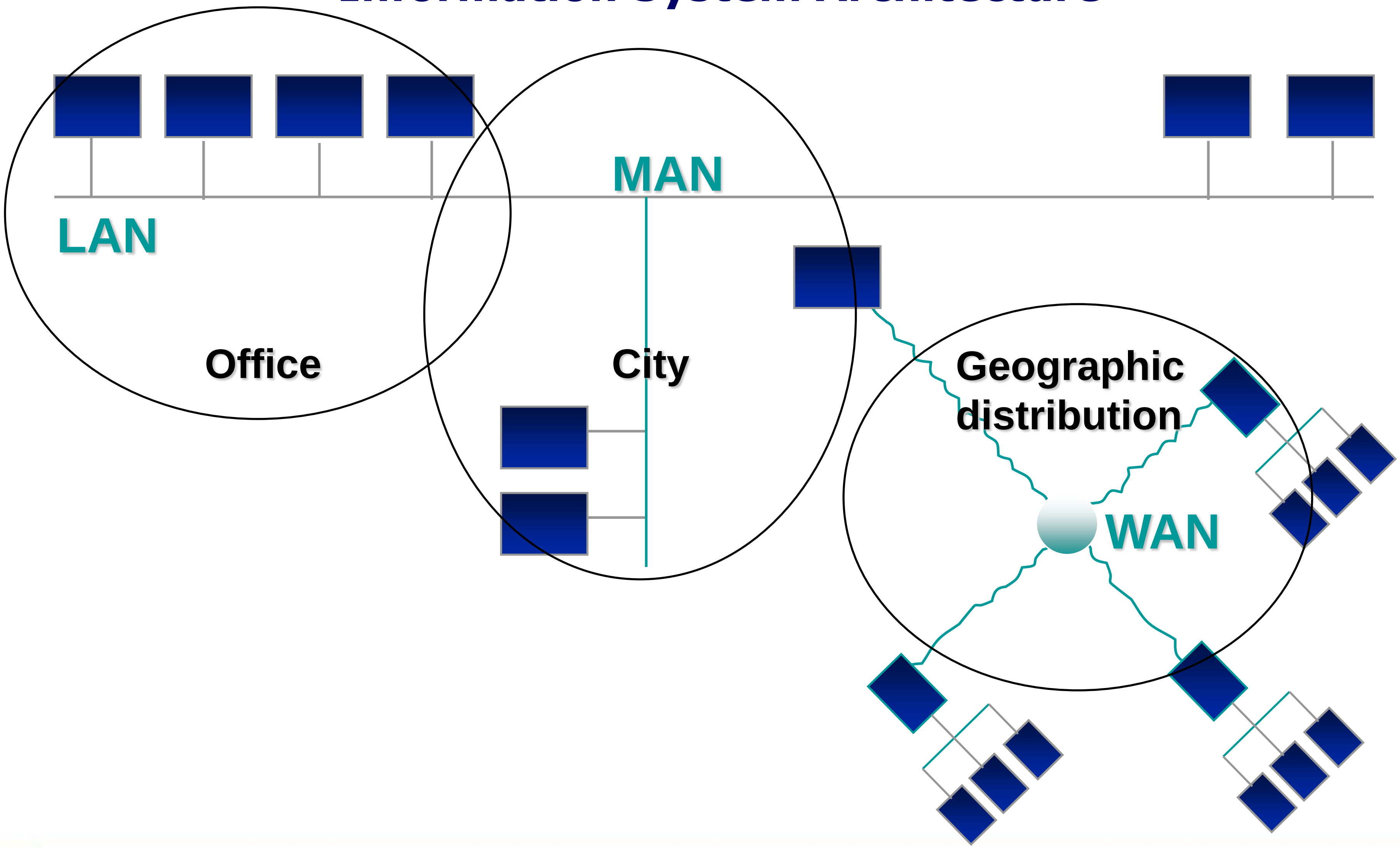
Advanced Databases

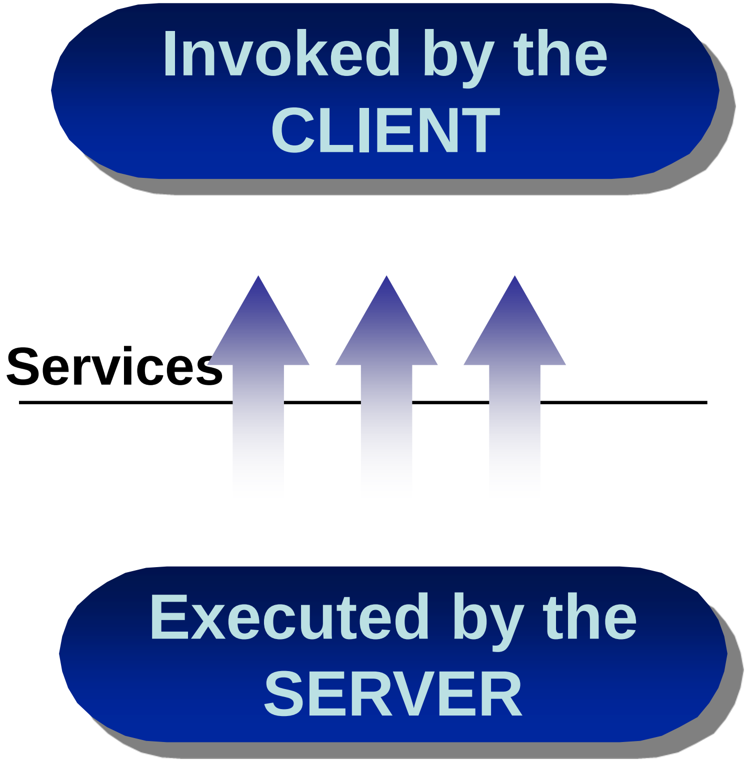
5 Distributed Databases

Information System Architecture



Client-Server Paradigm

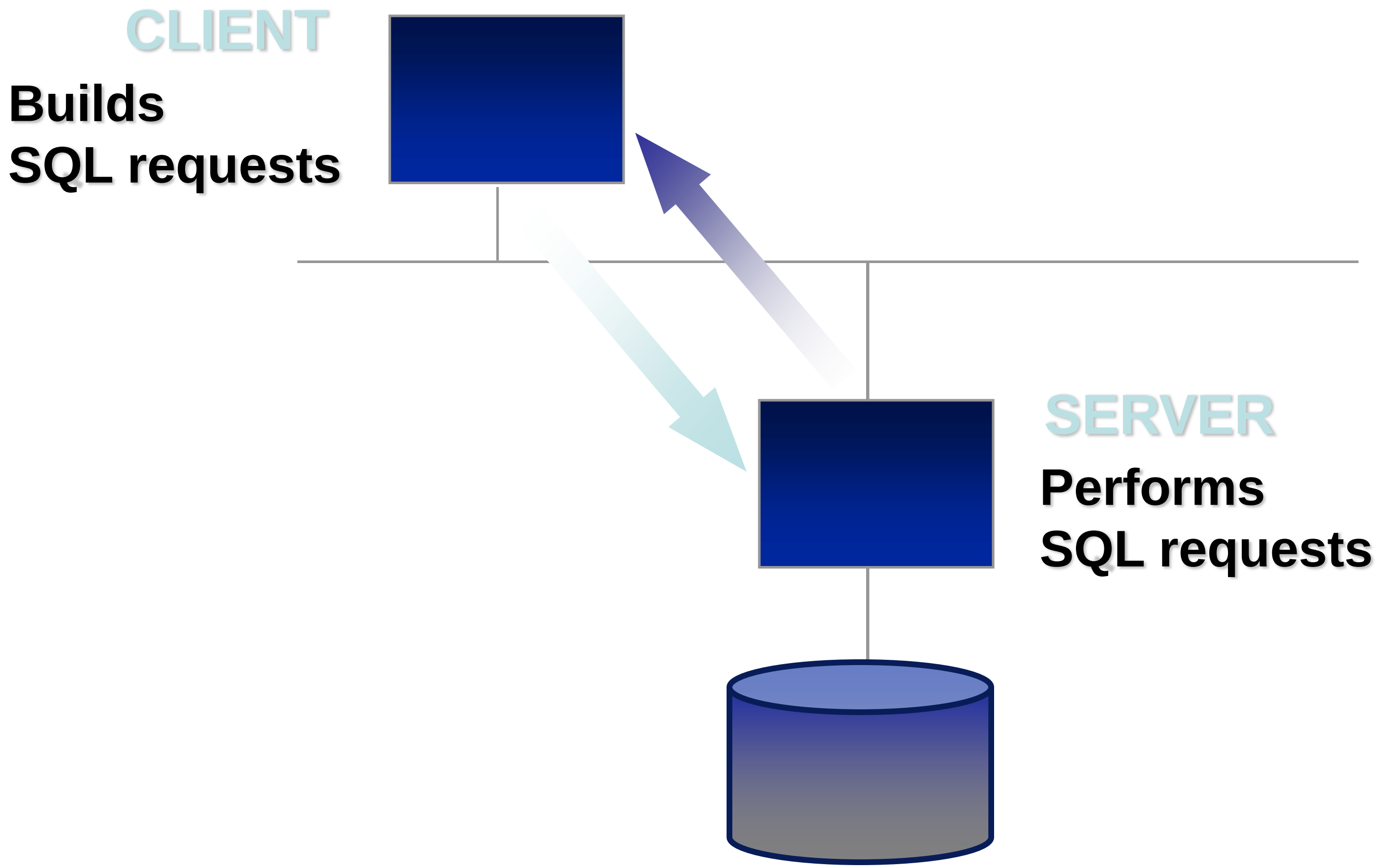
- Technique for software system design
- Two systems are involved:
  - Client: invokes services
  - Server: provides services
- The client process performs an active role, the server process is reactive
- A service interface is published by the server



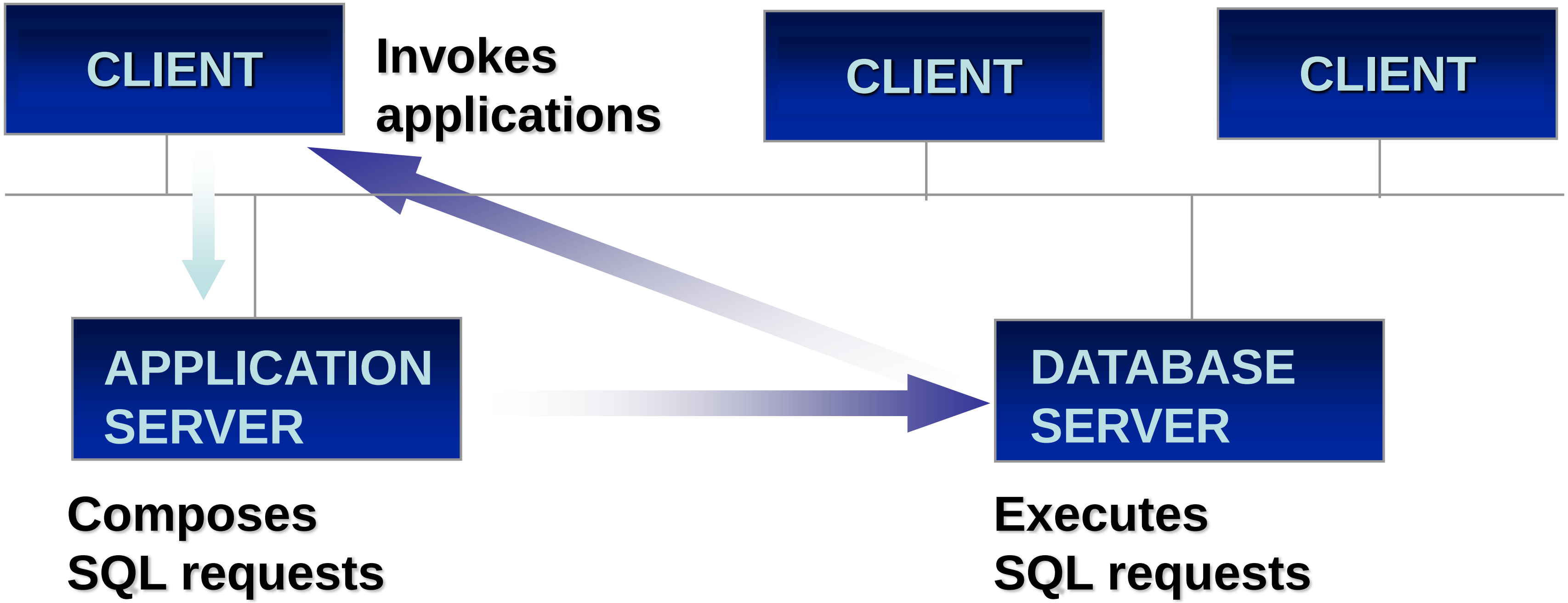
Client-Server in Information Systems

- Ideal functional separation
  - Client: presentation layer
  - Server: data management
- SQL: the perfect language for enacting separation
  - Client: formulates queries and shows results
  - Server: performs queries and calculates the results
  - Network: transfers activation commands (e.g., of SQL procedures) and query results

Traditional Client-Server architecture



Application Server Architecture



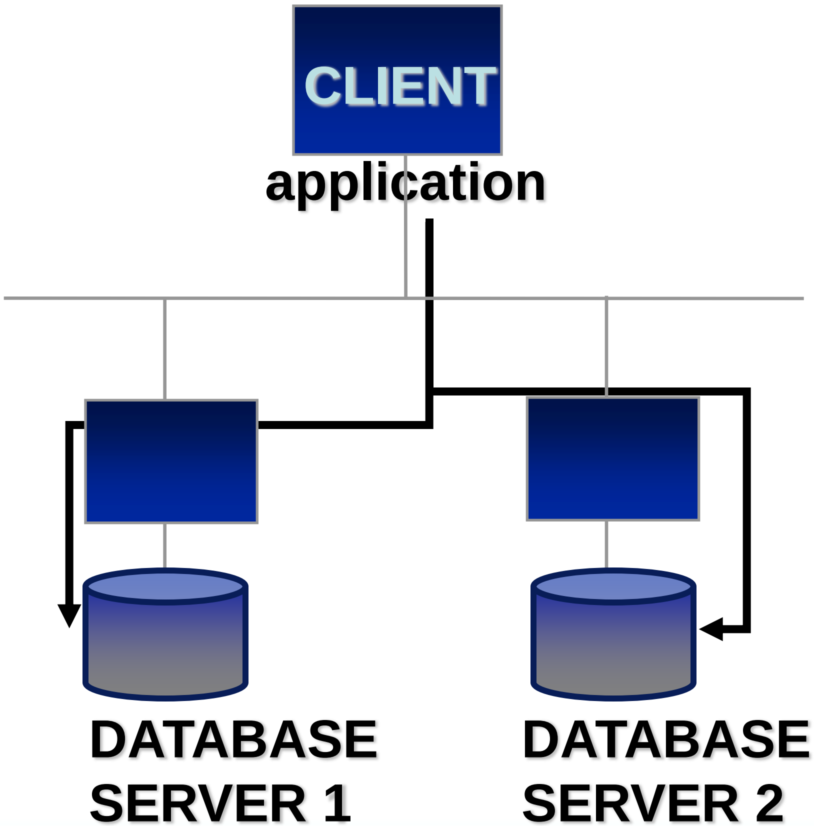
Processing of requests

- Input requests from clients are queued
- A dispatcher routes them to the appropriate software service (typically: multiple simple requests to the same “active server”, no need of loading into memory)
- Request is processed and an output message is put on the output queue, next routed to the requesting client
- Some systems can dynamically configure the number of software services depending on the input load
  - load balancing for given service classes
- Performance is achieved by parallelism and multi-processing (see next)

Data distribution

- Not only
  - Several databases
- But also
  - Applications that use data of different data sources

→**Distributed Database**

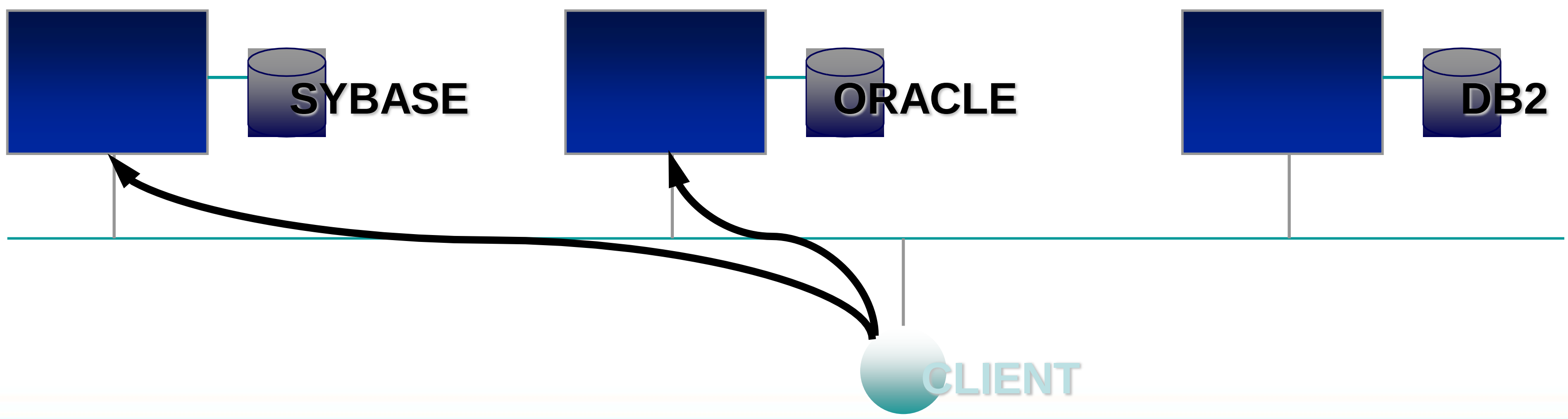


Motivations of Data Distribution

- Intrinsic distributed nature of the applications
- Evolution of computers:
  - Increased computation power
  - Reduced price
- Evolution of DBMS technology
- Interoperability standards

Distributed Database Types

- Classification based on the network:
  - LAN (Local Area Network)
  - WAN (Wide Area Network)
- Classification based on the involved databases:
  - Homogeneous system: all the same DBMS
  - Heterogeneous system: various DBMS



Typical Application Examples

	LAN	WAN
HOMOGENEOUS	Intra-division company management	Travel management and financial applications
HETEROGENEOUS	Inter-division company management	Integrated booking systems and inter-banking systems

Problems of Distributed Databases

- Independency and cooperation
- Transparency
- Efficiency
- Reliability

Independency and cooperation

- Independency needs are driven by:
  - Reaction to EDP divisions in the enterprise
  - Bringing knowledge and control local to the place where data are produced, used, and managed
  - Localizing most of the data flows and giving local autonomy over processing
- Cooperation needs are driven by:
  - Company-wide or business-to-business applications
  - Integrating several needs into a single, higher-level need (served by an application as a whole)

Data fragmentation

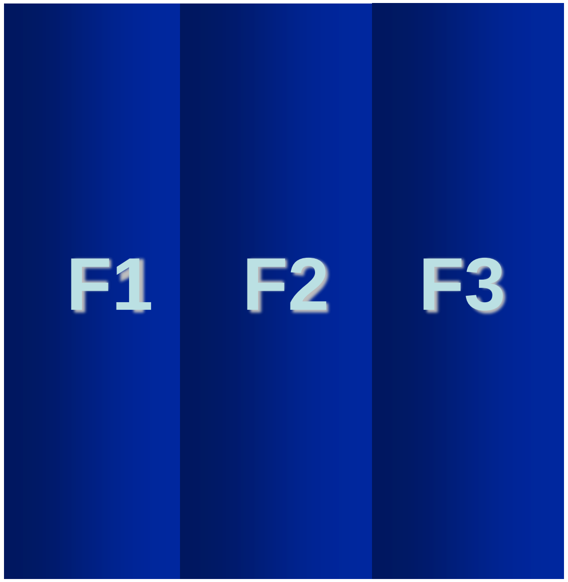
- Decomposition of the tables for allowing their distribution
- Properties:
  - Completeness: each data item of a table T must be present in one of its fragments T_i
  - Restorability: the content of a table *T* must be restorable from its fragments

Horizontal Fragmentation

- Fragments:
  - Sets of tuples
- Completeness:
  - availability of all the tuples
- Restorability:
  - UNION

Vertical Fragmentation

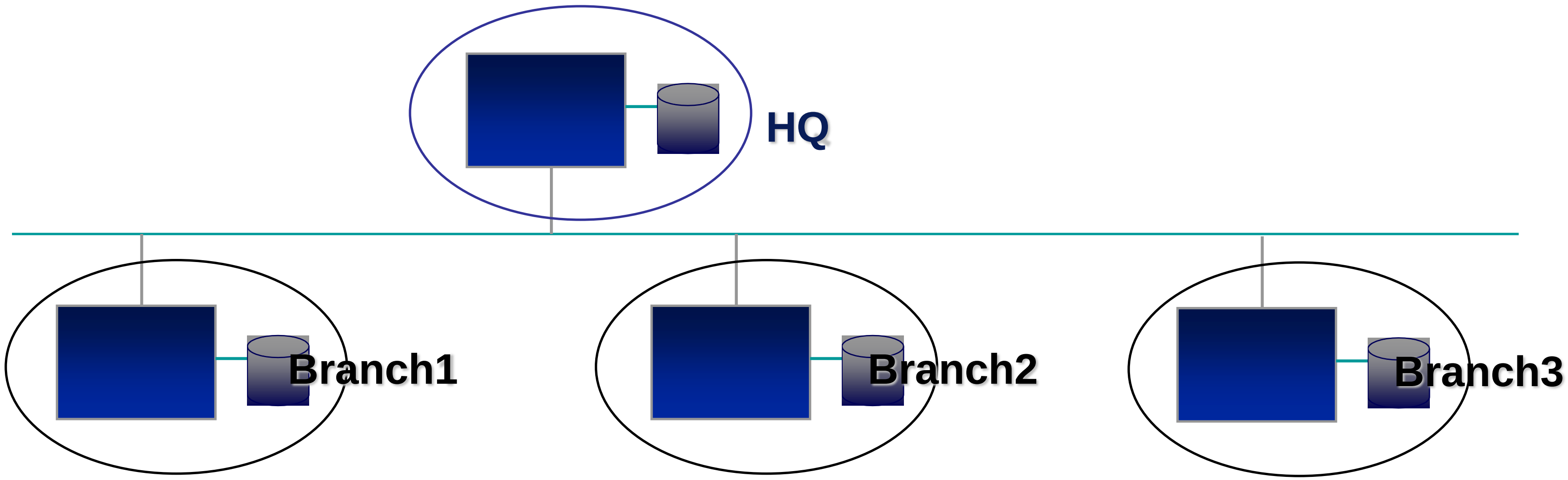
- Fragments:
  - Sets of attributes
- Completeness:
  - availability of all the attributes
- Restorability:
  - JOIN on the key



Example: bank accounts

ACCOUNT (Number, Name, Branch, Balance)

TRANSACTION (AccountNumber, Date, Incremental, Amount, Description)



(Primary) Horizontal Fragmentation

$Ri = \sigma_{Pi} R$

Example:

$Account1 = \sigma_{Branch=1}\ ACCOUNT$

$Account2 = \sigma_{Branch=2}\ ACCOUNT$

$Account3 = \sigma_{Branch=3}\ ACCOUNT$

Derived Horizontal Fragmentation

$S_i = S \rhd\!< R_i$

Example:

Transaction1 = TRANSACTION ⊳< ACCOUNT1

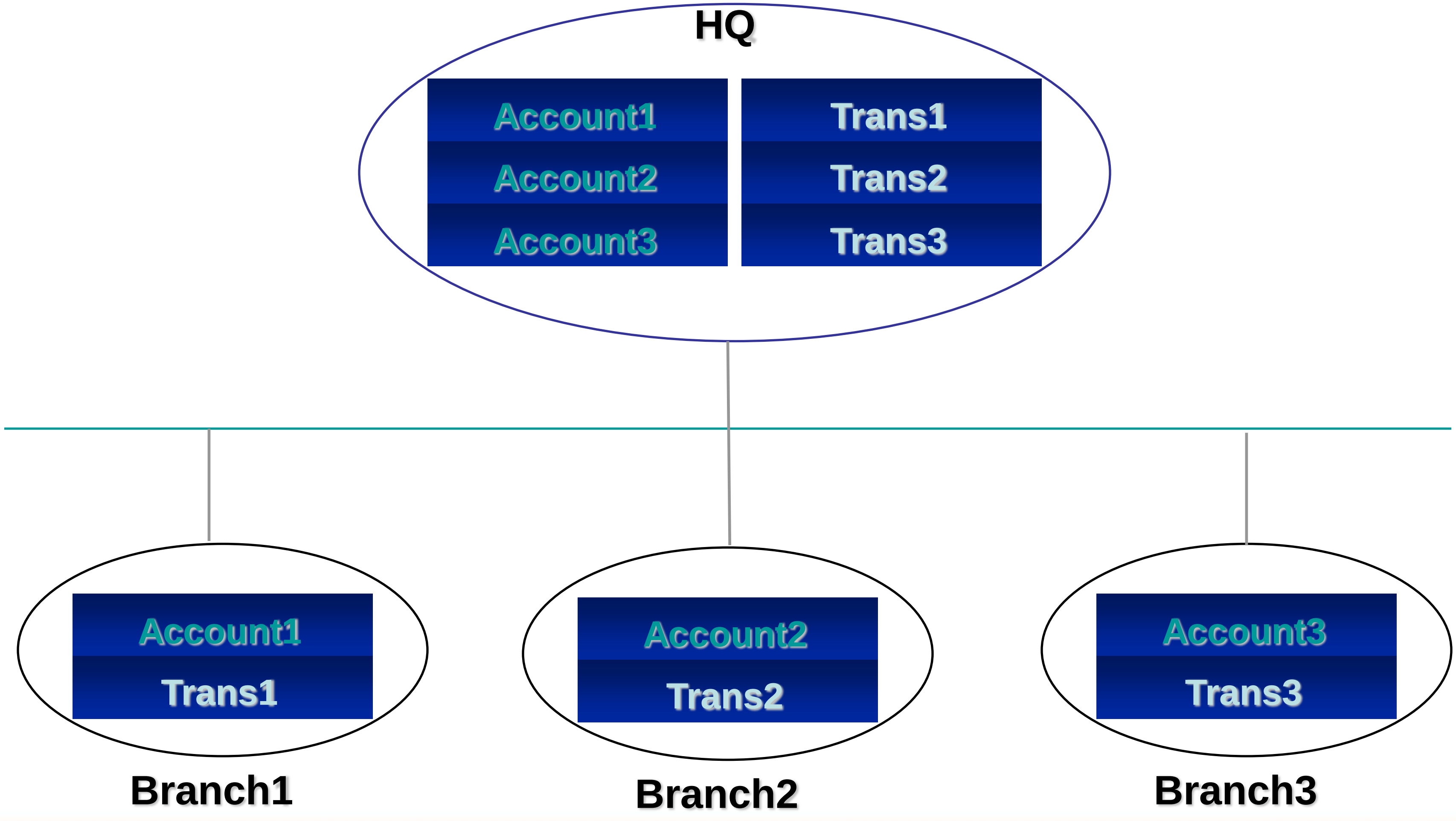
Transaction2 = TRANSACTION ⊳< ACCOUNT2

Transaction3 = TRANSACTION ⊳< ACCOUNT3

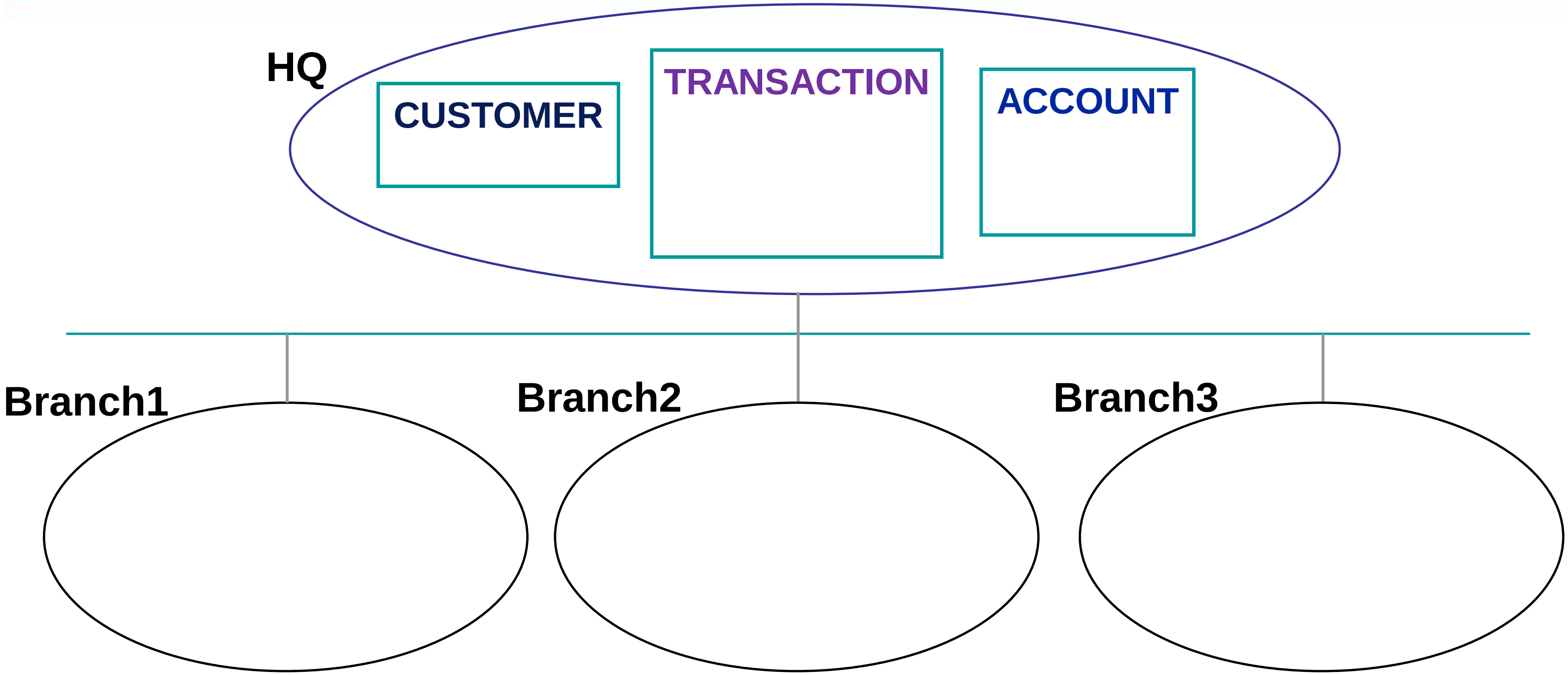
Fragment Allocation

- Network:
  - 3 peripherical sites, 1 central site
- Allocation:
  - Local
  - centralized

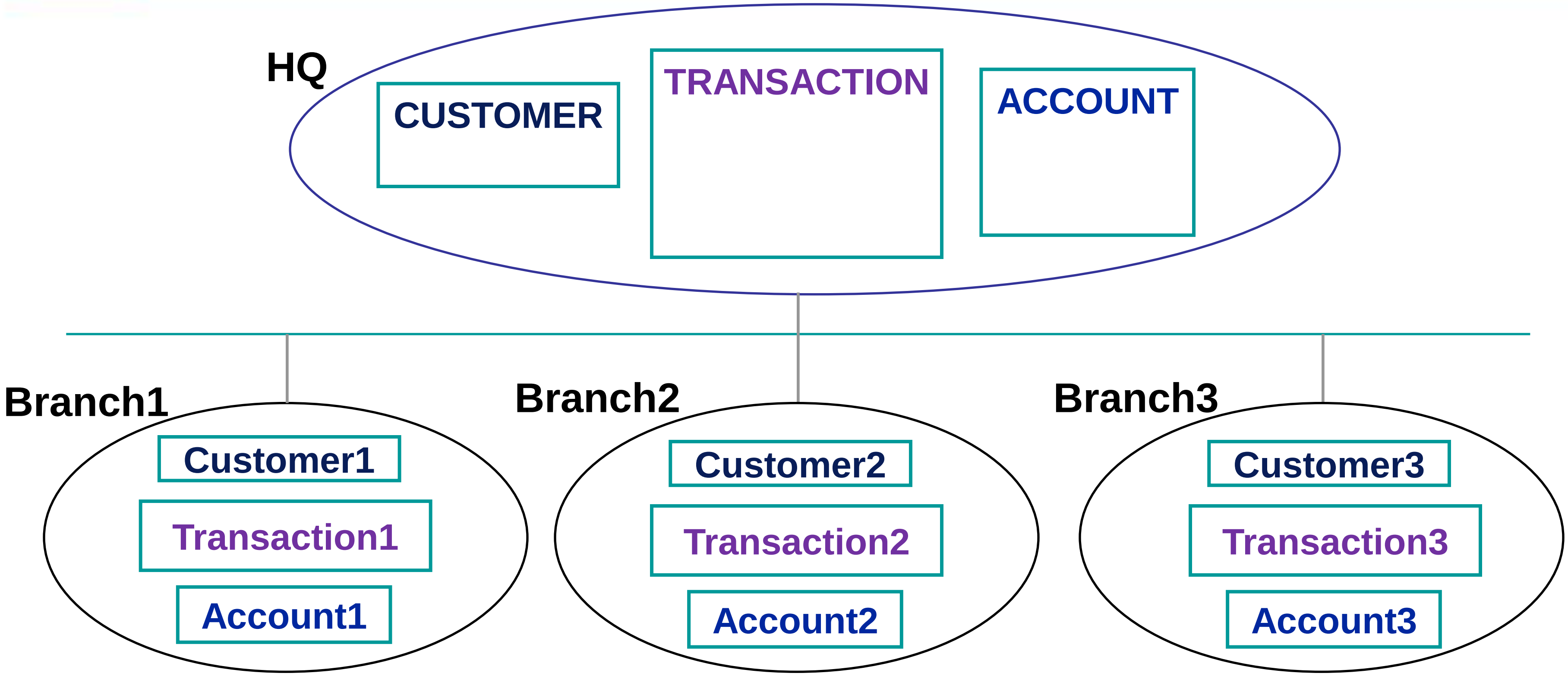
Fragment allocation



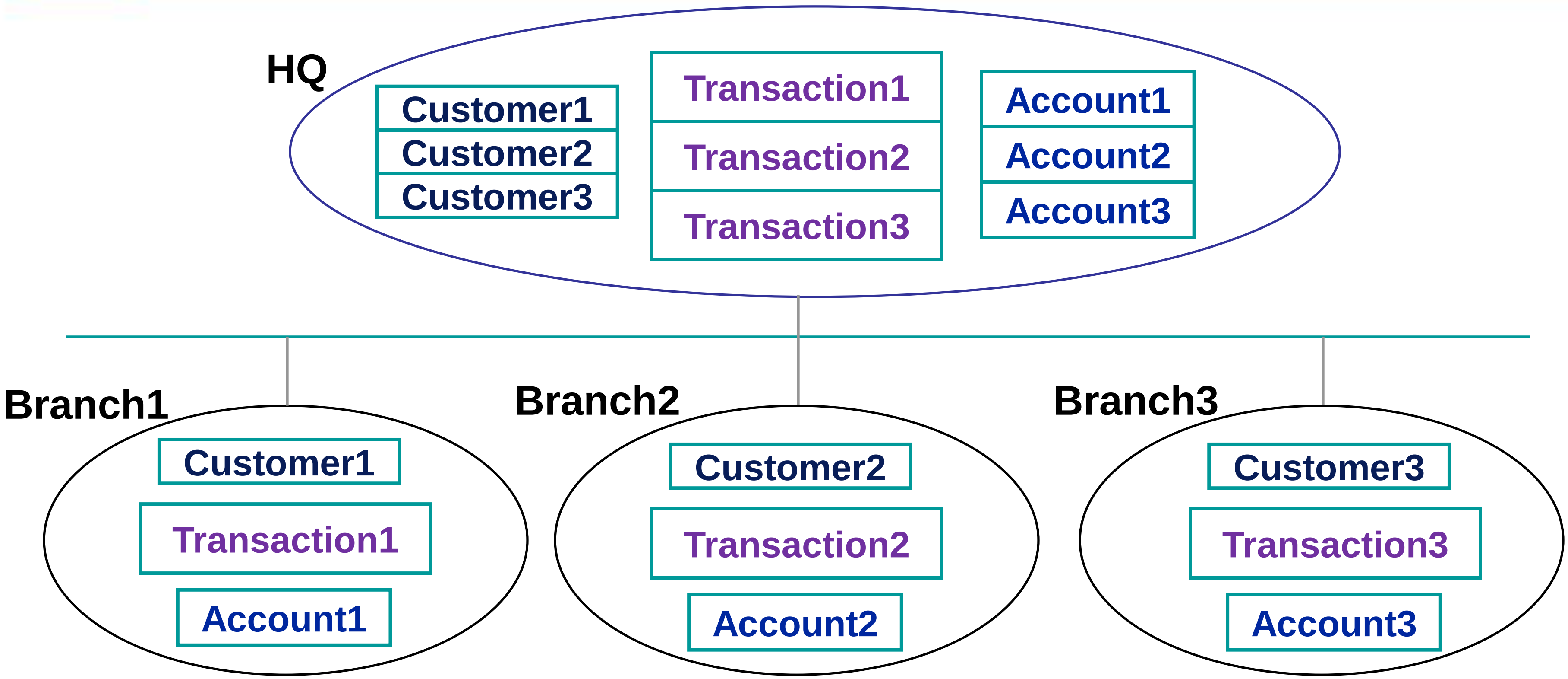
Fully centralized [A]



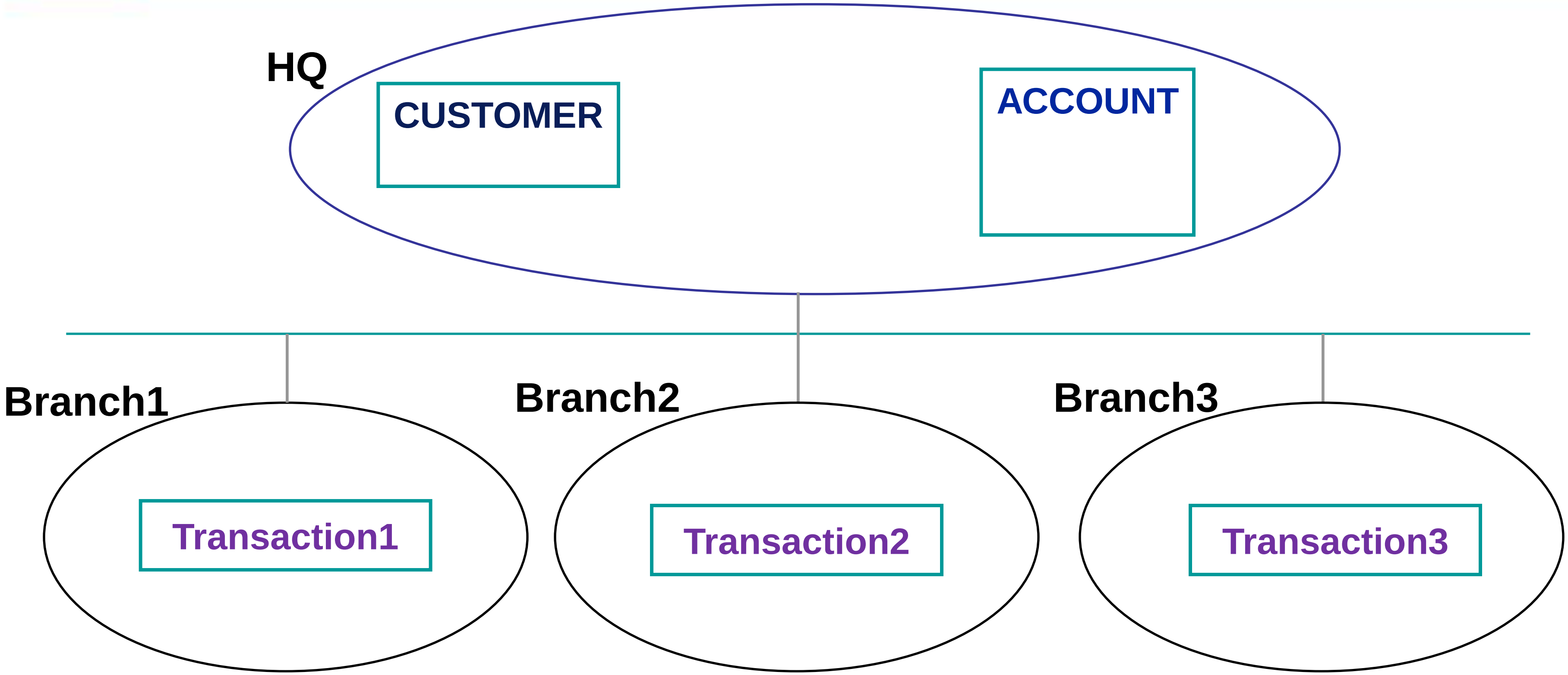
Centralized and distributed (fully replicated) [B]



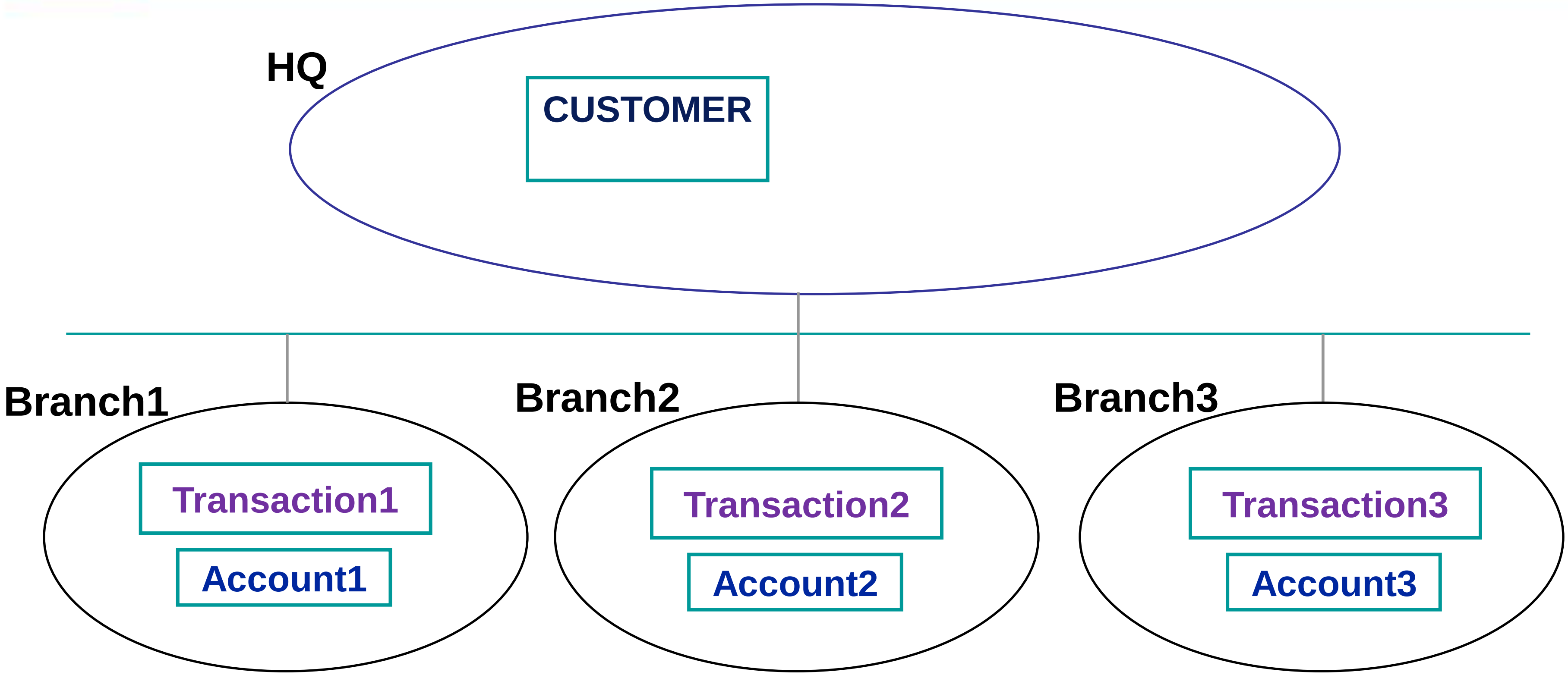
Centralized and distributed (fully replicated) [Bbis]



Partially distributed, no replication [C]

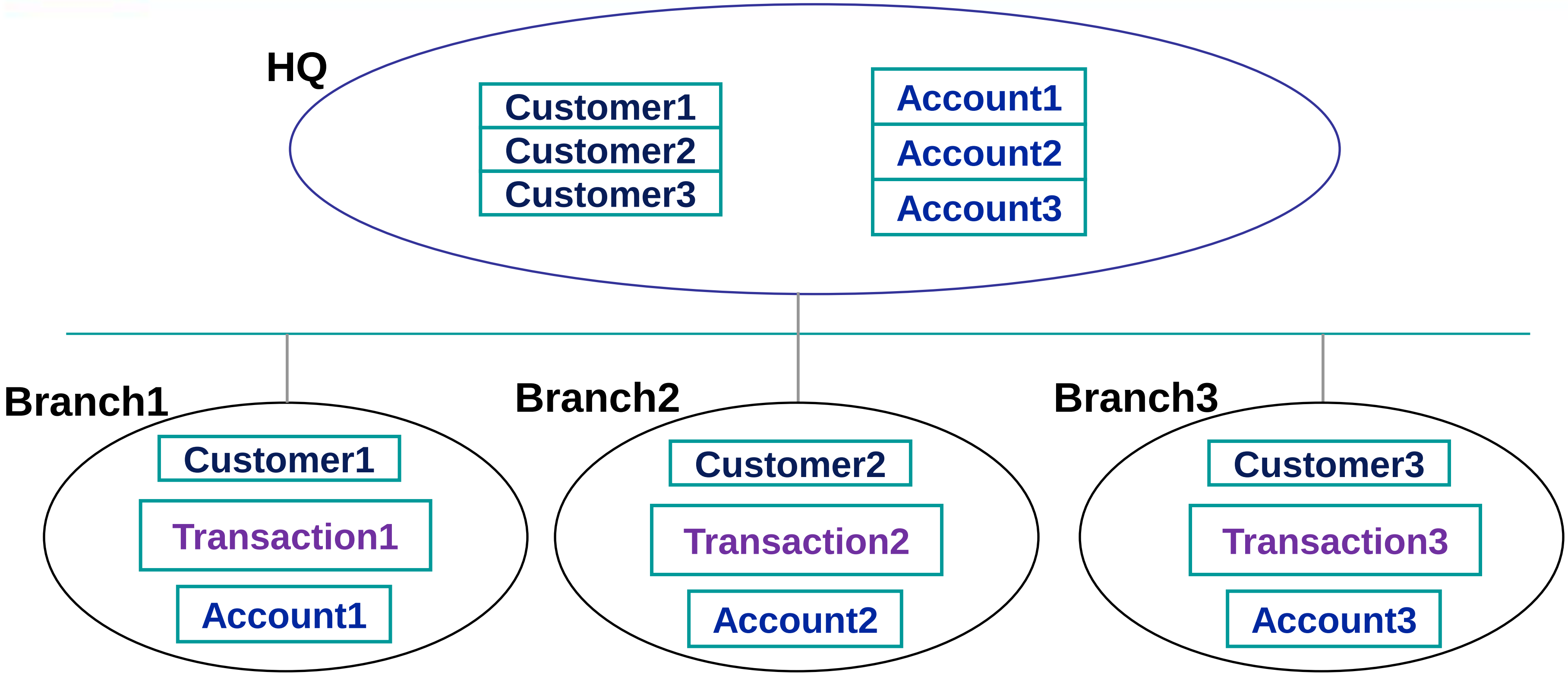


Partially distributed, no replication [D]

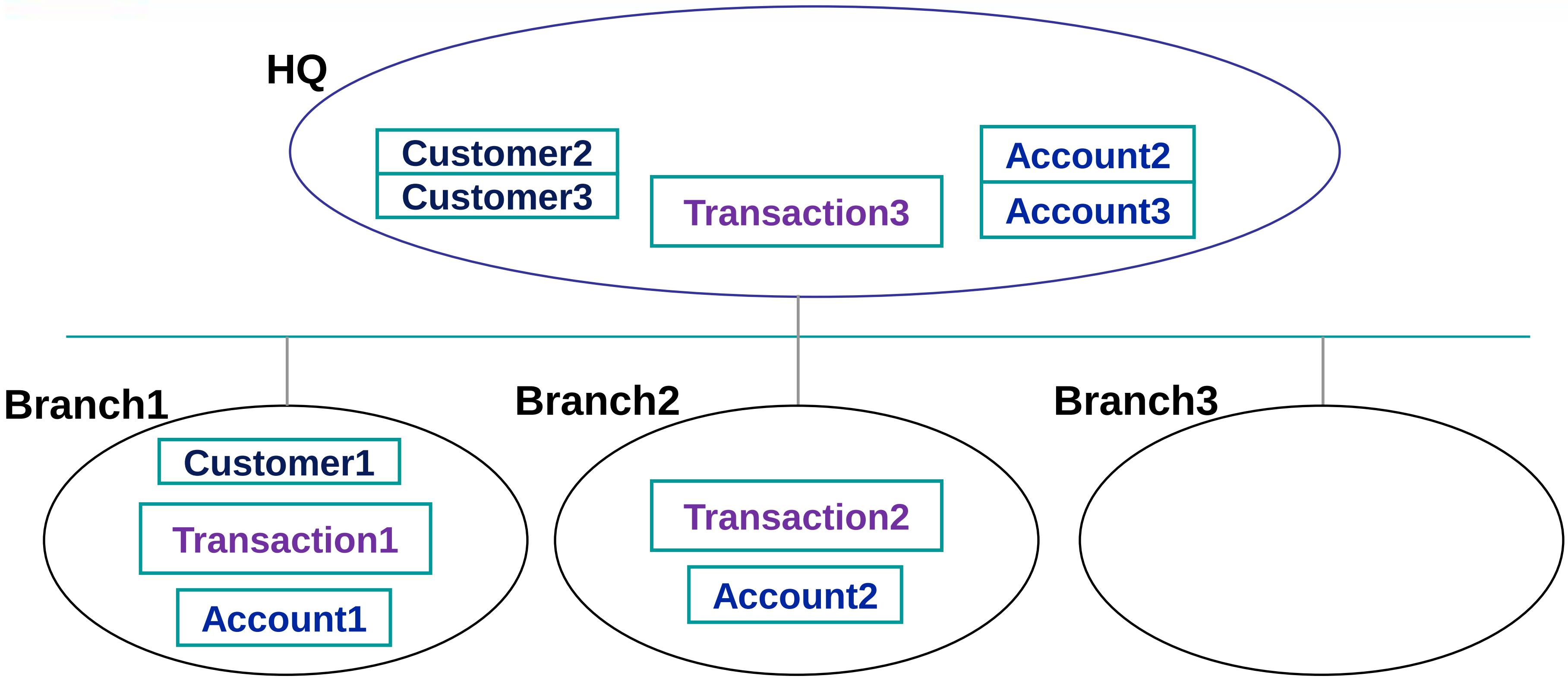


5 Distributed Databases

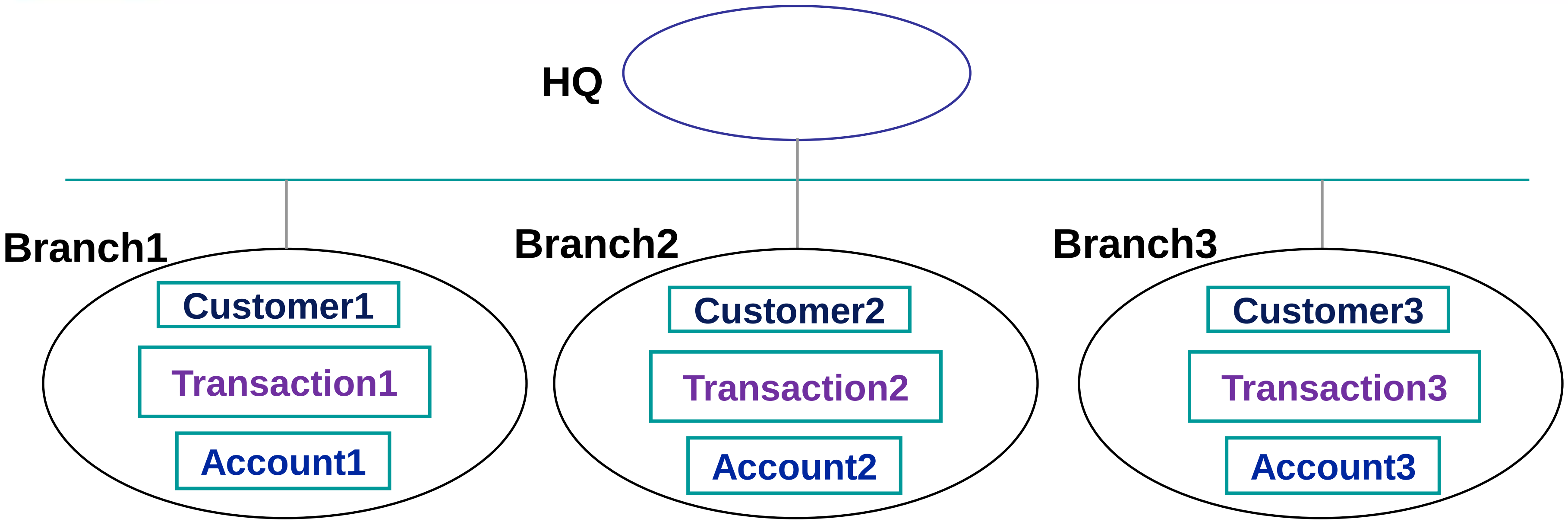
Partially centralized and fully distributed [E]



Asymmetric allocation [F]



Fully distributed (little replication) [G]



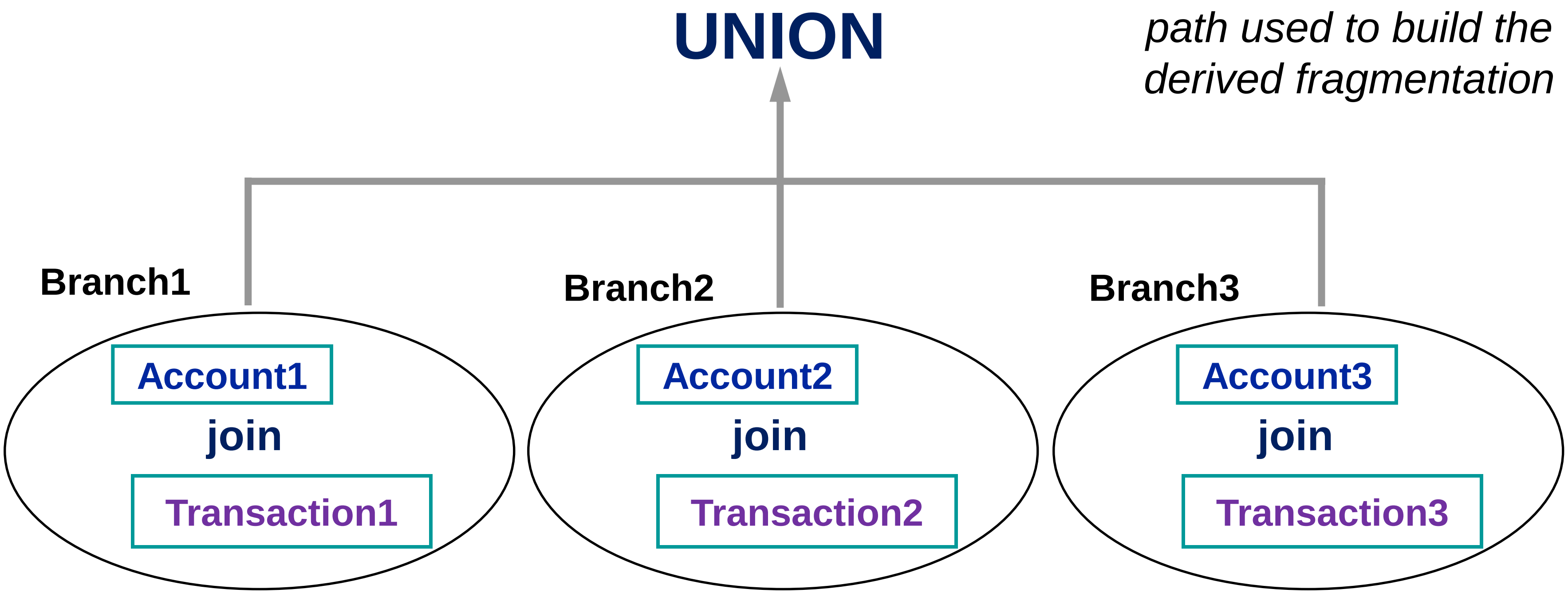
Customers owning more than one account are replicated on all branches on which they own at least one account

Distributed Join

- The most expensive distributed data analysis operation
- Consider a natural and frequent join operation:

Distributable Join

The join is along the join path used to build the derived fragmentation

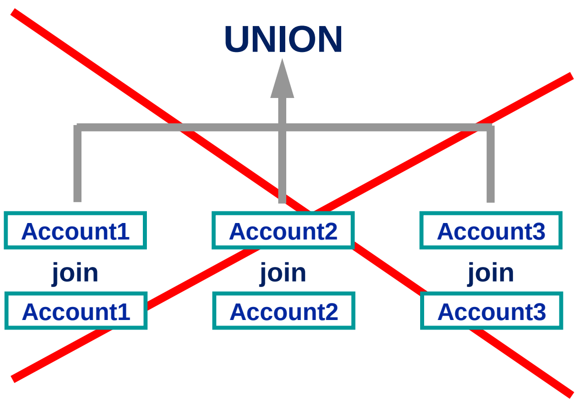


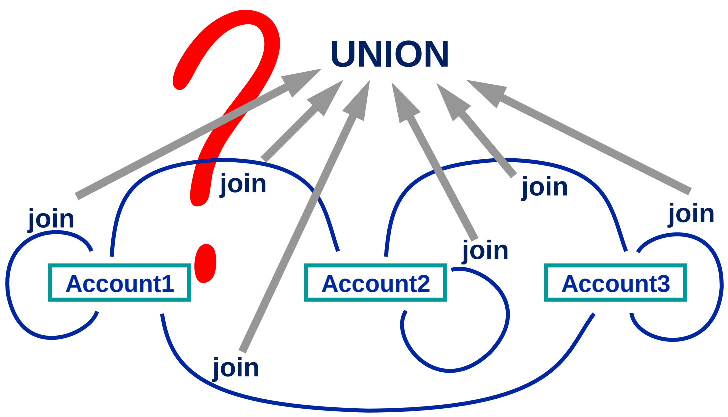
Requirements for Distributed Join

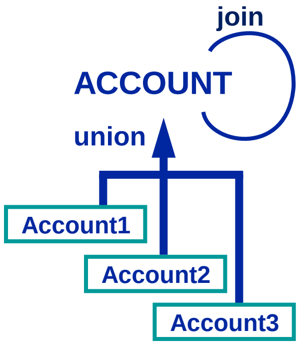
- The domains of the **join attributes** must be **partitioned** and each partition must be assigned to a couple of fragments
- Example: for numeric values between 1 and 30,000:
  - Partition 1 to 10,000
  - Partition 10,001 to 20,000
  - Partition 20,001 to 30,000
- Some parallel systems distribute the data on the disks at the beginning, to obtain this distribution

Problematic examples: a non-distributable join

- Problematic Query: **customers with more than one account**
  - A self join on ACCOUNT
- The self join is not the one used to guide the fragmentation
  - Couples of matching accounts can be on any node







Problematic examples: a problematic fragmentation

- Problematic fragmentation
  - We extend the database with the following table, tracing couples of transactions that are *internal* money transfers (both the sender and receiver are customers of the bank)

INTERNALTRANSFER(<u>Date</u>, <u>AccNoFrom</u>, <u>IncFrom</u>, AccNoTo, IncTo)

  - How to derive a fragmentation from ACCOUNT?
    - Based on the sending account? Or the receiving one?
    - What if we base it on both?
      - Both accounts may be on the same node, or different nodes...

Transparency Levels

- Different ways for composing queries, offered by commercial databases
- Three significant levels of transparency:
  - Transparency of fragmentation
  - Transparency of allocation
  - Transparency of language
- In *absence of transparency*, each DBMS accepts its own SQL ‘dialect’
  - The system is heterogeneous and the DBMSs do not support a common interoperability standard

Transparency of Fragmentation

- Query:
  - Extract the balance of the account 45

```
SELECT Balance
FROM Account
WHERE Number=45
```

Transparency of Allocation

- Hyp.:
  - Account 45 is subscribed at Branch 1 (local)

```
SELECT Balance
FROM Account1
WHERE Number=45
```

Transparency of Allocation

- Hyp.:
  - Allocation of Account 45 is unknown, but possibly it is located at Branch 1

```
SELECT Balance FROM Account1
WHERE Number=45
IF (NOT FOUND) THEN
( SELECT Balance FROM Account2
  WHERE Number=45
    UNION
  SELECT Balance FROM Account3
  WHERE Number=45 )
```

Transparency of Language

```
SELECT Balance FROM Account1@1
WHERE Number=45
IF (NOT FOUND) THEN
( SELECT Balance FROM Account2@C
  WHERE Number=45
     UNION
  SELECT Balance FROM Account3@C
  WHERE Number=45 )
```

Transparency of Fragmentation

- Query:
  - Extract the transactions of the accounts with negative balance

```
SELECT Number, Incremental, Amount
FROM Account AS C
  JOIN Transaction AS T
    ON C.Number=T.AccountNumber
WHERE Balance < 0
```

Transparency of Allocation (distributed join)

```
SELECT  Number, Incremental, Amount
        FROM Account1 JOIN Trans1 ON ……
        WHERE Balance < 0
UNION
SELECT Number, Incremental, Amount
        FROM Account2 JOIN Trans2 ON ……
        WHERE Balance < 0
UNION
SELECT Number, Incremental, Amount
        FROM Account3 JOIN Trans3 ON ……
        WHERE Balance < 0
```

Transparency of Language

```
SELECT  Number, Incremental, Amount
          FROM Account1@1 JOIN Trans1@1 ON ......
          WHERE Balance < 0
UNION
SELECT Number, Incremental, Amount
          FROM Account2@C JOIN Trans2@C ON ......
          WHERE Balance < 0
UNION
SELECT Number, Incremental, Amount
          FROM Account3@C JOIN Trans3@C ON ......
          WHERE Balance < 0
```

Transparency of Fragmentation

- Update:
  - Move Account 45 from Branch 1 to Branch 2

```
UPDATE Account
SET Branch = 2
WHERE Number = 45
    AND Branch = 1
```

Transparency of Allocation (and Replication)

```
INSERT INTO Account2
  SELECT  Number, Name, 2, Balance FROM Account1 WHERE Number = 45

INSERT INTO Trans2
  SELECT * FROM Trans1 WHERE AccountNumber = 45

DELETE FROM Trans1 WHERE AccountNumber = 45

DELETE FROM Account1 WHERE Number = 45
```

Transparency of Language

```
INSERT INTO Account2@2
  SELECT Number, Name, 2, Balance FROM Account1@C WHERE Number=45

INSERT INTO Account2@C
  SELECT Number, Name, 2, Balance FROM Account1@C WHERE Number=45

INSERT INTO Trans2@2
  SELECT * FROM Trans1@C WHERE Number=45
INSERT INTO Trans2@C
  SELECT * FROM Trans1@C WHERE Number=45
```

(similarly: 2 pairs of DELETE commands)

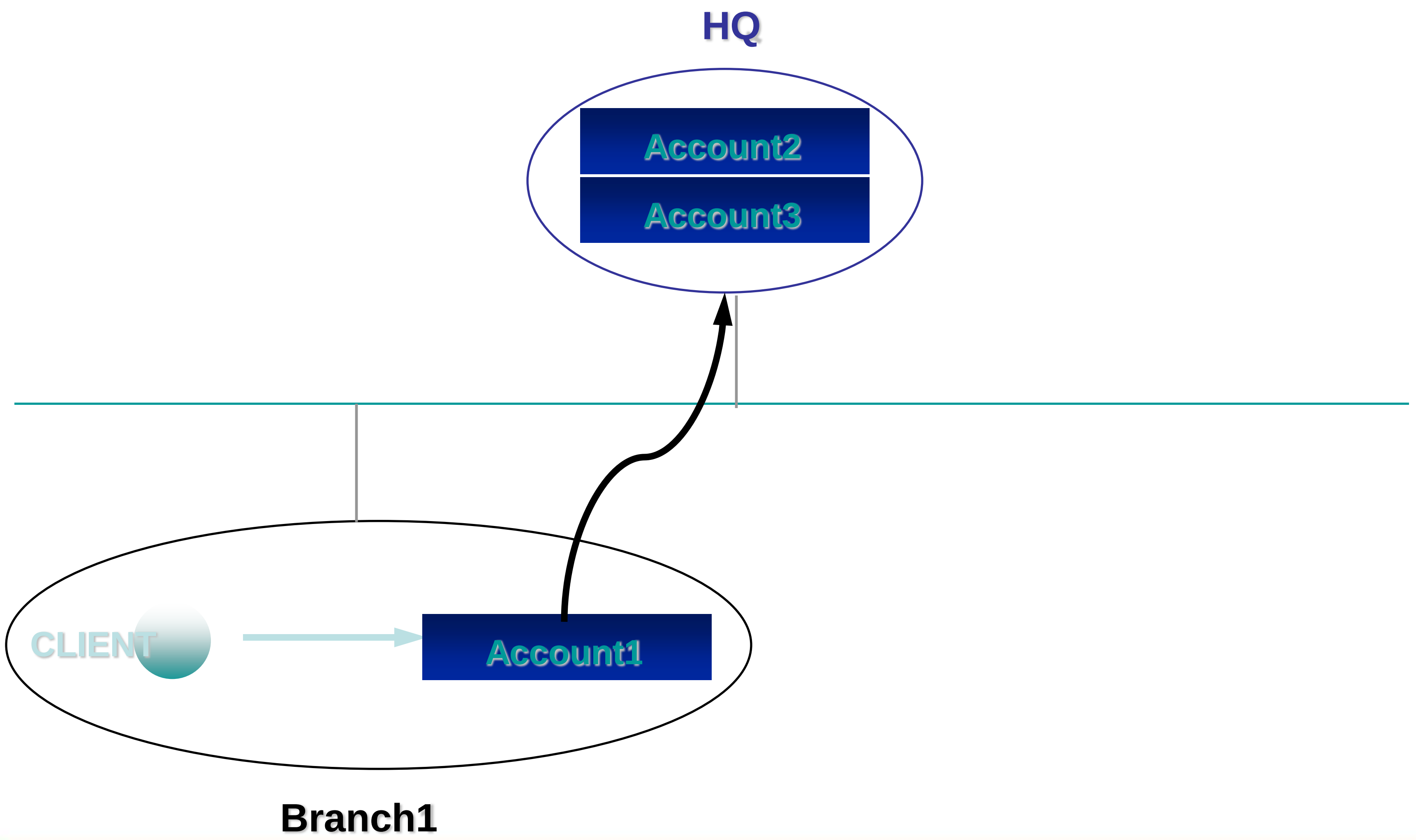
Distribution Design Problem

- Determining the best fragmentation and allocation of given tables
- Fragmentation should match locality characteristics, but there are trade offs.
  - With a university database, STUDENT allocated at the central admission office, COURSE distributed at the departments.
  - How should STUDY-PLAN & EXAM be fragmented?
  - Depending on the choice, ONE of the two joins with either STUDENT or COURSE is a distributed join, the other one is not.
- Allocation should give the ideal degree of redundancy
  - Redundancy speeds up retrieval and slows down updates
  - Redundancy increases availability and robustness

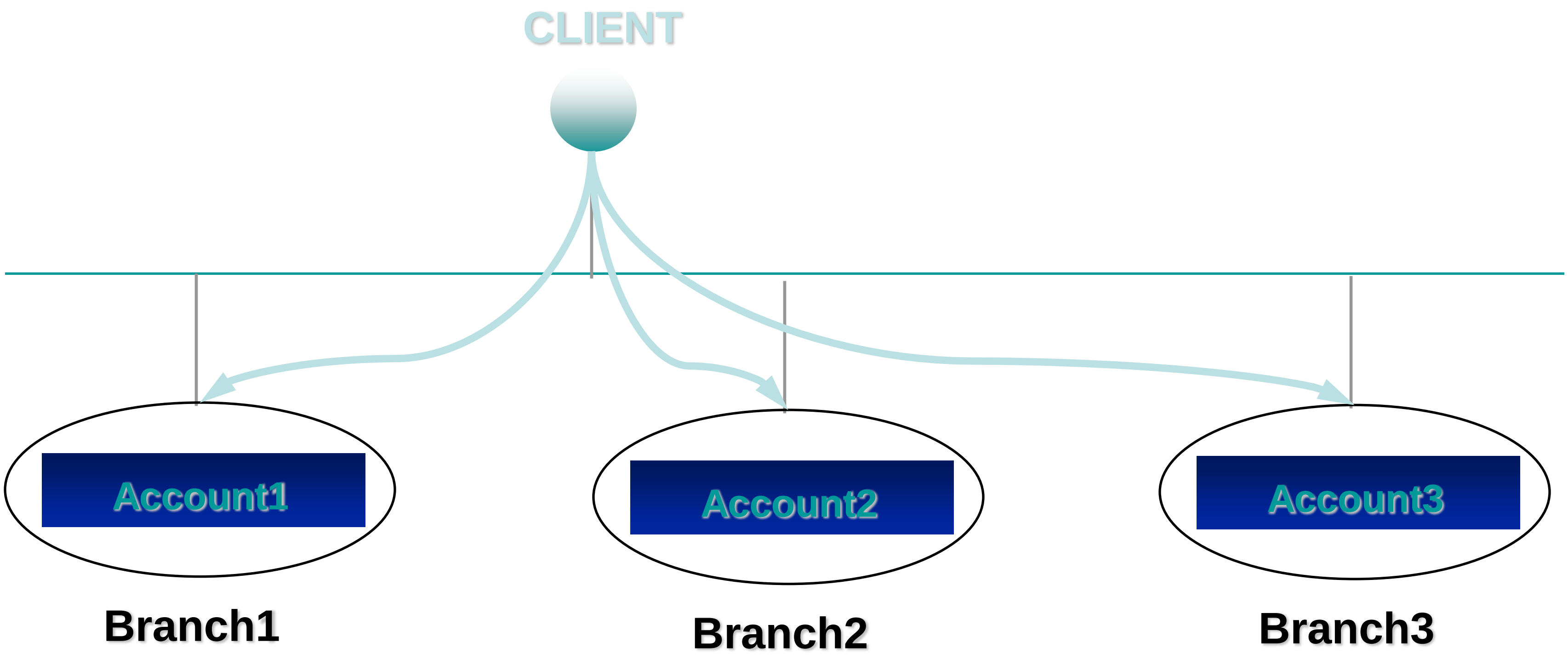
Efficiency

- Query optimization
- Execution time
    - Serial execution
    - Parallel execution

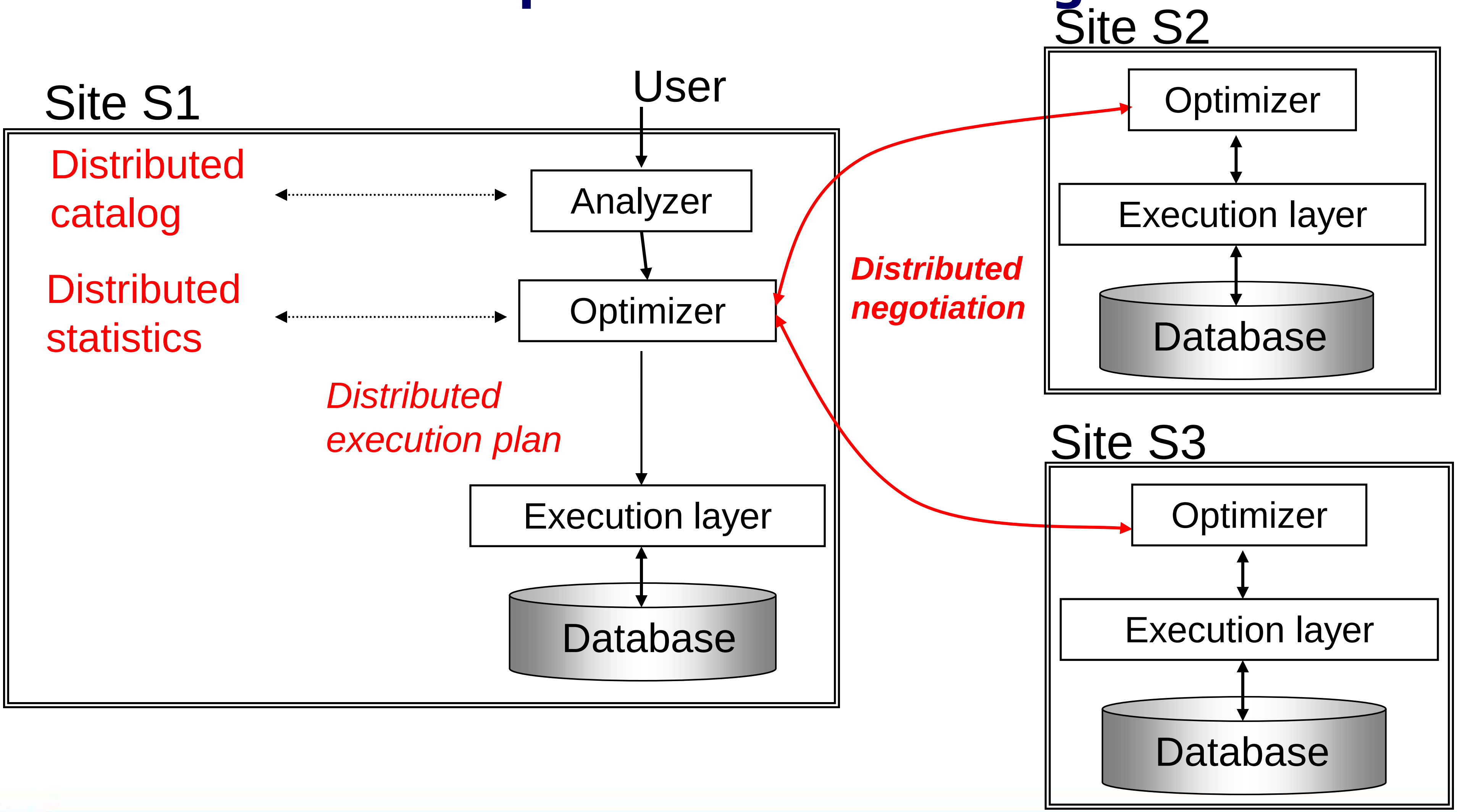
Serial Execution



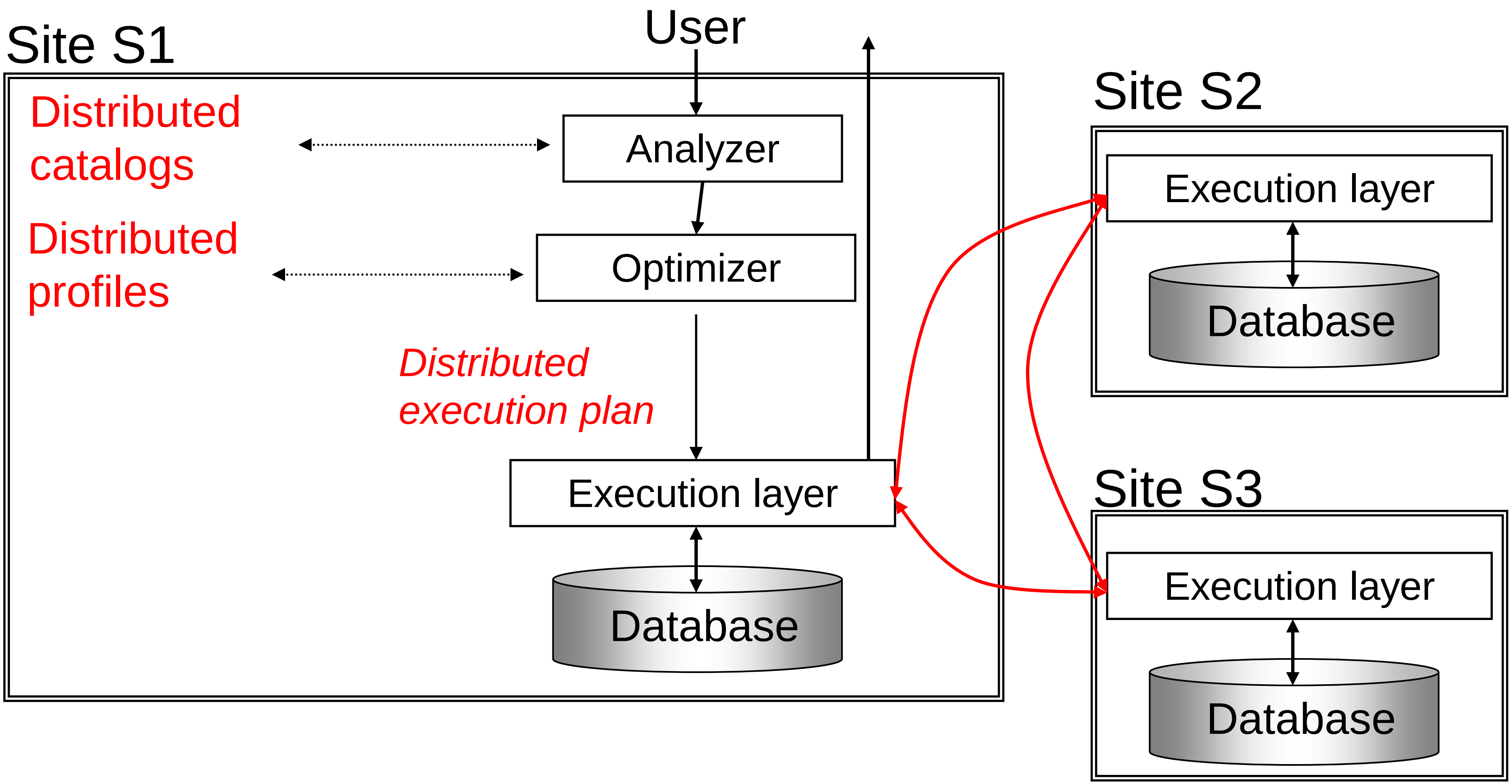
Parallel Execution



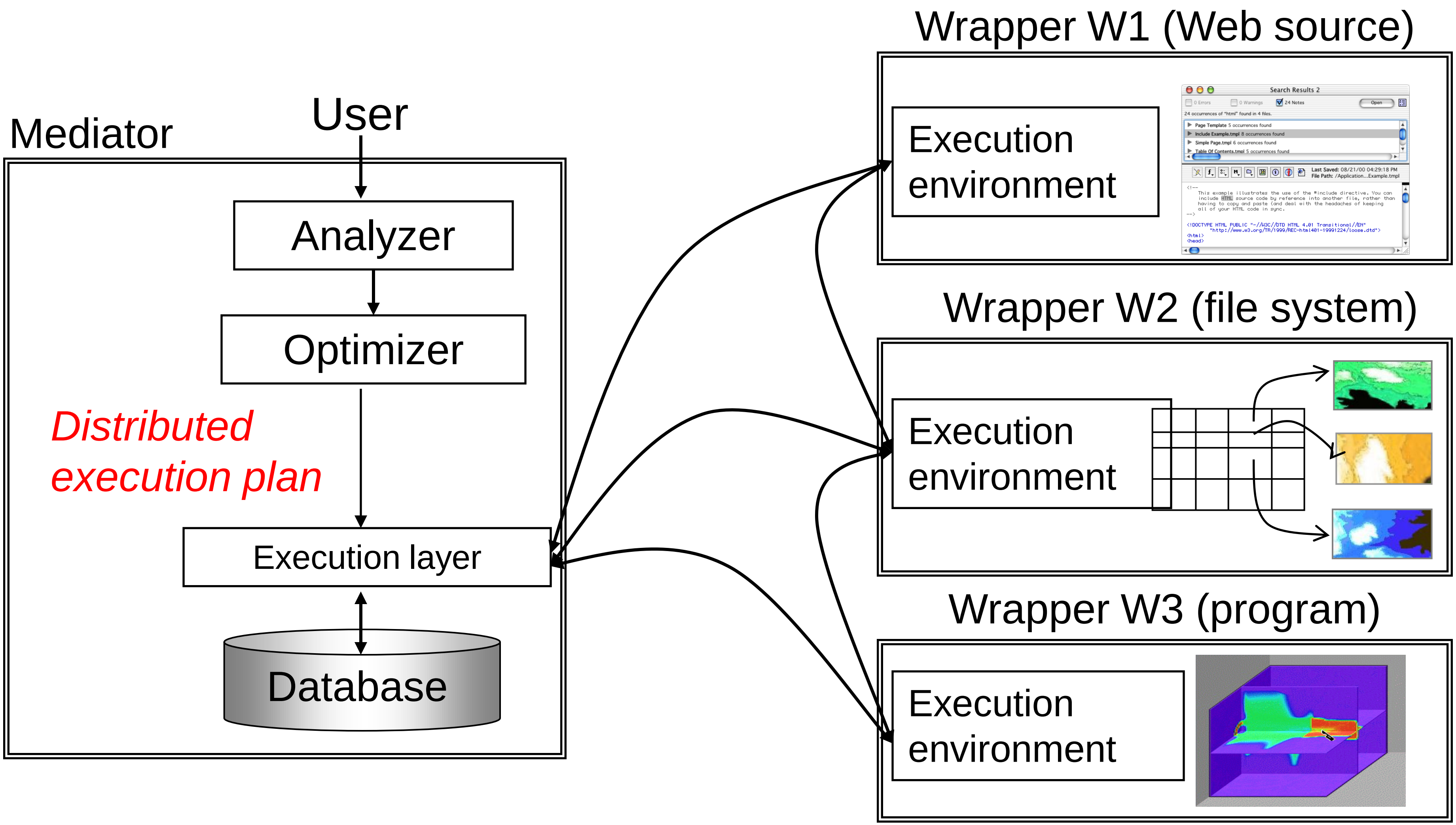
Distributed optimization with negotiation



Distributed database with master-slave optimization



Distributed system with mediator and wrappers



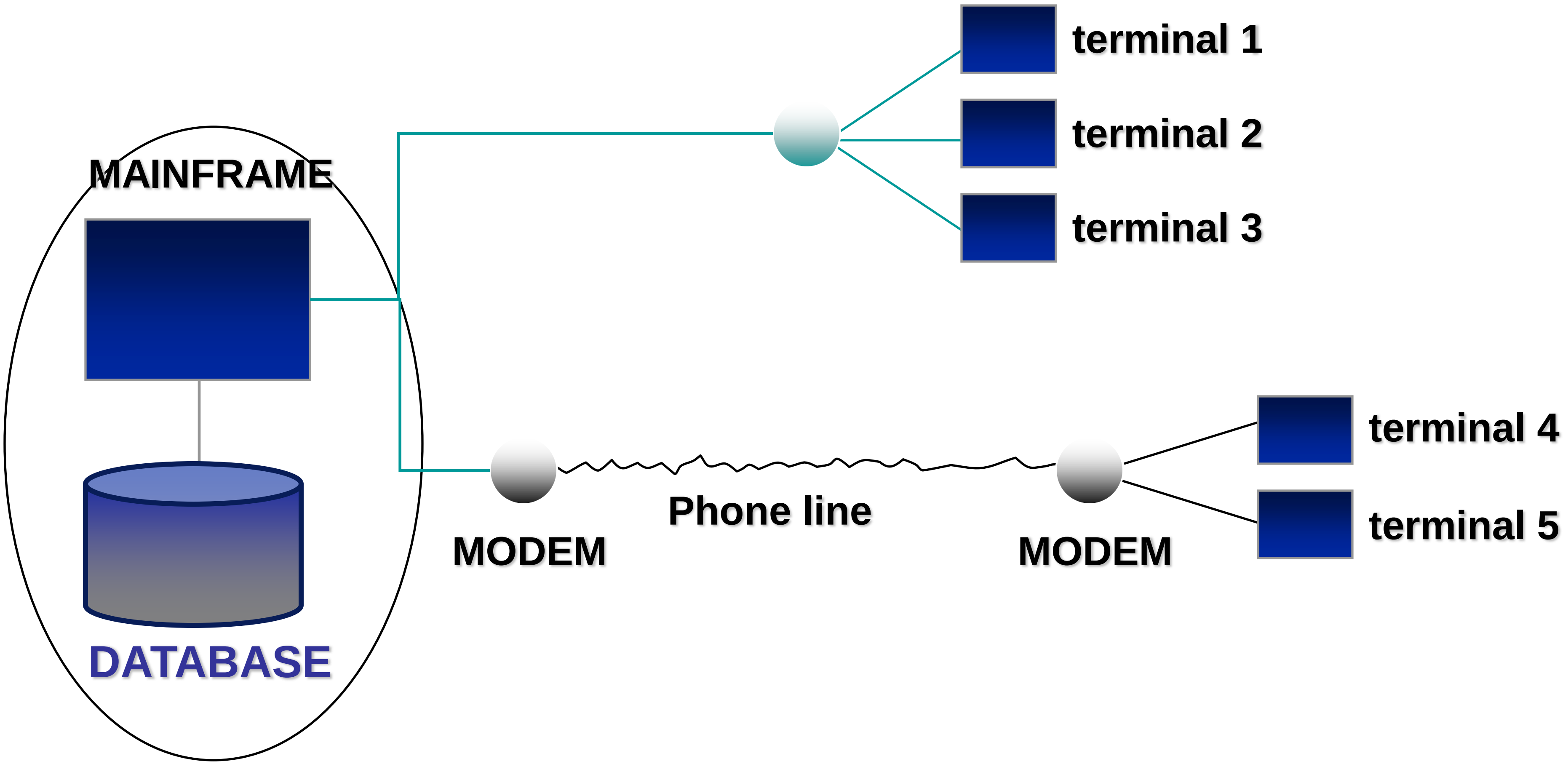
A pragmatic classification of distributed SQL applications

- **Remote request**: ability to support read-only SQL queries to a single remote database
- **Remote transaction**: ability to support SQL update queries to a single remote database
  - These are easily managed by ODBC/JDBC interfaces
- **Distributed transaction**: ability to support update transactions to many distributed database, every SQL query routed to a single database
  - Requires 2 phase commit
- **Distributed request**: ability to support update transactions to many distributed database, every SQL query possibly over many databases
  - Requires 2 phase commit and distributed query optimization

Legacy systems

- System architectures based on **mainframes** (powerful centralized computer), clients consist in very simple terminals (with textual interface)
- In many cases there is no source code or no documentation, thus they are very hard to change
- However they provide adequate performance and availability, that was obtained thanks to huge efforts on solid technologies
- Typically they are obsolete systems, but they still manage important applications: large banking applications, financial applications, flight booking systems

Legacy Systems



Legacy System

Obsolete Technologies of Legacy Systems

- Hardware (high $ cost for slow but very reliable performance)
- Software (Cobol, DL/1: 1960-1980)
- On separated archives (even without DBMS)

- However, legacy systems are still reliable for operations that require continuous 7days 24h availability

Gateway (Wrapper)

- Software system that:
  - Is able to transfer requests from a (input) context to another (output) context
  - Provides server capabilities to the input system and client capabilities to the output system
  - Performs the needed conversion for the different formats and languages of the two contexts

Usage of Gateway Systems

- Between transactional systems
- Towards legacy systems