Database Management Systems

Lecture 10

Evaluating Relational Operators

Query Optimization

Distributed Databases

Sabina S. CS

- running example - schema
  - Students (SID: integer, SName: string, Age: integer, RoundedGPA: integer)
  - Courses (CID: integer, CName: string, Description: string)
  - Exams (SID: integer, CID: integer, EDate: date, Grade: integer)

  - Students
    - every record has 50 bytes
    - there are 80 records / page
    - 500 pages
  - Courses
    - every record has 40 bytes
    - there are 100 records / page
    - 1 page
  - Exams
    - every record has 40 bytes
    - there are 100 records / page
    - 1000 pages

Enumeration of Alternative Plans

- query Q
    - consider a certain set of plans
    - choose the plan with the least estimated cost
        - algebraic equivalences
        - implementation techniques for Q's operators
- not all algebraically equivalent plans are enumerated (optimization costs would be too high)
- two main cases:
    - queries with one relation in the FROM clause
    - queries with two or more relations in the FROM clause

Enumeration of Alternative Plans

- <u>queries with one relation in the FROM clause</u>
  - i.e., no joins; only σ, π, grouping, aggregate operations
  - if there is only one σ or π or aggregate operation: consider implementation techniques and cost estimates discussed in previous lectures
  - if there is a combination of operations:
    - plans with / without indexes
  - example query:

```
SELECT S.RoundedGPA, COUNT(*)
FROM Students S
WHERE S.RoundedGPA > 5 AND S.Age = 20
GROUP BY S.RoundedGPA
HAVING COUNT(DISTINCT S.SName) > 5
```

$\pi_{S.RoundedGPA,\ COUNT(*)}($
$HAVING_{COUNT\ DISTINCT\ (S.SName) > 5}($
$GROUP\ BY_{S.RoundedGPA}($
$\pi_{S.RoundedGPA,\ S.SName}($
$\sigma_{S.RoundedGPA > 5\ \land\ S.Age = 20}($
$Students)))))$

Enumeration of Alternative Plans

* plans without indexes:

- apply σ, π while scanning Students
  - file scan
    - NPages(Students)
    - 500 I/Os
  - write out tuples to a temporary relation T:
    - NPages(Students) * RF(RoundedGPA > 5) * RF(Age = 20) * (size of a *pair <RoundedGPA, SName>* / size of a *Students tuple*)
    - RF for *RoundedGPA > 5*
      - 0.5
    - RF for *Age = 20*
      - 0.1
    - size of <RoundedGPA, SName>
      - about 0.8 * size of a Students tuple

Enumeration of Alternative Plans

* plans without indexes:

- apply σ, π while scanning Students
  - write out tuples to a temporary relation T:
    => 500 * 0.5 * 0.1 * 0.8 = 20 I/Os (temporary relation T)
- GROUP BY:
  - sort T in 2 passes
    - 4 * 20 = 80 I/Os
- HAVING, aggregations
  - no additional I/O
- **total cost**
  - 500 + 20 + 80 = **600 I/Os**

Enumeration of Alternative Plans

* plans that use an index:

- available indexes on Students - a2
  - hash index on <Age>
  - B+ tree index on <RoundedGPA>
  - B+ tree index on <RoundedGPA, SName, Age>
- <u>single-index access path:</u>
  - choose the index that provides the most selective access path
  - apply π, nonprimary selection terms (i.e., that don't match the index)
  - compute grouping and aggregation operations
  - example:
    - use the hash index on Age to retrieve Students with Age = 20
      - cost: retrieve index entries and corresponding Students tuples
    - apply condition *RoundedGPA > 5* to each retrieved tuple
    - retain RoundedGPA and SName

Enumeration of Alternative Plans

* plans that use an index:

- single-index access path – example:
  - write out tuples to a temporary relation
  - sort the temporary relation by RoundedGPA to identify groups
  - apply the HAVING condition (to eliminate some groups)
- multiple-index access path
  - several indexes using a2 / a3 match the selection condition, e.g., I1, I2
  - retrieve Rids_{I1}, Rids_{I2} using I1, I2
  - get tuples with rids in $\text{Rids}_{I1} \cap \text{Rids}_{I2}$ (tuples satisfying the primary selection terms of I1 and I2)
  - apply π, nonprimary selection terms
  - compute grouping and aggregation operations
  - example:
    - use the index on Age => rids of tuples with Age = 20 (R1)

Enumeration of Alternative Plans

* plans that use an index:

- multiple-index access path – example:
    - index on RoundedGPA => rids of tuples with RoundedGPA > 5 (R2)
    - retrieve tuples with rids in R1 ∩ R2
    - keep only RoundedGPA and SName
    - write out tuples to a temporary relation
    - sort the temporary relation by RoundedGPA to identify groups
    - apply the HAVING condition (to eliminate some groups)
- sorted index access path:
    - works well when the index is clustered
    - B+ tree index I with search key K
    - GROUP BY attributes – prefix of K
    - use the index to retrieve tuples in the order required by the GROUP BY clause

Enumeration of Alternative Plans

* plans that use an index:

- apply σ, π
- compute aggregation operations

- <u>sorted index access path</u> – example:
  - use the B+ tree index on RoundedGPA to retrieve Students tuples with RoundedGPA > 5, ordered by RoundedGPA
  - aggregations in HAVING, SELECT - computed on-the-fly
- <u>index-only access path:</u>
  - index I with search key K
  - all the attributes in the query are included in K

  => index-only scan, don't need to retrieve tuples from the relation
  - data entries: apply σ, perform π, sort the result (to identify groups), compute aggregate operations

  * obs. index I doesn't have to match the selections in the WHERE clause

Enumeration of Alternative Plans

* plans that use an index:

- <u>index-only access path</u>

  * obs. I – tree index, GROUP BY attributes – prefix of K

  => can avoid sorting

  - example:
    - use the B+ tree index on <RoundedGPA, SName, Age> to retrieve entries with RoundedGPA > 5, ordered by RoundedGPA
    - select entries with Age = 20
    - aggregation operations in the HAVING and SELECT clauses - computed on-the-fly

Enumeration of Alternative Plans

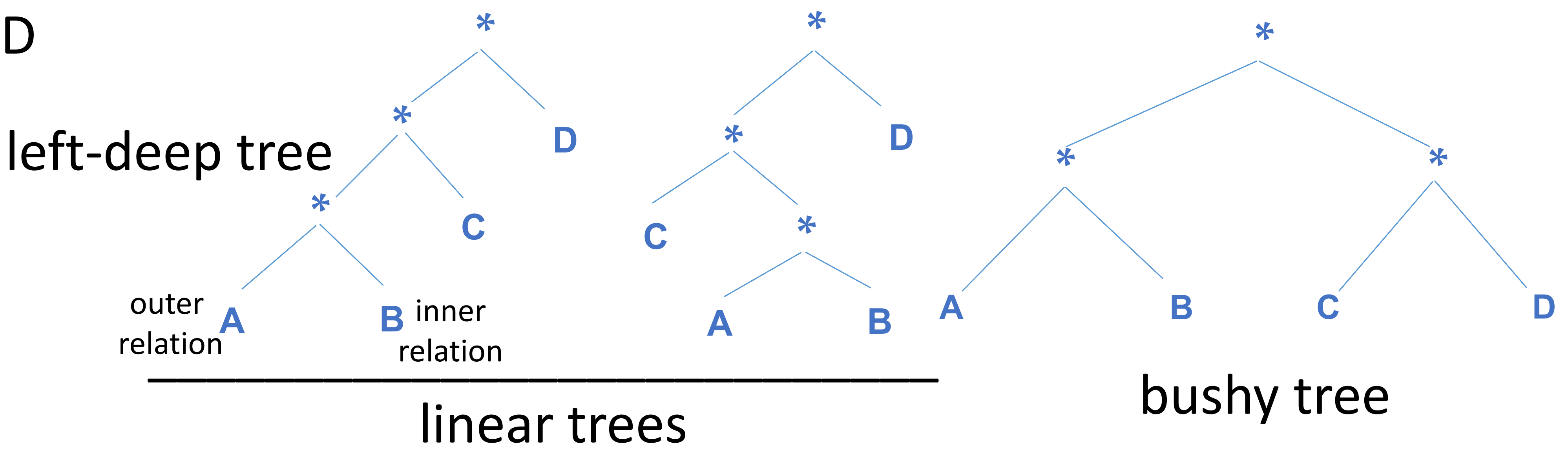
- <u>queries with several relations in the FROM clause:</u>
  - joins, cross-products => queries can be quite expensive
  - different join orders => intermediate relations of widely varying sizes => plans with very different costs

- class of plans considered by the optimizer
- plan enumeration

Enumeration of Alternative Plans

- <u>queries with several relations in the FROM clause</u>

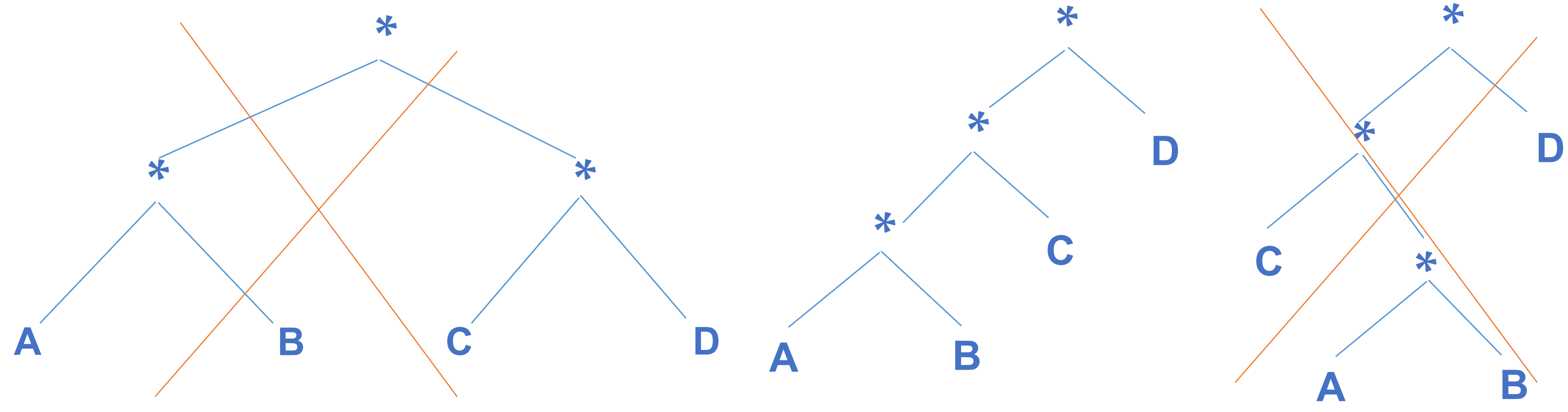
A * B * C * D



- linear trees:
  - at least one child of a join node is a base relation
- left-deep trees:
  - the right child of each join node is a base relation
- bushy tree: not linear

Enumeration of Alternative Plans

- <u>queries with several relations in the FROM clause</u>
- fundamental decision in System R:
  - only *left-deep trees* are considered



- motivation:
  - number of joins increases => number of alternative plans increases quickly => must prune the search space
  - left-deep trees generate all <u>fully pipelined plans</u> (all joins are evaluated using pipelining)

Enumeration of Alternative Plans

- <u>queries with several relations in the FROM clause</u>
- multiple passes:
  - pass 1:
    - find best 1-relation plan for each relation
  - pass 2:
    - find best way to join the result of each 1-relation plan (as the outer argument) with another relation (all 2-relation plans)

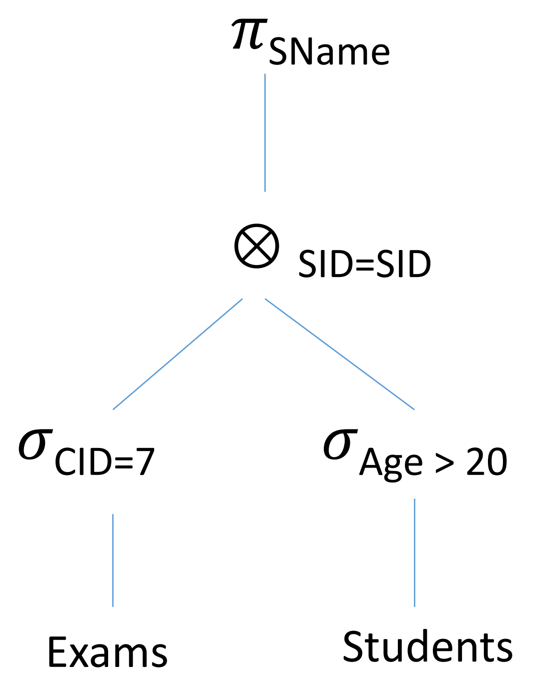
  …
  - repeat until the obtained plans contain all the relations in the query
- for each subset of relations, retain: the cheapest plan overall & the cheapest plan for each interesting ordering of tuples

Enumeration of Alternative Plans

- queries with several relations in the FROM clause
    - GROUP BY, aggregates are handled as a final step:
        - use a plan with an interesting ordering of tuples
        - use an additional sorting operator
- obs. avoid cross-products if possible

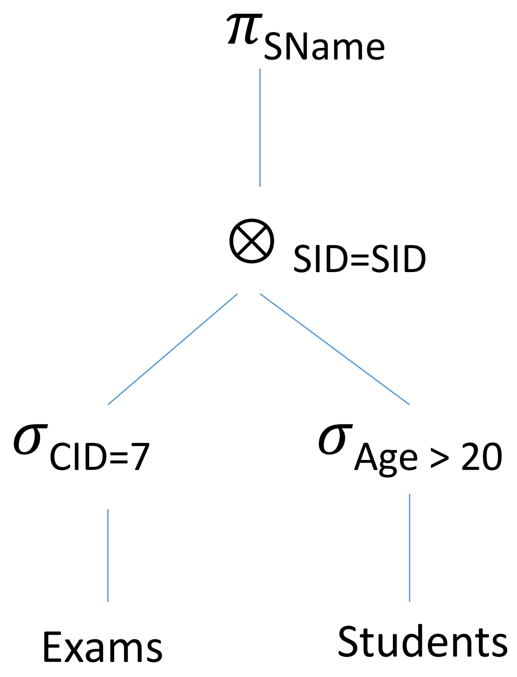
Example

- unclustered indexes using a2
- Students: B+ tree index on Age, hash index on SID
- Exams: B+ tree index on CID
- pass 1:
  - Students – 3 possible access paths:
    B+ tree index, hash index, file scan
    - *Age > 20* matches the B+ tree index on Age
    - hash index / file scan => probably much higher cost
    - keep the plan using the B+ tree index => retrieved tuples are ordered by Age
  - Exams – 2 possible access paths
    - *CID = 7* matches the B+ tree index on CID
    - better than file scan => keep the plan using the B+ tree index



Example

- Students: B+ tree index on Age, hash index on SID
- Exams: B+ tree index on CID
- pass 2:
  - consider (the result of) each plan from pass 1 as the outer argument and analyze how to join it with the other relation
  - e.g., Exams – outer argument
    - examine alternative access methods / join methods
    - access methods:
      - need Students tuples s.t. SID = *value* from outer tuple and Age > 20
      - hash index => Students tuples s.t. SID = *value* from outer tuple
      - B+ tree index => Students tuples s.t. Age > 20
    - join methods:
      - consider all available methods



Example

- Students: B+ tree index on Age, hash index on SID
- Exams: B+ tree index on CID
- pass 2:
  - consider (the result of) each plan from pass 1 as the outer argument and analyze how to join it with the other relation
  - e.g., Students – outer argument
    - access / join methods
    - access methods:
      - need Exams tuples s.t. SID = *value* from outer tuple and CID = 7
    - join methods:
      - consider all available methods
  - retain cheapest plan overall!

Nested Queries - optional

- usually handled using some form of nested loops evaluation
- correlated query - typical evaluation strategy:
    - the inner block is evaluated for each tuple of the outer block
- some strategies are not considered
    - e.g., index on SID in Students
    - best plan could be INLJ with Exams as the outer argument, and Students as the inner one; such a plan is never considered by the optimizer
- the unnested version of the query is typically optimized better

```
SELECT  S.SName
FROM  Students S
WHERE EXISTS
   (SELECT  *
    FROM  Exams E
    WHERE  E.CID=7
     AND E.SID=S.SID)
```

Equivalent unnested query:

```
SELECT  S.SName
FROM Students S, Exams E
WHERE E.SID=S.SID AND E.CID = 7
```

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

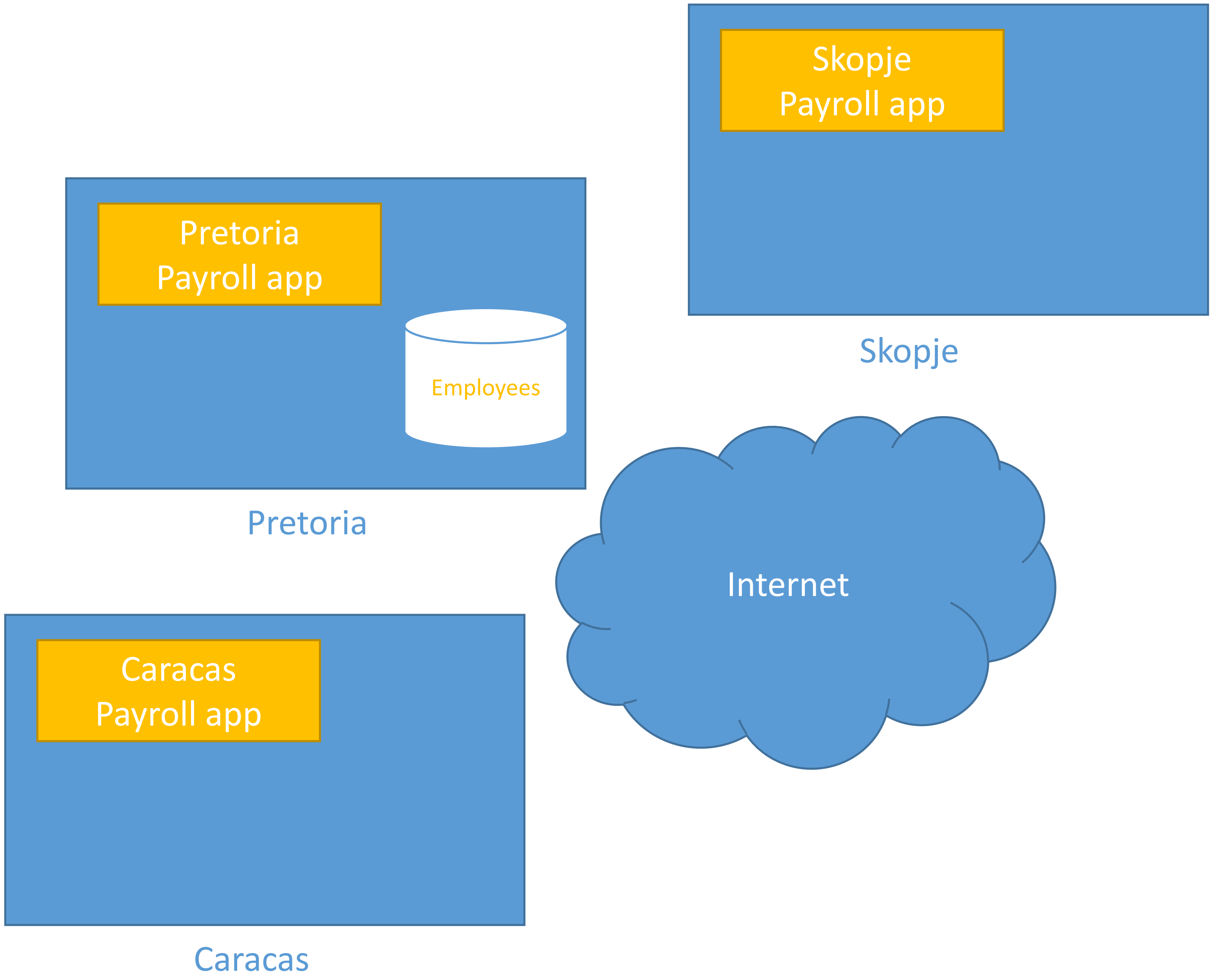
* properties:

- 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



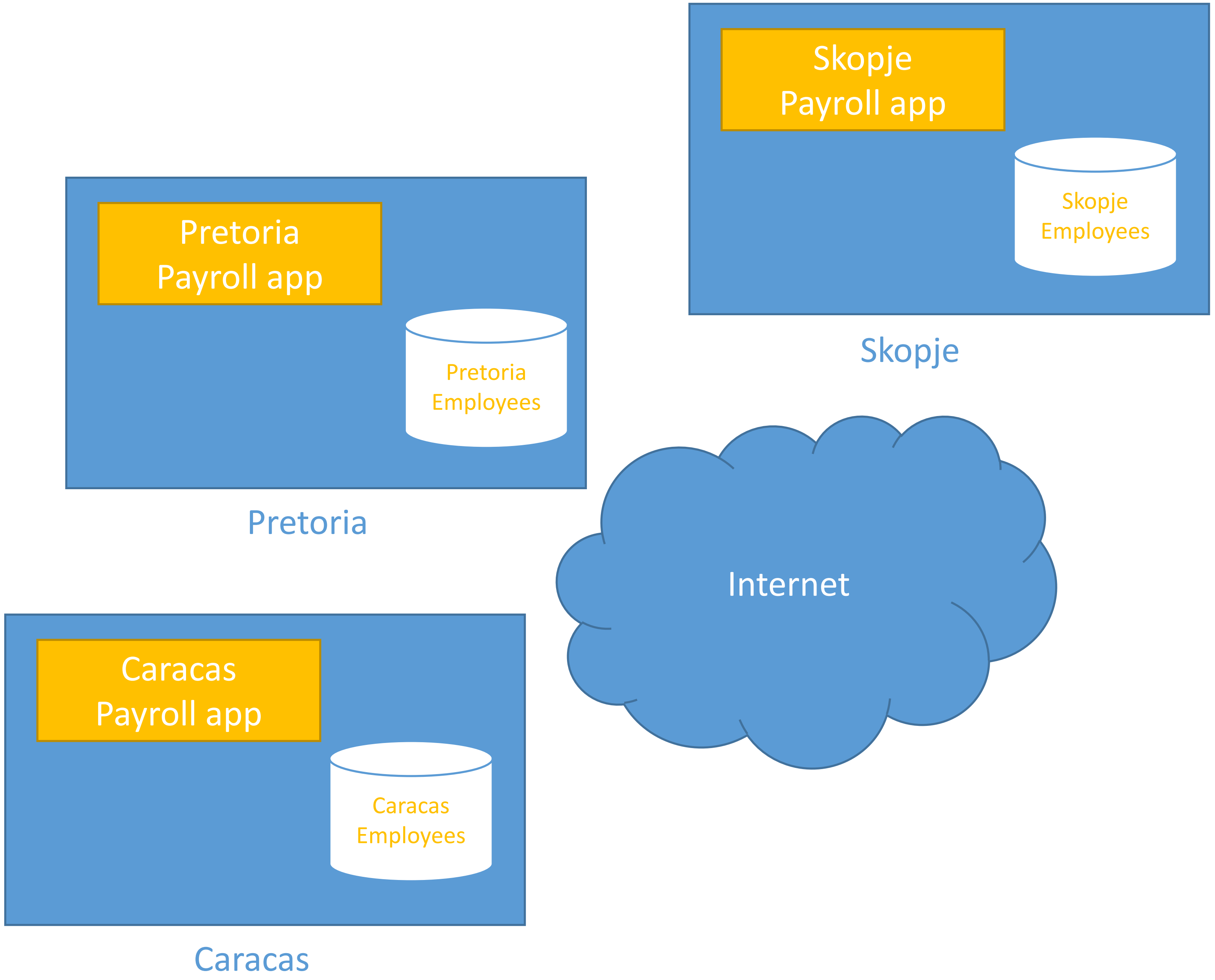
- *SiteX* payroll app – computing payrolls for *SiteX* employees

- 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

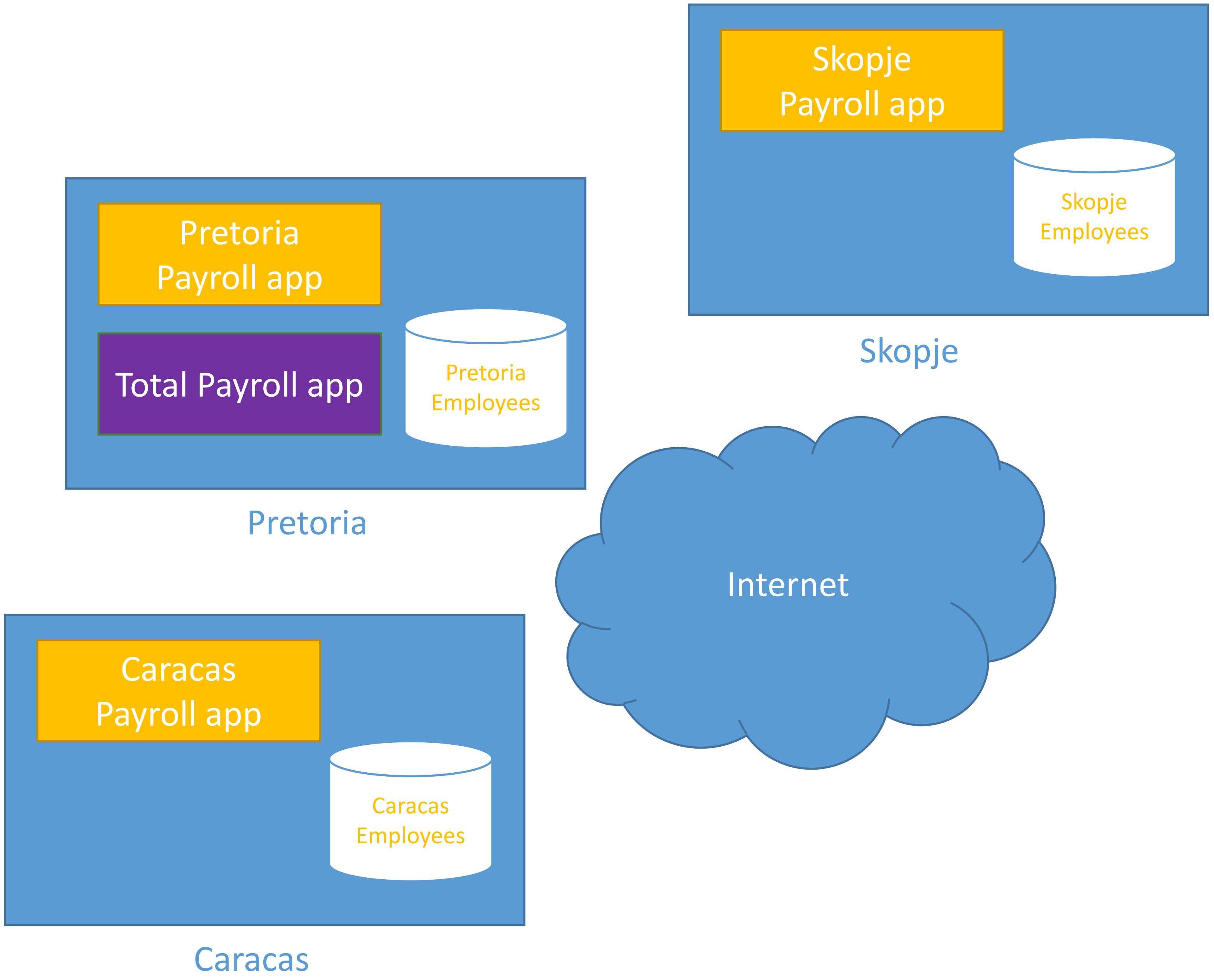


- store data about Pretoria employees in Pretoria, about Skopje employees in Skopje, etc.

=> improved performance for payroll apps in Caracas and Skopje

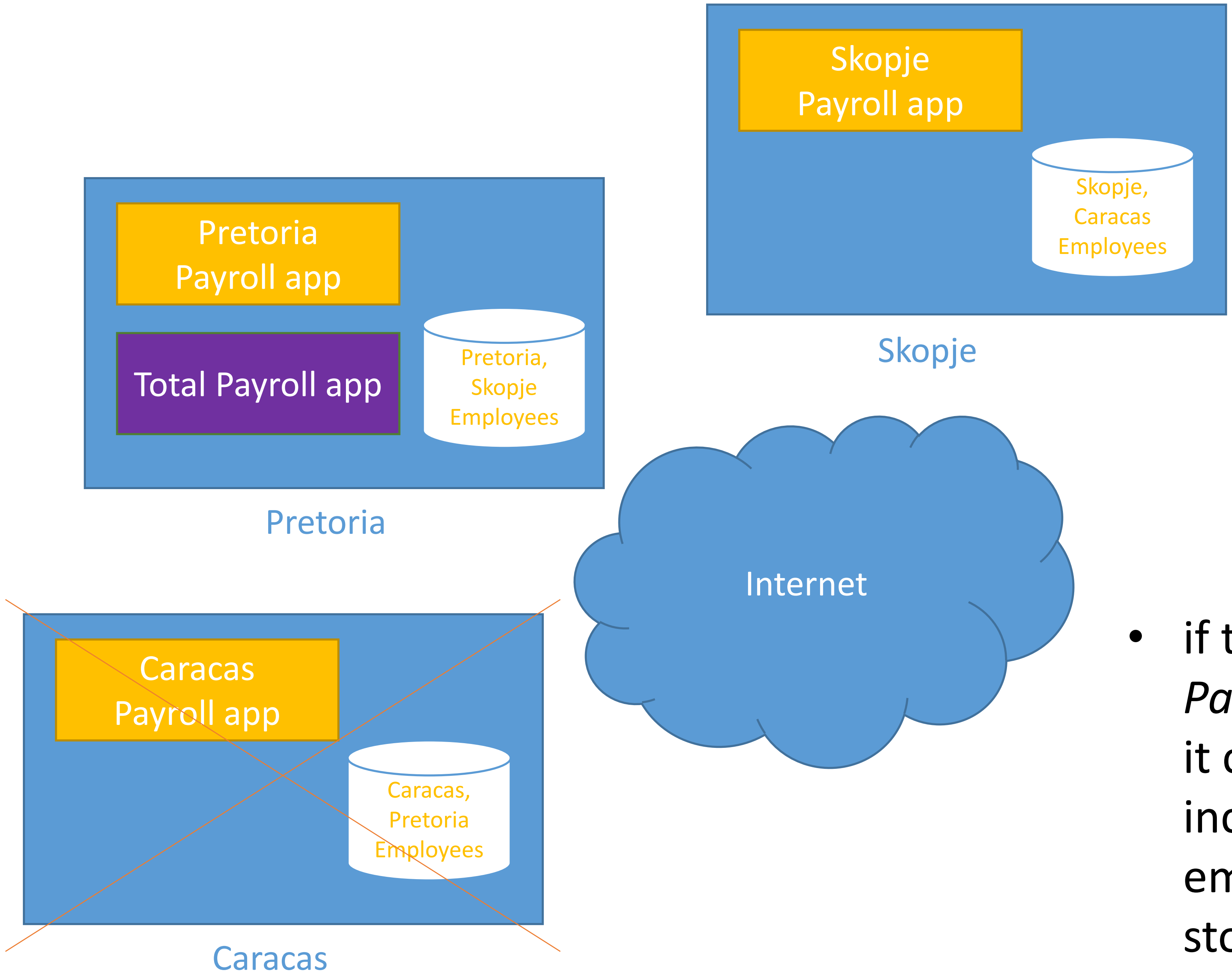
- if the Pretoria site becomes unavailable, payroll apps in Caracas and Skopje are still operational (as these apps only need data about Caracas employees and about Skopje employees, respectively)

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



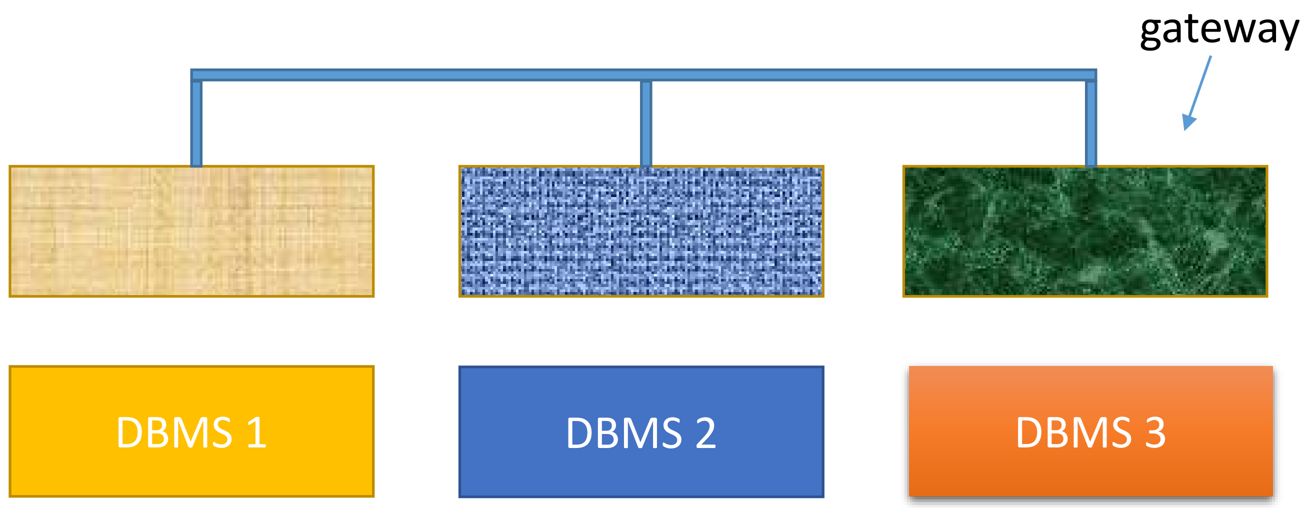
- data replication: at *Site X*, store data about *Site X* employees and data about employees from a different site, e.g., store data about Pretoria employees and Skopje employees in Pretoria (a copy of the Skopje employees data, which is stored in Skopje)
- if the Caracas site crashes, the *Total Payroll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

* <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
  - 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

* <u>distributed concurrency control:</u>

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

* <u>reliability of distributed databases:</u>

- 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
- <u>reduce</u> such <u>costs</u>: *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
- <u>reduce</u> such <u>costs</u>: *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*

* 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) =>

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

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

- <u>vertical fragmentation</u>
  - 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

- <u>hybrid fragmentation</u>
  - horizontal fragmentation + vertical fragmentation

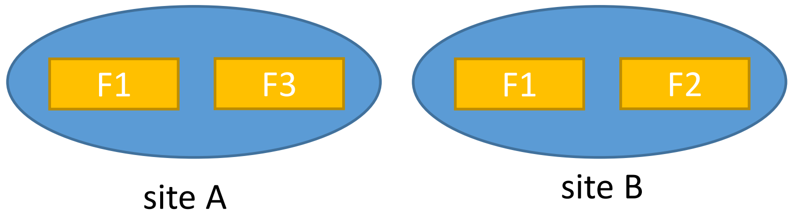
R1	R2
1, Radu 2, Ana 3, Ionel 6, Calin	1, 250, Eroilor 2, 200, Napoca 3, 150, Motilor 6, 250, Eroilor
R3	R4
4, Maria 5, Andi 7, Iulia	4, 400, Eroilor 5, 600, Napoca 7, 350, Motilor

* 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:
  - <u>increased availability of data</u>: 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)
  - <u>faster query evaluation</u>: 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: <u>synchronous versus asynchronous</u>
  - how are the copies of the data kept current when the relation is changed

Updating Distributed Data

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

=> data distribution is transparent to the user

- <u>asynchronous replication:</u>
    - 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)

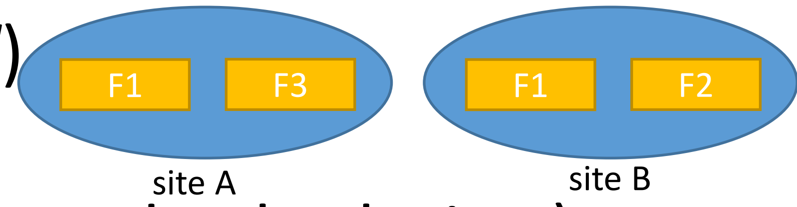
Updating Distributed Data – Synchronous Replication

* *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 is / 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

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

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
- [Si19] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts (7th Edition), McGraw-Hill, 2019
- [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