200, Napoca 3, 150, Motilor 6, 250, Eroilor
R3	R4
4, Maria 5, Andi 7, Iulia	4, 400, Eroilor 5, 600, Napoca 7, 350, Motilor

* 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:
  - suppose relation R has 3 copies: R1, R2, R3
  - transaction T modifies relation R
  - before T commits, it synchronizes all copies of R (R1, R2, R3 and R all contain the same data)

=> data distribution is transparent to the user: when reading relation R, the user doesn't need to know which copy of R it accesses, as all copies store correct, current data

- asynchronous replication:
  - transaction T modifies relation R
  - R's copies (R1, R2, R3) 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 (say R1 and R3) may see different data; but in the end, all copies of R will be synchronized

Updating Distributed Data

- asynchronous replication:

=> 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

- transaction T, object O (with several copies)
- objective:
  - no matter which copy of O it accesses, T should see the same value
  - 2 basic techniques: *voting* and *read-any write-all*

* *voting*

- to modify 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 at least one of them is current
- each copy has a version number (the copy that is current has the highest version number)

Updating Distributed Data – Synchronous Replication

* *voting*

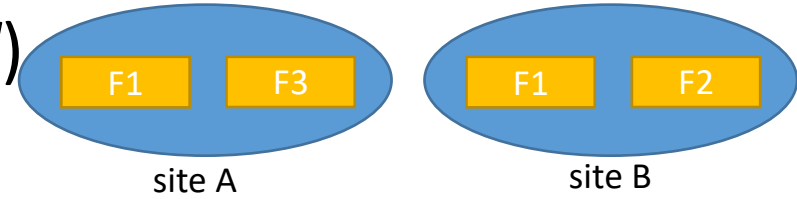
- not an attractive approach in most cases, because reads are usually much more common than writes (and reads are expensive in this approach)

* *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

- transaction T1 modifies object O; T1 is allowed to commit before all copies of O have been changed

  => users must know:
  - which copy they are reading
  - that copies may be outdated for brief periods of time
- 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 T1 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 (e.g., 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

- the replica could be a view over the modified relation
  - replication: incrementally updating the view as the relation changes

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

Distributed Query Processing

Researchers(RID: integer, Name: string, ImpactF: integer, Age: real)
AuthorContribution(RID: integer, PID: integer, Year: integer, Coord: 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
      - e.g., 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 every page in Researchers, bring in all the AuthorContribution pages from Lisbon
  - cost
    - scan Researchers: $500t_d$
    - scan AuthorContribution + ship all AuthorContribution pages (for every 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

* <u>fetch as needed</u>

- 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

* <u>fetch as needed</u>

- 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)

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, 2009
- [Ra07] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, McGraw-Hill, 2007, http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html
- [Si19] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts (7th Edition), McGraw-Hill, 2019
- [Ul11] ULLMAN, J., WIDOM, J., A First Course in Database Systems, http://infolab.stanford.edu/~ullman/fcdb.html