

# הרצאות 3 + 4


## Big Data Technologies and Datasets

### Lecture 2 – Distributed RDBMS

Based on slides by:  
Database System Concepts, 6th Ed

### Outline

- Distributed Data Storage
- Distributed Name Allocation
- Distributed Query Processing
- Distributed Transactions and Two Phase Commit
- Concurrency Control in Distributed Databases

 מתאר מה עושים כשיש לנו מגוון רחב של משתמשים

## Distributed Data Storage

3

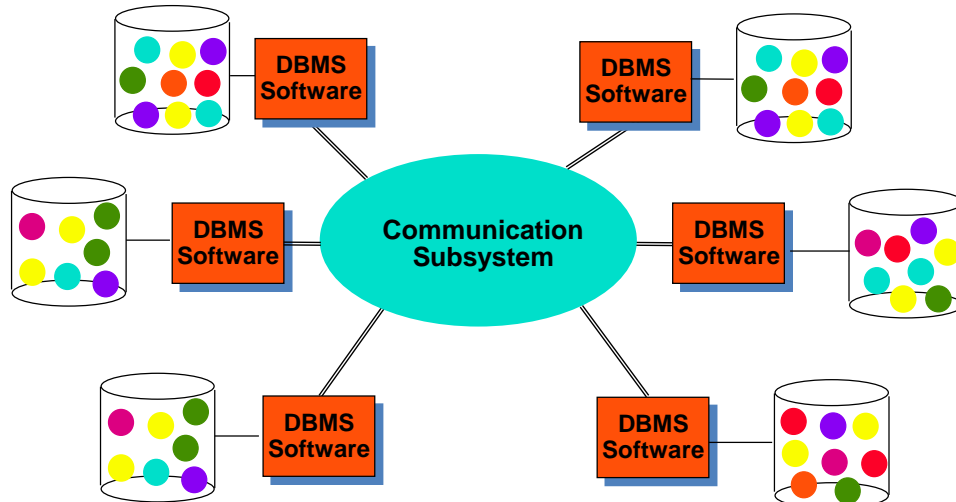
## Distributed Database System

- A distributed database system consists of loosely coupled sites that share no physical component
- Database systems that run on each site are independent of each other
- Transactions may access data at one or more sites

4

## Distributed Database

כל שרת יכול להריץ מערכת  
בסיסי נתונים שונה

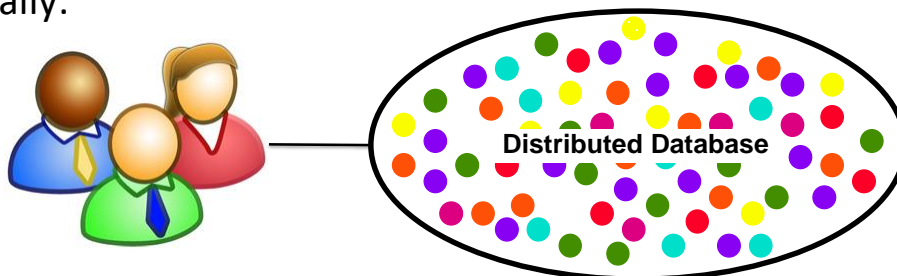


5

## Data Transparency

המשתמשים לא נחשפים לביזור, הם רואים  
מערכת אחת ועובדים אל מול המערכת הזו

- Degree to which system user may remain unaware of the details of how and where the data items are stored in a distributed system
- Ideally:



6

בהקשר הזה, אנו מבדילים בין שתי מערכות בסיסי נתונים מבוזרות

## Homogeneous vs. Heterogeneous

- In a homogeneous distributed database
  - All sites have identical software
  - Are aware of each other and agree to cooperate in processing user requests.
  - Each site surrenders part of its autonomy
  - Appears to users as a single system
- In a heterogeneous distributed database
  - Different sites may use different schemas and software
    - Difference in schema is a major problem for query processing
    - Difference in software is a major problem for transaction processing
  - Sites may not be aware of each other and may provide only limited facilities for cooperation in transaction processing
- We will focus on the homogeneous case

כל השרתים מריצים את אותה תכנת בסיסי נתונים (למשל מייאסקיול), והשרתים מכירים את הקיום אחד של השני ומשתפים פעולה כחלק ממערכת מבוזרת (משתתפים בפרוטוקולים כוללניים). מבחינת המשתמש, הוא רואה את (המערכת כמכלול) בניגוד לסוג ההטרוגני

בניגוד להומוגנית, כאן משתמשים בסכמות שונות. בנוסף אתרים או שרתים שונים לא בהכרח צריכים להכיר שיש שרתים אחרים שעובדים

7

מערכת הטרוגנית תהיה לנו למשל כשנרצה לאחד בין שני סניפים של בנקים לדוגמה, כשכל אחד מהם עובד עם מערכת בסיסי נתונים שונה. בכל מקרה, כשאנו מתכננים מאפס, נרצה להקים מערכת בסיסי נתונים הומוגנית

## Distributed Data Storage

- Consider the relational data model
- Replication
  - System maintains multiple copies, שכפול נתונים - כשיש לנו את אותה הטבלה או חלק ממנה (מועתק בכמה מקומות שונים) (מחשבים שונים)
- Fragmentation
  - Relation is partitioned into several fragments stored in distinct sites חלוקה של הנתונים, פיצול כך שכל חלק יושב על מחשב אחר
- Replication and fragmentation can be combined
  - Relation is partitioned into several fragments: system maintains several identical replicas of each such fragment. שילוב של השניים

8

## נרחיב על שלושת מושגי האחסון

### Data Replication

**יתרונות:** עמידות של הנתונים, הורדת העומס משרת בודד, מאפשר לבצע הרצות של שאילתות במקביל

- Can be done at different **granularity** levels
  - fragment, relation, database
- Advantages of Replication
  - **Availability:** failure of site containing relation  $r$  does not result in unavailability of  $r$  (if replicas exist).
  - **Parallelism:** queries on  $r$  may be processed by several nodes in parallel.
  - **Reduced data transfer:** relation  $r$  is available locally at each site containing a replica of  $r$ .
- Disadvantages of Replication
  - Increased cost of **updates** and complexity of **concurrency** control
    - concurrent updates to distinct replicas may lead to inconsistent data **unless special transaction management and concurrency control** mechanisms are implemented.

**חסרונות:** דורש יותר אחסון, כלומר יותר יקר. יותר קשה גם לנהל, עדכון מסוים צריך להתבצע בכמה מקומות שונים

### Data Fragmentation

**יתרונות:** 1. לא ניתן לשמור את כל המידע על שרת אחד בודד ולכן נרצה לפצל. ניתן להריץ תהליכים במקביל, כל שאילתא לפי סוג המידע בשרת מסוים. 2. ניתן להשיג רמה מסוימת של שרידות (אם המחשב נשרף, רק חלק מכל. 3. הולך לעזאזל

- Division of relation  $r$  into fragments  $r_1, r_2, \dots, r_n$ 
  - *Must* contain sufficient information to **reconstruct** relation  $r$ .
- **Horizontal fragmentation:**
  - Each tuple of  $r$  is assigned to one or more fragments
- **Vertical fragmentation**
  - The schema for relation  $r$  is split into several smaller schemas
    - All schemas must contain a common candidate key (or superkey) to ensure lossless join property.
    - A special attribute, the tuple-id attribute may be added to each schema to serve as a candidate key.

דוגמאות לשני סוגי חלוקות: חלוקה אופקית וחלוקה אנכית

## Horizontal Fragmentation of *account* Relation

כאן הטבלה מחולקת לשתי  
טבלאות לפי סניפים

<i>branch_name</i>	<i>account_number</i>	<i>balance</i>
Hillside	A-305	500
Hillside	A-226	336
Hillside	A-155	62

$$account_1 = \sigma_{branch\_name="Hillside"}(account) \leftarrow$$

הסיגמה מציינת באלגברה  
רלציונית סלקט, באותיות קטנות  
זה התנאי ב"זכור" ובסוגריים שם  
הטבלה עליה עושים את השאילתא

<i>branch_name</i>	<i>account_number</i>	<i>balance</i>
Valleyview	A-177	205
Valleyview	A-402	10000
Valleyview	A-408	1123
Valleyview	A-639	750

$$account_2 = \sigma_{branch\_name="Valleyview"}(account)$$

איחוד של הטבלאות מחזיר אותי  
לטבלת החשבונות המקורית. זה  
מקיים את הדרישה לשיחזור  
!הטבלה המקורית

## Vertical Fragmentation of *employee\_info* Relation

<i>branch_name</i>	<i>customer_name</i>	<i>tuple_id</i>
Hillside	Lowman	1
Hillside	Camp	2
Valleyview	Camp	3
Valleyview	Kahn	4
Hillside	Kahn	5
Valleyview	Kahn	6
Valleyview	Green	7

$$deposit_1 = \Pi_{branch\_name, customer\_name, tuple\_id}(employee\_info)$$

<i>account_number</i>	<i>balance</i>	<i>tuple_id</i>
A-305	500	1
A-226	336	2
A-177	205	3
A-402	10000	4
A-155	62	5
A-408	1123	6
A-639	750	7

$$deposit_2 = \Pi_{account\_number, balance, tuple\_id}(employee\_info)$$

חלוקה לשתי טבלאות לפי  
העמודות. שתי הטבלאות  
חולקות את אותו המפתח

כלומר, בכל תת-טבלה יש את  
כל השורות, אבל רק עמודות  
מסוימות

כדי שנוכל לשחזר את הטבלה  
המקורית (חובה לפי עיקרון  
הפרגמנטציה), יצרנו את  
העמודה שמשתמשת כמפתח  
*tuple\_id*

## Advantages of Fragmentation

- Allows parallel processing on fragments of a relation
- Allows a relation to be split so that tuples are located where they are most frequently accessed
- Some level of fault tolerance...



13

## Distributed Name Allocation

איתור של איפה המידע נמצא -  
המטא - דאטה

14

## Data Transparency

הרמה שבה המשתמש במערכת לא מודע לפרטים של איך ואיפה נשמרת כל פיסת מידע במערכת המבוזרת

- **Data transparency:** Degree to which system user may remain unaware of the details of how and where the data items are stored in a distributed system
- Consider transparency issues in relation to:
  - Fragmentation transparency
  - Replication transparency
  - Location transparency

המשתמש לא צריך לדעת:  
 1. שהמידע מחולק.  
 2. שהמידע משוכפל.  
 3. (מיקום המידע) באיזה שרת בדיוק הוא נמצא.  
 זה בדיוק הפוך מההיגיון של שקיפות בהקשר של פרטיות או בהקשר האינטואיטיבי

15

## Naming of Data Items - Criteria

קריטריונים חשובים לניימינג

1. Every data item must have a system-wide **unique name**.
2. It should be possible to **find the location** of data items efficiently.
3. It should be possible to **change the location** of data items transparently.
4. Each site should be able to **create new data items** autonomously.

**כלומר** - כל טבלה צריכה להיות בעלת שם ייחודי ביחס לכל השרתים.  
 2. בהינתן שם של טבלה, אמורה להיות לנו היכולת לדעת איפה היא מאוחסנת.  
 3. אמורה להיות לנו היכולת לשנות את המיקום של טבלה, למשל משרת אחד לאחר.  
 4. כל שרת אמור להיות מסוגל ליצור טבלה חדשה באופן עצמאי.

16



## Centralized Name Server

- Structure:
  - name server assigns all names
  - each site maintains a record of local data items
  - sites ask name server to locate names for non-local data items
- Advantages:
  - satisfies naming criteria 1-3
- Disadvantages:
  - does not satisfy naming criterion 4
    - name server is a potential performance bottleneck
    - name server is a single point of failure

**שרת השמות יחיד** - שרת שבו יש לי אינדקס של כל פריט מידע (טבלה) ובאיזה שרת הוא מאוחסן. כלומר השרת הזה הוא זה שיוצר את השמות ומחזיק מיפוי של שמות של טבלאות לשרתים בהם הם נמצאים. כששרת רוצה ליצור שם חדש הוא פונה לשרת הזה. החסרונות הם שהוא הופך להיות סוג של צוואר בקבוק בתהליך של קביעת שמות חדשים (במידה והוא עמוס) וגם מבטל את נקודה 4 מהשקף הקודם

17

## Use of Aliases

אלטרנטיבה לפיתרון המוצע בשקף הקודם לקיום הקריטריונים. האלטרנטיבה הזו היא מה שמשתמשים בו בפועל, והיא למעשה עונה על ארבעת הקריטריונים

- A possible alternative to the centralized scheme:
  - each site prefixes its own site identifier to any name that it generates i.e., *site 17.account*.
    - Fulfills having a unique identifier, and avoids problems associated with central control.
    - However, what if the data item moves to a different location?
- Solution
  - Create a set of aliases for data items
  - Store the mapping of aliases to the real names at each site.
- The user can be unaware of the physical location of a data item, and is unaffected if the data item is moved from one site to another.

18

הפיתרון: שמירת טבלת מיפויים נוספת בכל שרת, שמתעד את הטבלאות שהעברתי מהשרת הספציפי לשרתים אחרים. כלומר אם העברתי את משרת אחד bdl1.accounts טבלת אני אשמור בטבלת המיפוי הזו תיעוד לכך שהוא עבר ולאן הוא עבר (במקרה הזה שרת 2

יכול ליצור בעיות. עכשיו השם שכולל בתוכו תחילית של שרת אחד, נמצא בשרת אחר בכלל והתחילית הופכת ללא רלוונטית. גם אם אני אשנה את התחילית לשם של השרת החדש, יכול להיות שכבר קיימת לי טבלה כזאת: למשל `bdl2.accounts` כבר קיימת, ואני רוצה `bdl1.accounts` להעביר לשרת שתיים את הטבלה

## Distributed Query Processing

19

## Distributed Query Processing

- For centralized systems, the primary criterion for measuring the cost of a particular strategy is the number of disk accesses.
- The optimizer is responsible for...
- In a distributed system, other issues must be taken into account:
  - The cost of data transmission over the network.
  - The potential gain in performance from having several sites process parts of the query in parallel.
- Some processing and optimization examples for the distributed case follow...

20

## Query Transformation

- Translating algebraic query on a relation to an equivalent query over its fragments.
  - It must be possible to construct relation  $r$  from its fragments
  - Replace relation  $r$  by the expression to construct relation  $r$  from its fragments

- Consider the horizontal fragmentation of the *account* relation into

$$account_1 = \sigma_{branch\_name = "Hillside"}(account)$$

$$account_2 = \sigma_{branch\_name = "Valleyview"}(account)$$

- The query  $\sigma_{branch\_name = "Hillside"}(account)$  becomes:

$$\sigma_{branch\_name = "Hillside"}(account_1 \cup account_2)$$

- Which can be rewritten as:

$$\sigma_{branch\_name = "Hillside"}(account_1) \cup \sigma_{branch\_name = "Hillside"}(account_2)$$

השאיטת הזאת יכולה להתבצע במקביל

21

## Simple Query Optimization

$$account_1 = \sigma_{branch\_name = "Hillside"}(account)$$

$$account_2 = \sigma_{branch\_name = "Valleyview"}(account)$$

$$Query = \sigma_{branch\_name = "Hillside"}(account_1) \cup \sigma_{branch\_name = "Hillside"}(account_2)$$

- Since  $account_1$  has only tuples pertaining to the Hillside branch, we can eliminate the selection operation.
- We can also apply the definition of  $account_2$  to obtain  $\sigma_{branch\_name = "Hillside"}(\sigma_{branch\_name = "Valleyview"}(account))$
- This expression is the empty set regardless of the contents of the *account* relation.
- Finally, return  $account_1$  as the result of the query.

22

## Simple Join Processing

- Consider the following relational algebra expression in which the three relations are neither replicated nor fragmented:

$account \bowtie depositor \bowtie branch$

- $account$  is stored at site  $S_1$
- $depositor$  at  $S_2$
- $branch$  at  $S_3$
- For a query issued at site  $S_i$ , the system needs to produce the result at site  $S_i$

כל טבלה נמצאת בשרת אחר

Employee		
Name	Empld	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Sales

Dept	
DeptName	Manager
Finance	George
Sales	Harriet
Production	Charles

Employee $\bowtie$ Dept			
Name	Empld	DeptName	Manager
Harry	3415	Finance	George
Sally	2241	Sales	Harriet
George	3401	Finance	George
Harriet	2202	Sales	Harriet

את התוצאה בסוף נרצה בשרת בודד כלשהו. יכול להיות שרצה להעביר את כל הטבלאות לשרת שבו נמצאת הטבלה הכי גדולה, או השרת החזק ביותר

23

## Possible Processing Strategies

- Ship copies of all three relations to site  $S_i$  and choose a strategy for processing the entire query locally at site  $S_i$ .
- Ship a copy of the  $account$  relation from site  $S_1$  to site  $S_2$  and compute  $temp_1 = account \bowtie depositor$  at  $S_2$ .

Ship  $temp_1$  from  $S_2$  to  $S_3$ , and compute  $temp_2 = temp_1 \bowtie branch$  at  $S_3$ .

Ship the result  $temp_2$  to  $S_i$ .

- Devise similar strategies, exchanging the roles  $S_1, S_2, S_3$
- Must consider following factors:
  - amount of data being shipped
  - cost of transmitting a data block between sites
  - relative processing speed at each site

העברת כל הטבלאות לשרת אחד. כעת אפשר לעבוד רק עם אינדקס אחד, שקיים בשרת הבודד הזה

העברת הטבלאות באופן סיריאלי (אחד אחד). כך למעשה אפשר לשמור על כל האינדקסים של הטבלאות

24

## Semijoin Strategy

- Let  $r_1$  be a relation with schema  $R_1$  stored at site  $S_1$   
Let  $r_2$  be a relation with schema  $R_2$  stored at site  $S_2$
- Evaluate the expression  $r_1 \bowtie r_2$  and obtain the result at  $S_1$ .
  - Compute  $temp_1 \leftarrow \Pi_{R_1 \cap R_2}(r_1)$  at  $S_1$ .
  - Ship  $temp_1$  from  $S_1$  to  $S_2$ .
  - Compute  $temp_2 \leftarrow r_2 \bowtie temp_1$  at  $S_2$
  - Ship  $temp_2$  from  $S_2$  to  $S_1$ .
  - Compute  $r_1 \bowtie temp_2$  at  $S_1$ . This is the same as  $r_1 \bowtie r_2$ .

#פאי הוא כמו סלקט של אסקיואל

25

## Formal Definition

- The **semijoin** of  $r_1$  with  $r_2$ , is denoted by:

$$r_1 \ltimes r_2 = \Pi_{R_1}(r_1 \bowtie r_2)$$

- Thus,  $r_1 \ltimes r_2$  selects those tuples of  $r_1$  that contribute to  $r_1 \bowtie r_2$ .

Employee		
Name	EmpId	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Production

Dept	
DeptName	Manager
Sales	Bob
Sales	Thomas
Production	Katie
Production	Mark


Employee $\bowtie$ Dept		
Name	EmpId	DeptName
Sally	2241	Sales
Harriet	2202	Production

- In step 3 above,  $temp_2 = r_2 \ltimes r_1$ .
- For joins of several relations, the above strategy can be extended to a series of semijoin steps.

26

## Join Strategies that Exploit Parallelism

עד כה לא דיברנו על מקבול של כל  
העסק של השאלות

- 
- Consider  $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$  where:
    - relation  $r_i$  is stored at site  $S_i$
    - the result must be presented at site  $S_1$ .
  - $r_1$  is shipped to  $S_2$  and  $r_1 \bowtie r_2$  is computed at  $S_2$ : simultaneously  $r_3$  is shipped to  $S_4$  and  $r_3 \bowtie r_4$  is computed at  $S_4$
  - $S_2$  ships tuples of  $(r_1 \bowtie r_2)$  to  $S_1$  as they are being produced  
 $S_4$  ships tuples of  $(r_3 \bowtie r_4)$  to  $S_1$  as they are being produced
  - Once tuples of  $(r_1 \bowtie r_2)$  and  $(r_3 \bowtie r_4)$  arrive at  $S_1$   
 $(r_1 \bowtie r_2) \bowtie (r_3 \bowtie r_4)$  can be computed in parallel (s.f. pipeline join)  
i.e. with the computation of  $(r_1 \bowtie r_2)$  at  $S_2$  and the computation of  $(r_3 \bowtie r_4)$  at  $S_4$ .

27

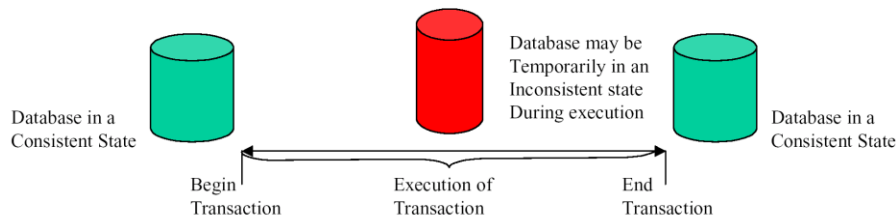
## Distributed Transactions and Two Phase Commit

עיבוד טרנזקציות מבזורות

28

## Transaction

- A transaction is a sequence of operations performed as a single logical unit of work that transform the system from one state to another while preserving system consistency.



מצב לוגי לא תקין לדוגמה: אם העברתי כסף מחשבון אחד לחשבון אחר, כאשר מהחשבון הנותן בוצעה הפחתה ביתרה אבל בחשבון המקבל עוד לא עלתה היתרה. כלומר הטרגזקציה לא נגמרה

29

## Example

- Consider an airline reservation example with the relations:
  - FLIGHT(FNO, DATE, SRC, DEST, STSOLD, CAP)
  - CUST(CNAME, ADDR, BAL)
  - FC(FNO, DATE, CNAME, SPECIAL)
- INSERT INTO CUST (CNAME,ADD,BAL) VALUES ("Mr. Cohen","20 Ben-Gurion St. Tel-Aviv",1250)
- SELECT STSOLD FROM FLIGHT WHERE FNO="001" and DATE=1/1/2003
- INSERT INTO FC(FNO,DATE,CNAME) VALUES ("001",1/1/2003,"Mr. Cohen")
- UPDATE FLIGHT SET STSOLD=STSOLD+1 WHERE FNO="001" and DATE=1/1/2003

30

## ACID

תכונות מאוד חשובות  
לבסיס נתונים רלציוני

- Four properties required (the ACID properties):

- Atomicity:

או שכל הפעולות קורות ביחד, או שאף פעולה לא קורת

– all actions in the transaction happen, or none happen

- Consistency:

עקביות עם חוקי הדאטה בייס

– a transaction preserves the consistency of the DB

- Isolation:

ביצוע הטרנזקציה הוא בלתי תלוי בטרנזקציות אחרות, בעיקר עבור משתמשים שרצים במקביל ומתחרים על משאבי המערכת (כמו מקום בטיסה)ץ

– execution of one transaction is isolated from that of other transactions

- Durability:

– if a transaction commits, its effects persist

ברגע שטרנזקציה הסתיימה, היא נשארת לנצח לא משנה מה יקרה

Concurrency Control - (also appears on slide 23)

A problem in which two transactions are applied simultaneously. Hence, the system will show incorrect information (e.g. two users applied step 4 in the exact same time. therefore, the system will show that only 1 sit was taken instead of 2). That's exactly the same problem that we talked about considering the "Isolation" principle.

## How?

Ways to mitigate the Concurrency control issue

- Locks
- Commit
- Rollback
- Log...

עוזר להתמודד עם הבידוד - כדי - Locks  
למנוע מצב שמתואר בדוגמה של הזמנת מקומות בטיסה. כל טרנזקציה תנעל את המשאבים בהם היא משתמשת

הטרנזקציה - commit  
הסתיימה והיא סופית

Rollback - the ability to reverse all actions to the start of the transaction

מנגנון שמראה את רצף ביצוע - Log  
הפעולות, אמור לעזור לנו בשחזור דברים שקרו. הלוג עצמו כתוב בדיסק ולא בזיכרון ((כדי שישמר



## Distributed Transactions

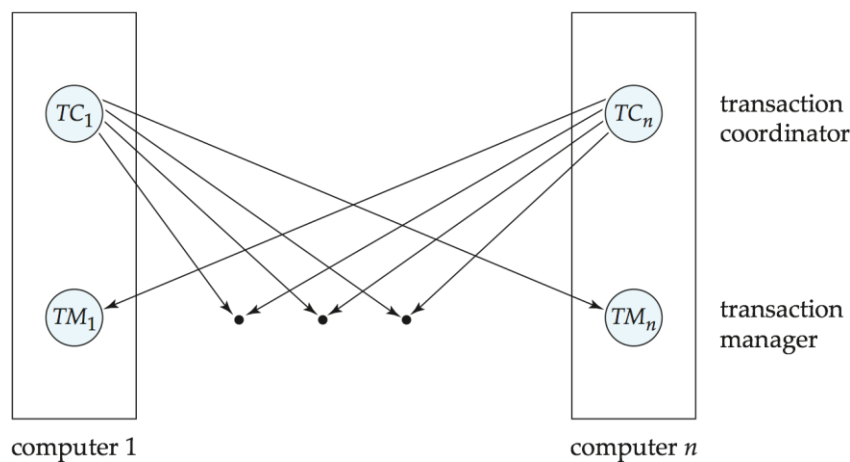
ביצוע טרנזקציות מבוזרות, במידה ובסיס  
הנתונים שלנו מבוזר

- Transactions may access data at several sites.
- Each site has a local **transaction manager** responsible for:
  - Maintaining a log for recovery purposes
  - Participating in coordinating the concurrent execution of the transactions executing at that site.
- Each site has a **transaction coordinator**, which is responsible for:
  - Starting the execution of transactions that originate at the site.
  - Distributing subtransactions at appropriate sites for execution.
  - Coordinating the termination of each transaction that originates at the site
    - which may result in the transaction being committed at all sites or aborted at all sites.

במצב שאחד מהמנגירים נכשל, הקואורדינאטור אומר  
(לכולם להיכשל) כי הטרנזקציה נכשלה

33

## Transaction System Architecture



34

## Commit Protocols

- Commit protocols are used to ensure **atomicity** across sites
  - a transaction which executes at multiple sites must either be committed at all sites, or aborted at all sites.
  - not acceptable to have a transaction committed at one site and aborted at another
- The *two-phase commit* (2PC) protocol is widely used
- The *three-phase commit* (3PC) protocol is more complicated and more expensive, but avoids some drawbacks of the two-phase commit protocol. This protocol is not used in practice.
- Other alternative exist too – e.g., persistent messages.

35

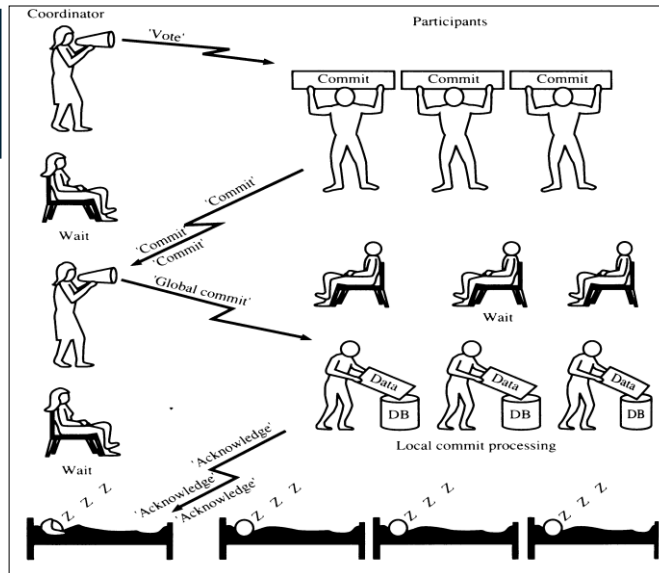
## Two Phase Commit Protocol (2PC)

- Assumes a **fail-stop** model
  - failed sites simply stop working, and do not cause any other harm, such as sending incorrect messages to other sites.
- Execution of the protocol is initiated by the coordinator after the last step of the transaction has been reached.
- The protocol involves all the local sites at which the transaction executed
- Let  $T$  be a transaction initiated at site  $S_j$ , and let the transaction coordinator at  $S_i$  be  $C_i$

36

## TWO-PHASE COMMIT - OK

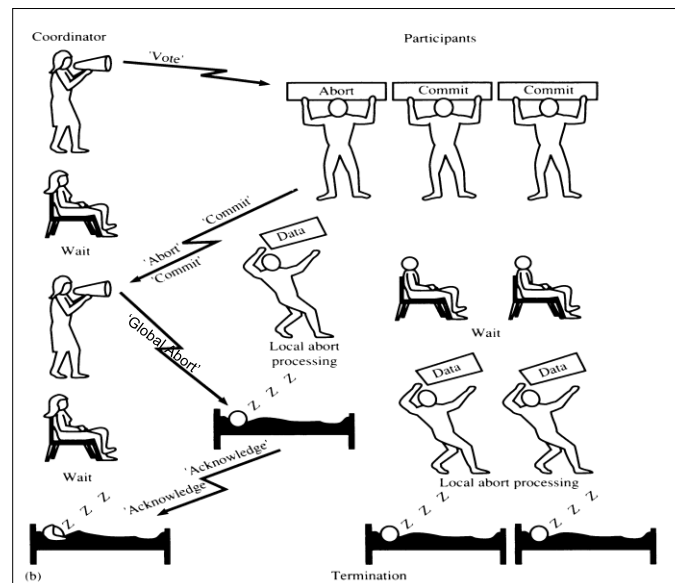
הקואורדינאטור שבווד הוא  
הקואורדינאטור של המחשב  
שיזם את הטרנזקציה  
(הקואורדינאטורים של שאר  
המחשבים לא עובדים)



37

## TWO-PHASE COMMIT - ABORT

במצב הזה, בגלל שאחד מהם  
נכשל, כולם נכשלו והמצב חוזר  
לקדמותו



38

## נצלול קצת ...יותר לפרטים

### Phase 1: Obtaining a Decision

מוסיף את הרשומה הזו ללוג,  $C_i$  הקואורדינאטור ושולח אותה לכל השרתים שבהם הטרנזקציה מתבצעת. בכל שרת, הטרנזקשיון מנג'ר מחליט אן הוא יכול לבצע את הטרנזקציה או לא, ומגיב בהתאם.

- Coordinator asks all participants to *prepare* to commit transaction  $T_i$ .
  - $C_i$  adds the records [**prepare  $T$** ] to the log
  - sends <**prepare  $T$** > messages to all sites at which  $T$  executed
- Upon receiving message, transaction manager at site determines if it can commit the transaction
  - if not:
    - add a record [**no  $T$** ] to the log
    - (records are not forced to stable storage)
    - send <**abort  $T$** > message to  $C_i$
  - if the transaction can be committed, then:
    - add the record [**ready  $T$** ] to the log
    - force *all records* for  $T$  to stable storage
    - send <**ready  $T$** > message to  $C_i$

39

### Phase 2: Recording the Decision

כמו שאמרנו, הטרנזקציה יכולה להתבצע אם הקואורדינאטור קיבל הודעה חיובית מכל השרתים. בכל מקרה, הוא מתעד את ההחלטה בלוג. באותו הרגע, ההחלטה היא בלתי-הפיכה

- $T$  can be committed if  $C_i$  received a <**ready  $T$** > message from all the participating sites: otherwise  $T$  must be aborted.
- Coordinator adds a decision record, [**commit  $T$** ] or [**abort  $T$** ], to the log. Once decision was sent to the log it is irrevocable (even if failures occur)
- Coordinator sends a message to each participant informing it of the decision (commit or abort)
- Participants take appropriate action locally
  - undo( $T$ ) if aborted
  - add [**commit  $T$** ] or [**abort  $T$** ] to the log

הקואורדינאטור מידע את השרתים בהחלטה הסופית, ולפי ההחלטה הזו כל שרת מבצע קומיט או נוטש בהתאם.

## System Failure Modes

- Failures unique to distributed systems:
  - Failure of a **site**.
  - Loss of **messages**
    - Handled by network transmission control protocols such as TCP-IP
  - Failure of a **communication link**
    - Handled by network protocols, by routing messages via alternative links
  - Network partition**
    - A network is said to be **partitioned** when it has been split into two or more subsystems that lack any connection between them
    - Note: a subsystem may consist of a single node
- Network partitioning and site failures are generally indistinguishable.

41

כעת נפרט על כל אחד מהבולטים

## Handling of Failures - Site Failure

מה שקורה זה שהשרת בוחן את הלוג שלו מיד לאחר שהוא "חוזר לתחייה", ובודק מה קרה עד רגע הקריסה. כך הוא יודע כיצד להגיב מיד לאחר ההתאוששות

When site  $S_i$  recovers, it examines its log to determine the fate of transactions active at the time of the failure.

- Log contain [**commit**  $T$ ] or [**abort**  $T$ ]: transaction completed, nothing to be done בגלל שאם אחד מהשניים האלה כתוב, סימן שהטרנזקציה הסתיימה
- Log contains **<no  $T$ >**: transaction was undone, nothing to be done
- <ready  $T$ >** record: site must consult  $C_i$  to determine the fate of  $T$ .
  - If  $T$  committed, write **<commit  $T$ >** record
  - If  $T$  aborted,  $\text{undo}(T)$ , write **<abort  $T$ >** record
- The log contains no log records concerning  $T$ :
  - Implies that  $S_k$  failed before responding to **<prepare  $T$ >**
  - since the failure of  $S_k$  precludes the sending of such a response, therefore the coordinator must have aborted  $T$
  - nothing to be done

42

ועכשיו בכיוון ההפוך - מה קורה עם הקואורדינאטור נכשל

## Handling of Failures- Coordinator Failure

בתכלס, אם זה קורה, השרתים מתייעצים אחד עם השני

- If coordinator fails while the commit protocol for T is executing then participating sites must decide on T's fate:
  - If an active site contains a [commit T] record in its log, then T must be committed.
  - If an active site contains an [abort T] record in its log, then T must be aborted.
  - If some active participating site does not contain a [ready T] record in its log, then the failed coordinator  $C_i$  cannot have decided to commit T.
    - Can therefore abort T;
    - Moreover, such a site must reject any subsequent <prepare T> message from  $C_i$
  - If none of the above cases holds, then all active sites must have a [ready T] record in their logs, but no additional control records (such as [abort T] or [commit T]).
    - In this case active sites must wait for  $C_i$  to recover, to find decision.
- Blocking problem: active sites may have to wait for failed coordinator to recover.

הבולט השלישי הוא בעצם מקרה שבו אין קומיט או אבורט בשרת מסוים

43

## Handling of Failures - Network Partition

אם הקואורדינאטור נמצא באותה מחיצה עם השרתים, לנפילה אין אפקט כלל

- If the coordinator and all its participants remain in one partition, the failure has no effect on the commit protocol.
- If the coordinator and its participants belong to several partitions:
  - Sites that are not in the partition containing the coordinator think the coordinator has failed, and execute the protocol to deal with failure of the coordinator.
    - No harm results, but sites may still have to wait for decision from coordinator.
  - The coordinator and the sites that are in the same partition think that the sites in the other partition have failed, and follow the usual commit protocol.
    - Again, no harm results

אין פה בעיה באמת, אבל יכול להיות שהשרתים יצטרכו לחכות עד שיחודש הקשר עם הקואורדינאטור

השרתים שנמצאים באותה המחיצה חושבים ששאר המחשבים נפלו. לכן, הם פועלים לפי אותו פרוטוקול ממקודם

44

## Recovery and Concurrency Control

במצב שהטרנזקציה היא לא קומיט ולא אבורט, אנחנו מחכים שהקואורדינאטור יקום לתחייה - אנחנו קצת תקועים. לכן, לא נרצה שהמצב יתקע את כל בסיס הנתונים, אלא שרק מה שקשור לטרנזקציה יתקע. כלומר - רק הטבלאות שקשורות לטרנזקציה ינעלו, והשאר יהיו זמינות

- **In-doubt** transactions have a [ready T], but neither a [commit T], nor an [abort T] log record.
- The recovering site must determine the commit-abort status of such transactions by contacting other sites; this can slow and potentially block recovery.
- Recovery algorithms can note lock information in the log.
  - Instead of [ready T], write out [ready T, L] L = list of locks held by T when the log is written (read locks can be omitted).
  - For every in-doubt transaction T, all the locks noted in the [ready T, L] log record are reacquired.
- After lock reacquisition, transaction processing can resume; the commit or rollback of in-doubt transactions is performed concurrently with the execution of new transactions.

45

## Concurrency Control

בקרה של ריבוי משתמשים / ריבוי הרצות (קשור לדוגמת הזמנת כרטיס הטיסה שלמעלה)

46

## Concurrency Control

Concurrency Control = mechanism of locks operation to support distributed DBs

- Modify concurrency control schemes for use in distributed environment.
- We assume that each site participates in the execution of a commit protocol to ensure global transaction atomicity.
- We assume all replicas of any item are updated

אנחנו מניחים שברקע יש פרוטוקול תמיכה  
בטרנזקציות כמו קומיט בשני שלבים ושכשאנחנו רוצים  
לעדכן טבלה, אנחנו מעדכנים את כל העותקים שלה.

47

## Single-Lock-Manager Approach

- System maintains a *single* lock manager that resides in a *single* chosen site, say  $S_i$
- When a transaction needs to lock a data item, it sends a lock request to  $S_i$  and lock manager determines whether the lock can be granted immediately
  - If yes, lock manager sends a message to the site which initiated the request
  - If no, request is delayed until it can be granted, at which time a message is sent to the initiating site

48



## Single-Lock-Manager Approach (Cont.)

- The transaction can read the data item from *any* one of the sites at which a replica of the data item resides.
- Writes must be performed on all replicas of a data item
- Advantages of scheme:
  - Simple implementation
  - Simple deadlock handling
- Disadvantages of scheme are:
  - Bottleneck: lock manager site becomes a bottleneck
  - Vulnerability: system is vulnerable to lock manager site failure.

49

## Distributed Lock Manager

- In this approach, functionality of locking is implemented by lock managers at each site
  - Lock managers control access to local data items
    - But special protocols may be used for replicas
- Advantage:
  - work is distributed and can be made robust to failures
- Disadvantage:
  - deadlock detection is more complicated
  - lock managers have to cooperate for deadlock detection
- Several variants of this approach
  - Primary copy
  - Majority protocol
  - Biased protocol

50

## Primary Copy

לכל טבלה (במקרה שלנו) אני בוחר עותק שיהיה העותק הראשי. כשטרנזקציה צריכה לנעול טבלה מסוימת, היא מבקשת לנעול אותה כך אני גם נועל את כל העותקים של אותה טבלה (זה קורה בגלל שאם מישהו ירצה לגשת לעותק שהוא אינו העותק הראשי של הטבלה, הוא יצטרך לנעול את העותק הראשי - אבל העותק הראשי כבר נעול).

- Choose one replica of data item to be the primary copy.
  - Site containing the replica is called the primary site for that data item
  - Different data items can have different primary sites
- When a transaction needs to lock a data item  $Q$ , it requests a lock at the primary site of  $Q$ .
  - Implicitly gets lock on all replicas of the data item
- Benefit
  - Concurrency control for replicated data handled similarly to unreplicated data - simple implementation.
- Drawback
  - If the primary site of  $Q$  fails,  $Q$  is inaccessible even though other sites containing a replica may be accessible.

51

## Majority Protocol

- Local lock manager at each site administers lock and unlock requests for data items stored at that site.
- When a transaction wishes to lock an unreplicated data item  $Q$  residing at site  $S_i$ , a message is sent to  $S_i$ 's lock manager.
  - If  $Q$  is locked in an incompatible mode, then the request is delayed until it can be granted.
  - When the lock request can be granted, the lock manager sends a message back to the initiator indicating that the lock request has been granted.

52

## Majority Protocol (Cont.)

- In case of replicated data
  - If  $Q$  is replicated at  $n$  sites, then a lock request message must be sent to more than half of the  $n$  sites in which  $Q$  is stored.
  - The transaction does not operate on  $Q$  until it has obtained a lock on a majority of the replicas of  $Q$ .
  - When writing the data item, transaction performs writes on *all* replicas.
- Benefit
  - Can be used even when some sites are unavailable
    - Omit details on how to handle writes in the presence of site failure...
- Drawback
  - Communication cost.
  - Potential for deadlock even with single item
    - e.g., each of 3 transactions may have locks on 1/3rd of the replicas of a data.

אם טבלה קיו משוכפלת, נניח ב-10 שרתים, אז בקשה לנעילה חייבת להישלח ל-6 שרתים או יותר בהם הטבלה מאוחסנת. הטרנזקציה לא מקבלת גישה לטבלה עד שהיא לא מקבלת נעילה מרוב העותקים (6 ומעלה). אם הטרנזקציה מעדכנת טבלה, היא מעדכנת גם את כל העותקים שלה. זה פותר לנו את הבעיה של הפתרון הקודם בכך שגם אם שרת אחד נופל, אנחנו עדיין יכולים לגשת למידע שאנחנו צריכים

53

## Biased Protocol

- Local lock manager at each site as in majority protocol, however, requests for shared locks are handled differently than requests for exclusive locks.
- **Shared locks.** When a transaction needs to read a data item  $Q$ , it simply requests a lock on  $Q$  from the lock manager at one site containing a replica of  $Q$ .
- **Exclusive locks.** When a transaction needs to write a data item  $Q$ , it requests a lock on  $Q$  from the lock manager at all sites containing a replica of  $Q$ .
- Advantage
  - imposes less overhead on **read** operations.
- Disadvantage
  - additional overhead on writes

נעילה בצורה אחרת של פעולות קריאה ושל פעולות כתיבה

כשאנחנו רוצים לקרוא פריט מידע מסוים, אנחנו רוכשים נעילה על העותק הספציפי אותו אנחנו קוראים. אם נרצה לכתוב, נצטרך לרכוש נעילה על \*כל\* העותקים

קריאה אנחנו יכולים לעשות בצורה מאוד מהירה ומקבילית. עבור כתיבה, אנחנו לא יכולים לעשות משהו אחר חוץ מהפעולה הזו, וגם היא יקרה יחסית

54

## Replication with Weak Consistency

- Many commercial databases support replication of data with weak degrees of consistency (i.e., without a guarantee of serializability)
- E.g.: **master-slave replication**: updates are performed at a single “master” site, and propagated to “slave” sites.
  - Propagation is not part of the update transaction: it is decoupled
    - May be immediately after transaction commits
    - May be periodic
  - Data may only be read at slave sites, not updated
    - No need to obtain locks at any remote site
  - Particularly useful for distributing information
    - E.g. from central office to branch-office
  - Also useful for running read-only queries offline from the main database

55

## Replication with Weak Consistency (Cont.)

- Replicas should see a transaction-consistent snapshot of the database
  - That is, a state of the database reflecting all effects of all transactions up to some point in the serialization order, and no effects of any later transactions.
- E.g. Oracle provides a create snapshot statement to create a snapshot of a relation or a set of relations at a remote site
  - snapshot refresh either by recomputation or by incremental update
  - Automatic refresh (continuous or periodic) or manual refresh

56