Database Final

Edited 5
23/05/2017

By AeAe CS 30

Contents

unit 8 : Normalization 1

Stage of Normalization 1

1 Normal Form 2

2 Normal Form 3

3 Normal Form 3

Boyce-Codd Normal Form 4

4 Normal Form

Denormalization

unit 14 : Transaction 5

ACID Properties 5

Transaction State 5

Schedules 6

Serial Schedule 6

Concurrent Schedule 6

Serializability 6

Conflicting Instruction 6

Conflicting Serializability 6

View Serializability 6

Recoverable Schedules 7

Cascading Rollbacks 7

Cascadeless Schedules 7

Concurrency Control 7

unit 15 : Concurrency Control 8

Lock-Base Protocols 8

Deadlock 8

Starvation 8

Two-Phase Locking Protocols 9

Automatic Acquisition of Locks 9

Deadlock Prevention protocols 9

Multiple Granularity 10

Timestamp-Base Protocols 10

Recoverability and Cascade Freedom 11
Thomas' Write Rule 11
Chapter 16 : Recovery System 12
Failure Classification 12
Storage Structure 12
Data Access 12
Log-Base Recovery 13
Undo and Redo Operations 13
Undo and Redo on Recovering from Failure (non-checkpoint) 14
Checkpoint 14
Recovery from Failure 14
Chapter 10 : Storage 15
Classification of Physical Storage Media 15
Physical Storage Media 15
Storage Hierarchy 15
Magnetic disk structure 16
Performance Measures of Disk 16
Chapter 12 : Query Processing 17
Measures of Query Cost 17
Selection Operation 17

คำเตือน 1 นี้ไม่ใช่สรุปที่ดี เพราะกูเขียนเรื่อยเปื่อยมาก 5555555

คำเตือน 2 เนื้อหาที่เขียนเกิดจากความเข้าใจตอนนั่งเรียน และอ่านจากชีทหรือในเน็ตเพิ่มเติม ถ้าอ่านตรงไหนแล้วคิดว่า เอ๊ะ นี่มันมั่วนี่หว่า ทักมาบอกด้วย (มีหลักฐานด้วยว่าที่ถูกเป็นไงก็จะดีมาก กูจะได้ไม่ต้องไปเช็คเพิ่ม)

คำเตือน 3 ไม่เข้าใจถามได้ ถ้าว่างจะตอบ ถ้าไม่ตอบคือไม่ว่าง 555555

Release Note

Edited 1 15/04/2017

- เพิ่ม chapter 8

Edited 2 22/04/2017

- เพิ่ม chapter 14

Edited 3 30/04/2017

- เพิ่ม chapter 15
- เพิ่มคำเตือน 2 3

Edited 4 22/05/2017

- เพิ่ม chapter 16
- เพิ่ม chapter 10 (ยังไม่เสร็จ)
- เปลี่ยนนาย A เป็นหนูษาแล้วนะ เบลอไปหน่อย

Edited 5 23/05/2017

- chapter 10 จบแค่นั้นเหละ อ่าน RAID ไม่รู้เรื่อง 55555
- เพิ่ม chapter 12
- แก้ chapter 8
    - transitive dependencies
    - BCNF
- ทำสารบัญให้ด้วยนะเออ

Chapter 8 Normalization

Normalization คืออะไร?

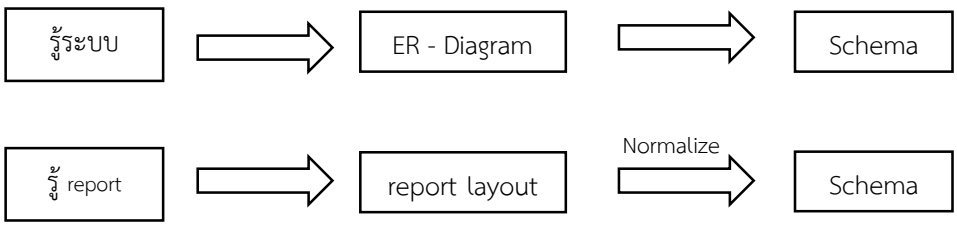
- normalization คือการปรับโครงสร้าง table ที่มีอยู่ใหม่ เพื่อให้ถูกต้องและลดการซ้ำซ้อน (redundancies) ของข้อมูล

- เป้าหมายสุดท้ายของ normalize คือการได้ schema ที่ไม่มี redundancies

- ให้เห็นภาพมากขึ้น จะเล่านิทานให้ฟัง

กาลครั้งหนึ่งนานมาแล้ว หนูษา รับจ้างเขียน db งานแรก บริษัท ABC ได้มาว่าจ้าง โดยให้หนูษา เข้าไปศึกษาระบบงานต่างๆของบริษัท เมื่อหนูษา ได้เข้าไปศึกษาจนเข้าใจโครงสร้างบริษัทแล้วจึงได้ร่าง ER – diagram ของบริษัทขึ้นมา จากนั้นจึงนำไปแปลงเป็น schema เพื่อทำ db ต่อไป

หลังจากนั้นบริษัท XYZ ได้มาจ้างหนูษา แต่คราวนี้ไม่ยอมให้หนูษา เข้าไปศึกษาบริษัท แต่ได้โยนตารางข้อมูลมาให้ แล้วบอกว่า “อยากให้โชว์ข้อมูลออกมาแบบเนี่ย ทำไงก็ได้ ขอแบบดีๆ ราคาถูกๆ” ไอ่ตารางที่ว่านี้ขอเรียกว่า report layout ล่ะกัน แต่ว่า table นี้มี redundant เต็มไปหมด ซึ่งในแง่การออกแบบแล้วมันโคตรแย่ จึงต้องทำให้ redundant พวกนี้หายไป วิธีนี้เรียกว่า normalization ขอบคุณที่อุส่าอ่านมาตั้งนาน ดูรูปข้างล่างก็เข้าใจล่ะ 55555



Stages of Normalization

normalize ก็มีระดับนะเออ เรียงตามนี้ 1NF, 2NF, 3NF, BCNF, 4NF (NF = Normal Form) แน่นอนว่า 4NF ดีสุด และ 1NF กากสุด แต่โดยปกติแล้ว ทำถึง 3NF ก็ถือว่าดีมากแล้ว

Let’s start !!

report layout เอาจากชีท ch.5 ของ Rob หน้า 7 นะ จะมี attribute ดังนี้

(PROJ_NUM, PROJ_NAME, EMP_NUM, EMP_NAME, JOB_CLASS, CHG_HOUR, HOURS)

อธิบายเพิ่มเติมอีกนิด ตารางนี้เป็นตารางแสดง project ที่มีอยู่ในบริษัท แล้วก็แสดงด้วยว่าใครเป็นคนทำ proj นี้ แล้วก็ใช้เวลาทำไปแล้วกี่ชม.

- PROJ_NUM หมายเลข project
- PROJ_NAME ชื่อ project
- EMP_NUM หมายพนักงาน
- EMP_NAME ชื่อพนักงาน
- JOB_CLASS ตำแหน่ง
- CHG_HOUR ค่าจ้างต่อ ชม. (มั้ง)
- HOURS เวลาที่แต่ละคนใช้ไปกับ project นี้

1 Normal Form

ตารางที่เป็น 1 NF ต้องมีคุณสมบัติดังนี้ 1. ไม่มี repeating group

2. ระบุ PK แล้ว

step 1 : ตามหา repeating group

- หา column ที่มีการซ้ำกันอย่างมีความหมาย (make sense เอาล่ะกัน)

จากตัวอย่างจะเห็นว่า PROJ_NUM, PROJ_NAME ซ้ำกันรัวๆ

* ข้อมูลที่ไม่เกิดการซ้ำนั้นจะเป็น multivalued ของส่วนที่ซ้ำ

จะได้แบบนี้ (PROJ_NUM, PROJ_NAME, {EMP_NUM, EMP_NAME, JOB_CLASS, CHG_HOUR, HOURS})

ถ้าหาได้แบบนี้ก็จะตีความได้ว่า 1 project จะมีคนทำหลายคน

หรืออีกแบบนึง ถ้าสมมติมองว่า 1 คน ทำหลาย project ล่ะ

ก็จะได้เป็นแบบนี้ (EMP_NUM, EMP_NAME, {PROJ_NUM, PROJ_NAME, HOURS, JOB_CLASS, CHG_HOUR})

* ที่เอา HOURS เข้าไปเป็น multivalued ด้วยก็เพราะว่า จำนวนชม.มันไม่ขึ้นกับคน

* ที่เอา JOB_CLASS, CHG_HOUR เข้าไปด้วยก็เพราะ สมมตินาย John เป็น developer อยู่ใน project A แต่อาจไปเป็น manager ใน project B ก็ได้ เพราะงั้นมันก็ไม่ควรฟิกไว้กับคน แล้วไอ่ CHG ก็เลือกไปขึ้นกับ job class ไง

บ่นมาตั้งนานเพื่อที่จะบอกว่า ดูได้หลายแบบ แล้วแต่จะมอง แต่ผลลัพธ์สุดท้ายต้องเหมือนกัน (อันนี้คิดว่านะ ยังไม่ได้ลองเช็คว่าเหมือนจริงมั๊ย)

step 2 : ระบุ PK

- หา PK ของ attribute ตัวนอก และ multivalued แล้วใช้เป็น PK ทั้งหมด

จากตัวอย่างจะได้ PROJ_NUM และ EMP_NUM ดังนั้นจะได้

(PROJ_NUM, EMP_NUM, PROJ_NAME, EMP_NAME, JOB_CLASS, CHG_HOUR, HOURS)

*** ถ้าไม่มี PK ทำไงดี?** ร้องชิบหายสิครับ ร้องเสร็จแล้วตั้งสติดีๆ ถ้ามันไม่มีก็สร้างให้มันก้สิ้นเรื่อง สมมติว่าตารางนี้เก็บแต่ชื่อพนักงาน แน่นอนว่าชื่อพนักงานมันซ้ำได้แน่ๆ งั้นก็สร้าง attribute EMP_ID ขึ้นมาใหม่ซะเลย แค่นี้ก็มี PK ล่ะ

step 3 : ตามหา functional dependencies

ก่อนจะตามหาต้องรู้จักกับมันก่อน functional dependencies คืออะไร?

$\{A_1\} \rightarrow \{A_2\}$ หมายความว่า ค่าของ $\{A_2\}$ ขึ้นอยู่กับค่าของ $\{A_1\}$ เสมอ

* ที่ให้ { } เพราะว่าไม่จำเป็นต้องเป็น Attribute ตัวเดียว อาจเป็นหลายๆตัวก็ได้

เช่น JOB_CLASS → CHG_HOUR ถ้า JOB_CLASS เป็น Database Designer CHG ก็จะเป็น 105.00 เสมอ

* เวลาหา functional dependencies ต้องเช็คดีๆ ถ้ามีกรณีนึงที่ขัดแย้ง จะไม่เป็นทันที

ex. ทุกช่องที่เป็น DB Designer CHG เป็น 105.00 หมดเลย แต่เสือกมีอยู่แถวนึงเป็น 100.00 ก็จะปฏิเสธ JOB_CLASS → CHG_HOUR ทันที

* ระวังไว้ว่าบทกลับไม่เป็นจริงเสมอไป หมายความว่า $A_2 \rightarrow A_1$ อาจไม่เป็นจริงก็ได้ เช่น มี JOB_CLASS คู่หนึ่งที่มีค่า CHG เท่ากัน

functional dependencies มีอยู่ 2 แบบ (ป่ะว่ะ)

- partial dependencies คือ $\{A_1\}$ เป็นส่วนหนึ่งของ PK

    * ย้ำว่าแค่ส่วนหนึ่ง ถ้า $\{A_1\}$ เป็น PK จะไม่ถือว่าเป็น partial dependencies

- transitive dependencies คือ $\{A_1\}$ และ $\{A_2\}$ ไม่ได้เป็นส่วนหนึ่งของ PK เลย

เกริ่นมานาน จากตัวอย่างก็จะได้

partial dependencies

PROJ_NUM -> PROJ_NAME

EMP_NUM -> EMP_NAME, JOB_CLASS, CHG_HOUR

transitive dependencies

JOB_CLASS -> CHG_HOUR

2 Normal Form

คุณสมบัติของ 2NF คือ 1. ไม่มี partial dependencies

step 1 แยก PK ที่มี partial dependencies ออกมาเป็นตารางใหม่

table1 : (PROJ_NUM, EMP_NUM, PROJ_NAME, EMP_NAME, JOB_CLASS, CHG_HOUR, HOURS)

table 2 : (PROJ_NUM)

table 3 : (EMP_NUM)

step 2 ย้าย attribute ที่มี dependencies กับ PK แต่ละตัวมาใส่ตารางใหม่

table1 : (PROJ_NUM, EMP_NUM, HOURS)

table 2 : (PROJ_NUM, PROJ_NAME)

table 3 : (EMP_NUM, EMP_NAME, JOB_CLASS, CHG_HOUR)

3 Normal Form

คุณสมบัติของ 3NF คือ 1. ไม่มี transitive dependencies

* จากตัวอย่าง table1, table2 เป็น 3NF แล้วเพราะไม่มี transitive dependencies

* แต่ table3 ยังมี JOB_CLASS -> CHG_HOUR อยู่

step 1 แยก $\{A_1\}$ ของ transitive ออกมาเป็นตารางใหม่ และให้เป็น PK

table 3 : (EMP_NUM, EMP_NAME, JOB_CLASS, CHG_HOUR)

table 4 : (JOB_CLASS)

step 2 ย้าย $\{A_2\}$ ของ transitive ไปยังตารางใหม่

table 3 : (EMP_NUM, EMP_NAME, JOB_CLASS)

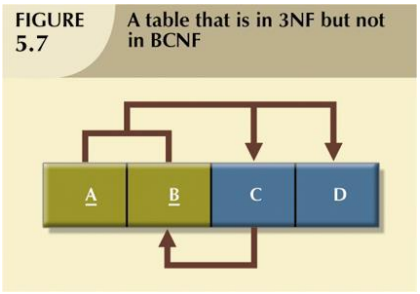
table 4 : (JOB_CLASS, CHG_HOUR)

Boyce-Codd Normal Form (BCNF)

คุณสมบัติของ BCNF 1. PK ไม่เป็น ~~transitive~~ functional dependencies ของ attribute อื่น

- เป็น NF พิเศษที่คั่นระหว่าง 3NF กับ

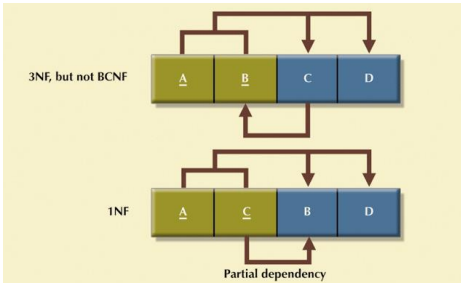
- จะทำก็ต่อเมื่อ บางส่วนของ PK ขึ้นกับ attribute ตัวอื่น (PK เป็น $\{A_2\}$)



อ่ะ ก๊อปรูปมาให้ดู จะเห็นว่า C -> B และ B เป็น PK เห็นม่ะ

step 1 สลับ $\{A_1\}$ มาเป็น PK จะทำให้ table กลายเป็น 1NF

step 2 ไล่ทำให้ 1NF ที่ได้ใหม่กลายเป็น 3NF



4 Normal Form

Denormalization

- การ join ตารางที่แยกออกมาให้กลับเป็นเหมือนเดิม

Chapter 14 : Transactions

transaction คือ program ที่เข้าถึง(access) และแก้ไข(update) ข้อมูล

ACID Properties

คือคุณสมบัติที่ database ต้องมีเพื่อรักษาไว้ซึ่งความถูกต้องของข้อมูลจากการทำงานของ transaction

1. Atomicity

สามารถทำ transaction จนเสร็จทั้งหมดได้ หรือถ้าไม่สามารถทำให้สำเร็จได้ก็ต้องย้อนกลับไปก่อนที่จะเริ่มทำได้ (roll back)

2. Consistency

- หลังจากจบ transaction ข้อมูลต้องมีความถูกต้อง
- ของที่ใช้เพื่อตรวจสอบความถูกต้องจะมีอยู่ 2 อย่าง
    - Explicitly specified integrity constraint คือพวก PK, FK
    - Implicit integrity constraint คือเงื่อนไขบางอย่าที่ยืนยันความถูกต้องของข้อมูลหลังจากทำ transaction เช่น
        - transaction ในการโอนเงินจาก A ไป B sum(A,B) ก่อนทำกับหลังทำต้องเท่ากัน
- ระหว่างทำงานอนุญาติให้ข้อมูลผิดพลาดได้ (temporarily inconsistent) แต่หลังจากทำงานเสร็จแล้วต้องถูกต้องเท่านั้น

3. Isolation

- การทำงานของ transaction ต้องเป็นแบบ serially
- แต่ทำงานแบบ concurrently เร็วกว่านะ งั้นก็อนุญาติให้ทำแบบ concurrently แต่ต้องทำทีล่ะ transaction (แล้วมันเป็น concurrently ตรงไหนว่ะ)

* serially คือการทำงานแบบ อนุกรม ทำทีล่ะงาน เสร็จงานนึงก่อนค่อยทำงานใหม่

* concurrently คือการทำงานแบบ ขนาน ทำงานหลายๆอย่างพร้อมกัน

4. Durability

- การแก้ไขข้อมูลใน database จะต้องได้รับการบันทึกลง hardware แน่นอน แม้ว่าระบบจะล่มก็ตาม

Transaction State

- Active	transaction ที่กำลังทำงานอยู่ จะอยู่ใน state นี้
- Partially Committed	transaction ที่กำลังทำงานคำสั่งสุดท้าย
- Failed	transaction ไม่สามารถทำงานต่อได้
- Aborted	หลังจากทำการ roll back แล้ว ซึ่งจะไปต่อได้ 2 แบบ
- restart	ทำ transaction นั้นใหม่ (ทำได้เมื่อไม่มี logical error อยู่ภายใน transaction)
- kill	เลิกทำ transaction นั้น
- Committed	transaction เสร็จสมบูรณ์

Schedules

- คือ ลำดับของ คำสั่ง(instructions) ของ transaction ที่จะทำงาน
- schedule จะต้องประกอบไปด้วย instruction ทั้งหมดของ transaction
- schedule จะต้องรักษาลำดับของ instruction ภายใน transaction
    * แยก instruction ได้ แต่ต้องไม่สลับลำดับภายใน transaction เดียวกัน
- transaction ที่ทำงานสำเร็จ จะต้องมีคำสั่ง commit ไว้ที่คำสั่งสุดท้าย
- transaction ที่ทำงานไม่สำเร็จ จะต้องมีคำสั่ง aborted ไว้ที่คำสั่งสุดท้าย

Serial Schedule

- schedule ที่ทำงานแบบ serial
- schedule แบบนี้จะมีผลการทำงานที่ถูกต้องแน่นอน (data consistency)

Concurrent Schedule

- schedule ที่มีการทำงานแบบ concurrent
- schedule แบบนี้อาจทำให้ข้อมูลผิดพลาด แต่ก็มีบาง schedule ที่มีผลการทำงานถูกต้องได้

Serializability

- schedule ใดๆ จะมีสมบัติ serializabilty ก็ต่อเมื่อ schedule นั้นมีผลการทำงานเหมือน serial schedule (equivalent to serial schedule)
- schedule ที่มีสมบัตินี้ จะมี data consistency แน่นอน

Conflicting Instruction

กำหนดให้ I_a และ I_b เป็น instruction จาก T_a และ T_b ตามลำดับ

I_a และ I_b จะเป็น non-conflict instruction (คำสั่งที่ไม่ขัดแย้งกัน) ก็ต่อเมื่อ

1. I_a และ I_b เข้าถึงคนละ file

หรือ 2. I_a และ I_b เข้าถึง file เดียวกัน แต่ต้องเป็น read ทั้งคู่

Conflict Serializability

- ถ้ามี schema S อยู่ และสามารถแปลงเป็น S’ ด้วยการสลับ non-conflict instruction ได้ จะเรียก S และ S’ ว่า *conflict equivalent*
- ถ้า S’ เป็น serial schedule แล้ว จะกล่าวได้ว่า S มีสมบัติ *conflict serializable*

View Serializability

- schema S และ S’ จะเป็น view equivalent ก็ต่อเมื่อ
    1. ถ้า T_i เป็นผู้ read(Q) คนแรกใน S ก็ต้องเป็นผู้ read(Q) คนแรกใน S’ ด้วย
    2. ถ้า T_i read(Q) อ่านข้อมูลที่ write(Q) โดย T_j ใน S’ ก็ต้องเป็นแบบนั้นในทุกๆการ read
    3. ถ้า T_i เป็นผู้ write(Q) คนสุดท้ายใน S ก็ต้องเป็นผู้ write(Q) คนสุดท้ายใน S’ ด้วย

    * พิจารณากับทุกๆ file หาก schedule นั้นทำงานมากกว่า 1 file

- ถ้า S’ เป็น serial schedule แล้ว S จะมีสมบัติ *view serializable*

- ถ้า S มีสมบัติ conflict serializable แล้ว S จะมี view serializable แน่นอน

- แต่ว่าถึงแม้ S จะมี view serializable แต่ก็อาจไม่เป็น conflict serializable

Recoverable Schedules

- S จะมีสมบัติ recoverable schedule เมื่อ

ถ้า T_b read ข้อมูลที่ write โดย T_a

แล้ว T_a ต้อง commit ก่อน T_b

Cascading Rollbacks

- เหตุการณ์ที่ transaction หนึ่ง failure แล้วทำให้ transaction อื่นต้อง roll back ด้วย

Cascadeless Schedules

- S จะต้องไม่เกิด cascading rollback ในทุกๆคู่ของ transaction

- schedule ที่เป็น cascadeless จะเป็น recoverable ด้วย

Concurrency Control

- database จะต้องมีกลไกจัดการให้ schedules มีสมบัติดังนี้

1. เป็น conflict หรือ view serializable
2. เป็น recoverable และ cascadeless

- การจัดให้มีแค่ transaction เดียวทำงานในแต่ละครั้ง จะเป็นการทำ serial schedule แต่ก็จะก่อให้เกิด poor degree of concurrency (concurrency อย่างกาก)

* poor degree of concurrency แสดงถึงว่าทำงานได้ช้าด้วย

Chapter 15 : Concurrency Control

เกริ่นก่อนเริ่ม

- บทนี้เรียนเรื่องอะไร? ถ้ายังจำได้ schedule ที่ทำงานแบบ concurrency นั้นเร็วกว่า serial แต่บางครั้งก็ทำงานผิดพลาดได้ การที่จะควบคุมให้ concurrency ไม่เกิดข้อผิดพลาดคือประเด็นของบทนี้

Lock-Base Protocols

- เป็น Protocols ที่ว่าด้วย transaction ต้อง lock data item ทุกครั้งที่มีการเข้าถึง และ unlock เมื่อไม่ใช้แล้ว

- Data item สามารถถูก lock โดย 2 mode ดังนี้

1. exclusive mode (X-lock)
   - item ที่ถูก lock ด้วยโหมดนี้ จะไม่ยอมให้ transaction อื่นเข้ามาใช้งาน ไม่ว่า transaction นั้นจะ request lock ใดก็ตาม
   - transaction จะ request X-lock ได้ก็ต่อเมื่อไม่มี lock ใดๆอยู่บน item นั้น
   - request ด้วยคำสั่ง **lock-X(D)** ส่วนมากจะใช้เมื่อต้องการ **write(D)**
2. shared mode (S-lock)
   - item ที่ถูก lock ด้วยโหมดนี้ จะยอมให้ transaction อื่นเข้ามาทำงานได้
   - transaction จะ request S-lock ได้ก็ต่อเมื่อไม่มี x-lock อยู่บน item นั้น
   - request ด้วยคำสั่ง **lock-S(D)** ส่วนมากจะใช้เมื่อต้องการ **read(D)**

- เมื่อ transaction สั่ง lock-X หรือ lock-S request นี้จะส่งไปยัง **concurrency-control manager** ซึ่งเป็นระบบที่คอยควบคุมการ lock

- transaction จะทำงานต่อได้ ก็ต่อเมื่อ request ที่ส่งไปได้รับอนุญาตแล้ว (granted)

* อย่างไรก็ดี Lock-Base Protocols ก็ยังไม่รับประกันว่าจะเกิด serializability

Deadlock (ภาวะ ฉันก็รักของฉัน เข้าใจบ้างไหม)

สถานการณ์สมมติ

T_1 ครอบครองข้อมูล A ไว้ และกำลัง request B

T_2 ครอบครองข้อมูล B ไว้ และกำลัง request A

ทั้งสองต่างก็ต้องการข้อมูลของอีกฝ่าย

วิธีแก้ไข

บังคับ T_1 หรือ T_2 ให้ roll back เพื่อปลดปล่อยข้อมูลที่ถือครองไว้

* ในชีวิตจริงอาจเกิด dead lock มากกว่า 2 ตัว อาจรอกันต่อไปเรื่อยๆ เป็นวงกลมก็ได้

Starvation (ภาวะหิวกระหาย)

สถานการณ์สมมติ

T_1 S-lock ข้อมูล A ไว้อยู่

T_2 ต้องการ X-lock ข้อมูล A เลยนั่งรอ

T_3 มาขอ lock-S(A) ได้เพราะ A ติดสถานะ S-lock อยู่เฉยๆ

T_4 มาขอ lock-S(A) อีกเหมือนกัน แต่ในตอนนี้ T_1ทำงานเสร็จไปแล้ว เหลือแต่ T_3

T_5 T_6 T_7 มาขอ lock-S(A) อีก

ตัดภาพกลับมาที่ T_2 นั่งรอแล้วรออีก manager ก็ไม่ยอมหันมา (น่าสงสารนะครับ)

- ปัญหา starvation นั้นเกิดจากการออกแบบ concurrency-control manager มาไม่ดี

Two-Phase Locking Protocols

- เมื่อกี้บอกไปแล้วว่า locking protocols ไม่รับประกัน serializability ก็เลยนำมันมาอัพเกรดให้ดีขึ้น
- two-phase locking รับประกัน **conflict-serializability**
- เป็น protocols ที่ว่าด้วยการแบ่ง transaction ออกเป็น 2 phase

1. Growing Phase
   - โค้ดใน phase นี้ จะร้องขอ lock เท่าไหร่ก็ได้ แต่จะ unlock ไม่ได้
   - สามารถอัพจาก lock-S ไปเป็น lock-X ได้ (upgrade)
2. Shrinking Phase
   - โค้ดใน phase นี้ จะ unlock เท่าไหร่ก็ได้ แต่จะร้องขอ lock ไม่ได้
   - สามารถลดจาก lock-X เป็น lock-S ได้ (downgrade)

- วิธีดูว่า transaction ไหนไม่เป็น two-phase locking ไม่ยาก

  หา unlock ตัวแรก แล้วหลังจาก unlock ตัวแรก ต้องไม่มี lock อยู่อีกเลย (downgrade ได้)

- วิธีทำให้ locking ธรรมดากลายเป็น two-phase locking แบบง่ายที่สุด

  ย้ายคำสั่ง lock ทั้งหมดไปไว้ต้นโปรแกรม และ unlock ทั้งหมดไปไว้ท้ายโปรแกรม

* two-phase locking ยังมีโอกาสเกิด deadlock อยู่

Automatic Acquisition of Locks

- เรื่องไรว่ะ
- การ **read(D)** จะต้องเช็คก่อนว่า T_i มี lock ของ D แล้วหรือยัง ถ้ามีแล้วก็อ่านได้ แต่ถ้ายังไม่มีก็ต้องรอจนกว่า D จะไม่มี X-lock แล้วค่อย lock-S(D) จากนั้นจึงจะทำการ read ได้
- การ **write(D)** จะต้องเช็คก่อนว่า T_i มี X-lock ของ D แล้วหรือยัง ถ้ามีแล้วก็เขียนได้ แต่ถ้ายังต้องรอจนกว่า D จะไม่มี lock ใดๆอยู่เลย แล้วค่อย request ของ X-lock หรืออัพเกรดจาก S-lock ไปเป็น X-lock แล้วจึงทำการ write

* lock จะได้รับการ unlock โดยอัตโนมัติ ถ้า transaction นั้น commit หรือ abort

Deadlock Prevention protocols

- วิธีการที่ใช้ในการป้องกัน deadlock

wait-die schemes	
wound-wait schemes	
Timeout-Base schemes	คอยบันทึกเวลาที่ transaction รอ ถ้า transaction ไหนรอนาน แสดงว่าเกิด deadlock ก็ kill ไป

Multiple Granularity

- แบ่งข้อมูลออกเป็นลำดับชั้น จากใหญ่ไปเล็ก (DB -> Area -> File -> record)
- สามารถเขียนเป็น tree แสดงลำดับของข้อมูลได้
- เมื่อ lock item ชั้นบน item ชั้นล่างทั้งหมดก็จะถูก lock ไปด้วย

    ex. lock $File_a$ => record ทั้งหมดใน $File_a$ ก็จะถูก lock ไปด้วย

    fine granularity ยิ่ง lock ระดับล่าง ยิ่งทำให้ high concurrency

    coarse granularity ยิ่ง lock ระดับบน ยิ่งทำให้ low concurrency

* Lock-Base Protocols เป็น protocols ที่เป็นหน้าที่ของ programmer

Timestamp-Base Protocols

- เป็น protocols ที่จะมอบ **timestamp** ให้กับ transaction แต่ล่ะตัวที่เข้ามาในระบบ
- timestamp เหมือนบัตรคิว ที่จะมีไว้เพื่อลำดับของ transaction ที่เข้ามา
- แทน timestamp ของ transaction i ด้วย **$TS(T_i)$**
- ระบบจะเก็บค่า 2 ค่า สำหรับแต่ล่ะ data item Q ดังนี้

    W-timestamp(Q) เพื่อเก็บ TS ที่สูงที่สุด ที่สามารถ write(Q) ได้สำเร็จ

    R-timestamp(Q) เพื่อเก็บ TS ที่สูงที่สุด ที่สามารถ read(Q) ได้สำเร็จ

- เมื่อ T_i ทำการ read(Q)

    if $TS(T_i) < W\text{-timestamp}(Q)$ then ปฏิเสธ read(Q) , roll back T_i

        - แสดงว่ามี transaction ที่มาทีหลัง T_i มาตัดหน้าเขียนก่อนที่ T_i จะอ่าน

    else

        execute read(Q) , $R\text{-timestamp}(Q) = \max(R\text{-timestamp}(Q), TS(T_i))$

- เมื่อ T_i ทำการ write(Q)

    if $TS(T_i) < R\text{-timestamp}(Q)$ then ปฏิเสธ write(Q), roll back T_i

        - แสดงว่ามี transaction ที่มาทีหลัง T_i มาตัดหน้าอ่านไปก่อนที่ T_i จะเขียนให้

    if $TS(T_i) < W\text{-timestamp}(Q)$ then ปฏิเสธ write(Q), roll back T_i

        - แสดงว่ามี transaction ที่มาทีหลัง T_i มาตัดหน้าเขียนก่อนที่ T_i จะเขียนให้

    else

        execute write(Q) , $W\text{-timestamp}(Q) = TS(T_i)$

* timestamp-ordering protocols รับประกัน serializability
* timestamp protocols ปราศจาก deadlock
* แต่ว่ายังไม่เป็น cascade-free และ recoverable
* timestamp-base protocols เป็น protocols ที่ระบบ DBMS เป็นผู้จัดการให้เอง

Recoverability and Cascade Freedom

- DBMS มีวิธีในการทำให้เป็น cascade freedom และ recoverable

solution 1 : คำสั่ง write จะถูกเขียนข้อมูลลง Disk จริงๆเมื่อจบ transaction เท่านั้น

solution 2 : lock ข้อมูล และจะรอให้ commit ก่อนที่จะ read

solution 3 : ใช้ commit dependencies (คือเหี้ยไรว่ะ)

Thomas' Write Rule

- ลุงโธมัส ที่เป็นใครก็ไม่รู้ ได้กล่าวไว้ว่า ไอ่เหี้ยตอนที่มึงจะ write(Q) อ่ะ แล้ว TS(T_i) < W-timestamp(Q) มึงก็แค่ปฏิเสธ write(Q) ตัวนี้ไปดิ่ ไม่ต้อง roll back T_i
- ปรากฏว่าทำตามที่ลุงแกพูดมาแล้วได้ concurrency ที่ดีขึ้นว่ะ ปรบมือให้ลุง
- ทำให้เป็น view-serializability ที่ไม่เป็น conflict-serializability ด้วยนะเออ นี้คือความเฟี้ยวของลุง

Chapter 16 : Recovery System

Failure Classification

- **Transaction failure**
  - **Logical errors** transaction ไม่สามารถทำงานจบได้ เพราะเกิด error จากภายในโค้ด
  - **System errors** database system สั่งหยุด transaction เนื่องจากเกิดปัญหา ex. deadlock
- **System crash**
  - ไฟดับ / โปรแกรมค้างทำระบบล่ม
  - system crash อาจก่อให้เกิดการเก็บข้อมูลไม่ถูกต้องได้ database system จะต้องมีวิธีการเช็คเพื่อป้องกันการผิดพลาดนี้
- **Disk failure**
  - หัวอ่านเขียนเสีย / อาจทำให้ข้อมูลทั้งหมดใน disk เสียหายได้

Storage Structure

- **Volatile storage**
  - เกิด system crash ได้
- **Nonvolatile storage**
  - ไม่เกิด system crash
  - แต่ถ้าล่มข้อมูลก้อาจหายได้
- **Stable storage**
  - storage ในตำนาน ที่กล่าวกันว่าจะไม่มีวัน failures อะไรทั้งนั้น (แปลได้เว่อดีจัง)
  - สามารถทำได้โดยการใช้ nonvolatile เก็บ copy ไว้หลยๆชุด

Data Access

- **Block** หน่วยที่ใช้ในการเก็บและขนย้ายข้อมูล
  - **Physical blocks** block ที่อยู่ใน disk
  - **Buffer blocks** block ชั่วคราว ที่อยู่ใน main memory
  - ข้อมูลใน block จะเคลื่อนย้ายไปมาระหว่าง disk และ main memory ด้วย 2 คำสั่งนี้
    - **input(B)** ย้าย block B จาก physical block ไปยัง buffer block
    - **output(B)** ย้าย block B จาก buffer block ไปยัง physical block, เขียนทับที่เดิมด้วย

* สมมติให้ data item แต่ล่ะชิ้น เก็บอยู่ใน block เดียว

- Work area

- แต่ล่ะ transaction ที่ทำงานอยู่ จะมีพื้นที่ทำงานของตัวเองใน main memory
- พื้นที่นี้จะเก็บ data item ซึ่ง copy ออกมาจาก buffer block
    * T_i จะมี copy ของ data item X เรียกว่า x_i
- data item จะเคลื่อนย้ายจาก buffer block ไปยัง work area ด้วย 2 คำสั่งนี้
    - **read(X)** copy ค่าของ X ไปยัง x_i
    - **write(X)** copy ค่าของ x_i ไปให้ X

* โปรแกรมเมอร์เขียนคำสั่งได้แค่ read / write ส่วน input / output OS จะเป็นคนจัดการเอง

* output ไม่จำเป็นต้องทำต่อจาก write ทันที จะทำเมื่อ OS คิดว่าสมควรจะทำ

Log-Based Recovery

- เป็นระบบที่สร้างขึ้นมาเพื่อที่จะทำให้ system สามารถทำ recovery ได้
- ในระบบจะมี **log** ซึ่งจะเก็บไว้ใน **stable storage**
- ในตัว log นี้ จะเก็บ **log record** ซึ่งไว้คอยบอกกิจกรรมที่เกิดขึ้นกับ database
    - **<T_i, start>** T_i เริ่มทำงาน
    - **<T_i, X, V_1, V_2>** T_i ทำ write(X) , V_1 ค่าเก่า, V_2 ค่าใหม่
    - **<T_i, commit>** T_i ทำงานเสร็จแล้ว
- มีวิธีการใช้ log อยู่ 2 แบบ
    - **Immediate database modification**
        - สามารถ output block ก่อนที่ transaction จะ commit ได้
    - **Deferred database modification**
        - ต้องรอ transaction commit ก่อน ถึงจะได้ output block ได้

Undo and Redo Operations

- **Undo** <T_i, X, V_1, V_2> **X = V_1**
    - ทำเมื่อ transaction นั้นยังทำงานไม่สำเร็จ แต่เกิด failure ขึ้นก่อน
    - **undo(T_i)** ทำการ undo ทุกๆ record ที่ทำโดย T_i * ไล่ขึ้นไป
        - เมื่อ undo แล้วจะเพิ่ม **<T_i, X, V_1>** ลงใน log (เรียกว่า **compensation log record**)
        - เมื่อ undo เสร็จแล้วจะเพิ่ม **<T_i, abort>** ลงใน log
- **Redo** <T_i, X, V_1, V_2> **X = V_2**
    - ทำเมื่อ transaction นั้นทำงานสำเร็จไปแล้ว แต่ยังไม่ย้ายข้อมูลจาก buffer ไป disk
    - **redo(T_i)** ทำการ redo ทุกๆ record ที่ทำโดย T_i * ไล่ลงมา
        - ไม่ต้องเพิ่ม log ใดๆ

Undo and Redo on Recovering from Failure (แบบไม่มี checkpoint)

if log has **<T_i, start>**

if log has **<T_i, commit>** or **<T_i, abort>**

redo(T_i)

else **undo(T_i)**

undo ทำจากล่างขึ้นบน
redo ทำจากบนลงล่าง

Checkpoint

- การ undo/redo record ทั้ง log โคตรช้า เพราะว่า
  1. ยิ่งระบบทำงานมานั้น การทำ redo/undo ก็ยิ่งต้องใช้เวลามาก
  2. ไม่มีความจำเป็นที่จะต้อง redo ข้อมูลที่ output ไปยัง disk แล้ว
- ปรับปรุงวิธีการ recovery หน่อย โดยทำการสร้าง **checkpoint** ขึ้นเป็นระยะๆ
  1. output **log record** ทั้งหมดใน main memory ไปยัง **stable storage**
  2. output **buffer block** ที่ถูกแก้ไขทั้งหมด ไปยัง **disk**
  3. เพิ่ม log record **<check point L>** เมื่อ L คือ list ของ transaction ที่กำลังทำงานอยู่

  * การ update ข้อมูลทั้งหมดจะถูกหยุด เมื่อกำลังทำ checkpoint

Recovery from failure

- การทำ recovery จะมี 2 phase คือ redo phase และ undo phase

- Redo phase

1. หา **<checkpoint L>** ตัวสุดท้าย, **undo list = L**
2. จากตำแหน่ง checkpoint ไล่ลงมา

<T_i, X, V_1, V_2>	X = V_2 (redo)
<T_i, X, V>	X = V
<T_i, start>	เพิ่ม T_i ลงใน undo list
<T_i, commit> | <T_i, abort>	ลบ T_i ออกจาก undo list

- Undo phase

- ไล่จากตำแหน่งล่างสุด (จุดที่เกิด failure) ขึ้นไป

1. **<T_i, X, V_1, V_2>** และ T_i อยู่ใน undo list
   - X = V_1 (undo)
   - เพิ่ม **<T_i, X, V_1>** ใน log
2. **<T_i, start>** และ T_i อยู่ใน undo list
   - เพิ่ม **<T_i, abort>** ใน log
   - ลบ T_i ออกจาก undo list
3. ทำจนกว่า undo list จะว่าง

หลังจากทำ recovery เสร็จแล้ว ข้อมูลต้องเหมือนตำแหน่งที่ failure ไม่ใช่ตำแหน่งที่ checkpoint ตำแหน่ง checkpoint มีไว้เพื่อให้ทำ recovery น้อยลงเท่านั้น

Chapter 10 : Storage ~~and File Structure~~

Classification of Physical Storage Media

- มีวิธีจำแนก storage อยู่หลายแบบ แต่ที่เห็นอาจารย์เน้นๆน่าจะเป็นวิธีนี้

- Reliability ความคงทนของข้อมูล

- **volatile storage** เมื่อไม่มีไฟ ข้อมูลหาย
- **nonvolatile storage** เมื่อไม่มีไฟ ข้อมูลไม่หาย

Physical Storage Media

- Cache

- เร็วที่สุด, แพงที่สุด, volatile

- Main memory

- ~~ที่เก็บน้อยเกินกว่าจะเก็บ database ได้ทั้งหมด~~ ปัจจุบันเพียงพอแล้วที่จะเก็บ database
- volatile

- Flash memory

-

- Magnetic disk

- ใช้สำหรับเก็บข้อมูลระยะยาว โดยทั่วไปก็ใช้ในการเก็บ database
- เวลาใช้จะต้องย้ายข้อมูลไปยัง main memory
  - * แต่ความเร็วของ disk นั้นช้ากว่า main memory มาก
- **direct access** สามารถอ่านข้อมูลที่ตำแหน่งใดก็ได้
- ข้อมูลไม่หายแม้มีการ failure ยกเว้นแต่จะเกิด disk failure ซึ่งเป็นไปได้ยาก

- Tape storage

- ใช้สำหรับทำ back up (ทำไว้กัน disk failure)
- **sequential-access** ต้องอ่านข้อมูลตามลำดับ
  - ทำให้ช้ากว่า disk มาก

Storage Hierarchy

- primary storage

- volatile
- cache, main memory

- secondary storage

- เรียกว่า **on-line storage**
- flash memory, magnetic disk

- tertiary storage

- เรียกว่า **off-line storage**
- magnetic tape, optical storage

Magnetic disk structure

- Disk

- แต่ล่ะแผ่น disk เรียกว่า **platter**
- แต่ล่ะ platter จะแบ่งออกเป็นวงๆ เรียกว่า **track**
- แต่ล่ะ track จะแบ่งออกเป็นส่วนๆ เรียกว่า **sector**
- track ที่ตรงกันในแต่ล่ะ platter เรียกว่า **cylinder**

- Read-Write head

- จะลอยอยู่เหนือ disk ไม่ได้ได้ติดกับ disk

- Disk controller

- interface ระหว่าง disk และ computer system
- คำนวณ **checksum** ของแต่ล่ะ sector เพื่อยืนยันความถูกต้องของข้อมูล
- disk controller มีหลายมาตราฐาน เช่น
  - **ATA, SATA, SCSI, SAS**

Performance Measures of Disk (การวัดการทำงานของ disk)

- Access time เวลาที่ใช้ในการเข้าถึงข้อมูล

- **seek time** เวลาที่ใช้ในการเข้าถึง track ที่ถูกต้อง
- **rotational latency** เวลาที่ sector ที่ถูกต้องจะหมุนมาเจอ head

- Data-transfer rate อัตราการขนย้ายข้อมูล

- Mean time to failure ระยะเวลาใช้งานเฉลี่ยที่คาดว่า disk จะไม่เกิด failure

RAID

Chapter 12 : Query Processing

Measures of Query Cost (เวลาที่ใช้ในการ query)

- วิธีคำนวณอย่างง่ายคือการใช้แค่ จำนวน block ที่ย้าย และ จำนวนครั้งในการ seek จะได้ว่า

  - **cost = $(b * t_T) + (S * t_s)$**

    b = จำนวน block t_T = เวลาที่ใช้ย้าย 1 block

    S = จำนวนครั้งที่ seek t_s = เวลาที่ใช้ seek 1 ครั้ง

  * สูตรนี้ยังไม่รวมเวลาที่ใช้ในการเขียนข้อมูลและเวลาของ cpu

- มีหลายวิธีที่ใช้เพื่อช่วยลดเวลาของ IO เช่น **buffer**

  * ถ้าข้อมูลอยู่ใน buffer แล้ว ไม่ต้องทำ input

Selection Operation

- A1 linear search

- เมื่อหาแบบปกติ

  cost = b_r * block transfers + 1 seek

- เมื่อหาโดยใช้ key

  cost = $(b_r/2)$ * block transfers + 1 seek

* b_r = จำนวน block ที่เก็บ record ทั้งหมดของ relation r

* การจะใช้ binary search จะดูโง่มาก เพราะว่าในความเป็นจริงแล้วข้อมูลมันไม่ได้เรียง

- A2 primary index, equality on key

- เรียกดูแค่ 1 record, search key เป็น index

- cost = $(h_i + 1) * (t_T + t_S)$

* h_i = ความสูงของ B^+ tree

- A3 primary index, equality on nonkey

- เรียกดูหลาย record, search key เป็น index

- cost = $h_i * (t_T + t_S) + (t_s + t_T) * b$

* b = จำนวน block ที่เก็บ record ที่ตรงกับเงื่อนไข

- A4 secondary index, equality on nonkey

- เรียกดู 1 record, search key เป็น index และ candidate key

  - cost = $(h_i + 1) * (t_T + t_S)$

- เรียกดูหลาย record, search key เป็น index แต่ไม่เป็น candidate key

  - cost = $(h_i + n) * (t_T + t_S)$

  * n = จำนวน record ที่ตรงเงื่อนไข แต่อยู่คนละ block กัน

  * ช้าจนอาจเทียบเท่า linear search เลย