Databases 2

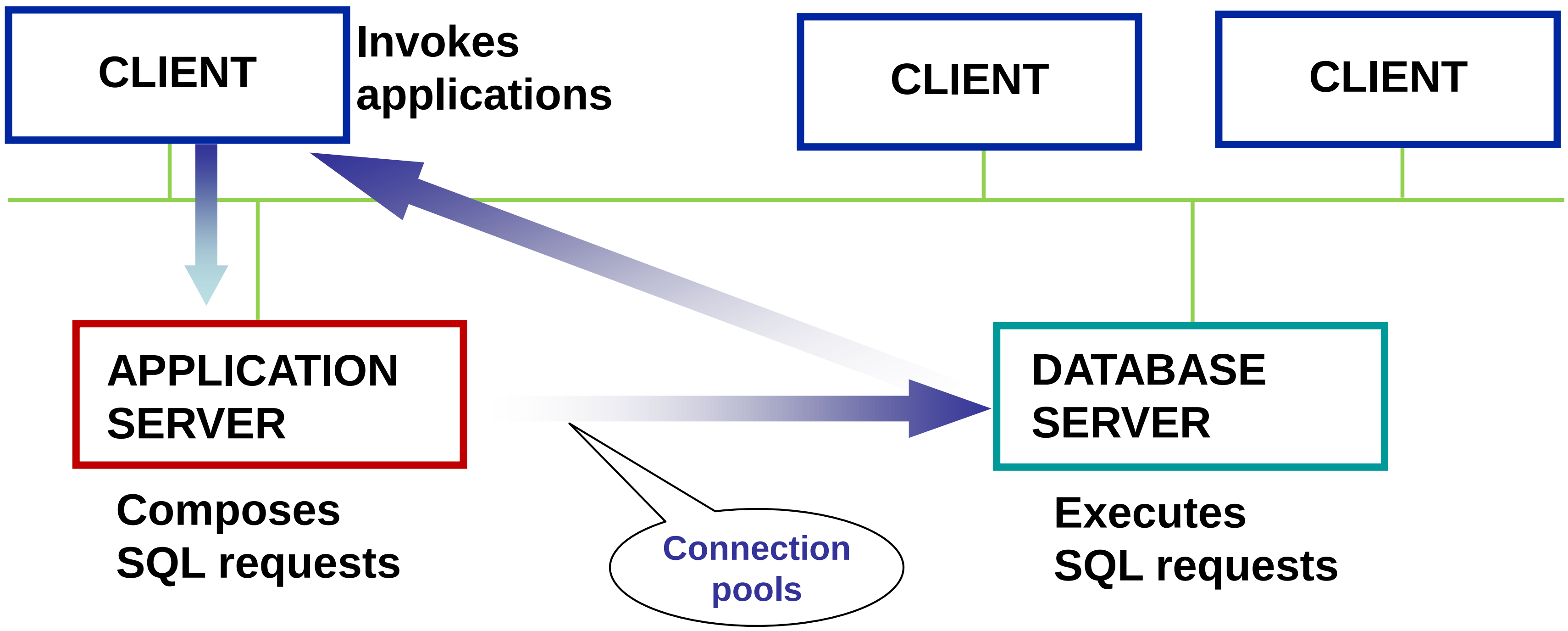
4

Distributed Databases

The Client-Server paradigm in Information Systems

- Client-Server: a well known paradigm in system design
  - Two systems are involved:
    - Clients **invoke** services **provided** by Servers
- In DB & Information systems the functional separation is ideal
  - **Clients**: for the presentation layer
  - **Servers**: for data management
- SQL: the perfect language for enacting this separation
  - **Clients**: formulate queries and show results
  - **Servers**: execute queries and calculate results
  - Network: in charge of transferring activation commands (e.g., of SQL procedures) and returning query results

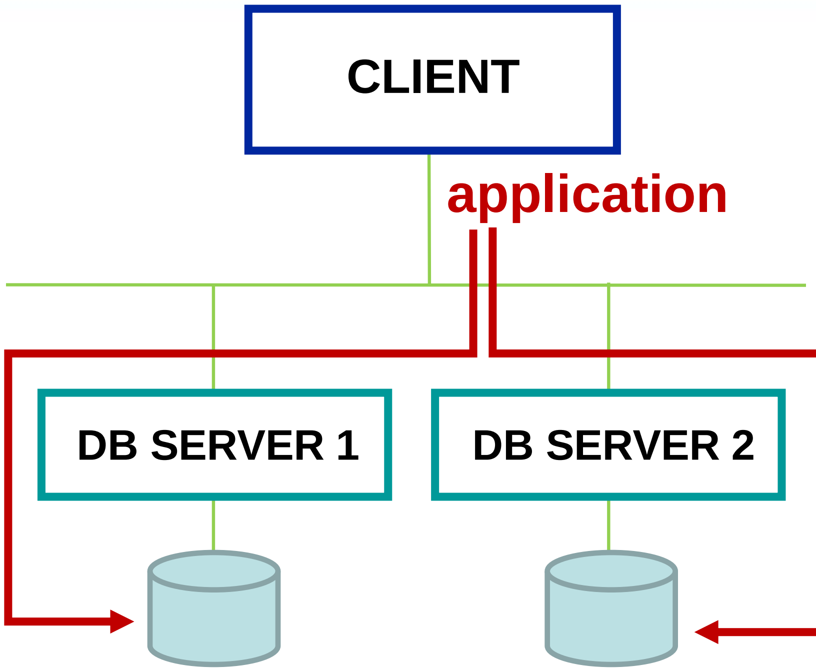
Typical Application Server Architecture



Data distribution

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

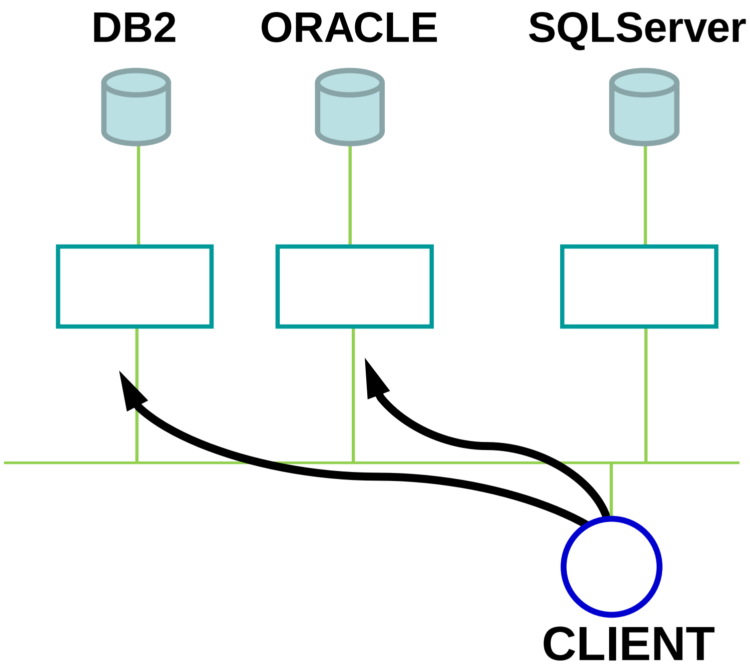
→Distributed Databases



Distributed Database Types and Applications

Classification of systems based on the involved databases:

- Homogeneous system:
  - all the same DBMS
  - Typical applications:
    - Intra-division company management, Travel management, financial applications
- Heterogeneous system:
  - Various DBMS
  - Typical applications:
    - Inter-division company management, integrated booking systems, inter-banking systems



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 F_i
  - **Restorability**: the content of a table T must be restorable from its fragments F_i

Table

PK	A	B	C	D	E	F	G

Horizontal Fragmentation

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

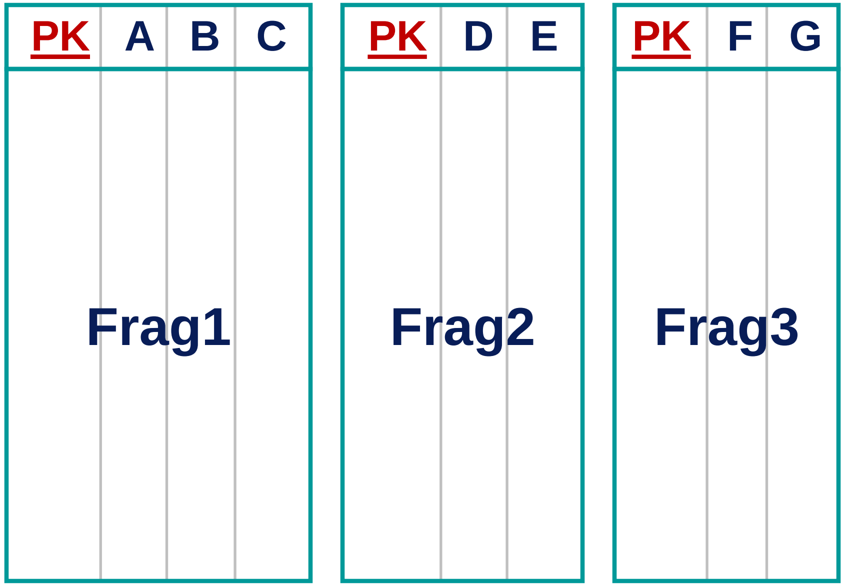
PK	A	B	C	D	E	F	G
Fragment1							

PK	A	B	C	D	E	F	G
Fragment2							

PK	A	B	C	D	E	F	G
Fragment3							

Vertical Fragmentation

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

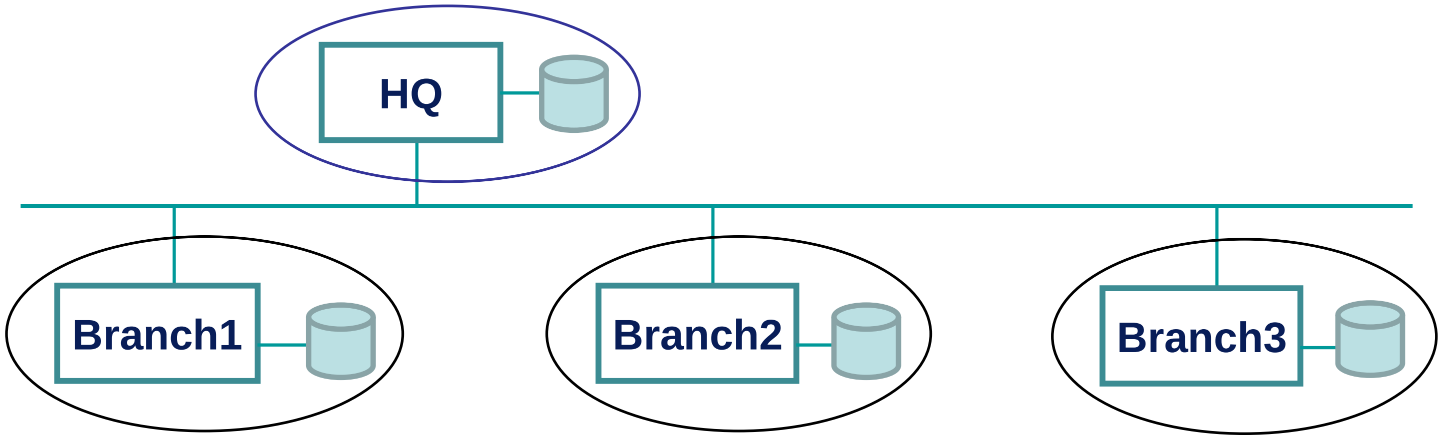


Example: bank accounts

CUSTOMER(<u>CustomerSSN</u>, Name, Address, Birthdate, email, telephone)

ACCOUNT(<u>Number</u>, CustomerSSN, Branch, Balance)

TRANSACTION(<u>AccountNumber</u>, <u>Date</u>, <u>Incremental</u>, Amount, Description)



Fragment Definition and Allocation

- **Network**:
  - 1 central node (in the Headquarters)
  - N peripherical nodes (one per branch) … [N=3]
- **Definition**
  - Fragments are defined by queries on the centralized DB
- **Allocation**:
  - Local to branches (distributed) vs
  - In the Headquarters (centralized)
- Many allocation schemes are possible
  - Each one with pros and cons

(Primary) Horizontal Fragmentation

$$R_i = \sigma_{P_i} R$$

Primary fragmentation
(a predicate P guides the fragmentation)

Example:

Account1 = $\sigma_{Branch=Branch1}$ ACCOUNT

Account2 = $\sigma_{Branch=Branch2}$ ACCOUNT

Account3 = $\sigma_{Branch=Branch3}$ ACCOUNT

In SQL:

Account $_i$:=
select *
from ACCOUNT
where Branch = "Branch $_i$"

Restorability

select * from Account1 union … select * from Account $_i$ union …

Derived Horizontal Fragmentation

$S_i = S \ltimes R_i$

Derived (secondary) **fragmentation**
a semijoin defines the fragmentation w.r.t. another already fragmented table (in turn, primary or derived)

Example:

Transaction1 = TRANSACTION ⋉ ACCOUNT1

Transaction2 = TRANSACTION ⋉ ACCOUNT2

Transaction3 = TRANSACTION ⋉ ACCOUNT3

Transactions are properly partitioned (**no overlap**)

Customer1 = CUSTOMER ⋉ ACCOUNT1

Customer2 = CUSTOMER ⋉ ACCOUNT2

Customer3 = CUSTOMER ⋉ ACCOUNT3

Customers may have more than one account (**overlap**)

Derived Horizontal Fragmentation

$$S_i = S \ltimes R_i$$

In SQL:

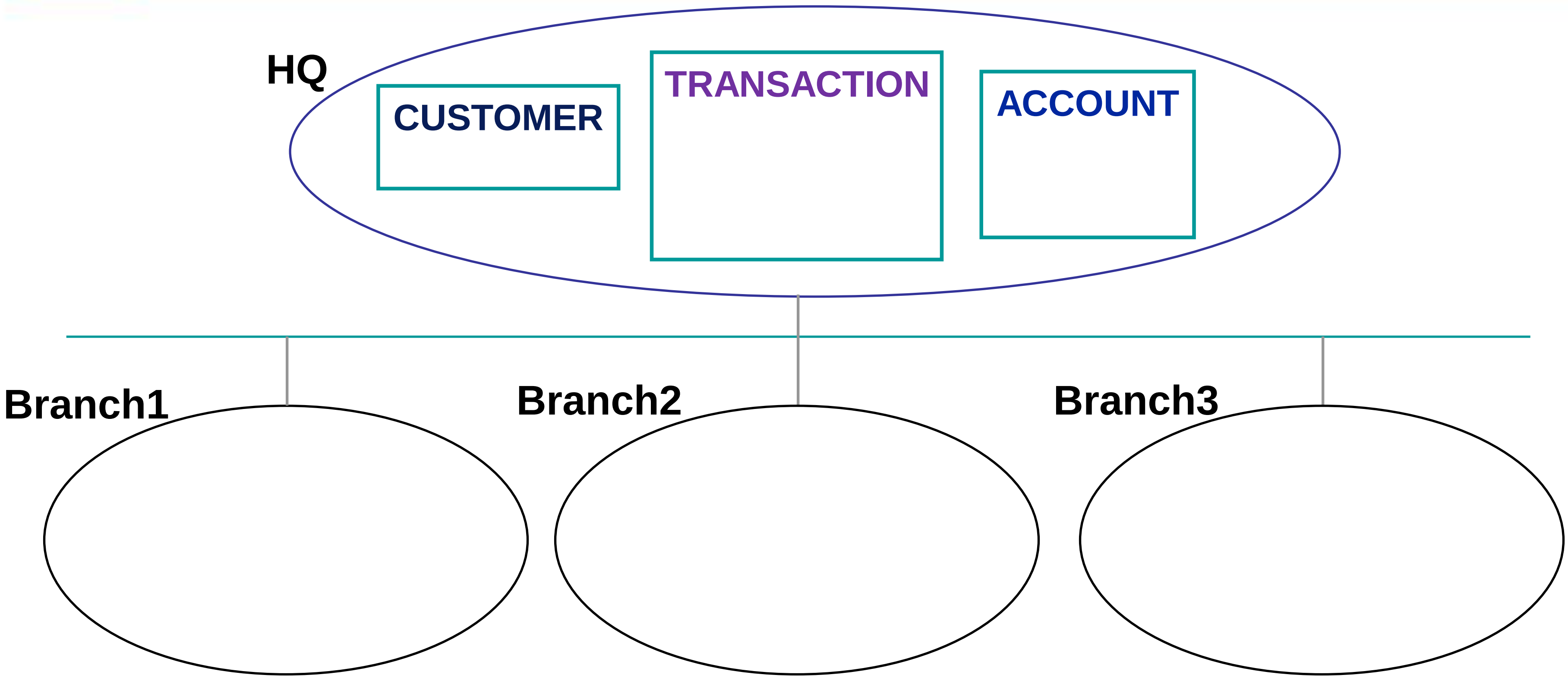
Transaction $_i$:=

```
select T.*
from  TRANSACTION T join Account i on AccountNumber = Number
```

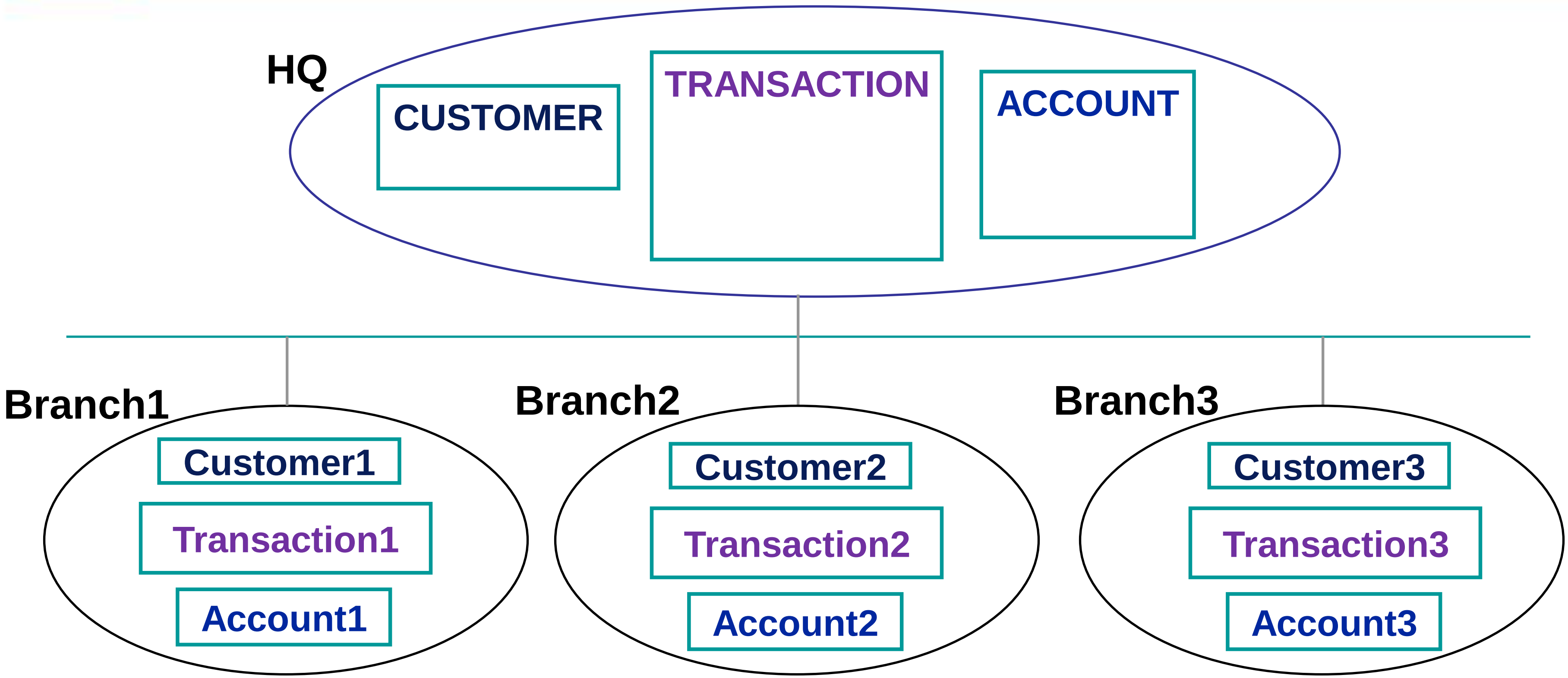
Customer $_i$:=

```
select distinct C.*
from  CUSTOMER C join Account i A on C.CustomerSSN = A.CustomerSSN
```

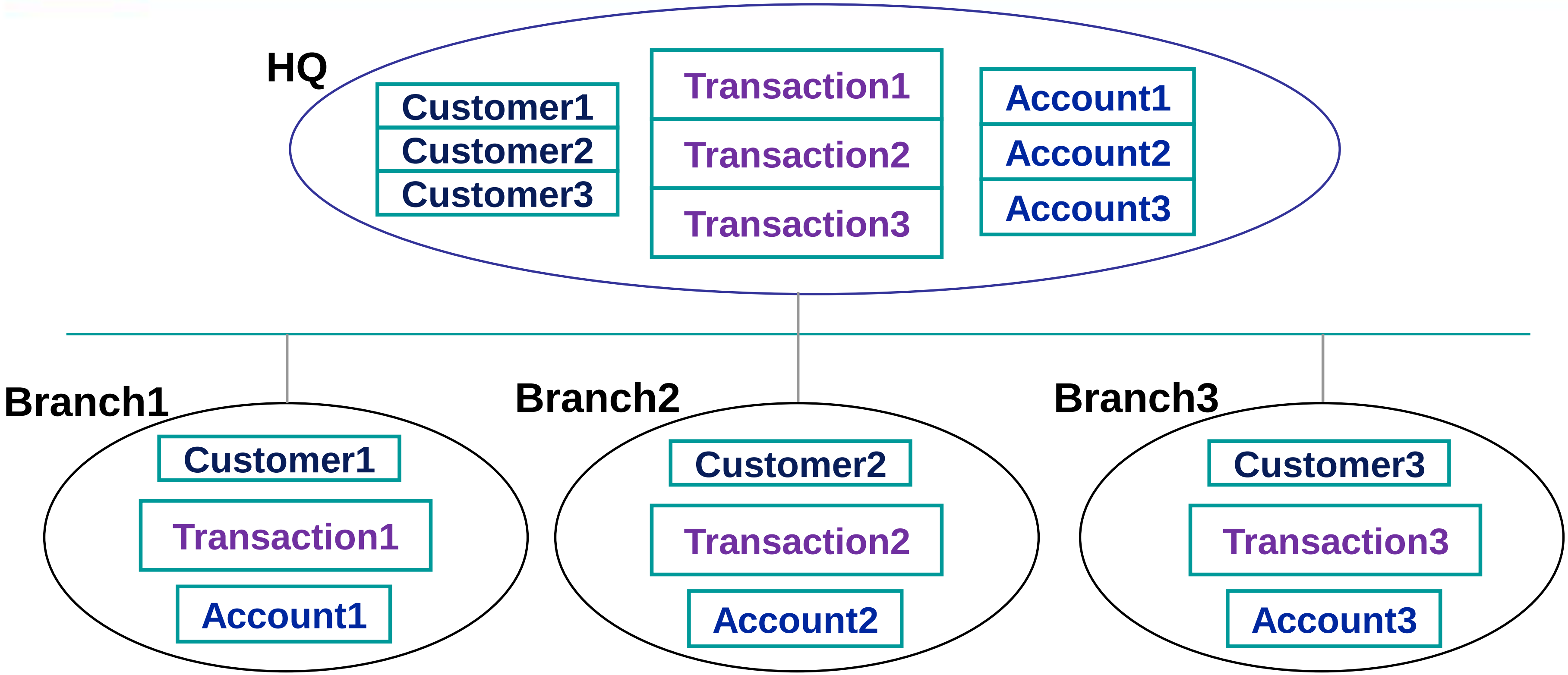
Fully centralized [A]



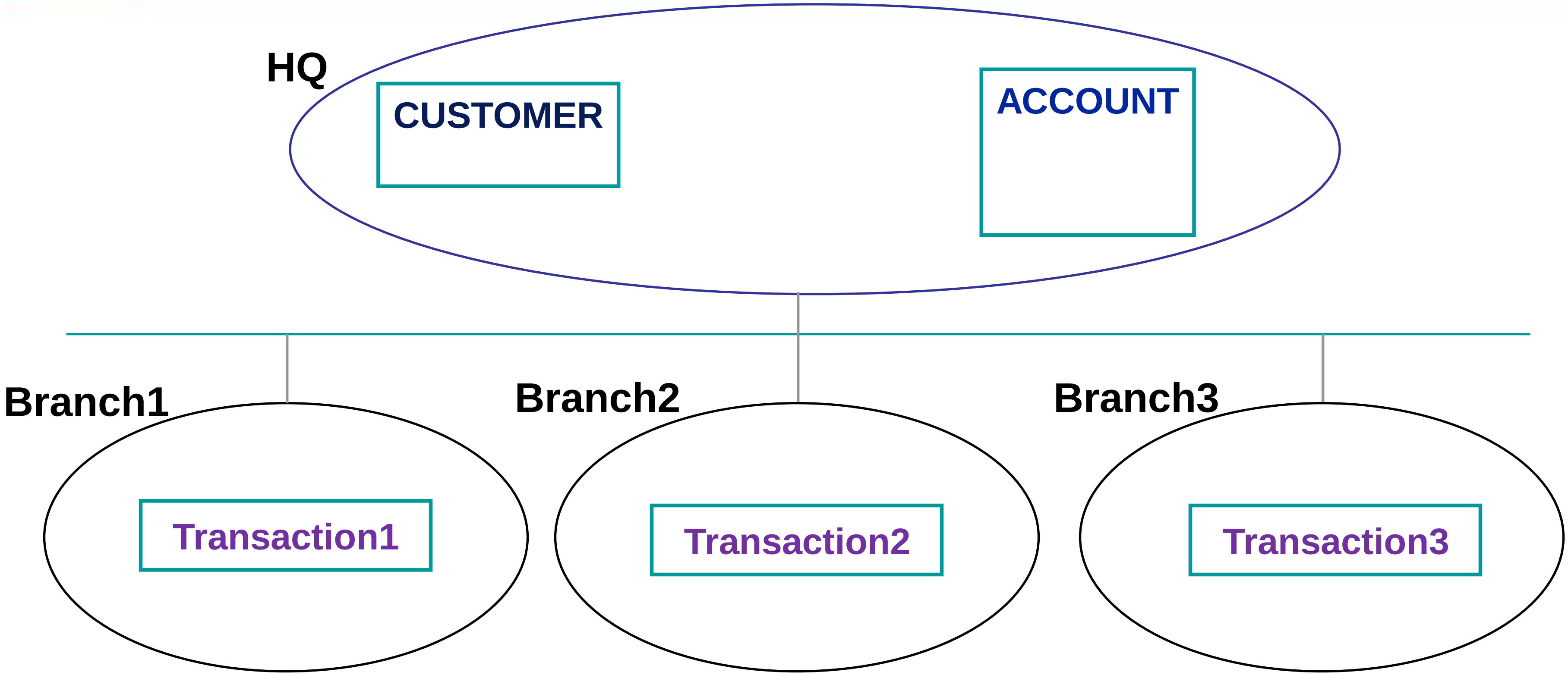
Centralized and distributed (fully replicated) [B]



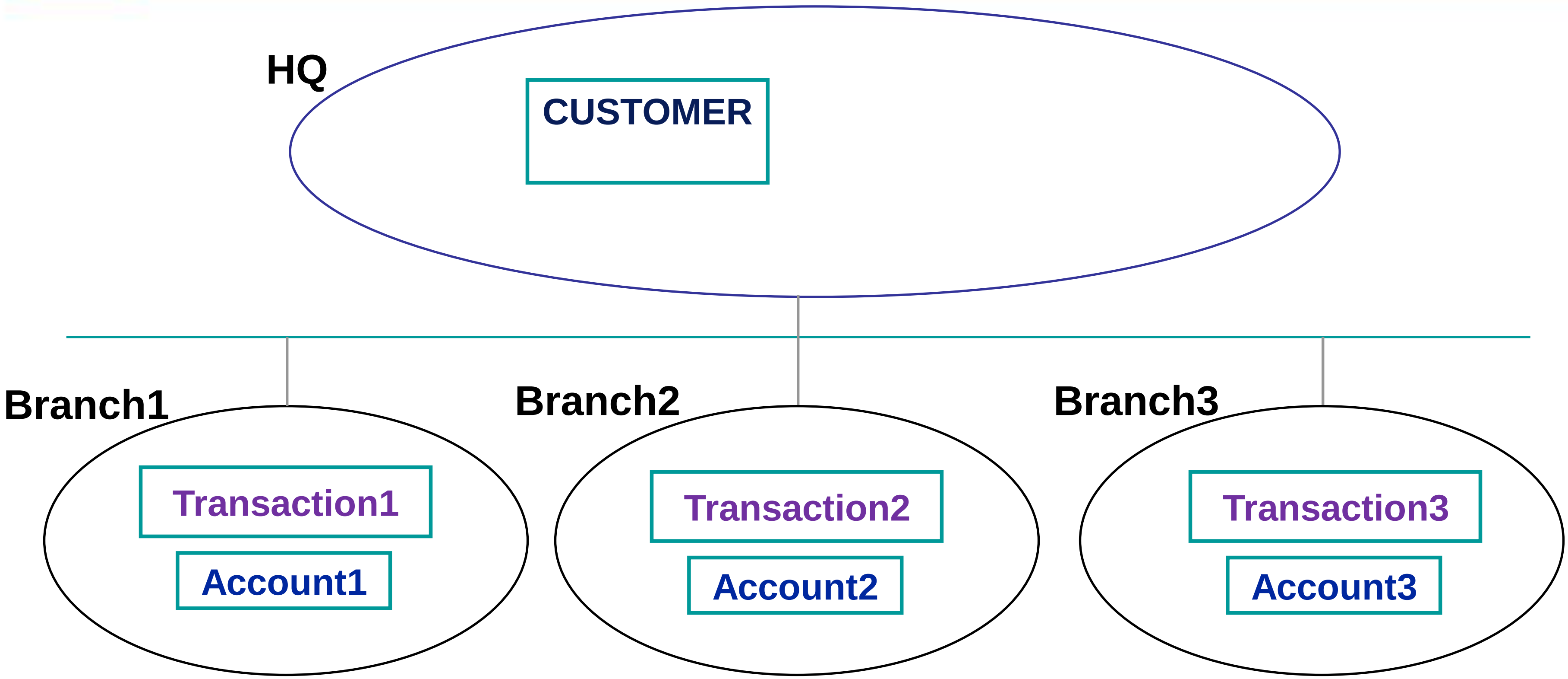
Centralized and distributed (fully replicated) [Bbis]



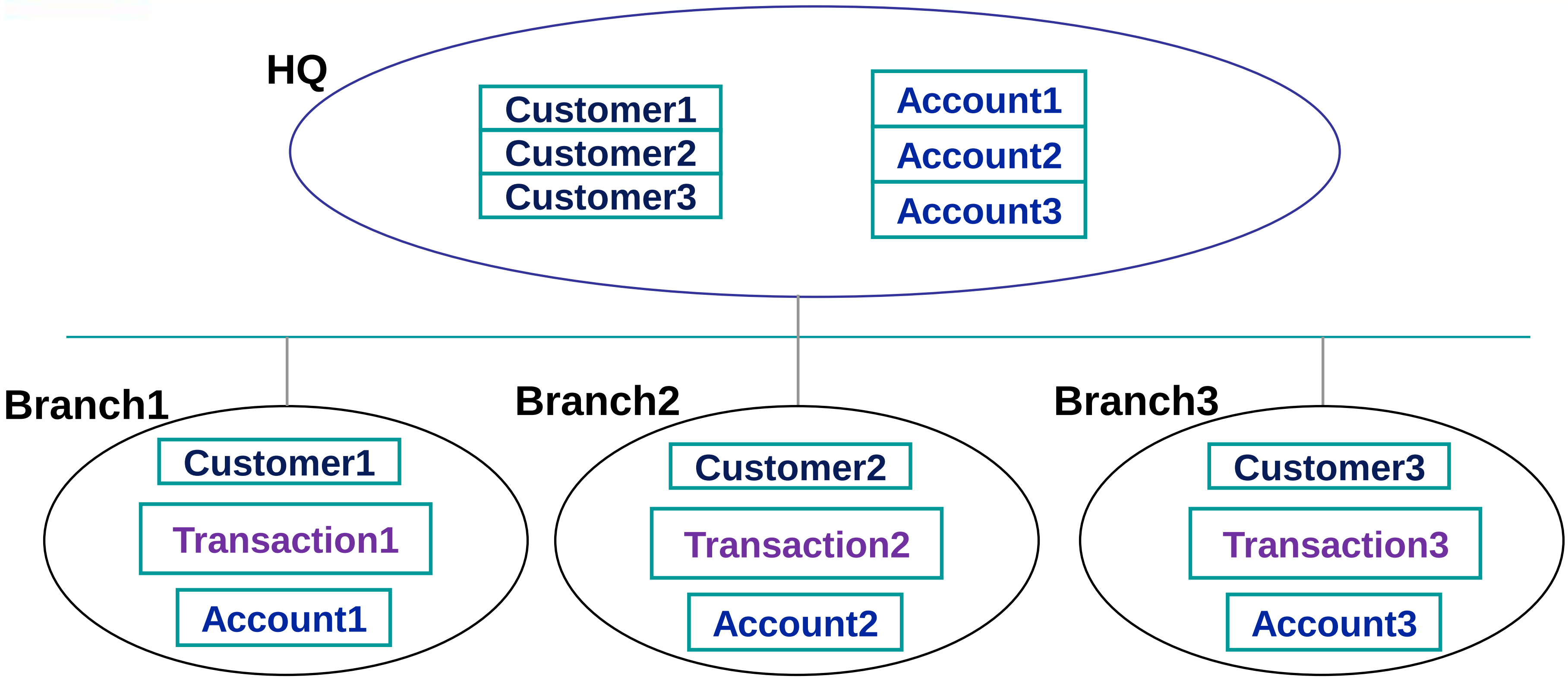
Partially distributed, no replication [C]



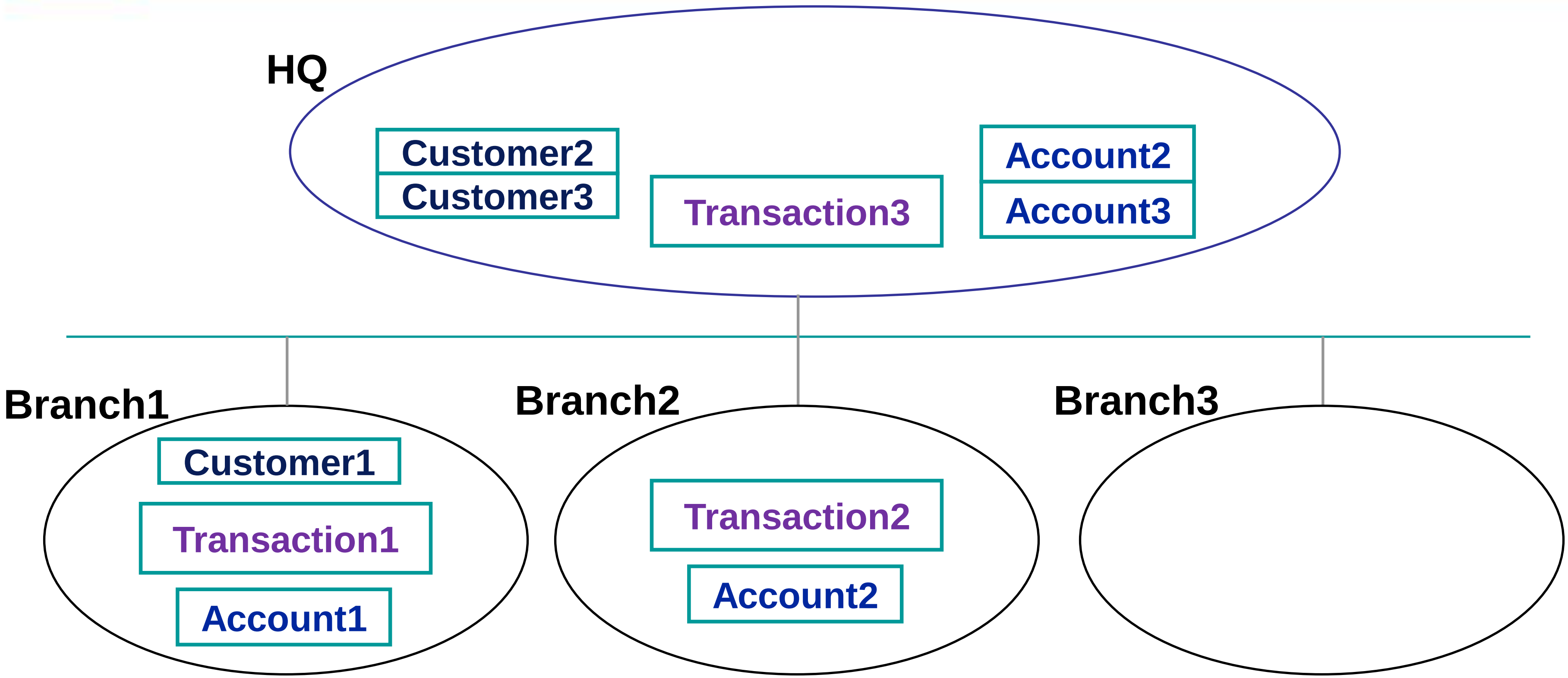
Partially distributed, no replication [D]



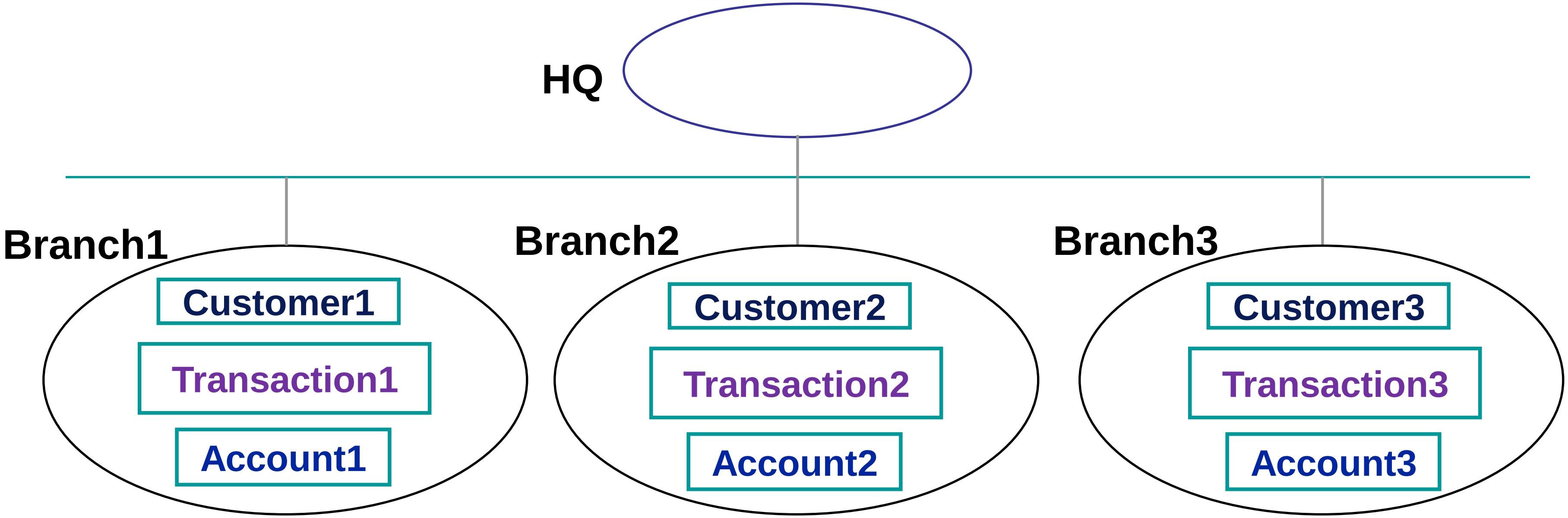
Partially centralized and fully distributed [E]



Asymmetric allocation [F]



Fully distributed (little replication) [G]



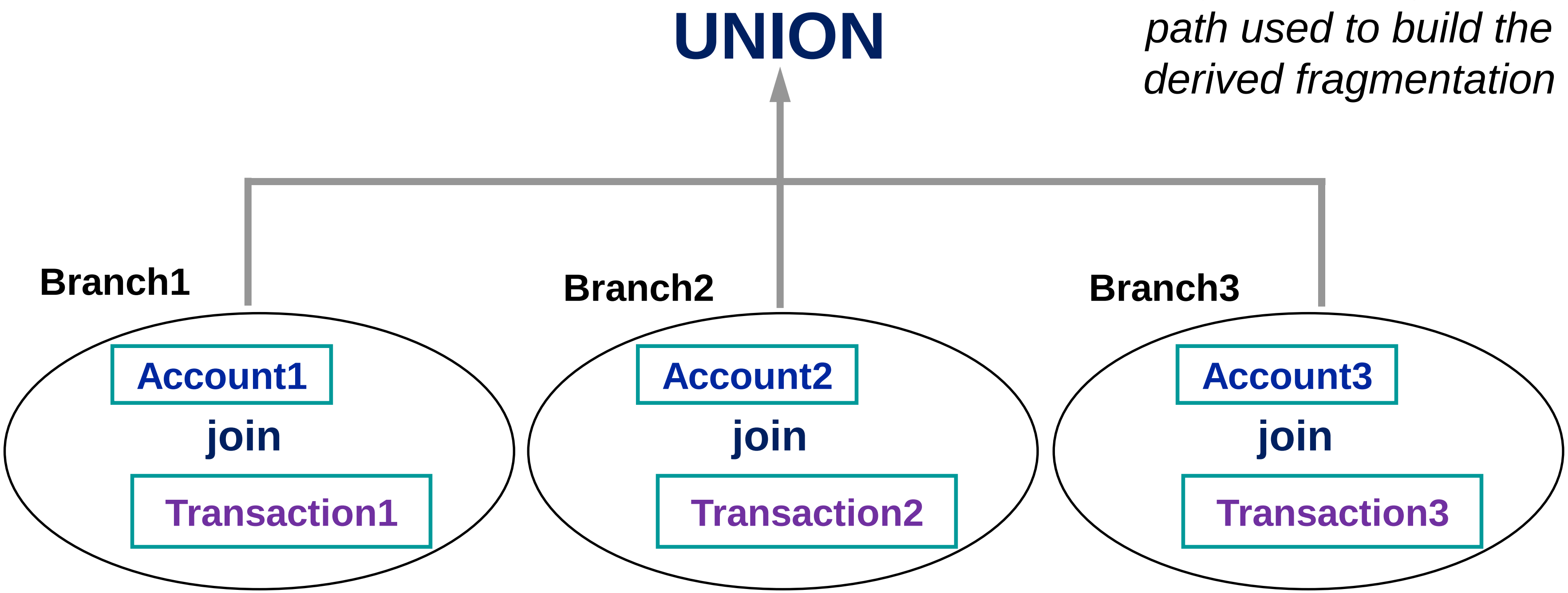
Customers owning more than one account are replicated on all branches on which they own at least one account

Distributed Join

- The most expensive operation on distributed data
- 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 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 the 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 of formulating queries, supported 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

CUSTOMER(CusSSN, Name, Addr, BD, email, tel)
ACCOUNT(Num, CusSSN, Branch, Balance)
TRANSACTION(AccNum, Date, Incr, Amount, Descr)

Transparency of Allocation

- Assumption (a):
  - The application that executes the query runs on Node 1 and knows that the account 45 was subscribed at Branch 1 (local application)

select Balance

from Account1

where Number = 45

CUSTOMER(<u>CusSSN</u>, Name, Addr, BD, email, tel)

ACCOUNT(<u>Num</u>, CusSSN, Branch, Balance)

TRANSACTION(<u>AccNum</u>, <u>Date</u>, <u>Incr</u>, Amount, Descr)

Transparency of Allocation

- Assumption (b):
  - The allocation of Account 45 is unknown, it could be located at any Branch (application running at Node 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@Branch1  where Number = 45
IF (NOT FOUND) THEN
( select Balance  from Account2@Branch2  where Number = 45
    union
  select Balance  from Account3@Branch3   where Number = 45 )
```

Transparency of Fragmentation

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

```
select Number, Incremental, Amount
from ACCOUNT join TRANSACTION on Number = AccountNumber
where Balance < 0
```

CUSTOMER(CusSSN, Name, Addr, BD, email, tel)
ACCOUNT(Num, CusSSN, Branch, Balance)
TRANSACTION(AccNum, Date, Incr, Amount, Descr)

Transparency of Allocation (distributable join)

```
select  Number, Incremental, Amount
     from Account1 join Transaction1 on Number=AccountNumber
              where Balance < 0
union
select Number, Incremental, Amount
     from Account2 join Transaction2 on …
              where Balance < 0
union
select Number, Incremental, Amount
     from Account3 join Transaction3 on …
              where Balance < 0
```

Transparency of Language

```
select  Number, Incremental, Amount
    from Account1@Branch1 join Transaction1@Branch1 on …
        where Balance < 0
union
select Number, Incremental, Amount
    from Account2@Branch2 join Transaction2@Branch2 on …
        where Balance < 0
union
select Number, Incremental, Amount
    from Account3@Branch3 join Transaction3@Branch3 on …
        where Balance < 0
```

Transparency of Fragmentation

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

CUSTOMER(<u>CusSSN</u>, Name, Addr, BD, email, tel)
ACCOUNT(<u>Num</u>, CusSSN, Branch, Balance)
TRANSACTION(<u>AccNum</u>, <u>Date</u>, <u>Incr</u>, Amount, Descr)

update ACCOUNT
set Branch = “Branch 2”
where Number = 45 *and Branch = “Branch 1”*

Transparency of Allocation

```
insert into Customer2
  select * from Customer1
  where CustomerSSN in ( select CustomerSSN
                         from Account1
                         where Number = 45 )
insert into Account2
  select * from Account1
  where Number = 45

insert into Transaction2
  select * from Transaction1
  where AccountNumber = 45

...
```

CUSTOMER(<u>CusSSN</u>, Name, Addr, BD, email, tel)
ACCOUNT(<u>Num</u>, CusSSN, Branch, Balance)
TRANSACTION(<u>AccNum</u>, <u>Date</u>, <u>Incr</u>, Amount, Descr)

Transparency of Allocation

...

delete from Transaction1
where AccountNumber = 45

delete from Account1
where Number = 45

delete from Customer1
where CustomerSSN not in
(select CustomerSSN
from Account1
where Number <> 45)

CUSTOMER(CusSSN, Name, Addr, BD, email, tel)
ACCOUNT(Num, CusSSN, Branch, Balance)
TRANSACTION(AccNum, Date, Incr, Amount, Descr)

The **order** of these operations is **critical**

- Deletions must normally follow insertions
- Integrity constraints may dictate the order of operations on the same node

```
AccountNumber...references Account(Number)
   on update cascade   on delete no action
CustomerSSN...references Customer(CustomerSSN)
   on update cascade   on delete no action
```

Transparency of Language

```
insert into Account2@Branch2
  select * from Account1@Branch1
  where Number = 45

insert into Transaction2@Branch2
  select * from Transaction1@Branch1
  where AccountNumber = 45

delete from Transaction1@Branch1
  where AccountNumber = 45

delete from Account1@Branch1
  where Number = 45
```

Distribution Design Problem

- Determining the best fragmentation and allocation of given tables
- Fragmentation should match locality characteristics, but there are trade offs
  - In a university database, with STUDENTs allocated at the central admission office and COURSEs distributed at the departments
  - How should EXAMs be fragmented?
  - Depending on the choice, **only one** of the two joins with either STUDENT or COURSE is a distributable join
    - Statistics on read and write frequency should be considered
- Allocation should give the ideal degree of redundancy
  - Redundancy speeds up retrieval and slows down updates
  - Redundancy increases availability and robustness