淺談 RDBMS 原理

Relational Database Management Systems

Yu-Shan Lin @ PythonHug

自我介紹

- 林玉山 (Yu-Shan Lin)
- 網路代號： SLMT
- 清華大學 Datalab 實驗室博士生
- 研究領域：雲端資料庫系統
- http://www.slmt.tw

學習動機

為什麼要學 RDBMS 裡面在做啥呢？

1. 可以幫助你管理 RDBMS

知己知彼，百戰百勝

怎麼說？

- 理解系統內部原理，可以讓管理員了解調整參數時會造成什麼影響

- Buffer Pool ?
- Join Buffer & Sort Buffer ?
- Locks ?
- Indices ?

2. 幫助提升 Coding 的能力

還有 trace code 的能力！

為什麼可以提昇 Coding 能力？

- Database management system (DBMS) 是一種很複雜又高度優化的系統
- 學習這類系統可以幫助理解
  - 怎麼讀懂這種大型系統的程式碼（像是 OS）
  - 改這類系統可能要考慮什麼
  - 優化技巧

“If you are good enough to write code for a DBMS, then you can write code on almost anything else.”

- Andy Pavlo @ CMU 15-721

RDBMS 的架構

裡面長什麼樣子呢？

RDBMS 的架構

- RDBMS 一般以經典的 IBM System R 為基礎來建構
- 可以大略切成三塊
  - Query Engine
  - Storage Engine
  - Transaction Management

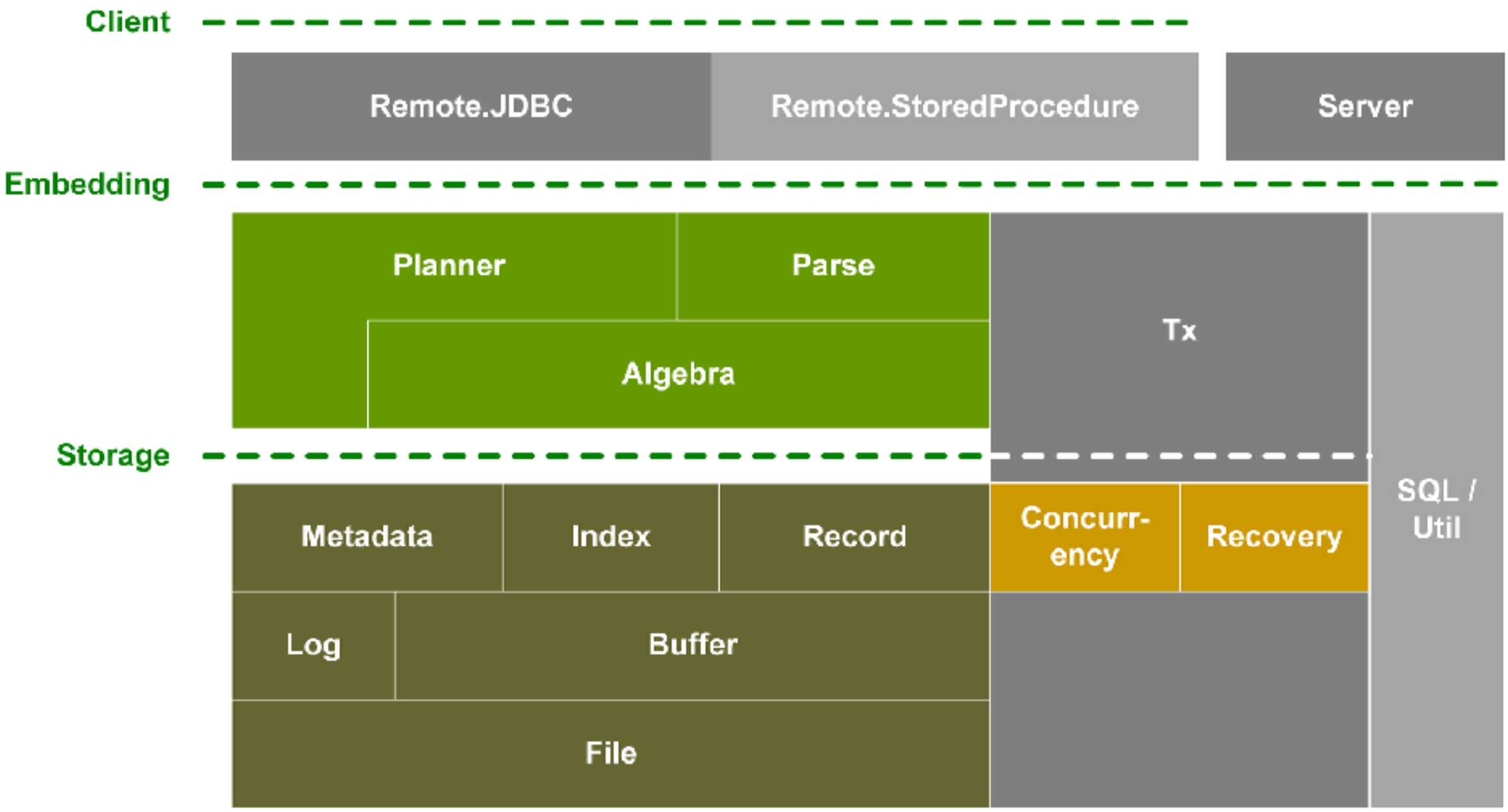
VanillaDB

http://www.vanilladb.org/

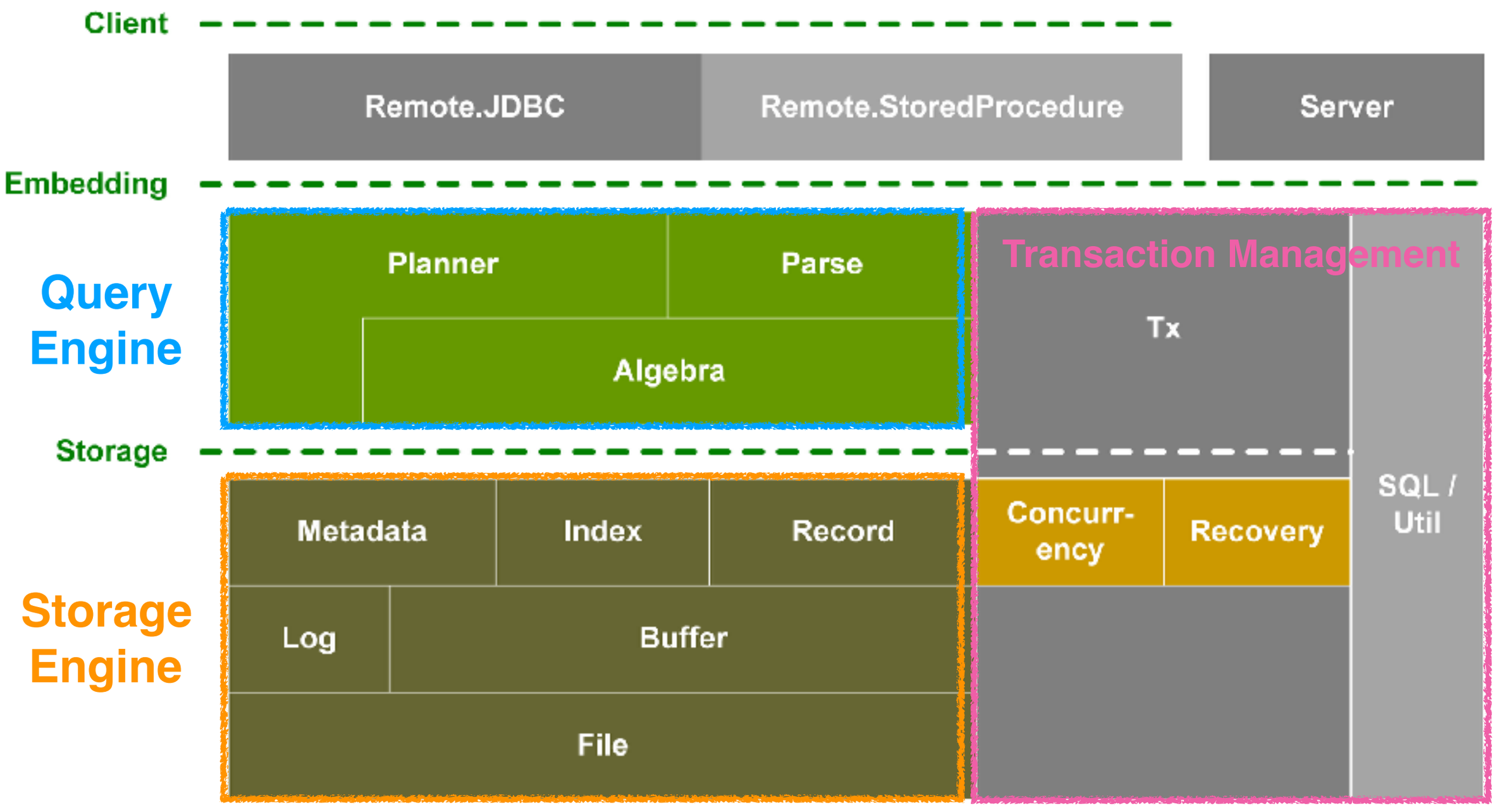
VanillaDB 子計畫

- VanillaCore
  - 執行於單台的 multi-threaded RDBMS
- VanillaBench
  - 測試 VanillaCore 的 Benchmarks
- VanillaComm
  - Group Communication
  - 為分散式系統鋪路

VanillaCore 的架構



VanillaCore 的架構



Query Engine

來看看 Query 的一天

範例：股票交易

id	name	balance
1	Red	3300
2	Blue	2200
3	Green	4500

account

buyer	stock_id	amount	time
1	103	50	7/19
1	297	300	8/1
1	31	230	8/5
2	45	40	8/7
3	24	100	9/2

stock_history

要求：找出 9/1 之後有買股票且至少有 3000 元的人

SQL 範例

id	name	balance
1	Red	3300
2	Blue	2200
3	Green	4500

account

buyer	stock_id	amount	time
1	103	50	7/19
1	297	300	8/1
1	31	230	8/5
2	45	40	8/7
3	24	100	9/2

stock_history

```
SELECT name FROM account, stock_history WHERE
    id = buyer AND balance > 3000 AND time >= 9/1;
```

Query 的一天

1. 詞性分析與斷詞 (Lexical Analysis & Tokenization)
2. 解析語意 (Parsing)
3. 計畫執行方式 (Planning)
4. 執行 (Executing)

詞性分析 (Lexical Analysis)

```
SELECT name FROM account, stock_history WHERE
    id = buyer AND balance > 3000 AND time >= 9/1;
```

詞性分析 (Lexical Analysis)

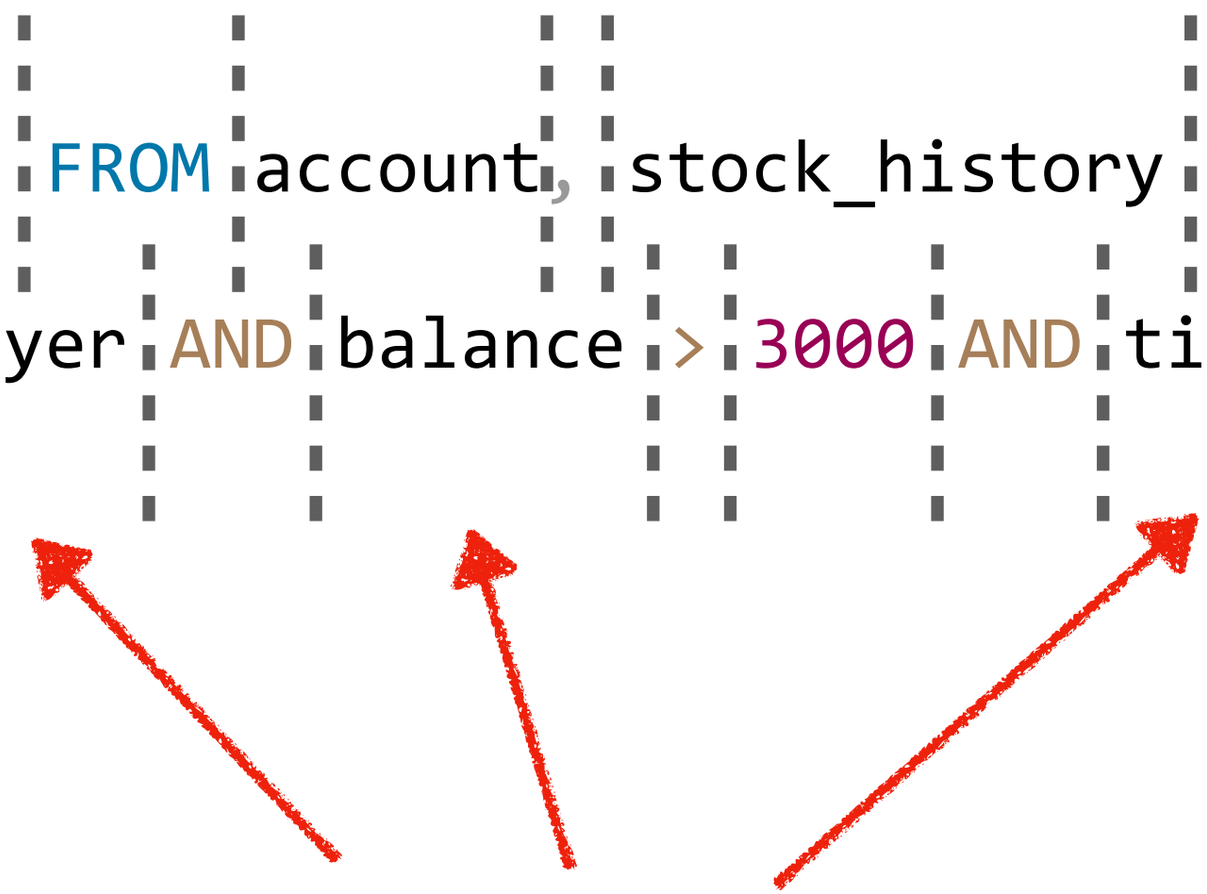
斷詞

```
SELECT name FROM account, stock_history WHERE
id = buyer AND balance > 3000 AND time >= 9/1;
```

詞性分析 (Lexical Analysis)

斷詞

```
SELECT name FROM account, stock_history WHERE
id = buyer AND balance > 3000 AND time >= 9/1;
```

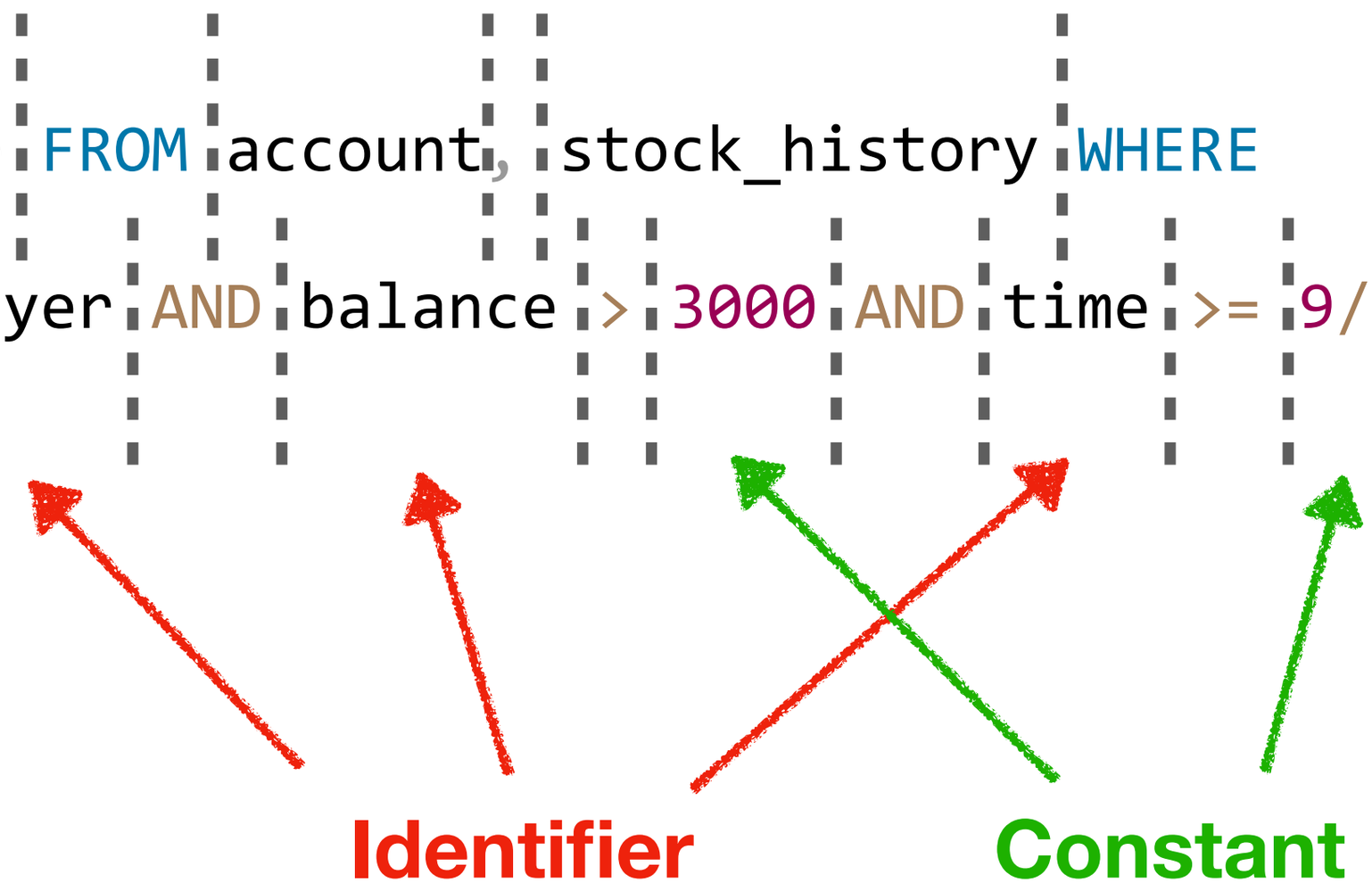


Identifier

詞性分析 (Lexical Analysis)

斷詞

```
SELECT name FROM account, stock_history WHERE
id = buyer AND balance > 3000 AND time >= 9/1;
```



詞性分析 (Lexical Analysis)

斷詞

```
SELECT name FROM account, stock_history WHERE
id = buyer AND balance > 3000 AND time >= 9/1;
```

Keywords **Identifier** **Constant**

語意解析

依照預定的文法來解析語意

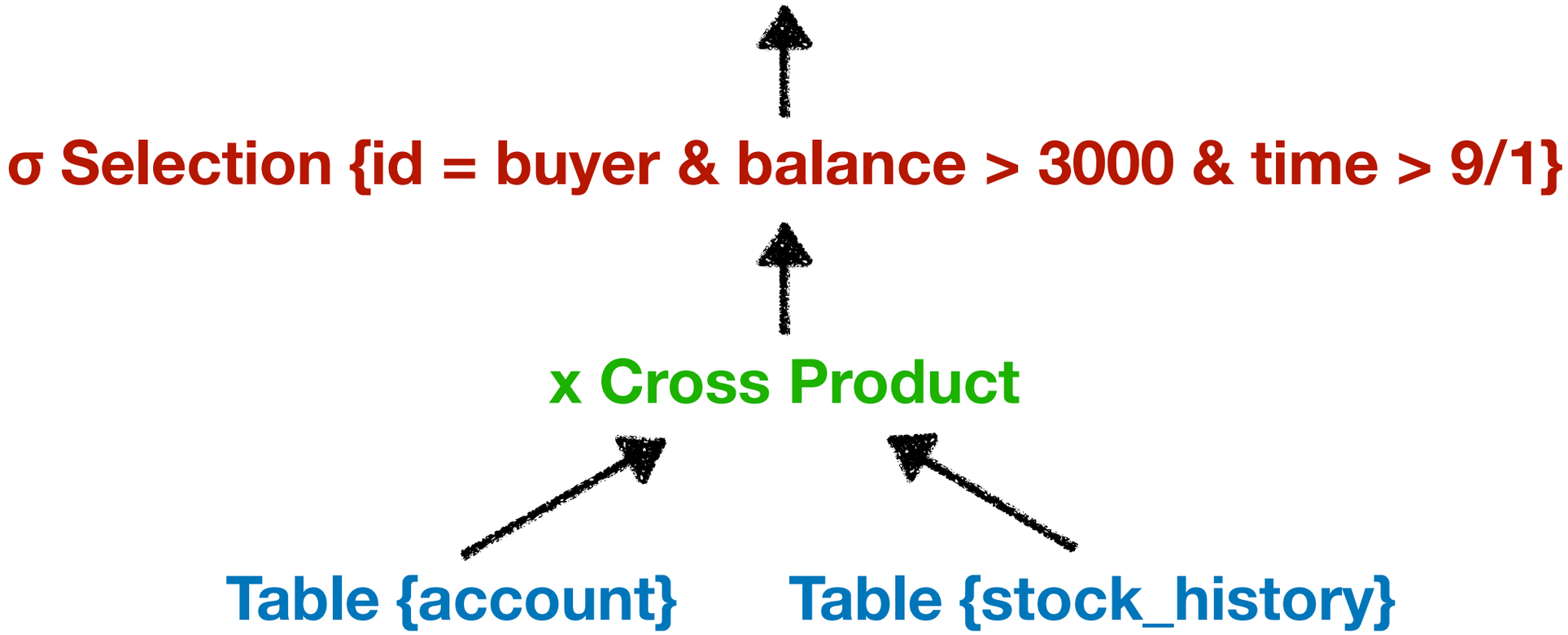
```
<Query>     := SELECT <ProjectSet> FROM <TableSet>
               [ WHERE <Predicate> ] [ GROUP BY <IdSet> ]
               [ ORDER BY <SortList> [ DESC | ASC ] ]
<IdSet>        := <Field> [ , <IdSet> ]
<TableSet>     := IdTok [ , <TableSet> ]
<AggFn>        := AVG(<Field>) | COUNT(<Field>) |
                  COUNT(DISTINCT <Field>) | MAX(<Field>) |
                  MIN(<Field>) | SUM(Field>)
<ProjectSet>   := <Field> | <AggFn> [ , <ProjectSet>]
<SortList>     := <Field> | <AggFn> [ , <SortList>]
```

Plan Trees

```
SELECT name FROM account, stock_history WHERE
    id = buyer AND balance > 3000 AND time >= 9/1;
```

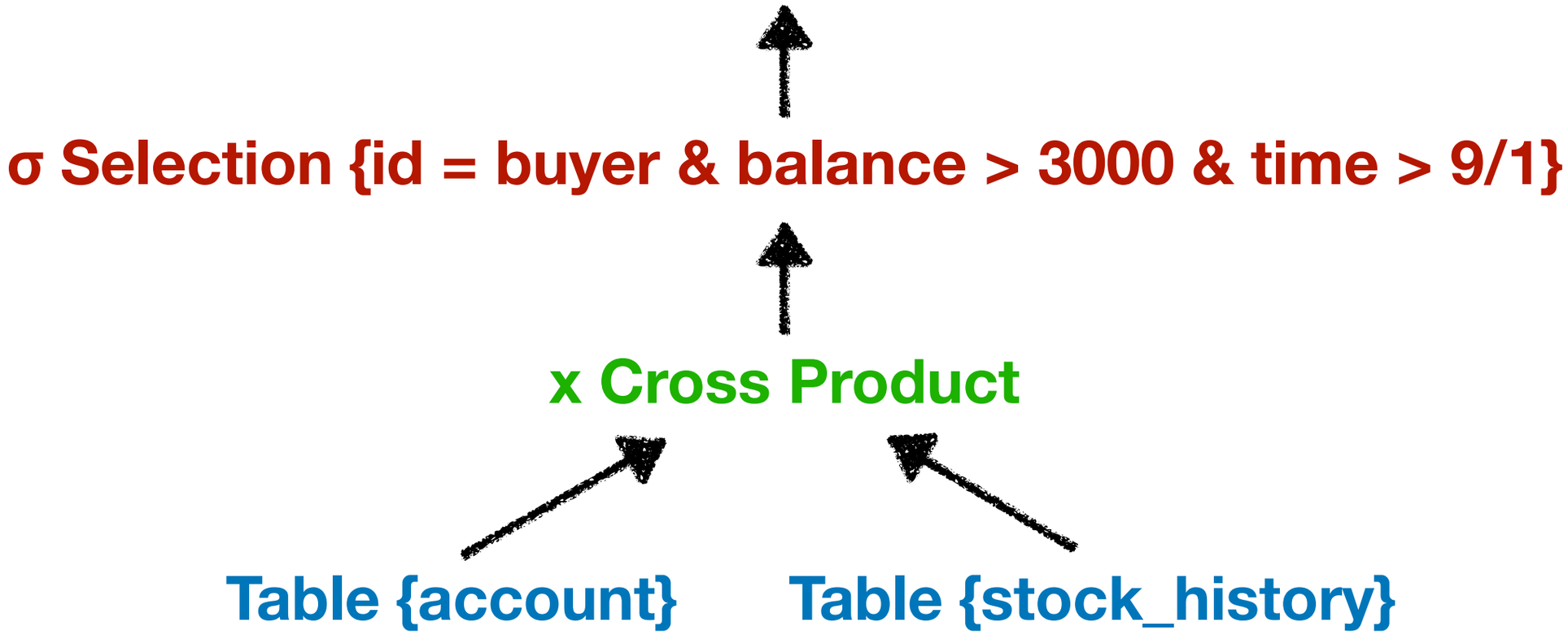
Plan Trees

```
SELECT name FROM account, stock_history WHERE
    id = buyer AND balance > 3000 AND time >= 9/1;
```



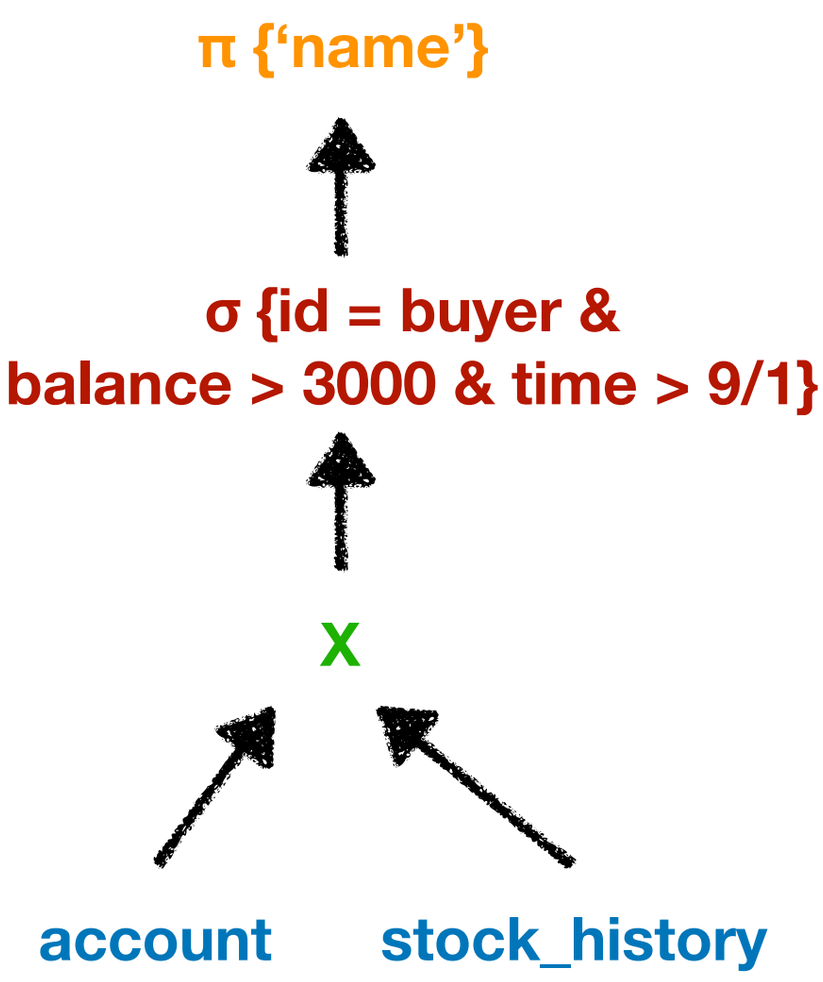
Plan Trees

```
SELECT name FROM account, stock_history WHERE
    id = buyer AND balance > 3000 AND time >= 9/1;
```



這些符號被稱為 Relational Algebra

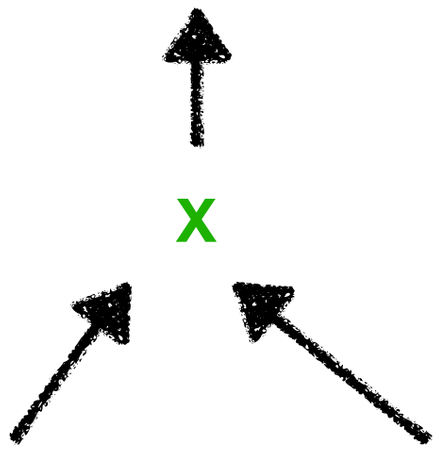
執行計畫


π {‘name’}
σ {id = buyer &
balance > 3000 & time > 9/1}
X
account
stock_history

執行計畫

π {‘name’}

σ {id = buyer &
balance > 3000 & time > 9/1}



id	name	balance
1	Red	3300
2	Blue	2200
3	Green	4500

buyer	stock_id	amount	time
1	103	50	7/19
1	297	300	8/1
1	31	230	8/5
2	45	40	8/7
3	24	100	9/2

執行計畫

π {‘name’}

↑

σ {id = buyer &
balance > 3000 & time > 9/1}

↑

X

account　　stock_history

id	name	balance
1	Red	3300
2	Blue	2200
3	Green	4500

buyer	stock_id	amount	time
1	103	50	7/19
1	297	300	8/1
1	31	230	8/5
2	45	40	8/7
3	24	100	9/2

執行計畫

π {‘name’}

↑

σ {id = buyer &
balance > 3000 & time > 9/1}

↑

X

↙ ↘

account　　stock_history

id	name	balance
1	Red	3300
2	Blue	2200
3	Green	4500

buyer	stock_id	amount	time
1	103	50	7/19
1	297	300	8/1
1	31	230	8/5
2	45	40	8/7
3	24	100	9/2

執行計畫

id	name	balance	buyer	stock_id	amount	time
1	Red	3300	1	103	50	7/19
1	Red	3300	1	297	300	8/1
1	Red	3300	1	31	230	8/5
1	Red	3300	2	45	40	8/7
…	…	…	…	…	…	…

π {‘name’}

σ {id = buyer & balance > 3000 & time > 9/

X

account

stock_history

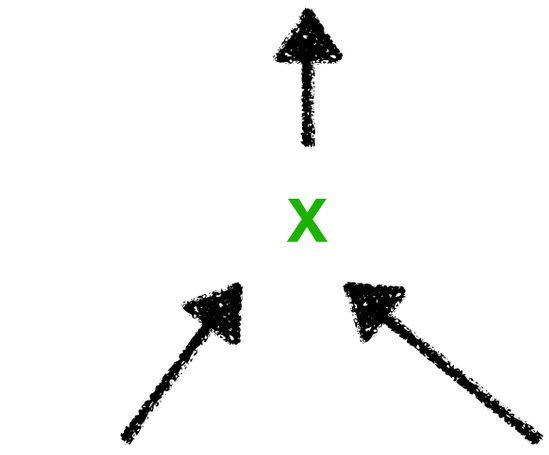
id	name	balance
1	Red	3300
2	Blue	2200
3	Green	4500

buyer	stock_id	amount	time
1	103	50	7/19
1	297	300	8/1
1	31	230	8/5
2	45	40	8/7
3	24	100	9/2

執行計畫

π {‘name’}

σ {id = buyer &
balance > 3000 & time > 9/1}



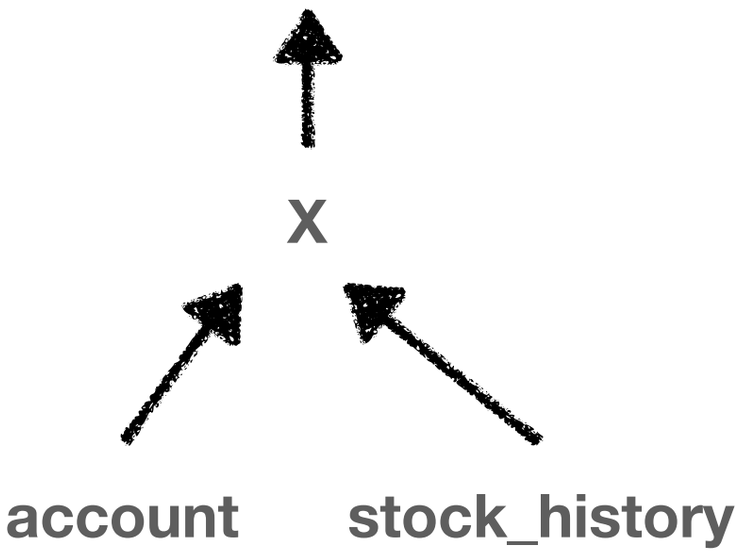
id	name	balance	buyer	stock_id	amount	time
3	Green	4000	3	24	100	9/2

id	name	balance	buyer	stock_id	amount	time
1	Red	3300	1	103	50	7/19
1	Red	3300	1	297	300	8/1
1	Red	3300	1	31	230	8/5
1	Red	3300	2	45	40	8/7
…	…	…	…	…	…	…

執行計畫

π {‘name’}

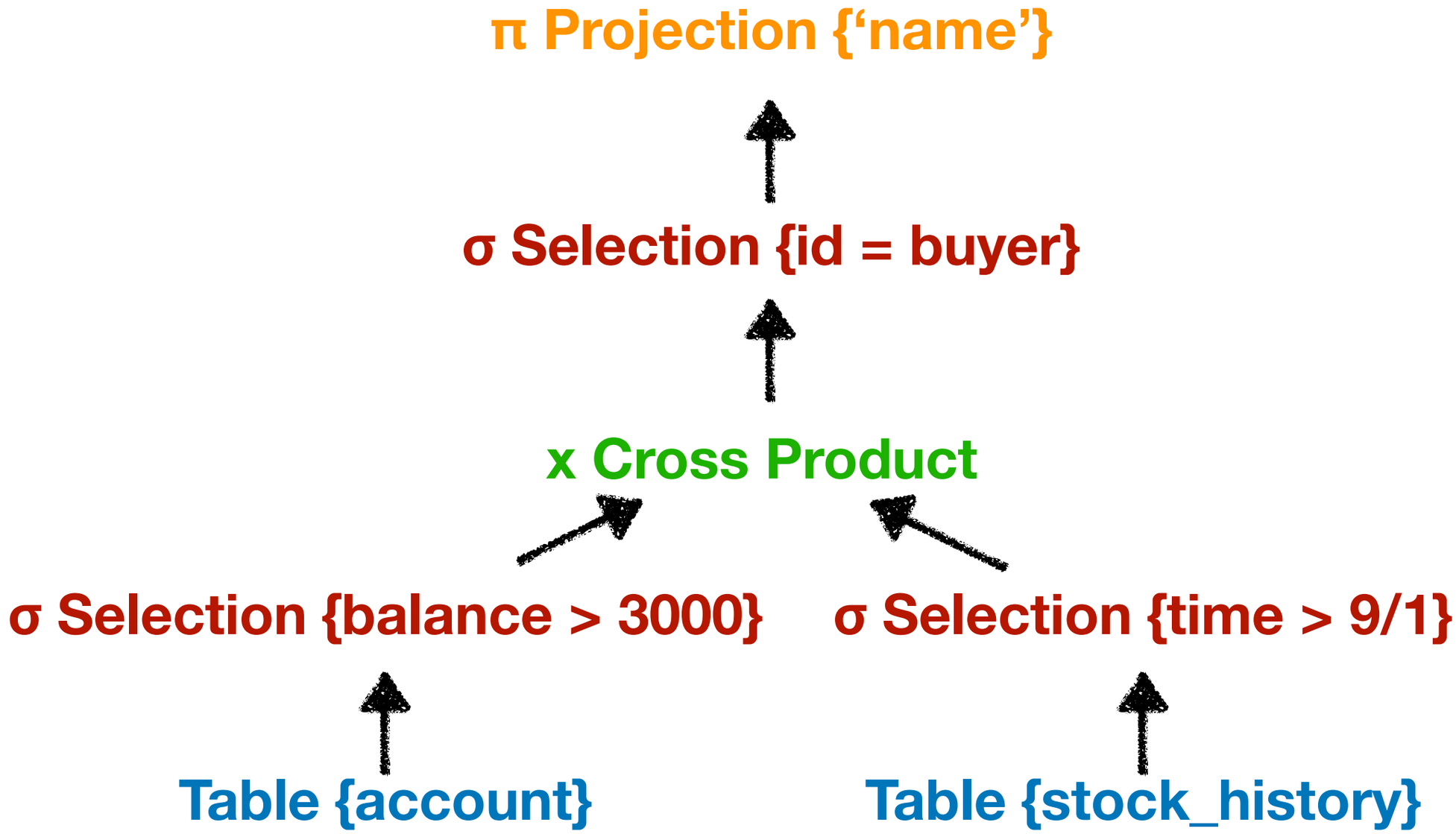
σ {id = buyer &
balance > 3000 & time > 9/1}



name
Green

id	name	balance	buyer	stock_id	amount	time
3	Green	4500	3	24	100	9/2

一個 Query 可能會有多種 Plan Tree



如何知道 Query 怎麼被執行？

如何知道 Query 怎麼被執行？

你可以叫你的 DBMS 解釋 (EXPLAIN) 給你聽

如何知道 Query 怎麼被執行？

你可以叫你的 DBMS 解釋 (EXPLAIN) 給你聽

```
SLMT=# EXPLAIN SELECT name FROM account, stock_history WHERE id = buyer
AND balance > 3000 AND time >= 901;
                                QUERY PLAN
-------------------------------------------------------------------------
 Hash Join  (cost=24.16..86.76 rows=835 width=58)
   Hash Cond: (stock_history.buyer = account.id)
   ->  Seq Scan on stock_history  (cost=0.00..32.12 rows=590 width=4)
         Filter: ("time" >= 901)
   ->  Hash  (cost=20.62..20.62 rows=283 width=62)
         ->  Seq Scan on account  (cost=0.00..20.62 rows=283 width=62)
               Filter: (balance > 3000)
(7 rows)
```

PostgreSQL

Planners

- 又被叫做 **Query Optimizer**
- Planner 的難點
  - 如何找出一個好的執行計畫？
  - 如何正確估計執行計畫的效率？
- 好的 Planner 帶你上天堂

幾種著名的找法

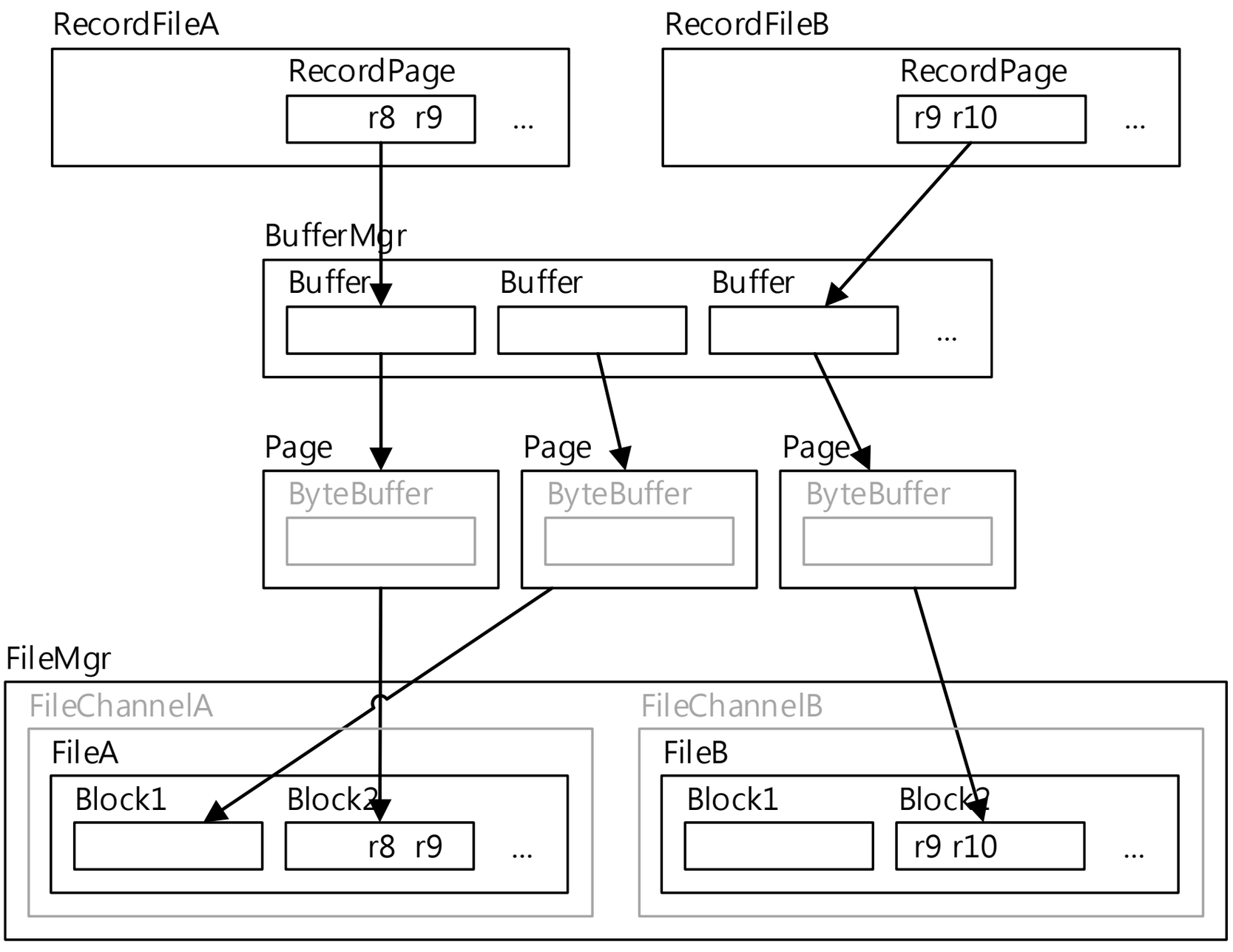
- Heuristic-based Optimization - INGRES, 1979
- Heuristic+cost-based Join Search - System R, 1979
- Simulated Annealing - Postgres, 1987
- Starburst Optimizer - DB2, 1988
- Volcano Optimizer - Academic, 1993
- Cascades Optimizer - SQL Server, 1995

光這個主題就夠發 40 年的 paper 惹！

Storage Engine

狀況相對簡單一些

Disk-based Storage



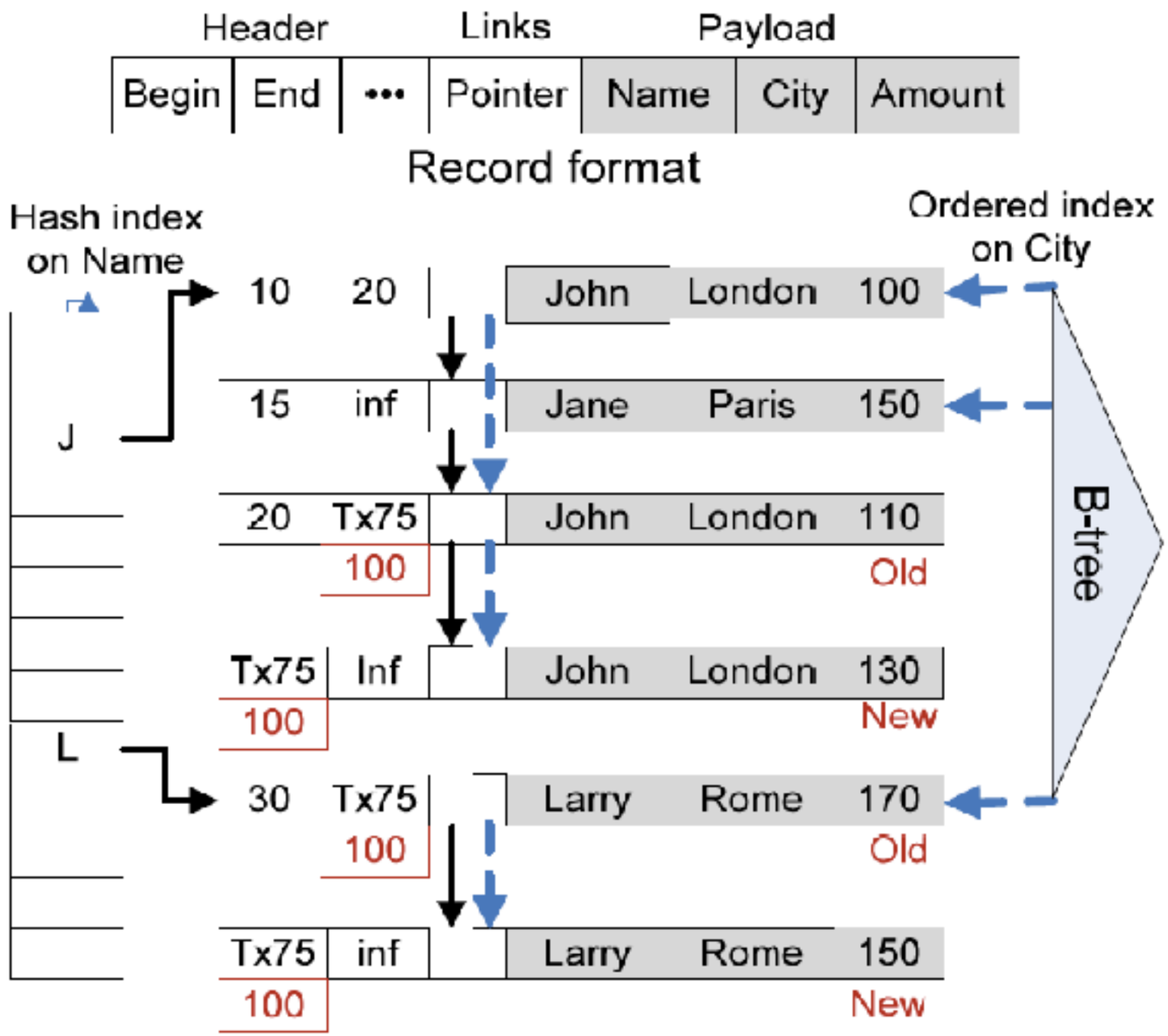
難點

- 硬碟是以 block 為單位存取資料
  - 所以需要小心安排資料
    - 根據管理方式可能會有不同程度的 fragmentation
- 可是放硬碟存取很慢
  - 所以需要 cache
  - 管理 cache 也有很多方法 (FIFO, LIFO, LRU…)

最近學界很流行
Main-Memory DBMS

- 假設資料都塞的進記憶體
- MMDBMS 的優勢
  - 不需要考慮 blocks 的設計 (Disk 是什麼？能吃嗎？)
  - 節省不需要的 mutex 與 latch
  - 大幅放寬 storage engine 的障礙

Hekaton, 2013



Peleton, 2016

	ID	IMAGE-ID	NAME	PRICE	DATA
Tile A-1	101	201	ITEM-101	10	DATA-101
	102	202	ITEM-102	20	DATA-102
	103	203	ITEM-103	30	DATA-103
Tile Group A					
Tile B-1	104	204	ITEM-104	40	DATA-104
	105	205	ITEM-105	50	DATA-105
	106	206	ITEM-106	60	DATA-106
Tile Group B					
Tile C-1	107	207	ITEM-107	70	DATA-107
	108	208	ITEM-108	80	DATA-108
	109	209	ITEM-109	90	DATA-109
	110	210	ITEM-110	100	DATA-110
Tile Group C					

Transaction Management

讓 RDBMS 強大的關鍵

Transactions

Transactions

Transactions

```
BEGIN TRANSACTION;

UPDATE account SET balance = balance - 100 WHERE name = "Red";

UPDATE account SET balance = balance + 100 WHERE name = “Blue";

COMMIT TRANSACTION;
```

Transactions

```
BEGIN TRANSACTION;

UPDATE account SET balance = balance - 100 WHERE name = "Red";

UPDATE account SET balance = balance + 100 WHERE name = “Blue";

COMMIT TRANSACTION;
```

ACID 

ACID

- 原子性 (Atomicity)
- 一致性 (Consistency)
- 獨立性 (Isolation)
- 持久性 (Durability)

A - Atomicity

- 全有或全無 (All or nothing)

A - Atomicity

- 全有或全無 (All or nothing)

```
BEGIN TRANSACTION;
UPDATE account ...
UPDATE account ...
COMMIT TRANSACTION;
```

None

```
BEGIN TRANSACTION;
UPDATE account ...
UPDATE account ...
COMMIT TRANSACTION;
```

Half

```
BEGIN TRANSACTION;
UPDATE account ...
UPDATE account ...
COMMIT TRANSACTION;
```

All

A - Atomicity

- 全有或全無 (All or nothing)

```
BEGIN TRANSACTION;
UPDATE account ...
UPDATE account ...
COMMIT TRANSACTION;
```

None

✔

```
BEGIN TRANSACTION;
UPDATE account ...
UPDATE account ...
COMMIT TRANSACTION;
```

Half

✘

```
BEGIN TRANSACTION;
UPDATE account ...
UPDATE account ...
COMMIT TRANSACTION;
```

All

✔

C - Consistency

- 資料要符合使用者設定的條件

C - Consistency

- 資料要符合使用者設定的條件

使用者設定： Sum(balance) = 10000

C - Consistency

- 資料要符合使用者設定的條件

使用者設定： Sum(balance) = 10000

```
BEGIN TRANSACTION;

UPDATE account ...

UPDATE account ...

COMMIT TRANSACTION;
```

In Progress

C - Consistency

- 資料要符合使用者設定的條件

使用者設定： Sum(balance) = 10000

```
BEGIN TRANSACTION;

UPDATE account ...

UPDATE account ...

COMMIT TRANSACTION;
```

In Progress

Sum(balance) = 9900

C - Consistency

- 資料要符合使用者設定的條件

使用者設定： Sum(balance) = 10000

```
BEGIN TRANSACTION;

UPDATE account ...

UPDATE account ...

COMMIT TRANSACTION;
```

In Progress

Sum(balance) = 9900

不一致 !!

C - Consistency

- 資料要符合使用者設定的條件

使用者設定： Sum(balance) = 10000

```
BEGIN TRANSACTION;

UPDATE account ...

UPDATE account ...

COMMIT TRANSACTION;
```

In Progress

Sum(balance) = 9900

不一致 !!

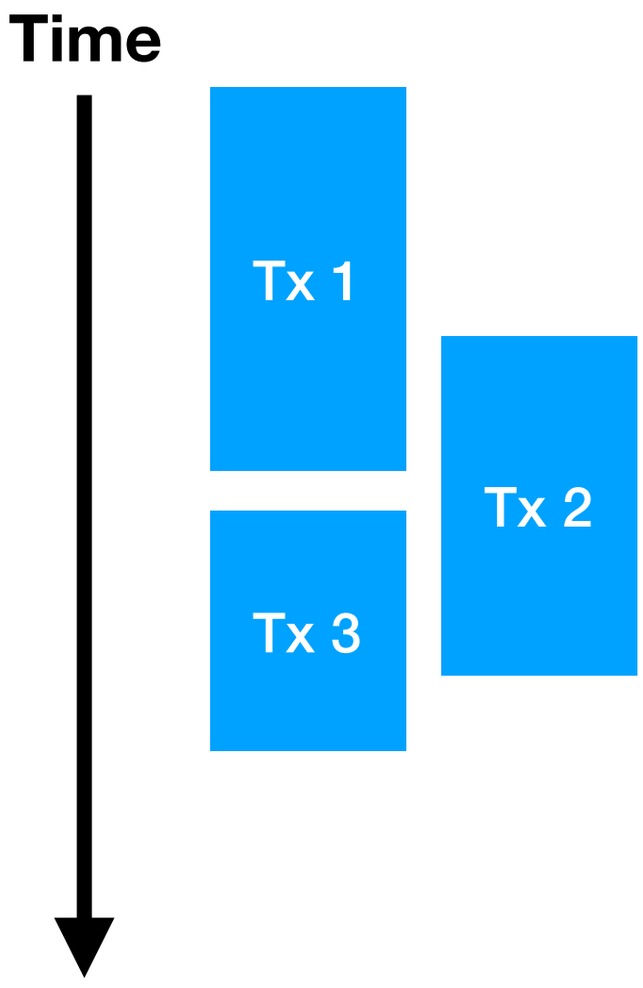
其他 transactions 不能看到這個

I - Isolation

Transaction 並行執行的結果必須符合某一種依序執行的結果

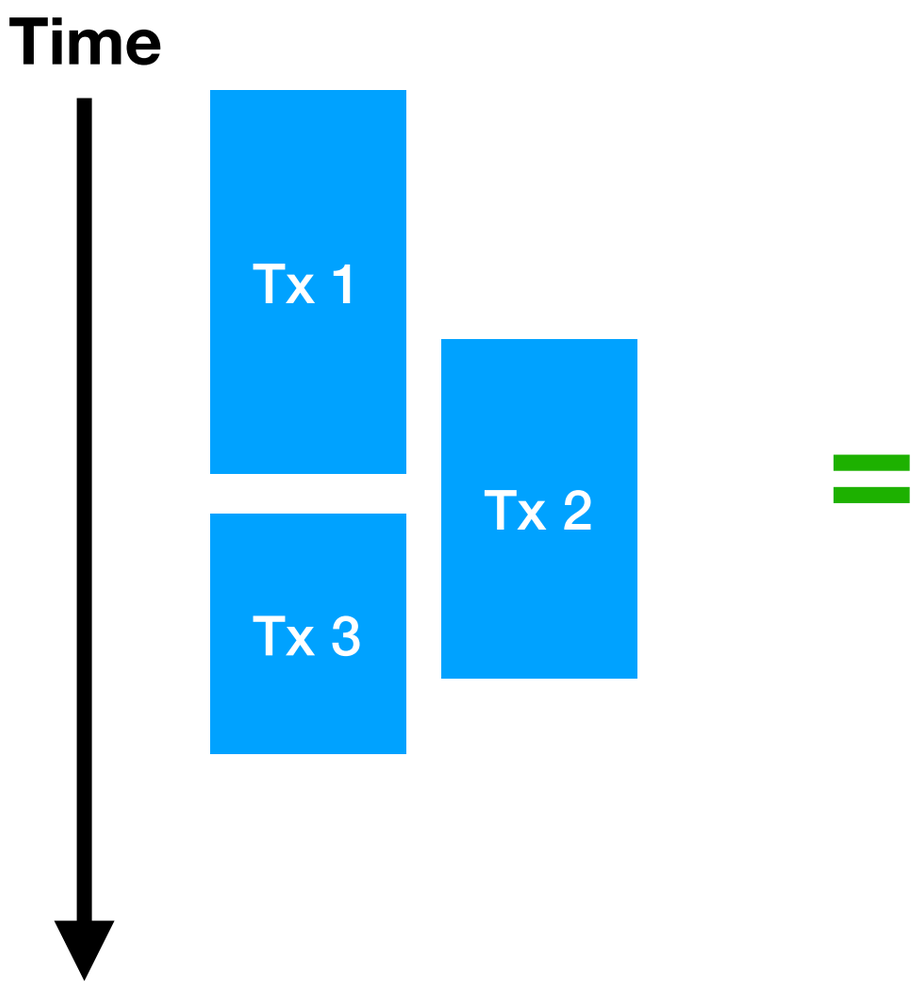
I - Isolation

Transaction 並行執行的結果必須符合某一種依序執行的結果



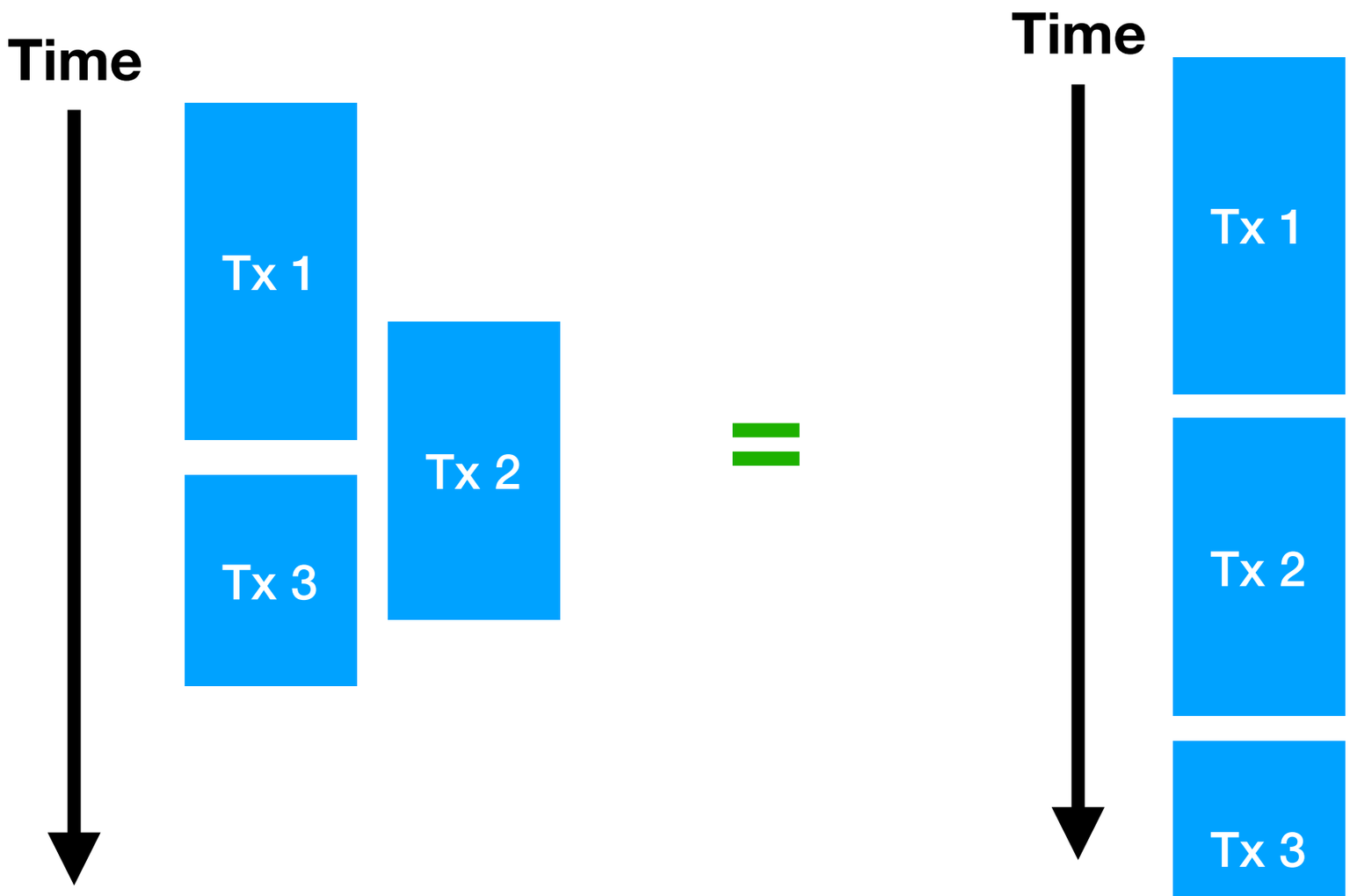
I - Isolation

Transaction 並行執行的結果必須符合某一種依序執行的結果



I - Isolation

Transaction 並行執行的結果必須符合某一種依序執行的結果



Isolation Levels

Isolation 也有分很多種

Level	Dirty Reads	Non-Repeatable Reads	Phantoms
Read Uncommitted	可能發生	可能發生	可能發生
Read Committed	安全	可能發生	可能發生
Repeatable Read	安全	安全	可能發生
Serializable	安全	安全	安全

Isolation Levels

Isolation 也有分很多種

Level	Dirty Reads	Non-Repeatable Reads	Phantoms
Read Uncommitted	可能發生	可能發生	可能發生
Read Committed	安全	可能發生	可能發生
Repeatable Read	安全	安全	可能發生
Serializable	安全	安全	安全

小知識：MySQL's InnoDB 預設是用 Repeatable Read

話說，我們真的需要這麼嚴謹嗎？

話說，我們真的需要這麼嚴謹嗎？

根據研究，其實大部分的人是用 READ COMMITTED！[1]

[1] “What Are We Doing With Our Lives? Nobody Cares About Our Research on Concurrency Control” in SIGMOD’17
[2] “ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications” in SIGMOD’17

話說，我們真的需要這麼嚴謹嗎？

根據研究，其實大部分的人是用 READ COMMITTED！[1]

不過放寬的 Isolation Levels 具有某些安全疑慮 [2]

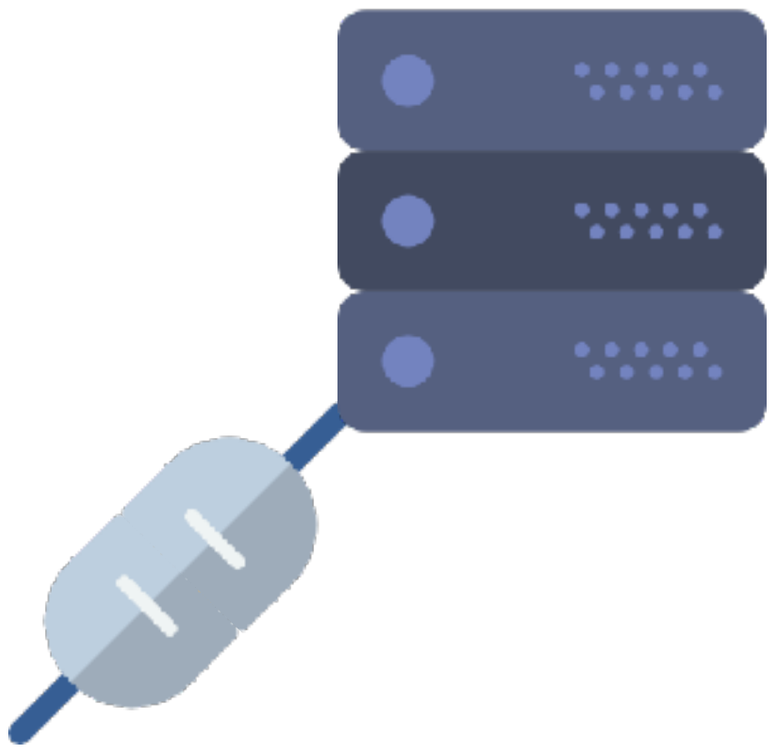
[1] “What Are We Doing With Our Lives? Nobody Cares About Our Research on Concurrency Control” in SIGMOD’17
[2] “ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications” in SIGMOD’17

D - Durability

- 已經 commit 的資料必須被保留下來

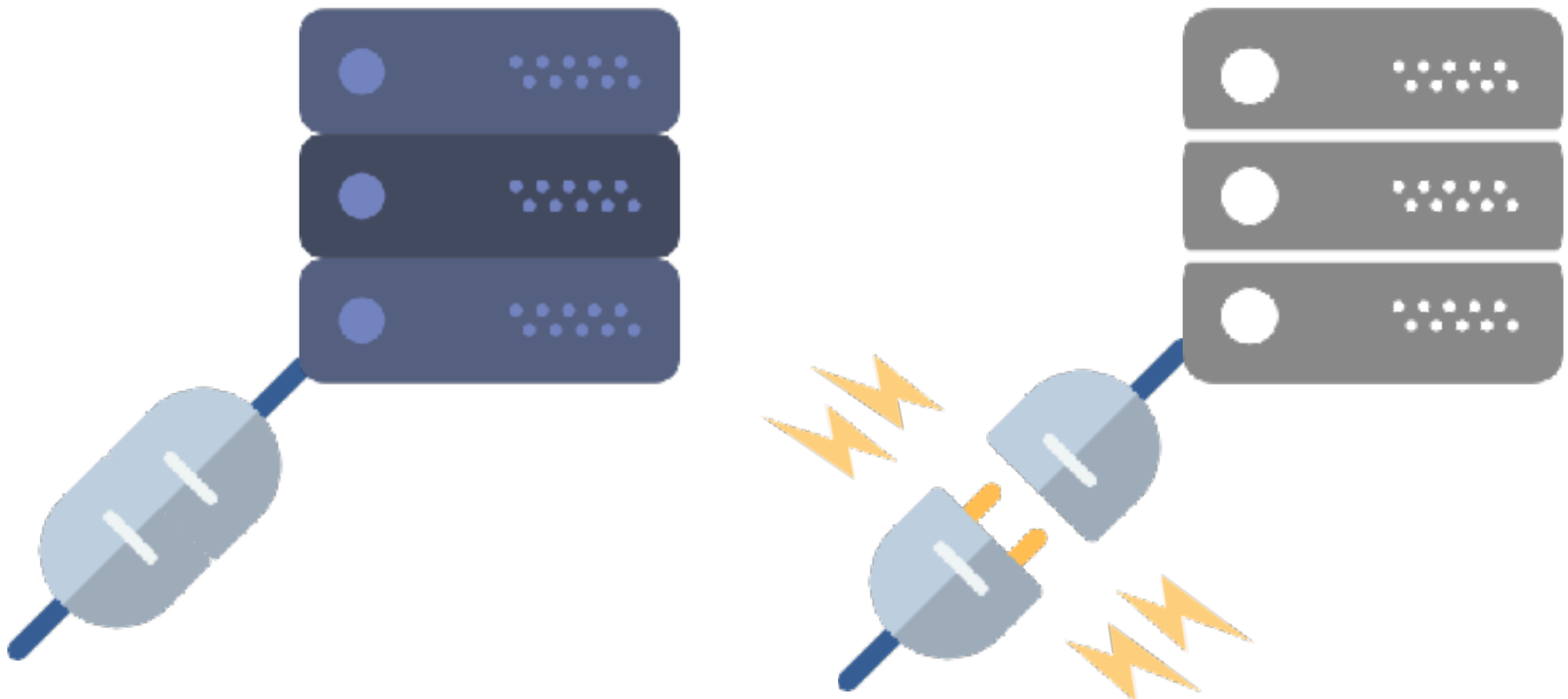
D - Durability

- 已經 commit 的資料必須被保留下來



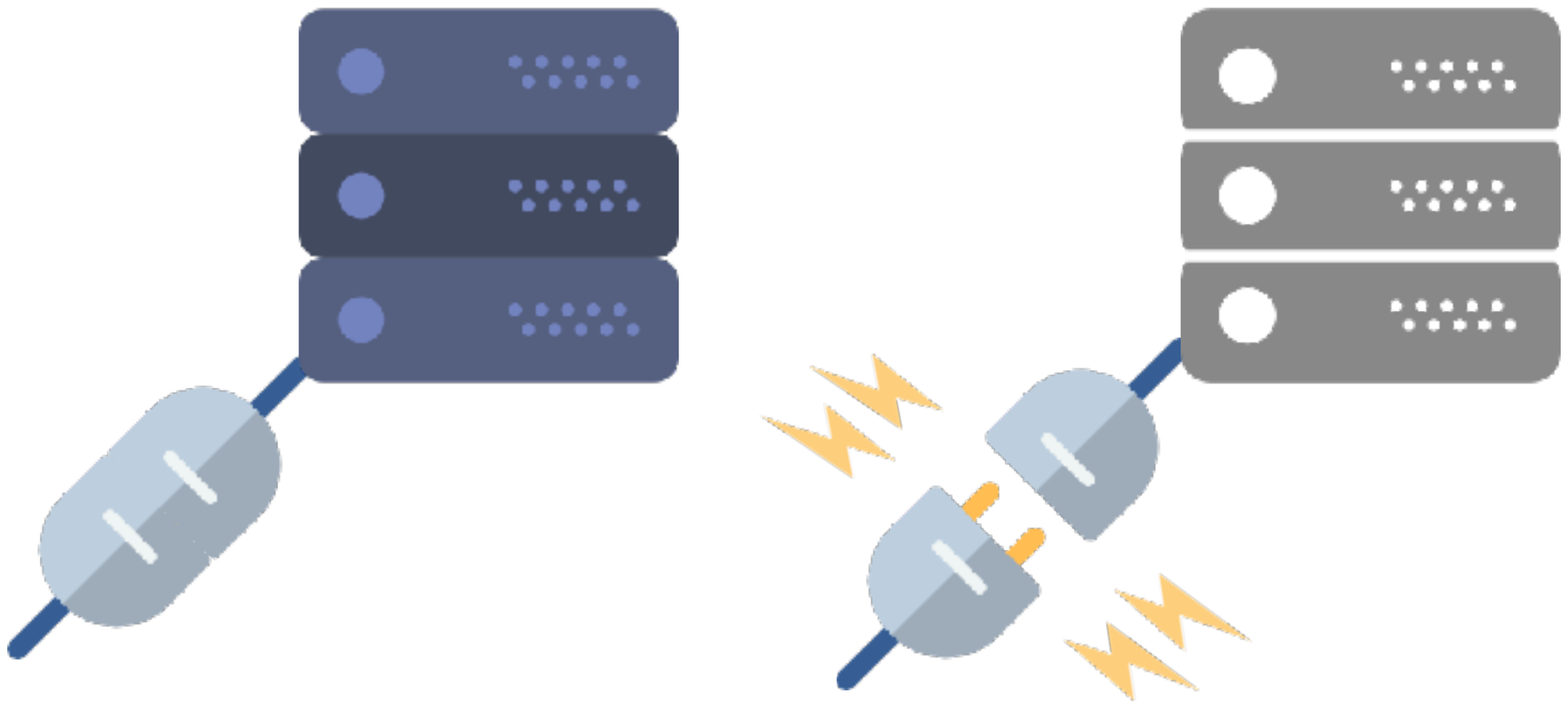
D - Durability

- 已經 commit 的資料必須被保留下來



D - Durability

- 已經 commit 的資料必須被保留下來



就算機器忽然斷電資料也要留下

來看看怎麼實作 ACID

Atomicity ?

Atomicity ?

我們需要 logs

Logging

- RDBMS 紀錄你的每一次更新

```
BEGIN TRANSACTION;
UPDATE account SET balance = balance - 100 WHERE name = "Red";
UPDATE account SET balance = balance + 100 WHERE name = “Blue";
COMMIT TRANSACTION;
```

SQLs

```
<Tx 1, Begin>
<Tx 1, Set Value, Record 1, Offset 30, Old 3300, New 3200>
<Tx 1, Set Value, Record 2, Offset 30, Old 2200, New 2300>
<Tx 1, Commit>
```

Logs

Undoing

- RDBMS 可以根據 logs 取消已經之前的操作

```
BEGIN TRANSACTION;
UPDATE account SET balance = balance - 100 WHERE name = "Red";
UPDATE account SET balance = balance + 100 WHERE name = “Blue";
COMMIT TRANSACTION;
```

```
<Tx 1, Begin>
<Tx 1, Set Value, Record 1, Offset 30, Old 3300, New 3200>
```

Undoing

- RDBMS 可以根據 logs 取消已經之前的操作

```
BEGIN TRANSACTION;
UPDATE account SET balance = balance - 100 WHERE name = "Red";
UPDATE account SET balance = balance + 100 WHERE name = “Blue";
COMMIT TRANSACTION;
```

```
<Tx 1, Begin>
<Tx 1, Set Value, Record 1, Offset 30, Old 3300, New 3200>
```

Undoing

- RDBMS 可以根據 logs 取消已經之前的操作

```
BEGIN TRANSACTION;
UPDATE account SET balance = balance - 100 WHERE name = "Red";
UPDATE account SET balance = balance + 100 WHERE name = “Blue";
COMMIT TRANSACTION;
```

```
<Tx 1, Begin>
<Tx 1, Set Value, Record 1, Offset 30, Old 3300, New 3200>
```

Undo: 將值改回 3300

一個小問題

我該先更新 record

還是

先寫 log 呢？

一個小問題

我該先更新 record

還是

先寫 log 呢？

Quick Answer: Write-Ahead Logging

有興趣可以自己查查～

Consistency ?

Consistency ?

Locks 可以幫上忙！

把正在存取的東西鎖起來

```
UPDATE account
SET balance = balance - 100
WHERE name = "Red";
```

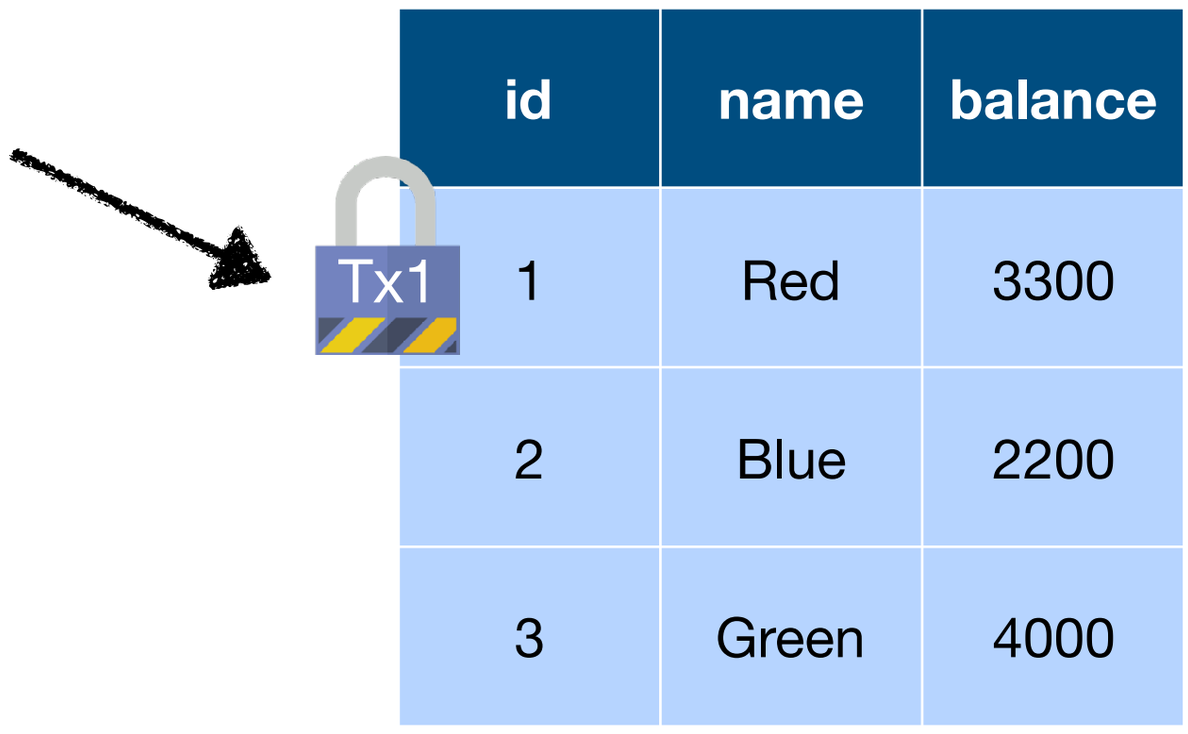
Transaction 1

id	name	balance
1	Red	3300
2	Blue	2200
3	Green	4000

把正在存取的東西鎖起來

```
UPDATE account
SET balance = balance - 100
WHERE name = "Red";
```

Transaction 1



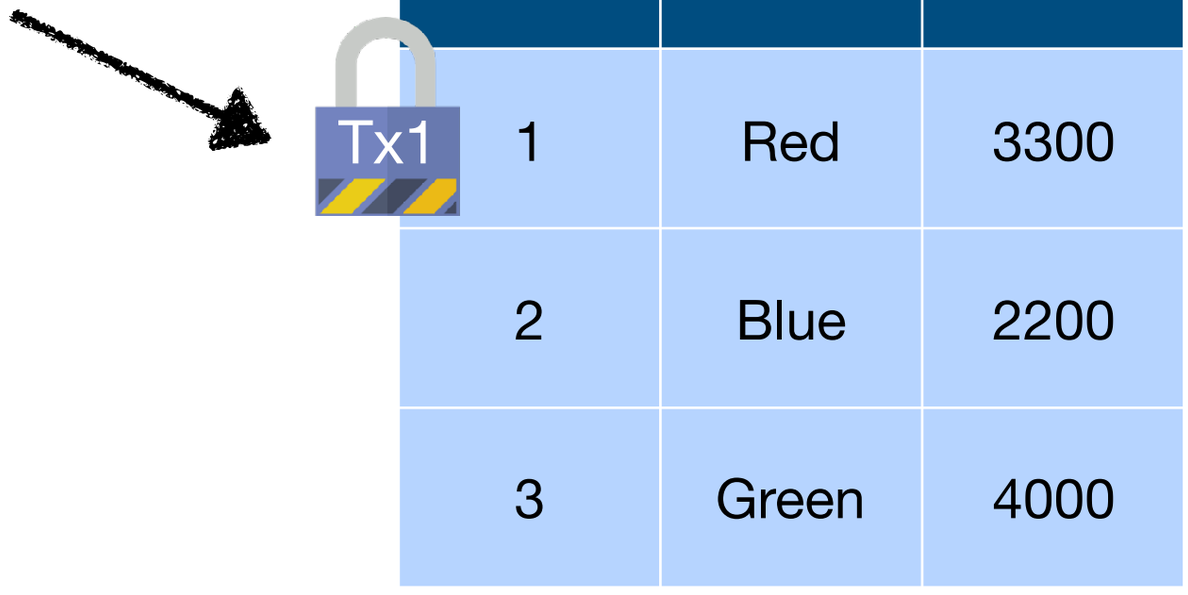
把正在存取的東西鎖起來

```sql
UPDATE account
SET balance = balance - 100
WHERE name = "Red";
```

Transaction 1

```sql
SELECT balance FROM account
WHERE name = "Red";
```

Transaction 2



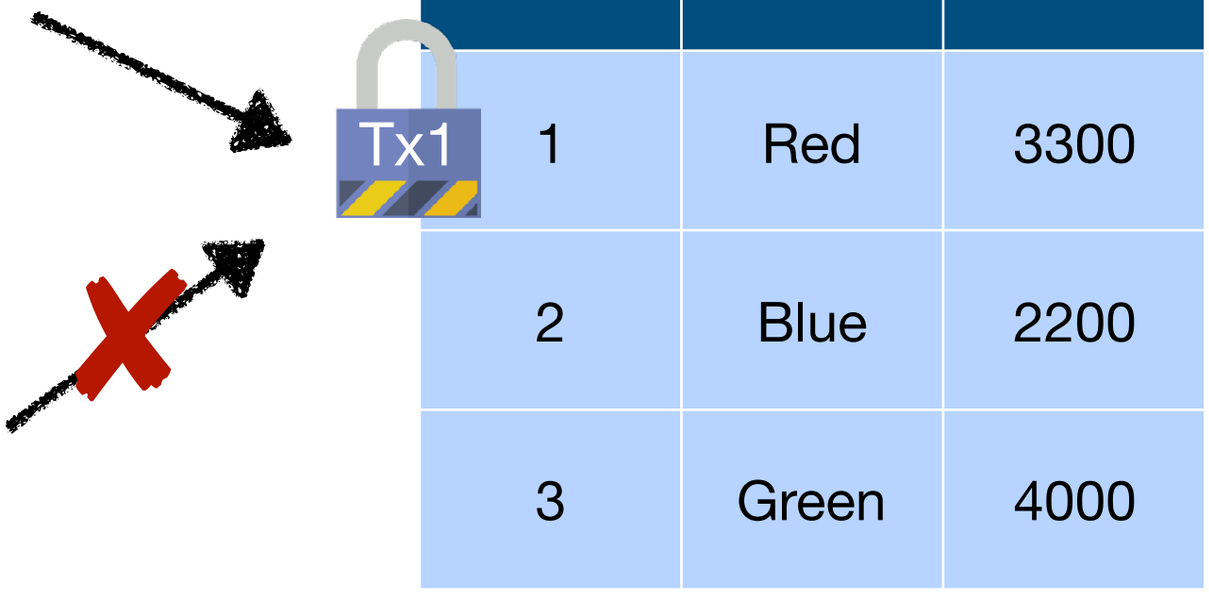
把正在存取的東西鎖起來

```sql
UPDATE account
SET balance = balance - 100
WHERE name = "Red";
```

Transaction 1

```sql
SELECT balance FROM account
WHERE name = "Red";
```

Transaction 2



有種東西叫做 Deadlock

Transaction 1

Transaction 2

id	name	balance
1	Red	3300
2	Blue	2200
3	Green	4000

Deadlock 了怎麼辦？

- 考驗 Operating Systems 學得怎樣的時候到啦！
- 常見做法
  - Deadlock-detection
  - Deadlock-avoidance
  - Deadlock-free locking
- 注意每種做法的 trade-off

Isolation ?

Isolation ?

又是 Locks！

Isolation Levels

- 這裡的難點是在於怎麼同時支援不同的 Isolation Level
  - 什麼時候取得 lock
  - 什麼時候釋放 lock
  - 哪種 lock 要拿？
- 有興趣可以自己查查看 :)

Durability ?

Durability ?

你需要的是...

Durability ?

你需要的是...

一顆硬碟！（或很多顆）

Durability ?

你需要的是...

一顆硬碟！（或很多顆）

記得存檔

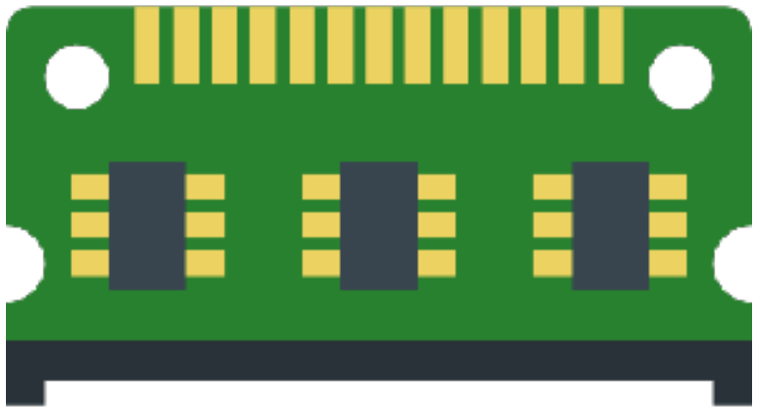
等等... 硬碟不是很慢嗎

等等... 硬碟不是很慢嗎

好吧... 你可能需要...

等等... 硬碟不是很慢嗎

好吧... 你可能需要...

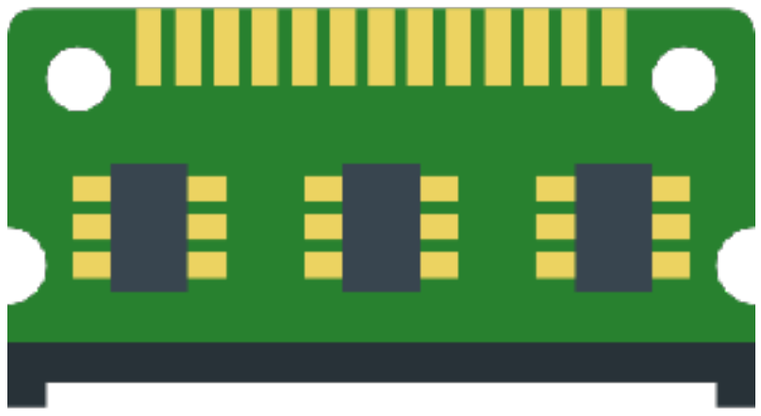


幾打的記憶體加速一下

嗯... 那電腦斷電時
存在記憶體的資料怎麼辦

嗯... 那電腦斷電時
存在記憶體的資料怎麼辦

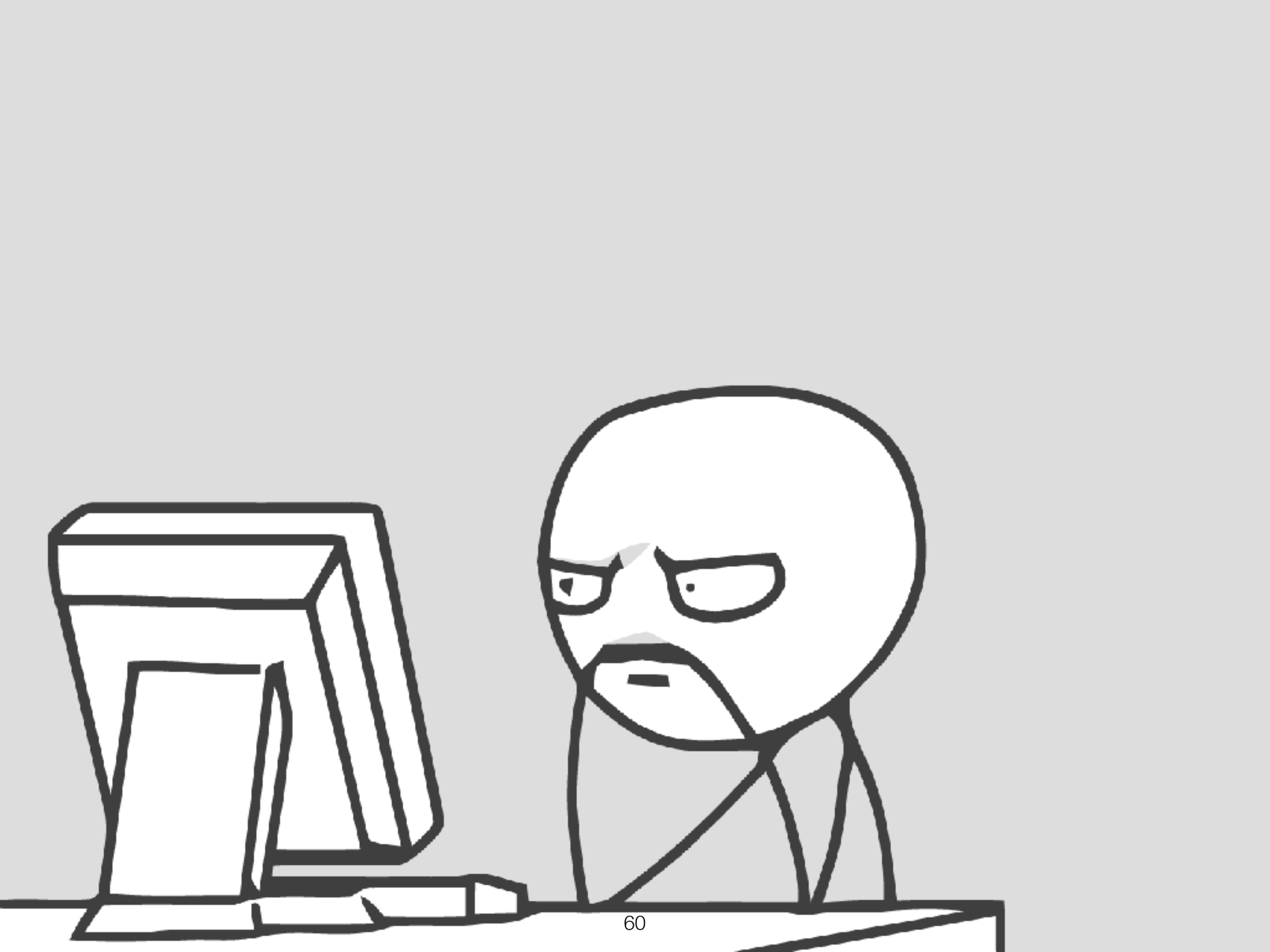
這個嗎... 你必須確保資料在 commit 之後都存進硬碟了

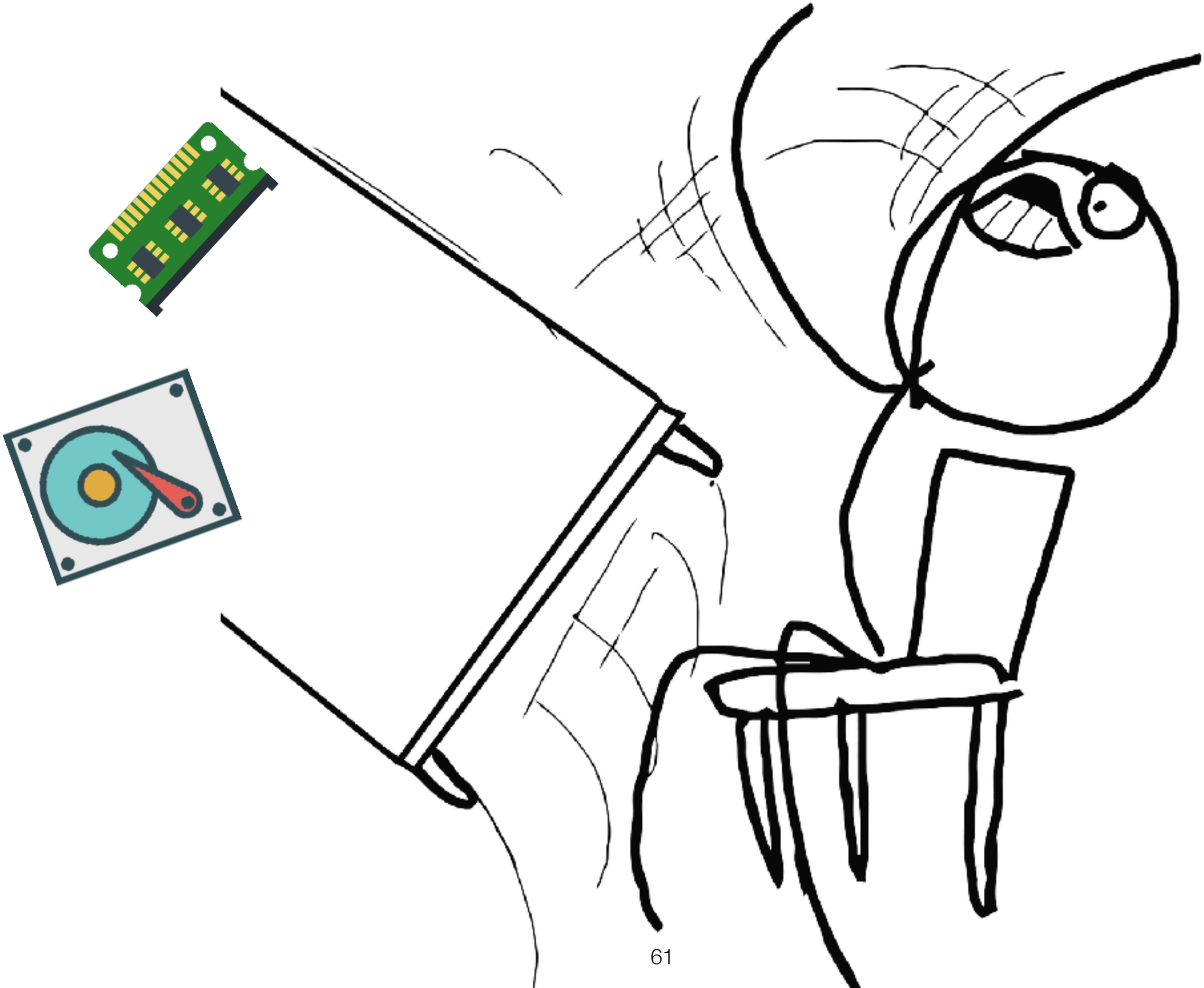


嗯... 那電腦斷電時
存在記憶體的資料怎麼辦

這個嗎... 你必須確保資料在 commit 之後都存進硬碟了

誒... 可是這樣 commit 不會很慢嗎？





其實只要存 logs 就好

DBMS 可以看你的 log 來復原資料

“If you are good enough to write code for a DBMS, then you can write code on almost anything else.”

- Andy Pavlo @ CMU 15-721

References

- 台灣資源
  - Introduction to Database System @ NTHU
- 國外資源
  - Introduction to Database Systems @ CMU
  - Advanced Database Systems @ CMU

Q&A

VanillaDB