DATABASE
week 3 Summary

Dr Na Yao

So far

- 3.1 Normalization
- 3.2 Advanced Normalization
- 3.3 Transaction management
- 3.3 Distributed DBMS

All Very important!!!

Normalization &
Advanced Normalization

Everything is important, but
this is the **MOST** important

Functional dependency

- Important concept used for normalization.

- Functional dependency describes relationship between attributes.

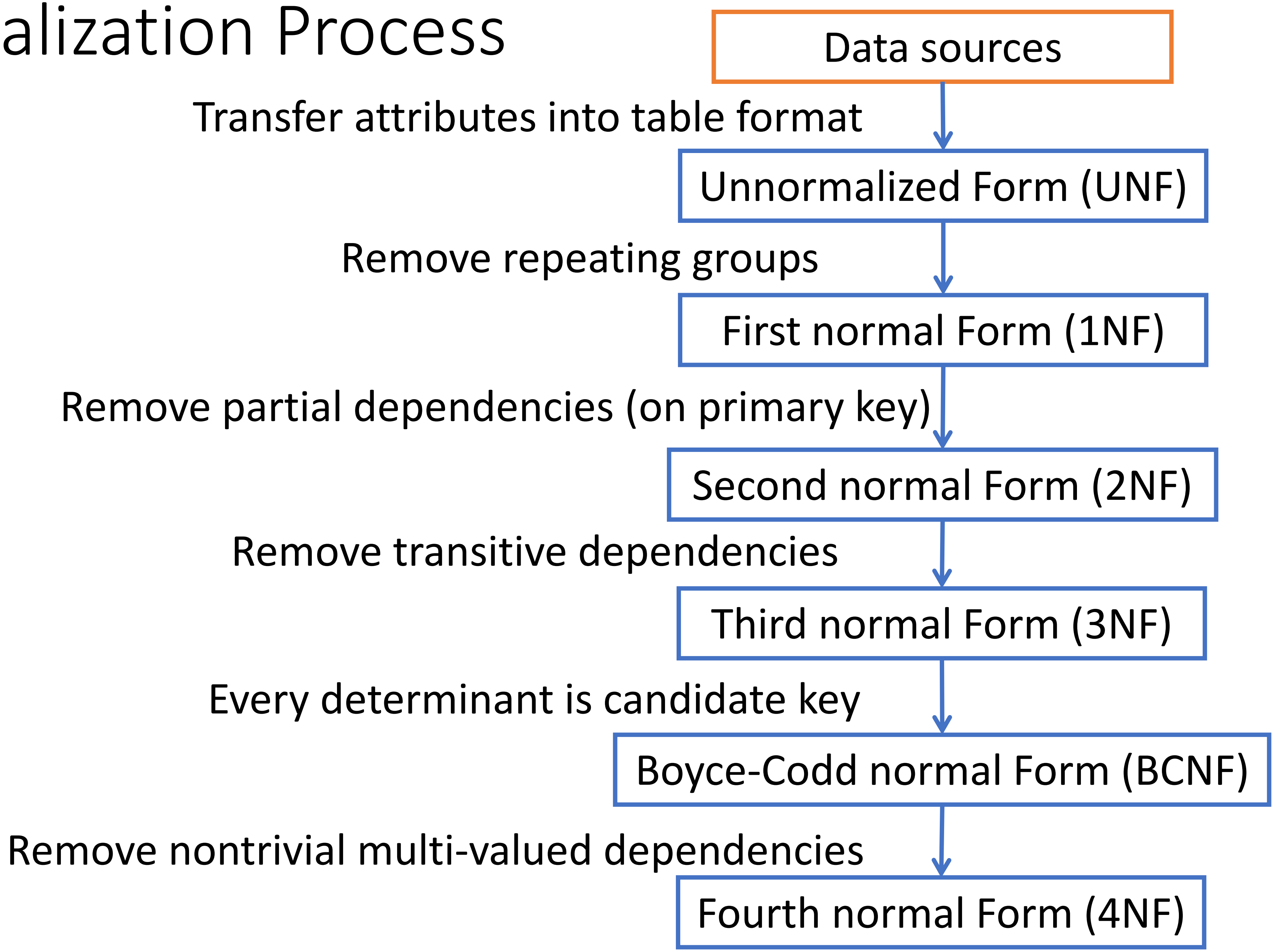
- For example, if A and B are attributes of relation R, B is functionally dependent on A (denoted $A \rightarrow B$), if each value of A in R is associated with exactly one value of B in R.

Other concepts

- Full functional dependency
- Partial functional dependency
- Transitive functional dependency

- Multi-valued dependency
- Trivial and nontrivial multi-valued dependency

Normalization Process



Transaction management

Everything is important, but this is the **MOST** important

ACID Properties of Transactions

- **A**tomicity
- **C**onsistency
- **I**solation
- **D**urability

CONCURRENCY =

MANY TRANSACTIONS ARE SIMULTANELIUSLY RUN IN THE DATABASE

Concurrency problems

- Might **violate ACID** properties!!!!
  - different transactions may read/write the same data **concurrently**
  - breaking the **isolation** property.

- **3 main problems**
  1. Lost updates problem
  2. Uncommitted updates problem
  3. Incorrect analysis problem

- Solution: **2 Phase-Locking**

Lost Update Problem

- Successfully completed update is **overridden** by another user.

Time	T1	T2	X
t_1		`begin_transaction`	100
t_2	`begin_transaction`	`read(X)`	100
t_3	`read(X)`	`X = X+100`	100
t_4	`X = X-10`	`write(X)`	200
t_5	`write(X)`	`commit`	90
t_6	`commit`		90

Solution: preventing T1 from reading X until after the update.

Uncommitted Dependency Problem

Time	T3	T4	X
t_1		`begin_transaction`	100
t_2		`read(X)`	100
t_3		`X = X+100`	100
t_4	`begin_transaction`	`write(X)`	200
t_5	`read(X)`	…	200
t_6	`X = X-10`	`rollback`	100
t_7	`write(X)`		190
t_8	`commit`		190

- **Solution:** preventing T3 from reading X until after T4 commits or aborts.

2PL

Inconsistent Analysis Problem

Inconsistent Analysis = *dirty/unrepeatable read*

Time	T5	T6	X	Y	Z	sum
t_1		`begin_transaction`	100	50	25	
t_2	`begin_transaction`	`Sum = 0`	100	50	25	0
t_3	`read(X)`	`read(X)`	100	50	25	0
t_4	`X = X-10`	`sum = sum + X`	100	50	25	100
t_5	`write(X)`	`read(Y)`	90	50	25	100
t_6	`read(Z)`	`sum = sum + Y`	90	50	25	150
t_7	`Z = Z + 10`		90	50	25	150
t_8	`write(Z)`		90	50	35	150
t_9	`commit`	`read(Z)`	90	50	35	150
t_{10}		`sum = sum + Z`	90	50	35	185
t_{11}		`commit`	90	50	35	185

Solution: preventing T6 from reading X (and Z) until after T5 completed updates.

Two-Phase Locking (2PL)

- All locking operations precede unlock operation in the transaction.
- Principle of 2PL
  - Every transaction must lock an item (read or write) before accessing it
  - Once a lock has been released, no new items can be locked
- Two phases for transaction:
  1. Growing phase - acquires all locks but cannot release any locks **= LOCKS**
  2. Shrinking phase - releases locks but cannot acquire any new locks **= UNLOCKS**

Preventing the Lost Update Problem using 2PL

Time	T1	T2	X
t_1		begin_transaction	100
t_2	begin_transaction	write_lock(X)	100
t_3	write_lock(X)	read(X)	100
t_4	WAIT	X:=X+100	100
t_5	WAIT	write(X)	200
t_6	WAIT	commit/unlock(X)	200
t_7	read(X)		200
t_8	X:=X-10		200
t_9	write(X)		190
t_10	commit/unlock(X)		190

Preventing the Uncommitted Dependency Problem using 2PL

Time	T3	T4	X
t_1		begin_transaction	100
t_3		write_lock(X)	100
t_4		read(X)	100
t_5	begin_transaction	X:=X+100	100
t_4	write_lock(X)	write(X)	200
t_5	WAIT	rollback/unlock(X)	100
t_4	read(X)		100
t_6	X:=X-10		100
t_7	write(X)		90
t_8	commit/unlock(X)		90

Time	T5	T6	X	Y	Z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	Sum = 0	100	50	25	0
t_3	write_lock(X)		100	50	25	0
t_4	read(X)	read_lock(X)	100	50	25	0
t_5	X = X-10	WAIT	100	50	25	0
t_6	write(X)	WAIT	90	50	25	0
t_7	write_lock(Z)	WAIT	90	50	25	0
t_8	read(Z)	WAIT	90	50	25	0
t_9	Z = Z + 10	WAIT	90	50	25	0
t_{10}	write(Z)	WAIT	90	50	35	0
t_{11}	Commit/unlock(X,Z)	WAIT	90	50	35	0
t_{12}		read(X)	90	50	35	0
t_{13}		sum = sum + X	90	50	35	90
t_{14}		read_lock(Y)	90	50	35	90
t_{15}		read(Y)	90	50	35	90
t_{16}		sum = sum + Y	90	50	35	140
t_{17}		read_lock(Z)	90	50	35	140
t_{18}		read(Z)	90	50	35	140
t_{19}		sum = sum + Z	90	50	35	175
t_{20}		commit	90	50	35	175

DDBMS

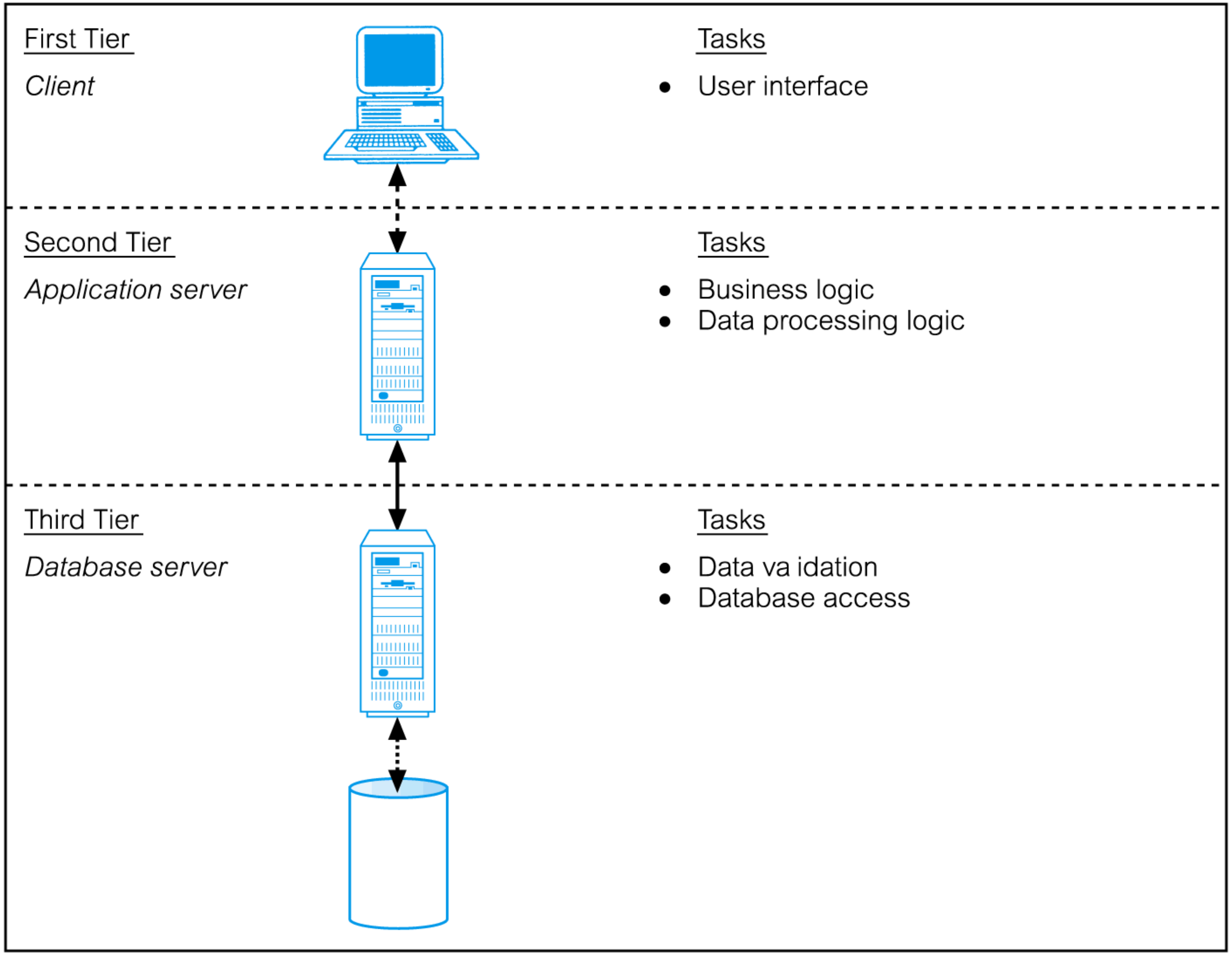
Everything is important, but
this is the **MOST** important

Client/Server Architecture

1. **The user interface layer:** runs on the end-user's computer (the *client*).
2. **The business logic and data processing layer:** this middle tier runs on a server and is often called the *application server*.
3. **A DBMS:** stores the data required by the middle tier. This tier may run on a separate server called the *database server*.

(The three-tier architecture can be extended to *multi*-tiers, with additional tiers added to provide more flexibility and scalability.)

Three-Tier Client-Server Architecture

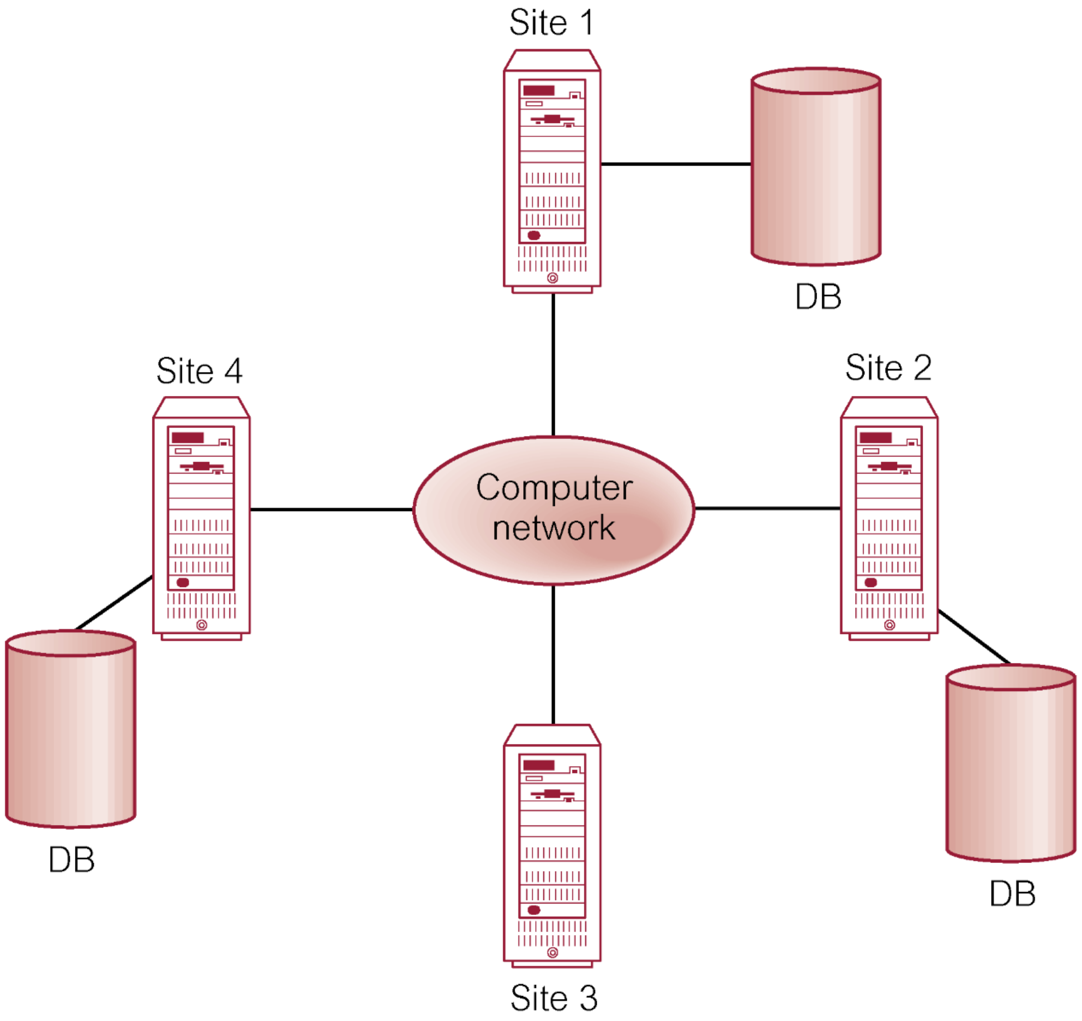


Distributed Processing != Distributed DBMS

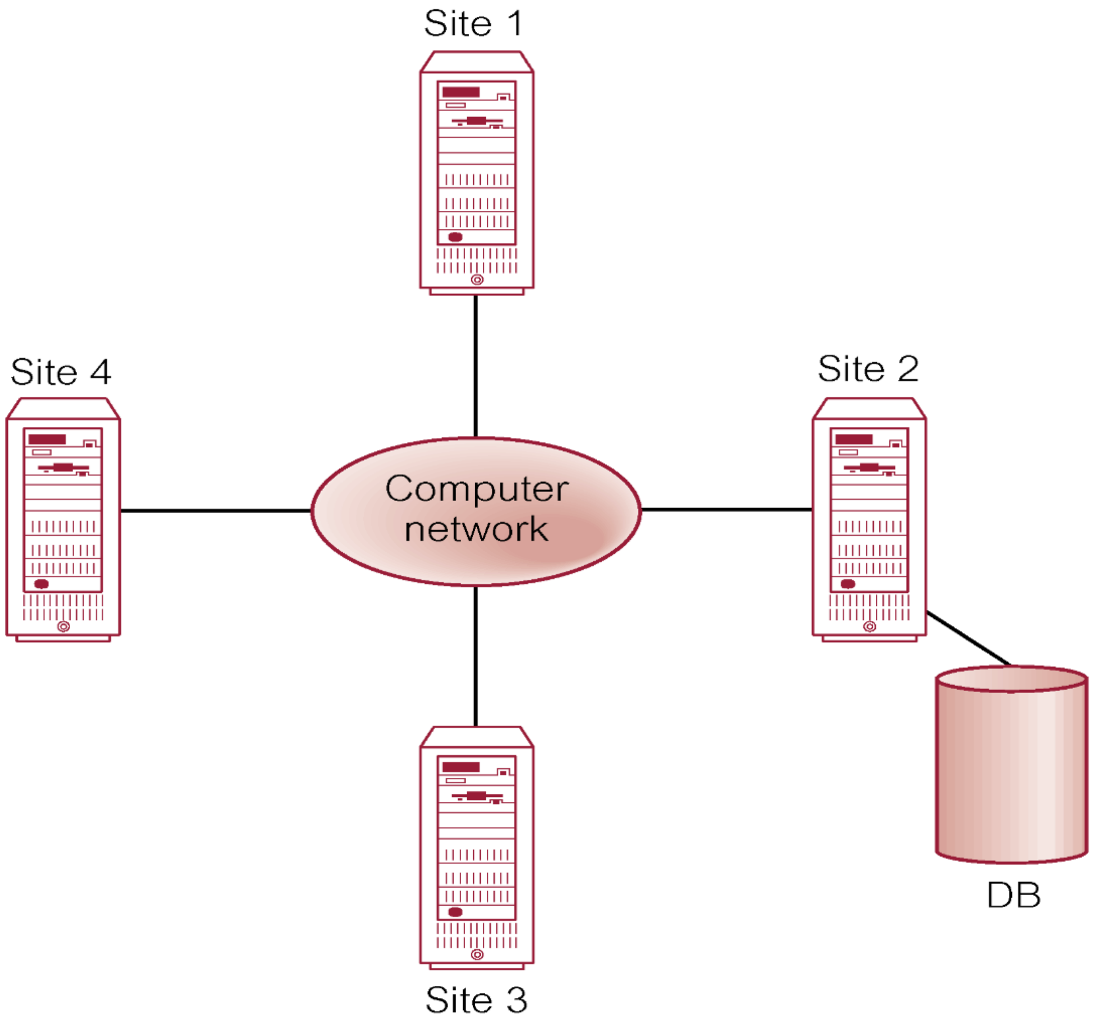
- Distributed Processing:
  - centralized database
  - that can be accessed over a computer network

$\neq$

- Distributed DBMS:
  - Single logical database that is split into fragments
  - These fragments are distributed over a computer network



DDBMS



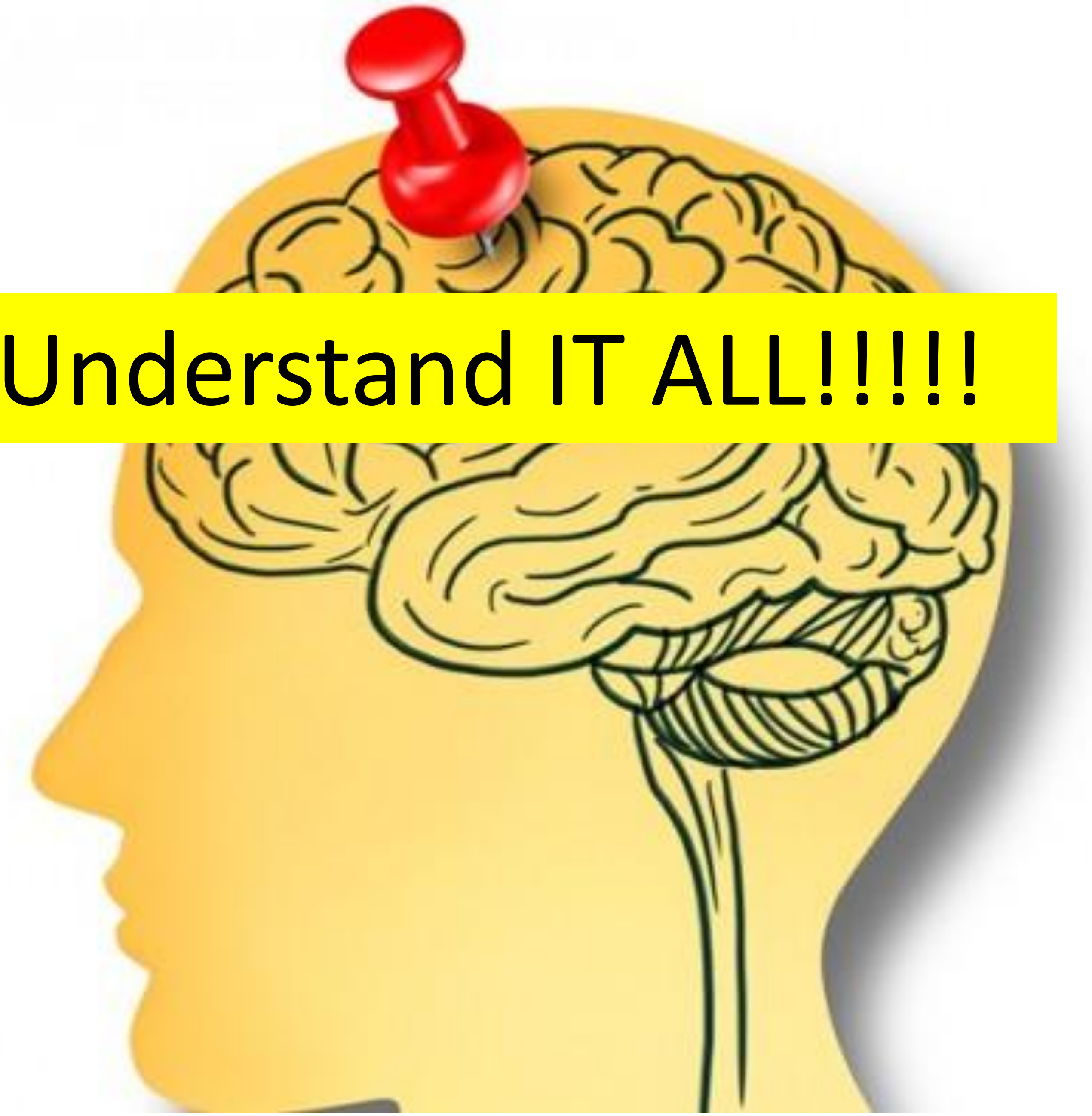
Distributed Processing

DDBMS Design Methodology

- **Normalization** to produce a design for the global relations.
- Examine **topology** of system to determine where databases will be located.
- Analyse most important transactions and identify appropriateness of horizontal/vertical **fragmentation**.
- Decide which relations are **not to be fragmented**.
- Examine **relations** on 1 side of relationships and determine a suitable fragmentation schema. Relations on many sides may be suitable for horizontal derived fragmentation.
- Check for **situations where either vertical or mixed fragmentation would be appropriate** (that is, where transactions require access to a subset of the attributes of a relation).

DDBMS vs Relational DBMS Design Methodology

- **Normalization** to produce a design for the global relations.
- Examine **topology** of system to determine where databases will be located.
- Analyse most important transactions and identify appropriateness of horizontal/vertical **fragmentation**.
- Decide which relations are **not to be fragmented**.
- Examine **relations** on 1 side of relationships and determine a suitable fragmentation schema. Relations on many sides may be suitable for horizontal derived fragmentation.
- Check for **situations where either vertical or mixed fragmentation would be appropriate** (that is, where transactions require access to a subset of the attributes of a relation).



Understand IT ALL!!!!!