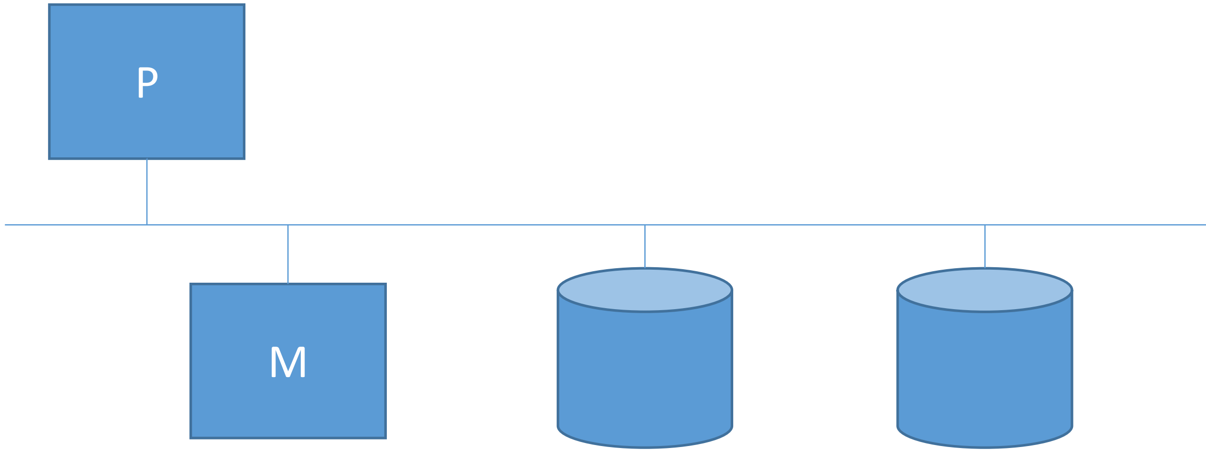
Database Management Systems

Lecture 11

Distributed Databases

- centralized DB systems
  - all data - single site
  - transaction processing - sequential
  - single front end
  - one place to manage all the locks
  - processor fails => system fails



- distributed systems
  - multiple processors (+ memories)
  - autonomous, heterogeneous components

Distributed Database Systems

- the data is stored at several sites
- each site is managed by a DBMS that can run independently of the other sites
- location of data – impact on:
  - query processing
  - query optimization
  - concurrency control
  - recovery

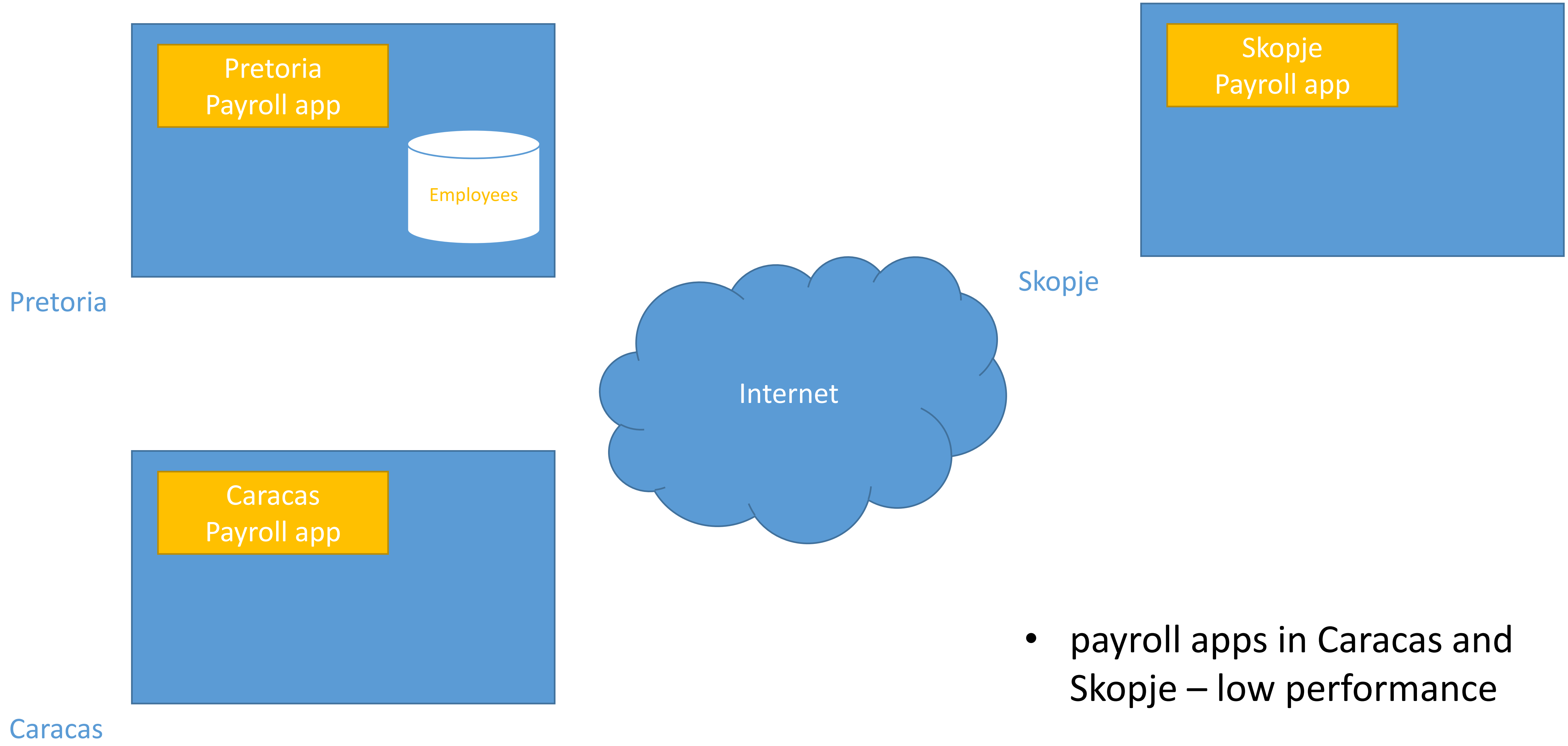
Distributed Database Systems

- impact of data distribution – transparent:
  - <u>distributed data independence</u>
    - users should be able to write queries without knowing / specifying the actual location of the data
    - an extension of the physical and logical data independence principles
    - cost-based query optimization that takes into account communication costs & differences in local computation costs
  - <u>distributed transaction atomicity</u>
    - users should be able to write transactions accessing multiple sites just as they would write local transactions
    - i.e., transactions are atomic
      - all changes persist if the transaction commits, none persist if it aborts

Distributed Databases - Motivating Example

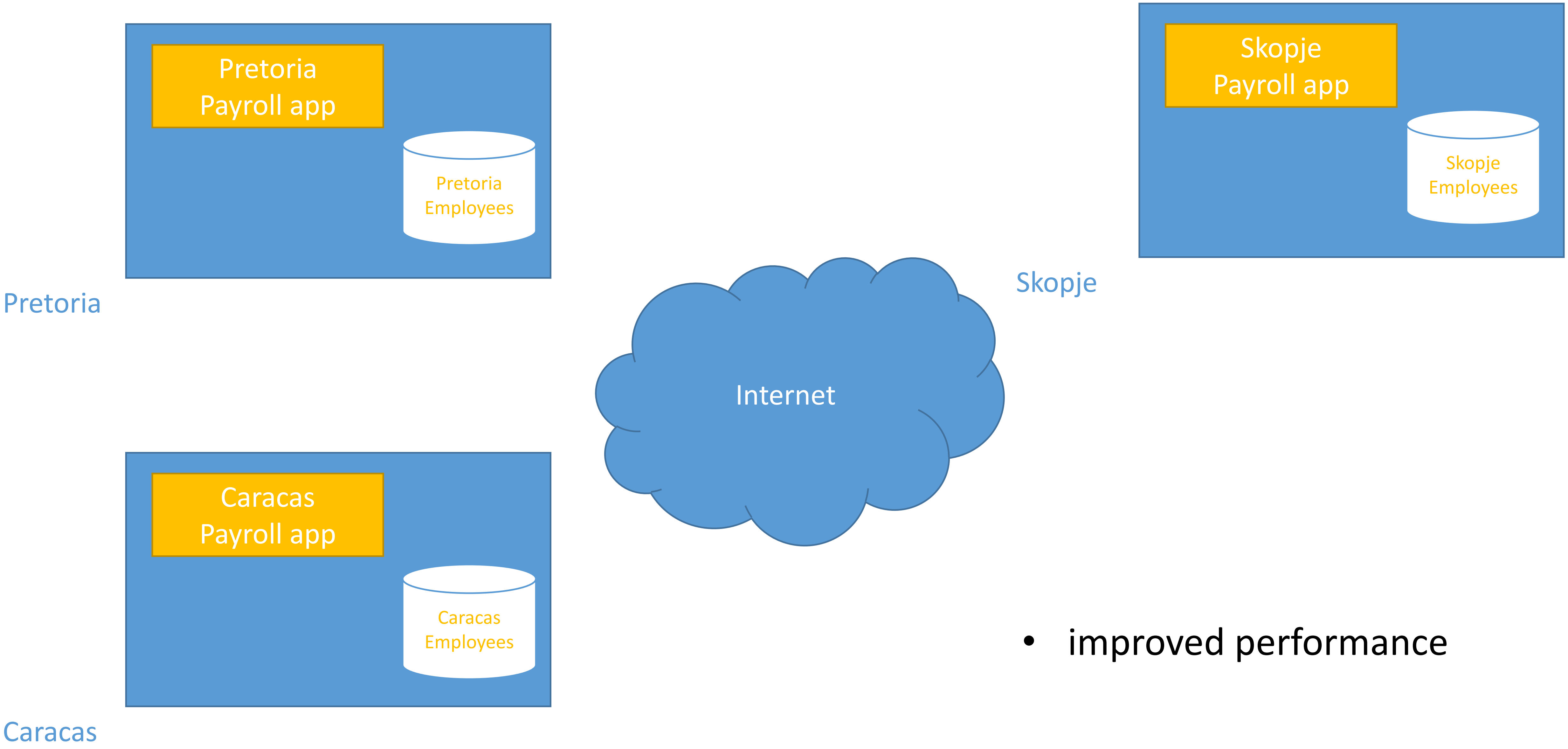
- company with offices in Pretoria, Skopje, Caracas
- in general, an employee's data is managed at the office where the employee works
    - e.g., payroll, benefits, hiring data, etc
- periodically, the company needs access to all the employees' data
    - e.g., compute the total payroll expenses for the balance sheet
    - e.g., compute the annual bonus, which depends on the global net profit
- where should we store the employee data table?

Distributed Databases - Motivating Example



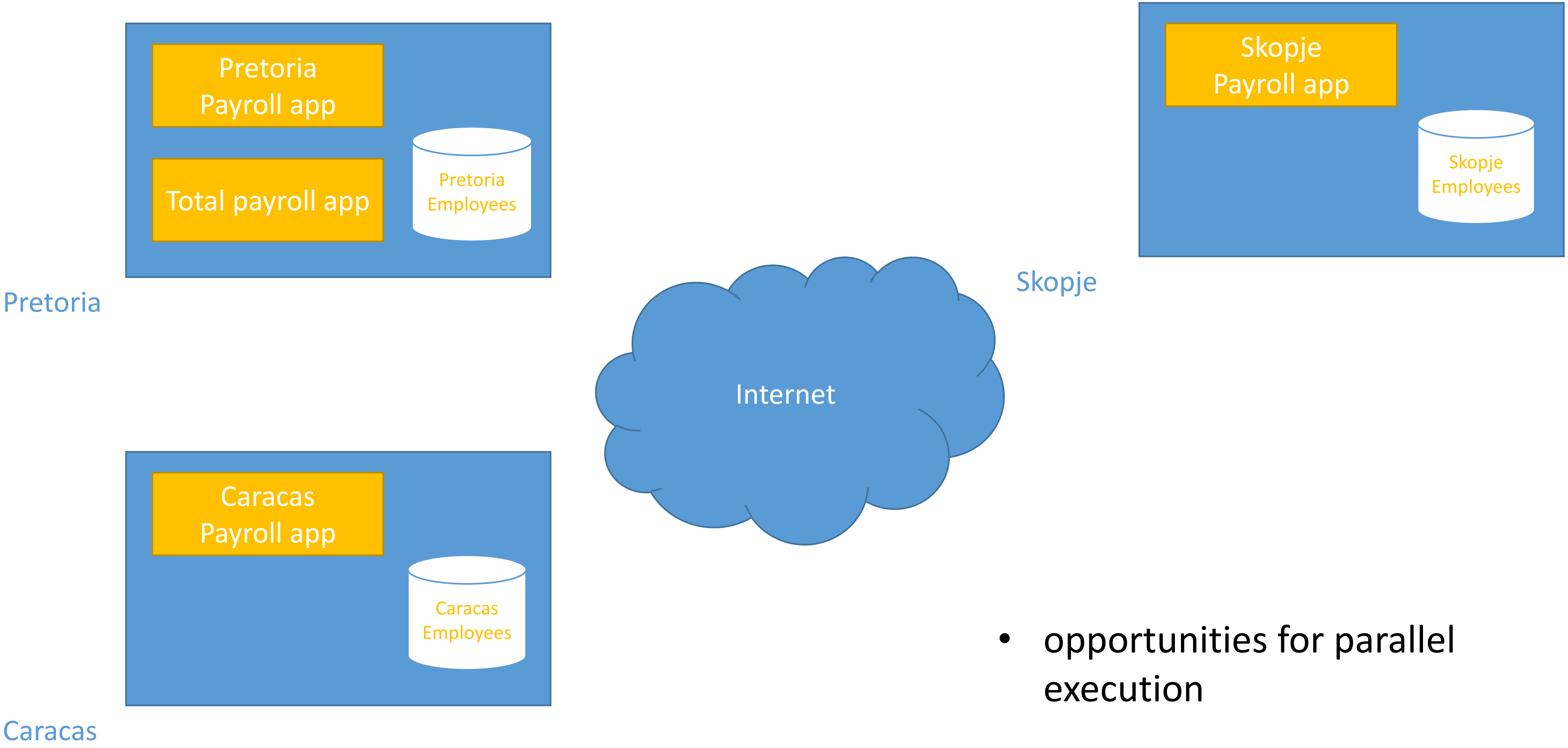
- payroll apps in Caracas and Skopje – low performance

Distributed Databases - Motivating Example



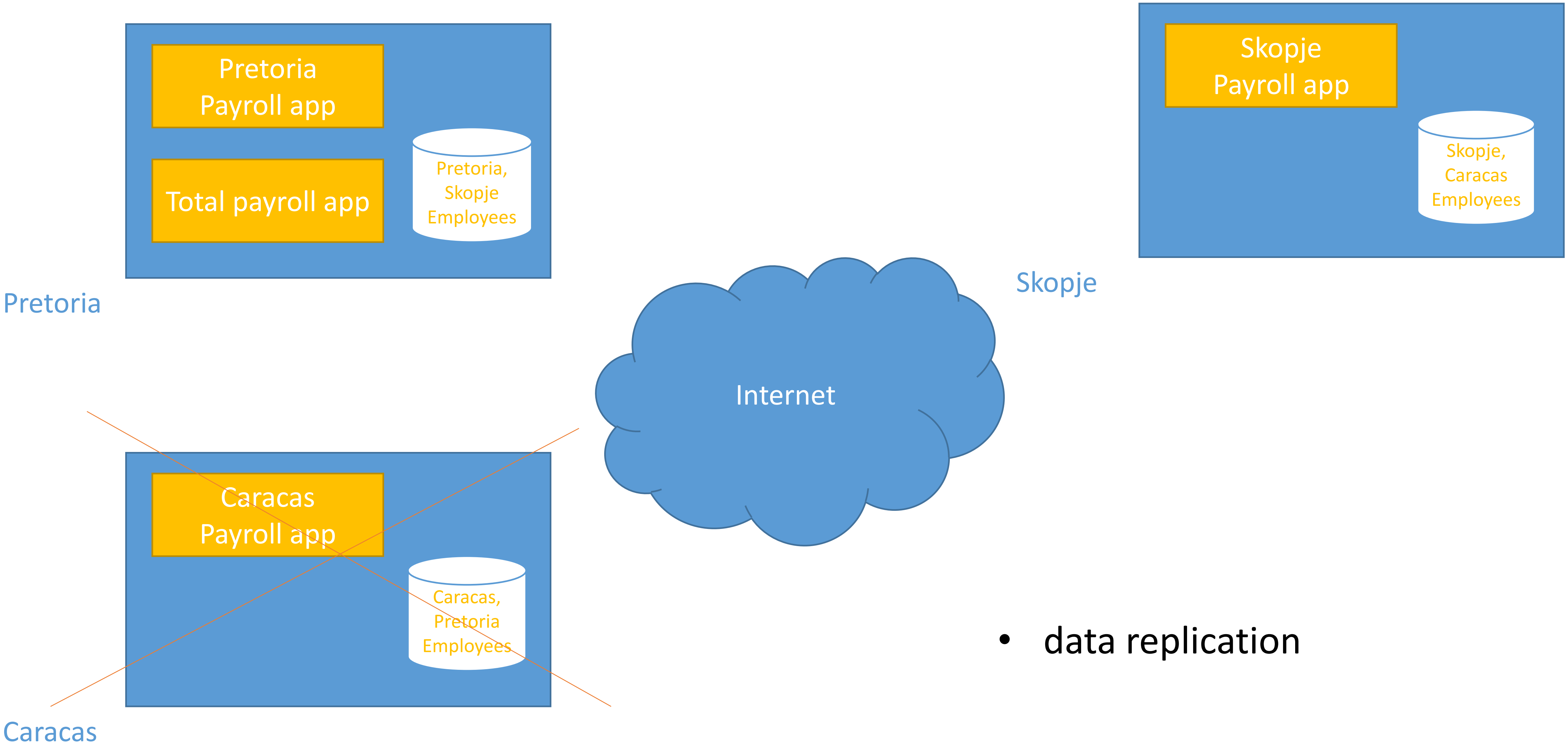
- improved performance

Distributed Databases - Motivating Example



- opportunities for parallel execution

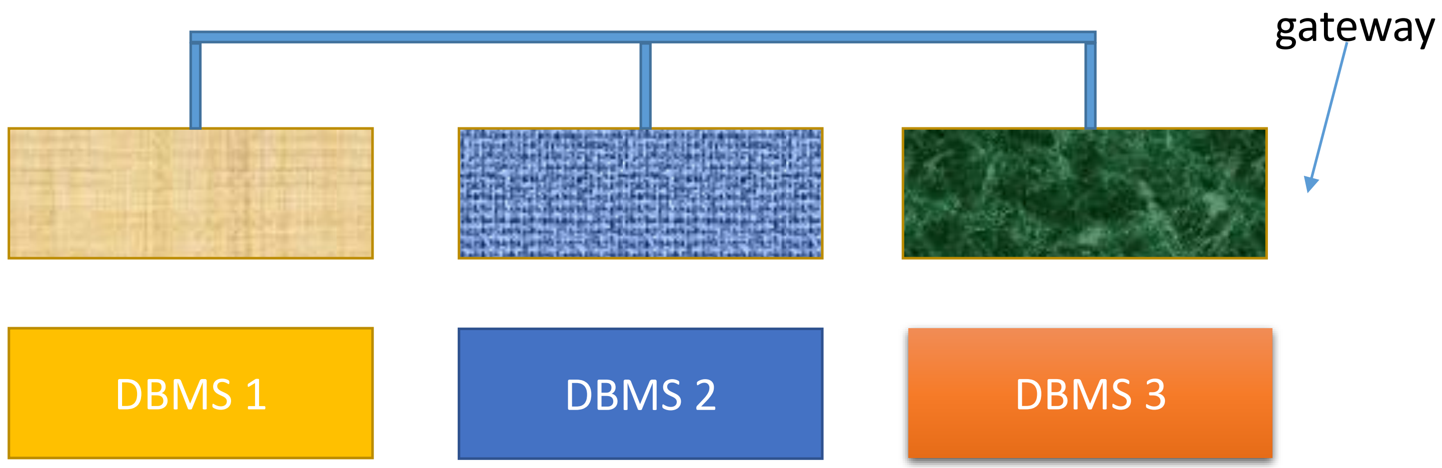
Distributed Databases - Motivating Example



- data replication

Types of Distributed Databases

- homogeneous
  - every site runs the same DBMS software
- heterogeneous (multidatabase system)
  - different sites run different DBMSs (different RDBMSs or even non-relational DBMSs)
- gateway
  - a software component that accepts requests (in some subset of SQL), submits them to local DBMSs, and returns the answers to the requestors (in some standard format)



Distributed Databases - Challenges

- distributed database design
  - deciding where to store the data
  - depends on the data access patterns of the most important applications
  - two sub-problems
    - fragmentation
    - allocation

- distributed query processing
  - centralized query plan
    - objective: minimize the number of disk I/Os
  - distributed setting - additional factors to consider:
    - communication costs
    - opportunity for parallelism

  => the space of possible query plans is much larger

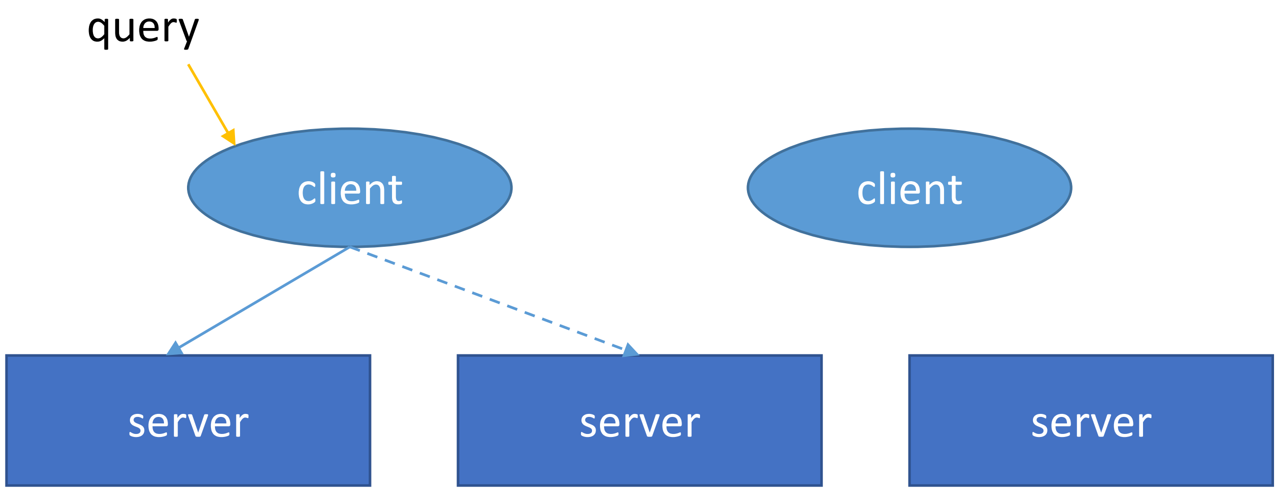
Distributed Databases - Challenges

- distributed concurrency control
  - transaction schedules must be globally serializable
  - distributed deadlock management

- reliability of distributed databases
  - transaction failures
    - one or more processors may fail
    - the network may fail
  - data must be synchronized

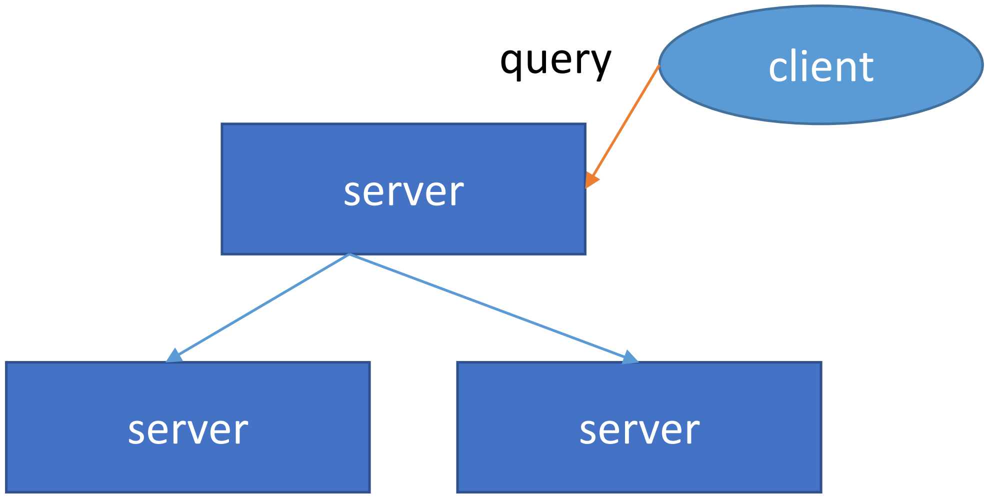
Distributed DBMS Architectures

- Client-Server Systems
  - the client submits queries to a single site
  - query processing is done at the server



Distributed DBMS Architectures

- Collaborating Server Systems
  - queries can span several sites



Storing Data

- relations stored at several sites
- accessing relations at remote sites => message-passing costs
- reduce costs:
    - fragment a relation across several sites
        - store fragments where they are most often accessed
    - replicate a relation at each site where it's needed the most

Storing Data

- fragmentation
  - break a relation into smaller relations (fragments)
  - store the fragments instead of the relation itself

- horizontal
- vertical
- hybrid

- example
  - relation Accounts(accnum, name, balance, branch)

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

Storing Data

- fragmentation
  - horizontal fragmentation
    - 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

- fragmentation
  - 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

Storing Data

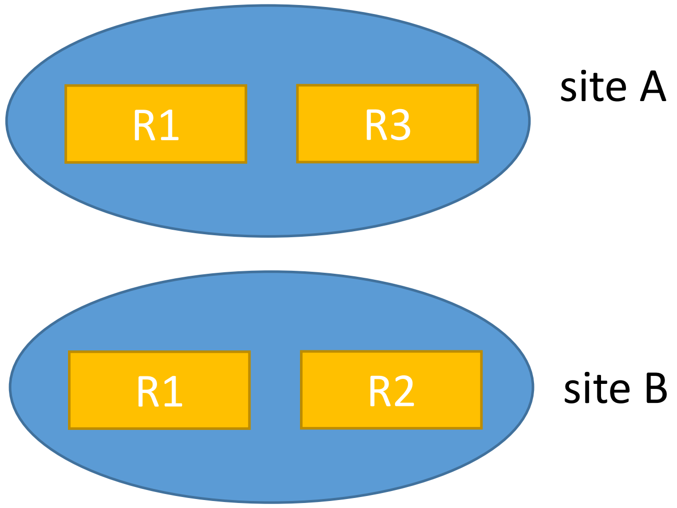
- fragmentation
  - hybrid fragmentation
    - horizontal fragmentation + vertical fragmentation

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

Storing Data

- replication
  - store multiple copies of:
    - a relation
    - a relation fragment

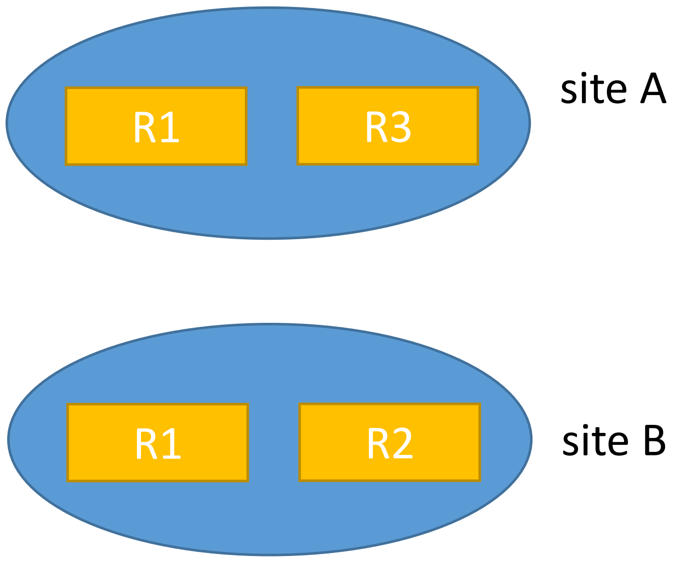
  i.e., an entire relation / 1 or several fragments of a relation can be replicated at one or several sites



  - example
    - R is fragmented into R1, R2, R3
    - R1 is stored at both sites

Storing Data

- replication
  - motivation
    - increased availability of data
      - need R1, query Q uses R1 from site A
        - if site A goes down or a communication link fails, Q can use another active server (e.g., site B)
    - faster query evaluation
      - can use a local copy of the data to avoid communication costs
  - types
    - 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 all copies of R

  => 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

  => 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

Synchronous Replication

- transaction T, object O (with several copies)
- goal
  - no matter which copy of O it accesses, T should see the same value
  - 2 basic techniques
    - *voting*
    - *read-any write-all*
- voting
  - to modify O, T must write a majority of its copies
  - when reading O, T must read enough copies to make sure it's seeing at least one current copy
  - e.g., 10 copies; 7 copies written when updating; 4 copies should be read
  - each copy has a version number
  - not an attractive approach in most cases, because reads are usually much more common than writes

Synchronous Replication

- transaction T, object O (with several copies)
- read-any write-all
  - modify O
    - T must write all copies
  - read O
    - T can read any copy
  - fast reads, slower writes (relative to the voting technique)
  - most common approach to synchronous replication

Synchronous Replication - Costs

- before an update transaction T can commit, it must lock all copies of the modified relation / fragment
  - T sends lock requests to remote sites
  - while waiting for the response, T holds on to other locks
- site / link failures
  - T cannot commit until the network / sites are back up
- no failures, locks immediately obtained
  - still, T must follow an expensive commit protocol when committing, with several messages being exchanged

=> asynchronous replication is more used

Asynchronous Replication

- transaction T modifies object O
- T is allowed to commit before all copies of O have been changed
- readers can access just one copy of O

  => 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*)

Asynchronous Replication

- peer-to-peer replication
  - several copies of object O can be *master copies* (i.e., updatable)
  - changes to a master copy must be propagated to the other copies
  - conflict resolution strategy
    - 2 master copies are changed in a conflicting manner
      - e.g., site 1: Dana’s age is changed to 30; site 2: Dana's age is changed to 40; which value is correct?
    - in general - ad hoc approaches to conflict resolution

Asynchronous Replication

- peer-to-peer replication
  - best utilized when conflicts do not arise:
    - each master site owns a fragment (usually a horizontal fragment)
      - any 2 fragments updatable by different master sites are disjoint
    - updating rights are held by one master site at a time

Asynchronous Replication

- primary site replication
  - exactly one copy of an object O is designated as the *primary* or *master* copy
  - the primary copy is published
  - other sites can subscribe to (fragments of) the primary copy - *secondary copies*
  - secondary copies cannot be updated
  - main issue
    - how are the changes to the primary copy propagated to the secondary copies?
      - *capture* the changes made by committed transactions
      - *apply* these changes to the secondary copies

Asynchronous Replication

- primary site replication
  - 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

Asynchronous Replication

- primary site replication
  - capture
    - procedural capture
      - capture is performed through a procedure that is automatically invoked (e.g., a trigger)
      - typically, the procedure just takes a snapshot of the primary copy
    - log-based capture
      - smaller overhead
      - smaller delay
      - but it depends on proprietary log details

Asynchronous Replication

- primary site replication
  - apply
    - applies changes collected in the Capture step (from the CDT / snapshot) to the secondary copies
      - primary site can continuously send the CDT
      - secondary sites can periodically request a snapshot or (the latest portion of) the CDT from the primary site
        - interval between requests - timer / application program
      - each secondary site runs a copy of the Apply process

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(<u>RID: integer</u>, Name: string, ImpactF: integer, Age: real)
AuthorContribution(<u>RID: integer, PID: integer, Year: integer</u>, Coord: string)

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

Distributed Query Processing

- estimate the cost of evaluation strategies
  - number of I/O operations
  - number of pages shipped among sites, i.e., take into account communication costs
  - t_d
    - time to R / W a page from / to disk
  - t_s
    - time to ship a page from one site to another

Distributed Query Processing

- nonjoin queries in a distributed DBMS
  - impact of fragmentation / replication on simple operations
    - scanning a relation, selection, projection

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

- <u>horizontal fragmentation</u>
  - all Researchers tuples with ImpactF < 6 - stored at New York
  - all Researchers tuples with ImpactF >= 6 - stored at Lisbon
  - DBMS
    - evaluates the query at New York and Lisbon
    - takes the union of the obtained results

Distributed Query Processing

- nonjoin queries in a distributed DBMS

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

  - horizontal fragmentation
    - all Researchers tuples with ImpactF < 6 - stored at New York
    - all Researchers tuples with ImpactF >= 6 - stored at Lisbon
    - DBMS
      - computes SUM(Age) and number of Age values at New York and Lisbon
      - then computes the average age of all researchers with ImpactF in the specified range

Distributed Query Processing

- nonjoin queries in a distributed DBMS

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

  - horizontal fragmentation
    - all Researchers tuples with ImpactF < 6 - stored at New York
    - all Researchers tuples with ImpactF >= 6 - stored at Lisbon
    - DBMS
      - evaluates the query only at Lisbon

Distributed Query Processing

- nonjoin queries in a distributed DBMS
  - <u>vertical fragmentation</u>
    - RID, ImpactF - stored at New York
    - Name, Age - stored at Lisbon
    - DBMS
      - adds a field that contains the id of the corresponding tuple from Researchers to both fragments
      - rebuilds the Researchers relation by joining the 2 fragments on the common field (the tuple-id field)
      - 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
    - 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., New York, Lisbon, 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

* <u>fetch as needed</u>

- page-oriented nested loops in New York
  - obs. bring in all AuthorContribution pages for each Researchers tuple => much higher cost
  - <u>optimization</u>
    - 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)

Distributed Query Processing

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

* <u>fetch as needed</u>

- 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
    - cost
      - $(1.2 + 2.5)t_d$

=> <u>total cost</u>: $500t_d + 40.000(3.7t_d + 2.5t_s)$

Distributed Query Processing

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

* <u>ship to one site</u>

- 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
    - improved version of Sort-Merge Join
      - combine the merging phase of sorting with the merging phase of the join

        => SMJ cost: 3(number of R pages + number of A pages)

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

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

Distributed Query Processing

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

* <u>ship to one site</u>

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

Distributed Query Processing

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

* <u>semijoin</u>

- 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

* <u>bloomjoin</u>

- at New York
  - compute a bit-vector of some size k
    - hash Researchers tuples (using the join column) 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) into the range 0 to k-1

Distributed Query Processing

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

* <u>bloomjoin</u>

- 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

References

- [Ra00] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (2nd Edition), McGraw-Hill, 2000
- [Da03] DATE, C.J., An Introduction to Database Systems (8th Edition), Addison-Wesley, 2003
- [Ga08] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book, Prentice Hall Press, 2008
- [Ra07] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, McGraw-Hill, 2007, http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html
- [Si10] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, McGraw-Hill, 2010, 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